



**School of Computer Science**

**CS 343**

**Concurrent and Parallel Programming**

**Course Notes\* Winter 2025**

<https://www.student.cs.uwaterloo.ca/~cs343>

[μC++ download](#) or [Github](#) (installation: `sudo sh u++-7.0.0.sh`)

January 7, 2025

### **Outline**

An introduction to concurrent programming, with an emphasis on language constructs. Major topics include: exceptions, coroutines, atomic operations, critical sections, mutual exclusion, semaphores, high-level concurrency, deadlock, interprocess communication, process structuring on shared memory architectures. Students learn how to structure, implement and debug complex control-flow.

---

\*Permission is granted to make copies for personal or educational use.



# Contents

<b>1</b>	<b>Advanced Control Flow (Review)</b>	<b>1</b>
1.1	Static multi-level exit . . . . .	2
1.2	Dynamic Memory Allocation . . . . .	4
<b>2</b>	<b>Nonlocal Transfer</b>	<b>7</b>
2.1	Traditional Approaches . . . . .	9
2.2	Dynamic Multi-level Exit . . . . .	11
2.3	Exception Handling . . . . .	13
2.4	Terminology . . . . .	14
2.5	Execution Environment . . . . .	15
2.6	Implementation . . . . .	16
2.7	Static/Dynamic Call/Return . . . . .	17
2.8	Static Propagation . . . . .	17
2.9	Dynamic Propagation . . . . .	18
	2.9.1 Termination . . . . .	19
	2.9.2 Resumption . . . . .	22
2.10	Exceptional Example . . . . .	22
<b>3</b>	<b>Coroutine</b>	<b>25</b>
3.1	Semi-Coroutine . . . . .	26
	3.1.1 Fibonacci Sequence . . . . .	26
	3.1.1.1 Direct . . . . .	26
	3.1.1.2 Routine . . . . .	27
	3.1.1.3 Class . . . . .	28
	3.1.1.4 Coroutine . . . . .	28
	3.1.2 Format Output . . . . .	30
	3.1.2.1 Direct . . . . .	30
	3.1.2.2 Routine . . . . .	31
	3.1.2.3 Class . . . . .	31
	3.1.2.4 Coroutine . . . . .	32
	3.1.3 Correct Coroutine Usage . . . . .	33
	3.1.4 Coroutine Construction . . . . .	34
3.2	$\mu$ C++ EHM . . . . .	35
3.3	Exception Type . . . . .	35
3.4	Inherited Members . . . . .	35

3.5	Raising	36
3.6	Handler	37
3.6.1	Termination	37
3.6.2	Resumption	37
3.6.3	Termination/Resumption	38
3.6.4	Object Binding	39
3.6.5	Bound Handlers	39
3.7	Nonlocal Exceptions	40
3.8	Memory Management	43
3.9	Semi-Coroutine Examples	43
3.9.1	Same Fringe	43
3.9.2	Device Driver	44
3.9.2.1	Direct	44
3.9.2.2	Coroutine	45
3.9.3	Producer-Consumer	47
3.10	Full Coroutines	49
3.10.1	Ping/Pong	51
3.10.2	Producer-Consumer	52
3.11	Coroutine Languages	54
3.11.1	Python 3.5	54
3.11.2	JavaScript	55
3.11.3	C++20 Coroutines	57
<b>4</b>	<b>More Exceptions</b>	<b>59</b>
4.1	Derived Exception-Type	59
4.2	Catch-Any	60
4.3	Exception Parameters	60
4.4	Exception List	61
4.5	Destructor	62
4.6	Multiple Exceptions	63
<b>5</b>	<b>Concurrency</b>	<b>65</b>
5.1	Why Write Concurrent Programs	65
5.2	Why Concurrency is Difficult	65
5.3	Concurrent Hardware	66
5.4	Execution States	68
5.5	Threading Model	69
5.6	Concurrent Systems	70
5.7	Speedup	71
5.8	Thread Creation	73
5.8.1	<b>COBEGIN/COEND</b>	73
5.8.2	<b>START/WAIT</b>	74
5.8.3	Actor	75
5.8.4	Thread Object	78
5.9	Termination Synchronization	79

5.10	Divide-and-Conquer	79
5.11	Exceptions	81
5.12	Synchronization and Communication During Execution	83
5.13	Communication	84
5.14	Critical Section	84
5.15	Static Variables	85
5.16	Mutual Exclusion Game	85
5.17	Self-Testing Critical Section	86
5.18	Software Solutions	87
5.18.1	Lock	87
5.18.2	Alternation	87
5.18.3	Declare Intent	88
5.18.4	Retract Intent	88
5.18.5	Prioritized Retract Intent	89
5.18.6	Dekker (modified retract intent)	89
5.18.7	Peterson (modified declare intent)	91
5.18.8	N-Thread Prioritized Entry	92
5.18.9	N-Thread Bakery (Tickets)	94
5.18.10	Tournament	95
5.18.11	Arbiter	96
5.19	Hardware Solutions	97
5.19.1	Test/Set Instruction	98
5.19.2	Swap Instruction	98
5.19.3	Fetch and Increment Instruction	99
<b>6</b>	<b>Locks</b>	<b>101</b>
6.1	Lock Taxonomy	101
6.2	Spin Lock	101
6.2.1	Implementation	102
6.3	Blocking Locks	103
6.3.1	Mutex Lock	104
6.3.1.1	Implementation	104
6.3.1.2	uOwnerLock	108
6.3.1.3	Mutex-Lock Release-Pattern	108
6.3.1.4	Stream Locks	109
6.3.2	Synchronization Lock	110
6.3.2.1	Implementation	110
6.3.2.2	uCondLock	113
6.3.2.3	Programming Pattern	114
6.3.3	Barrier	114
6.3.3.1	Fetch Increment Barrier	116
6.3.3.2	uBarrier	116
6.3.4	Binary Semaphore	118
6.3.4.1	Implementation	119
6.3.5	Counting Semaphore	120

6.3.5.1	Implementation	121
6.4	Lock Programming	122
6.4.1	Precedence Graph	122
6.4.2	Buffering	124
6.4.2.1	Unbounded Buffer	124
6.4.2.2	Bounded Buffer	125
6.4.3	Lock Techniques	126
6.4.4	Readers and Writer Problem	127
6.4.4.1	Solution 1	128
6.4.4.2	Solution 2	129
6.4.4.3	Solution 3	130
6.4.4.4	Solution 4	130
6.4.4.5	Solution 5	132
6.4.4.6	Solution 6	134
6.4.4.7	Solution 7	136
<b>7</b>	<b>Concurrent Errors</b>	<b>139</b>
7.1	Race Condition	139
7.2	No Progress	139
7.2.1	Live-lock	139
7.2.2	Starvation	140
7.2.3	Deadlock	140
7.2.3.1	Synchronization Deadlock	141
7.2.3.2	Mutual Exclusion Deadlock	141
7.3	Deadlock Prevention	142
7.3.1	Synchronization Prevention	142
7.3.2	Mutual Exclusion Prevention	142
7.4	Deadlock Avoidance	143
7.4.1	Banker's Algorithm	143
7.4.2	Allocation Graphs	144
7.5	Detection and Recovery	146
7.6	Which Method To Chose?	147
<b>8</b>	<b>Indirect Communication</b>	<b>149</b>
8.1	Critical Regions	149
8.2	Conditional Critical Regions	150
8.3	Monitor	150
8.4	Scheduling (Synchronization)	151
8.4.1	External Scheduling	152
8.4.2	Internal Scheduling	153
8.5	Readers/Writer	155
8.6	Exceptions	158
8.7	Nested Monitor Calls	159
8.8	Intrusive Lists	160
8.9	Counting Semaphore, V, P vs. Condition, Signal, Wait	161

8.10	Monitor Types	162
8.11	Java Monitor	165
<b>9</b>	<b>Direct Communication</b>	<b>169</b>
9.1	Task	169
9.2	Scheduling	170
9.2.1	External Scheduling	170
9.2.2	Internal Scheduling	173
9.2.3	Accepting the Destructor	176
9.3	Increasing Concurrency	177
9.3.1	Server Side	177
9.3.1.1	Internal Buffer	178
9.3.1.2	Administrator	179
9.3.2	Client Side	180
9.3.2.1	Returning Values	180
9.3.2.2	Tickets	181
9.3.2.3	Call-Back Routine	181
9.3.2.4	Futures	182
<b>10</b>	<b>Optimization</b>	<b>189</b>
10.1	Sequential Optimizations	189
10.2	Memory Hierarchy	190
10.2.1	Cache Review	191
10.2.2	Cache Coherence	192
10.3	Concurrent Optimizations	194
10.3.1	Disjoint Reordering	195
10.3.2	Eliding	196
10.3.3	Replication	196
10.4	Memory Model	197
10.5	Preventing Optimization Problems	197
<b>11</b>	<b>Other Approaches</b>	<b>201</b>
11.1	Atomic (Lock-Free) Data-Structure	201
11.1.1	Compare and Set Instruction	201
11.1.2	Lock-Free Stack	202
11.1.3	ABA problem	203
11.1.4	Hardware Fix	204
11.2	Safe Memory Reclamation Problem	205
11.3	Exotic Atomic Instruction	206
11.4	General-Purpose GPU (GPGPU)	209
11.5	Concurrency Languages	211
11.5.1	Ada 95	211
11.5.2	SR/Concurrent C++	213
11.5.3	Java	214
11.5.4	Go	215

11.5.5 C++11 Concurrency . . . . .	218
11.6 Threads & Locks Library . . . . .	221
11.6.1 java.util.concurrent . . . . .	221
11.6.2 Pthreads . . . . .	223
11.7 OpenMP . . . . .	225
<b>Index</b>	<b>229</b>



# 1 Advanced Control Flow (Review)

- **Within** a routine, basic and advanced control structures allow virtually any control flow.
- For predicate only, **while** and **for** are interchangeable.

GOOD	GOOD
<pre>while ( predicate ) {   S1 }</pre>	<pre>for ( ; predicate ; ) {   S1 }</pre>

**for** allows adding/removing loop index for debugging.

- Do not use **while** to simulate **for**.

BAD	GOOD
<pre>int i = 0; while ( i &lt; 10 ) {   S1   i += 1; }</pre>	<pre>for ( int i = 0; i &lt; 10; i +=1 ) {   S1 }</pre>

- **Multi-exit loop** (or mid-test loop) has one or more exit locations occurring *within* the body of the loop, not just top (**while**) or bottom (**do-while**).

```
for ( ;; ) {           // infinite loop, while ( true )
  ...
  if ( ... ) break;   // middle exit
  ...
}
```

- Exit condition *reversed* from **while** and *outdented* (eye-candy) for readability
- Eliminates priming (duplicated) code necessary with **while**.

<pre>cin &gt;&gt; d; // priming while ( ! cin.fail() ) {   ...   cin &gt;&gt; d; }</pre>	<pre>for ( ;; ) {   cin &gt;&gt; d;   if ( cin.fail() ) break;   ... }</pre>
--	--

- Do not use multi-exit to simulate **while/for**, especially for loop index.

BAD	GOOD
<pre>for ( int i = 0; ; i += 1 ) {   if ( i == 10 ) break;   S1 }</pre>	<pre>for ( int i = 0; i &lt; 10; i += 1 ) {   S1 }</pre>

- A loop exit **NEVER** needs an **else** clause.

BAD	GOOD	BAD	GOOD
<pre>for ( ;; ) {   S1   if ( C1 ) {     S2   } else {     break;   }   S3 }</pre>	<pre>for ( ;; ) {   S1   if ( ! C1 ) break;   S2   S3 }</pre>	<pre>for ( ;; ) {   S1   if ( C1 ) {     break;   } else {     S2   }   S3 }</pre>	<pre>for ( ;; ) {   S1   if ( C1 ) break;   S2   S3 }</pre>

S2 is logically part of loop body *not* part of an if.

- Allow multiple exit conditions.

<pre>for ( ;; ) {   S1   if ( C1 ) { E1; break; }   S2   if ( C2 ) { E2; break; }   S3 }</pre>	<pre>bool flag1 = false, flag2 = false; while ( ! flag1 &amp;&amp; ! flag2 ) {   S1   if ( C1 ) flag1 = true;   } else {     S2     if ( C2 ) flag2 = true;   } else {     S3   } } if ( flag1 ) E1; else E2;</pre>
--	---

- Eliminate **flag variables** used solely to affect control flow, i.e., variable does not contain data associated with computation.
- **Flag variables are the variable equivalent to a goto** because they can be set/reset/tested at arbitrary locations in a program.

## 1.1 Static multi-level exit

- **Static multi-level exit** exits multiple control structures where exit point is *known* at compile time.
- Labelled exit (**break/continue**) provides this capability.

$\mu$ C++ / Java	C / C++
<pre> <b>BK:</b> { // good eye-candy   ... declarations ...   <b>SW:</b> switch ( ... ) {     <b>FR:</b> for ( ... ) {       ... <b>break BK;</b> ... // exit block       ... <b>break SW;</b> ... // exit switch       ... <b>break FR;</b> ... // exit loop     }     ...   }   ... } </pre>	<pre> {   ... declarations ...   switch ( ... ) {     for ( ... ) {       ... <b>goto BK;</b> ...       ... <b>goto SW;</b> ...       ... <b>goto FR;</b> ... // or break     } <b>FR;</b> ;     ...   } <b>SW;</b> ; // bad eye-candy   ... } <b>BK;</b> ; </pre>

- Why is it good practice to label all exits?
- Eliminate all flag variables with **multi-level exit!**

<pre> <b>F1:</b> for ( i = 0; i &lt; 10; i += 1 ) {   <b>F2:</b> for ( j = 0; j &lt; 10; j += 1 ) {     ...     if ( ... ) <b>break F2;</b> // outdent     ... // rest of loop     if ( ... ) <b>break F1;</b> // outdent     ... // rest of loop   } // for   ... // rest of loop } // for </pre>	<pre> <b>bool flag1 = false;</b> for ( i = 0; i &lt; 10 &amp;&amp; ! <b>flag1</b>; i += 1 ) {   <b>bool flag2 = false;</b>   for ( j = 0; j &lt; 10 &amp;&amp;     ! <b>flag1</b> &amp;&amp; ! <b>flag2</b>; j += 1 ) {     ...     if ( ... ) <b>flag2 = true;</b>     else {       ... // rest of loop       if ( ... ) <b>flag1 = true;</b>       else {         ... // rest of loop       } // if     } // if   } // for   if ( ! <b>flag1</b> ) {     ... // rest of loop   } // if } // for </pre>
--	--

- **Occasionally a flag variable is necessary!**

```

// Retain state from one inner lexical (static) scope to another.
int val; bool valDefault = false;
switch ( argc ) {
  ...
  case 3:
    if ( strcmp( argv[4], "d" ) ) valDefault = true; // default ?
    else val = stoi( argv[4] ); // value
  ...
} // switch

```

```

for ( ;; ) {
    ...
    if ( valDefault ) // do something
    else // do another
    ...
} // for

```

- Other uses of multi-level exit to remove duplicate code.

duplication		no duplication
<pre> if ( C1 ) {     S1;     if ( C2 ) {         S2;         if ( C3 ) {             S3;         } else             S4;     } else         S4; } else     S4; </pre>	<pre> C: {     if ( C1 ) {         S1;         if ( C2 ) {             S2;             if ( C3 ) {                 S3;             } else                 break C;         }     } } S4; // only once </pre>	<pre> {     if ( C1 ) {         S1;         if ( C2 ) {             S2;             if ( C3 ) {                 S3;             } else                 goto C;         }     } } S4; // only once C: ; </pre>

- Normal and labelled **break** are a **goto** with limitations.
  1. Cannot loop (only forward branch)  $\Rightarrow$  only loop constructs branch back.
  2. Cannot branch *into* a control structure.
- **Only use goto to perform static multi-level exit, e.g., simulate labelled break and continue.**

## 1.2 Dynamic Memory Allocation

- Stack allocation eliminates explicit storage-management and is more efficient than heap allocation — **“Use the STACK, Luke.”**

<pre> { // GOOD, use stack     cin &gt;&gt; size;     int arr[size]; // VLA, g++     ... // use arr[i] } </pre>	<pre> { // BAD, unnecessary dynamic allocation     cin &gt;&gt; size;     int * arr = new int[size];     ... // use arr[i]     delete [] arr; // why “[ ]”? } </pre>
---	--

- These are the situations where dynamic (heap) allocation is necessary.
  1. When storage must outlive the block in which it is allocated (ownership change).

```

Type * rtn(...) {
    Type * tp = new Type; // MUST USE HEAP
    ... // initialize/compute using tp
    return tp; // storage outlives block
} // tp deleted later

```

Similar to necessary flag variable: to retain state from a lower level.

2. When the amount of data read is unknown.

```
vector<int> input;
int temp;
for ( ;; ) {
    cin >> temp;
    if ( cin.fail() ) break;
    input.push_back( temp ); // implicit dynamic allocation
}
```

Does switching to `emplace_back` help?

3. When an array of objects must be initialized via the object's constructor and each element has a different value.

```
struct Obj {
    const int id; ... // possibly other fields
    Obj( int id ) : id{ id } { ... }
};
ostream & operator<<( ostream & os, const Obj & obj ) {
    return os << obj.id;
}
// need to initialize each array element
Obj obj[10]; // no default constructor! declaration fails
for ( int id = 0; id < 10; id += 1 )
    obj[id].id = id; // Obj::id is const! assignment fails
```

- o Forced to use pointers and dynamic allocation.

```
Obj * objs[size];
for ( int id = 0; id < size; id += 1 )
    objs[id] = new Obj{ id };
...
for ( int id = 0; id < size; id += 1 )
    delete objs[id];
```

- o Two declaration/initialization alternatives: `uArray` and `unique_ptr`.

<pre>{ // GOOD, use stack     cin &gt;&gt; size;     uArray( Obj, objs, size ); // macro     for ( int id = 0; id &lt; size; id += 1 )         objs[id]( id ); // constructor call     ... } // implicit array deallocate</pre>	<pre>{ // BAD, unnecessary dynamic allocation     cin &gt;&gt; size;     unique_ptr&lt;Obj&gt; objs[size];     for ( int id = 0; id &lt; size; id += 1 )         objs[id] = make_unique&lt;Obj&gt;( id );     ... } // implicit array deallocate</pre>
---	--

- o Like `unique_ptr`, `uArray` allocates `objs` without element constructor calls (placement `new` allocation) **and it proves subscript checking**.
- o Calls to `(..)` or `make_unique<T>(..)` initialize array elements.
- o Allocation for `uArray` is  $O(1)$  in stack; `unique_ptr` is  $O(N)$  in heap.
- o As for `unique_ptr`, use `*` for object and `->` for field access.

```
for ( int id = 0; id < size; id += 1 )
    cout << *objs[id] << ' ' << objs[id]->id << endl;
```

- o Use `uArray` instead of `unique_ptr` or `std::vector` as both use the heap.

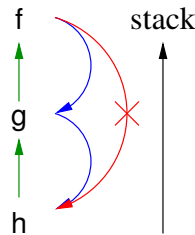
4. When large local variables are allocated on a small stack.

<pre> _Coroutine C {   void main() { // 64K stack     Obj <b>objs[100000]</b>; // overflow     // implicit initialize with ctor calls     ...   } // implicitly array deallocate }; </pre>	<pre> _Coroutine C {   void main() {     <b>uArrayPtr( Obj, objs, 100000 )</b>     // explicit initialize with ctor calls     ...   } // implicitly array deallocate }; </pre>
--	--

- `uArrayPtr` dynamically allocates entire array of objects in heap, and implicitly frees it at the end of the block (like `unique_ptr`).
- However, 1 allocation versus N for `unique_ptr` and subscript checking.
- Alternatives are large stacks (waste virtual space, see 3.8) or dynamic stack growth (complex and pauses).

## 2 Nonlocal Transfer

- Routine **activation** (call/invocation) introduces complex control flow.
- **Among** routines, control flow is controlled by call/return mechanism.



- routine h calls g calls f
- cannot return from f to h, terminating g's activation
- **Modularization**: from software engineering, any contiguous code block can be factored into a (helper) routine and called in the program (modulo scoping rules).
- Modularization fails when refactoring exits, e.g., multi-level exits:

```

B1: for ( i = 0; i < 10; i += 1 ) {
    ...
    B2: for ( j = 0; j < 10; j += 1 ) {
        ...
        if ( ... ) break B1;
        ...
    }
    ...
}

int rtn( ... ) {
    B2: for ( j = 0; j < 10; j += 1 ) {
        ...
        if ( ... ) break B1;
        ...
    }
    B1: for ( i = 0; i < 10; i += 1 ) {
        ... w = rtn( ... ) ...
    }
}

```

Does this compile?

- Software pattern: many routines have **multiple outcomes**.
  - normal: return normal result and transfer after call
  - exceptional: return alternative result and **not** transfer after call
- **Nonlocal transfer** allows a routine to transfer back to its caller but **not** after the call.

*C Two alternate return parameters, denoted by \* and implicitly named 1 and 2*

```

subroutine AltRet( c, *, * )
    integer c
    if ( c == 0 ) return      ! normal return
    if ( c == 1 ) return 1   ! alternate return
    if ( c == 2 ) return 2   ! alternate return
end

```

*C* Statements labelled 10 and 20 are alternate return points

```

call AltRet( 0, *10, *20 )
print *, "normal return 1"
call AltRet( 1, *10, *20 )
print *, "normal return 2"
return
10 print *, "alternate return 1"
call AltRet( 2, *10, *20 )
print *, "normal return 3"
return
20 print *, "alternate return 2"
stop
end

```

```

$ gfortran AltRtn.for
$ a.out
normal return 1
alternate return 1
alternate return 2

```

- Generalization of multi-exit loop and multi-level exit.
  - Control structures ends normally or with an exceptional transfer.
- Pattern acknowledges:
  - algorithms can have multiple outcomes
  - separating outcomes makes it easy to read and maintain a program
- **Pattern does not handle multiple levels of nested modularization.**
- If AltRet is further modularized, new routine has an alternate return to AltRet, which retains its alternate return to its caller.

*C* Two alternate return parameters, denoted by \* and implicitly named 1 and 2

```

subroutine AltRet2( c, *, * )
integer c
if ( c == 0 ) return ! normal return
if ( c == 1 ) return 1 ! alternate return
return 2
end

```

*C* Two alternate return parameters, denoted by \* and implicitly named 1 and 2

```

subroutine AltRet( c, *, * )
integer c
call AltRet2( c, *30, *40 )
return
30 return 1
40 if ( c == 2 ) return 2 ! alternate return
end

```

- Why not call AltRet2( c, \*10, \*20 )?



## 2.1 Traditional Approaches

- What are the traditional approaches for handling the multiple-outcome pattern?
- **return code**: returns value indicating normal or exceptional execution. e.g., printf() returns number of bytes transmitted or negative value.
- **status flag**: set shared (global) variable indicating normal or exceptional execution; the value remains as long as it is not overwritten. e.g., errno variable in UNIX.
- **fix-up routine**: a global and/or local routine called for an exceptional event to fix-up and return a corrective result so a computation can continue.

```
int fixup( int i, int j ) { ... } // local routine
rtn( a, b, fixup ); // fixup called for exceptional event
```

e.g., C++ has global routine-pointer new\_handler called when **new** fails.

- Techniques are often combined, e.g.:

```
if ( printf(..) < 0 ) {           // check return code for error
    perror( "printf:" );         // errno describes specific error
    abort();                     // terminate program
}
```

- **return union**: modern approach combining result/return-code and requiring return-code check on result access.
- **ALL** routines must return an appropriate union.

```
optional< int * > Malloc( size_t size ) {
    if ( random() % 2 ) return (int *)malloc( sizeof( int ) );
    return nullopt;           // no storage
}
optional< int > rtn( ) {
    optional< int * > p = Malloc( sizeof( int ) );
    if ( ! p ) return nullopt; // malloc successful (true/false) ?
    **p = 7; // compute
    if ( random() % 2 ) return **p;
    return nullopt;           // bad computation
}
int main() {
    srand( getpid() );
    optional< int > ret = rtn();
    if ( ret ) cout << *ret << endl; // rtn successful?
    else cout << "no storage or bad computation" << endl;
}
$ repeat 5 a.out
no storage or bad computation
7
no storage or bad computation
7
7
```

```

enum Alloc { NoStorage };
variant< int *, Alloc > Malloc( size_t size ) {
    if ( random() % 2 ) return (int *)malloc( sizeof( int ) );
    return NoStorage;
}
enum Comp { BadComp };
variant< int, Alloc, Comp > rtn( ) {
    variant< int *, Alloc > p = Malloc( sizeof( int ) );
    if ( ! holds_alternative<int *>(p) ) return NoStorage; // malloc successful ?
    *get<int *>(p) = 7;
    if ( random() % 2 ) return *get<int *>(p);
    return BadComp;
}

int main() {
    srand( getpid() );
    variant< int, Alloc, Comp > ret = rtn();
    if ( holds_alternative<int>(ret) ) cout << get<int>(ret) << endl;
    else if ( holds_alternative<Comp>(ret) ) cout << "bad computation" << endl;
    else cout << "no storage" << endl;
}
$ repeat 5 a.out
no storage
bad computation
no storage
bad computation
7

```

- Forces checking, unless explicitly access without holds\_alternative.
- Like Fortran, only returns one level.
- Drawbacks of traditional techniques:
  - checking return code or status flag is optional  $\Rightarrow$  can be delayed or omitted, i.e., passive versus active
  - return code mixes exceptional and normal values  $\Rightarrow$  enlarges type or value range; normal/exceptional type/values should be independent
- Testing and handling of return code or status flag is often done locally (inline), otherwise information may be lost; but local testing/handling:
  - makes code difficult to read; each call results in multiple statements
  - can be inappropriate, e.g., library routines should **not terminate program**
- Nonlocal testing from nested routine calls is difficult as multiple codes are returned for analysis, compounding the mixing problem.
- Status flag can be overwritten before examined, and cannot be used in a concurrent environment because of sharing issues (e.g., save errno)

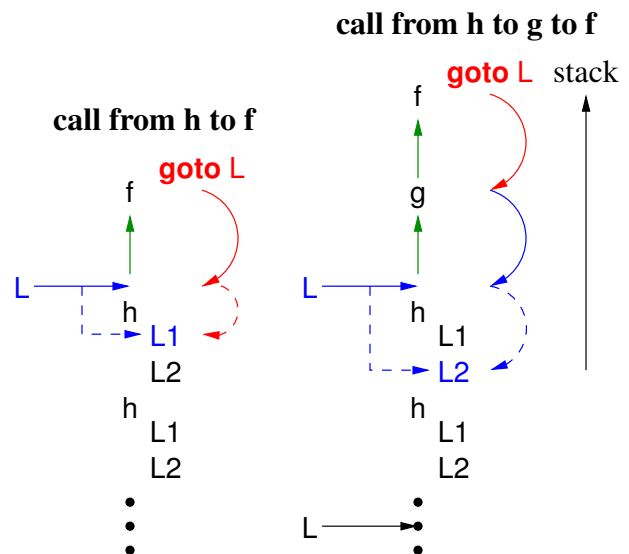
- Local fix-up routines increases the number of parameters.
  - increase cost of each call
  - must be passed through multiple levels enlarging parameter lists even when the fix-up routine is not used
- Nonlocal (global) fix-up routines, implemented with global routine pointer, have identical problems with status flags (e.g., `new_handler`).

## 2.2 Dynamic Multi-level Exit

- Rather than returning one level at a time, simpler for new modularized routine to bypass intermediate steps and transfer directly to original caller.
  - e.g., `AltRet2` transfers directly to program main, instead of `AltRet2` to `AltRet` to program main.
- **Dynamic multi-level exit** (DME) extend call/return semantics to transfer in the *reverse* direction to normal routine calls, requiring nonlocal transfer.

```

label L;
void f( int i ) {
    // nonlocal return
    if ( i == ... ) goto L;
}
void g( int i ) {
    if ( i > 1 ) { g( i - 1 ); return; }
    f( i );
}
void h( int i ) {
    if ( i > 1 ) { h( i - 1 ); return; }
    L = L1; // set dynamic transfer-point
    f( 1 ); goto S1;
L1: // handle L1 nonlocal return
S1: // continue normal execution
    L = L2; // set dynamic transfer-point
    g( 1 ); goto S2;
L2: // handle L2 nonlocal return
S2: // continue normal execution
}
  
```



- **label variable** contains:
  1. pointer to a block activation on the stack;
  2. transfer point within the block.
- Nonlocal transfer, **goto L**, is a two-step operation.
  1. direct control flow to the specified activation on the stack;
  2. then go to the transfer point (label constant) within the routine.

- Therefore, a label value is not statically/lexically determined.
  - recursion in g  $\Rightarrow$  unknown distance between f and h on stack.
  - what if L is set during the recursion of h?
- This complexity is why label constants have local scope.
- ***Transfer between goto and label value causes termination of stack block.***
- First, nonlocal transfer from f transfers to the label L1 in h's routine activation, terminating f's activation.
- Second, nonlocal transfer from f transfers to the static label L2 in the stack frame for h, terminating the stack frame for f and g.
- Termination is implicit for direct transferring to h or requires stack unwinding if activations contain objects with destructors or finalizers.
- DME is possible in C using:
  - jmp\_buf to declare a label variable,
  - setjmp to initialize a label variable,
  - longjmp to goto a label variable.
- DME allows multiple forms of returns to any level.
  - Normal return transfers to statement after the call, often implying completion of routine's algorithm.
  - Exceptional return transfers to statement **not** after the call, indicating an ancillary completion (but not necessarily an error).
- Unfortunately, nonlocal transfer is too general, allowing branching to almost anywhere, i.e., the goto problem.
- Simulate nonlocal transfer with return codes.

<pre> <b>label L;</b> void f( int i, int j ) {     for ( ... ) {         int k;         if ( i &lt; j &amp;&amp; k &gt; i ) <b>goto L;</b>         ...     } } void g( int i ) {     for ( ... ) {         int j;         ... f( i, j ); ...     } } void h() {     <b>L = L1;</b>     for ( ... ) {         int i;         ... g( i ); ...     }     ... <b>return;</b> // normal     <b>L1:</b> ... // exceptional } </pre>	<pre> int f( int i, int j ) {     <b>bool flag = false;</b>     for ( <b>! flag &amp;&amp;</b> ... ) {         int k;         if ( i &lt; j &amp;&amp; k &gt; i ) <b>flag = true;</b>         <b>else { ... }</b>     }     <b>if ( ! flag ) { ... }</b>     <b>return flag ? -1 : 0;</b> } int g( int i ) {     <b>bool flag = false;</b>     for ( <b>! flag &amp;&amp;</b> ... ) {         int j;         ... <b>if ( f( i, j ) == -1 ) flag = true</b>         <b>else { ... }</b>     }     <b>if ( ! flag ) { ... }</b>     <b>return flag ? -1 : 0;</b> } void h() {     <b>bool flag = false;</b>     for ( <b>! flag &amp;&amp;</b> ... ) {         int i;         ... <b>if ( g( i ) == -1 ) flag = true;</b>         <b>else { ... }</b>     }     <b>if ( ! flag ) { ... return; }</b>     ... } </pre>
---	---

## 2.3 Exception Handling

- DME, i.e., nonlocal transfer among routines, is often called **exception handling**.
- Exception handling is more than error handling.
- An **exceptional event** is an event that is (usually) known to exist but which is *ancillary* to an algorithm.
  - an exceptional event usually occurs with low frequency
  - e.g., division by zero, I/O failure, end of file, pop empty stack
- An **exception handling mechanism** (EHM) provides some or all of the alternate kinds of control-flow.
- Very difficult to simulate EHM with simpler control structures.
- Exceptions are supposed to make certain programming tasks easier, like robust programs.

- Robustness results because exceptions are active versus passive, forcing programs to react immediately when an exceptional event occurs.
- An EHM is not a panacea and only as good as the programmer using it.

## 2.4 Terminology

- **execution** is the language unit in which an exception can be raised, usually any entity with its own runtime stack.
- **exception type** is a type name representing an exceptional event.
- **exception** is an instance of an exception type, generated by executing an operation indicating an ancillary (exceptional) situation in execution.
- **raise (throw)** is the special operation that creates an exception.
- **source execution** is the execution raising an exception.
- **faulting execution** is the execution changing control flow due to a raised exception.
- **local exception** is when an exception is raised and handled by the same execution  $\Rightarrow$  source = faulting.
- **nonlocal exception** is when an exception is raised by a source execution but **delivered** to a different faulting execution  $\Rightarrow$  source  $\neq$  faulting.
- **concurrent exception** is a nonlocal exception, where the source and faulting executions are executing concurrently.
- **propagation** directs control from a raise in the source execution to a handler in the faulting execution.
- **propagation mechanism** is the rules used to locate a handler.
  - most common propagation-mechanisms give precedence to handlers higher in the lexical/call stack
    - \* specificity versus generality
    - \* efficient linear search during propagation
- **handler** is inline (nested) routine responsible for handling raised exception.
  - handler **catches** exception by **matching** with one or more exception types
  - after catching, a handler executes like a normal subroutine
  - handler can return, reraise the current exception, or raise a new exception
  - **reraise** terminates current handling and continues propagation of caught exception.
    - \* useful if a handler cannot deal with an exception but needs to propagate same exception to handler further down the stack.

- \* provided by a raise statement without an exception type:
  - ... **throw**; // no exception type
  - where a raise must be in progress.
- an exception is **handled** only if the handler returns rather than reraises
- **guarded block** is a language block with associated handlers, e.g., try-block in C++/Java.
- **unguarded block** is a block with no handlers.
- **termination** means control cannot return to the raise point.
  - all blocks on the faulting stack from the raise block to the guarded block handling the exception are terminated, called **stack unwinding**
- **resumption** means control returns to the raise point  $\Rightarrow$  no stack unwinding.
- EHM = Exception Type + Raise (exception) + Propagation + Handlers

## 2.5 Execution Environment

- The execution environment has a significant effect on an EHM.
- An object-oriented concurrent environment requires a more complex EHM than a non-object-oriented sequential environment.
- E.g., objects may have destructors that must be executed no matter how the object ends, i.e., by normal or exceptional termination.

```

class T {
    int *i;
    T() { i = new int[10]; ... }
    ~T() { delete [] i; ... } // must free storage
};
L: {
    T t; // constructor must be executed
    ... if ( ... ) break L;
    ...
} // destructor must be executed

```

- Control structures with **finally** clauses must always be executed (e.g., Java/ $\mu$ C++).

Java	$\mu$ C++
<pre> L: try {     infile = new Scanner( new File( "abc" ) );     ... if ( ... ) break L;     ... } finally { // always executed     infile.close(); // must close file } </pre>	<pre> L: try {     infile = new ifstream( "abc" );     ... if ( ... ) break L; // alt 1     ... // alt 2 } _Finally { // always executed     infile.close(); // must close file     delete infile; // deallocate } </pre>

- Hence, terminating a block complicates the EHM as object destructors (and recursively for nested objects) and **finally** clauses must be executed.
- For C++, a direct nonlocal transfer is often impossible, because of local objects with destructors, requiring linear stack unwinding.
- Also, complex execution-environment involving continuation, coroutine, task, each with its own execution stack.
- Given multiple stacks, an EHM can be more sophisticated, resulting in more complexity.
  - e.g., if no handler is found in one stack, it is possible to continue propagating the exception in another stack.

## 2.6 Implementation

- DME is *limited* in most programming languages using exception handling.

<pre> <b>struct E {};</b> // label <b>void f(...)</b> {     ...     <b>throw E();</b> // raise     // control never returns here } <b>int main()</b> {     <b>try</b> {         f(...);     } <b>catch( E )</b> {...} // handler 1     <b>try</b> {         f(...);     } <b>catch( E )</b> {...} // handler 2     ... } </pre>	<pre> <b>label L;</b> <b>void f(...)</b> {     ...     <b>goto L;</b> } <b>int main()</b> {     <b>L = L1;</b> // set transfer-point     f(...); <b>goto S1;</b>     L1: // handle nonlocal return     <b>S1: L = L2;</b> // set transfer-point     f(...); <b>goto S2;</b>     L2: // handle nonlocal return     S2: ; ... } </pre>
---	--

- To implement throw/catch, the throw must know the last guarded block with a handler for the raised exception type.
- One approach is to:
  - associate a label variable with each exception type
  - set label variable on entry to each guarded block with handler for the type
  - reset label variable on exit to previous value, i.e., previous guarded block for that type
- However, setting/resetting label variable on **try** block entry/exit has a cost (small).
  - rtn called million times but exception E never raised  $\Rightarrow$  million unnecessary operations.

```

void rtn( int i ) {
    try {
        ...
    } catch( E ) { ... }
}

```

// set label on entry  
// reset label on exit



- Instead, **catch**/destructor data is stored once externally for each block and handler found by linear search during a stack walk (no direct transfer).
- Advantage, millions of **try** entry/exit, but only tens of exceptions raised.
- Hence, termination is often implemented using zero cost on guarded-block entry but an expensive approach on raise.

## 2.7 Static/Dynamic Call/Return

- All routine/exceptional control-flow can be characterized by two properties:
  1. static/dynamic call: routine/exception name at the call/raise is looked up statically (compile-time) or dynamically (runtime).
  2. static/dynamic return: after a routine/handler completes, it returns to its static (definition) or dynamic (call) context.

return/handled	call/raise	
	static	dynamic
static	1) sequel	3) termination exception
dynamic	2) routine	4) routine pointer, virtual routine, resumption

- E.g., case 2) is a normal routine, with static name lookup at the call and a dynamic return.

## 2.8 Static Propagation (Sequel)

- Case 1) is called a **sequel**, which is a routine with no return value, where:
  - the sequel name is looked up lexically at the call site, but
  - control returns to the end of the block in which the sequel is declared.

```

A: for ( ;; ) {
    B: for ( ;; ) {
        C: for ( ;; ) {
            ...
            if ( ... ) { break A; }
            ...
            if ( ... ) { break B; }
            ...
            if ( ... ) { break C; }
            ...
        }
    }
}

for ( ;; ) {
    sequel S1( ... ) { ... } // nested
    void M1( ... ) {
        ... if ( ... ) S1( ... ); ...
    }
    for ( ;; ) {
        sequel S2( ... ) { ... } // nested
        C: for ( ;; ) {
            M1( ... ); // modularize
            if ( ... ) S2( ... ); // modularize
            ...
            if ( ... ) break C;
            ...
        }
    } // S2 static return
} // S1 static return

```

- Without a sequel, it is impossible to modularize code with static exits.
- $\Rightarrow$  propagation is along the lexical structure
- Adheres to the termination model, as the stack is unwound.
- Sequel handles termination for a *non-recoverable* event (simple exception handling).

```

{ // new block
  sequel StackOverflow(...) { ... } // handler
  class stack {
    void push( int i ) {
      if (...) StackOverflow(...); // 2nd outcome
    } // 1st outcome
    ...
  };
  stack s;
  ... s.push( 3 ); ... // overflow ?
} // sequel returns here

```

- The advantage of the sequel is the handler is statically known (like static multi-level exit), and can be as efficient as a direct transfer.
- The disadvantage is that the sequel only works for monolithic programs because it must be statically nested at the point of use.
  - Fails for modular (library) code as the static context of the module and user code are disjoint.
  - E.g., if stack is separately compiled, the sequel call in push no longer knows the static blocks containing calls to it.

## 2.9 Dynamic Propagation

- Cases 3) and 4) are called termination and resumption, and both have dynamic raise with static/dynamic return, respectively.
- Dynamic propagation/static return (case 3) is also called dynamic multi-level exit (see Section 2.2, p. 11).
- The advantage is that dynamic propagation works for separately-compiled programs.
- The disadvantage (advantage) of dynamic propagation is the handler is not statically known.
  - without dynamic handler selection, the same action and context for that action is executed for every exceptional change in control flow.

### 2.9.1 Termination

- For termination:
  - control transfers from the start of propagation to a handler  $\Rightarrow$  dynamic raise (call)
  - when handler returns, it performs a static return  $\Rightarrow$  stack is unwound (like sequel)
- There are 2 basic termination forms for a *non-recoverable* operation: terminate and retry.
- **terminate** provides *limited* mechanism for block transfer on the call stack, like labelled **break**.

```

struct E {}; // label
void f(...) {
    ...
    throw E(); // raise
    // control never returns here
}
int main() {
    try {
        f(...);
    } catch( E ) {...} // handler 1
    try {
        f(...);
    } catch( E ) {...} // handler 2
    ...
}

```

- No intermediate code to forward alternative outcome (see return union examples page 9).

```

struct NoStorage {};
struct BadComp {};
int * Malloc( size_t size ) {
    if ( random() % 2 ) return (int *)malloc( sizeof( int ) );
    throw NoStorage();
}
int rtn( ) {
    int * p = Malloc( sizeof( int ) );
    // DO NOT HAVE TO FORWARD NoStorage
    *p = 7; // compute
    if ( random() % 2 ) return *p;
    throw BadComp();
}
int main() {
    srand( getpid() );
    try { cout << rtn() << endl; }
    catch( BadComp ) { cout << "bad computation" << endl; }
    catch( NoStorage ) { cout << "no storage" << endl; }
}

```

- C++ I/O can be toggled to raise exceptions versus return codes (like  $\mu$ C++).

C++	$\mu$ C++
<pre> ifstream infile; ofstream outfile; <b>outfile.exceptions( ios_base::failbit );</b> <b>infile.exceptions( ios_base::failbit );</b> switch ( argc ) {   case 3:     try {       outfile.open( argv[2] );     } <b>catch( ios_base::failure &amp; ) {...</b>       // fall through to handle input file   case 2:     try {       infile.open( argv[1] );     } <b>catch( ios_base::failure &amp; ) {...</b>       break;   default:     ... } // switch string line; try {   for ( ;; ) { // loop until end-of-file     getline( infile, line );     outfile &lt;&lt; line &lt;&lt; endl;   } } <b>catch( ios_base::failure &amp; ) {}</b> </pre>	<pre> ifstream infile; ofstream outfile;  switch ( argc ) {   case 3:     try {       outfile.open( argv[2] );     } <b>catch( uFile::Failure &amp; ) {...</b>       // fall through to handle input file   case 2:     try {       infile.open( argv[1] );     } <b>catch( uFile::Failure &amp; ) {...</b>       break;   default:     ... } // switch string line;  for ( ;; ) {   getline( infile, line );   <b>if ( infile.fail() ) break;</b> // no eof exception   outfile &lt;&lt; line &lt;&lt; endl; } </pre>

- ios::exception mask indicates stream state-flags throw an exception if set
- failure exception raised after failed open or end-of-file when failbit set in exception mask
- $\mu$ C++ provides exceptions for I/O errors, but no exception for eof.
- $\mu$ C++ also provides **bound catch clause**, where catch depends on the object raising exception.

```

_Exception E {};
struct Obj {
  void mem() { ... _Throw E{}; ... } // implicitly store 'this' into exception
};
int main() {
  Obj obj1, obj2;
  try {
    ... obj1.mem(); ... obj2.mem(); ... _Throw E{};
  } catch( obj1.E ) { // only handle E from obj1 exception
  } catch( obj2.E ) { // only handle E from obj2 exception
  } catch( E ) { // handle any E exception
  }
}

```

- Better separation of alternate outcomes without flag variables.

```

#include <fstream>
istream * infile = &cin;           // default value
ostream * outfile = &cout;        // default value
try {
    switch ( argc ) {
        case 3: case 2:
            // open input file first as output creates file
            infile = new ifstream();
            dynamic_cast<ifstream *>(infile)->open( argv[3] );
            if ( argc == 3 ) {
                outfile = new ofstream();
                dynamic_cast<ofstream *>(outfile)->open( argv[4] );
            } // if
            // FALL THROUGH
        case 1:                       // defaults
            break;
        default:                       // wrong number of options
            // error
    } // switch
} catch( *infile.uFile::Failure & ) { // input open failed
    cerr << "Error! Could not open input file \" " << argv[3] << "\" " << endl;
    exit( EXIT_FAILURE );           // TERMINATE
} catch( *outfile.uFile::Failure & ) { // output open failed
    cerr << "Error! Could not open output file \" " << argv[4] << "\" " << endl;
    exit( EXIT_FAILURE );           // TERMINATE
} // try

```

- Why separate declaration and open versus `infile = new ifstream( argv[3] )`?
- Why is dynamic cast necessary for open?
- **retry** is a combination of termination with special handler semantics, i.e., restart the guarded block handling the exception (Eiffel). (Pretend end-of-file is an exception of type Eof.)

Retry	Simulation
<pre> char readfiles( char *files[], int N ) {     int i = 0, value;     ifstream infile;     infile.open( files[i] );      try {         ... infile &gt;&gt; value; ...     } retry( Eof ) {         i += 1;         infile.close();         if ( i == N ) goto Finished;         infile.open( files[i] );     }     Finished: ; } </pre>	<pre> char readfiles( char *files[], int N ) {     int i = 0, value;     ifstream infile;     infile.open( files[i] );     while ( true ) {         try {             ... infile &gt;&gt; value; ...         } catch( eof ) {             i += 1;             infile.close();             if ( i == N ) break;             infile.open( files[i] );         }     } } </pre>

- Because `retry` can be easily simulated, it is seldom supported directly.

## 2.9.2 Resumption

- **Resumption** provides a *limited* mechanism to generate new blocks on the call stack:
  - control transfers from the start of propagation to a handler  $\Rightarrow$  dynamic raise (call)
  - when handler returns, it is dynamic return  $\Rightarrow$  stack is NOT unwound (like routine)
- A resumption handler is a corrective action so a computation can continue.

```

void f( ..., /* no fixups */ ) {
    if ( ... ) resume E();
    // control returns here
}
int main() {
    try {
        f( ... ); // no fixups
    } catch( E ) {
        // handler 1
    }
    try {
        f( ... ); // no fixups
    } catch( E ) {
        // handler 2
    }
}

void f( ..., void (*fixup)( ... ) ) {
    if ( ... ) fixup( ... );
    // control returns here
}
void fixup1( ... ) {
    // handler 1
}
void fixup2( ... ) {
    // handler 2
}
int main() {
    f( ..., fixup1 );
    f( ..., fixup2 );
}

```

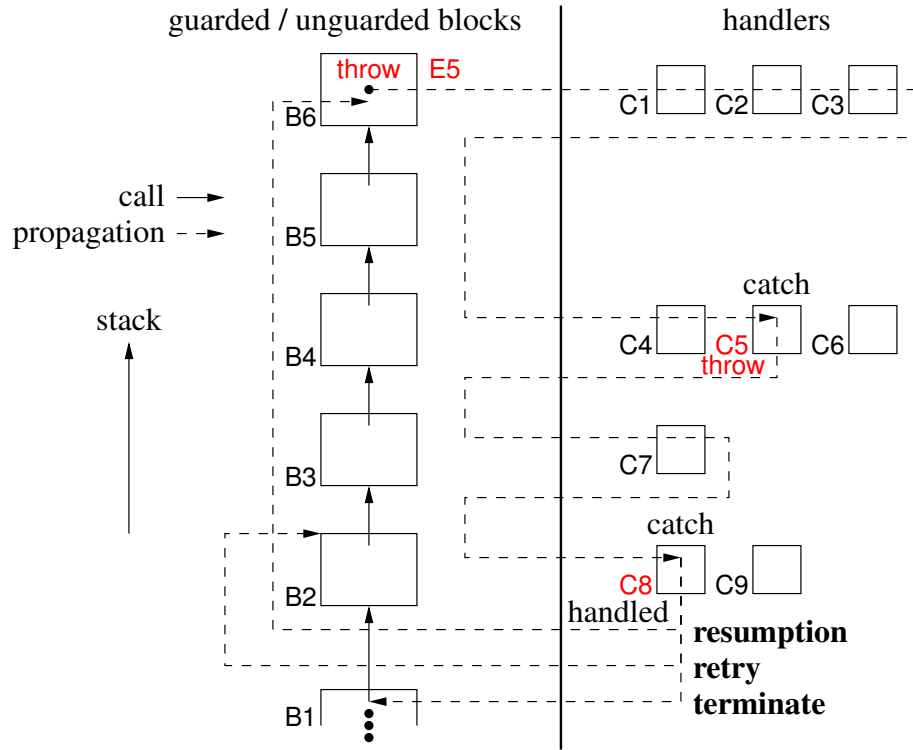
- No intermediate code to forward fixup down to raise point.

## 2.10 Exceptional Example

```

B1 {
B2   try {
B3     try {
B4       try {
B5         {
B6           try {
                ... throw E5(); ...
C1             } catch( E7 ) { ... }
C2             catch( E8 ) { ... }
C3             catch( E9 ) { ... }
                }
C4           } catch( E4 ) { ... }
C5           catch( E5 ) { ... throw; ... }
C6           catch( E6 ) { ... }
C7         } catch( E3 ) { ... }
C8       } catch( E5 ) { ... resume/retry/terminate }
C9     } catch( E2 ) { ... }
}

```

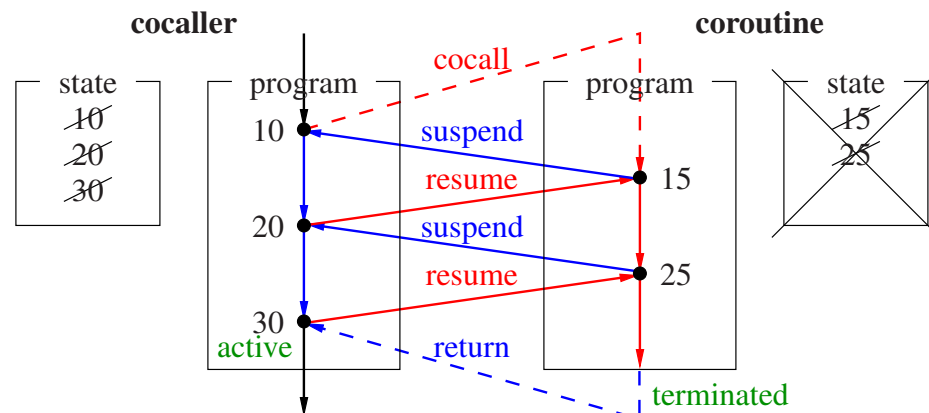






### 3 Coroutine

- A **coroutine** is a routine that can also be suspended at some point and resumed from that point when control returns.
- The state of a coroutine consists of:
  - an **execution location**, starting at the beginning of the coroutine and remembered at each suspend.
  - an **execution state** holding the data created by the code the coroutine is executing. ⇒ each coroutine has its own stack, containing its local variables and those of any routines it calls.
  - an **execution status**—**active** or **inactive** or **terminated**—which changes as control resumes and suspends in a coroutine.
- Hence, a coroutine does not start from the beginning on each activation; it is activated at the point of last suspension.
- In contrast, a routine always starts execution at the beginning and its local variables only persist for a single activation.



- A coroutine handles the class of problems that need to retain state between calls (e.g. plugin, device driver, finite-state machine).
- A coroutine executes synchronously with other coroutines; hence, no concurrency among coroutines.
- Coroutines are the precursor to concurrent tasks, and introduce the complex concept of suspending and resuming on separate stacks.
- Two different approaches are possible for activating another coroutine:
  1. A **semi-coroutine** acts asymmetrically, like non-recursive routines, by implicitly reactivating the coroutine that previously activated it.

2. A **full coroutine** acts symmetrically, like recursive routines, by explicitly activating a member of another coroutine, which directly or indirectly reactivates the original coroutine (activation cycle).

- These approaches accommodate two different styles of coroutine usage.

## 3.1 Semi-Coroutine

### 3.1.1 Fibonacci Sequence

$$f(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ f(n-1) + f(n-2) & n \geq 2 \end{cases}$$

- 3 states, producing unbounded sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

#### 3.1.1.1 Direct

- Compute and print Fibonacci numbers.

```
int main() {
    int fn, fn1, fn2;
    fn = 0; fn1 = fn;           // 1st case
    cout << fn << endl;
    fn = 1; fn2 = fn1; fn1 = fn; // 2nd case
    cout << fn << endl;
    for ( ;; ) {               // infinite loop
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn; // general case
        cout << fn << endl;
    }
}
```

- Convert to routine that generates a sequence of Fibonacci numbers on each call (no output):

```
int main() {
    for ( int i = 1; i <= 10; i += 1 ) { // first 10 Fibonacci numbers
        cout << fibonacci() << endl;
    }
}
```

- Examine different solutions.

## 3.1.1.2 Routine

```

int fn1, fn2, state = 1; // global variables
int fibonacci() {
    int fn;
    switch ( state ) {
        case 1:
            fn = 0; fn1 = fn; state = 2;
            break;
        case 2:
            fn = 1; fn2 = fn1; fn1 = fn; state = 3;
            break;
        case 3:
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            break;
    }
    return fn;
}

```

- unencapsulated global variables necessary to retain state between calls
- only one fibonacci generator can run at a time
- execution state must be explicitly retained

```

#define FIB_INIT { 0, 1 } /* first two Fibonacci numbers */
struct Fibonacci { int fn2, fn1; };
int fib( Fibonacci & f ) {
    int ret = f.fn2;
    int fn = f.fn1 + f.fn2; // only last state (3) in Fibonacci definition
    f.fn2 = f.fn1; f.fn1 = fn;
    return ret;
}
int main() {
    Fibonacci f1 = FIB_INIT, f2 = FIB_INIT; // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << fib( f1 ) << " " << fib( f2 ) << endl;
    }
}

```

- unencapsulated program global variables become encapsulated structure variables
- multiple fibonacci generators (objects) can run at a time
- execution state removed by precomputing first 2 Fibonacci numbers and returning  $f(n-2)$

## 3.1.1.3 Class

```

class Fibonacci {
    int fn, fn1, fn2, state = 1; // global class variables
public:
    int operator()() {          // functor
        switch ( state ) {
            case 1:
                fn = 0; fn1 = fn; state = 2;
                break;
            case 2:
                fn = 1; fn2 = fn1; fn1 = fn; state = 3;
                break;
            case 3:
                fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
                break;
        }
        return fn;
    }
};
int main() {
    Fibonacci f1, f2; // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1() << " " << f2() << endl;
    } // for
}

```

- unencapsulated program global variables become encapsulated object global variables
- multiple fibonacci generators (objects) can run at a time
- execution state still explicit or use initialization trick

## 3.1.1.4 Coroutine

```

_Coroutine Fibonacci { // : public uBaseCoroutine
    int fn; // used for communication
    void main() { // distinguished member
        int fn1, fn2; // retained between resumes
        fn = 0; fn1 = fn;
        suspend(); // return to last resume
        fn = 1; fn2 = fn1; fn1 = fn;
        suspend(); // return to last resume
        for ( ;; ) {
            fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
            suspend(); // return to last resume
        }
    }
public:
    int operator()() { // functor
        resume(); // transfer to last suspend
        return fn;
    }
};

```

```

int main() {
    Fibonacci f1, f2;    // multiple instances
    for ( int i = 1; i <= 10; i += 1 ) {
        cout << f1() << " " << f2() << endl;
    }
}

```

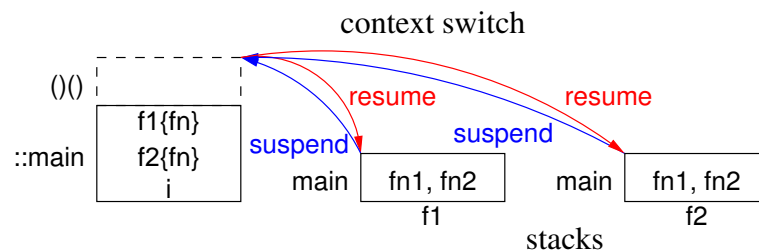
- **no explicit execution state!** (see direct solution)
- `_Coroutine` type wraps coroutine and provides *all* **class** properties
- distinguished member `main` (coroutine `main`) can be suspended and resumed
- no parameters or return value (supplied by **public** members and communication variables).
- coroutine `main` can be called (even recursively), but normally a **private/protected** member. Why?
- **compile with u++ command**
- All coroutines inherit from base type `uBaseCoroutine`:

```

class uBaseCoroutine {
protected:
    void resume();           // context switch to this
    void suspend();         // context switch to last resumer
    virtual void main() = 0; // starting routine for coroutine
public:
    uBaseCoroutine();
    uBaseCoroutine( unsigned int stackSize ); // set stack size
    void verify();           // check stack
    const char * setName( const char * name ); // printed in error messages
    const char * getName() const;
    uBaseCoroutine & starter() const; // coroutine performing first resume
    uBaseCoroutine & resumer() const; // coroutine performing last resume
};

```

- Program `main` called from hidden coroutine  $\Rightarrow$  **has coroutine properties**.
- `resume/suspend` cause a **context switch** between coroutine stacks



- first `resume` starts `main` on new stack (`cocall`); subsequent `resumes` reactivate last `suspend`.

- suspend reactivates last resume
- object becomes a coroutine on first resume; coroutine becomes an object when main ends
- routine frame at the top of the stack *knows* where to activate execution
- suspend/resume are **protected** members to prevent external calls. Why?
- Coroutine main does not have to return before a coroutine object is deleted.
- When deleted, a coroutine's stack is always unwound and any destructors executed. Why?
- **Warning, do not use catch(..) (catch any) in a coroutine, if it may be deleted before terminating, because a cleanup exception is raised to force stack unwinding (implementation issue).**

### 3.1.2 Format Output

Unstructured input:

```
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Structured output:

```
abcd  efgh  ijkl  mnop  qrst
uvwx  yzab  cdef  ghij  klmn
opqr  stuv  wxyz
```

blocks of 4 letters, separated by 2 spaces, grouped into lines of 5 blocks.

#### 3.1.2.1 Direct

- Read characters and print formatted output.

```
int main() {
    int g, b;
    char ch;
    cin >> noskipws;           // turn off white space skipping
    for ( ;; ) {               // for as many characters
        for ( g = 0; g < 5; g += 1 ) { // groups of 5 blocks
            for ( b = 0; b < 4; b += 1 ) { // blocks of 4 chars
                for ( ;; ) {         // for newline characters
                    cin >> ch;       // read one character
                    if ( cin.fail() ) goto fini; // eof ? multi-level exit
                    if ( ch != '\n' ) break; // ignore newline
                }
                cout << ch;          // print character
            }
            cout << " ";            // print block separator
        }
        cout << endl;              // print group separator
    }
    fini;
    if ( g != 0 || b != 0 ) cout << endl; // special case
}
```

- Convert to routine passed one character at a time to generate structured output (no input).

### 3.1.2.2 Routine

```

int g, b; // global variables
void fmtLines( char ch ) {
    if ( ch != -1 ) { // not EOF ?
        if ( ch == '\n' ) return; // ignore newline
        cout << ch; // print character
        b += 1;
        if ( b == 4 ) { // block of 4 chars
            cout << " "; // block separator
            b = 0;
            g += 1;
        }
        if ( g == 5 ) { // group of 5 blocks
            cout << endl; // group separator
            g = 0;
        }
    } else {
        if ( g != 0 || b != 0 ) cout << endl; // special case
    }
}
int main() {
    char ch;
    cin >> noskipws; // turn off white space skipping
    for ( ;; ) { // for as many characters
        cin >> ch;
        if ( cin.fail() ) break; // eof ?
        fmtLines( ch );
    }
    fmtLines( -1 ); // indicate EOF
}

```

- must retain variables b and g between successive calls.
- only one instance of formatter
- linearize (flatten) loops: one loop, lots of **if** statements

### 3.1.2.3 Class

```

class Format {
    int g, b; // global class variables
public:
    Format() : g( 0 ), b( 0 ) {}
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) {
        if ( ch == '\n' ) return; // ignore newline
        cout << ch; // print character
        b += 1;
    }
}

```

```

        if ( b == 4 ) {           // block of 4 chars
            cout << " ";       // block separator
            b = 0;
            g += 1;
        }
        if ( g == 5 ) {         // group of 5 blocks
            cout << endl;      // group separator
            g = 0;
        }
    }
};

int main() {
    Format fmt;
    char ch;
    cin >> noskipws;           // turn off white space skipping
    for ( ;; ) {               // for as many characters
        cin >> ch;             // read one character
        if ( cin.fail() ) break; // eof ?
        fmt.prt( ch );
    }
}

```

- Solves encapsulation and multiple instances issues, but explicitly managing execution state.

### 3.1.2.4 Coroutine

```

_Coroutine Format {
    char ch;                       // used for communication
    int g, b;                       // global because used in destructor
    void main() {
        for ( ;; ) {               // for as many characters
            for ( g = 0; g < 5; g += 1 ) { // groups of 5 blocks
                for ( b = 0; b < 4; b += 1 ) { // blocks of 4 characters
                    for ( ;; ) {    // for newline characters
                        suspend();
                        if ( ch != '\n' ) break; // ignore newline
                    }
                    cout << ch;     // print character
                }
                cout << " ";       // print block separator
            }
            cout << endl;         // print group separator
        }
    }
};

public:
    Format() { resume(); }         // start coroutine
    ~Format() { if ( g != 0 || b != 0 ) cout << endl; }
    void prt( char ch ) { Format::ch = ch; resume(); }
};

```

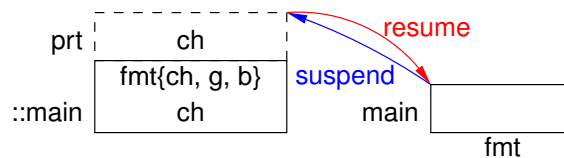


```

int main() {
    Format fmt;
    char ch;
    cin >> noskipws;           // turn off white space skipping
    for ( ;; ) {
        cin >> ch;             // read one character
        if ( cin.fail() ) break; // eof ?
        fmt.prt( ch );
    }
}

```

- resume in constructor allows coroutine main to get to 1st input suspend.



### 3.1.3 Correct Coroutine Usage

- **Eliminate computation or flag variables retaining information about execution state.**
- E.g., sum even and odd digits of 10-digit number, where each digit is passed to coroutine:

BAD: Explicit Execution State	GOOD: Implicit Execution State
<pre> for ( int i = 0; i &lt; 10; i += 1 ) {     if ( i % 2 == 0 ) // even ?         even += digit;     else         odd += digit;     suspend(); } </pre>	<pre> for ( int i = 0; i &lt; 5; i += 1 ) {     even += digit;     suspend();     odd += digit;     suspend(); } </pre>

- Right example illustrates coroutine “Zen”; let it do the work.
- E.g., a BAD solution for the previous Fibonacci generator is:

```

void main() {
    int fn1, fn2, state = 1;
    for ( ;; ) {
        switch (state) {           // no Zen
            case 1:
                fn = 0; fn1 = fn;
                state = 2;
                break;
            case 2:
                fn = 1; fn2 = fn1; fn1 = fn;
                state = 3;
                break;
            case 3:
                fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
                break;
        }
        suspend();                // no Zen
    }
}

```

- Coroutine's capabilities not used:
  - explicit flag variable controls execution state
  - original program structure lost in **switch** statement
- Must do more than just *activate* coroutine main to demonstrate understanding of retaining data and execution state within a coroutine.

### 3.1.4 Coroutine Construction

- Fibonacci and formatter coroutines express original algorithm structure (no restructuring).
- When possible, simplest coroutine construction is to write a direct (stand-alone) program.
- Convert to coroutine by:
  - putting processing code into coroutine main,
  - converting reads if program is consuming or writes if program is producing to suspend,
    - \* Fibonacci consumes nothing and produces (generates) Fibonacci numbers ⇒ convert writes (cout) to suspends.
    - \* Formatter consumes characters and only indirectly produces output (as side-effect) ⇒ convert reads (cin) to suspends.
  - use interface members and communication variables to transfer data in/out of coroutine.
- This approach is impossible for advanced coroutine problems.

### 3.2 $\mu$ C++ EHM

The following features characterize the  $\mu$ C++ EHM:

- exceptions must be generated from a specific kind of type.
- supports two kinds of raising: throw and resuming.
- supports two kinds of handlers, termination and resumption, matching with the kind of raise.
- supports propagation of nonlocal and concurrent exceptions.
- all exception types (user, runtime, and I/O) are grouped into a hierarchy.

### 3.3 Exception Type

- C++ allows any type to be used as an exception type.
- $\mu$ C++ restricts exception types to those types defined by `_Exception`.

```
_Exception exception-type-name { ... };
```

- An exception type has all the properties of a **class**.
- Every exception type must have a public default and copy constructor.
- An exception is the same as a class-object with respect to creation and destruction.

```
_Exception D { ... };
D d;           // local creation
_Resume d;
D *dp = new D; // dynamic creation
_Resume *dp;
delete dp;
_Throw D();   // temporary local creation
```

### 3.4 Inherited Members

- Each exception type inherits the following members from `uBaseException`:

```
class uBaseException { // like std::exception
    uBaseException( const char *const msg = "" );
    const char *const message() const;
    const uBaseCoroutine &source() const;
    const char *sourceName() const;
    virtual void defaultTerminate();
    virtual void defaultResume();
};
```

- `uBaseException( const char *const msg = "" )` – `msg` is printed if the exception is not caught.

- Message string is copied so it is safe to use within an exception even if the context of the raise is deleted.
- message returns the string message associated with an exception.
- source returns the coroutine/task that raised the exception.
  - coroutine/task may be deleted when the exception is caught so this reference may be undefined.
- sourceName returns the name of the coroutine/task that raised the exception.
  - name is copied from the raising coroutine/task when exception is created.
- defaultTerminate is implicitly called if an exception is thrown but not handled.
  - default action is to forward an UnhandledException exception to resumer/joiner.
- defaultResume is implicitly called if an exception is resumed but not handled.
  - default action is to throw the exception.

### 3.5 Raising

- There are two raising mechanisms: throwing and resuming.

```

    _Throw [ exception-type ] ;
    _Resume [ exception-type ] [ _At uBaseCoroutine-id ] ;
  
```

- If **\_Throw** has no *exception-type*, it is a rethrow.
- If **\_Resume** has no *exception-type*, it is a reresume.
- The optional **\_At** clause allows the specified exception or the currently propagating exception to be raised at another coroutine or task.
- Nonlocal/concurrent raise restricted to resumption as raising execution-state is often unaware of the handling execution-state.
- Resumption allows faulting execution greatest flexibility: it can process the exception as a resumption or rethrow the exception for termination.
- Exceptions in  $\mu\text{C++}$  are propagated differently from C++.

C++	$\mu\text{C++}$
<pre> class B {}; class D : public B {}; void f( B &amp; t ) { ... throw t; ... } try {     D m;     f( m ); } catch( D &amp; ) { cout &lt;&lt; "D" &lt;&lt; endl; } catch( B &amp; ) { cout &lt;&lt; "B" &lt;&lt; endl; }           </pre>	<pre> _Exception B {}; _Exception D : public B {}; void f( B &amp; t ) { ... _Throw t; ... } try {     D m;     f( m ); } catch( D &amp; ) { cout &lt;&lt; "D" &lt;&lt; endl; } catch( B &amp; ) { cout &lt;&lt; "B" &lt;&lt; endl; }           </pre>

- In C++, routine *f* is passed an object of derived type *D* but throws an object of base type *B*.
- In  $\mu$ C++, routine *f* is passed an object of derived type *D* and throws the original object of type *D*.
- This change allows handlers to catch the specific (derived) rather than the general (base) exception-type.

## 3.6 Handler

- $\mu$ C++ has two kinds of handlers, termination and resumption, which match with the kind of raise.

### 3.6.1 Termination

- The  $\mu$ C++ termination handler is the **catch** clause of a **try** block, i.e., same as in C++.

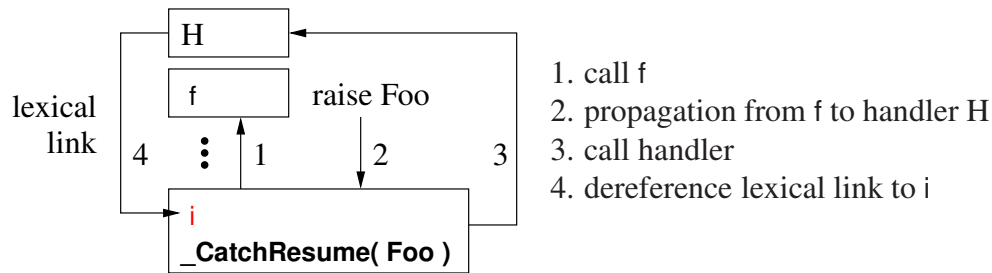
### 3.6.2 Resumption

- $\mu$ C++ extends the **try** block to include resumption handlers.
- Resumption handler is denoted by a **\_CatchResume** clause after **try** body:

```
try {
    ...
}
  _CatchResume( E1 ) { ... } // must appear before catch clauses
  // more _CatchResume clauses
  _CatchResume( ... ) { ... } // must be last _CatchResume clause
  catch( E2 ) { ... } // must appear after _CatchResume clauses
  // more catch clauses
  catch( ... ) { ... } // must be last catch clause
```

- Any number of resumption handlers can be associated with a **try** block.
- All **\_CatchResume** handlers must precede any **catch** handlers, solely to make reading the two sets of clauses easier.
- Like **catch(...)** (catch-any), **\_CatchResume(...)** must appear at the end of the list of the resumption handlers.
- Resumption handler can access types and variables visible in its local scope.

```
typedef int Foo;
Foo i;
try {
    f(...) // f is recursive and raises Foo
} _CatchResume( Foo & e ) { // handler H
    Foo fix = i; // use type and variable in local scope
    ... e = fix ... // change _Resume block
}
```



- **lexical link** is like **this** but to declaration block rather than object (lambda capture [&]).
- **Resumption handler cannot perform a break, continue, goto, or return.**
  - Resumption handler is corrective action so computation can continue.
  - If correction impossible, handler should **throw** an exception not step into an enclosing block to cause the stack to unwind.

```

B: try {
    f(); // recursive calls and _Resume E()
} _CatchResume( E e ) { // handler H
    ... break B; // force static return (disallowed)
    _Throw e; // force recovery (allowed)
}

```

- Assume **try** body makes recursive calls to f, so **break** must unwind stack to transfer into stack frame B (nonlocal transfer).
- Throw may find another recovery action closer to raise point than B that can deal with the problem.

### 3.6.3 Termination/Resumption

- The raise dictates set of handlers examined during propagation:
  - terminating propagation (**\_Throw**) only examines termination handlers (**catch**),
  - resuming propagation (**\_Resume**) only examines resumption handlers (**\_CatchResume**).
- Exception types in each set can overlap.

```

_Exception E {};
void rtn() {
    try {
        _Resume E();
    } _CatchResume( E & e ) { ... _Throw e; } // H1
    catch( E & e ) { ... } // H2
}

```

- Resumption handler H1 is invoked by the resume in the **try** block generating call stack:

```
rtn → try{ _CatchResume( E ), catch( E ) → H1
```

- Handler H1 throws E and the stack is unwound until the exception is caught by termination-handler **catch**( E ) and handler H2 is invoked.

rtn → H2

- The termination handler is available as resuming does not unwind the stack.
- Note interaction between resuming, defaultResume, and throwing:

```
_Exception R {};
void rtn() {
    try {
        _Resume R();           // resume not throw
    } catch( R & ) { ... }     // H1, no _CatchResume!!!
}
```

- This generates the following call stack as there is no eligible resumption handler (or there is a handler but marked ineligible):

rtn → **try{}catch( R )** → defaultResume

- **When defaultResume is called, the default action throws R** (see Section 3.4, p. 35).

rtn → H1

- Terminating propagation unwinds the stack until there is a match with the **catch** clause in the **try** block.

### 3.6.4 Object Binding

- **\_Resume**/**\_Throw** implicitly store the **this** associated with the member raising an exception.
- For a static member or free routine, there is no binding (no **this**).
- For non-local raise, the binding is the coroutine/task object executing the raise.

### 3.6.5 Bound Handlers

- **catch** / **\_CatchResume** provide object-specific matching.

```
catch( raising-object . exception-declaration ) { ... }
_CatchResume( raising-object . exception-declaration ) { ... }
```

- The “catch-any” handler, “...”, does not have a bound form.
- An exception is caught when the bound and handler objects are equal, and the raised exception equals the handler exception or its base-type.

### 3.7 Nonlocal Exceptions

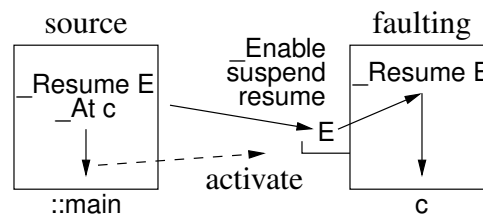
- Nonlocal exceptions are exceptions raised by a source execution at a faulting execution.
- Nonlocal exceptions are possible because each coroutine (execution) has its own stack.
- Nonlocal exceptions are raised using `_Resume ... _At ...`

```

_Exception E {};
_Coroutine C {
    void main() {
        // initialization, no nonlocal delivery
        try { // setup handlers
            _Enable { // allow nonlocal exceptions
                ... suspend(); ... // inside suspend is _Resume E();
            } // disable all nonlocal exceptions
        } _CatchResume( E ) { ... // option 1: continue after suspend
        } catch( E ) { ... // option 2: continue after try
        }
        // finalization, no nonlocal delivery
    }
public:
    C() { resume(); } // prime try (not always possible)
    void mem() { resume(); }
};
int main() {
    C c;
    _Resume E() _At c; // exception pending
    c.mem(); // trigger exception
}

```

- For nonlocal resumption, `_Resume` is a *proxy* for actual raise in the faulting coroutine ⇒ **non-local resumption becomes local resumption.**



- While source delivers nonlocal exception immediately, propagation only occurs when faulting becomes active.  
⇒ **must suspend back to or call a member that does a resume of the faulting coroutine**
- Faulting coroutine performs local `_Resume` implicitly at detection points for nonlocal exceptions, e.g., in `_Enable`, `suspend`, `resume`.
- Handler does not return to the proxy raise; control returns to the implicit local raise at exception delivery, e.g., back in `_Enable`, `suspend`, `resume`.



- Multiple nonlocal exceptions are queued and delivered in FIFO order depending on the current enabled exceptions.
- **Nonlocal delivery is initially disabled for a coroutine**, so handlers can be set up before any exception can be delivered (also see Section 5.11, p. 81).
- Hence, nonlocal exceptions must be explicitly enabled before delivery can occur with `_Enable`.
- `μC++` allows dynamic enabling and disabling of individual exception types versus all exception types.

```

    _Enable <E1><E2>... {
        // exceptions E1, E2 enabled
    }
    _Disable <E1><E2>... {
        // exceptions E1, E2 disabled
    }

```

- No exceptions is shorthand for specifying all nonlocal exceptions.
- Nested `_Enable` or `_Disable` blocks are additive  $\Rightarrow$  set of enabled or disabled exceptions increases on entry and decreases on exit.

```

    _Enable <E> { // E enabled
        _Enable <F> // E & F enabled
        _Enable { // all except. enabled
            } // E & F enabled
        } // E enabled
    }
    _Disable <E> { // E disabled
        _Disable <F> // E & F disabled
        _Disable { // all except. disabled
            } // E & F disabled
        } // E disabled
    }

```

- Nested `_Enable` and `_Disable` blocks are subtractive  $\Rightarrow$  set of enabled or disabled exceptions decreases on entry and increases on exit.

```

    _Enable <E> { // E enabled
        _Disable <E> { // E disabled
            _Disable { // all except. disabled
                } // E disabled
            } // E enabled
        }
    }
    _Disable <E> { // E disabled
        _Enable <E> { // E enabled
            _Enable { // all except. enabled
                } // E enabled
            } // E disabled
        }
    }

```

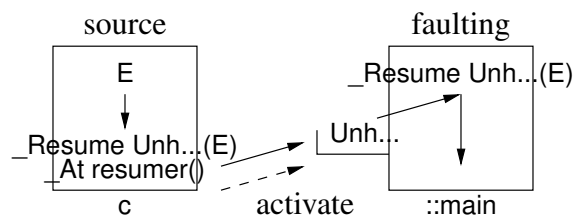
- An unhandled exception in a coroutine raises a nonlocal exception of type `uBaseCoroutine::UnhandledException` at the coroutine's *last resumer* and then terminates the coroutine.

```

_Exception E {};
_Coroutine C {
    void main() { _Throw E(); } // unwind
    // default Terminate ⇒ _Resume UnhandledException()_At resumer()
    // ⇒ coroutine activates last resumer (not starter) and terminates
public:
    void mem() { resume(); } // inside resume is _Resume UnhandledException()
};
int main() {
    C c;
    try { c.mem(); // resume coroutine
    } _CatchResume( uBaseCoroutine::Unh... & ) { ... // option 1: continue after resume
    } catch( uBaseCoroutine::Unh... & ) { ... // option 2: continue after try
    }
}

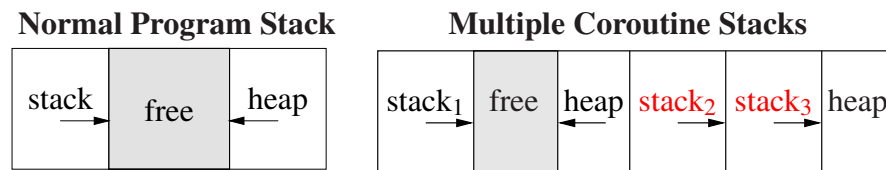
```

- Call to `c.mem` resumes coroutine `c` and then coroutine `c` throws exception `E` but does not handle it.
- When the base of `c`'s stack is reached, an exception of type `uBaseCoroutine::UnhandledException` is raised at `::main`, since it last resumed `c`.



- **\_CatchResume** continues from resume (dynamic return, fixup)
- **catch** continues after handler (static return, recover)
- Forwarding can occur across any number of coroutines, until a task main forwards and then the program terminates by calling `main`'s `set_terminate`.
- The original `E` exception is in the `UnhandledException` exception and can be thrown by `uh.triggerCause()`.
- *If the original (`E`) exception has a default-terminate routine, it can override `UnhandledException` behaviour (e.g., `abort`), or return and let it happen.*
- *While the coroutine terminates, control returns to its last resumer rather than its starter.*
- *Exception `UnhandledException` (and a few others) are always enabled.*

## 3.8 Memory Management



- Normally program stack expands to heap; but coroutine stacks expand to next stack.
- In fact, coroutine stacks are normally allocated in the heap.
- Default  $\mu$ C++ coroutine stack size is 256K **and it does not grow**.
- Adjust initial coroutine stack-size through coroutine constructor:

```

_Coroutine C {
public:
    C() : uBaseCoroutine( 8192 ) {}; // default 8K stack
    C( int size ) : uBaseCoroutine( size ) {}; // user specified stack size
    ...
};
C x, y( 16384 ); // x has an 8K stack, y has a 16K stack

```

- Check for stack overflow using coroutine member verify:

```

void main() {
    ... // declarations
    verify(); // check for stack overflow
    ... // code
}

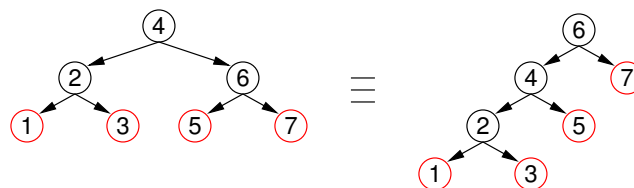
```

- Be careful allocating arrays in the coroutine main; sometimes necessary to allocate large arrays in heap. (see Point 4, p. 6)

## 3.9 Semi-Coroutine Examples

### 3.9.1 Same Fringe

- Two binary trees have same fringe if all leafs are equals from left to right.



- Requires iterator to traverse a tree, return the value of each leaf, and continue the traversal.
- No direct solution without additional data-structure (e.g., stack) to manage tree traversal.

- Coroutine uses recursive tree-traversal but suspends during traversal to return value.

```

template< typename T > class Btree {
    struct Node { ... }; ... // other members
public:
    _Coroutine Iterator {
        Node * cursor;
        void walk( Node * node ) { // walk tree
            if ( node == nullptr ) return;
            if ( node->left == nullptr && node->right == nullptr ) { // leaf?
                cursor = node;
                suspend(); // multiple stack frames
            } else {
                walk( node->left ); // recursion
                walk( node->right ); // recursion
            }
        }
        void main() { walk( cursor ); cursor = nullptr; }

public:
        Iterator( Btree<T> & btree ) : cursor( &btree.root ) {}
        T * next() {
            resume();
            return cursor;
        }
    };
    ... // other members
};

template<class T> bool sameFringe( BTree<T> & tree1, BTree<T> & tree2 ) {
    Btree<T>::Iterator iter1( btree1 ), iter2( btree2 ); // iterator for each tree
    T * t1, * t2;
    for ( ;; ) {
        t1 = iter1.next(); t2 = iter2.next();
        if ( t1 == nullptr || t2 == nullptr ) break; // one traversal complete ?
        if ( *t1 != *t2 ) return false; // elements not equal ?
    }
    return t1 == nullptr && t2 == nullptr; // both traversals completed ?
}

```

### 3.9.2 Device Driver

- Parse transmission protocol and return message text, e.g.:

... **STX** ...message ... **ESC ETX** ...message ... **ETX** 2-byte CRC ...

#### 3.9.2.1 Direct

```

int main() {
    enum { STX = '\002', ESC = '\033', ETX = '\003' };
    enum { MaxMsgLnth = 64 };
    unsigned char msg[MaxMsgLnth];
    ...
}

```

```

try {
  msg: for ( ;; ) {                                // parse messages
    int lnth = 0, checkval;
    do {
      byte = input( infile );                       // read bytes, throw Eof on eof
    } while ( byte != STX );                       // message start ?
    eom: for ( ;; ) {                               // scan message data
      byte = input( infile );
      switch ( byte ) {
        case STX:
          ...                                       // protocol error
          continue msg;                          // uC++ labelled continue
        case ETX:
          // end of message
          break eom;                               // uC++ labelled break
        case ESC:
          // escape next byte
          byte = input( infile );
          break;
      } // switch
      if ( lnth >= MaxMsgLnth ) { // buffer full ?
        ...                                       // length error
        continue msg;                          // uC++ labelled continue
      } // if
      msg[lnth] = byte;                          // store message
      lnth += 1;
    } // for
    byte = input( infile );                       // gather check value
    checkval = byte;
    byte = input( infile );
    checkval = (checkval << 8) | byte;
    if ( ! crc( msg, lnth, checkval ) ) ... // CRC error
  } // for
} catch( Eof ) {}
...
} // main

```

### 3.9.2.2 Coroutine

- Called by interrupt handler for each byte arriving at hardware serial port.

```

_Coroutine DeviceDriver {
  enum { STX = '\002', ESC = '\033', ETX = '\003' };
  enum { MaxMsgLnth = 64 };
  unsigned char byte;
  unsigned char * msg;
public:
  DeviceDriver( unsigned char * msg ) : msg( msg ) { resume(); }
  void next( unsigned char b ) { // called by interrupt handler
    byte = b;
    resume();
  }
}

```

```

private:
void main() {
  msg: for ( ;; ) {                               // parse messages
    int lnth = 0, checkval;
    do {
      suspend();
    } while ( byte != STX );                       // message start ?
    eom: for ( ;; ) {                               // scan message data
      suspend();

      switch ( byte ) {
        case STX:
          ...                                     // protocol error
          continue msg;                          // uC++ labelled continue
        case ETX:
          break eom;                              // end of message
          // uC++ labelled break
        case ESC:
          suspend();                              // escape next byte
          break;                                  // get escaped character
      } // switch

      if ( lnth >= MaxMsgLnth ) { // buffer full ?
        ...                                     // length error
        continue msg;                          // uC++ labelled continue
      } // if
      msg[lnth] = byte;                          // store message
      lnth += 1;
    } // for

    suspend();                                   // gather check value
    checkval = byte;
    suspend();
    checkval = (checkval << 8) | byte;
    if ( ! crc( msg, lnth, checkval ) ) ... // CRC error
  } // for
} // main
}; // DeviceDriver

```

## 3.9.3 Producer-Consumer

```

_Coroutine Cons {
    int p1, p2, status; bool done;
    void main() { // starter prod
        // 1st resume starts here
        int money = 1;
        for ( ; ! done; ) {
            cout << "cons " << p1 << " "
                << p2 << " pay $"
                << money << endl;
            status += 1;
            suspend(); // activate delivery or stop
            money += 1;
        }
        cout << "cons stops" << endl;
    } // suspend / resume(starter)
public:
    Cons() : status(0), done(false) {}
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1; Cons::p2 = p2;
        resume(); // activate main
        return status;
    }
    void stop() { done = true; resume(); } // activate main
};

_Coroutine Prod {
    Cons & c;
    int N;
    void main() { // starter ::main
        // 1st resume starts here
        for ( int i = 0; i < N; i += 1 ) {
            int p1 = rand() % 100; // products
            int p2 = rand() % 100;
            cout << "prod " << p1
                << " " << p2 << endl;
            int status = c.delivery( p1, p2 );
            cout << " stat " << status << endl;
        }
        c.stop();
        cout << "prod stops" << endl;
    } // suspend / resume(starter)

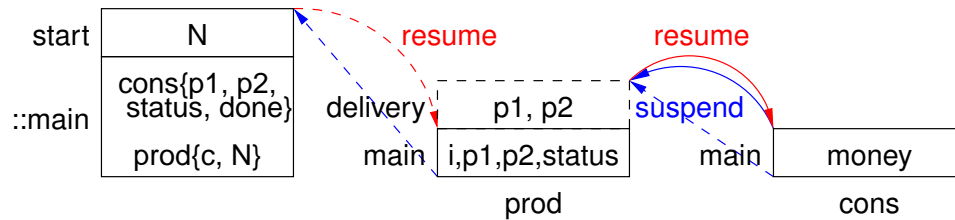
public:
    Prod( Cons & c ) : c(c) {}
    void start( int N ) {
        Prod::N = N;
        resume(); // activate main
    }
};

```

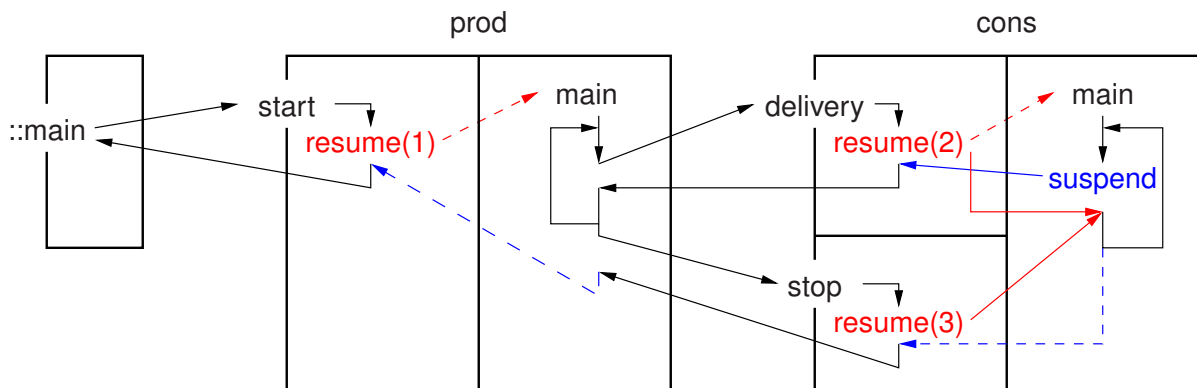
```

int main() {
  Cons cons;           // create consumer
  Prod prod( cons );  // create producer
  prod.start( 5 );    // start producer
}

```



- Do both Prod and Cons need to be coroutines?
- When coroutine main returns, it activates the coroutine that *started* main.
- The **starter** coroutine is the coroutine that does the first resume (cocall).
  - prod started cons.main, so control goes to prod suspended in stop.
  - ::main started prod.main, so control goes to ::main suspended in start.
- For semi-coroutines, the starter is often the last (only) resumer, so it seems coroutine main implicitly suspends on termination.

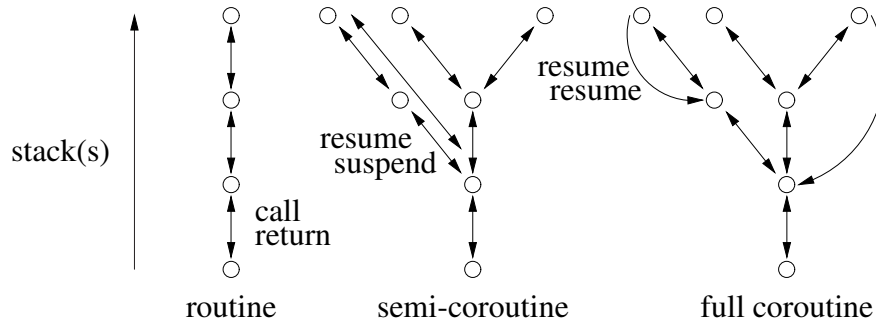


- **dashed red** ⇒ create stack and resume coroutine main (cocall)
- **solid red** ⇒ resume coroutine at last suspend
- **solid blue** ⇒ resume last resumer
- **dashed blue** ⇒ resume *starter*



### 3.10 Full Coroutines

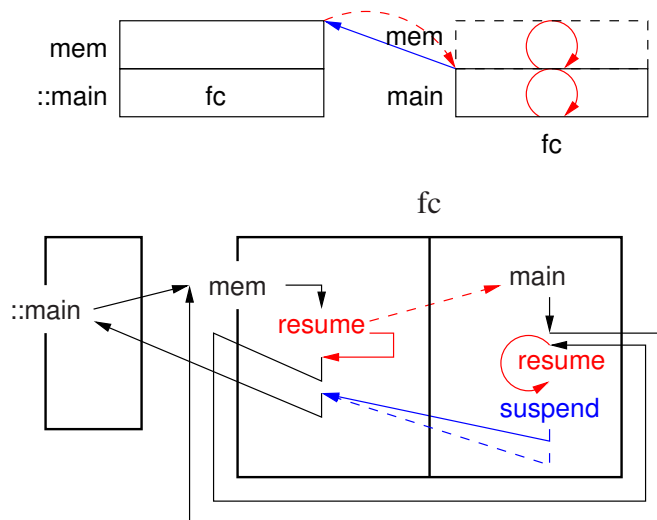
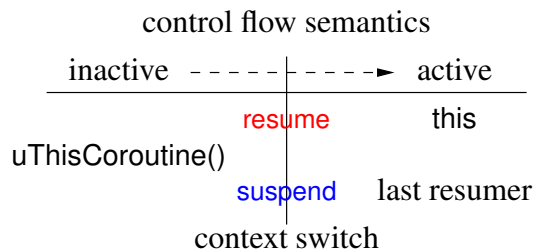
- **Semi-coroutine** activates the member routine that activated it.
- **Full coroutine** has a resume cycle; semi-coroutine does not form a resume cycle.



- A semi-coroutine is allowed to perform call/return operations because it subsumes the notion of a routine.
- A full coroutine is allowed to perform semi-coroutine operations because it subsumes the notion of semi-coroutine.

```

_Coroutine Fc {
    void main() { // starter ::main
        mem(); // ?
        resume(); // ?
        suspend(); // ?
    }
public:
    void mem() { resume(); }
};
int main() {
    Fc fc;
    fc.mem();
}
    
```



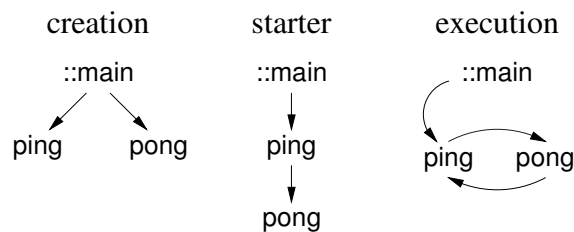
- Suspend inactivates the current active coroutine (`uThisCoroutine`), and activates last resumer.
- Resume inactivates the current active coroutine (`uThisCoroutine`), and activates the current object (**this**).
- Hence, the current object *must* be a non-terminated coroutine.
- Note, **this** and `uThisCoroutine` change at different times.
- Exception: last resumer not changed when resuming self because no practical value.
- Full coroutines can form an arbitrary topology with an arbitrary number of coroutines.
- There are 3 phases to any full coroutine program.
  1. starting the cycle
  2. executing the cycle
  3. stopping the cycle (return to the program main)
- Starting the cycle requires each coroutine to know at least one other coroutine.
- The problem is mutually recursive references.
 

```
Fc x(y), y(x); // does not compile, why?
```
- One solution is to make closing the cycle a special case.
 

```
Fc x, y(x);
x.partner( y );
```
- Once the cycle is created, execution around the cycle can begin.
- Stopping can be as complex as starting, *because a coroutine goes back to its starter*.
- For full-coroutines, the starter is often *not* the last resumer, so coroutine main does not appear to implicitly suspend on termination.
- But it is necessary to activate the program main to finish (unless `exit` is used).
- The starter stack always gets back to the program main.
- Again, it is unnecessary to terminate all coroutines, just delete them.

## 3.10.1 Ping/Pong

- Full-coroutine control-flow with 2 identical coroutines:

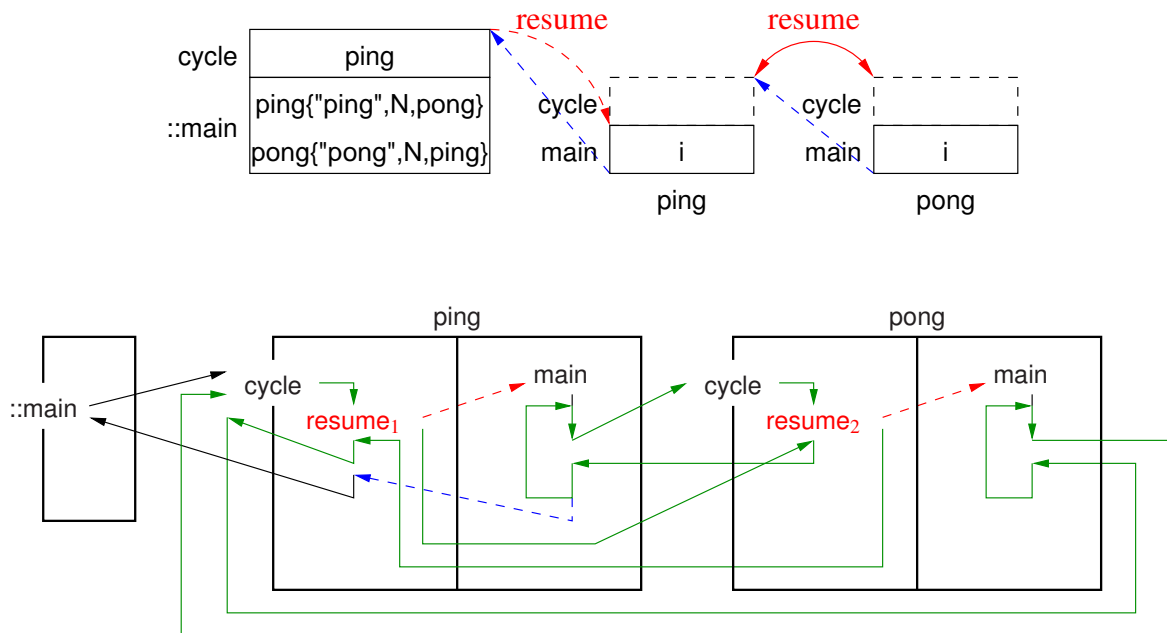


```

_Coroutine PingPong {
  const char * name;
  const unsigned int N;
  PingPong * part;
  void main() { // ping' s starter ::main, pong' s starter ping
    for ( unsigned int i = 0; i < N; i += 1 ) {
      cout << name << endl;
      part->cycle();
    }
  }
public:
  PingPong( const char * name, unsigned int N, PingPong & part )
    : name( name ), N( N ), part( &part ) {}
  PingPong( const char * name, unsigned int N ) : name( name ), N( N ) {}
  void partner( PingPong & part ) { PingPong::part = &part; }
  void cycle() { resume(); }
};
int main() {
  enum { N = 20 };
  PingPong ping( "ping", N ), pong( "pong", N, ping );
  ping.partner( pong );
  ping.cycle();
}
  
```

- ping created without partner; pong created with partner.
- ping makes pong partner, closing cycle.
- Why is PingPong::part a pointer rather than reference?
- cycle resumes ping  $\Rightarrow$  ::main is ping's starter
- ping calls pong's cycle member, resuming pong so ping is pong's starter.
- pong calls ping's cycle member, **resuming ping in pong's cycle member**.
- Each coroutine cycles N times, **becoming inactive in the other's cycle member**.
  - ping ends first, because it started first, resuming its starter ::main in ping's cycle member.

- `::main` terminates with terminated coroutine `ping` and unterminated coroutine `pong`.
- Assume `ping`'s declaration is changed to `ping( "ping", N + 1 )`.
  - `pong` ends first, because it cycles less, resuming its starter `ping` in `pong`'s cycle member.
  - `ping` ends second, resuming its starter `::main` in `ping`'s cycle member.
  - `::main` terminates with terminated coroutines `ping` and `pong`.



### 3.10.2 Producer-Consumer

- Full-coroutine control-flow and bidirectional communication with 2 non-identical coroutines:

```

_Coroutine Prod {
  Cons * c;
  int N, money, receipt;
  void main() { // starter ::main
    // 1st resume starts here
    for ( int i = 0; i < N; i += 1 ) {
      int p1 = rand() % 100; // products
      int p2 = rand() % 100;
      cout << "prod " << p1
            << " " << p2 << endl;
      int status = c->delivery(p1, p2);
      cout << "prod rec $" << money
            << " stat " << status << endl;
      receipt += 1;
    }
    c->stop();
    cout << "prod stops" << endl;
  }
}

```

```

public:
  int payment( int money ) {
    Prod::money = money;
    resume(); // activate prod in
    return receipt; // Cons::delivery
  }
  void start( int N, Cons & c ) {
    Prod::N = N; Prod::c = &c;
    receipt = 0;
    resume(); // activate Prod::main
  }
};

```

```

_Coroutine Cons {
  Prod & p;
  int p1, p2, status;
  bool done;
  void main() { // starter prod
    // 1st resume starts here
    int money = 1, receipt;
    for ( ; ! done; ) {
      cout << "cons " << p1 << " "
        << p2 << " pay $"
        << money << endl;
      status += 1;
      receipt = p.payment(money);
      cout << "cons #"
        << receipt << endl;
      money += 1;
    }
    cout << "cons stops" << endl;
  }
}

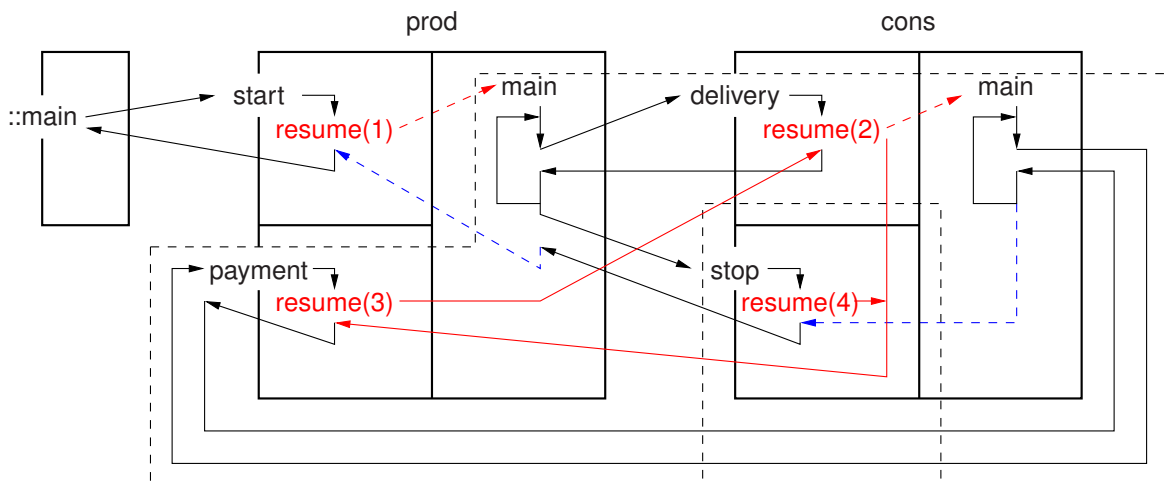
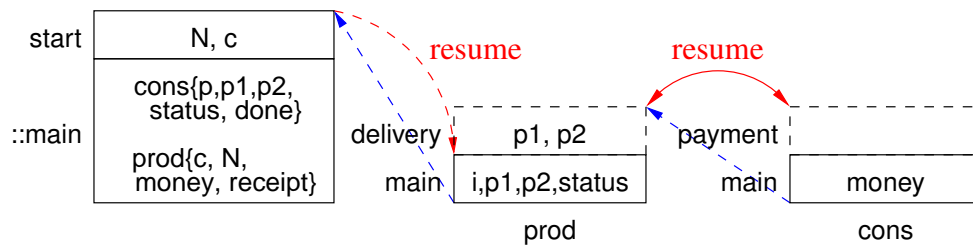
```

```

public:
  Cons(Prod & p) : p(p), status(0), done(false) {}
  int delivery( int p1, int p2 ) {
    Cons::p1 = p1; Cons::p2 = p2;
    resume(); // Cons::main 1st time, then
    return status; // cons in Prod::payment
  }
  void stop() {
    done = true;
    resume(); // cons in Prod::payment
  }
};
int main() {
  Prod prod;
  Cons cons( prod );
  prod.start( 5, cons );
}

```

- Cheat using forward reference for Cons at c->delivery and c->stop. Fix by?



- Black dashed-line same control flow as ping/pong.
- Remove flag variable from full-coroutine producer-consumer.

```

_Exception Stop {};
_Coroutine Prod {
    Cons * c;
    int N, money, receipt;
    void main() {
        for ( int i = 0; i < N; i += 1 ) {
            int p1 = rand() % 100;
            int p2 = rand() % 100;
            cout << "prod " << ...
            int status = c->delivery(p1, p2);
            cout << "prod rec $" << ...
            receipt += 1;
        }
        _Resume Stop() _At *c;
        suspend(); // restart cons
        cout << "prod stops" << endl;
    }
public:
    int payment( int money ) {
        Prod::money = money;
        resume();
        return receipt;
    }
    void start( int N, Cons & c ) {
        Prod::N = N; Prod::c = &c;
        receipt = 0;
        resume();
    }
};

```

```

_Coroutine Cons {
    Prod & p;
    int p1, p2, status = 0;
    void main() {
        int money = 1, receipt;
        try {
            for ( ;; ) {
                cout << "cons " << p1 << ...
                status += 1;
                receipt = p.payment( money );
                cout << "cons #" << ...
                _Enable; // trigger exception
                money += 1;
            }
        } catch( Stop & ) {}
        cout << "cons stops" << endl;
    }
public:
    Cons( Prod & p ) : p( p ) {}
    int delivery( int p1, int p2 ) {
        Cons::p1 = p1; Cons::p2 = p2;
        resume();
        return status;
    }
};

```

### 3.11 Coroutine Languages

- Coroutine implementations have two forms:
  1. stackless: use the caller's stack and a fixed-sized local-state
  2. stackful: separate stack and a fixed-sized (class) local-state
- Stackless coroutines cannot call other routines and then suspend, i.e., only suspend in the coroutine main.
- Generators/iterators are often simple enough to be stackless using yield.
- Simula, CLU, C#, Ruby, Python, JavaScript, Lua, F# all support yield constructs.

#### 3.11.1 Python 3.5

- Stackless, semi coroutines, routine versus class, no calls, single interface
- Fibonacci (see Section 3.1.1.4, p. 28)

```

def Fibonacci( n ):
    fn = 0; fn1 = fn
    yield fn
    fn = 1; fn2 = fn1; fn1 = fn
    yield fn
    # while True:
    for i in range( n - 2 ):
        fn = fn1 + fn2; fn2 = fn1; fn1 = fn
        yield fn

f1 = Fibonacci( 10 )
f2 = Fibonacci( 10 )
for i in range( 10 ):
    print( next( f1 ), next( f2 ) )
for fib in Fibonacci( 15 ):
    print( fib )

```

*# coroutine main*  
*# suspend*  
*# suspend*  
*# for infinite generator*  
*# suspend*  
*# objects*  
*# resume*  
*# use generator as iterator*

- Format (see Section 3.1.2.4, p. 32)

```

def Format():
    try:
        while True:
            for g in range( 5 ):
                for b in range( 4 ):
                    print( (yield), end=' ' ) # receive from send
                    print( ' ', end=' ' ) # block separator
                print() # group separator
            except GeneratorExit:
                if g != 0 | b != 0:
                    print()

fmt = Format()
next( fmt )
for i in range( 41 ):
    fmt.send( ' a' )

```

*# groups of 5 blocks*  
*# blocks of 4 characters*  
*# receive from send*  
*# block separator*  
*# group separator*  
*# destructor*  
*# special case*  
*# prime generator*  
*# send to yield*

- send takes only one argument, and no cycles ⇒ no full coroutine

### 3.11.2 JavaScript

- Similar to Python: stackless, semi coroutines, routine versus class, no calls, single interface
- Embedded in HTML with I/O from web browser.
- Fibonacci (see Section 3.1.1.4, p. 28)

```

<!DOCTYPE html><html>
<head><meta charset="utf-8" /><title>Fibonacci Coroutine</title></head>
<body><button id="button">Click for next Fibonacci number!</button>
    <p id="output"></p></body>
<script>

```

```

function * Fibonacci() {
  var fn = 0, fn1 = 0, fn2 = 0;    // JS bug: initialize vars or lost on suspend
  yield fn;                       // return fn to resumer
  fn = 1; fn2 = fn1; fn1 = fn;
  yield fn;                       // return fn to resumer
  for ( ;; ) {
    fn = fn1 + fn2; fn2 = fn1; fn1 = fn;
    yield fn;                     // return fn to resumer
  }
}

const button = document.getElementById( ' button' );
const output = document.getElementById( ' output' );
var count = 0, suffix;
var fib = Fibonacci();
button.addEventListener( "click", event => {
  if (count % 10 == 1) suffix = "st";
  else if (count % 10 == 2) suffix = "nd";
  else suffix = "th";
  output.textContent = count + suffix + " Fibonacci: " + fib.next().value;
  count += 1;
});
</script></body></html>

```

- Format (see Section 3.1.2.4, p. 32)

```

<!DOCTYPE html><html>
<head><meta charset="utf-8" /><title>Format Coroutine</title></head>
<body><input placeholder="Type characters!" size=50><p id="output"></p></body>
<script>

function * Format() {
  var g = 0, b = 0, ch = '';    // JS bug: initialize vars or lost on suspend
  for ( ;; ) {
    for ( g = 0; g < 5; g += 1 ) {
      for ( b = 0; b < 4; b += 1 ) {
        ch = yield;
        output.innerHTML += ch; // console.log adds \n
      }
      output.innerHTML += " ";
    }
    output.innerHTML += "<br>";
  }
}

const inputBox = document.querySelector( ' input' );
const output = document.getElementById( ' output' );
var format = Format();
format.next();                // prime generator
inputBox.addEventListener( ' keypress' , event => {
  format.next( event.key );
});
</script></body></html>

```



### 3.11.3 C++20 Coroutines

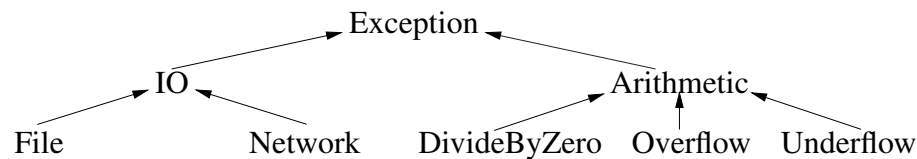
- C++20 has an API for coroutines and outline code to build stackless, stackful, or even fibres (tasks without preemption).
- This capability cannot be used directly. It requires writing significant low-level implementation code.



## 4 More Exceptions

### 4.1 Derived Exception-Type

- **derived exception-types** is a mechanism for inheritance of exception types, like inheritance of classes.
- Provides a kind of polymorphism among exception types:



- Provides ability to handle an exception at different degrees of specificity along the hierarchy.
- Possible to catch a more general exception-type in higher-level code where the implementation details are unknown.
- Higher-level code should catch general exception-types to reduce tight coupling to the specific implementation.
  - tight coupling may force unnecessary changes in the higher-level code when low-level code changes.
- Exception-type inheritance allows a handler to match multiple exceptions, e.g., a base handler can catch both base and derived exception-type.
- To handle this case, most propagation mechanisms perform a linear search of the handlers for a guarded block and select the first matching handler.

```
try { ...  
} catch( Arithmetic & ) { ...  
} catch( Overflow ) { ... // never selected!!!  
}
```

- When subclassing, it is best to catch an exception by reference:

```
struct B {};  
struct D : public B {};  
try {  
    throw D(); // _Throw in uC++  
} catch( B e ) { // truncation  
    // cannot down-cast  
}  
  
try {  
    throw D(); // _Throw in uC++  
} catch( B & e ) { // no truncation  
    ... dynamic_cast<D>(e) ...  
}
```

- Otherwise, exception is truncated from its dynamic type to static type specified at the handler, and cannot be down-cast to the dynamic type.
- Notice, catching truncation (see page 59) is different from raising truncation, which does not occur in  $\mu\text{C++}$  with **\_Throw**.

## 4.2 Catch-Any

- **catch-any** is a mechanism to match any exception propagating through a guarded block.
- With exception-type inheritance, catch-any can be provided by the root exception-type, e.g., `catch( Exception )` in Java.
- Otherwise, special syntax is needed, e.g., `catch( ... )` in C++.
- For termination, catch-any is used as a general cleanup when a non-specific exception occurs.
- For resumption, this capability allows a guarded block to gather or generate information about control flow (e.g., logging).

```

try {
    ...
} _CatchResume( ... ) { // catch-any
    ... // logging
    _Resume; // reresume for fixup
} catch( ... ) { // catch-any
    ... // cleanup
    _Throw; // rethrow for recovery
}

```

- Java finalization:

```

try { ...
} catch( E ) { ... }
... // other catch clauses
} finally { // always executed
    ... // cleanup
    // possibly rethrow
}

```

provides catch-any capabilities and handles the non-exceptional case.

- difficult to mimic in C++, even with RAII, because of local variables.

## 4.3 Exception Parameters

- **Exception parameters** allow passing information from the raise to a handler.
- Inform a handler about details of the exception, and to modify the raise site to fix an exceptional situation.
- Different EHMs provide different ways to pass parameters.
- In C++/Java, parameters are defined inside the exception:

```

struct E {
    int i;
    E( int i ) : i(i) {}
};
void f( ... ) { ... throw E( 3 ); ... } // argument
int main() {
    try {
        f( ... );
    } catch( E p ) { // parameter, value or reference
        ... p.i ...
    }
}

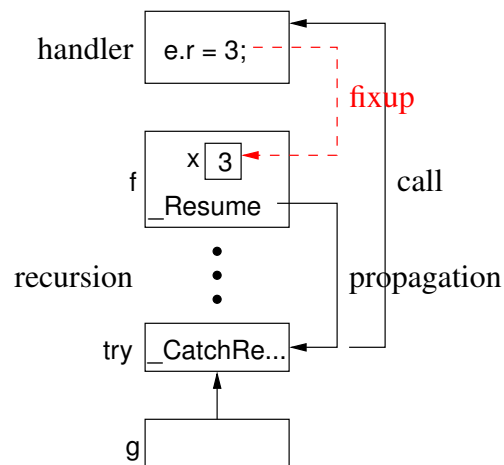
```

- For resumption, values at raise modified via reference/pointer in caught exception:

```

_Exception E {
public:
    int & r;
    E( int & r ) : r( r ) {}
};
void f() {
    int x;
    ... _Resume E( x ); ...
}
void g() {
    try {
        f();
    } _CatchResume( E & e ) {
        ... e.r = 3; ...
    }
}

```



## 4.4 Exception List

- Missing exception handler for arithmetic overflow in control software caused [Ariane 5 rocket](#) to self-destruct (\$370 million loss).
- **exception list** is part of a routine's prototype specifying which exception types may propagate from the routine to its caller.

```
int g() throw(E) { ... throw E(); }
```

- This capability allows:
  - static detection of a raised exception not handled locally or by its caller
  - runtime detection where the exception may be converted into a special **failure exception** or the program terminated.
- 2 kinds of checking:
  - checked/unchecked exception-type (Java, inheritance based, static check)

- checked/unchecked routines (C++, exception-list based, dynamic check) (deprecated C++11, replaced with **noexcept**)
- While checked exception-types are useful for software engineering, reuse is precluded.
- E.g., consider the simplified C++ template routine `sort`:

```
template<class T> void sort( T items[] ) throw( ?, ?, ... ) {
    // using bool operator<( const T &a, const T &b );
```

using the operator routine `<` in its definition.

- Impossible to know all exception types that propagated from routine `<` for every type.
- Since only a fixed set of exception types can appear in `sort`'s exception list, some sortable types are precluded.
- Exception lists can preclude reuse for arguments of routine pointers (functional style) and/or polymorphic methods/routines (OO style):

```
struct E {};
// throw NO exceptions
void f( void (*p)() noexcept ) {
    p();
}
void g() noexcept(false) { throw E(); }
void h() {
    try { ... f( g ); ... }
    catch( E ) {}
}

struct B { // throw NO exceptions
    virtual void g() noexcept {}
    void f() { g(); }
};
struct D : public B {
    void g() noexcept(false) { throw E(); }
    void h() {
        try { ... f(); ... }
        catch( E ) {}
    }
};
```

- Left example, routine `h` has an appropriate **try** block and passes the version of `g` to `f` that raises exception-type `E`.
- However, checked exception-types preclude this case because the signature of argument `g` is less restrictive than parameter `p` of `f`.
- Right example, member routine `D::h` calls `B::f`, which calls `D::g` that raises exception-type `E`.
- However, checked exception types preclude this case because the signature of `D::g` is less restrictive than `B::g`.
- Finally, determining an exception list for a routine can become impossible for concurrent exceptions because they can propagate at any time.

## 4.5 Destructor

- Destructor is implicitly **noexcept** ⇒ *cannot* raise an exception.

- Destructor **can** raise an exception, if marked **noexcept(false)**, or inherits from class with **noexcept(false)** destructor.

<pre> <b>struct</b> E {}; <b>struct</b> C {     ~C() <b>noexcept(false)</b> { <b>throw</b> E(); } }; <b>try</b> {           // outer try     C x;         // raise on deallocation     <b>try</b> {       // inner try         C y;    // raise on deallocation     } <b>catch</b>( E ) {...} // inner handler } <b>catch</b>( E ) {...} // outer handler </pre>	<table border="0"> <tr> <td style="padding-right: 10px;">y's destructor</td> <td style="padding-right: 20px;"> </td> <td><b>throw</b> E</td> <td style="padding-right: 20px;"> </td> <td>x's destructor</td> </tr> <tr> <td></td> <td></td> <td><b>inner try</b></td> <td></td> <td><b>outer try</b></td> </tr> <tr> <td></td> <td></td> <td>  y</td> <td></td> <td>  x</td> </tr> </table>	y's destructor		<b>throw</b> E		x's destructor			<b>inner try</b>		<b>outer try</b>			y		x
y's destructor		<b>throw</b> E		x's destructor												
		<b>inner try</b>		<b>outer try</b>												
		y		x												

- y's destructor called at end of inner **try** block, it raises an exception E, which unwinds destructor and **try**, and handled at inner **catch**
- x's destructor called at end of outer **try** block, it raises an exception E, which unwinds destructor and **try**, and handled at outer **catch**

## 4.6 Multiple Exceptions

- An exception handler can generate an arbitrary number of nested exceptions.

<pre> <b>struct</b> E {}; <b>int</b> cnt = 3; <b>void</b> f( <b>int</b> i ) {     <b>if</b> ( i == 0 ) <b>throw</b> E();     <b>try</b> {         f( i - 1 );     } <b>catch</b>( E ) { // handler h         cnt -= 1;         <b>if</b> ( cnt &gt; 0 ) f( 2 );         ... <b>throw</b>; ...     } } <b>int</b> main() { f( 2 ); } </pre>	<table border="0"> <tr><td>h</td><td>✓</td></tr> <tr><td>f</td><td></td></tr> <tr><td>f</td><td></td></tr> <tr><td>h</td><td>✓ throw E<sub>2</sub></td></tr> <tr><td>f</td><td></td></tr> <tr><td>f</td><td></td></tr> <tr><td>h</td><td>✓ throw E<sub>1</sub></td></tr> <tr><td>f</td><td></td></tr> <tr><td>f</td><td></td></tr> </table>	h	✓	f		f		h	✓ throw E <sub>2</sub>	f		f		h	✓ throw E <sub>1</sub>	f		f	
h	✓																		
f																			
f																			
h	✓ throw E <sub>2</sub>																		
f																			
f																			
h	✓ throw E <sub>1</sub>																		
f																			
f																			

- Exceptions are nested as handler can rethrow its matched exception when control returned.
- However, multiple exceptions cannot propagate simultaneously.
  - Cannot start second exception without handler to deal with first exception, i.e., cannot drop exception and start another.
  - Cannot postpone first exception because second exception may remove its handlers during stack unwinding.
- Only destructor code can intervene during propagation.
- Hence, a destructor **cannot** raise an exception during propagation; it can only start propagation.

```
try {  
    C x;           // raise on deallocation  
    throw E();  
} catch( E ) {...}
```

- Raise of E causes unwind of inner **try** block.
- x's destructor called during unwind, it raises an exception E, which one should be used?
- Check if exception is being propagated with `uncaught_exceptions()`.



## 5 Concurrency

- A **thread** is an independent sequential execution path through a program.
  - Each thread is scheduled for execution separately and independently from other threads.
- A **process** is a program component (like a routine) that **has its own thread** and has the same state information as a coroutine.
- A **task** is a program component that **has its own thread** but is
  - reduced along some particular dimension (like the difference between a boat and a ship, one is physically smaller than the other).
  - It is often the case that a process has its own memory, while tasks share a common memory.
  - A task is sometimes called a light-weight process (LWP).
- **Parallel execution** is when 2 or more operations occur simultaneously, which can only occur when multiple processors (CPUs) are present.
- **Concurrent execution** is any situation in which execution of multiple threads *appears* to be performed in parallel.
  - It is the threads of control associated with processes and tasks that result in concurrent execution, **not the processors**.

### 5.1 Why Write Concurrent Programs

- Dividing a problem into multiple executing threads is an important programming technique just like dividing a problem into multiple routines.
- Expressing a problem with multiple executing threads may be the natural (best) way of describing it.
- Multiple executing threads can enhance execution-time efficiency by taking advantage of inherent concurrency in an algorithm and any parallelism available in the computer system.

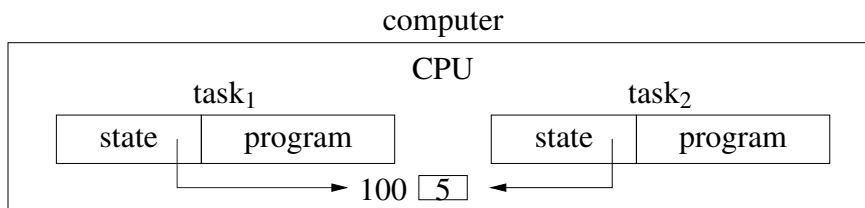
### 5.2 Why Concurrency is Difficult

- to understand:
  - While people can do several things concurrently, the number is small because of the difficulty in managing and coordinating them.
  - Especially when the things interact with one another.
- to specify:
  - How can/should a problem be broken up so that parts of it can be solved at the same time as other parts?

- How and when do these parts interact or are they independent?
- If interaction is necessary, what information must be communicated during the interaction?
- to debug:
  - Concurrent operations proceed at varying speeds and in non-deterministic order, hence execution is not repeatable (Heisenbug).
  - Reasoning about multiple streams or threads of execution and their interactions is much more complex than for a single thread.
- E.g. Moving furniture out of a room; can't do it alone, but how many helpers and how to do it quickly to minimize the cost?
- How many helpers?
  - 1,2,3, ... N, where N is the number of items of furniture
  - more than N?
- Where are the bottlenecks?
  - the door out of the room, items in front of other items, large items
- What communication is necessary between the helpers?
  - which item to take next
  - some are fragile and need special care
  - big items need several helpers working together

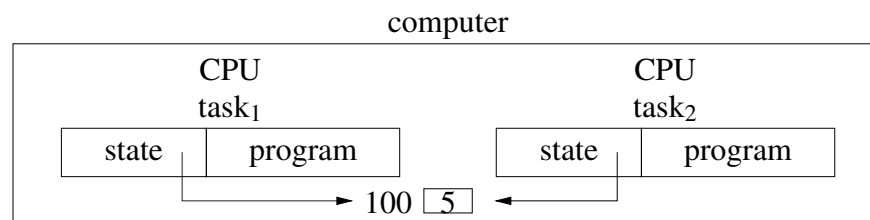
### 5.3 Concurrent Hardware

- Concurrent execution of threads is possible with only one CPU (**uniprocessor**); **multitasking** for multiple tasks or **multiprocessing** for multiple processes.

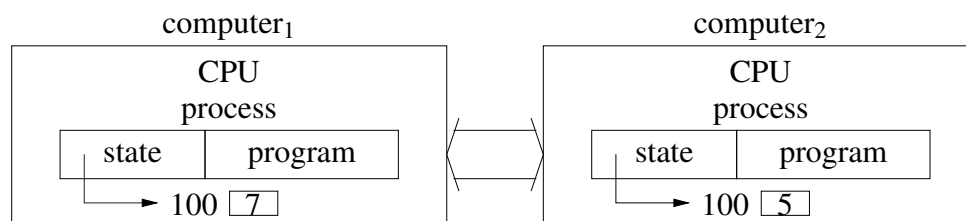


- Parallelism is simulated by context switching the threads on the CPU.
- Most of the issues in concurrency can be illustrated without parallelism.
- Pointers among tasks work because memory is shared.
- **Unlike coroutines, task switching may occur at non-deterministic program locations, i.e., between any two machine instructions.**

- Introduces all the difficulties in concurrent programs.
    - \* programs must be written to work regardless of non-deterministic ordering of program execution.
  - Switching happens *explicitly* but conditionally when calling routines.
    - \* routine may or may not context switch depending on hidden (internal) state (cannot predict)
  - Switching can happen *implicitly* because of an external **interrupt** independent of program execution.
    - \* e.g., page fault, I/O, or timer interrupt
    - \* timer interrupts divide execution (between instructions) into discrete time-slices occurring at non-deterministic time intervals
    - \* ⇒ **task execution is not continuous**
  - If interrupts affect **scheduling** (execution order), it is called **preemptive**, otherwise the scheduling is **non-preemptive**.
  - Programmer cannot predict execution order, unlike coroutines.
  - Granularity of context-switch is instruction level for preemptive (harder to reason) and routine level for non-preemptive.
- In fact, every computer has multiple CPUs: main CPU(s), bus CPU, graphics CPU, disk CPU, network CPU, etc.
  - Concurrent/parallel execution of threads is possible with multiple CPUs sharing memory (**multiprocessor**):



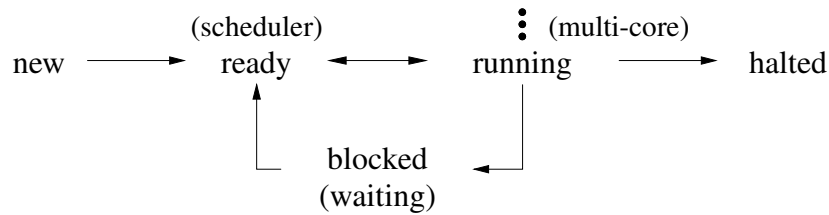
- Pointers among tasks work because memory is shared.
- Concurrent/parallel execution of threads is possible with single/multiple CPUs on different computers with *separate memories* (**distributed system**):



- Pointers among tasks do NOT work because memory is not shared.

## 5.4 Execution States

- A thread may go through the following states during its execution.



- **State transitions** are initiated in response to events (e.g., interrupts):
  - entering the system (new → ready)
  - assigning thread to computing resource, e.g., CPU (ready → running)
  - timer alarm for preemption (running → ready)
  - long-term delay versus spinning (running → blocked)
  - completion of delay, e.g., network or I/O completion (blocked → ready)
  - normal completion or error, e.g., segment fault (running → halted)
- Thread cannot bypass the “ready” state during a transition so the scheduler maintains complete control of the system.
- Non-deterministic “ready ↔ running” transition ⇒ basic operations unsafe:

```

int i = 0;    // shared
task0        task1
i += 1       i += 1
  
```

- If increment implemented with single **inc i** instruction, transitions can only occur before or after instruction, not during.
- If increment is replaced by a load-store sequence, transitions can occur during sequence.

```

ld  r1,i      // load into register 1 the value of i
...          // PREEMPTION
add r1,#1    // add 1 to register 1
...          // PREEMPTION
st  r1,i     // store register 1 into i
  
```

- If both tasks increment 10 times, the expected result is 20.
- True for single instruction, false for load-store sequence.
- Many failure cases for load-store sequence where *i* does not reach 20.

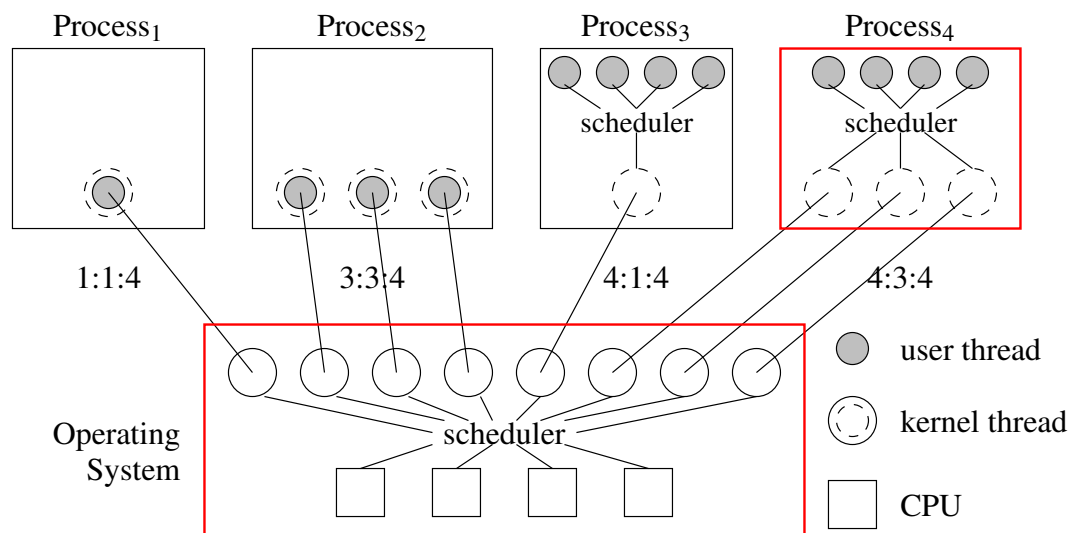
- Remember, context switch saves and restores registers for each coroutine/task.

task0	task1
<b>1st iteration</b>	
ld r1,i (r1 <- 0)	
add r1,#1 (r1 <- 1)	
	<b>1st iteration</b>
	ld r1,i (r1 <- 0)
	add r1,#1 (r1 <- 1)
	st r1,i (i <- 1)
	<b>2nd iteration</b>
	ld r1,i (r1 <- 1)
	add r1,#1 (r1 <- 2)
	st r1,i (i <- 2)
	<b>3rd iteration</b>
	ld r1,i (r1 <- 2)
	add r1,#1 (r1 <- 3)
	st r1,i (i <- 3)
<b>1st iteration</b>	
st r1,i (i <- 1)	

- The 3 iterations of **task1** are lost when overwritten by **task0**.
- Hence, sequential operations, however small (increment), are unsafe in a concurrent program.

## 5.5 Threading Model

- For multiprocessor systems, a **threading model** defines relationship between threads and CPUs.
- OS manages CPUs providing logical access via **kernel threads (virtual processors)** *scheduled* across the CPUs.



- More kernel threads than CPUs to provide multiprocessing, i.e., run multiple programs simultaneously.
- A process may have multiple kernel threads to provide parallelism if multiple CPUs.
- A program may have user threads scheduled on its process's kernel threads.
- User threads are a low-cost structuring mechanism, like routines, objects, coroutines (versus high-cost kernel thread).
- Relationship is denoted by user:kernel:CPU, where:
  - 1:1:C (kernel threading) – 1 user thread maps to 1 kernel thread
  - N:N:C (generalize kernel threading) –  $N \times 1:1$  kernel threads (Java/Pthreads/C++)
  - M:1:C (user threading) – M user threads map to 1 kernel thread (no parallelism)
  - M:N:C (user threading) – M user threads map to N kernel threads (Go,  $\mu$ C++)
- Often the CPU number (C) is omitted.
- Can recursively add **nano threads** (stackless) on top of user threads (stackful), and **virtual machine** below OS.

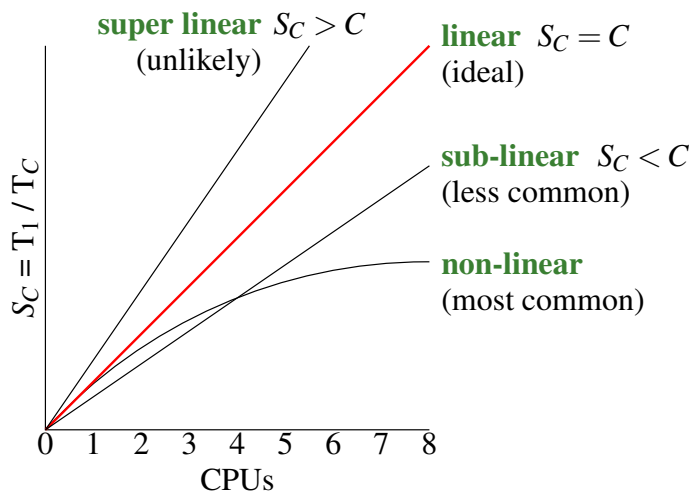
## 5.6 Concurrent Systems

- Concurrent systems can be divided into 3 major types:
  1. those that attempt to **discover implicit** concurrency in an otherwise sequential program, e.g., parallelizing loops and access to data structures
  2. those that provide concurrency through **implicit** constructs, which a programmer uses to build a concurrent program
  3. those that provide concurrency through **explicit** constructs, which a programmer uses to build a concurrent program
- In type 1, there is a fundamental limit to how much concurrency can be found and current techniques only work on a certain class of problems.
- In type 2, concurrency is accessed indirectly via specialized mechanisms (e.g., pragmas or parallel **for**) and threads are implicitly managed.
- In type 3, concurrency is accessed directly and threads explicitly managed.
- Types 1 & 2 are always built from type 3.
- **To solve all concurrency problems, threads need to be explicit.**
- Both implicit and explicit mechanisms are complementary, and hence, can appear together in a single programming language.

- However, the limitations of implicit mechanisms require that explicit mechanisms always be available to achieve maximum concurrency.
- Some concurrent systems provide a single technique or paradigm that must be used to solve all concurrent problems.
- While a particular paradigm may be very good for solving certain kinds of problems, it may be awkward or preclude other kinds of solutions.
- Therefore, a good concurrent system must support a variety of different concurrent approaches, while at the same time not requiring the programmer to work at too low a level.
- In all cases, as concurrency increases, so does the complexity to express and manage it.

## 5.7 Speedup

- Program **speedup** is  $S_C = T_1/T_C$ , where  $C$  is number of CPUs and  $T_1$  is sequential execution.
- E.g., 1 CPU takes 10 seconds,  $T_1 = 10$  (**user time**), 4 CPUs takes 2.5 seconds,  $T_4 = 2.5$  (**real time**)  $\Rightarrow S_4 = 10/2.5 = 4$  times speedup (linear).



- Aspects affecting speedup (assume sufficient parallelism for concurrency):
  1. amount of concurrency
  2. critical path among concurrency
  3. scheduler efficiency
- An algorithm/program is composed of sequential and concurrent sections.
- E.g., sequentially read matrix, concurrently subtotal rows, sequentially total subtotals.
- **Amdahl's law** (Gene Amdahl): concurrent section of program is  $P$  making sequential section  $1 - P$ , then maximum speedup using  $C$  CPUs is:

$$S_C = \frac{1}{(1-P) + P/C} \text{ where } T_1 = 1, T_C = \text{sequential} + \text{concurrent}$$

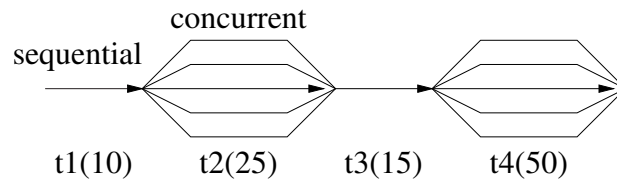
- Normalize:  $T_1 = 10/10 = 1$ ,  $T_4 = 2.5/10 = .25$ .

$$S_4 = \frac{1}{(1-1) + 1 \times .25} = 4 \text{ times}, P = 1 \Rightarrow (100\%) \text{ of } T_4 \text{ is concurrent}$$

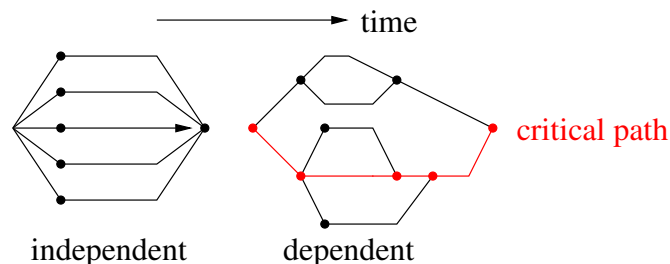
- Change  $P = .8(80\%)$  so  $T_4 = .8 \times .25 = .2$  is concurrent and  $1 - .8 = .2(20\%)$  is sequential.

$$S_4 = \frac{1}{(1-.8) + .8 \times .25} = \frac{1}{.2 + .2} = 2.5 \text{ times}, \text{ because of sequential code}$$

- As  $C$  goes to infinity,  $P/C$  goes to 0, so maximum speedup is  $1/(1-P)$ , i.e., time for sequential section.
- Speedup falls rapidly as sequential section  $(1-P)$  increases.
- E.g., sequential section =  $.2(20\%)$ ,  $S_C = 1/(1-.8) \Rightarrow$  max speedup 5.
- **Concurrent programming consists of minimizing sequential section  $(1-P)$ .**
- E.g., program has 4 stages:  $t_1 = 10$ ,  $t_2 = 25$ ,  $t_3 = 15$ ,  $t_4 = 50$  (time units)



- $T_C = 10 + 25 + 15 + 50 = 100$  (time units)
- Concurrently speedup sections  $t_2$  by 5 times and  $t_4$  by 10 times.
- $T_C = 10 + 25 / 5 + 15 + 50 / 10 = 35$  (time units)  
Speedup =  $100 / 35 = 2.86$  times
- Large reductions for  $t_2$  and  $t_4$  have only minor effect on speedup.
- Formula does not consider any increasing costs for the concurrency, i.e., administrative costs, so results are optimistic.
- While sequential sections bound speedup, concurrent sections bound speedup by the **critical path** of computation.





- **independent execution** : all threads created together and do not interact.
- **dependent execution** : threads created at different times and interact.
- Longest path bounds speedup (even for independent execution).
- Finally, speedup can be affected by scheduler efficiency/ordering (often no control), e.g.:
  - greedy scheduling : run a thread as long as possible before context switching (not very concurrent).
  - LIFO scheduling : give priority to newly waiting tasks (starvation).
- Therefore, it is difficult to achieve significant speedup for many algorithms/programs.
- In general, benefit comes when many programs achieve some speedup so there is an overall improvement on a multiprocessor computer.

## 5.8 Thread Creation

- Concurrency requires 3 mechanisms in a programming language.
  1. creation – cause another thread of control to come into existence.
  2. synchronization – establish timing relationships among threads, e.g., same time, same rate, happens before/after.
  3. communication – transmit data among threads.
- Thread creation must be a primitive operation; cannot be built from other operations in a language.
- ⇒ need new construct to create a thread and define where the thread starts execution.

### 5.8.1 COBEGIN/COEND

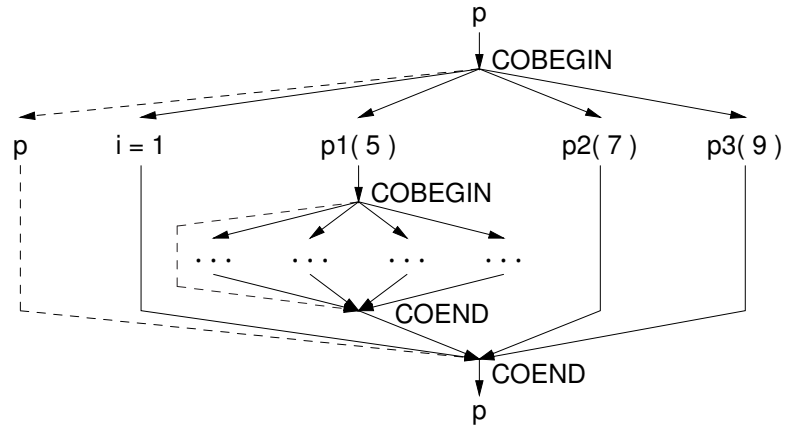
- Compound statement with statements run by multiple threads.

```

#include <uCobegin.h>
int i;
void p1(...); void p2(...); void p3(...);
// initial thread creates threads
COBEGIN // threads execute statement in block
    BEGIN i = 1; ... END
    BEGIN p1( 5 ); ... END // order and speed of internal
    BEGIN p2( 7 ); ... END // thread execution is unknown
    BEGIN p3( 9 ); ... END
COEND // initial thread waits for all internal threads to
// finish (synchronize) before control continues

```

- Implicit or explicit concurrency?
- A **thread graph** represents thread creations:



- Restricted to creating trees (lattice) of threads.
- Use recursion to create dynamic number of threads.

```

void loop( int N ) {
    if ( N != 0 ) {
        COBEGIN
            BEGIN p1( ... ); END
            BEGIN loop( N - 1 ); END // recursive call
        COEND // wait for return of recursive call
    }
}
cin >> N;
loop( N );

```

- What does the thread graph look like?

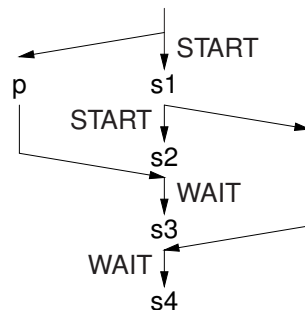
### 5.8.2 START/WAIT

- Start thread in routine and wait (join) at thread termination, allowing arbitrary thread graph:

```

#include <uCobegin.h>
int i;
void p( int i ) {...}
int f( int i ) {...}
auto tp = START( p, 5 );
s1 // continue execution, do not wait for p
auto tf = START( f, 8 );
s2 // continue execution, do not wait for f
WAIT( tp ); // wait for p to finish
s3
i = WAIT( tf ); // wait for f to finish
s4

```



- Allows same routine to be started multiple times with different arguments.
- Implicit or explicit concurrency?

- COBEGIN/COEND can only approximate this thread graph:

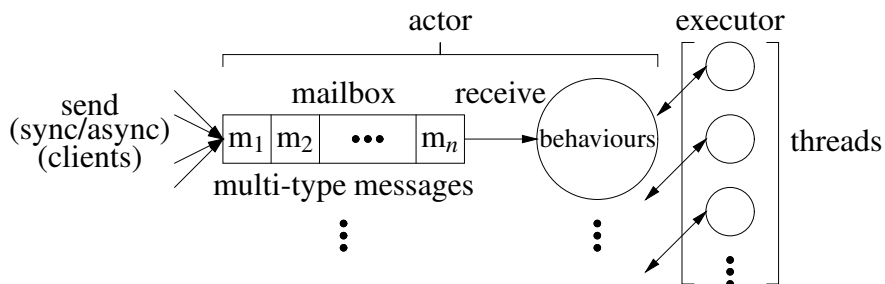
```
COBEGIN
  BEGIN p( 5 ); END
  BEGIN s1;
    COBEGIN
      BEGIN f( 8 ); END
      BEGIN s2; END
    COEND // wait for f!
  END
COEND
s3; s4;
```

- START/WAIT can simulate COBEGIN/COEND:

```
COBEGIN
  BEGIN p1(...) END
  BEGIN p2(...) END
COEND
auto t1 = START( p1, ... )
auto t2 = START( p2, ... )
WAIT t1
WAIT t2
```

### 5.8.3 Actor

- An **actor** (Hewitt/Agha) is a unit of work *without* a thread, like BEGIN/END.



- An executor thread matches an actor with a message and runs the actor's behaviour, like COBEGIN/COEND
- Communication is via polymorphic queue of messages (mailbox)  $\Rightarrow$  dynamic type-checking.
- Usually no shared information among actors and no blocking is allowed.
- Actor systems in popular languages: CAF (C++), ProtoActor (Go), Akka (Scala).
- Must declare messages and actors.

```
#include <uActor.h>
struct StrMsg : public uActor::Message { // derived message
  string val; // string message
  StrMsg( string val ) : Message( uActor::Delete ), // delete after use
    val( val ) {}
};
```

```

_Actor Hello { // : public uActor
    Allocation receive( Message & msg ) { // receive base type
        Case( StrMsg, msg ) { // discriminate derived message
            ... msg_d->val; ... // access derived message
        } else Case( StopMsg, msg ) return Delete; // delete actor
        return Nodelete; // reuse actor
    }
};

int main() {
    uActor::start(); // start actor system
    *new Hello() | *new StrMsg( "hello" ) | uActor::stopMsg;
    *new Hello() | *new StrMsg( "bonjour" ) | uActor::stopMsg;
    uActor::stop(); // wait for all actors to terminate
}

```

- Implicit or explicit concurrency?
- Must start actor system (and create thread pool) (**uActor::start()**).
- Actor must receive at least one message to start.
- Messages received in FIFO order from mailbox and executed sequentially.
- Received *derived* message accessed through name *msg\_d*.
- Send messages with operator `|`.
- (StartMsg) `uActor::startMsg` / (StopMsg) `uActor::stopMsg` persistent predefined messages.
- Must wait for actors to complete (**uActor::stop()**).
- Most actor systems leverage garbage collection to manage actors and messages, and the actor system ends after all actors terminate.
- C++ does not have garbage collection so actors/messages use explicit storage-management returning an allocation status for each actor/message.

```

class uActor {
public:
    enum Allocation { Nodelete, Delete, Destroy, Finished }; // allocation actions
    struct Message {
        Allocation allocation;           // allocation action
        ...
    };
    class SenderMsg : public Message {
        uActor * sender();              // sender actor
    };
    static struct StartMsg : public uActor::SenderMsg {} startMsg; // start actor
    static struct StopMsg : public uActor::SenderMsg {} stopMsg; // terminate actor
    static void start(); // create executor to run actors
    static bool stop(); // wait for all actors to terminate or timeout
private:
    Allocation allocation;              // allocation action
};

```

**Nodelete** ⇒ actor or message persists after an actor returns from receive. Use for multi-use actors or messages during their life time. (message default)

**Delete** ⇒ actor or message is deleted after an actor returns from receive. Use with dynamically allocated actors or messages at completion.

**Destroy** ⇒ actor's or message's destructor is called after an actor returns from receive but storage is not deallocated. Use with placement allocated actors or messages at completion.

**Finished** ⇒ actor is marked finished after it returns from receive but neither the destructor is called nor storage deallocated. (No action for a message.) Use with stack allocated actors or messages at completion.

- The executor finds an actor with messages and passes the first message to the actor to process.
- After the actor returns, the executor checks what to do with the message and actor.

```

#include <uActor.h>
struct StrMsg : public uActor::Message { // default Nodelete
    string val;
    StrMsg( string val ) : val( val ) {}
};

_Actor Hello {
    Allocation receive( Message & msg ) {
        Case( StrMsg, msg ) {
            ... msg_d->val ...;
        }
        return Finished; // no delete/destroy but remove from actor system
    }
};

```

```

int main() {
    uActor::start();
    Hello hellos[2]; // stack allocate actors and messages
    StrMsg hello( "hello" ), bonjour( "bonjour" );
    hellos[0] | hello;
    hellos[1] | bonjour;
    uActor::stop();
} // DEALLOCATE ACTORS/MESSAGES

```

- One shot actor with single string message (no stopMsg).

### 5.8.4 Thread Object

- C++ is an object-oriented programming language, which suggests:
  - wrap the thread in an object to leverage all class features
  - use object allocation/deallocation to define thread lifetime rather than control structure

```

        _Task T { // thread type
            void main() {...} // thread starts here
        };
COBEGIN { // { int i, j, k; } ???
        T t; // create object on stack, start thread
COEND } // wait for thread to finish

START T * t = new T; // create thread object on heap, start thread
WAIT delete t; // wait for thread to finish

```

- Block-terminate/delete must wait for each task's thread to finish. Why?
- Unusual to:
  - create object in a block and not use it
  - allocate object and immediately delete it.
- Simulate COBEGIN/COEND with `_Task` object by creating type for each statement:

<pre> int i; _Task T1 {     void main() { i = 1; } }; _Task T2 {     void main() { p1(5); } }; _Task T3 {     void main() { p2(7); } }; _Task T4 {     void main() { p3(9); } }; </pre>	<pre> int main() {     { // COBEGIN         T1 t1; T2 t2; T3 t3; T4 t4;     } // COEND } void p1(...) {     { // COBEGIN         T5 t5; T6 t6; T7 t7; T8 t8;     } // COEND } </pre>
---	--

- Simulate START/WAIT with `_Task` object by creating type for each call:

```

int i;
_Task T1 {
    void main() { p(5); }
};
_Task T2 {
    int temp;
    void main() { temp = f(8); }
public:
    ~T2() { i = temp; }
};

```

```

int main() {
    T1 * tp = new T1; // start T1
    ... s1 ...
    T2 * tf = new T2; // start T2
    ... s2 ...
    delete tp; // wait for p
    ... s3 ...
    delete tf; // wait for f
    ... s4 ...
}

```

- Variable `i` cannot be assigned until `tf` is deleted, otherwise the value could change in `s2/s3`.
- Implicit or explicit concurrency?

## 5.9 Termination Synchronization

- A thread terminates when:
  - it finishes normally
  - it finishes with an error
  - it is killed by its parent (or sibling) (not supported in  $\mu\text{C++}$ )
  - because the parent terminates (not supported in  $\mu\text{C++}$ )
- Children can continue to exist even after the parent terminates (although this is rare).
  - E.g. sign off and leave child process(es) running
- Synchronizing at termination is possible for independent threads.
- Termination synchronization may be used to perform a final communication.

## 5.10 Divide-and-Conquer

- Divide-and-conquer is characterized by ability to subdivide work across data  $\Rightarrow$  work can be performed independently on the data.
- Work performed on each data group is identical to work performed on data as whole.
- Taken to extremes, each data item is processed independently, but administration of concurrency becomes greater than cost of work.
- Only termination synchronization is required to know when the work is done
- Partial results are then processed further if necessary.

- Sum rows of a matrix concurrently using concurrent statement:

```

#include <uCobegin.h>
int main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix
    COFOR( r, 0, rows,
    // for ( int r = 0; r < rows; r += 1 )
        subtotals[r] = 0; // r is loop number
        for ( int c = 0; c < cols; c += 1 )
            subtotals[r] += matrix[r][c];
    ); // wait for threads
    for ( int r = 0; r < rows; r += 1 ) {
        total += subtotals[r]; // total subtotals
    }
    cout << total << endl;
}

```

	matrix				subtotals
T <sub>0</sub> Σ	23	10	5	7	0
T <sub>1</sub> Σ	-1	6	11	20	0
T <sub>2</sub> Σ	56	-13	6	0	0
T <sub>3</sub> Σ	-2	8	-5	1	0
	total				Σ

- COFOR *logically* creates end – start threads, indexed start..end – 1 one per loop body.
- Implicit or explicit concurrency?
- Sum rows of a matrix concurrently using actors:

```

_Actor Adder {
    int * row, cols, & subtotal; // communication
    Allocation receive( Message & ) { // only startMsg
        subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
        return Delete; // delete actor (match new)
    }
public:
    Adder( int row[], int cols, int & subtotal ) :
        row( row ), cols( cols ), subtotal( subtotal ) {}
};

int main() {
    ... // same
    uActor::start(); // start actor system
    for ( int r = 0; r < rows; r += 1 ) { // actor per row
        *new Adder( matrix[r], cols, subtotals[r] ) | uActor::startMsg;
    }
    uActor::stop(); // wait for all actors to terminate
    ... // same
} // main

```

**Constructor is a cheat message and may not be allowed.**

- Sum rows of a matrix concurrently using concurrent objects:



```

_Task Adder {
    int * row, cols, & subtotal;           // communication
    void main() {
        subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
    }
public:
    Adder( int row[], int cols, int & subtotal ) :
        row( row ), cols( cols ), subtotal( subtotal ) {}
};

int main() {
    ... // same
    Adder * adders[rows];
    for ( int r = 0; r < rows; r += 1 ) { // start threads to sum rows
        adders[r] = new Adder( matrix[r], cols, subtotals[r] );
    }
    for ( int r = 0; r < rows; r += 1 ) { // wait for threads to finish
        delete adders[r];
        total += subtotals[r];           // total subtotals
    }
    cout << total << endl;
}

int main() {
    ... // same
    {
        uArrayPtr( Adder, adders, rows );
        for ( int r = 0; r < rows; r += 1 ) { // start threads to sum rows
            adders[r]( matrix[r], cols, subtotals[r] );
        }
    } // wait for tasks to terminate
    for ( int r = 0; r < rows; r += 1 ) {
        total += subtotals[r]; // total subtotals
    }
}

```

- Why create objects in the heap versus stack?
- Does it matter in what order objects are created?
- Does it matter in what order objects are deleted? (critical path)

## 5.11 Exceptions

- Exceptions can be handled locally within a task, or nonlocally among coroutines, or concurrently among tasks.
  - All concurrent exceptions are nonlocal, but nonlocal exceptions can also be sequential.
- Local task exceptions are different for coroutines and tasks.

- Unhandled exception goes to coroutine's last resumer and task's joiner.
- Nonlocal exceptions are possible because each coroutine/task has its own stack (execution state)
- Nonlocal exceptions between a task and a coroutine are the same as between coroutines (single thread).
- Concurrent exceptions among tasks are more complex due to the multiple threads.
- A concurrent exception provides an additional kind of communication among tasks.
- For example, two tasks may begin searching for a key in different sets:

```

_Exception StopEvent {};
_Task Searcher {
    Searcher * partner;
    void main() {
        try {
            _Enable {                               // allow nonlocal exceptions
                ...                                  // search
                if ( key == ... ) {                // found result
                    _Resume StopEvent() _At *partner; // stop partner
                    _Throw StopEvent(); // stop me
                }
            }
        } catch( StopEvent ) {...} // reset for next search
    }
}

```

- When one task finds the key, it informs the other task to stop searching.
- For a concurrent raise, the source execution may only block while queueing the event for delivery at the faulting execution.
- After event is delivered, faulting execution it is not interrupted, it polls:
  - when an **\_Enable** statement begins/ends,
  - after a call to suspend/resume,
  - after a call to yield,
  - after a call to **\_Accept** unblocks for RendezvousFailure.

**Therefore exception delivery is NOT instantaneous and task continues running.**

- Similar to coroutines, see Section 3.7, p. 40, an unhandled exception for a task raises the nonlocal exception `uBaseCoroutine::UnhandledException` at the task's *joiner* and then terminates the task.

```

    _Exception E {};
    _Task T {
        void main() { _Throw E(); } // unwind
    };
    int main() {
        try {
            { // extra block
                T t;
            } // continue_CatchResume
        } _CatchResume( uBaseCoroutine::UnhandledException & ) {...} // one of
        catch( uBaseCoroutine::UnhandledException & ) {...}
        // catch continues after try
    }

```

- Forwarding of UnhandledException occurs across any number of tasks (and coroutines), until the program main forwards and the program terminates by calling main's set\_terminate.

## 5.12 Synchronization and Communication During Execution

- Synchronization occurs when one thread waits until another thread has reached a certain execution point (state and code).
- One place synchronization is needed is in transmitting data between threads.
  - One thread has to be ready to transmit the information and the other has to be ready to receive it, simultaneously.
  - Otherwise one might transmit when no one is receiving, or one might receive when nothing is transmitted.

<pre> <b>bool</b> Insert = <b>false</b>, Remove = <b>false</b>; <b>int</b> Data;  _Task Prod {     <b>int</b> N;     <b>void</b> main() {         <b>for</b> ( <b>int</b> i = 1; i &lt;= N; i += 1 ) { 1           <b>Data = i;</b> // transfer data 2           <b>Insert = true;</b> 3           <b>while ( ! Remove ) {}</b> // busy wait 4           <b>Remove = false;</b>         }     } <b>public:</b>     Prod( <b>int</b> N ) : N( N ) {} }; </pre>	<pre> _Task Cons {     <b>int</b> N;     <b>void</b> main() {         <b>int</b> data;         <b>for</b> ( <b>int</b> i = 1; i &lt;= N; i += 1 ) { 1           <b>while ( ! Insert ) {}</b> // busy wait 2           <b>Insert = false;</b> 3           <b>data = Data;</b> // remove data 4           <b>Remove = true;</b>         }     } <b>public:</b>     Cons( <b>int</b> N ) : N( N ) {} }; <b>int</b> main() {     Prod prod( 5 ); Cons cons( 5 ); } </pre>
---	---

- 2 infinite loops! No, because of implicit switching between threads.

- cons synchronizes (waits) until prod transfers some data, then prod waits for cons to remove the data.
- A loop waiting for an event among threads is called a **busy wait**.
- Are 2 synchronization flags necessary?

### 5.13 Communication

- Once threads are synchronized there are many ways that information can be transferred from one thread to the other.
- If the threads are in the same memory, then information can be transferred by value or address (e.g., reference parameter).
- If the threads are not in the same memory (distributed), then transferring information by value is straightforward but by address is difficult.

### 5.14 Critical Section

- Threads may access non-concurrent objects, like a file or linked-list.
- There is a potential problem if there are multiple threads attempting to operate on the same object simultaneously.
- Not a problem if the operation on the object is **atomic** (not divisible).
- This means no other thread can modify any partial results during the operation on the object (but the thread can be interrupted).
- Where an operation is composed of many instructions, it is often necessary to make the operation atomic.
- A group of instructions on an associated object (data) that must be performed atomically is called a **critical section**.
- Preventing simultaneous execution of a critical section by multiple threads is called **mutual exclusion**.
- Must determine when concurrent access is allowed and when it must be prevented.
- Handle by detecting any sharing and serialize all access.
  - Wasteful if threads are only reading.
- Improve by differentiating between reading and writing
  - allow multiple readers or a single writer; still wasteful as a writer may only write at the end of its usage.
- **Need to minimize the amount of mutual exclusion (i.e., make critical sections as small as possible, Amdahl's law) to maximize concurrency.**

## 5.15 Static Variables

- **Warning:** static variables in a class are shared among all objects generated by that class.
- These shared variables may need mutual exclusion for correct usage.
- However, a few special cases where **static** variables can be used safely, e.g., task constructor.
- If task objects are generated serially, **static** variables can be used in the constructor.
- E.g., assigning each task its own name:

```

_Task T {
    static int tid;
    string name; // must supply storage
    ...
public:
    T() {
        name = "T" + to_string( tid ); // shared read
        setName( name.c_str() ); // name task
        tid += 1; // shared write
    }
    ...
};
int T::tid = 0; // initialize static variable in .C file
T t[10]; // 10 tasks with individual names

```

- Task constructor is executed by the creating thread, so array constructors executed sequentially.
- This approach only works if one task creates all the objects and initialization data is internal.
- Instead of **static** variables, pass a task identifier to the constructor:

```

T::T( int tid ) { ... } // constructor naming
T * t[10];
for ( int i = 0; i < 10; i += 1 ) {
    t[i] = new T( i ); // individual names
}
uArray( T, t, 10 );
for ( int i = 0; i < 10; i += 1 ) {
    t[i]( i );
}

```

- In general, it is best to avoid using shared **static** variables in a concurrent program.

## 5.16 Mutual Exclusion Game

- Is it possible to write code guaranteeing a statement (or group of statements) is always serially executed by 2 threads?
- Rules of the Game:
  1. Only one thread can be in a critical section at a time with respect to a particular object (**safety**).

2. Threads may run at arbitrary speed and in arbitrary order, while the underlying system guarantees a thread makes progress (i.e., threads get some CPU time).
  3. If a thread is not in the entry or exit code controlling access to the critical section, it may not prevent other threads from entering the critical section.
  4. In selecting a thread for entry to a critical section, a selection cannot be postponed indefinitely (**liveness**). *Not* satisfying this rule is called **indefinite postponement** or **livelock**.
  5. After a thread starts entry to the critical section, it must eventually enter. *Not* satisfying this rule is called **starvation**.
- **Indefinite postponement and starvation are related by busy waiting.**
  - Unlike synchronization, looping for an event in mutual exclusion **must** ensure eventual progress.
  - Threads waiting to enter can be serviced in any order, as long as each thread eventually enters.
  - If threads are *not* serviced in first-come first-serve (FCFS) order of arrival, there is a notion of **unfairness**
  - Unfairness implies waiting threads are overtaken by arriving threads, called **barging**.

### 5.17 Self-Testing Critical Section

```

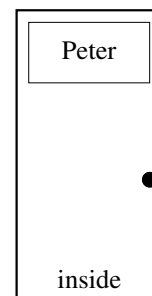
void CriticalSection() {
    static uBaseTask * curr; // shared
    curr = &uThisTask();
    for ( int i = 1; i <= 100; i += 1 ) {
        ...
        if ( curr != &uThisTask() ) {
            abort( "interference" );
        }
    }
}

```

```

// work
// check

```



- What is the minimum number of interference tests and where?
- Why are multiple tests useful?

## 5.18 Software Solutions

### 5.18.1 Lock

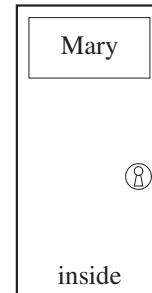
```

enum Yale { CLOSED, OPEN } Lock = OPEN; // shared

_Task PermissionLock {
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      while ( ::Lock == CLOSED ) {} // entry protocol
      ::Lock = CLOSED;
      CriticalSection();           // critical section
      ::Lock = OPEN;              // exit protocol
    }
  }
public:
  PermissionLock() {}
};
int main() {
  PermissionLock t0, t1;
}

```

Breaks rule 1



### 5.18.2 Alternation

```

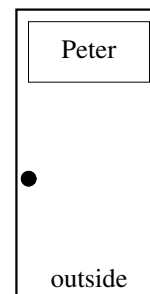
int Last = 0; // shared

_Task Alternation {
  int me;

  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      while ( ::Last == me ) {} // entry protocol
      CriticalSection();         // critical section
      ::Last = me;              // exit protocol
    }
  }
public:
  Alternation( int me ) : me( me ) {}
};
int main() {
  Alternation t0( 0 ), t1( 1 );
}

```

Breaks rule 3



## 5.18.3 Declare Intent

```

enum Intent { WantIn, DontWantIn };

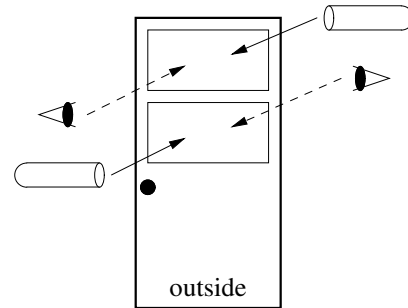
_Task DeclIntent {
  Intent & me, & you;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      me = WantIn;           // entry protocol
      while ( you == WantIn ) {}
      CriticalSection();     // critical section
      me = DontWantIn;      // exit protocol
    }
  }
};

public:
  DeclIntent( Intent & me, Intent & you ) :
    me(me), you(you) {}
};

int main() {
  Intent me = DontWantIn, you = DontWantIn;
  DeclIntent t0( me, you ), t1( you, me );
}

```

Breaks rule 4



## 5.18.4 Retract Intent

```

enum Intent { WantIn, DontWantIn };

_Task RetractIntent {
  Intent & me, & you;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      for ( ;; ) {           // entry protocol
        me = WantIn;
        if ( you == DontWantIn ) break;
        me = DontWantIn;
        while ( you == WantIn ) {}
      }
      CriticalSection();     // critical section
      me = DontWantIn;      // exit protocol
    }
  }
};

public:
  RetractIntent( Intent & me, Intent & you ) : me(me), you(you) {}
};

int main() {
  Intent me = DontWantIn, you = DontWantIn;
  RetractIntent t0( me, you ), t1( you, me );
}

```

Breaks rule 4

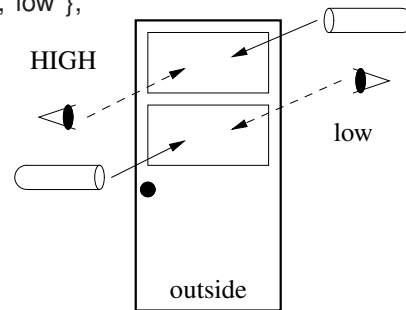


## 5.18.5 Prioritized Retract Intent

```

enum Intent { WantIn, DontWantIn }; enum Priority { HIGH, low };
_Task PriorityEntry {
  Intent & me, & you; Priority priority;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
      for ( ;; ) { // entry protocol
        me = WantIn;
        if ( you == DontWantIn ) break;
        if ( priority == low ) {
          me = DontWantIn;
          while ( you == WantIn ) {} // busy wait
        }
      }
      CriticalSection(); // critical section
      me = DontWantIn; // exit protocol
    }
  }
};
public:
  PriorityEntry( Priority p, Intent & me, Intent & you ) : priority(p), me(me), you(you) {}
};
int main() {
  Intent me = DontWantIn, you = DontWantIn;
  PriorityEntry t0( HIGH, me, you ), t1( low, you, me );
} // main

```



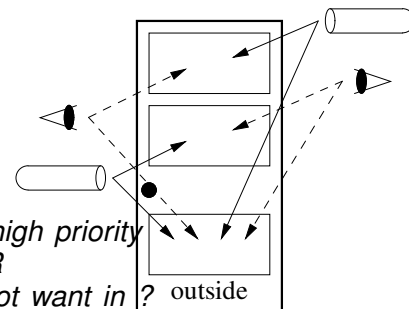
Breaks rule 5

## 5.18.6 Dekker (modified retract intent)

```

enum Intent { WantIn, DontWantIn };
Intent * Last;
_Task Dekker {
  Intent & me, & you;
  void main() {
    for ( int i = 1; i <= 1000; i += 1 ) {
1      for ( ;; ) { // entry protocol, high priority
2        me = WantIn; // READ FLICKER
3        if ( you == DontWantIn ) break; // does not want in
4        if ( ::Last == &me ) { // low priority task ?
5          me = DontWantIn; // retract intent, READ FLICKER
6          while ( ::Last == &me // low priority busy wait
              && you == WantIn ) {}
        }
7        CriticalSection();
8        if ( ::Last != &me ) // exit protocol
9          ::Last = &me; // READ FLICKER
10       me = DontWantIn; // READ FLICKER
    }
  }
};
public:
  Dekker( Intent & me, Intent & you ) : me(me), you(you) {}
};

```



```

int main() {
    Intent me = DontWantIn, you = DontWantIn;
    ::Last = &me;          // arbitrary who starts as last
    Dekker t0( me, you ), t1( you, me );
}

```

- Dekker's algorithm appears **RW-safe**.
  - On cheap multi-core computers, read/write is not atomic.
  - Hence, simultaneous writes scramble bits, and for simultaneous read/write, read sees flickering bits during write.
  - RW-safe means a mutual-exclusion algorithm works for non-atomic read/write.
  - Dekker has no simultaneous W/W because intent reset *after* alternation in exit protocol.
  - Dekker has simultaneous R/W but all are equality so works *if final value never flickers*.
- 2015 Hesselink found two failure case if final values flickers:

1.	T <sub>0</sub>	T <sub>1</sub>
	9 ::Last = &me	
	10 me = DontWantIn	
	(flicker DontWantIn)	
		3 you == DontWantIn ( <b>true</b> )
		7 Critical Section
		9 ::Last = &me
	(flicker WantIn)	
		3 you == DontWantIn ( <b>false</b> )
		4 ::Last == &me ( <b>true</b> )
		6 low priority wait
	(flicker DontWantIn)	
	terminate	
		6 ::Last == &me ( <b>true</b> , spin forever)

T<sub>1</sub> spins forever (break rule 4)

2.	T <sub>0</sub>	T <sub>1</sub>
	7 Critical Section	
	9 ::Last = &me	
	(flicker you T <sub>1</sub> )	
		6 ::Last == &me
		&& you == WantIn ( <b>true</b> )
	(flicker me T <sub>0</sub> )	
	10 me = DontWantIn	
	(repeat)	(repeat)

T<sub>1</sub> starvation (break rule 5)

- RW-safe version ([Hesselink](#))

- line 6: add conjunction `you == WantIn`  
⇒ stop spinning
- line 8: add conditional assignment to `::Last`  
⇒ not assigning at line 9 when `::Last != &me` prevents flicker so  $T_1$  makes progress.
- Dekker has **unbounded overtaking** (not starvation) because *race loser retracts intent*.
- ⇒ thread exiting critical does not exclude itself for reentry.
  - $T_0$  exits critical section and attempts reentry
  - $T_1$  is now high priority (`Last != me`) but delays in low-priority busy-loop and resetting its intent.
  - $T_0$  can enter critical section unbounded times until  $T_1$  resets its intent
  - $T_1$  sets intent ⇒ bound of 1 as  $T_0$  can be entering or in critical section
- Unbounded overtaking is allowed by rule 3: not preventing entry to the critical section by the delayed thread.

### 5.18.7 Peterson (modified declare intent)

```

enum Intent { WantIn, DontWantIn };
Intent * Last;

_Task Peterson {
    Intent & me, & you;
    void main() {
        for ( int i = 1; i <= 1000; i += 1 ) {
1           me = WantIn;           // entry protocol, order matters
2           ::Last = &me;         // RACE!
3           while ( you == WantIn && ::Last == &me ) {}
4           CriticalSection();     // critical section
5           me = DontWantIn;      // exit protocol
        }
    }
public:
    Peterson( Intent & me, Intent & you ) : me(me), you(you) {}
};
int main() {
    Intent me = DontWantIn, you = DontWantIn;
    Peterson t0(me, you), t1(you, me);
}

```

- Peterson's algorithm is RW-**unsafe** requiring atomic read/write operations.
- Peterson has **bounded overtaking** because *race loser does not retracts intent*.
- ⇒ thread exiting critical excludes itself for reentry.
  - $T_0$  exits critical section and attempts reentry
  - $T_0$  runs race by itself and loses

- T0 must wait (Last == me)
  - T1 eventually sees (Last != me)
- Bounded overtaking is allowed by rule 3 because the prevention is occurring *in the entry protocol*.
  - Can line 2 be moved before 1?

```

1  2  ::Last = &me;           // RACE!
2  1  me = WantIn;          // entry protocol
3  3  while ( you == WantIn && ::Last == &me ) {}
4  4  CriticalSection();      // critical section
5  5  me = DontWantIn;       // exit protocol

```

- T0 executes Line 1  $\Rightarrow$  ::Last = T0
- T1 executes Line 1  $\Rightarrow$  ::Last = T1
- T1 executes Line 2  $\Rightarrow$  T1 = WantIn
- T1 enters CS, because T0 == DontWantIn
- T0 executes Line 2  $\Rightarrow$  T0 = WantIn
- T0 enters CS, because ::Last == T1

### 5.18.8 N-Thread Prioritized Entry

```

enum Intent { WantIn, DontWantIn };
_Task NTask { // Lamport (simpler version of Burns-Lynch)
  Intent * intents;           // position & priority
  int N, priority, i, j;
  void main() {
    for ( i = 1; i <= 1000; i += 1 ) {
      // step 1, wait for tasks with higher priority
      do {                       // entry protocol
        intents[priority] = WantIn;
        // check if task with higher priority wants in
        for ( j = priority-1; j >= 0; j -= 1 ) {
          if ( intents[j] == WantIn ) {
            intents[priority] = DontWantIn;
            while ( intents[j] == WantIn ) {}
            break;
          }
        }
      } while ( intents[priority] == DontWantIn );
    }
  }
}

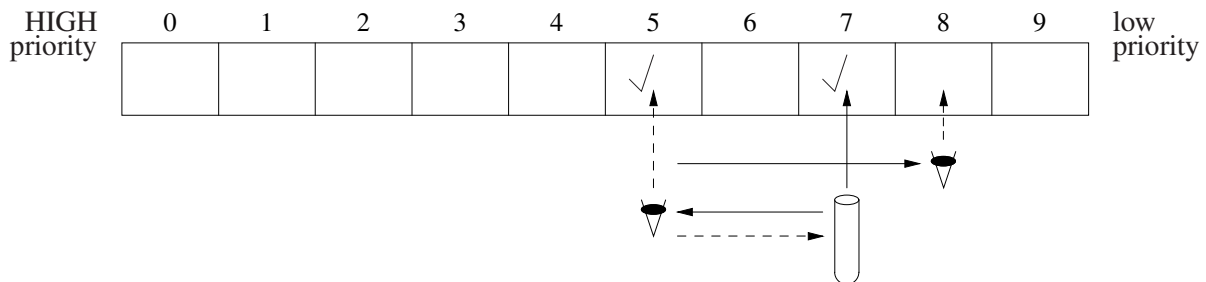
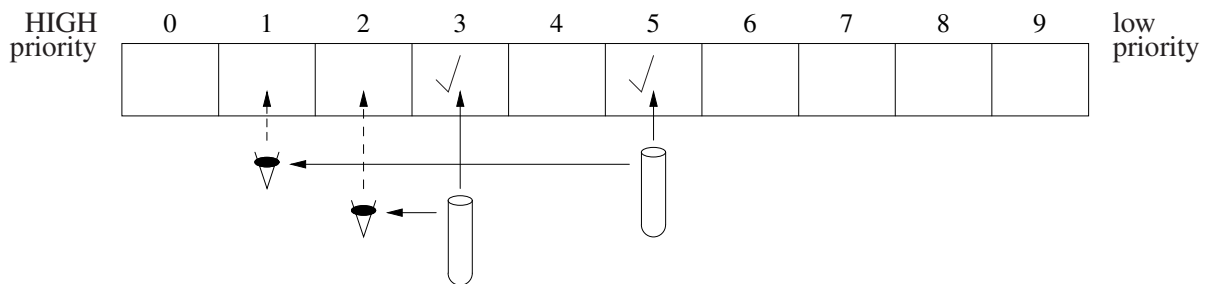
```

```

// step 2, wait for tasks with lower priority
for ( j = priority+1; j < N; j += 1 ) {
    while ( intents[j] == WantIn ) {}
}
CriticalSection();
intents[priority] = DontWantIn;    // exit protocol
}
}
public:
    NTask( Intent i[], int N, int p ) : intents(i), N(N), priority(p) {}
};

```

Breaks rule 5



- Only  $N$  bits needed.
- No known solution for all 5 rules using only  $N$  bits.
- Other  $N$ -thread solutions use more memory.  
(best: 3-bit RW-unsafe, 4-bit RW-safe).

## 5.18.9 N-Thread Bakery (Tickets)

```

_Task Bakery { // (Lamport) Hehner-Shyamasundar
  int * ticket, N, priority;
  void main() {
    for ( int i = 0; i < 1000; i += 1 ) {
      // step 1, select a ticket
      ticket[priority] = 0;           // highest priority
      int max = 0;                    // O(N) search
      for ( int j = 0; j < N; j += 1 ) { // for largest ticket
        int v = ticket[j];           // can change so copy
        if ( v != INT_MAX && max < v ) max = v;
      }
      max += 1;                       // advance ticket
      ticket[priority] = max;
      // step 2, wait for ticket to be selected
      for ( int j = 0; j < N; j += 1 ) { // check tickets
        while ( ticket[j] < max ||
              (ticket[j] == max && j < priority) ) {}
      }
      CriticalSection();
      ticket[priority] = INT_MAX;     // exit protocol
    }
  }
}
public:
  Bakery( int t[], int N, int p ) : ticket(t), N(N), priority(p) {}
};

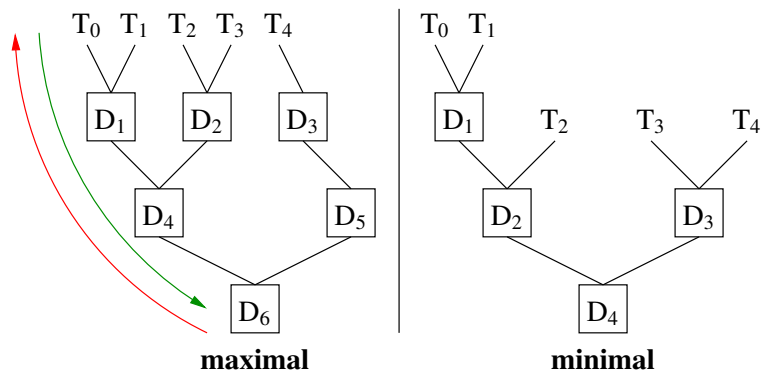
```

HIGH priority	0	1	2	3	4	5	6	7	8	9	low priority
	$\infty$	$\infty$	17	$\infty$	0	18	18	0	20	19	

- ticket value of  $\infty$  (INT\_MAX)  $\Rightarrow$  don't want in
  - ticket value of **0**  $\Rightarrow$  selecting ticket
  - ticket selection is unusual
  - tickets are not unique  $\Rightarrow$  use position as secondary priority
  - low ticket and position  $\Rightarrow$  high priority
  - ticket values cannot increase indefinitely  $\Rightarrow$  could fail (probabilistically correct)
  - ticket value reset to INT\_MAX when no attempted entry
  - $NM$  bits, where  $M$  is the ticket size (e.g., 32 bits)
  - Lamport RW-safe
  - Hehner/Shyamasundar RW-unsafe
- assignment ticket[priority] = max can flickers to INT\_MAX  $\Rightarrow$  other tasks proceed

### 5.18.10 Tournament

- Binary (d-ary) tree with  $\lceil N/2 \rceil$  start nodes and  $\lceil \lg N \rceil$  levels.



- Thread assigned to start node, where it begins mutual exclusion process.
- Each node is like a Dekker or Peterson 2-thread algorithm.
- Tree structure tries to find compromise between fairness and performance.
- Exit protocol must retract intents in *reverse* order.
- Otherwise race between retracting/released threads along same tree path:
  - $T_0$  retracts its intent (left) at  $D_1$ ,
  - $T_1$  (right) now moves from  $D_1$  to  $D_4$ , sets its intent at  $D_4$  (left), and with no competition at  $D_4$  proceeds to  $D_6$  (left),
  - $T_0$  (left) now retracts the intent at  $D_4$  set by  $T_1$ ,
  - $T_{2/3}$  continue from  $D_2$ , sets its intent at  $D_4$  (right), and with no competition at  $D_4$  (left) proceeds to  $D_6$ , which ultimately violates mutual exclusion.
- No overall livelock because each node has no livelock.
- No starvation because each node guarantees progress, so each thread eventually reaches the root.
- Tournament algorithm RW-safety depends on MX algorithm; tree traversal is local to each thread.
- Tournament algorithms have unbounded overtaking as no synchronization among the nodes of the tree.
- For a minimal binary tree, the tournament approach uses  $(N - 1)M$  bits, where  $(N - 1)$  is the number of tree nodes and  $M$  is the node size (e.g., intent, turn).

```

_Task TournamentMax { // Taubenfeld-Buhr
    struct Token { int intents[2], turn; }; // intents/turn
    static Token ** t; // triangular matrix
    int depth, id;

    void main() {
        unsigned int lid; // local id at each tree level
        for ( int i = 0; i < 1000; i += 1 ) {
            lid = id; // entry protocol
            for ( int lv = 0; lv < depth; lv += 1 ) {
                binary_prologue( lid & 1, &t[lv][lid >> 1] );
                lid >>= 1; // advance local id for next tree level
            }
            CriticalSection( id );
            for ( int lv = depth - 1; lv >= 0; lv -= 1 ) { // exit protocol
                lid = id >> lv; // retract reverse order
                binary_epilogue( lid & 1, &t[lv][lid >> 1] );
            }
        }
    }
}
public:
    TournamentMax( struct Token * t[], int depth, int id ) :
        t( t ), depth( depth ), id( id ) {}
};

```

- Can be optimized to 3 shifts and exclusive-or using Peterson 2-thread for binary.
- Path from leaf to root is fixed per thread  $\Rightarrow$  table lookup possible using max or min tree.

### 5.18.11 Arbiter

- Create full-time arbitrator task to control entry to critical section.

```

bool intents[N], serving[N]; // initialize to false

_Task Client {
    int me;
    void main() {
        for ( int i = 0; i < 100; i += 1 ) {
            intents[me] = true; // entry protocol
            while ( ! serving[me] ) {} // busy wait
            CriticalSection();
            serving[me] = false; // exit protocol
        }
    }
}
public:
    Client( int me ) : me( me ) {}
};

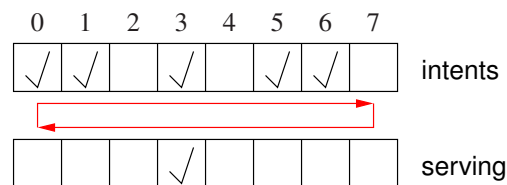
```



```

_Task Arbiter {
  void main() {
    int i = N; // force cycle to start at id=0
    for ( ;; ) {
      do { // circular search => no starvation
        i = (i + 1) % N; // advance next client
      } while ( ! intents[i] ); // not want in ?
      intents[i] = false; // retract intent on behalf of client
      serving[i] = true; // wait for exit from critical section
      while ( serving[i] ) {} // busy wait
    }
  }
};

```



- Mutual exclusion becomes synchronization between arbiter and clients.
- Arbiter never uses the critical section  $\Rightarrow$  no indefinite postponement.
- Arbiter cycles through waiting clients (not FCFS)  $\Rightarrow$  no starvation.
- RW-unsafe due to read flicker.
- Cost is creation, management, and execution (continuous busy waiting) of arbiter task.

## 5.19 Hardware Solutions

- Software solutions to the critical-section problem rely on
  - shared information,
  - communication among threads,
  - (maybe) atomic memory-access.
- Hardware solutions introduce level below software level.
- Cheat by making assumptions about execution impossible at software level.
  - E.g., control order and speed of execution.
- Allows elimination of much of the shared information and the checking of this information required in the software solution.
- Special instructions to perform an **atomic read and write operation**.
- Sufficient for multitasking on a single CPU.

### 5.19.1 Test/Set Instruction

- Simple lock of critical section fails:

```
int Lock = OPEN;           // shared
// each task does
while ( Lock == CLOSED ); // fails to achieve (read)
Lock = CLOSED;           // mutual exclusion (write)
// critical section
Lock = OPEN;
```

- The test-and-set instruction performs an atomic read and fixed assignment.

```
int Lock = OPEN; // shared

int TestSet( int & b ) { // begin atomic
    int temp = b;
    b = CLOSED;
    // end atomic
    return temp;
}

void Task::main() { // each task does
    while( TestSet( Lock ) == CLOSED );
    // critical section
    Lock = OPEN;
}
```

- if test/set returns open  $\Rightarrow$  loop stops and lock is set to closed
- if test/set returns closed  $\Rightarrow$  loop executes until the other thread sets lock to open
- Works for N threads attempting entry to critical section and only depends on one shared datum (lock).
- However, rule 5 is broken, as there is no guarantee of eventual progress.
- In multiple CPU case, hardware (bus) must also guarantee multiple CPUs cannot interleave these special R/W instructions on same memory location.

### 5.19.2 Swap Instruction

- The swap instruction performs an atomic interchange of two separate values.

```
int Lock = OPEN; // shared

void Swap( int & a, & b ) { // begin atomic
    int temp;
    temp = a;
    a = b;
    b = temp;
    // end atomic
}

void Task::main() { // each task does
    int dummy = CLOSED;
    do {
        Swap( Lock, dummy );
    } while( dummy == CLOSED );
    // critical section
    Lock = OPEN;
}
```

- if dummy returns open  $\Rightarrow$  loop stops and lock is set to closed
- if dummy returns closed  $\Rightarrow$  loop executes until the other thread sets lock to open

### 5.19.3 Fetch and Increment Instruction

- The fetch-and-increment instruction performs an increment between the read and write.

```

int Lock = 0; // shared

int FetchInc( int & val ) {
    // begin atomic
    int temp = val;
    val += 1;
    // end atomic
    return temp;
}

void Task::main() { // each task does
    while ( FetchInc( Lock ) != 0 );
    // critical section
    Lock = 0;
}

```

- Often fetch-and-increment is generalized to add any value  $\Rightarrow$  also decrement with negative value.
- Lock counter can overflow during busy waiting and starvation (rule 5).
- Use ticket counter to solve both problems (Bakery Algorithm, see Section 5.18.9, p. 94):

```

class ticketLock {
    unsigned int tickets, serving;
public:
    ticketLock() : tickets( 0 ), serving( 0 ) {}
    void acquire() { // entry protocol
        int ticket = FetchInc( tickets ); // obtain a ticket
        while ( ticket != serving ) {} // busy wait
    }
    void release() { // exit protocol
        serving += 1;
    }
};

```

- Ticket overflow is a problem only if all values used simultaneously, and FIFO service  $\Rightarrow$  no starvation.

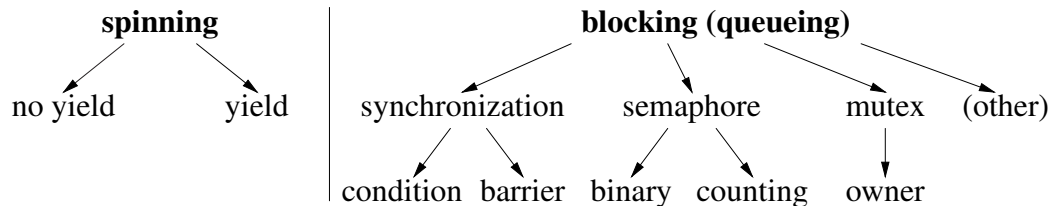


## 6 Locks

- Package software/hardware locking into abstract type for general use.
- Locks are constructed for synchronization or mutual exclusion or both.

### 6.1 Lock Taxonomy

- Lock implementation is divided into two general categories: spinning and blocking.



- Spinning locks busy wait until an event occurs  $\Rightarrow$  task oscillates between ready and running states due to time slicing.
- Blocking locks do not busy wait, but block until an event occurs  $\Rightarrow$  **some other mechanism must unblock waiting task when the event happens.**
- Within each category, different kinds of spinning and blocking locks exist.

### 6.2 Spin Lock

- A **spin lock** is implemented using busy waiting, which loops checking for an event to occur.

```
while( TestSet( Lock ) == CLOSED ); // use up time-slice (no yield)
```

- So far, when a task is busy waiting, it loops until:
  - critical section becomes unlocked or an event happens.
  - waiting task is preempted (time-slice ends) and put back on ready queue.

Hence, CPU is wasting time constantly checking the event.

- To increase uniprocessor efficiency, a task can:
  - explicitly terminate its time-slice
  - move back to the ready state after only **one** event-check fails. (Why one?)
- Task member yield relinquishes time-slice by *rescheduling* running task back onto ready queue.

```
while( TestSet( Lock ) == CLOSED ) uThisTask().yield(); // relinquish time-slice
```

- To increase multiprocessor efficiency, a task can yield after  $N$  event-checks fail. (Why  $N$ ?)

- Some spin-locks allow adjustment of spin duration, called **adaptive spin-lock**.
- Most spin-lock implementations break rule 5, i.e., no bound on service.  $\Rightarrow$  possible starvation of one or more tasks.
- Spin lock is appropriate and necessary in situations where there is no other work to do.

### 6.2.1 Implementation

- $\mu$ C++ provides a non-yielding spin lock, `uSpinLock`, and a yielding spin lock, `uLock`.

<pre>class uSpinLock { public:     uSpinLock(); // open     void acquire();     bool tryacquire();     void release(); };</pre>	<pre>class uLock { public:     uLock( unsigned int value = 1 );     void acquire();     bool tryacquire();     void release(); };</pre>
---	---

- Both locks are built directly from an atomic hardware instruction.
- Lock starts closed (0) or opened (1); waiting tasks compete to acquire lock after release.
- In theory, starvation could occur; in practice, it is seldom a problem.
- `tryacquire` makes one attempt to acquire the lock, i.e., it does not wait.
- It is *not* meaningful to read or to assign to a lock variable, or copy a lock variable, e.g., pass it as a value parameter.
- synchronization

<pre>_Task T1 {     uLock &amp; lk;     void main() {         ...         S1         lk.release();         ...     } public:     T1( uLock &amp; lk ) : lk(lk) {} };  int main() {     uLock lock( 0 ); // closed     T1 t1( lock );     T2 t2( lock ); }</pre>	<pre>_Task T2 {     uLock &amp; lk;     void main() {         ...         lk.acquire();         S2         ...     } public:     T2( uLock &amp; lk ) : lk(lk) {} };</pre>
---	--

- mutual exclusion

```

_Task T {
    uLock & lk;
    void main() {
        ...
        lk.acquire();
        // critical section
        lk.release();
        ...
        lk.acquire();
        // critical section
        lk.release();
        ...
    }
public:
    T( uLock & lk ) : lk(lk) {}
};

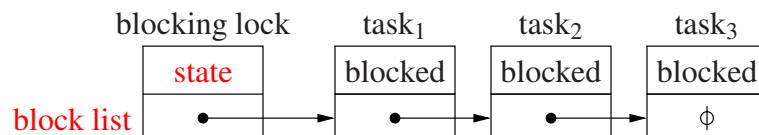
int main() {
    uLock lock( 1 ); // open
    T t0( lock ), t1( lock );
}

```

- Does this solution afford maximum concurrency?
- Depends on critical sections: **independent** (disjoint) or **dependent**.
- How many locks are needed for mutual exclusion?

### 6.3 Blocking Locks

- For spinning locks,
  - acquiring task(s) is solely responsible for detecting an open lock after the releasing task opens it.
- For blocking locks,
  - acquiring task makes **one** check for open lock and blocks
  - releasing task has sole responsibility for detecting blocked acquirer and transferring lock, or just releasing lock.
- Blocking locks reduce busy waiting by having releasing task do additional work: **cooperation**.
  - What advantage does the releasing task get from doing the cooperation?
- Therefore, all blocking locks have
  - state to facilitate lock semantics
  - list of blocked acquirers



- Which task is scheduled next from the list of blocked tasks?

### 6.3.1 Mutex Lock

- **Mutex lock** is used solely to provide mutual exclusion.
- Restricting a lock to just mutual exclusion:
  - separates lock usage between synchronization and mutual exclusion
  - permits optimizations and checks as the lock only provides one specialized function
- Mutex locks are divided into two kinds:
  - **single acquisition** : task that acquired the lock cannot acquire it again
  - **multiple acquisition** : lock owner can acquire it multiple times, called an **owner lock**
- Multiple acquisition can handle looping or recursion involving a lock:

```
void f() {
    ...
    lock.acquire();
    ... f();      // recursive call within critical section
    lock.release();
}
```

- May require only one release to unlock, or as many releases as acquires.

#### 6.3.1.1 Implementation

- Multiple acquisition lock manages owner state (**blue**).

```
class MutexLock {
    bool avail;           // resource available ?
    Task * owner         // lock owner
    queue<Task> blocked; // blocked tasks
    SpinLock lock;      // mutex nonblocking lock

public:
    MutexLock() : avail( true ), owner( nullptr ) {}
    void acquire() {
        lock.acquire();           // barging
        while ( ! avail && owner != currThread() ) { // busy waiting
            // add self to lock' s blocked list
            yieldNoSchedule();    // do not reschedule to ready queue
            lock.acquire();       // reacquire spinlock
        }
        avail = false;
        owner = currThread();    // set new owner
        lock.release();
    }
}
```



```

void release() {
    lock.acquire();
    if ( owner != currThread() ) ... // ERROR CHECK
    owner = nullptr; // no owner
    if ( ! blocked.empty() ) {
        // remove task from blocked list and make ready
    }
    avail = true; // reset
    lock.release(); // RACE
}
};

```

- yieldNoSchedule yields the processor time-slice but does not reschedule thread to ready queue.
- Single or multiple unblock for multiple acquisition?
- avail is necessary as queue can be empty but critical section occupied.
- Problem: **blocking occurs holding spin lock!**
- ⇒ release lock before blocking

```

// add self to blocked list of lock
lock.release(); // allow releasing task to unblock next waiting task
// PREEMPTION ⇒ put on ready queue
yieldNoSchedule();

```

- **Race between blocking and unblocking tasks.**
- Blocking task releases spin lock but preempted *before* yield and put onto ready queue.
- Unblocking task can enter, see blocking task on lock's blocked list, and put on ready queue.
- But task is still on the ready queue because of the preemption!
- Need *magic* to atomically yield without scheduling *and* release spin lock.
- Magic is often accomplished with more cooperation:
 

```
yieldNoSchedule( lock );
```
- Spin lock is passed to the runtime system, which does the yield without schedule and then, on behalf of the user thread, unlocks the lock.
- Alternative approach is park/unpark, where each thread blocks on a private binary semaphore (see Section 6.4.4.6, p. 134 private semaphore).
- Disabling and enabling interrupts is too expensive.
- Note, the runtime system violates order and speed of execution by being non-preemptable.

- Problem: avail and lock reset  $\Rightarrow$  acquiring tasks can **barge** ahead of released task.
- Released task must check again (**while**)  $\Rightarrow$  busy waiting  $\Rightarrow$  starvation
- **Barging avoidance** (cooperation): hold avail between releasing and unblocking task (bounded overtaking).

```

void acquire() {
    lock.acquire();           // barging
    if ( ! avail && owner != currThread() ) { // avoid barging
        // add self to lock' s blocked list
        yieldNoSchedule( lock ); // lock release
        lock.acquire();       // IN GENERAL, reacquire
    } else {
        avail = false;
    }
    owner = currThread(); // set new owner, may not be safe
    lock.release();
}

void release() {
    lock.acquire();
    owner = nullptr; // no owner
    if ( ! blocked.empty() ) {
        // remove task from blocked list and make ready
    } else {
        avail = true; // conditional reset
    }
    lock.release(); // RACE
}

```

- Bargers enter mutual-exclusion protocol but block so released task does not busy wait (**if** rather than **while**).
- Mutual exclusion is **conceptually passed** from releasing to unblocking tasks (baton passing).
- If signalled task does not reacquire the spinlock, there can be a race, e.g., owner is read/written simultaneously.
- Because a spinlock is unfair, an infinite stream of bargers can prevent the released task from acquiring the spinlock (short/long-term starvation).
- **Barging prevention** (cooperation): hold lock between releasing and unblocking task (unbounded overtaking).

```

void acquire() {
    lock.acquire();           // prevention barging
    if ( ! avail && owner != currThread() ) {
        // add self to lock' s blocked list
        yieldNoSchedule( lock );
        // DO NOT REACQUIRE LOCK
    } else avail = false;
    owner = currThread();    // set new owner
    lock.release();
}

void release() {
    lock.acquire();
    owner = nullptr;        // no owner
    if ( ! blocked.empty() ) {
        // remove task from blocked list and make ready
        // DO NOT RELEASE LOCK
    } else {
        avail = true;       // conditional reset
        lock.release();     // NO RACE
    }
}

```

- **Critical section is not bracketed by the spin lock when lock is passed.**
- Alternative (cooperation): leave lock owner at front of blocked list to act as availability and owner variable.

```

class MutexLock {
    queue<Task> blocked;      // blocked tasks
    SpinLock lock;          // nonblocking lock
public:
    void acquire() {
        lock.acquire();      // prevention barging
        if ( blocked.empty() ) { // no one waiting ?
            node.owner = currThread();
            // add self to lock's blocked list
        } else if ( blocked.head().owner != currThread() ) { // not owner ?
            // add self to lock' s blocked list
            yieldNoSchedule( lock );
            // DO NOT REACQUIRE LOCK
        }
        lock.release();
    }
}

```

```

void release() {
    lock.acquire();
    // REMOVE TASK FROM HEAD OF BLOCKED LIST
    if ( ! blocked.empty() ) {
        // MAKE TASK AT FRONT READY BUT DO NOT REMOVE
        // DO NOT RELEASE LOCK
    } else {
        lock.release();    // NO RACE
    }
}
};

```

- If critical section acquired, blocked list must have a node on it to check for in-use.

### 6.3.1.2 uOwnerLock

- $\mu$ C++ provides a multiple-acquisition mutex-lock, uOwnerLock:

```

class uOwnerLock {
public:
    uOwnerLock();
    uBaseTask * owner();
    unsigned int times();
    void acquire();
    bool tryacquire();
    void release();
};

```

- owner() returns **nullptr** if no owner, otherwise address of task that currently owns lock.
- times() returns number of times lock has been acquired by owner task.
- Must release as many times as acquire.
- Otherwise, operations same as for uLock but with blocking instead of spinning for acquire.

### 6.3.1.3 Mutex-Lock Release-Pattern

- To ensure a mutual exclusion lock is always released use the following patterns.

- executable statement – finally clause

```

uOwnerLock lock;
lock.acquire();
try {
    ...                // protected by lock
} _Finally {
    lock.release();
}

```

- allocation/deallocation (RAII – Resource Acquisition Is Initialization)

```

class RAII {           // create once
    uOwnerLock & lock;
public:
    RAII( uOwnerLock & lock ) : lock( lock ) { lock.acquire(); }
    ~RAII() { lock.release(); }
};
uOwnerLock lock;
{
    RAII raii( lock );    // lock acquired by constructor
    ...                  // protected by lock
}                        // lock release by destructor

```

- Lock always released on normal, local transfer (**break/return**), and exception.
- Cannot be used for barging prevention. Why?

#### 6.3.1.4 Stream Locks

- Specialized mutex lock for I/O based on uOwnerLock.
- Concurrent use of C++ streams can produce unpredictable results.
  - if two tasks execute:

```

task1 : cout << "abc " << "def " << endl;
task2 : cout << "uvw " << "xyz " << endl;

```

any of the outputs can appear:

abc def	abc uvw xyz	uvw abc xyz def	abuvwc dexf	uvw abc def
uvw xyz	def		yz	xyz

- $\mu$ C++ provides: `osacquire` for output streams and `isacquire` for input streams.
- Most common usage is to create an anonymous stream lock for a cascaded I/O expression:

```

task1 : osacquire( cout ) << "abc " << "def " << endl;
task2 : osacquire( cout ) << "uvw " << "xyz " << endl;

```

constraining the output to two different lines in either order:

abc def	uvw xyz
uvw xyz	abc def

- Multiple I/O statements can be protected using block structure:

```

{ // acquire the lock for stream cout for block duration
  osacquire acq( cout ); // named stream lock
  cout << "abc";
  osacquire( cout ) << "uvw " << "xyz " << endl; // OK?
  cout << "def";
} // implicitly release the lock when "acq" is deallocated

```

- Which *locking-release* pattern is used by stream locks?

### 6.3.2 Synchronization Lock

- **Synchronization lock** is used solely to block tasks waiting for synchronization.
- Weakest form of blocking lock as its only state is list of blocked tasks.
  - ⇒ **acquiring task always blocks** (no state to make it conditional)  
Need ability to yield time-slice and block versus yield and go back on ready queue.
  - ⇒ **release is lost when no waiting task** (no state to remember it)
- Often called a **condition lock**, with wait / signal(notify) for acquire / release.

#### 6.3.2.1 Implementation

- Like mutex lock, synchronization lock needs mutual exclusion for safe implementation.
- Location of mutual exclusion classifies synchronization lock:

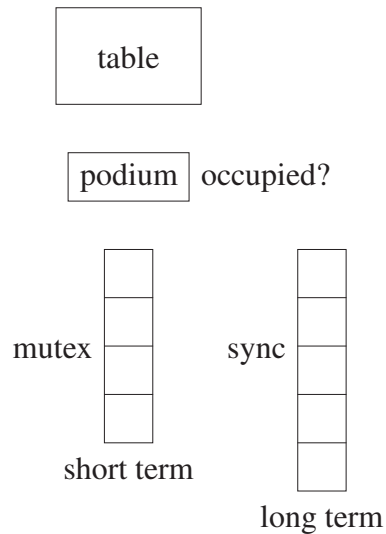
**external locking** use an external lock to protect task list,

**internal locking** use an internal lock to protect state (lock is extra state).

- external locking

```
class SyncLock {
    Task * list;
public:
    SyncLock() : list( nullptr ) {}
    void acquire() {
        // add self to task list
        yieldNoSchedule();
    }
    void release() {
        if ( list != nullptr ) {
            // remove task from blocked list and make ready
        }
    }
};
```

- Use external state to avoid lost release.
- Need mutual exclusion to protect task list and possible external state.
- Releasing task detects a blocked task and performs necessary cooperation.
- Usage pattern:
  - Cannot enter a restaurant if all tables are full.
  - Must acquire a lock to check for an empty table because state can change.
  - If no free table, block on waiting-list until a table becomes available or **leave** (balk) and eat somewhere else.



- Why is a single waiting queue (bench) inadequate?

```

// shared variables
MutexLock m;           // external mutex lock
SyncLock s;           // synchronization lock
bool occupied = false; // indicate if event has occurred
// acquiring task
m.acquire();           // mutual exclusion to examine state & possibly block
if ( occupied ) {     // event not occurred ?
    if ( /* do not wait */ ) { m.release(); /* go elsewhere */ }
    s.acquire();       // long-term block for event
    m.acquire();       // require mutual exclusion to set state
}
occupied = true;      // set
m.release();
... EAT! ...
// releasing task
m.acquire();           // mutual exclusion to examine state
occupied = false;     // reset
s.release();           // possibly unblock waiting task
m.release();           // release mutual exclusion

```

- **Blocking occurs holding external mutual-exclusion lock!**
- ⇒ release lock before blocking by modifying synchronization-lock acquire.

```

void SyncLock::acquire( MutexLock & m ) { // add parameter
    // add self to task list
    m.release(); // release external mutex-lock
    // PREEMPTION ⇒ put on ready queue
    yieldNoSchedule();
    // possibly reacquire mutexlock
}

```

- **As before, preemption results in race between blocking and unblocking tasks.**
- To prevent race, need to cooperate with scheduler.

```
void SyncLock::acquire( MutexLock & m ) {
    // add self to task list
    yieldNoSchedule( m ); // scheduler unlocks m
    // possibly reacquire mutexlock
}
```

- Or, protecting mutex-lock is bound at synchronization-lock creation and used implicitly.
- Now use first usage pattern.

```
// acquiring task
m.acquire(); // mutual exclusion to examine state & possibly block
if ( occupied ) { // event not occurred ?
    if ( /* do not wait */ ) { m.release(); /* go elsewhere */ }
    s.acquire( m ); // block for event and release mutex lock
    ...
}
```

- Has the race been prevented?
- Problem: barging can occur when releasing task resets occupied.
  - ⇒ non-FIFO order and possible starvation
- Note, same problems as inside mutex lock but occurring *outside* between mutex and synchronization locks.
- Use barging avoidance:

```
// releasing task
m.acquire(); // mutual exclusion to examine state
if ( ! s.empty() ) s.release(); // unblock, no reset
else occupied = false; // reset
m.release(); // release mutual exclusion
```

or prevention:

```
// releasing task
m.acquire(); // mutual exclusion to examine state
if ( ! s.empty() ) s.release(); // unblock, no reset
else { occupied = false; m.release(); } // reset & release
```

- internal locking



```

class SyncLock {
    Task * list;           // blocked tasks
    SpinLock lock;       // internal lock
public:
    SyncLock() : list( nullptr ) {}
    void acquire( MutexLock & m ) { // optional external lock
        lock.acquire();
        // add self to task list
        m.release(); // release external mutex-lock
        CAN BE INTERRUPTED HERE
        yieldNoSchedule( lock );
        m.acquire(); // possibly reacquire after blocking
    }

    void release() {
        lock.acquire();
        if ( list != nullptr ) {
            // remove task from blocked list and make ready
        }
        lock.release();
    }
};

```

- Why does acquire still take an external lock?
- Why is the race after releasing the external mutex-lock not a problem?
- Has the busy wait been removed from the blocking lock?

### 6.3.2.2 uCondLock

- $\mu$ C++ provides an internal synchronization-lock, uCondLock.

```

class uCondLock {
public:
    uCondLock();
    void wait( uOwnerLock & lock );
    bool signal();
    bool broadcast();
    bool empty();
};

```

- wait and signal are used to block a thread on and unblock a thread from the queue of a condition, respectively.
- wait atomically blocks the calling task and **releases argument owner-lock**.
- **wait reacquires its argument owner-lock before returning.**
- signal unblocks a single task in FIFO order.
- broadcast unblocks all waiting tasks.

- signal/broadcast do nothing for an empty condition and return false; otherwise, return true.
- empty returns **false** if blocked tasks on the queue and **true** otherwise.

### 6.3.2.3 Programming Pattern

- Using synchronization locks is complex because they are weak.
- Must provide external mutual-exclusion and protect against loss signal (release).
- Why is synchronization more complex for blocking locks than spinning (uLock)?

```

bool done = false;

_Task T1 {
    uOwnerLock & mlk;
    uCondLock & clk;

    void main() {
        mlk.acquire(); // prevent lost signal
        if ( ! done ) // signal occurred ?
            // signal not occurred
            clk.wait( mlk ); // atomic wait/release
            // mutex lock re-acquired after wait
            mlk.release(); // release either way
        S2;
    }
public:
    T1( uOwnerLock & mlk,
        uCondLock & clk ) :
        mlk(mlk), clk(clk) {}
};

int main() {
    uOwnerLock mlk;
    uCondLock clk;
    T1 t1( mlk, clk );
    T2 t2( mlk, clk );
}

```

```

_Task T2 {
    uOwnerLock & mlk;
    uCondLock & clk;

    void main() {
        S1;
        mlk.acquire(); // prevent lost signal
        done = true; // remember signal occurred
        clk.signal(); // signal lost if not waiting
        mlk.release();
    }
public:
    T2( uOwnerLock & mlk,
        uCondLock & clk ) :
        mlk(mlk), clk(clk) {}
};

```

### 6.3.3 Barrier

- A **barrier** coordinates a group of tasks performing a concurrent operation surrounded by sequential operations.
- Hence, a barrier is for synchronization and cannot build mutual exclusion.
- Two kinds of barrier: threads equal group size ( $T == G$ ) or threads greater than group size ( $T > G$ ).
- Unlike previous synchronization locks, a **barrier retains state about the events it manages**: number of tasks blocked on the barrier.
- Since manipulation of this state requires mutual exclusion, most barriers use internal locking.

- E.g., 3 tasks must execute a section of code in a particular order: S1, S2 and S3 must *all* execute before S5, S6 and S7.

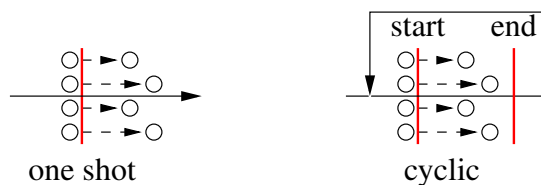
```

T1::main() {      T2::main() {      T3::main() {
    ...
    S1            S2            S3
    b.block();    b.block();    b.block();
    S5            S6            S7
    ...
}                }                }

int main() {
  Barrier b( 3 );
  T1 x( b );
  T2 y( b );
  T3 z( b );
}

```

- Barrier is initialized to control 3 tasks and passed to each task by reference (not copied).
- Barrier blocks each task at call to block until all tasks have called block.
- Last task to call block does not block and releases other tasks (cooperation).
- Hence, all tasks leave together (synchronized) after arriving at the barrier.
- Note, must specify in advance total number of block operations before tasks released.
- Two common uses for barriers:



```

Barrier start(G+1), end(G+1); // shared
Coordinator
// start G tasks so they can initialize
// general initialization
start.block(); // wait for threads to start
// do other work
end.block(); // wait for threads to end
// general close down and possibly loop

```

#### Workers

```

// initialize
start.block(); // wait for threads to start
// do work
end.block(); // wait for threads to end
// close down

```

- Two barriers allow Coordinator to accumulate results (subtotals) while Workers reinitialize (read next row).
- Alternative is last Worker does coordination, but prevents Workers reinitializing during coordination.
- Why not use termination synchronization and create new tasks for each computation?
  - creation and deletion of computation tasks is expensive

### 6.3.3.1 Fetch Increment Barrier

- spinning,  $T == G$ , flag ensures waiting threads exit barrier even if fast threads change count.

```

struct Barrier {
    size_t group = 0;
    volatile bool flag = false;
    volatile size_t count = 0;
};
void block( Barrier & b ) {
    size_t negflag = ! b.flag;
    if ( FetchInc( b.count, 1 ) < b.group - 1 ) {
        await( b.flag == negflag ); // spin
    } else {
        // SAFE ACTION BEFORE TRIGGERING BARRIER
        b.count = 0;
        b.flag = negflag;
    }
}

```

- Construct failure scenario for `await( b.count == 0 )`.

### 6.3.3.2 uBarrier

- $\mu$ C++ barrier is a blocking,  $T > G$ , barging-prevention coroutine, where the coroutine main can be resumed by the last task arriving at the barrier.

```

#include <uBarrier.h>
_Cormonitor uBarrier { // think _Coroutine
protected:
    void main() { for ( ;; ) suspend(); } // points of synchronization
    virtual void last() { resume(); } // called by last task to barrier
public:
    uBarrier( unsigned int total );
    unsigned int total() const; // # of tasks synchronizing
    unsigned int waiters() const; // # of waiting tasks
    void reset( unsigned int total ); // reset # tasks synchronizing
    virtual void block(); // wait for Gth thread, which calls last, unblocks waiting thread
};

```

- Member `last` is called by the  $G_{th}$  (last) task to the barrier, and then all blocked tasks are released.
- `uBarrier` has implicit mutual exclusion  $\Rightarrow$  no barging  $\Rightarrow$  only manages synchronization
- User barrier is built by:
  - inheriting from `uBarrier`
  - redefining `last` and/or `block` member and possibly coroutine main
  - possibly initializing `main` from constructor

- E.g., previous matrix sum (see page 80) adds subtotals in order of task termination, but barrier can add subtotals in order produced.

```

_Cormonitor Accumulator : public uBarrier {
    int total_ = 0, temp;
    uBaseTask * Gth_ = nullptr;
protected:
    void last() { // reset and remember Gth task
        temp = total_; total_ = 0;
        Gth_ = &uThisTask();
    }
public:
    Accumulator( int rows ) : uBarrier( rows ) {}
    void block( int subtotal ) {
        total_ += subtotal;
        uBarrier::block();
    }
    int total() { return temp; }
    uBaseTask * Gth() { return Gth_; }
};

_Task Adder {
    int * row, size;
    Accumulator & acc;
    void main() {
        int subtotal = 0;
        for ( unsigned int r = 0; r < size; r += 1 ) subtotal += row[r];
        acc.block( subtotal ); // provide subtotal; block for completion
    }
public:
    Adder( int row[], int size, Accumulator & acc ) :
        size( size ), row( row ), acc( acc ) {}
};

int main() {
    enum { rows = 10, cols = 10 };
    int matrix[rows][cols];
    Accumulator acc( rows ); // barrier synchronizes each summation
    // read matrix
    {
        uArray( Adder, adders, rows );
        for ( unsigned int r = 0; r < rows; r += 1 )
            adders[r]( matrix[r], cols, acc );
    } // wait adders
    cout << acc.total() << " " << acc.Gth() << endl;
}

```

- Why not have task delete itself after unblocking from `uBarrier::block()` and make program main the coordinator?

```

void block( int subtotal ) {
    total_ += subtotal; uBarrier::block();
    delete &uThisTask();
}
// program main
acc.block( 0 );

```

- Coroutine barrier can be reused many times, e.g., read in a new matrix in Accumulator::main after each summation.
- Why can a barrier not be used within a COFOR? (any implicit concurrency)

### 6.3.4 Binary Semaphore

- **Binary semaphore** (Edsger W. Dijkstra) is blocking equivalent to yielding spin-lock.
- Provides synchronization *and* mutual exclusion.

```
Semaphore lock(0); // 0 => closed, 1 => open, default 1
```

- More powerful than synchronization lock as it remembers state about an event.
- Names for acquire and release from Dutch terms
- acquire is P

- passeren  $\Rightarrow$  to pass
- prologen  $\Rightarrow$  (proberen) to try (verlagen) to decrease

```
lock.P(); // wait to enter
```

P waits if the semaphore counter is zero and then decrements it.

- release is V
- vrijgeven  $\Rightarrow$  to release
- verhogen  $\Rightarrow$  to increase

```
lock.V(); // release lock
```

V increases the counter and unblocks a waiting task (if present).

- When the semaphore has only two states (open/closed), it is called a **binary semaphore**.
- synchronization

```

_Task T1 {
    BinSem & lk;
    void main() {
        ...
        S1
        lk.V();
        ...
    }
public:
    T1( BinSem & lk ) : lk(lk) {}
};

```

```

_Task T2 {
    BinSem & lk;
    void main() {
        ...
        lk.P();
        S2
        ...
    }
public:
    T2( BinSem & lk ) : lk(lk) {}
};

```

```
int main() {
    BinSem lock( 0 ); // closed
    T1 t1( lock );
    T2 t2( lock );
}
```

- mutual exclusion

```
_Task T {
    BinSem & lk;
    void main() {
        ...
        lk.P();
        // critical section
        lk.V();
        ...
        lk.P();
        // critical section
        lk.V();
        ...
    }
public:
    T( BinSem & lk ) : lk(lk) {}
};
```

```
int main() {
    BinSem lock( 1 ); // start open
    T t0( lock ), t1( lock );
}
```

### 6.3.4.1 Implementation

- Implementation has:
  - blocking task-list
  - avail indicates if event has occurred (state)
  - spin lock to protect state

```
class BinSem {
    queue<Task> blocked; // blocked tasks
    bool avail; // resource available ?
    SpinLock lock; // mutex nonblocking lock
public:
    BinSem( bool start = true ) : avail( start ) {}
    void P() {
        lock.acquire(); // prevention barging
        if ( ! avail ) {
            // add self to lock' s blocked list
            yieldNoSchedule( lock );
            // DO NOT REACQUIRE LOCK
        }
        avail = false;
        lock.release();
    }
};
```

```

void V() {
    lock.acquire();
    if ( ! blocked.empty() ) {
        // remove task from blocked list and make ready
        // DO NOT RELEASE LOCK
    } else {
        avail = true;           // conditional reset
        lock.release();        // NO RACE
    }
}
};

```

- Same as single-acquisition mutexLock but can initialize avail.
- Higher cost for synchronization if external lock already acquired.
  - Might need P( S ) to atomically block and release the synchronization semaphore S.

### 6.3.5 Counting Semaphore

- Augment the definition of P and V to allow a multi-valued semaphore.
- What does it mean for a lock to have more than open/closed (unlocked/locked)?
  - $\Rightarrow$  critical sections allowing  $N$  simultaneous tasks.
- Augment V to allow increasing the counter an arbitrary amount.
- synchronization
  - Three tasks must execute so S2 and S3 only execute after S1 has completed.

```

T1::main() {      T2::main() {      T3::main() {
    ...           ...           S1
    lk.P();      lk.P();      lk.V(); // lk.V(2)
    S2          S3          lk.V();
    ...           ...           ...
}               }               }

int main() {
    CntSem lk( 0 ); // closed
    T1 x( lk );
    T2 y( lk );
    T3 z( lk );
}

```

- mutual exclusion
  - Critical section allowing up to 3 simultaneous tasks.



```

_Task T {
    CntSem & lk;
    void main() {
        ...
        lk.P();
        // up to 3 tasks in
        // critical section
        lk.V();
        ...
    }
public:
    T( CntSem & lk ) : lk(lk) {}
};

int main() {
    CntSem lk( 3 ); // allow 3
    T t0( lk ), t1( lk ), ...;
}

```

- Must know in advance the total number of P's on the semaphore.

### 6.3.5.1 Implementation

- Change availability into counter, and set to some maximum on creation.
- Decrement counter on acquire and increment on release.
- Block acquiring task when counter is 0.
- Negative counter indicates number of waiting tasks.

```

class CntSem {
    queue<Task> blocked; // blocked tasks
    int cnt; // resource being used ?
    SpinLock lock; // nonblocking lock
public:
    CntSem( int start = 1 ) : cnt( start ) {}

    void P() {
        lock.acquire();
        cnt -= 1;
        if ( cnt < 0 ) {
            // add self to lock's blocked list
            yieldNoSchedule( lock );
            // DO NOT REACQUIRE LOCK
        }
        lock.release();
    }

    void V() {
        lock.acquire();
        cnt += 1;
        if ( cnt <= 0 ) {
            // remove task from blocked list and make ready
            // DO NOT RELEASE LOCK
        } else {
            lock.release(); // NO RACE
        }
    }
};

```

- In general, binary/counting semaphores are used in two distinct ways:
  1. For synchronization, if the semaphore starts at 0  $\Rightarrow$  waiting for an event to occur.
  2. For mutual exclusion, if the semaphore starts at 1(N)  $\Rightarrow$  controls a critical section.
- $\mu\text{C++}$  provides a counting semaphore, `uSemaphore`, which subsumes a binary semaphore.

```
#include <uSemaphore.h>
class uSemaphore {
public:
    uSemaphore( unsigned int count = 1 );
    void P();
    bool TryP();
    void V( unsigned int times = 1 );
    int counter() const;
    bool empty() const;
};
```

- P decrements the semaphore counter; if the counter is greater than or equal to zero, the calling task continues, otherwise it blocks.
- TryP returns **true** if the semaphore is acquired and **false** otherwise (never blocks).
- V wakes up the task blocked for the longest time if there are tasks blocked on the semaphore and increments the semaphore counter.
- If V is passed a positive integer N, the semaphore is V ed N times.
- The member routine counter returns the value of the semaphore counter:
  - negative means  $\text{abs}(N)$  tasks are blocked waiting to acquire the semaphore, and the semaphore is locked;
  - zero means no tasks are waiting to acquire the semaphore, and the semaphore is locked;
  - positive means the semaphore is unlocked and allows  $N$  tasks to acquire the semaphore.
- The member routine empty returns **false** if there are threads blocked on the semaphore and **true** otherwise.

## 6.4 Lock Programming

### 6.4.1 Precedence Graph

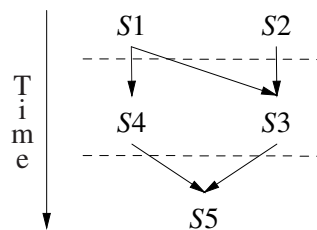
- Binary P and V in with COBEGIN are as powerful as START and WAIT.
- E.g., execute statements so the result is the same as serial execution but concurrency is maximized.

```

S1: a := 1
S2: b := 2
S3: c := a + b
S4: d := 2 * a
S5: e := c + d

```

- Analyse which data and code depend on each other.
- i.e., statement S1 and S2 are independent  $\Rightarrow$  can execute in either order or at the same time.
- Statement S3 is dependent on S1 and S2 because it uses both results.
- Display dependencies graphically in a **precedence graph** (different from process graph).

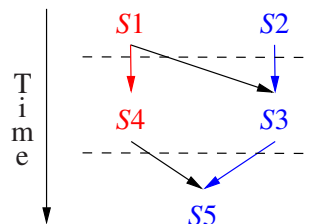


```

Semaphore L1(0), L2(0), L3(0), L4(0);
COBEGIN
  BEGIN a := 1; V(L1); END;
  BEGIN b := 2; V(L2); END;
  BEGIN P(L1); P(L2); c := a + b; V(L3); END;
  BEGIN P(L1); d := 2 * a; V(L4); END;
  BEGIN P(L3); P(L4); e := c + d; END;
COEND

```

- Does this solution work?
- Optimal solution: minimum threads, M, and traverse M paths through precedence graph.

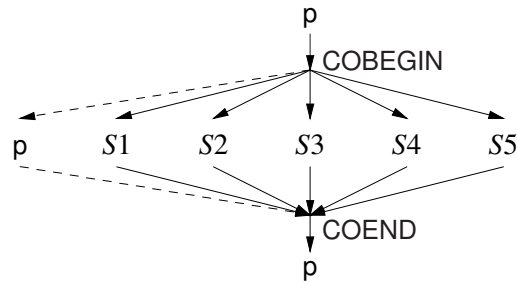


```

Semaphore L1(0), L2(0);
COBEGIN
  BEGIN a := 1; V(L1); d := 2 * a; V(L2); END;
  BEGIN b := 2; P(L1); c := a + b; P(L2); e := c + d; END;
COEND

```

- process graph (different from precedence graph)

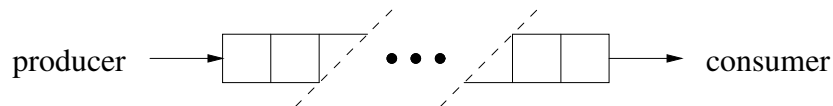


## 6.4.2 Buffering

- Tasks communicate unidirectionally through a queue.
- Producer adds items to the back of a queue.
- Consumer removes items from the front of a queue.

### 6.4.2.1 Unbounded Buffer

- Two tasks communicate through a queue of unbounded length.



- Because tasks work at different speeds, producer may get ahead of consumer.
  - Producer never has to wait as buffer has infinite length.
  - Consumer has to wait if buffer is empty  $\Rightarrow$  wait for producer to add.
- Queue is shared between producer/consumer, and counting semaphore controls access.

```

#define QueueSize  $\infty$ 
int front = 0, back = 0;
int Elements[QueueSize];
uSemaphore full(0);
void Producer::main() {
    for (;;) {
        // produce an item
        // add to back of queue
        full.V();
    }
    // produce a stopping value
    full.V();
}

```

```

void Consumer::main() {
    for (;;) {
        full.P();
        // take an item from the front of the queue
        if ( stopping value ? ) break;
        // process or consume the item
    }
}

```

- Is there a problem adding and removing items from the shared queue?
- Is the full semaphore used for mutual exclusion or synchronization?

#### 6.4.2.2 Bounded Buffer

- Two tasks communicate through a queue of bounded length.
- Because of bounded length:
  - Producer has to wait if buffer is full  $\Rightarrow$  wait for consumer to remove.
  - Consumer has to wait if buffer is empty  $\Rightarrow$  wait for producer to add.
- Use counting semaphores to account for the finite length of the shared queue.

```

uSemaphore full(0), empty(QueueSize);
void Producer::main() {
    for ( ;; ) {
        // produce an item
        empty.P();
        // add element to buffer
        full.V();
    }
    // produce a stopping value
    full.V();
}
void Consumer::main() {
    for ( ;; ) {
        full.P();
        // remove element from buffer
        if ( stopping value ? ) break;
        // process or consume the item
        empty.V();
    }
}

```

- Does this produce maximum concurrency?
- Can it handle multiple producers/consumers?

34	13	9	10	-3
----	----	---	----	----

full	empty
<del>0</del>	<del>5</del>
<del>1</del>	<del>4</del>
<del>2</del>	<del>3</del>
<del>3</del>	<del>2</del>
<del>4</del>	<del>1</del>
5	0

### 6.4.3 Lock Techniques

- Many possible solutions; need systematic approach.
- A **split binary semaphore** is a collection of semaphores where at most one of the collection has the value 1.
  - I.e., the sum of the semaphores is always less than or equal to one.
  - Used when different kinds of tasks have to block separately.
  - Cannot differentiate tasks blocked on the same semaphore (condition) lock. Why?
  - E.g., A and B tasks block on different semaphores so they can be unblocked based on kind, but collectively manage 2 semaphores like it was one.
- Split binary semaphores can be used to solve complicated mutual-exclusion problems by a technique called **baton passing**.
- The rules of baton passing are:
  - there is exactly one (conceptual) baton
  - nobody moves in the entry/exit code unless they have it
  - once the baton is released, cannot read/write variables in entry/exit
- E.g., baton is conceptually acquired in entry/exit protocol and passed from signaller to signalled task (see page 106).

```

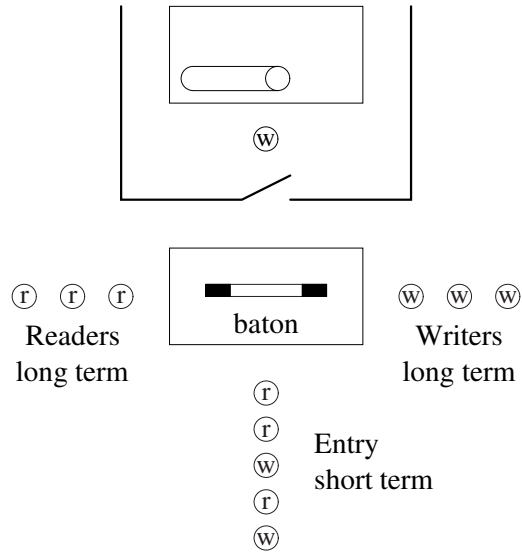
class BinSem {
    queue<Task> blocked;
    bool avail;
    SpinLock lock;
public:
    BinSem( bool start = true ) : avail( start ) {}
    void P() {
        lock.acquire(); PICKUP BATON, CAN ACCESS STATE
        if ( ! avail ) {
            // add self to lock' s blocked list
            PUT DOWN BATON, CANNOT ACCESS STATE
            yieldNoSchedule( lock );
            // UNBLOCK WITH SPIN LOCK ACQUIRED
            PASSED BATON, CAN ACCESS STATE
        }
        avail = false;
        lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
    }
    void V() {
        lock.acquire(); PICKUP BATON, CAN ACCESS STATE
        if ( ! blocked.empty() ) {
            // remove task from blocked list and make ready
            PASS BATON, CANNOT ACCESS STATE
        } else {
            avail = true;
            lock.release(); PUT DOWN BATON, CANNOT ACCESS STATE
        }
    }
};

```

- Can mutex/condition lock perform baton passing to prevent barging?
  - Not if signalled task must implicitly re-acquire the mutex lock before continuing.
  - ⇒ signaller must release the mutex lock.
  - There is now a race between signalled and calling tasks, resulting in barging.

#### 6.4.4 Readers and Writer Problem

- Multiple tasks sharing a resource: some reading the resource and some writing the resource.
- Allow multiple concurrent reader tasks simultaneous access, but serialize access for writer tasks (a writer may read).
- Use split-binary semaphore to segregate 3 kinds of tasks: arrivers, readers, writers.
- Use baton-passing to help understand complexity.



### 6.4.4.1 Solution 1

```

uSemaphore entry(1), rwait(0), wwait(0); // split binary semaphores
int rdel = 0, wdel = 0, rcnt = 0, wcnt = 0; // auxiliary counters
void Reader::main() {
    entry.P(); // pickup baton
    if ( wcnt > 0 ) { // occupied ?
        rdel += 1; entry.V(); // put baton down
        rwait.P(); rdel -= 1; // passed baton
    }
    rcnt += 1;
    if ( rdel > 0 ) { // waiting readers ?
        rwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
    // READ
    entry.P(); // pickup baton
    rcnt -= 1;
    if ( rcnt == 0 && wdel > 0 ) { // waiting writers ?
        wwait.V(); // pass baton
    } else {
        entry.V(); // put baton down
    }
}

```



```

void Writer::main() {
    entry.P();                // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) { // occupied ?
        wdel += 1; entry.V(); // put baton down
        wwait.P(); wdel -= 1; // passed baton
    }
    wcnt += 1;
    entry.V();                // put baton down
    // WRITE
    entry.P();                // pickup baton
    wcnt -= 1;
    if ( rdel > 0 ) {         // waiting readers ?
        rwait.V();           // pass baton
    } else if ( wdel > 0 ) { // waiting writers ?
        wwait.V();           // pass baton
    } else {
        entry.V();           // put baton down
    }
}

```

- Problem: reader only checks for writer in resource, never writers waiting to use it.
  - ⇒ readers barge ahead of writers who already waited.
  - ⇒ continuous stream of readers (actually only 2 needed) prevent waiting writers from making progress (starvation).

#### 6.4.4.2 Solution 2

- Give writers priority and make the readers wait.
  - Works most of the time because normally 80% readers and 20% writers.
- Change entry protocol for reader to the following:

```

entry.P();                // pickup baton
if ( wcnt > 0 || wdel > 0 ) { // waiting writers?
    rdel += 1; entry.V(); // put baton down
    rwait.P(); rdel -= 1; // passed baton
}
rcnt += 1;
if ( rdel > 0 ) {         // waiting readers ?
    rwait.V();           // pass baton
} else {
    entry.V();           // put baton down
}

```

- Also, change writer's exit protocol to favour writers:

```

entry.P();                // pickup baton
wcnt -= 1;
if ( wdel > 0 ) {        // check writers first
    rwait.V();           // pass baton
} else if ( rdel > 0 ) {
    rwait.V();           // pass baton
} else {
    entry.V();           // put baton down
}

```

- ⇒ writers barge.
- ⇒ continuous stream of writers cause reader starvation.

#### 6.4.4.3 Solution 3

- Fairness on simultaneous arrival is solved by alternation (Dekker's solution).
- E.g., use last flag to indicate the kind of tasks last using the resource, i.e., reader or writer.
- On exit, first select from opposite kind, e.g., if last is reader, first check for waiting writer otherwise waiting reader, then update last.
- Flag is unnecessary if readers wait when there is a waiting writer, and all readers started after a writer.
- ⇒ put writer's exit-protocol back to favour readers.

```

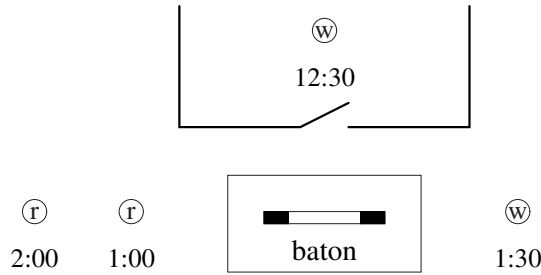
entry.P();                // pickup baton
wcnt -= 1;
if ( rdel > 0 ) {        // check readers first
    rwait.V();           // pass baton
} else if ( wdel > 0 ) {
    wwait.V();           // pass baton
} else {
    entry.V();           // put baton down
}

```

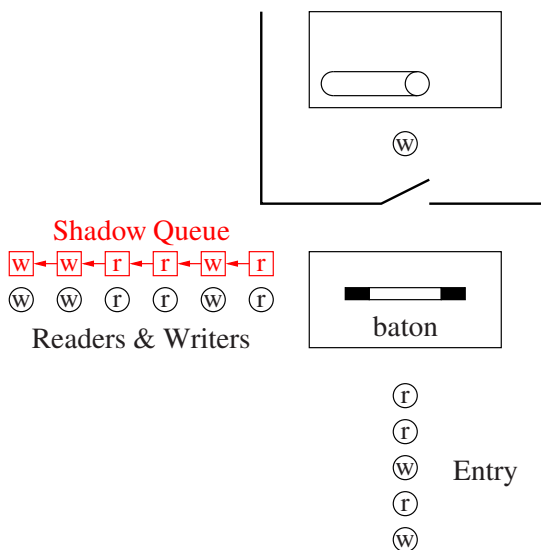
- **Arriving** readers cannot barge ahead of waiting writers and **unblocking** writers cannot barge ahead of a waiting reader
- ⇒ alternation for simultaneous waiting.

#### 6.4.4.4 Solution 4

- Problem: temporal barging!
- Staleness/freshness for last flag and staleness with no-flag.



- Alternation for simultaneous waiting means when writer leaves resource:
  - both readers enter  $\Rightarrow$  2:00 reader reads data that is **stale**; should read 1:30 write
  - writer enters and overwrites 12:30 data (never seen)  $\Rightarrow$  1:00 reader reads data that is too **fresh** (i.e., missed reading 12:30 data)
- Staleness/freshness can lead to plane or stock-market crash.
- Service readers and writers in **temporal order**, i.e., first-in first-out (FIFO), but allow multiple concurrent readers.
- Have readers and writers wait on same semaphore  $\Rightarrow$  collapse split binary semaphore.
- **But now lose kind of waiting task!**
- Introduce shadow queue to retain kind of waiting task on semaphore:



```

uSemaphore entry(1), rwwait(0); // readers/writers, temporal order
int rwdel = 0, rcnt = 0, wcnt = 0; // auxiliary counters
enum RW { READER, WRITER }; // kinds of tasks
queue<RW> rw_id; // queue of kinds
void Reader::main() {
    entry.P(); // pickup baton
    if ( wcnt > 0 || rwdel > 0 ) { // anybody waiting?
        rw_id.push( READER ); // store kind
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
        rw_id.pop();
    }
    rcnt += 1;
    if ( rwdel > 0 && rw_id.front() == READER ) { // more readers ?
        rwwait.V(); // pass baton
    } else
        entry.V(); // put baton down
    // READ
    entry.P(); // exit protocol
    rcnt -= 1;
    if ( rcnt == 0 && rwdel > 0 ) { // last reader ?
        rwwait.V(); // pass baton
    } else
        entry.V(); // put baton down
}

void Writer::main() {
    entry.P(); // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) {
        rw_id.push( WRITER ); // store kind
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
        rw_id.pop();
    }
    wcnt += 1;
    entry.V(); // put baton down
    // WRITE
    entry.P(); // pickup baton
    wcnt -= 1;
    if ( rwdel > 0 ) { // anyone waiting ?
        rwwait.V(); // pass baton
    } else
        entry.V(); // put baton down
}

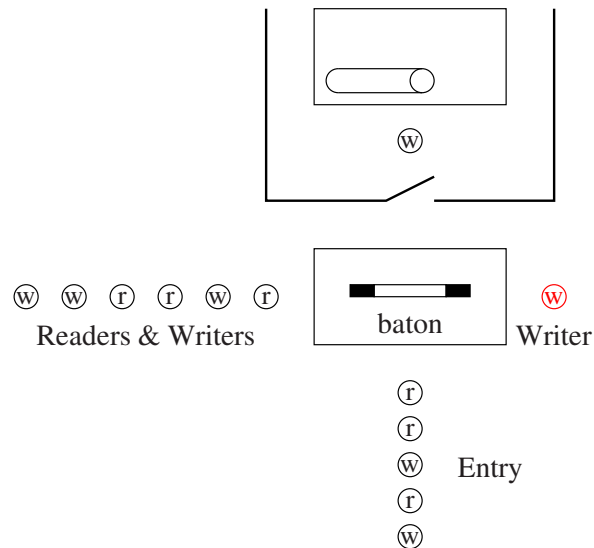
```

- Why can task pop *front* node on shadow queue when unblocked?

#### 6.4.4.5 Solution 5

- Cheat on cooperation:
  - allow 2 checks for write instead of 1
  - use reader/writer bench and writer chair.

- On exit, if chair empty, unconditionally unblock task at front of reader/writer semaphore.
- ⇒ **reader can incorrectly unblock a writer.**
- This writer now waits second time but in chair.
- Chair is always checked first on exit (higher priority than bench).



```

uSemaphore entry(1), rwait(0), wwait(0);
int rwdel = 0, wdel = 0, rcnt = 0, wcnt = 0; // auxiliary counters
void Reader::main() {
    entry.P(); // pickup baton
    if ( wcnt > 0 || wdel > 0 || rwdel > 0 ) {
        rwdel += 1; entry.V(); rwait.P(); rwdel -= 1;
    }
    rcnt += 1;
    if ( rwdel > 0 ) { // more readers ?
        rwait.V(); // pass baton
    } else
        entry.V(); // put baton down
    // READ
    entry.P(); // pickup baton
    rcnt -= 1;
    if ( rcnt == 0 ) { // last reader ?
        if ( wdel != 0 ) { // writer waiting ?
            wwait.V(); // pass baton
        } else if ( rwdel > 0 ) { // anyone waiting ?
            rwait.V(); // pass baton
        } else
            entry.V(); // put baton down
    } else
        entry.V(); // put baton down
}

```

```

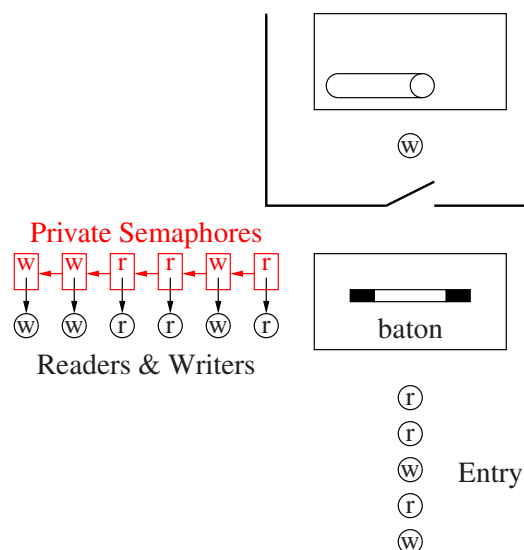
void Writer::main() {
    entry.P();                // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) { // first wait ?
        rwdel += 1; entry.V(); rwwait.P(); rwdel -= 1;
        if ( rcnt > 0 ) { // second wait ?
            wdel += 1; entry.V(); wwait.P(); wdel -= 1;
        }
    }
    wcnt += 1;
    entry.V();                // put baton down
    // WRITE
    entry.P();                // pickup baton
    wcnt -= 1;
    if ( rwdel > 0 ) { // anyone waiting ?
        rwwait.V(); // pass baton
    } else
        entry.V(); // put baton down
}

```

#### 6.4.4.6 Solution 6

- Still temporal problem when tasks move from one blocking list to another.
- In solutions, reader/writer entry-protocols have code sequence:
  - ... entry.V(); **INTERRUPTED HERE** Xwait.P();
- For writer:
  - pick up baton and see readers using resource
  - put baton down, entry.V(), but time-sliced before wait, Xwait.P().
  - another writer does same thing, and this can occur to any depth.
  - writers restart in any order or immediately have another time-slice
  - e.g., 2:00 writer goes ahead of 1:00 writer ⇒ freshness problem.
- For reader:
  - pick up baton and see writer using resource
  - put baton down, entry.V(), but time-sliced before wait, Xwait.P().
  - writers that arrived ahead of reader do same thing
  - reader restarts before any writers
  - e.g., 2:00 reader goes ahead of 1:00 writer ⇒ staleness problem.
- Need atomic block and release ⇒ magic like turning off time-slicing.
  - Xwait.P( entry ); // uC++ semaphore

- Alternative: ticket
  - readers/writers take ticket (see Section 5.18.9, p. 94) before putting baton down
  - to pass baton, serving counter is incremented and then **WAKE ALL BLOCKED TASKS**
  - each task checks ticket with serving value, and one proceeds while others reblock
  - starvation not an issue as waiting queue is bounded length, but inefficient
  
- Alternative: private semaphore
  - list of **private semaphores**, one for each waiting task, versus multiple waiting tasks on a semaphore.
  - add list node before releasing entry lock, which establishes position, then block on private semaphore.
  - to pass baton, private semaphore at head of the queue is V-ed, if present.
  - if task blocked on private semaphore, it is unblocked
  - if task not blocked due to time-slice, V is remembered, and task does not block on P.



```

uSemaphore entry(1);
int rwdel = 0, rcnt = 0, wcnt = 0;
struct RWnode {
    RW rw; // kinds of task
    uSemaphore sem; // private semaphore
    RWnode( RW rw ) : rw(rw), sem(0) {}
};
queue<RWnode *> rw_id;
void Reader::main() {
    entry.P(); // pickup baton
    if ( wcnt > 0 || ! rw_id.empty() ) { // anybody waiting?
        RWnode r( READER );
        rw_id.push( &r ); // store kind
        rwdel += 1; entry.V(); r.sem.P(); rwdel -= 1;
        rw_id.pop();
    }
    rcnt += 1;
    if ( rwdel > 0 && rw_id.front()->rw == READER ) { // more readers ?
        rw_id.front()->sem.V(); // pass baton
    } else
        entry.V(); // put baton down
    // READ
    entry.P(); // pickup baton
    rcnt -= 1;
    if ( rcnt == 0 && rwdel > 0 ) { // last reader ?
        rw_id.front()->sem.V(); // pass baton
    } else
        entry.V(); // put baton down
}

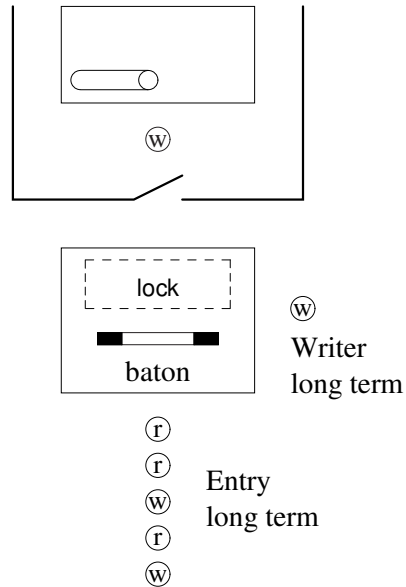
void Writer::main() {
    entry.P(); // pickup baton
    if ( rcnt > 0 || wcnt > 0 ) { // resource in use ?
        RWnode w( WRITER );
        rw_id.push( &w ); // remember kind of task
        rwdel += 1; entry.V(); w.sem.P(); rwdel -= 1;
        rw_id.pop();
    }
    wcnt += 1;
    entry.V();
    // WRITE
    entry.P(); // pickup baton
    wcnt -= 1;
    if ( rwdel > 0 ) { // anyone waiting ?
        rw_id.front()->sem.V(); // pass baton
    } else
        entry.V(); // put baton down
}

```

#### 6.4.4.7 Solution 7

- Ad hoc solution with questionable split-binary semaphores and baton-passing.





- Tasks wait in temporal order on entry semaphore.
- Only one writer ever waits on the writer chair until readers leave resource.
- **Waiting writer blocks holding baton to force other arriving tasks to wait on entry.**
- Semaphore lock is used only for mutual exclusion.
- Sometimes acquire two locks to prevent tasks entering and leaving.
- Release in opposite order.

```

uSemaphore entry(1);           // two locks open
uSemaphore lock(1), wwait(0);
int rcnt = 0, wdel = 0;

void Reader::main() {
    entry.P();                 // entry protocol
    lock.P();
    rcnt += 1;
    lock.V();
    entry.V();                 // put baton down
    // READ
    lock.P();                 // exit protocol
    rcnt -= 1;                 // critical section
    if ( rcnt == 0 && wdel == 1 ) { // last reader & writer waiting ?
        lock.V();
        wwait.V();           // pass baton
    } else
        lock.V();
}

```

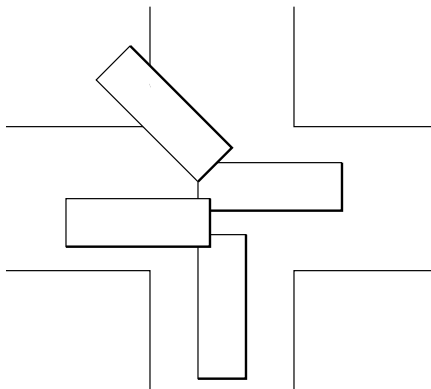
```
void Writer::main() {
    entry.P();           // entry protocol
    lock.P();
    if ( rcnt > 0 ) {   // readers waiting ?
        wdel += 1;
        lock.V();
        wwait.P();     // wait for readers
        wdel -= 1;     // unblock with baton
    } else
        lock.V();
    // WRITE
    entry.V();         // exit protocol
}
```

- Is temporal order preserved?
- While solution is smaller, harder to reason about correctness.
- Does not generalize for other kinds of complex synchronization and mutual exclusion.

## 7 Concurrent Errors

### 7.1 Race Condition

- A **race condition** occurs when there is missing:
  - synchronization
  - mutual exclusion

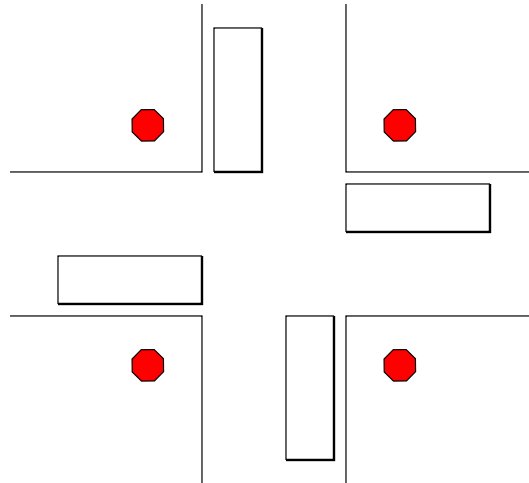


- Two or more tasks race along assuming synchronization or mutual exclusion has occurred.
- Can be very difficult to locate (thought experiments).
  - Aug. 14, 2003 Northeastern blackout : worst power outage in North American history.
  - Race condition buried in four million lines of C code.
  - “in excess of three million online operational hours in which nothing had ever exercised that bug.”

### 7.2 No Progress

#### 7.2.1 Live-lock

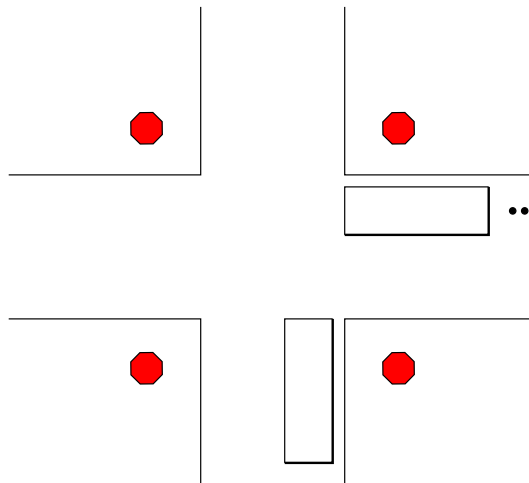
- Indefinite postponement: “You go first” problem on simultaneous arrival (symptom consuming CPU).
- Caused by poor scheduling in entry protocol. (Oracle with cardboard test)



- Always mechanism to break tie on simultaneous arrival to deal with live-lock.

### 7.2.2 Starvation

- A selection algorithm ignores one or more tasks so they are never executed, i.e., lack of long-term fairness (symptom consuming CPU).
- Long-term (infinite) starvation is extremely rare, but short-term starvation can occur and is a problem.



- Like live-lock, starving task might be ready at any time, switching among active, ready and possibly blocked states.

### 7.2.3 Deadlock

- **Deadlock** is the state when one or more processes are waiting for an event that will not occur.
- Unlike live-lock/starvation, deadlocked task is (usually) blocked so not consuming CPU.



### 7.3 Deadlock Prevention

- Eliminate one or more of the conditions required for a deadlock from an algorithm  $\Rightarrow$  deadlock can never occur.

#### 7.3.1 Synchronization Prevention

- Eliminate all synchronization from a program
- $\Rightarrow$  no communication
- $\Rightarrow$  impossible in most cases

#### 7.3.2 Mutual Exclusion Prevention

- Deadlock can be prevented by eliminating one of the 5 conditions:

##### 1. no mutual exclusion

- $\Rightarrow$  no shared resources
- $\Rightarrow$  impossible in most cases

##### 2. no hold & wait: do not give any resource, unless all resources can be given

```

uSemaphore L1(1), L2(1);           // open
task1 task2
L1.P() L2.P()      L1.P() L2.P()    // acquire all locks at start
  R1          R2          R2          R1    // access resource
          R1 & R2          R2 & R1    // access resources

```

- $\Rightarrow$  poor resource utilization
- possible starvation

##### 3. allow preemption

- Preemption is dynamic  $\Rightarrow$  cannot apply statically.

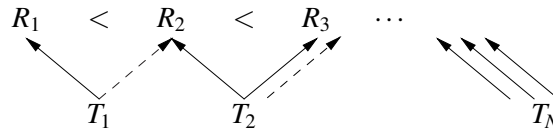
##### 4. no circular wait: by controlling order of resource allocations

```

uSemaphore L1(1), L2(1);           // open
task1 task2
L1.P()          L1.P()          // acquire same locks
  R1          // access resource
  L2.P()          L2.P()          // acquire same locks
          R2          // access resource
          R2 & R1 // access resources

```

- Use an **ordered resource** policy:



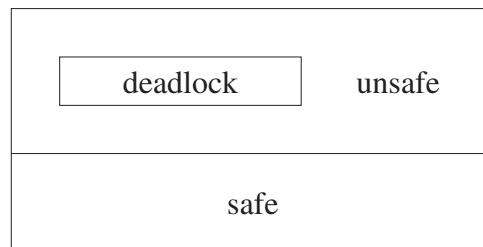
- divide all resources into classes  $R_1, R_2, R_3$ , etc.
- rule: can only request a resource from class  $R_i$  if holding no resources from any class  $R_j$  for  $j \geq i$
- unless each class contains only one resource, requires requesting several resources simultaneously
- denote the highest class number for which  $T$  holds a resource by  $h(T)$
- if process  $T_1$  is requesting a resource of class  $k$  and is blocked because that resource is held by process  $T_2$ , then  $h(T_1) < k \leq h(T_2)$
- as the preceding inequality is strict, a circular wait is impossible
- in some cases there is a natural division of resources into classes that makes this policy work nicely
- in other cases, some processes are forced to acquire resources in an unnatural sequence, complicating their code and producing poor resource utilization

5. prevent simultaneous occurrence:

- Show previous 4 rules cannot occur simultaneously.

## 7.4 Deadlock Avoidance

- Monitor all lock blocking and resource allocation to detect any potential formation of deadlock.



- Achieve better resource utilization, but additional overhead to avoid deadlock.

### 7.4.1 Banker's Algorithm

- Demonstrate a safe sequence of resource allocations that  $\Rightarrow$  no deadlock.
- However, requires a process state its maximum resource needs.

	R1	R2	R3	R4	
	6	12	4	2	total resources (TR)
T1	4	10	1	1	maximum needed
T2	2	4	1	2	for execution
T3	5	9	0	1	(M)
T1	3	5	1	0	currently
T2	1	2	1	0	allocated
T3	1	2	0	0	(C)

resource request (T1, R1)  $2 \rightarrow 3$

T1	1	5	0	1	needed to
T2	1	2	0	2	execute
T3	4	7	0	1	( $N = M - C$ )

- Is there a safe order of execution that avoids deadlock should each process require its maximum resource allocation?

current available resources

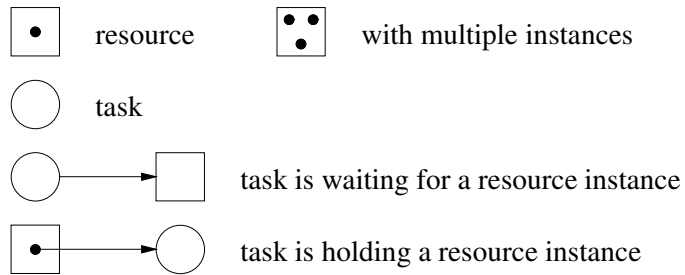
	1	3	2	2	( $CR = TR - \sum C_{cols}$ )
T2	0	1	2	0	( $CR = CR - N_{T2}$ )
	2	5	3	2	( $CR = CR + M_{T2}$ )
T1	1	0	3	1	( $CR = CR - N_{T1}$ )
	5	10	4	2	( $CR = CR + M_{T1}$ )
T3	1	3	4	1	( $CR = CR - N_{T3}$ )
	6	12	4	2	( $CR = CR + M_{T3}$ )

- So a safe order exists (the left column in the table above) and hence the Banker's Algorithm allows the resource request.
- If there is a choice of processes to choose for execution, it does not matter which path is taken.
- Example: If T1 or T3 could go to their maximum with the current resources, then choose either. A safe order starting with T1 exists if and only if a safe order starting with T3 exists.
- Does task scheduling need to be adjusted to the safe sequence?
- The check for a safe order can be performed for every allocation of resource to a process (optimizations are possible, i.e., same thread asks for another resource).

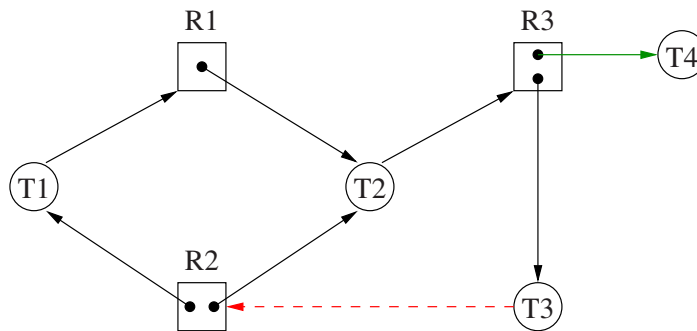
## 7.4.2 Allocation Graphs

- One method to check for potential deadlock is to graph processes and resource usage at each moment a resource is allocated.





- Multiple instances are put into a resource so that a specific resource does not have to be requested. Instead, a generic request is made.



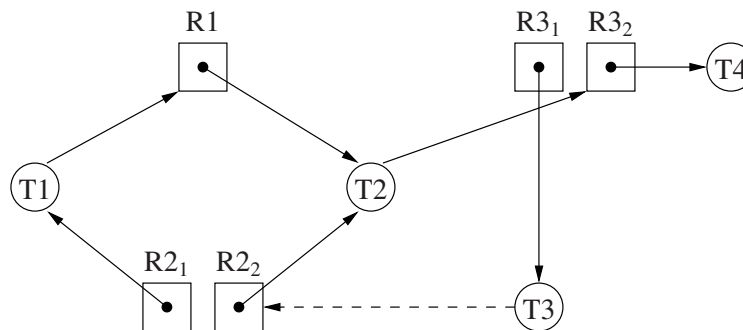
- If a graph contains no cycles, no process in the system is deadlocked.
- If any resource has several instances, a cycle  $\not\Rightarrow$  deadlock.

T1 → R1 → T2 → R3 → T3 → R2 → T1 (cycle)

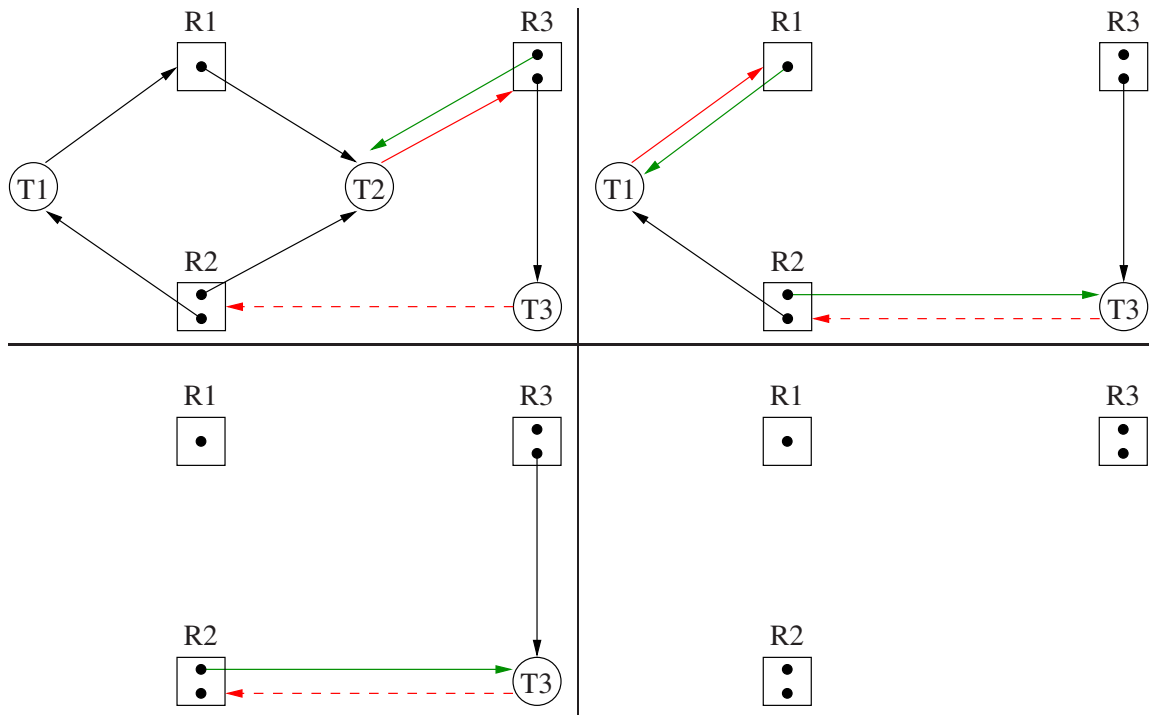
T2 → R3 → T3 → R2 → T2 (cycle)

- If T4 releases its resource, the cycle is broken.

- Create **isomorphic graph** without multiple instances (expensive and difficult):



- If each resource has one instance, a cycle  $\Rightarrow$  deadlock.
- Use **graph reduction** to locate deadlocks:



- Problems:
  - When choices for tasks, selection is tricky (like isomorphic graph).
  - For large graphs, detecting cycles is expensive.
  - Many graphs to examine over time, one for each particular allocation state of the system.

## 7.5 Detection and Recovery

- Instead of avoiding deadlock let it happen and recover.
  - $\Rightarrow$  ability to discover deadlock
  - $\Rightarrow$  preemption
- Discovering deadlock is difficult, e.g., build and check for cycles in allocation graph.
  - not on each resource allocation, but every T seconds or every time a resource cannot be immediately allocated
  - **Try  $\mu\text{C++}$  debugging macros to locate deadlock.**
- Recovery involves preemption of one or more processes in a cycle.
  - decision is not easy and must prevent starvation
  - The preemption victim must be restarted, from beginning or some previous checkpoint state, if you cannot guarantee all resources have not changed.
  - even that is not enough as the victim may have made changes before the preemption.

## 7.6 Which Method To Chose?

- Maybe “none of the above”: just ignore the problem
  - if some process is blocked for rather a long time, assume it is deadlocked and abort it
  - do this automatically in transaction-processing systems, manually elsewhere
- Of the techniques studied, only the ordered resource policy turns out to have much practical value.



## 8 Indirect Communication

- P and V are low level primitives for protecting critical sections and establishing synchronization between tasks.
- Shared variables provide the actual information that is communicated.
- Both of these can be complicated to use and may be incorrectly placed.
- Split-binary semaphores and baton passing are complex.
- Need higher level facilities that perform some of these details automatically.
- Get help from programming-language/compiler.

### 8.1 Critical Regions

- Declare which variables are to be shared, as in:

```
VAR v : SHARED INTEGER      MutexLock v_lock;
```

- Access to shared variables is restricted to within a REGION statement, and within the region, mutual exclusion is guaranteed.

```
REGION v DO                v_lock.acquire()
    // critical section      ... // x = v; (read)  v = y (write)
END REGION                v_lock.release()
```

- Simultaneous reads are impossible!
- Modify to allow reading of shared variables outside the critical region and modifications in the region.
- Problem: reading partially updated information while a task is updating the shared variable in the region.
- Nesting can result in deadlock.

```
VAR x, y : SHARED INTEGER

task1                task2
REGION x DO          REGION y DO
    ...              ...
    REGION y DO      REGION x DO
    ...              ...
    END REGION       END REGION
    ...              ...
END REGION           END REGION
```

## 8.2 Conditional Critical Regions

- Introduce a condition that must be true as well as having mutual exclusion.

```

REGION v DO
    AWAIT conditional-expression
    ...
END REGION

```

- E.g., The consumer from the producer-consumer problem.

```

VAR Q : SHARED QUEUE<INT,10>

REGION Q DO
    AWAIT NOT EMPTY( Q ) buffer not empty
    take an item from the front of the queue
END REGION

```

- If the condition is false, the region lock is released and entry is started again (busy waiting).
- To prevent busy waiting, block on queue for shared variable, and on region exit, search for true conditional-expression and unblock.

## 8.3 Monitor

- A **monitor** is an abstract data type that combines shared data with serialization of its modification.

```

_Monitor name {
    shared data
    members that see and modify the data
};

```

- A **mutex member** (short for mutual-exclusion member) is one that does NOT begin execution if there is another active mutex member.
  - $\Rightarrow$  a call to a mutex member may become blocked waiting entry, and queues of waiting tasks may form.
  - Public member routines of a monitor are implicitly mutex and other kinds of members can be made explicitly mutex with qualifier (**\_Mutex**).
- Basically each monitor has a lock which is Ped on entry to a monitor member and Ved on exit.

```

class Mon {
    MutexLock mlock;
    int v;           // shared data
public:
    int x(...) {    // mutex member
        mlock.acquire();
        ...        // int temp = v;
        mlock.release();
        return v;  // return temp;
    }
};

```

- Recursive entry is allowed (owner mutex lock), i.e., one mutex member can call another or itself.
- Unhandled exceptions raised within a monitor should always release the implicit monitor locks so the monitor can continue to function.
- Destructor must be mutex, so ending a block with a monitor or deleting a dynamically allocated monitor, blocks if thread in monitor.
- Atomic counter using a monitor:

```

_Monitor AtomicCounter {
    int counter;    // shared data
public:
    AtomicCounter( int init = 0 ) : counter( init ) {}
    int inc() { counter += 1; return counter; } // mutex members
    int dec() { counter -= 1; return counter; }
};

AtomicCounter a, b{ 1 }, c = { 3 };
... a.inc(); ... // accessed by multiple threads
... b.dec(); ...
... c.inc(); ...

```

## 8.4 Scheduling (Synchronization)

- A monitor may want to schedule tasks in an order different from the order in which they arrive (bounded buffer, readers/write with staleness/freshness).
- There are two techniques: external and internal scheduling.
  - *external* is scheduling tasks outside the monitor and is accomplished with the accept statement.
  - *internal* is scheduling tasks inside the monitor and is accomplished using condition variables with signal and wait.

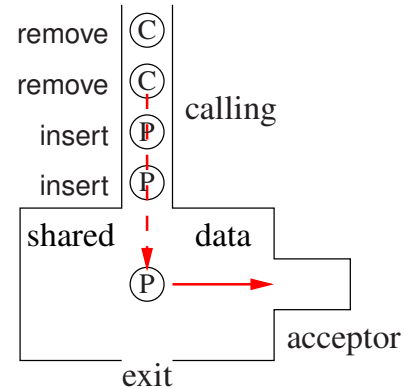
### 8.4.1 External Scheduling

- The `accept` statement controls which mutex members can accept calls.
- By preventing certain members from accepting calls at different times, it is possible to control scheduling of tasks.
- Each `_Accept` defines what cooperation must occur for the accepting task to proceed.
- E.g. Bounded Buffer

```

_Monitor BoundedBuffer {
    int front = 0, back = 0, count = 0;
    int elements[20];
public:
    _Nomutex int query() const { return count; }
    [_Mutex] void insert( int elem );
    [_Mutex] int remove();
};
void BoundedBuffer::insert( int elem ) {
    if ( count == 20 ) _Accept( remove ) ;
    elements[back] = elem;
    back = ( back + 1 ) % 20;
    count += 1;
}
int BoundedBuffer::remove() {
    if ( count == 0 ) _Accept( insert ) ;
    int elem = elements[front];
    front = ( front + 1 ) % 20;
    count -= 1;
    return elem;
}

```



- Queues of tasks form outside the monitor, waiting to be accepted into either insert or remove.
- An acceptor blocks all calls except a call to the specified mutex member(s) occurs. (uses barging prevention)
- Accepted call is executed like a conventional member call.
- When the accepted task exits the mutex member (or waits), the acceptor continues.
- If the accepted task does an accept, it blocks, forming a stack of blocked acceptors.
- Alternative calls that satisfy acceptor's requirement are possible:

```

_Accept( insert || remove ) ; // one of insert or remove

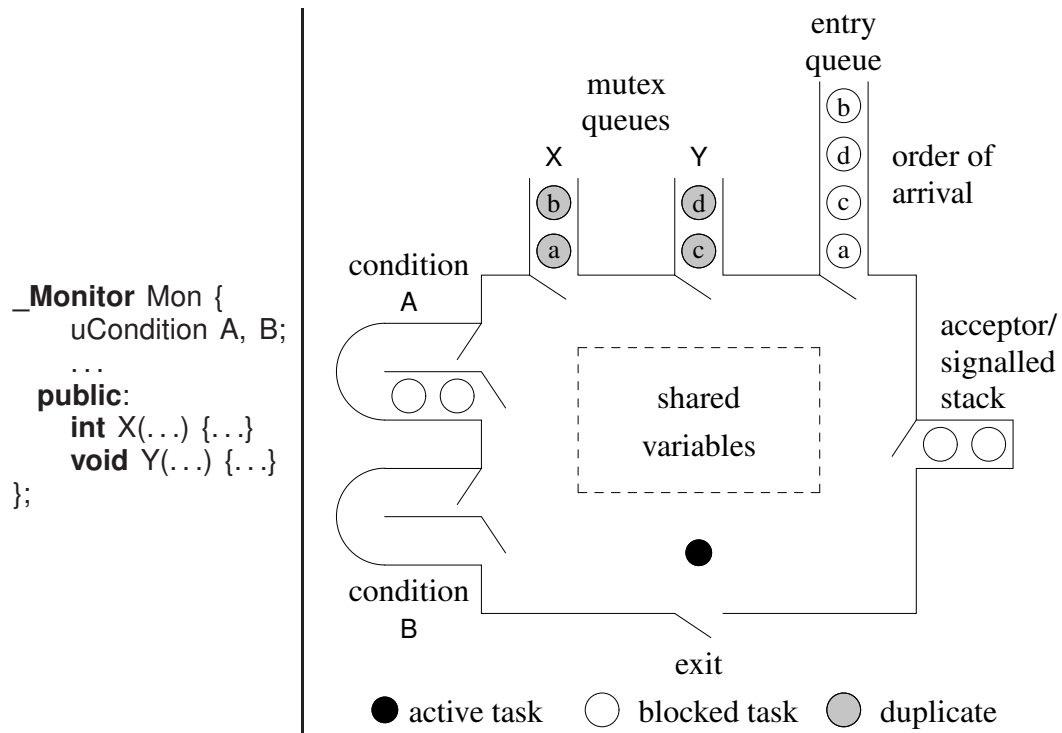
```

- `_Nomutex` member  $\Rightarrow$  multiple threads in monitor  $\Rightarrow$  cannot be accepted.
- External scheduling is simple because unblocking (signalling) is implicit.

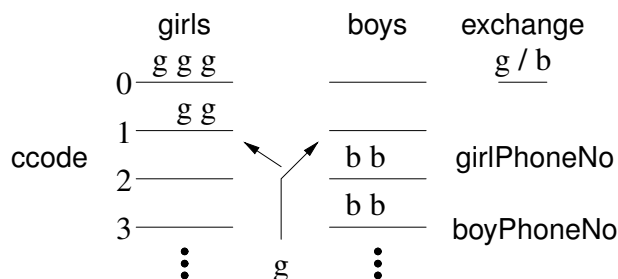




- **wait()** blocks the current thread, and restarts a signalled task or implicitly releases the monitor lock.
- **signal()** unblocks the thread on the front of the condition queue *after* the signaller thread blocks or exits.
- **signalBlock()** unblocks the thread on the front of the condition queue and blocks the signaller thread.
- General Model



- **entry queue** is FIFO list of calling tasks to the monitor.
- When to use external or internal scheduling?
- External is easier to specify and explain over internal with condition variables.
- However, external scheduling cannot be used if:
  - scheduling depends on member parameter value(s), e.g., compatibility code for dating
  - scheduling must block in the monitor but cannot guarantee the next call fulfills cooperation
- Dating service



```

_Monitor DatingService {
    enum { CCodes = 20 }; // compatibility codes
    uCondition girls[CCodes], boys[CCodes], exchange;
    int girlPhoneNo, boyPhoneNo;
public:
    int girl( int phoneNo, int ccode ) {
        if ( boys[ccode].empty() ) { // no compatible boy ?
            girls[ccode].wait(); // wait for boy
            girlPhoneNo = phoneNo; // make phone number available
            exchange.signal(); // wake boy from chair
        } else {
            girlPhoneNo = phoneNo; // make phone number available
            // signalBlock() & remove exchange
            boys[ccode].signal(); // wake boy
            exchange.wait(); // sit in chair
        }
        return boyPhoneNo;
    }
    int boy( int phoneNo, int ccode ) {
        // same as above, with boy/girl interchanged
    }
};

```

- Also, possible to use signal with empty bench (ccode) as chair.

## 8.5 Readers/Writer

- Solution 3 (Section 6.4.4.3, p. 130), no bargers, 5 rules, not temporal

```

_Monitor ReadersWriter {
    int rcnt = 0, wcnt = 0;
    uCondition readers, writers;
public:
    void startRead() {
        if ( wcnt != 0 || ! writers.empty() ) readers.wait();
        rcnt += 1;
        readers.signal();
    }
    void endRead() {
        rcnt -= 1;
        if ( rcnt == 0 ) writers.signal();
    }
};

```

```

void startWrite() {
    if ( wcnt !=0 || rcnt != 0 ) writers.wait();
    wcnt = 1;
}
void endWrite() {
    wcnt = 0;
    if ( ! readers.empty() ) readers.signal();
    else writers.signal();
}
};

```

- Problem: has the same usage protocol as P and V.

ReadersWriter rw;		
<b>readers</b>	<b>writers</b>	
rw.startRead()	rw.startWrite()	// 2-step protocol
// read	// write	
rw.endRead()	rw.endWrite()	

- Simplify protocol:

ReadersWriter rw;		
<b>readers</b>	<b>writers</b>	
rw.read(...)	rw.write(...)	// 1-step protocol

- Implies only one read/write action, or pass pointer to read/write action.

- Alternative interface:

```

_Monitor ReadersWriter {
    _Mutex void startRead() { ... }
    _Mutex void endRead() { ... }
    _Mutex void startWrite() { ... }
    _Mutex void endWrite() { ... }
public:
    _Nomutex void read(...) { // no const or mutable
        startRead(); // acquire mutual exclusion
        // read, no mutual exclusion
        endRead(); // release mutual exclusion
    }
    _Nomutex void write(...) { // no const or mutable
        startWrite() // acquire mutual exclusion
        // write
        endWrite() // release mutual exclusion
    }
};

```

- Alternative interface, and remove wcnt (barging prevention):

```

_Monitor ReadersWriter {
  _Mutex void startRead() {
    if ( ! writers.empty() ) readers.wait();
    rcnt += 1;
    readers.signal();
  }
  _Mutex void endRead() { ... }

public:
  _Nomutex void read(...) { // no const or mutable
    startRead();           // acquire mutual exclusion
    // read, no mutual exclusion
    endRead();             // release mutual exclusion
  }
  void write(...) {        // acquire mutual exclusion
    if ( rcnt != 0 ) writers.wait(); // release/reacquire
    // write, mutual exclusion
    if ( ! readers.empty() ) readers.signal();
    else writers.signal();
  }
};

```

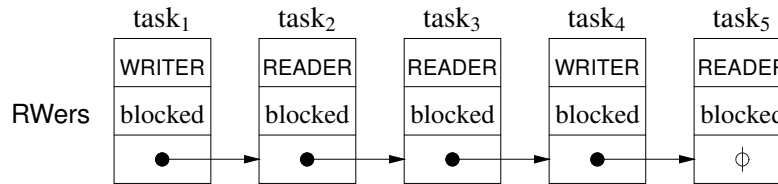
- Solution 4 (Section 6.4.4.4, p. 130), condition shadow queue with type `uintptr_t` data.

```

_Monitor ReadersWriter {
  int rcnt = 0, wcnt = 0;
  uCondition RWers;
  enum RW { READER, WRITER };
public:
  void startRead() {
    if ( wcnt !=0 || ! RWers.empty() ) RWers.wait( READER );
    rcnt += 1;
    if ( ! RWers.empty() && RWers.front() == READER ) RWers.signal();
  }
  void endRead() {
    rcnt -= 1;
    if ( rcnt == 0 ) RWers.signal();
  }

  void startWrite() {
    if ( wcnt != 0 || rcnt != 0 ) RWers.wait( WRITER );
    wcnt = 1;
  }
  void endWrite() {
    wcnt = 0;
    RWers.signal();
  }
};

```



- Use shadow queue to solve dining service, i.e., shadow with phone number.
- $\mu\text{C++}$  `uCondLock` and `uSemaphore` also support shadow queues with type `uintptr_t` data.
- Solution 8, external scheduling

```

_Monitor ReadersWriter {
    int rcnt = 0, wcnt = 0;
public:
    void endRead() {
        rcnt -= 1;
    }
    void endWrite() {
        wcnt = 0;
    }
    void startRead() {
        if ( wcnt > 0 ) _Accept( endWrite );
        rcnt += 1;
    }
    void startWrite() {
        if ( wcnt > 0 ) _Accept( endWrite );
        else while ( rcnt > 0 ) _Accept( endRead );
        wcnt = 1;
    }
};

```

- Why has the order of the member routines changed?

## 8.6 Exceptions

- An exception raised in a monitor member propagates to the caller's thread.

```

_Monitor M {
public:
    void mem1() {
        ... if ( ... ) _Throw E(); ... // E goes to caller
    } // uRendezvousFailure goes to "this"
    void mem2() {
        try {
            ... if ( ... ) _Accept( mem1 ); ...
        } catch( uMutexFailure::RendezvousFailure & ) { // implicitly enabled
            // deal with rendezvous failure
        } // try
    }
};

```

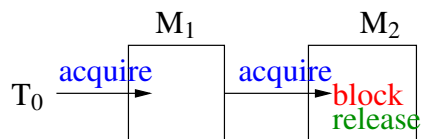
- Caller in `M::mem1` gets exception `E` propagated on its stack.
- On exiting `M::mem1`, caller implicitly raises non-local `RendezvousFailure` exception at monitor acceptor's thread to identify failed cooperation.
- `RendezvousFailure` always enabled  $\Rightarrow$  **`_Enable`** block unnecessary.
- For multiple **`_Accept`** clauses

```
_Accept( mem2 || mem3 || ... );
```

flag variable required to know which member failed.

## 8.7 Nested Monitor Calls

- **Nested monitor problem**: acquire monitor (lock)  $M_1$ , call to monitor  $M_2$ , and wait on condition in  $M_2$ .



- Monitor  $M_2$ 's mutex lock is released by wait, but monitor  $M_1$ 's monitor lock is NOT released  $\Rightarrow$  potential deadlock.
- Releasing all locks can inadvertently release a lock, e.g., incorrectly release  $M_0$  before  $M_1$ .
- Same problem occurs with locks.
- Called **lock composition** problem.
- Nested monitor used as guardian lock for readers/writer problem (like external scheduling RW page 158).

```
_Monitor RW {
  _Monitor RWN {
    uCondition bench;
    int rcnt = 0;
  public:
    void startRead() { rcnt += 1; }
    void endRead() {
      rcnt -= 1;
      if ( rcnt == 0 ) bench.signal();
    }
    void startEndWrite() {
      if ( rcnt > 0 ) bench.wait();           // blocking holding rw
      // sequential write
    }
  } rwn;
}
```

```

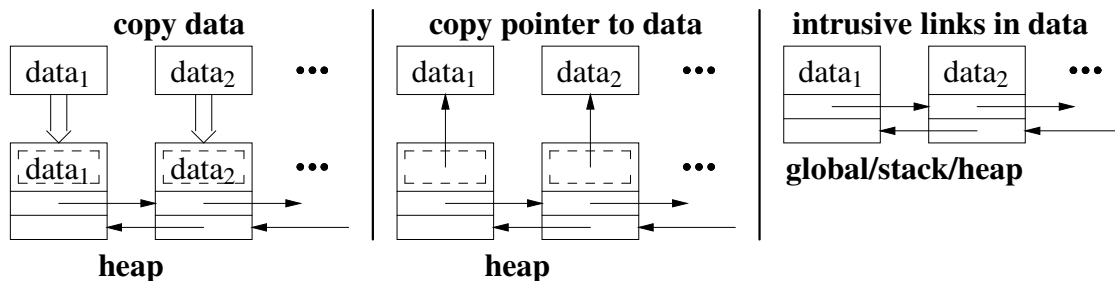
    _Mutex void mutexRead() { rwn.startRead(); }
public:
    void write() { rwn.startEndWrite(); }
    _Nomutex void read() {
        mutexRead();
        // concurrent reads
        rwn.endRead();           // let readers out
    }
};

```

- If the writer waits in rwn, it prevent both readers and writers acquiring rw, which prevents starvation and forces FIFO ordering.

## 8.8 Intrusive Lists

- Non-contiguous variable-length data-structures, e.g., list, dictionary, normally require dynamic allocation as the structure increases/decreases when adding/deleting nodes.
- Three kinds of collections: copy data, copy pointer, and intrusive pointers:



**copy** creates a collection node with link fields,  $\Rightarrow$  dynamic allocation for links and possibly data, copies data and/or data-pointer into node, and links node into collection.

**intrusive** assumes a node with data and link fields,  $\Rightarrow$  no dynamic allocation for collection links or copying.

- Programmer manages node lifetime for copy pointer and intrusive.
- $\mu$ C++ provides **intrusive data-structures** allowing global/stack/heap nodes and no copying.

```

struct Node : public uColable {
    int i;
    Node( int i ) : i{ i } {}
};
int main() {
    Node n1{ 1 }, n2{ 2 }, n3{ 3 };           // stack nodes
    uStack<Node> s;
    s.push( &n1 ); s.push( &n2 ); s.push( &n3 ); // no dynamic allocation
    Node * sp;
    for ( uStackIter<Node> si(s); si >> sp; ) cout << sp->i << " ";
    cout << endl;
}

```



- $\mu\text{C++}$  implementation uses private intrusive links for *non-copyable* objects like a coroutine or task, e.g., tasks on ready queue.
- Intrusive links have two formats: one link field (uColable) for a collection, and two link fields (uSeqable) for a sequence.



```
class stacknode : public uColable { ... }
class queuenode : public uColable { ... }
class seqnode : public uSeqable { ... }
```

- Template classes uStack/uQueue (singly linked) are collections and uSequence (doubly linked) is a sequence.
- uSeqable node appears in sequence/collection; uColable node appears only in a collection.
- Each kind of intrusive list has associated iterators: uStackIter, uQueueIter, uSeqIter.
- See [μC++ reference manual](#) Appendix C for details and examples.
- Concurrency pattern shows how threads use intrusive lists to prevent dynamic allocation.

```
if ( ... ) {
    Node n{ ... }           // allocate on thread stack
    queue.add( n );
    // block
    queue.drop();          // node n must be at head/tail of list
} // automatically free n
```

- Lifetime of node is duration of blocked thread (see above pattern in shadow queue page 132 and private semaphore page 136).

## 8.9 Counting Semaphore, V, P vs. Condition, Signal, Wait

- There are several important differences between these mechanisms:
  - P only blocks if semaphore = 0, wait always blocks
  - V before P affects the P, while signal before wait is lost (no state)
  - multiple Vs may start multiple tasks simultaneously, while multiple signals only start one task at a time because each task must exit serially through the monitor

- Possible to simulate P and V using a monitor:

```

_Monitor semaphore {
    int sem;
    uCondition semcond;
public:
    semaphore( int cnt = 1 ) : sem( cnt ) {}
    void P() {
        if ( sem == 0 ) semcond.wait();
        sem -- 1;
    }
    void V() {
        sem += 1;
        semcond.signal();
    }
};

```

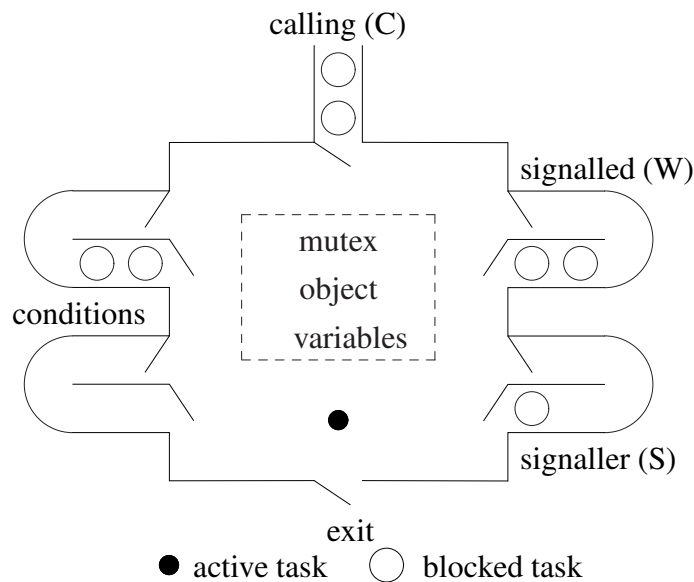
- Can this simulation be reduced?

## 8.10 Monitor Types

- **explicit scheduling** occurs when:
  - An accept statement blocks the active task on the acceptor stack and makes a task ready from the specified mutex member queue.
  - A signal moves a task from the specified condition to the signalled stack.
- **implicit scheduling** occurs when a task waits in or exits from a mutex member, and a new task is selected first from the A/S stack, then the entry queue.

explicit scheduling	internal scheduling (signal)
	external scheduling (accept)
implicit scheduling	monitor selects (wait/exit)

- Monitors are classified by the implicit scheduling (who gets control) of the monitor when a task waits or signals or exits.
- Implicit scheduling can select from the calling (C), signalled (W), and signaller (S) queues.



- Assigning different relative priorities to these queues creates different monitors (e.g.,  $C < W < S$ ).

	relative priority	
1	$C < W < S$	<b>Useful, has Prevention</b>
2	$C < S < W$	no barging
3	$C = W < S$	<b>Usable, needs Avoidance</b>
4	$C = S < W$	barging, starvation without avoidance
5	$C = W = S$	<b>Rejected, Confusing</b>
6	$C < W = S$	arbitrary selection
7	$S = W < C$	<b>Rejected, Unsound</b>
8	$W < S = C$	uncontrolled barging,
9	$W < C < S$	unpreventable starvation
10	$S < W = C$	
11	$S < C < W$	
12	$W < S < C$	
13	$S < W < C$	

- Implicit Signal

- Monitors either have an explicit signal (statement) or an implicit signal (automatic signal).
- The implicit signal monitor has no condition variables or explicit signal statement.
- Instead, there is a `waitUntil` statement, e.g.:

`waitUntil logical-expression`

- The implicit signal causes a task to wait until the conditional expression is true.

```

_Monitor BoundedBuffer {
    int front = 0, back = 0, count = 0;
    int elements[20];
public:
    _Nomutex int query() const { return count; }
    void insert( int elem ) {
        waitUntil count != 20; // not in uC++
        elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
    }
    int remove() {
        waitUntil count != 0; // not in uC++
        int elem = elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        return elem;
    }
};

```

- Additional restricted monitor-type requiring the signaller exit immediately from monitor (i.e., signal  $\Rightarrow$  return), called **immediate-return signal**.
  - not powerful enough to handle all cases, e.g., dating service, but optimizes the most common case of signal before return.
- Remaining monitor types:

signal type	priority	no priority
Blocking	Priority Blocking (Hoare) $C < S < W$ ( $\mu$ C++ signalBlock)	No Priority Blocking $C = S < W$
Nonblocking	Priority Nonblocking $C < W < S$ ( $\mu$ C++ signal)	No Priority Nonblocking $C = W < S$ (Java/C#)
Implicit Signal	Priority Implicit Signal $C < W$	No Priority Implicit Signal $C = W$

- no-priority blocking requires the **signaller task** to recheck the waiting condition in case of a barging task.
  - $\Rightarrow$  use a **while** loop around a signal
- no-priority non-blocking requires the **signalled task** to recheck the waiting condition in case of a barging task.
  - $\Rightarrow$  use a **while** loop around a wait
- implicit (automatic) signal is good for **prototyping** but have poor performance.

- priority nonblocking has no barging and optimizes signal before return (supply cooperation).
- priority blocking has no barging and handles internal cooperation within the monitor (wait for cooperation).
- coroutine monitor (**\_Cormonitor**)
  - coroutine with implicit mutual exclusion on calls to specified member routines:

```
_Mutex _Coroutine C { // _Cormonitor
  void main() {
    ... suspend() ...
    ... suspend() ...
  }
  public:
  void m1( ... ) { ... resume(); ... } // mutual exclusion
  void m2( ... ) { ... resume(); ... } // mutual exclusion
  ... // destructor is ALWAYS mutex
};
```

- can use resume(), suspend(), condition variables (wait(), signal(), signalBlock()) or **\_Accept** on mutex members.
- coroutine can now be used by multiple threads, e.g., coroutine print-formatter accessed by multiple threads.

## 8.11 Java Monitor

- Java has **synchronized** class members (i.e., **\_Mutex** members but **incorrectly named**), and a **synchronized** statement.
- All classes have **one** implicit condition variable and these routines to manipulate it:

```
public wait();
public notify();
public notifyAll();
```

- Java concurrency library has multiple conditions but incompatible with language condition (see Section 11.6.1, p. 221).
- Internal scheduling is no-priority nonblocking ⇒ **barging**
  - wait statements must be in while loops to recheck conditions.
- Bounded buffer:

```

class Buffer {
    // buffer declarations
    private int count = 0;
    public synchronized void insert( int elem ) {
        while ( count == Size ) wait(); // busy-waiting
        // add to buffer
        count += 1;
        if ( count == 1 ) notifyAll();
    }
    public synchronized int remove() {
        while ( count == 0 ) wait(); // busy-waiting
        // remove from buffer
        count -= 1;
        if ( count == Size - 1 ) notifyAll();
        return elem;
    }
}

```

- Only one condition queue, producers/consumers wait together  $\Rightarrow$  unblock all tasks.
- Only one condition queue  $\Rightarrow$  certain solutions are difficult or impossible.
- Erroneous Java implementation of barrier:

```

class Barrier { // monitor
    private int N, count = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        count += 1; // count each arriving task
        if ( count < N )
            try { wait(); } catch( InterruptedException e ) {}
        else // barrier full
            notifyAll(); // wake all barrier tasks
        count -= 1; // uncount each leaving task
    }
}

```

- Nth task does notifyAll, leaves monitor and performs its *i*th step, and then races back (barging) into the barrier before any notified task restarts.
  - It sees count still at N and incorrectly starts its *i*th+1 step before the current tasks have completed their *i*th step.
- Fix by modifying code for Nth task to set count to 0 (barging avoidance) and removing count -= 1.

```

else { // barrier full
    count = 0; // reset count
    notifyAll(); // wake all barrier tasks
}

```

- Technically, still wrong because of **spurious wakeup**  $\Rightarrow$  requires loop around wait.

```

if ( count < N )
    while ( ??? ) // cannot be count < N as count is always < N
        try { wait(); } catch( InterruptedException e ) {}

```

- Requires more complex implementation.

```

class Barrier { // monitor
    private int N, count = 0, generation = 0;
    public Barrier( int N ) { this.N = N; }
    public synchronized void block() {
        int mygen = generation;
        count += 1; // count each arriving task
        if ( count < N ) // barrier not full ? => wait
            while ( mygen == generation )
                try { wait(); } catch( InterruptedException e ) {}
        else { // barrier full
            count = 0; // reset count
            generation += 1; // next group
            notifyAll(); // wake all barrier tasks
        }
    }
}

```

- Misconception of building condition variables in Java with nested monitors:

```

class Condition { // try to build condition variable
    public synchronized void Wait() {
        try { wait(); } catch( InterruptedException ex ) {};
    }
    public synchronized void Notify() { notify(); }
}
class BoundedBuffer {
    // buffer declarations
    private Condition full = new Condition(), empty = new Condition();
    public synchronized void insert( int elem ) {
        while ( count == NoOfElems ) empty.Wait(); // block producer
        // add to buffer
        count += 1;
        full.Notify(); // unblock consumer
    }
    public synchronized int remove() {
        while ( count == 0 ) full.Wait(); // block consumer
        // remove from buffer
        count -= 1;
        empty.Notify(); // unblock producer
        return elem;
    }
}

```

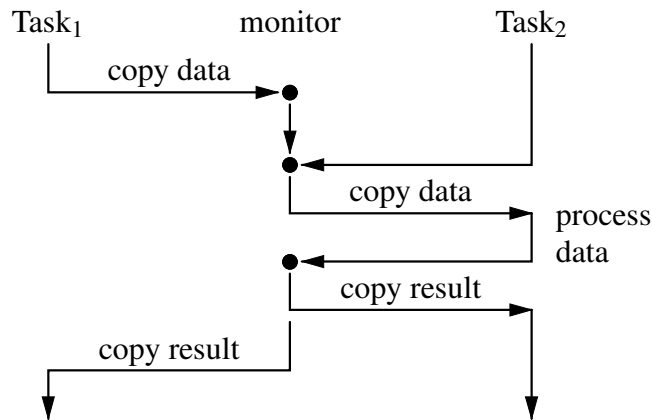
- Deadlocks at empty.Wait()/full.Wait() as buffer monitor-lock is not released.



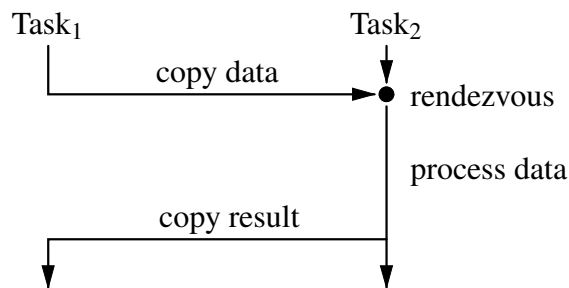


## 9 Direct Communication

- Monitors work well for passive objects that require mutual exclusion because of sharing.
- However, communication among tasks with a monitor is indirect.
- Problem: point-to-point with reply indirect communication:



- Point-to-point with reply direct communication:



- Tasks can communicate directly by calling each others member routines.
- Member call is synchronous, nonlocal exception is asynchronous and indirect.

### 9.1 Task

- A task is like a coroutine because it has a distinguished member, (task main), which has its own execution state.
- A task is unique because it has a thread of control, which begins execution in the task main when the task is created.
- A task is like a monitor because it provides mutual exclusion (and synchronization) so only one thread is active in the object.

- public members of a task are implicitly mutex and other kinds of members can be made explicitly mutex.
  - external scheduling allows direct calls to mutex members (task's thread blocks while caller's executes).
  - without external scheduling, tasks must *call out* to communicate  $\Rightarrow$  third party, or somehow emulate external scheduling with internal.
- In general, basic execution properties produce different abstractions:

object properties		member routine properties	
thread	stack	No S/ME	S/ME
No	No	<b>1</b> class	<b>2</b> monitor
No	Yes	<b>3</b> coroutine	<b>4</b> coroutine-monitor
Yes	No	<b>5 reject</b>	<b>6 reject</b>
Yes	Yes	<b>7 reject?</b>	<b>8</b> task

- When thread or stack is missing it comes from calling object.
- Abstractions are not ad-hoc, rather derived from basic properties.
- Each of these abstractions has a particular set of problems it can solve, and therefore, each has a place in a programming language.

## 9.2 Scheduling

- A task may want to schedule access to itself by other tasks in an order different from the order in which requests arrive.
- As for monitors, there are two techniques: external and internal scheduling.

### 9.2.1 External Scheduling

- As for a monitor (see Section 8.4.1, p. 152), the accept statement can be used to control which mutex members of a task can accept calls.

```

_Task BoundedBuffer {
    int front = 0, back = 0, count = 0;
    int Elements[20];
public:
    _Nomutex int query() const { return count; }
    void insert( int elem ) {
        if ( count == 20 ) _Accept( remove ); // move to main
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
    }
    int remove() {
        if ( count == 0 ) _Accept( insert ); // move to main
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        return elem;
    }
private:
    void main() {
        for ( ;; ) { // INFINITE LOOP!!!
            _Accept( insert || remove ); // no synchronization
            _When ( count != 20 ) _Accept( insert ) { // after call
            } or _When ( count != 0 ) _Accept( remove ) { // after call
            } // _Accept
        }
    }
};

```

- $\_Accept( m1 \parallel m2 ) \mathbf{S1} \equiv \_Accept( m1 ) \mathbf{S1}; \text{ or } \_Accept( m2 ) \mathbf{S1}; // S2$   
 $if ( C1 \parallel C2 ) \mathbf{S1} \equiv if ( C1 ) \mathbf{S1}; \text{ else if } ( C2 ) \mathbf{S1}; // S2$
- Extended version allows different `_When`/code after call for each accept.
- The `_When` clause is like the condition of conditional critical region:
  - The condition must be true (or omitted) *and* a call to the specified member must exist before a member is accepted.
- If all the accepts are conditional and false, the statement does nothing (like `switch` with no matching `case`).
- If some conditionals are true, but there are no outstanding calls, the acceptor is blocked until a call to an appropriate member is made.
- If several members are accepted and outstanding calls exist to them, a call is selected based on the order of the `_Accepts`.
  - Hence, the order of the `_Accepts` indicates their relative priority for selection if there are several outstanding calls.

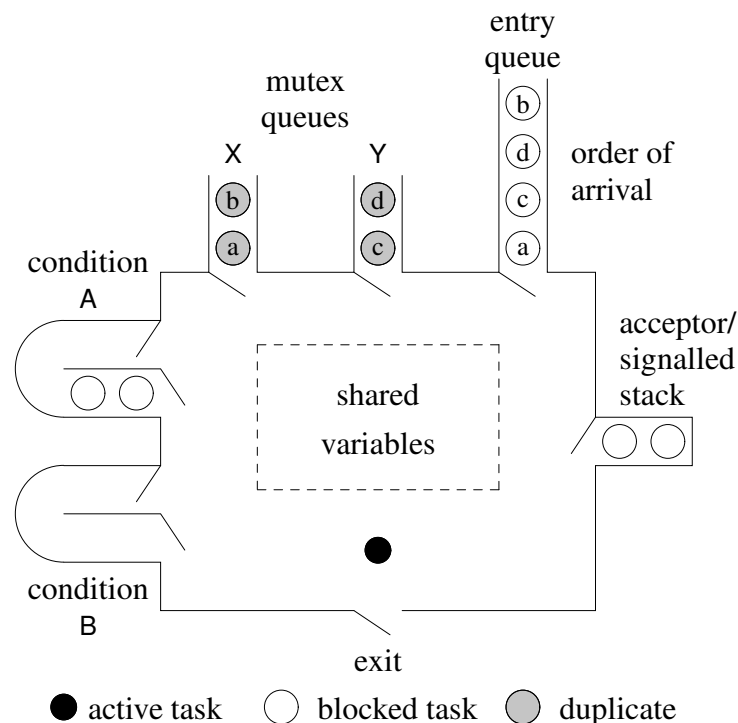
- Is there a potential starvation problem?
- Why are accept statements moved from member routines to the task main?
- Why is BoundedBuffer::main defined at the end of the task?
- Equivalence using **if** statements:
 

```
if ( 0 < count && count < 20 ) _Accept( insert || remove ); // not full/empty
else if ( count < 20 ) _Accept( insert ); // not full
else /* if ( 0 < count ) */ _Accept( remove ); // not empty
```

- Generalize from 2 to 3 conditionals/members:

```
if      ( C1 && C2 && C3 ) _Accept( M1 || M2 || M3 );
else if ( C1 && C2 ) _Accept( M1 || M2 );
else if ( C1 && C3 ) _Accept( M1 || M3 );
else if ( C2 && C3 ) _Accept( M2 || M3 );
else if ( C1 ) _Accept( M1 );
else if ( C2 ) _Accept( M2 );
else if ( C3 ) _Accept( M3 );
```

- Necessary to ensure that for every true conditional, only the corresponding members are accepted.
- $2^N - 1$  **if** statements needed to simulate  $N$  accept clauses.
- The acceptor is pushed on the top of the A/S stack and normal implicit scheduling occurs ( $C < W < S$ ).



- Once accepted call completes or caller wait()s, the statement after the accepting **\_Accept** clause is executed and the accept statement is complete.
- If there is a terminating **\_Else** clause and no **\_Accept** can be executed immediately, the terminating **\_Else** clause is executed.

```
_Accept( ... ) {
} or _Accept( ... ) {
} _Else { ... } // executed if no callers
```

- Hence, the terminating **\_Else** clause allows a conditional attempt to accept a call without the acceptor blocking (tryacquire).
- To achieve greater concurrency in the bounded buffer, change to:

```
void insert( int elem ) {
    Elements[back] = elem;
}
int remove() {
    return Elements[front];
}
private:
void main() {
    for ( ;; ) {
        _When ( count != 20 ) _Accept( insert ) {
            back = (back + 1) % 20;
            count += 1;
        } or _When ( count != 0 ) _Accept( remove ) {
            front = (front + 1) % 20;
            count -= 1;
        } // _Accept
    }
}
```

### 9.2.2 Internal Scheduling

- Scheduling among tasks inside the monitor.
- As for monitors, condition, signal and wait are used.

```

_Task BoundedBuffer {
    uCondition full, empty;
    int front = 0, back = 0, count = 0;
    int Elements[20];
public:
    _Nomutex int query() const { return count; }
    void insert( int elem ) {
        if ( count == 20 ) empty.wait();
        Elements[back] = elem;
        back = ( back + 1 ) % 20;
        count += 1;
        full.signal();
    }

    int remove() {
        if ( count == 0 ) full.wait();
        int elem = Elements[front];
        front = ( front + 1 ) % 20;
        count -= 1;
        empty.signal();
        return elem;
    }

private:
    void main() {
        for ( ;; ) {
            _Accept( insert || remove );
            // do other work
        }
    }
};

```

- Requires combination of internal and external scheduling.
- **Rendezvous is logically pending when wait restarts \_Accept task, but post \_Accept statement still executed (no RendezvousFailure).**
  - Acceptor must eventually complete rendezvous for waiting caller.
- Try moving code to achieve greater concurrency.

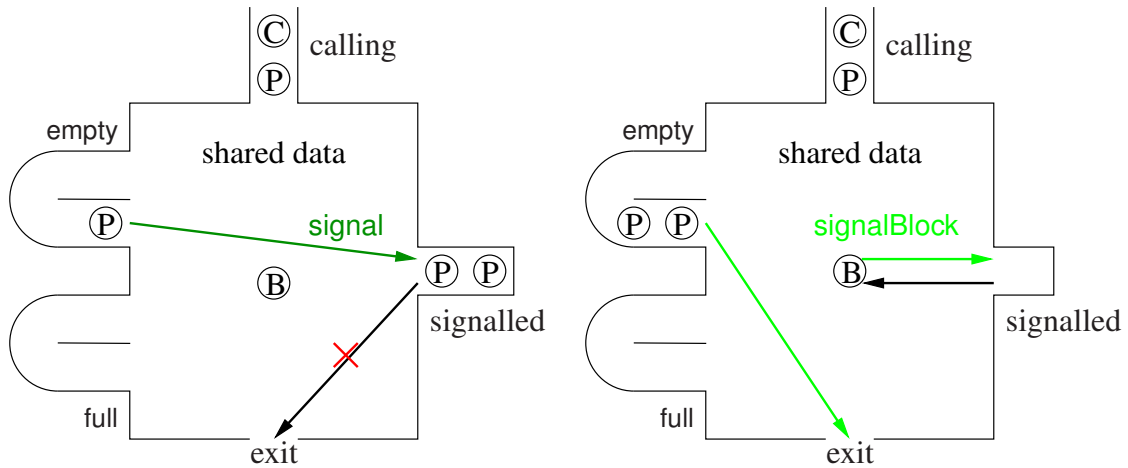
```

void insert( int elem ) {
    if ( count == 20 ) empty.wait(); // only wait if necessary
    Elements[back] = elem;
}
int remove() {
    if ( count == 0 ) full.wait(); // only wait if necessary
    return Elements[front];
}
private:
void postInsert() { // helper members
    back = ( back + 1 ) % size;
    count += 1;
}
void postRemove() {
    front = ( front + 1 ) % size;
    count -= 1;
}

void main() {
    for ( ;; ) {
        _Accept( insert ) {
            if ( count != 20 ) { // producer did not wait ?
                postInsert();
                if ( ! full.empty() ) { // waiting consumers ?
                    full.signal(); // wake and adjust
                    postRemove();
                }
            }
        }
        or _Accept( remove ) {
            if ( count != 0 ) { // consumer did not wait ?
                postRemove();
                if ( ! empty.empty() ) { // waiting producers ?
                    empty.signal(); // wake and adjust
                    postInsert();
                }
            }
        }
    } // _Accept
} // for
}

```

- Must prevent starvation by producers (use **\_When** or flip **\_Accept** clauses).
- Must change signal to signalBlock.



- Signalled tasks cannot leave because buffer task continues in monitor.
- Signal-blocked tasks leave immediately because buffer-task blocks.

### 9.2.3 Accepting the Destructor

- Common way to terminate a task is to have a stop member:

```

_Task BoundedBuffer {
public:
    ...
    void stop() {} // empty
private:
    void main() {
        // start up
        for ( ;; ) {
            _Accept( stop ) { // terminate ?
                break;
            } or _When ( count != 20 ) _Accept( insert ) {
                ...
            } or _When ( count != 0 ) _Accept( remove ) {
                ...
            } // _Accept
        }
        // close down
    }
}

```

- Call stop when task is to stop:

```

int main() {
    BoundedBuffer buf;
    // create producer & consumer tasks
    // delete producer & consumer tasks
    buf.stop(); // no outstanding calls to buffer
    // maybe do something with buf (print statistics)
} // delete buf

```



- If termination and deallocation follow one another, accept destructor:

```

void main() {
    for ( ;; ) {
        _Accept( ~BoundedBuffer ) {
            break;
        } or _When ( count != 20 ) _Accept( insert ) { ...
        } or _When ( count != 0 ) _Accept( remove ) { ...
        } // _Accept
    }
    // close down
}

```

- However, the semantics for accepting a destructor are different from accepting a normal mutex member.
- When the call to the destructor occurs, the caller blocks immediately if there is thread active in the task because a task's storage cannot be deallocated while in use.
- When the destructor is accepted, the caller is blocked and pushed onto the A/S stack *instead of the acceptor*.
- Therefore, control restarts at the accept statement *without* executing the destructor member.
- Allows mutex object to clean up before termination (monitor or task).
- ***Task now behaves like a monitor because its thread is halted.***
- Only when the caller to the destructor is popped off the A/S stack by the implicit scheduling is the destructor executed.
- The destructor can reactivate any blocked tasks on condition variables and/or the acceptor/signalled stack.

### 9.3 Increasing Concurrency

- 2 task involved in direct communication: client (caller) & server (callee)
- possible to increase concurrency on both the client and server side

#### 9.3.1 Server Side

- Server manages a resource and server thread should introduce additional concurrency (assuming no return value).

No Concurrency	Some Concurrency
<pre> _Task server1 { public:   void mem1(...) { S1 }   void mem2(...) { S2 }   void main() {     ...     _Accept( mem1 );     or _Accept( mem2 );   } } </pre>	<pre> _Task server2 { public:   void mem1(...) { S1.copy-in }   int mem2(...) { S2.copy-out }   void main() {     ...     _Accept( mem1 ) { S1.work }     or _Accept( mem2 ) { S2.work };   } } </pre>

- No concurrency in left example as server is blocked, while client does work.
- Alternatively, client blocks in member, server does work, and server unblocks client.
- Some concurrency possible in right example if service can be factored into administrative (S1.copy) and work (S1.work) code.
  - i.e., move code from the member to statement executed after member is accepted.
- Small overlap between client and server (client gets away earlier) increasing concurrency.

### 9.3.1.1 Internal Buffer

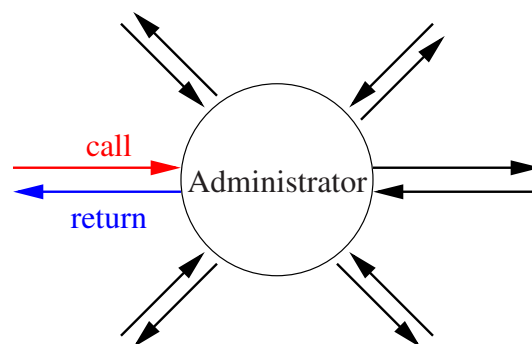
- The previous technique provides buffering of size 1 between the client and server.
- Use a larger internal buffer to allow clients to get in and out of the server faster?
- I.e., an internal buffer can be used to store the arguments of multiple clients until the server processes them.
- However, there are several issues:
  - Unless the average time for production and consumption is approximately equal with only a small variance, the buffer is either always full or empty.
  - Because of the mutex property of a task, no calls can occur while the server is working, so clients cannot drop off their arguments.
 

**The server could periodically accept calls while processing requests from the buffer (awkward).**
  - Clients may need to wait for replies, in which case a buffer does not help unless there is an advantage to processing requests in non-FIFO order.
- Only way to free server's thread to receive new requests and return finished results to clients is add another thread.
- Additional thread is a **worker task** that calls server to get work from buffer and return results to buffer.
- Note, customer (client), manager (server) and employee (worker) relationship.

- Number of workers has to balance with number of clients to maximize concurrency (bounded-buffer problem).

### 9.3.1.2 Administrator

- An **administrator** is a server managing multiple clients and worker tasks.
- The key is that an administrator does little or no “real” work; its job is to manage.
- Management means delegating work to others, receiving and checking completed work, and passing completed work on.
- An administrator is called by others, so an administrator is always accepting calls.



- An administrator makes no call to another task because calling may block the administrator.
- An administrator usually maintains a list of work to pass to **worker tasks**.

- Typical workers are:

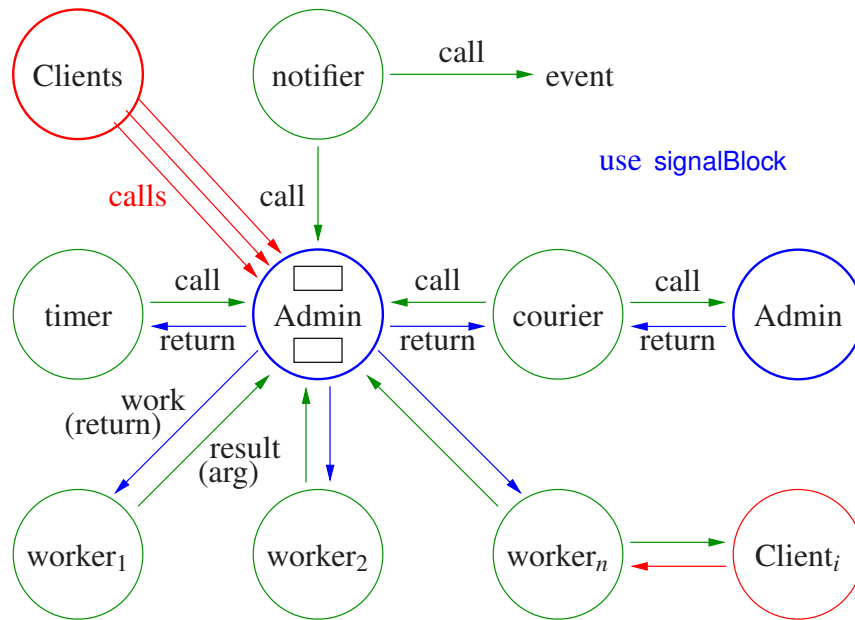
**timer** - prompt the administrator at specified time intervals

**notifier** - perform a potentially blocking wait for an external event (key press)

**simple worker** - do work given to them by and return the result to the administrator

**complex worker** - do work given to them by administrator and interact directly with client of the work

**courier** - perform a potentially blocking call on behalf of the administrator



### 9.3.2 Client Side

- While a server can attempt to make a client's delay as short as possible, not all servers do it.
- In some cases, a client may not have to wait for the server to process a request (producer/consumer problem)
- This can be accomplished by an asynchronous call from the client to the server, where the caller does not wait for the call to complete.
- Asynchronous call requires implicit buffering between client and server to store the client's arguments from the call.
- $\mu\text{C++}$  provides only synchronous call, i.e., the caller is delayed from the time the arguments are delivered to the time the result is returned (like a procedure call).
- It is possible to build asynchronous facilities out of the synchronous ones and vice versa.

#### 9.3.2.1 Returning Values

- If a client only drops off data to be processed by the server, the asynchronous call is simple.
- However, if a result is returned from the call, i.e., from the server to the client, the asynchronous call is significantly more complex.
- To achieve asynchrony in this case, a call must be divided into two calls:

```

callee.start( arg );           // provide arguments
// caller performs other work asynchronously
result = callee.wait();        // obtain result

```

- Not same as START/WAIT because server thread exists.

- many-to-one (start/wait) versus one-to-one (START/WAIT)
- Time between calls allows calling task to execute asynchronously with task performing operation on the caller's behalf.
- If result is not ready when second call is made
  - caller blocks
  - caller has to call again (poll).
- However, this requires a protocol so when the client makes the second call, the correct result can be found and returned.

#### 9.3.2.2 Tickets

- One form of protocol is the use of a token or ticket.
- The first part of the protocol transmits the arguments specifying the desired work and a ticket (like a laundry ticket) is returned immediately.
- The second call *pulls* the result by passing the ticket.
- The ticket is matched with a result, and the result is returned if available or the caller is blocks or polls until the result is available.
- However, protocols are error prone because the caller may not obey the protocol (e.g., never retrieve a result, use the same ticket twice, forged ticket).

#### 9.3.2.3 Call-Back Routine

- Another protocol is to transmit (register) a routine on the initial call.
- When the result is ready, the routine is called by the task generating the result, passing it the result.
- The call-back routine cannot block the server; it can only store the result and set an indicator (e.g., V a semaphore) known to the client.
- The original client must *poll* the indicator or block until the indicator is set.
- The advantage is that the server can *push* the result back to the client faster (nagging the client to pickup).
- Also, the client can write the call-back routine, so they can decide to poll or block or do both.

### 9.3.2.4 Futures

- A **future** provides the same asynchrony as above but without an explicit protocol.
- The protocol becomes implicit between the future and the task generating the result.
- Furthermore, it removes the difficult problem of when the caller should try to retrieve the result.
- In detail, a future is an object that is a subtype of the result type expected by the caller.
- Instead of two calls as before, a single call is made, passing the appropriate arguments, and a future is returned.

```
future = callee.work( arg );           // provide arguments, return future
// perform other work asynchronously
i = future + ...;                     // obtain result, may block if not ready
```

- The future is returned immediately and it is empty.
- The caller “believes” the call completed and continues execution with an empty result value.
- The future is filled in at some time in the “future”, when the result is calculated.
- If the caller tries to use the future before its value is filled in, the caller is implicitly blocked.
- The general design for a future is:

```
class Future : public ResultType {
    friend _Task server;           // allow server to access internal state
    ResultType result;           // place result here
    uSemaphore avail;           // wait here if no result
    Future * link;               // intrusive data structure
public:
    Future() : avail( 0 ) {}

    ResultType get() {
        avail.P();               // wait for result
        return result;
    }
};
```

- the semaphore is used to block the caller if the future is empty
- the link field is used to chain the future onto a server work-list.
- Unfortunately, the syntax for retrieving the value of the future is awkward as it requires a call to the `get` routine.
- Also, in languages without garbage collection, the future must be explicitly deleted.
- $\mu$ C++ provides two forms of template futures, which differ in storage management (like Actors/Messages).

- Explicit-Storage-Management future (Future\_ESM<T>) must be allocated and deallocated explicitly by the client.
- Implicit-Storage-Management future (Future\_ISM<T>) automatically allocates and frees storage (when future no longer in use, GC).
- Focus on Future\_ISM as simpler to use but less efficient in certain cases.
- Basic set of operations for both types of futures, divided into client and server operations.

### Client

- Future value:

```

#include <uFuture.h>
Server server; // server thread handles async calls
Future_ISM<int> f[10];
for ( int i = 0; i < 10; i += 1 ) {
    f[i] = server.perform( i ); // asynchronous server call
}
// work asynchronously while server processes requests
for ( int i = 0; i < 10; i += 1 ) { // retrieve async results
    osacquire( cout ) << f[i]() << ' ' << f[i]() + i << endl;
}
f[3]() = 3; // DISALLOWED: OTHER THREADS READING VALUE
...
f[3].reset(); // reset future => empty and can be reused (be careful)
...
f[3].cancel(); // attempt to stop server and clients from usage

```

- After the future result is retrieved, it can be retrieved again cheaply (no blocking).
- Why is combining osacquire( cout ) and f[i]() dangerous?
- Future pointer:

```

#include <uFuture.h>
Server server; // server thread handles async calls
int val
Future_ISM<int *> fval;
fval = server.perform( val ); // async call to server (change val by reference)
// work asynchronously while server processes requests
osacquire( cout ) << *fval() << endl; // synchronize on retrieve value
val = 3; // ALLOWED: BUT FUTURE POINTER IS STILL READ-ONLY

```

available – returns **true** if asynchronous call completed, otherwise **false**. complete  $\Rightarrow$  result available, server raised exception, or call cancelled

operator() – (function call) returns **read-only** copy of future result.

block if future unavailable; raise exception if exception returned by server.

future result can be retrieved multiple times by any task ( $\Rightarrow$  read-only) until the future is reset or destroyed.

reset – mark future as empty  $\Rightarrow$  current future value is unavailable  $\Rightarrow$  future can be reused.

cancel – attempts to cancel the asynchronous call the future refers to.

Clients waiting for the result are unblocked, and exception of type `uCancelled` is raised at them.

cancelled – returns **true** if the future is cancelled and **false** otherwise.

## Server

```

_Task Server {
    struct Work {
        int i; // argument(s)
        Future_ISM<int> result; // result
        Work( int i ) : i( i ) {}
    };
    Future_ISM<int> perform( int i ) { // called by clients
        Work *w = new Work( i ); // create work request
        requests.push_back( w ); // add to list of requests
        return w->result; // return future in request
    }

    // server or server's worker does
    Work *w = requests.front(); // take next work request
    requests.pop_front(); // remove request
    int r = ... w->i ...; // compute result using argument w->i
    w->result.delivery( r ); // insert result into future
    delete w; // CLIENT FUTURE NOT DELETED (REF COUNTING)
};

```

delivery( T result ) – copy client result into the future, unblocking clients waiting for the result.

delivery( uBaseEvent \* cause ) – copy exception into the future, and the exception is thrown at waiting clients.

For future to manage exception lifetime, the exception must be dynamically allocated.

```

_Exception E {};
Future_ISM<int> result;
result.delivery( new E ); // deleted by future

```

The exception is implicitly deleted when the future is deleted or reset.

## Complex Future Access (client side)

- **select statement** waits for one or more **heterogeneous** futures based on logical selection-criteria.
- Simplest select statement has a single **\_Select** clause, e.g.:

```

_Select( selector-expression );

```



- Selector-expression must be satisfied before execution continues.
- For a single future, the expression is satisfied if and only if the future is available.

```
_Select( f1 ); // expensive (wait)      ≡  x = f1(); // expensive (wait),
x = f1(); // cheap, value or exception    // value or exception
```

- Selector is only select blocked until `f1.available()` is true.
- Does not return future value or throw exception.
- Multiple futures may appear in a compound selector-expression, related using logical operators `||` and `&&`:

```
_Select( f1 || f2 && f3 );
```

- Normal operator precedence applies: `_Select( ( f1 || ( f2 && f3 ) ) )`.
- Execution waits until either future `f1` is available or both futures `f2` and `f3` are available.
- Selector-expression is evaluated from left to right, even for operators of equal priority ⇒ when multiple subexpressions are true, the left-most subexpression satisfies the select statement.
- For any selector expression containing an `||` operator, some futures in the expression may be unavailable after the selector expression is satisfied.
- E.g., in the above, if future `f1` becomes available, neither, one or both of `f2` and `f3` may be available.
- **or** and **and** keywords relate the `_Select` clauses like operators `||` and `&&` relate futures in a select-expression, including precedence.

```
_Select( f1 || f2 && f3 ); ≡ _Select( f1 )
                        or _Select( f2 )
                        and _Select( f3 );
```

- Parentheses may be used to specify evaluation order.

```
_Select( ( f1 || ( f2 && f3 ) ) ) ≡ ( _Select( f1 )
                        or ( _Select( f2 )
                        and _Select( f3 ) ) );
```

- A `_Select` clause may be guarded with a logical expression and have code executed after a future receives a value:

```
_When ( conditional-expression ) _Select( f1 )
    statement-1                    // action, future available
or
_When ( conditional-expression ) _Select( f2 )
    statement-2                    // action, future available
and _When ( conditional-expression ) _Select( f3 )
    statement-3                    // action, future available
```

- Each **\_Select**-clause action is executed when its sub-selector expression is satisfied, i.e., when each future becomes available.
- However, control does not continue until the selector expression associated with the entire statement is satisfied.
- E.g., if f2 becomes available, statement-2 is executed but the selector expression for the entire statement is **not** satisfied so control blocks again.
- When either f1 or f3 become available, statement-1 or 3 is executed, and the selector expression for the entire statement is satisfied so control continues.
- If a guard is false, execution continues without waiting for that future to become available.

```

    _When( true ) _Select( f1 ) {...}      _When( true ) _Select( f1 ) {...}
    or
    _When( false ) _Select( f2 ) {...} ≡
    and
    _When( true ) _Select( f3 ) {...}      _When( true ) _Select( f3 ) {...}
  
```

Assume only f3 becomes available, execution continues.

- An action statement is triggered only once for its selector expression, even if the selector expression is compound.

```

    _Select( f1 )
      statement-1
    or _Select( f2 && f3 )
      statement-2           // triggered once after both available
  
```

- In statement-2, both futures f2 and f3 are available.
- However, for ||:

```

    _Select( f1 || f2 )
      statement-1           // triggered once after one available
    and _Select( f3 )
      statement-2
  
```

- In statement-1, only one future f1 or f2 caused the action to be triggered.
- Hence, it is necessary to check which of the two futures is available.
- A select statement can be non-blocking using a terminating **\_Else** clause, e.g.:

```

    _Select( selector-expression )
      statement           // action
    _When ( conditional-expression ) _Else // terminating clause
      statement           // action
  
```

- The **\_Else** clause *must* be the last clause of a select statement.

- If its guard is true or omitted and the select statement is not immediately true, then the action for the **\_Else** clause is executed and control continues.
- If the guard is false, the select statement blocks as if the **\_Else** clause is not present.
- Complex synchronization: wait for 3 different events or 1 to stop.

```

Future_ISM<int> fi;
Future_ISM<double> fd;
struct Msg { int i, j; }; Future_ISM<Msg> fm;
struct Stop {}; Future_ISM<Stop> fs;
struct Cont {}; Future_ISM<Cont> fc;
_Task Worker {
    void main() {
        for ( ;; ) {
            _Select( fi ) { cout << fi() << endl; fi.reset(); }
            and _Select( fd ) { cout << fd() << endl; fd.reset(); }
            and _Select( fm ) { cout << fm()->i << " "
                            << fm()->j << endl; fm.reset(); }
            or _Select( fs ) { cout << "stop" << endl; break; }
            fc.delivery( Cont{} ); // synchronize
        }
    }
};
int main() {
    Worker worker;
    for ( int i = 0; i < 10; i += 1 ) {
        fi.delivery( i );
        fd.delivery( i + 2.5 );
        fm.delivery( Msg{ i, 2 } );
        fc(); fc.reset(); // wait for 3 futures to be processed
    }
    fs.delivery( Stop{} );
} // wait for worker to terminate

```



## 10 Optimization

- A computer with infinite memory and speed requires no optimizations to use less memory or run faster (space/time).
- With finite resources, optimization is useful/necessary to conserve resources and for good performance.
- Furthermore, most programs are not written in optimal order or in minimal form.
  - OO, Functional, SE are seldom optimal approaches on von Neumann machine.
- General forms of optimizations are:
  - **reordering**: data and code are reordered to increase performance in certain contexts.
  - **eliding**: removal of unnecessary data, data accesses, and computation.
  - **replication**: processors, memory, data, code are duplicated because of limitations in processing and communication speed (speed of light).
- Optimized program must be isomorphic to original  $\Rightarrow$  produce same result for fixed input.
- Kinds of optimizations are restricted by the kind of execution environment.

### 10.1 Sequential Optimizations

- Most programs are sequential; even concurrent programs are
  - (large) sections of sequential code per thread connected by
  - small sections of concurrent code where threads interact (protected by synchronization and mutual exclusion (SME))
- **Sequential** execution presents simple semantics for optimization.
  - operations occur in **program order**, i.e., sequentially
- Dependencies result in partial ordering among a set of statements (precedence graph):
  - **data dependency** (R  $\Rightarrow$  read, W  $\Rightarrow$  write)

$R_x \rightarrow R_x$	$W_x \rightarrow R_x$	$R_x \rightarrow W_x$	$W_x \rightarrow W_x$
$y = x;$	$x = 0;$	$y = x;$	$x = 0;$
$z = x;$	$y = x;$	$x = 3;$	$x = 3;$

Which statements can be reordered?

- **control dependency**

```
1  if ( x == 0 )
2      y = 1;
```

Statements cannot be reordered as line 1 determines if 2 is executed.

- To achieve better performance, compiler/hardware make changes:

- reorder disjoint (independent) operations (**variables have different addresses**)

$R_x \rightarrow R_y$	$W_x \rightarrow R_y$	$R_x \rightarrow W_y$	$W_x \rightarrow W_y$
$t = x;$	$x = 0;$	$x == 1;$	$y = 0;$
$s = y;$	$y == 1;$	$y = 3;$	$x = 3;$

Which statements can be reordered?

- elide unnecessary operations (transformation/dead code)

```
x = 0; // unnecessary, immediate change
x = 3;
```

```
for ( int i = 0; i < 10000; i += 1 ); // unnecessary, no loop body
```

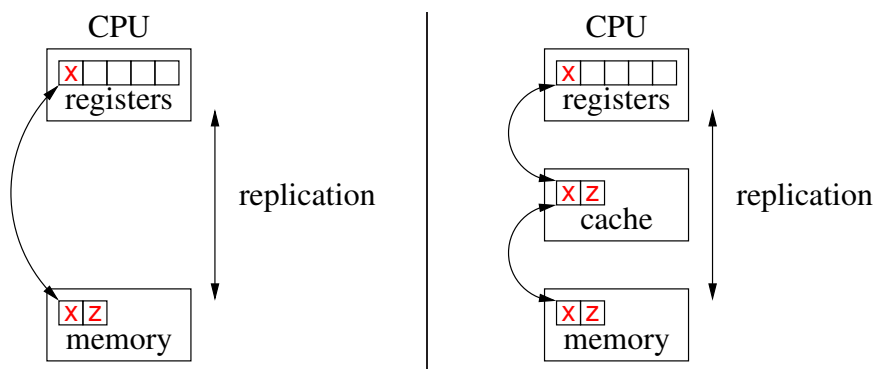
```
int factorial( int n, int acc ) { // tail recursion
    if ( n == 0 ) return acc;
    return factorial( n - 1, n * acc ); // convert to loop
}
```

- execute in parallel if multiple functional-units (adders, floating units, pipelines, cache)

- Very complex reordering, reducing, and overlapping of operations allowed.
- Overlapping implies micro-parallelism, **but limited capability in sequential execution.**

## 10.2 Memory Hierarchy

- Complex memory hierarchy:



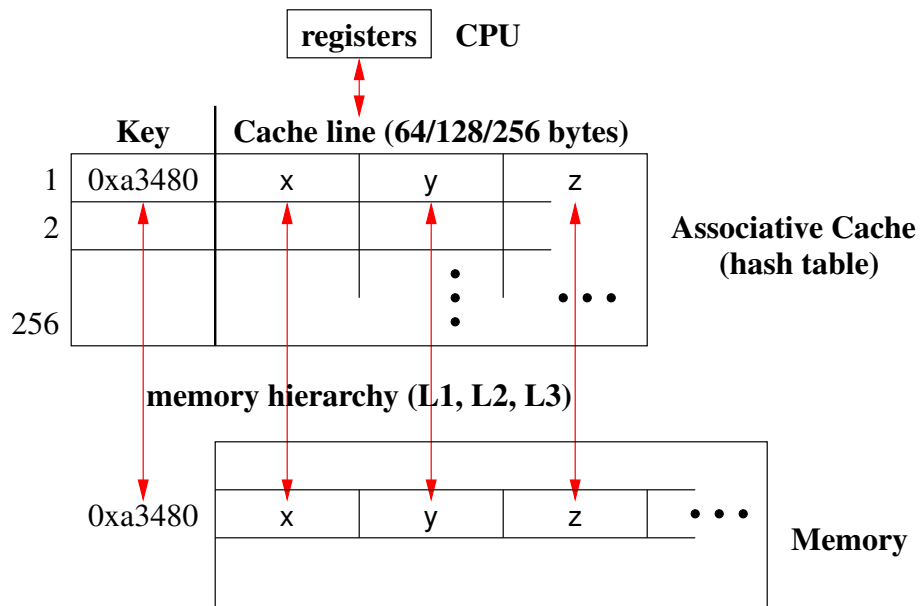
- Optimizing data flow along this hierarchy defines a computer's speed.
- Hardware aggressively optimizes data flow for sequential execution.
- Having basic understanding of cache is essential to understanding performance of both sequential and concurrent programs.

### 10.2.1 Cache Review

- Problem: CPU 100(0) times faster than memory (100,00(0) times faster than disk).
- Solution: copy data from general memory into very, very fast local-memory (registers).
- Problem: billions of bytes of memory but only 6–256 registers.
- Solution: move highly accessed data *within* a program from memory to registers for as long as possible and then back to memory.
- Problem: quickly run out of registers as more data accessed.
  - ⇒ must rotate data from memory through registers dynamically.
  - compiler attempts to keep highly used variables in registers (LRU, requires oracle)
- Problem: does not handle highly accessed data *among* programs (threads).
  - each context switch saves and restores most registers to memory
  - registers are private and cannot be shared
- Solution: use hardware **cache** (automatic registers) to stage data without pushing to memory and allow sharing of data among programs.

```

int x, y, z;
x += 1;    ld  r1,0xa3480    // load register 1 from x
           add r1,#1        // increment
           st  r1,0xa3480    // store register 1 to x
  
```

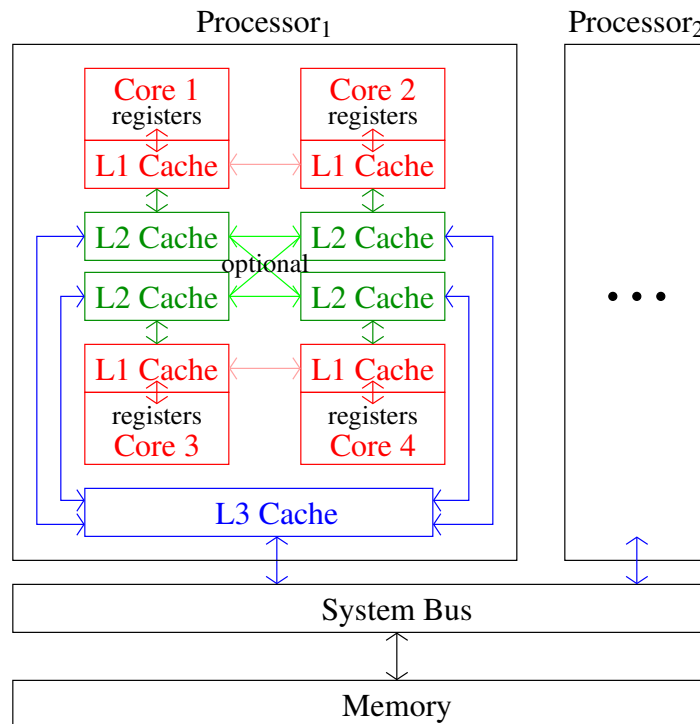


- Caching transparently hides the latency of accessing main memory.
- Cache loads in 64/128/256 bytes, called **cache line**, with addresses multiple of line size.

- When  $x$  is loaded into register 1, a cache line containing  $x$ ,  $y$ , and  $z$  are implicitly copied up the memory hierarchy from memory through caches.
- When cache is full, data evicted, i.e., remove old cache-lines to bring in new (LRU).
- When program ends, its addresses are flushed from the memory hierarchy.
- In theory, cache can eliminate registers, but registers provide small addressable area (register window) with short addresses (3-8 bits for 8-256 registers)  $\Rightarrow$  shorter instructions.

### 10.2.2 Cache Coherence

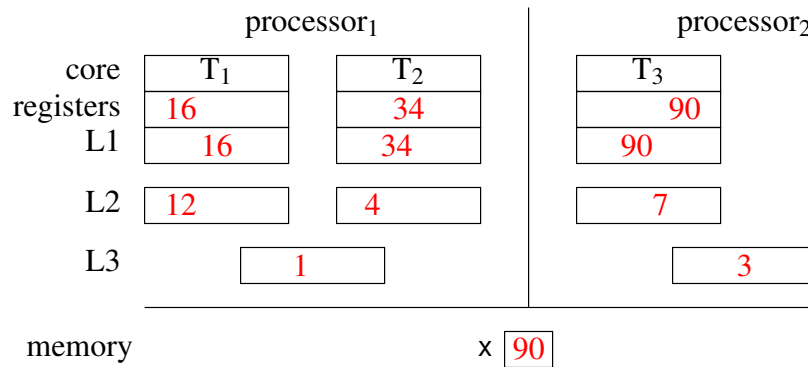
- Multi-level caches used, each larger but with diminishing speed (and cost).
- E.g., 64K L1 cache (32K Instruction, 32K Data) per core, 256K L2 cache per core, and 8MB L3 cache shared across cores.



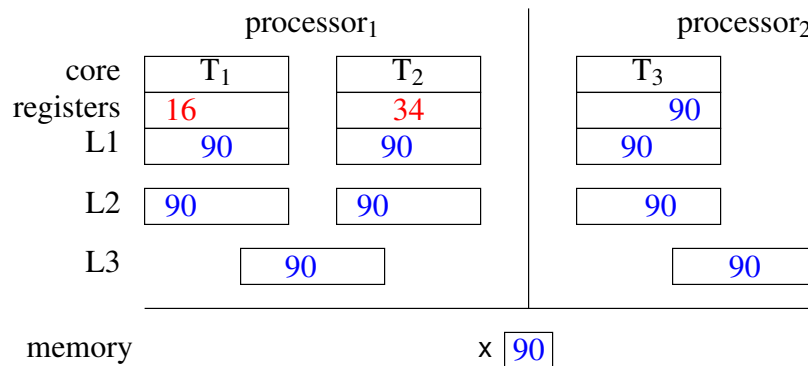
- Data reads logically percolate variables from memory up the memory hierarchy, making cache copies, to registers.
- Why is it necessary to eagerly move reads up the memory hierarchy?
- Data writes from registers to variables logically percolate down the memory hierarchy through cache copies to memory.
- Why is it advantageous to lazily move writes down the memory hierarchy?
- If OS moves program to another processor, all caching information is invalid and the program's data-hierarchy reforms.



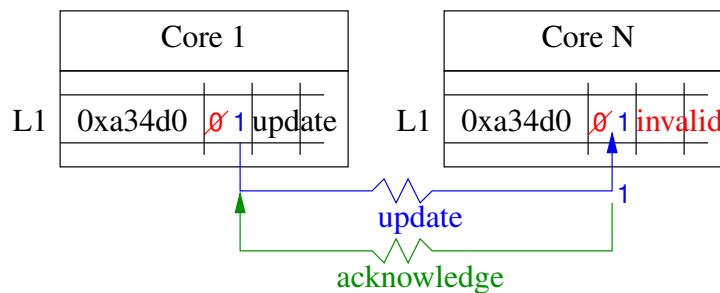
- Unlike registers, *all* cache values are shared across the computer.
- Hence, variable can be replicated in a large number of locations.
- Without cache coherence for shared variable x (madness)



- With cache coherence (snooping or directory-based) for shared variable x



- **Cache coherence** is hardware protocol ensuring update of duplicate data.
- **Cache consistency** addresses *when* processor sees update ⇒ bidirectional synchronization.
- **Prevent flickering and scrambling during simultaneous R/W or W/W.**



- Eager cache-consistency means data changes appear instantaneous by waiting for acknowledge from all cores (complex/expensive).

- Lazy cache-consistency allows reader to see own write before acknowledgement  $\Rightarrow$  **concurrent programs read stale data!**
  - writes eventually appear in (largely) same order as written
  - critical section works as writes to shared variable appear before write to lock release
  - otherwise, spin (lock) until write appears
- If threads continually read/write same memory locations, they invalidate duplicate cache lines, resulting in excessive cache updates.
  - called **cache thrashing**
  - updated value bounces from one cache to the next
- Because cache line contains multiple variables, cache thrashing can occur inadvertently, called **false sharing**.
- Thread 1 read/writes  $x$  while Thread 2 read/writes  $y \Rightarrow$  no direct shared access, but indirect sharing as  $x$  and  $y$  share cache line.
  - Fix by separating  $x$  and  $y$  with sufficient storage (padding) to be in next cache line.
  - Difficult for dynamically allocated variables as memory allocator positions storage.

thread 1	thread 2
<code>int *x = new int</code>	<code>int *y = new int;</code>

x and y may or may not be on same cache line.

### 10.3 Concurrent Optimizations

- In sequential execution, **strong memory ordering**: reading always returns last value written.
- In concurrent execution, **weak memory ordering**: reading can return previously written value or value written in future.
  - happens on multi-processor because of scheduling and buffering (see scrambling/flickering in Section 5.18.6, p. 89 and freshness/staleness in Section 6.4.4.4, p. 130).
  - notion of **current** value becomes blurred for shared variables unless everyone can see values assigned simultaneously.
- SME control order and speed of execution, otherwise non-determinism causes random results or failure (e.g., race condition, Section 7.1, p. 139).
- Sequential sections accessing private variables can be optimized normally **but not across concurrent boundaries**.
- Concurrent sections accessing shared variables can be corrupted by sequential optimizations  $\Rightarrow$  restrict optimizations to ensure correctness.
- For correctness and performance, identify concurrent code and only restrict *its* optimization.

- What/how to restrict depends on what sequential assumptions are implicitly applied by hardware and compiler (programming language).
- Following examples show how sequential optimizations cause failures in concurrent code.

### 10.3.1 Disjoint Reordering

- $R_x \rightarrow R_y$  allows  $R_y \rightarrow R_x$

Reordering disjoint reads does not cause problems. Why?

- $W_x \rightarrow R_y$  allows  $R_y \rightarrow W_x$

- In Dekker entry protocol (see Section 5.18.6, p. 89)

1	<b>me = WantIn;</b> // W		1	<b>temp = you;</b> // R
2	<b>while ( you == WantIn ) {</b> // R		1	<b>me = WantIn;</b> // W
3	...}		2	<b>while ( temp == WantIn ) {</b>
			3	...}

both threads read DontWantIn, both set WantIn, both see DontWantIn, and proceed.

- $R_x \rightarrow W_y$  allows  $W_y \rightarrow R_x$

- In synchronization flags (see Section 5.12, p. 83), allows interchanging lines 1 & 3 for Cons:

	Cons			Cons
1	<b>while ( ! Insert );</b> // R		3	data = <b>Data;</b> // W
2	Insert = <b>false;</b>		1	<b>while ( ! Insert );</b> // R
3	data = <b>Data;</b> // W		2	Insert = <b>false;</b>

allows reading of uninserted data

- $W_x \rightarrow W_y$  allows  $W_y \rightarrow W_x$

- In synchronization flags (see Section 5.12, p. 83), allows interchanging lines 1 & 2 in Prod and lines 3 & 4 in Cons:

	Prod			Prod
1	<b>Data = i;</b> // W		2	<b>Insert = true;</b> // W
2	<b>Insert = true;</b> // W		1	<b>Data = i;</b> // W

allows reading of uninserted data

- In Peterson's entry protocol, allows interchanging lines 1 & 2 (see Section 5.18.7, p. 91):

1	<b>me = WantIn;</b> // W		2	<b>::Last = &amp;me;</b> // W
2	<b>::Last = &amp;me;</b> // W		1	<b>me = WantIn;</b> // W

allows race before either task sets its intent and both proceed

- Compiler uses all of these reorderings to break mutual exclusion:

<b>lock.acquire()</b>	// critical section	<b>lock.acquire()</b>	<b>lock.acquire()</b>
// critical section	<b>lock.acquire()</b>	<b>lock.release();</b>	<b>lock.release();</b>
<b>lock.release();</b>	<b>lock.release();</b>		// critical section

- moves lock entry/exit after/before critical section because entry/exit variables not used in critical section.
- E.g., **double-check locking** for *concurrent* singleton-pattern:

```

int * ip = nullptr;           // shared (volatile for correctness)
...
if ( ip == nullptr ) {      // no storage ?
    lock.acquire();         // attempt to get storage (race)
    if ( ip == nullptr ) {  // still no storage ? (double check)
        ip = new int( 0 );  // obtain and initialize storage
    }
    lock.release();
}

```

Why do the first check? Why do the second check?

- Fails if last two writes are reordered,  $W_{malloc}$  and  $W_{ip}$ , disjoint variables:

```

call    malloc                // new storage address returned in r1
st    #0,(r1)               // initialize storage
st    r1,ip                 // initialize pointer

```

see ip but uninitialized.

### 10.3.2 Eliding

- For high-level language, compiler decides when/which variables are loaded into registers and for how long.
- Elide reads (loads) by copying (replicating) value into a register:

```

Task1                Task2
...
flag = false // write    register = flag; // one read, auxiliary variable
                        while ( register ); // cannot see change by T1

```

- Hence, variable logically disappears for duration in register.
- ⇒ task spins forever in busy loop if R before W.
- Also, elide meaningless sequential code:

```

sleep( 1 ); // unnecessary in sequential program

```

⇒ task misses signal by not delaying

### 10.3.3 Replication

- Why is there a benefit to reorder R/W?
- Modern processors increase performance by executing multiple instructions in parallel (data flow, precedence graph (see 6.4.1)) on **replicated hardware**.
  - internal pool of instructions taken from program order
  - begin simultaneous execution of instructions with inputs

- collect results from finished instructions
- feed results back into instruction pool as inputs
- $\Rightarrow$  instructions with independent inputs execute out-of-order
- From sequential perspective, disjoint reordering is *unimportant*, so hardware starts many instruction simultaneously.
- From concurrent perspective, disjoint reordering is *important*.

## 10.4 Memory Model

- Manufacturers define set of optimizations performed implicitly by processor.
- Set of optimizations indirectly define a **memory model**.

Relaxation Model	W $\rightarrow$ R	R $\rightarrow$ W	W $\rightarrow$ W	Lazy cache update
atomic consistent (AT)				
sequential consistency (SC)				✓
total store order (TSO)	✓			✓
partial store order (PSO)	✓	✓		✓
weak order (WO)	✓	✓	✓	✓
release consistency (RC)	✓	✓	✓	✓

- AT has events occur instantaneously  $\Rightarrow$  slow or impossible (distributed).
- SC accepts all events cannot occur instantaneously  $\Rightarrow$  may read old values
- SC still strong enough for software mutual-exclusion (Dekker 5.18.6 / Peterson 5.18.7).
  - SC often considered minimum model for concurrency.
- No hardware supports just AT/SC.
  - TSO (x86/SPARC), PSO, WO (ARM, Alpha), RC (PowerPC)

## 10.5 Preventing Optimization Problems

- All optimization problems result from races on shared variables.
- If shared data is protected by locks (implicit or explicit),
  - locks define the sequential/concurrent boundaries,
  - boundaries must preclude optimizations that affect concurrency.
- Called **race free** as synchronization and mutual exclusion preclude races.
- However, race free does have races.

- Races are internal to locks, which lock programmer must deal with.
- Two approaches:
  - ad hoc: programmer manually augments all data races with pragmas to restrict compiler/hardware optimizations: not portable but often optimal.
  - formal: language has memory model and mechanisms to abstractly define races in program: portable but often baroque and suboptimal.
- data access / compiler (C/C++): **volatile** qualifier
  - Force variable loads and stores to/from registers (at **sequence points**)
  - created for longjmp or force access for memory-mapped devices
  - for architectures with few registers, practically all variables are implicitly volatile. Why?
  - Java **volatile** / C++11 **atomic** stronger  $\Rightarrow$  prevent eliding **and** disjoint reordering  $\Rightarrow$  SC
- program order / compiler (static): disable inlining, **asm**("" ::: "memory");
- memory order / runtime (dynamic): sfence, lfence, mfence (x86)
  - guarantee previous stores and/or loads are completed, before continuing.
- atomic operations test-and-set, which often imply fencing
- cache is normally invisible and does not cause issues (except for DMA)
- mechanisms to fix issues are specific to compiler or platform
  - difficult, low-level, diverse semantics, not portable  $\Rightarrow$  **read carefully!**
- Dekker for TSO:

```

#define CALIGN __attribute__(( aligned (64) )) // cache-line alignment
#define Pause() __asm__ __volatile__ ( "pause" ::: ) // efficient busy wait
enum Intent { WantIn, DontWantIn };
//#define ATOMIC
#ifndef ATOMIC
#define Fence() __asm__ __volatile__ ( "mfence" ) // prevent hardware reordering
typedef volatile Intent VIntent;
typedef volatile Intent * volatile VIntentPtr;
#else
#define Fence()
#include <atomic>
typedef std::atomic<Intent> VIntent;
typedef std::atomic<std::atomic<Intent> *> VIntentPtr;
#endif

```

```

_Task Dekker {
    VIntent & me, & you;
    VIntentPtr & Last;
    void main() {
        for ( unsigned int i = 0; i < 1'000'000; i += 1 ) {
            for ( ;; ) { // entry protocol
                me = WantIn; // high priority
                Fence();
                if ( you == DontWantIn ) break;
                if ( Last == &me ) { // high priority ?
                    me = DontWantIn;
                    while ( Last == &me ) Pause(); // low priority
                }
                Pause();
            }
            CriticalSection(); // critical section
            Last = &me; // exit protocol
            me = DontWantIn;
        }
    }
};

public:
    Dekker( VIntent & me, VIntent & you, VIntentPtr *& Last ) :
        me(me), you(you), Last(Last) {}
};

int main() {
    VIntent me CALIGN = DontWantIn, you CALIGN = DontWantIn,
    VIntentPtr Last = &me;
    Dekker t0(me, you, Last), t1(you, me, Last);
};

```

- C++ atomic automatically fences shared variables, but can be suboptimal.
- Locks built with these features ensure SC for protected shared variables.
  - **no user races and strong locks** ⇒ **SC memory model**





## 11 Other Approaches

### 11.1 Atomic (Lock-Free) Data-Structure

- **Lock free** data-structure have operations, which are critical sections, but performed without **ownership**.
  - e.g., add/remove node without any blocking duration (operation takes constant atomic time)
- Lock-free is still locking (misnomer)  $\Rightarrow$  spin for conceptual lock  $\Rightarrow$  busy-waiting (starvation).
- If guarantees eventual progress, called **wait free**.

#### 11.1.1 Compare and Set Instruction

- The compare-and-set(assign) instruction performs an atomic compare and conditional assignment CAS (erroneously called compare-and-swap).

```
int Lock = OPEN; // shared

bool CAS( int & val,
  int comp, int nval ) {
  // begin atomic
  if ( val == comp ) {
    val = nval;
    return true;
  }
  return false;
  // end atomic
}

void Task::main() { // each task does
  while ( ! CAS( Lock, OPEN, CLOSED ) );
  // critical section
  Lock = OPEN;
}
```

- if compare/assign returns true  $\Rightarrow$  loop stops and lock is set to closed
- if compare/assign returns false  $\Rightarrow$  loop executes until the other thread sets lock to open
- Alternative implementation assigns comparison value with the value when not equal.

```
bool CAV( int & val, int & comp, int nval ) {
  // begin atomic
  if ( val == comp ) {
    val = nval;
    return true;
  }
  comp = val; // return changed value
  return false;
  // end atomic
}
```

- Assignment when unequal useful to restart operations with new changed value.

### 11.1.2 Lock-Free Stack

- E.g., build a stack with lock-free push and pop operations.

```

class Stack {
    Node * top;           // pointer to stack top
public:
    struct Node {
        // data
        Node * next;     // intrusive pointer to next node
    };
    void push( Node & n );
    Node * pop();
};

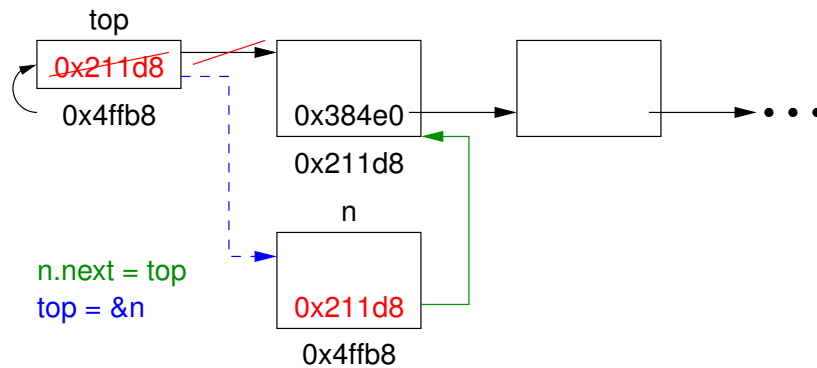
```

- Use CAS to atomically update top pointer when nodes pushed or popped concurrently.

```

void Stack::push( Node & n ) {
    for ( ;; ) {           // busy wait
        n.next = top;     // link new node to top node
        if ( CAS( top, n.next, &n ) ) break; // attempt to update top node
    }                     // top = &n
}

```

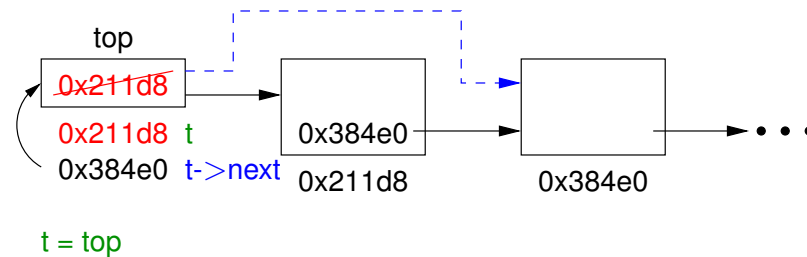


- Create new node, n, at 0x4ffb8 to be added.
- Set n.next to top.
- CAS tries to assign new top &n to top.
- CAS fails if top changed since copied to n.next
- If CAS failed, update n.next to top, and try again.
- CAS succeeds when top == n.next, i.e., no push or pop between setting n.next and trying to assign &n to top.
- CAS copies changed value to n.next, so eliminates resetting t = top in busy loop.

```

Node * Stack::pop() {
    Node * t;
    for ( ;; ) {
        // busy wait
        t = top; // copy current top
        if ( t == nullptr ) return t; // empty list ?
        if ( CAS( top, t, t->next ) ) return t; // attempt to update top node
        // top = t->next
    }
}

```



- Copy top node, 0x4ffb8, to t for removal.
- If not empty, attempt CAS to set new top to next node, t->next.
- CAS fails if top changed since copied to t.
- If CAS failed, update t to top, and try again.
- CAS succeeds when top == t->next, i.e., no push or pop between setting t and trying to assign t->next to top.
- CAV copies the changed value into t, so eliminates resetting t = top in busy loop.

### 11.1.3 ABA problem

- Pathological failure for series of pops and pushes, called **ABA problem**.
- Given stack with 3 nodes:
 

top → A → B → C
- Popping task,  $T_i$ , sets t to A and dereferenced t->next to get next node B for argument to CAS.
- $T_i$  is now time-sliced **before the CAS**, and while blocked, nodes A and B are popped, and A is pushed again:

top → A → C // B is gone!

- When  $T_i$  restarts, CAS successfully removes A as same header before time-slice.
- **But now incorrectly sets top to its next node B:**

top → B → ???

stack is now corrupted!!!

### 11.1.4 Hardware Fix

- Probabilistic solution for stack exists using double-wide CAVD instruction, which compares and assigns 64/128-bit values for 32/64-bit architectures.

```

bool CAVD( uintS_t &val, uintS_t &comp, uintS_t nval ) {
    // begin atomic
    if ( val == comp ) {           // 64/128-bit compare
        val = nval;                // 64/128-bit assignment
        return true;
    }
    comp = val;                    // 64/128-bit assignment
    return false;
    // end atomic
}

```

- Now, associate counter (ticket) with header node:

```

class Stack {
    union Link {
        struct {                 // 32/64-bit x 2
            Node * top;         // pointer to stack top
            uintptr_t count;    // count each push
        };
        uintS_t atom;           // 64/128-bit integer
    } link;
public:
    struct Node {
        // resource data
        Link next;              // pointer to next node/count (resource)
    };
    Stack() { link.atom = 0; }
    void push( Node & n );
    Node * pop();
};

```

- Increment counter in push (**only count pushes**) so pop can detect ABA if node re-pushed.

```

void Stack::push( Node & n ) {
    n.next = link;              // atomic assignment unnecessary
    for ( ;; ) {                 // busy wait
        if ( CAVD( link.atom, n.next.atom,
                    (Link){ &n, n.next.count + 1 }.atom ) ) break;
    }
    // link.top = &n, link.count = n.next.count + 1
}

```

- CAVD used to copy entire header to n.next, as structure assignment (2 fields) is not atomic.
- In busy loop, copy local idea of top to next of new node to be added.
- CAVD tries to assign new top-header to (h).

- If top has not changed since copied to n.next, update top to n (new top), and **increment counter**.
- If top has changed, CAVD copies changed values to n.next, so try again.

```

Node * Stack::pop() {
    Link t = link;           // atomic assignment unnecessary
    for ( ;; ) {           // busy wait
        if ( t.top == nullptr ) return nullptr; // empty stack ?
        if ( CAVD( link.atom, t.atom,
                   (Link){ t.top->next.top, t.count }.atom ) ) return t.top;
    }                       // link.top = t.top->next.top, link.count = t.count
}

```

- CAVD used to copy entire header to t, as structure assignment (2 fields) is not atomic.
  - In busy loop, check if pop on empty stack and return **nullptr**.
  - If not empty, CAVD tries to assign new top t.top->next.top,t.count to h.
  - If top has not changed since copied to t, update top to t.top->next.top (new top).
  - If top has changed, CAVD copies changed values to t, so try again.
- ABA problem (mostly) fixed:
    - top,3 → A → B → C
  - Popping task,  $T_i$ , has t set to A,3 and dereferenced B from t.top->next in argument of CAVD.
  - $T_i$  is time-sliced, and while blocked, nodes A and B are popped, and A is pushed again:
    - top,4 → A → C // adding A increments counter
  - When  $T_i$  restarts, CAVD fails as header A,3 not equal top A,4.
  - Only probabilistic correct as counter finite (like ticket counter).
    - task  $T_i$  is time-sliced and sufficient pushes wrap counter to value stored in  $T_i$ 's header,
    - node A just happens to be at the top of the stack when  $T_i$  unblocks.
    - doubtful if failure arises, given 32/64-bit counter and pathological case.

## 11.2 Safe Memory Reclamation Problem

- All lock-free data-structures dereference a pointer to copy a link from a node, e.g., pop:

```

t = top;
<interrupted>
if ( CAS( top, t, t->next ) ) return t;

```

- However, a thread can be interrupted after copying top (before CAS).

- Another thread removes the top node, frees it, and its storage is returned to the OS.
- The interrupted thread restarts, dereferences  $t$ , but that address is invalid  $\Rightarrow$  segment fault.
- Normally, dereference is benign as storage is not removed from the address space (very rare).
- Fixing this life-time problem requires safe memory reclamation (SMR).
  - Complex (100s of lines of code) for advanced data structures (queue, deque, tree).
- All solutions must determinate when a node has no references (like garbage collection).
  - each thread maintains a list of accessed nodes, called **hazard pointers**
  - thread updates its hazard pointers while other threads are reading them
  - thread removes a node by hiding it on a private list and periodically scans the hazard lists of other threads for references to that node
  - if no pointers are found, the node can be freed
- For lock-free stack:  $x, y, z$  are memory addresses
  - first thread puts  $x$  on its hazard list
  - second thread cannot reuse  $x$ , because of hazard list
  - second thread must create new object at different location
  - first thread detects change
- Summary: locking versus lock-free
  - lock-free can only handle limited set of critical sections  
lock can protect arbitrarily complex critical section
  - lock-free has no ownership (hold-and-wait)  $\Rightarrow$  no deadlock
  - lock ownership  $\Rightarrow$  interrupted thread can underutilize critical section
  - lock-free without SMR does not ensure eventual progress (breaks rule 5)
  - no performance difference in general case

### 11.3 Exotic Atomic Instruction

- VAX computer has instructions to atomically insert and remove a node to/from the head or tail of a circular doubly linked list.

```

struct links {
    links * front, * back;
}
bool INSQUE( links &entry, links &pred ) {    // atomic execution
    // insert entry following pred
    return entry.front == entry.back;        // first node inserted ?
}
bool REMQUE( links &entry ) {                // atomic execution
    // remove entry
    return entry.front == null;              // last node removed ?
}

```

- MIPS processor has two instructions that generalize atomic read/write cycle: LL (load locked) and SC (store conditional).

- LL instruction loads (reads) a value from memory into a register, and sets a hardware **reservation** on the memory from which the value is fetched.
- Register value can be modified, even moved to another register.
- SC instruction stores (writes) new value back to original or another memory location.
- However, store is conditional and occurs only if no interrupt, exception, or write has occurred at LL reservation.
- Failure indicated by setting the register containing the value to be stored to 0.
- E.g., implement test-and-set with LL/SC:

```

int testSet( int &lock ) {    // atomic execution
    int temp = lock;            // read
    lock = 1;                   // write
    return temp;                // return previous value
}
testSet:                        // register $4 contains pointer to lock
    ll $2,$($4)                // read and lock location
    or $8,$2,1                 // set register $8 to 1 (lock | 1)
    sc $8,$($4)               // attempt to store 1 into lock
    beq $8,$0,testSet         // retry if interference between read and write
    j $31                    // return previous value in register $2

```

- Does not suffer from ABA problem.

```

Node * pop( Header &h ) {
    Node * t, next;
    for ( ;; ) {                // busy wait
        t = LL( top );
        if ( t == nullptr ) break; // empty list ?
        next = t->next
        if ( SC( top, next ) ) break; // attempt to update top node
    }
    return t;
}

```

- SC detects any *change* to top, whereas CAS only detects a specific value change to top (is top not equal to A).
- LL and SC are not required to match.
- However, most architectures support weak LL/SC.
  - \* reservation granularity may be cache line or memory block rather than word
  - \* no nesting or interleaving of LL/SC pairs, and prohibit memory access between LL and SC.
- Cannot implement atomic swap of 2 memory locations as two reservations are necessary (register to memory swap is possible).
- Hardware transactional memory allows 4, 6, 8 reservations, e.g., Advanced Synchronization Facility (ASF) proposal in AMD64.
- Like database **transaction** that optimistically executes change, and either commits changes, or rolls back and restarts if interference.
  - SPECULATE : start speculative region and clear zero flag ; next instruction checks for abort and branches to retry.
  - LOCK : MOV instructions indicates location for atomic access, but moves not visible to other CPUs.
  - COMMIT : end speculative region
    - \* if no conflict, make MOVs visible to other CPUs.
    - \* if conflict to any move locations, set failure, discard reservations and restore registers back to instruction following SPECULATE
- Can implement several data structures without ABA problem.
- Software Transactional Memory (STM) allows any number of reservations.
  - atomic blocks of arbitrary size:
 

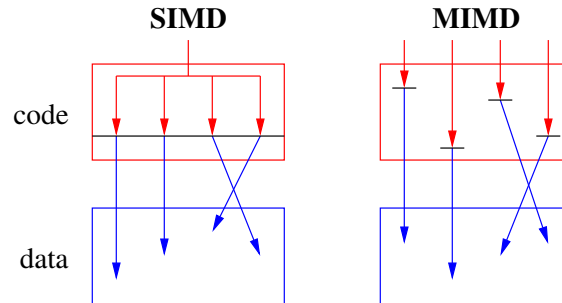
```

void push( header & h, node & n ) {
    atomic {
      n.next = top;
      top = &n;
    }
  }
          // SPECULATE
          // LOCK/MOV
          // COMMIT
          
```
  - records all memory locations read and written, and all values mutated.
    - \* bookkeeping costs and rollbacks typically result in performance degradation
  - alternative implementation inserts locks to protect shared access
    - \* finding all access is difficult and ordering lock acquisition is complex



## 11.4 General-Purpose GPU (GPGPU)

- Graphic Processing Unit (GPU) is a **coprocessor** to main computer, with separate memory and processors.
- GPU is a Single-Instruction Multiple-Data(Thread) (SIMD(T)) architecture versus Multiple-Instruction Multiple-Data (MIMD)



- SIMD occurs at instruction level, e.g., `i &= 0x34fe256`.

```
ld  r3, i
and r3, 0x34fe256
st  r3, i
```

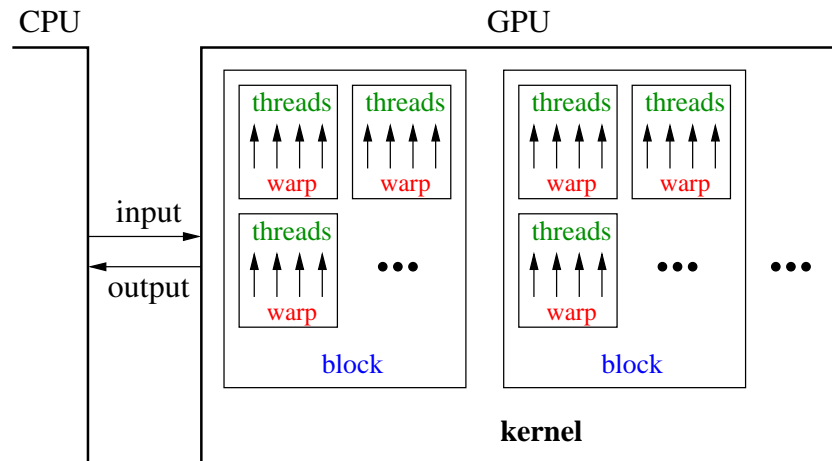
Does not loop 64 times “anding” each bit; 64 parallel “ands” are done simultaneously.

- Can `i += 1` be SIMD at the instruction level?
- Problems in branching code

```
if ( a[i] % 2 == 0 ) {
    a[i] /= 2;           // true threads
} else {
    a[i] += 3;         // false threads
}
```

- All threads test the condition (create mask of true and false)
  - true threads execute “then” instructions, false threads execute NOP (no-operation)
  - false threads execute “else” instructions, true threads execute NOP
- In general, critical path is time to execute both clauses of `if` (no speedup).
- Complex contortions to eliminate different forms of branching.
- GPU structure
  - **kernel** manages multiple blocks (loaded/controlled by CPU)
  - **block** executes the same code
  - **warp** synchronizes execution (one instruction decoder per warp)

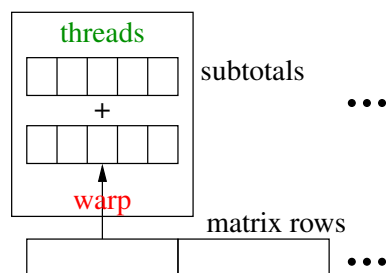
- **thread** computes value



- blocks may be barrier-synchronized
- synchronization among blocks  $\Rightarrow$  finishing kernel and launching new one
- Instead of cache to optimize latency in warp, large register file is used to optimize throughput.
  - GPUs have enough duplicate registers to store state of several warps.
- Kernel is memory-bound  $\Rightarrow$  data layout extremely important performance consideration.

```
// kernel routine, handle contiguous matrix, different ID for each thread
kernel void GPUsum( float * matrix[], float subtotals[], int rows ) {
#   define sub(m, r, c) ((typeof(m[0][0]) *)m)[r * rows + c]
    subtotals[ID] = 0.0;
    for ( int r = 0; r < rows; r += 1 )
        subtotals[ID] += sub( matrix, r, ID );
}
```

- Add rows by columns.



- Warps scheduled to run when their required data is loaded from memory.
- CPU sets up GPU memory, loads memory, launches code, retrieves results.

```

int main() {
    int rows, cols;
    cin >> rows >> cols;    // matrix size
    // optimal to use contiguous matrix
    float matrix[rows][cols], subtotals[rows], total = 0.0;
    // ... fill matrix
    float * matrix_d, * subtotals_d;    // matrix/subtotals buffer on GPU
    // allocate space on GPU
    GPUAlloc( &matrix_d, sizeof(matrix) );
    GPUAlloc( &subtotals_d, sizeof(subtotals) );
    // copy matrix to GPU
    GPUMemcpy( matrix_d, matrix, sizeof(matrix), GPUMemcpyHostToDevice );
    // compute matrix sum on GPU
    GPUsum<<< 1, cols >>>( matrix_d, subtotals_d, rows );
    // do asynchronous work!!!
    // copy subtotals from GPU, may block
    GPUMemcpy( subtotals, subtotals_d, sizeof(subtotals), GPUMemcpyDeviceToHost );
    for ( int i = 0; i < cols; i += 1 ) total += subtotals[i];
    cout << total << endl;
}

```

- Most modern multi-core CPUs have similar model using vector-processing.
  - Simulate warps and use concurrency framework ( $\mu$ C++) to schedule blocks.

## 11.5 Concurrency Languages

### 11.5.1 Ada 95

- Restricted implicit (automatic) signal monitor, e.g., monitor bounded-buffer.

```

protected type buffer is -- _Monitor
    entry insert( elem : in ElemType ) when count < Size is -- mutex member
    begin
        -- add to buffer
        count := count + 1;
    end insert;
    entry remove( elem : out ElemType ) when count > 0 is -- mutex member
    begin
        -- remove from buffer, return via parameter
        count := count - 1;
    end remove;
private:
    ... // buffer declarations
    count : Integer := 0;
end buffer;

```

- The **when** clause is only be used at start of entry routine not within.
- The **when** expression can contain only global-object variables; parameter or local variables are disallowed  $\Rightarrow$  no direct dating-service.

- Eliminate restrictions and dating service is solvable.

```

_Monitor DatingService {
    AUTOMATIC_SIGNAL;
    int girls[noOfCodes], boys[noOfCodes]; // count girls/boys waiting
    bool exchange; // performing phone-number exchange
    int girlPhoneNo, boyPhoneNo; // communication variables
public:
    int girl( int phoneNo, int ccode ) {
        girls[ccode] += 1;
        if ( boys[ccode] == 0 ) { // no boy waiting ?
            WAITUNTIL( boys[ccode] != 0, , ); // use parameter, not at start
            boys[ccode] -= 1; // decrement dating pair
            girls[ccode] -= 1;
            girlPhoneNo = phoneNo; // girl' s phone number for exchange
            exchange = false; // wake boy
        } else {
            girlPhoneNo = phoneNo; // girl' s phone number before exchange
            exchange = true; // start exchange
            WAITUNTIL( ! exchange, , ); // wait until exchange complete, not at start
        }
        EXIT();
        return boyPhoneNo;
    }
    // boy
};

```

- Threads provided by task object using kernel threads, e.g., task bounded-buffer.

```

task type buffer is -- _Task
... -- buffer declarations
count : integer := 0;
begin -- thread starts here (task main)
loop
    select -- _Accept
        when count < Size => -- guard
        accept insert(elem : in ElemType) do -- mutex member
            -- add to buffer
            count := count + 1;
        end;
        -- executed if this accept called
    or
        when count > 0 => -- guard
        accept remove(elem : out ElemType) do -- mutex member
            -- remove from buffer, return via parameter
            count := count - 1;
        end;
    end select;
end loop;
end buffer;
var b : buffer -- create a task

```

- **select** is external scheduling and only appears in **task** main.
- Hence, Ada has no direct internal-scheduling mechanism, i.e., no condition variables.
- Instead a **requeue** statement can be used to make a **blocking** call to another (usually non-public) mutex member of the object.
- The original call is re-blocked on that mutex member's entry queue, which can be subsequently accepted when it is appropriate to restart it.
- However, all **requeue** techniques suffer the problem of dealing with accumulated temporary results:
  - If a call must be postponed, its temporary results must be returned and bundled with the initial parameters before forwarding to the mutex member handling the next step,
  - or the temporary results must be re-computed at the next step (if possible).
- In contrast, waiting on a condition variable automatically saves the execution location and any partially computed state.

### 11.5.2 SR/Concurrent C++

- SR and Concurrent C++ have tasks with external scheduling using an **accept** statement.
- But no condition variables or **requeue** statement.
- To ameliorate lack of internal scheduling add a **when** and **by** clause on the **accept** statement.
- **when** clause is allowed to reference caller's arguments via parameters of mutex member:

```

select
  accept mem( code : in Integer )
    when code % 2 = 0 do ...    -- accept call with even code
or
  accept mem( code : in Integer )
    when code % 2 = 1 do ...    -- accept call with odd code
end select;
```

- **when** placed after the **accept** clause so parameter names are defined.
- **when** referencing parameter  $\Rightarrow$  implicit search of waiting tasks on mutex queue  $\Rightarrow$  locking mutex queue.
- Select longest waiting if multiple true **when** clauses.
- **by** clause is calculated for each true **when** clause and the **minimum by** clause is selected.

```

select
  accept mem( code : in Integer )
    when code % 2 = 0 by -code do ...-- accept largest even code
or
  accept mem( code : in Integer )
    when code % 2 = 1 by code do ...-- accept smallest odd code
end select;

```

- Select longest waiting if multiple by clauses with same minimum.
- **by** clause exacerbates the execution cost of computing **accept** clause.
- While **when/by** removes some internal scheduling and/or requeues, constructing expressions can be complex.
- Still situations that cannot be handled, e.g., if selection criteria involves multiple parameters:
  - select lowest even value of code1 and highest odd value of code2 if there are multiple lowest even values.
  - selection criteria involves information from other mutex queues such as the dating service (girl must search the boy mutex queue).
- Often simplest to unconditionally accept a call allowing arbitrarily examination, and possibly postpone (internal scheduling).

### 11.5.3 Java

- Java's concurrency constructs are derived from Modula-3.
- Java Thread is like  $\mu\text{C++}$  uBaseTask, and all tasks must explicitly inherit from it:

```

class Thread implements Runnable {
  public Thread();
  public Thread(String name);
  public String getName();
  public void setName(String name);
  public void run(); // uC++ main
  public synchronized void start();
  public static Thread currentThread();
  public static void yield();
  public final void join();
}

```

- Tasks must explicitly inherit from Thread:

```

class MyTask extends Thread { // inheritance
    private int arg;           // communication variables
    private int result;
    public MyTask() {...}     // task constructors
    public void run() {...}   // task main
    public int result() {...} // return result
    // unusual to have more members
}

```

- Java requires explicit starting of a thread by calling `start` after the thread's declaration.  
⇒ coding convention to start thread or inheritance is precluded (can only start a thread once)
- Thread starts in member `run` and are kernel threads.
- Termination synchronization is accomplished by calling `join`.
- Returning a result on thread termination is accomplished by member(s) returning values from the task's global variables.

```

mytask th = new MyTask(...); // create and initialized task
th.start();                   // start thread
// concurrency
th.join();                    // wait for thread termination
a2 = th.result();             // retrieve answer from task object

```

- Like  $\mu\text{C++}$ , when the task's thread terminates, it becomes an object, hence allowing the call to `result` to retrieve a result.
- (see Section 8.11, p. 165 for monitors)
- While it is possible to have public **synchronized** members of a task:
  - no mechanism to manage direct calls, i.e., no `accept` statement
  - ⇒ complex emulation of external scheduling with internal scheduling for direct communication
- Java now has “virtual” (light-weight) threads (project Loom) (requires `start` and `join`).

#### 11.5.4 Go

- Non-object-oriented, light-weight (like  $\mu\text{C++}$ ), preemptive threads (called **goroutine**).
- **go** statement (like `start/fork`) creates new user thread running in routine.

```

go foo( 3, f ) // start thread in routine foo

```

- Arguments may be passed to goroutine but return value is discarded.
- **Cannot reference goroutine object** ⇒ no direct communication.

- All threads terminate silently when program terminates.
- Threads synchronize/communicate via **channel** (CSP)
  - ⇒ **paradigm shift from routine call.**
- Channel is a typed shared buffer with 0 to N elements.

```
ch1 := make( chan int, 100 )    // integer channel with buffer size 100
ch2 := make( chan string )     // string channel with buffer size 0
ch2 := make( chan chan string ) // channel of channel of strings
```

- Buffer size > 0 ⇒ up to N asynchronous calls; otherwise, synchronous call.
- Operator **<-** performs send/receive.
  - send: `ch1 <- 1` // channel left-hand side
  - receive: `s <- ch2` // channel right-hand side
- Channel can be constrained to only send or receive; otherwise bi-directional.
- More like futures and `_Select` with asynchronous call.
- Use synchronous (0-size buffers) to match  $\mu\text{C++}$  synchronous `_Accept`.

```
package main
import "fmt"
func main() {

    type Msg struct{ i, j int }
    ch1 := make( chan int )
    ch2 := make( chan float32 )
    ch3 := make( chan Msg )
    hand := make( chan string )
    shake := make( chan string )
    gortn := func() {
        var i int; var f float32; var m Msg
        L: for {
            select { // wait for messages
                case i = <- ch1: fmt.Println( i )
                case f = <- ch2: fmt.Println( f )
                case m = <- ch3: fmt.Println( m )

                case <- hand: break L // sentinel
            } // select
        } // for
        shake <- "SHAKE" // completion
    } // gortn
    go gortn() // start thread in gortn
    ch1 <- 0 // different messages
    ch2 <- 2.5
    ch3 <- Msg{1, 2}
    hand <- "HAND" // sentinel value
    <-shake // wait for completion
}
```

```
#include <iostream>
using namespace std;
_Task Gortn {
public:
    struct Msg { int i, j; };
    void mem1( int i ) { Gortn::i = i; }
    void mem2( float f ) { Gortn::f = f; }
    void mem3( Msg m ) { Gortn::m = m; }
private:
    int i; float f; Msg m; // communication
    void main() {

        L: for ( ;; ) {

            _Accept( mem1 ) cout << i << endl;
            or _Accept( mem2 ) cout << f << endl;
            or _Accept( mem3 ) cout << "{" << m.i
                << " " << m.j << "}" << endl;
            or _Accept( ~Gortn ) break L;

        } // for
    } // main
}; // Gortn
int main() {
    Gortn gortn; // start thread in task
    gortn.mem1( 0 );
    gortn.mem2( 2.5 );
    gortn.mem3( (Gortn::Msg){ 1, 2 } );
} // wait for completion
```



- Locks

```

type Mutex           // mutual exclusion lock
func (m * Mutex) Lock()
func (m * Mutex) Unlock()
type Cond           // synchronization lock
func NewCond(l Locker) * Cond
func (c * Cond) Broadcast()
func (c * Cond) Signal()
func (c * Cond) Wait()
type Once           // singleton-pattern
func (o * Once) Do(f func())

```

```

type RWMutex        // readers/writer lock
func (rw * RWMutex) Lock()
func (rw * RWMutex) RLock()
func (rw * RWMutex) RLocker() Locker
func (rw * RWMutex) RUnlock()
func (rw * RWMutex) Unlock()

```

```

type WaitGroup      // countdown lock
func (wg * WaitGroup) Add(delta int)
func (wg * WaitGroup) Done()
func (wg * WaitGroup) Wait()

```

- Atomic operations

```

func AddInt32(val * int32, delta int32) (new int32)
func AddInt64(val * int64, delta int64) (new int64)
func AddUint32(val * uint32, delta uint32) (new uint32)
func AddUint64(val * uint64, delta uint64) (new uint64)
func AddUintptr(val * uintptr, delta uintptr) (new uintptr)
func CompareAndSwapInt32(val * int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(val * int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(val * unsafe.Pointer, old, new unsafe.Pointer) (swapped bool)
func CompareAndSwapUint32(val * uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(val * uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(val * uintptr, old, new uintptr) (swapped bool)
func LoadInt32(addr * int32) (val int32)
func LoadInt64(addr * int64) (val int64)
func LoadPointer(addr * unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr * uint32) (val uint32)
func LoadUint64(addr * uint64) (val uint64)
func LoadUintptr(addr * uintptr) (val uintptr)
func StoreInt32(addr * int32, val int32)
func StoreInt64(addr * int64, val int64)
func StorePointer(addr * unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr * uint32, val uint32)
func StoreUint64(addr * uint64, val uint64)
func StoreUintptr(addr * uintptr, val uintptr)

```

### 11.5.5 C++11 Concurrency

- C++11 `std::thread` is an OO wrapper over pthreads (use compilation flag `-pthread`)  $\Rightarrow$  kernel threads.
- Thread creation: start/wait (fork/join) approach.

```

class thread {
public:
    template <class Fn, class... Args>
        explicit thread( Fn && fn, Args &&... args );
    void join();                // termination synchronization
    bool joinable() const;     // true => joined, false otherwise
    void detach();             // independent lifetime
    id get_id() const;         // thread id
};

```

- Passing multiple arguments uses C++11's variadic template feature to provide a type-safe call chain via thread constructor to the *callable* routine.
- Any entity that is *callable* (functor) may be started:

```

#include <thread>
void hello( const string & s ) {           // callable
    cout << "Hello " << s << endl;
}
class Hello {                              // functor
    int result;
public:
    void operator()( const string & s ) { // callable
        cout << "Hello " << s << endl;
    }
};

int main() {
    thread t1( hello, "Peter" );           // start thread in routine "hello"
    Hello h;                               // thread object
    thread t2( h, "Mary" );                // start thread in functor "h"
    // work concurrently
    t1.join();                             // termination synchronization
    // work concurrently
    t2.join();                             // termination synchronization
} // must join before closing block

```

- Thread starts implicitly at point of declaration.
- Instead of join, thread can run independently by detaching:

```

t1.detach();           // "t1" must terminate for program to end

```

- Beware dangling pointers to local variables:

```
int main() {
    string s( "Fred" );           // local variable
    thread t( hello, s );        // reference to s
    t.detach();                  // Free Willy
    pthread_exit( 0 );           // allows detached threads to continue
} // "s" deallocated and "t" running with reference to "s"
```

- **It is an error to deallocate thread object before join or detach.**

- Locks

- mutex, recursive, timed, recursive-timed

```
class mutex {
public:
    void lock();           // acquire lock
    void unlock();        // release lock
    bool try_lock();      // nonblocking acquire
};
```

- condition

```
class condition_variable {
public:
    void notify_one();    // unblock one
    void notify_all();   // unblock all
    void wait( mutex &lock ); // atomically block & release lock
};
```

- Scheduling is no-priority nonblocking  $\Rightarrow$  barging  $\Rightarrow$  wait statements must be in while loops to recheck conditions.

```
#include <mutex>
class BoundedBuffer {           // simulate monitor
    // buffer declarations
    mutex mlock;               // monitor lock
    condition_variable empty, full;
    void insert( int elem ) {
        mlock.lock();
        while (count == Size ) empty.wait( mlock ); // release lock
        // add to buffer
        count += 1;
        full.notify_one();
        mlock.unlock();
    }
};
```

```

int remove() {
    mlock.lock();
    while( count == 0 ) full.wait( mlock ); // release lock
    // remove from buffer
    count -= 1;
    empty.notify_one();
    mlock.unlock();
    return elem;
}
};

```

- Futures

```

#include <future>
big_num pi( int decimal_places ) {...}
int main() {
    future<big_num> PI = async( pi, 1200 ); // PI to 1200 decimal places
    // work concurrently
    cout << "PI " << PI.get() << endl; // block for answer
}

```

- Atomic types/operations

atomic\_flag, atomic\_bool, atomic\_char, atomic\_schar, atomic\_uchar, atomic\_short, atomic\_ushort, atomic\_int, atomic\_uint, atomic\_long, atomic\_ulong, atomic\_llong, atomic\_ullong, atomic\_wchar\_t, atomic\_address, atomic<T>

```

typedef struct atomic_itype {
    bool operator=(int-type) volatile;
    void store(int-type) volatile;
    int-type load() const volatile;
    int-type exchange(int-type) volatile;
    bool compare_exchange(int-type &old_value, int-type new_value) volatile;
    int-type fetch_add(int-type) volatile;
    int-type fetch_sub(int-type) volatile;
    int-type fetch_and(int-type) volatile;
    int-type fetch_or(int-type) volatile;
    int-type fetch_xor(int-type) volatile;

    int-type operator++() volatile;
    int-type operator++(int) volatile;
    int-type operator--() volatile;
    int-type operator--(int) volatile;
    int-type operator+=(int-type) volatile;
    int-type operator-=(int-type) volatile;
    int-type operator&=(int-type) volatile;
    int-type operator|=(int-type) volatile;
    int-type operator^=(int-type) volatile;
} atomic_itype;

```

## 11.6 Threads & Locks Library

### 11.6.1 java.util.concurrent

- Java library is sound because of memory-model and language is concurrent aware.
- Synchronizers : Semaphore (counting), CountdownLatch, CyclicBarrier, Exchanger, Condition, Lock, ReadWriteLock
- Use new locks to build a monitor with multiple condition variables.

```

class BoundedBuffer {                                     // simulate monitor
    // buffer declarations
    final Lock mlock = new ReentrantLock();             // monitor lock
    final Condition empty = mlock.newCondition();
    final Condition full = mlock.newCondition();
    public void insert( Object elem ) throws InterruptedException {
        mlock.lock();
        try {
            while (count == Size ) empty.await(); // release lock
            // add to buffer
            count += 1;
            full.signal();
        } finally { mlock.unlock(); } // ensure monitor lock is unlocked
    }

    public Object remove() throws InterruptedException {
        mlock.lock();
        try {
            while( count == 0 ) full.await(); // release lock
            // remove from buffer
            count -= 1;
            empty.signal();
            return elem;
        } finally { mlock.unlock(); } // ensure monitor lock is unlocked
    }
}

```

- Condition is nested class within ReentrantLock  $\Rightarrow$  condition implicitly knows its associated (monitor) lock.
- Scheduling is still no-priority nonblocking  $\Rightarrow$  barging  $\Rightarrow$  wait statements must be in while loops to recheck condition.
- No connection with implicit condition variable of an object.
- **Do not mix implicit and explicit condition variables.**
- Executor/Future : (actors with futures)
  - Executor is a server with one or more worker tasks (worker pool).
  - Future is closure with work for executor (Callable) and place for result.
  - Call to executor submit is asynchronous and returns a future.

- Result is retrieved using get routine, which may block until result inserted by executor.

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
public class Matrix {
    public static void main( String[] args )
        throws InterruptedException, ExecutionException {
        class Adder implements Callable<Integer> {
            int row[], cols; // communication
            public Integer call() {
                int subtotal = 0;
                for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
                return subtotal;
            }
            Adder( int [] r, int c ) { row = r; cols = c; }
        }

        int rows = 10, cols = 10;
        int matrix[][] = new int[rows][cols], total = 0;
        // read matrix
        ExecutorService executor = Executors.newFixedThreadPool( 4 );
        List<Future<Integer>> subtotals = new ArrayList<Future<Integer>>();
        for ( int r = 0; r < rows; r += 1 ) // send off work for executor
            subtotals.add( executor.submit( new Adder( matrix[r], cols ) ) );
        for ( int r = 0; r < rows; r += 1 ) // wait for results
            total += subtotals.get( r ).get(); // retrieve result
        System.out.println( total );
        executor.shutdown();
    }
}

```

- $\mu$ C++ also has fixed thread-pool executor (used with actors).

```

struct Adder { // routine, functor or lambda
    int * row, cols; // communication
    int operator()() { // functor operator
        int subtotal = 0;
        for ( int c = 0; c < cols; c += 1 ) subtotal += row[c];
        return subtotal;
    }
    Adder( int row[ ], int cols ) : row( row ), cols( cols ) {}
};

```

```

int main() {
    const int rows = 10, cols = 10;
    int matrix[rows][cols], total = 0;
    // read matrix
    uExecutor executor( 4 );           // kernel threads
    Future_ISM<int> subtotals[rows];
    Adder * adders[rows];
    for ( int r = 0; r < rows; r += 1 ) { // send off work for executor
        adders[r] = new Adder( matrix[r], cols );
        subtotals[r] = executor.sendrecv( *adders[r] );
    }
    for ( int r = 0; r < rows; r += 1 ) { // wait for results
        total += subtotals[r]();
        delete adders[r];
    }
    cout << total << endl;
}

```

- Collections : `LinkedBlockingQueue`, `ArrayBlockingQueue`, `SynchronousQueue`, `PriorityBlockingQueue`, `DelayQueue`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`.

- Create threads that interact indirectly through atomic data structures, e.g., producer/-consumer interact via `LinkedBlockingQueue`.

- Atomic Types using compare-and-set (see Section 11.1.1, p. 201) (i.e., lock-free).

`AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray`, `AtomicReference<V>`, `AtomicReferenceArray<E>`

<code>int v;</code>		1
<code>AtomicInteger i = new AtomicInteger();</code>		2 2
<code>i.set( 1 );</code>		1 1
<code>System.out.println( i.get() );</code>		2 1
<code>v = i.addAndGet( 1 ); // i += delta</code>		1 2
<code>System.out.println( i.get() + " " + v );</code>		
<code>v = i.decrementAndGet(); // -i</code>		
<code>System.out.println( i.get() + " " + v );</code>		
<code>v = i.getAndAdd( 1 ); // i += delta</code>		
<code>System.out.println( i.get() + " " + v );</code>		
<code>v = i.getAndDecrement(); // i-</code>		
<code>System.out.println( i.get() + " " + v );</code>		

### 11.6.2 Pthreads

- C libraries built around routine abstraction, mutex/condition locks (“attribute” parameters not shown), and kernel threads.

```

int pthread_create( pthread_t * new_thread_ID,
                    void * (* start_func)(void *), void * arg );
int pthread_join( pthread_t target_thread, void ** status );
pthread_t pthread_self( void );
int pthread_yield(void);

int pthread_mutex_init( pthread_mutex_t * mp ); // constructor
int pthread_mutex_lock( pthread_mutex_t * mp );
int pthread_mutex_unlock( pthread_mutex_t * mp );
int pthread_mutex_destroy( pthread_mutex_t * mp ); // destructor

int pthread_cond_init( pthread_cond_t * cp ); // constructor
int pthread_cond_wait( pthread_cond_t * cp, pthread_mutex_t * mutex );
int pthread_cond_signal( pthread_cond_t * cp );
int pthread_cond_broadcast( pthread_cond_t * cp );
int pthread_cond_destroy( pthread_cond_t * cp ); // destructor

```

- Thread starts in routine `start_func` via `pthread_create` with single **void** \* parameter.
- Termination synchronization is performed by `pthread_join` with single **void** \* result.

```

void * rtn( void * arg ) { ... }
int i = 3, r, rc;
pthread_t t; // thread id
rc = pthread_create( &t, rtn, (void *)i ); // create and initialized task
if ( rc != 0 ) ... // check for error
// concurrency
rc = pthread_join( t, &r ); // wait for thread termination and result
if ( rc != 0 ) ... // check for error

```

- All C library approaches have type-unsafe communication.
- No external scheduling  $\Rightarrow$  complex direct-communication emulation.
- Internal scheduling is no-priority nonblocking  $\Rightarrow$  barging  $\Rightarrow$  wait statements must be in while loops to recheck conditions

```

typedef struct { // simulate monitor
    // buffer declarations
    pthread_mutex_t mutex; // mutual exclusion
    pthread_cond_t full, empty; // synchronization
} buffer;

// write your own constructor/destructor
void ctor( buffer * buf ) { // constructor
    ...
    pthread_mutex_init( &buf->mutex );
    pthread_cond_init( &buf->full );
    pthread_cond_init( &buf->empty );
}

```



```

void dtor( buffer * buf ) { // destructor
    pthread_mutex_lock( &buf->mutex ); // must be mutex
    ...
    pthread_cond_destroy( &buf->empty );
    pthread_cond_destroy( &buf->full );
    pthread_mutex_destroy( &buf->mutex );
}

void insert( buffer * buf, int elem ) {
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == Size )
        pthread_cond_wait( &buf->empty, &buf->mutex );
    // add to buffer
    buf->count += 1;
    pthread_cond_signal( &buf->full );
    pthread_mutex_unlock( &buf->mutex );
}

int remove( buffer * buf ) {
    pthread_mutex_lock( &buf->mutex );
    while ( buf->count == 0 )
        pthread_cond_wait( &buf->full, &buf->mutex );
    // remove from buffer
    buf->count -= 1;
    pthread_cond_signal( &buf->empty );
    pthread_mutex_unlock( &buf->mutex );
    return elem;
}

```

- Since there are no constructors/destructors in C, explicit calls are necessary to ctor/dtor before/after use.
- All locks must be initialized and finalized.
- Mutual exclusion must be explicitly defined where needed.
- Condition locks should only be accessed with mutual exclusion.
- `pthread_cond_wait` atomically blocks thread and releases mutex lock, which is necessary to close race condition on baton passing.

## 11.7 OpenMP

- Shared memory, implicit thread management (programmer hints), kernel threads, some explicit locking.
- Communicate with compiler with `#pragma` directives.

```
#pragma omp ...
```

- fork/join model

- fork: initial thread creates a team of parallel threads (including itself)
- each thread executes the statements in the region construct
- join: when team threads complete, synchronize and terminate, except initial thread which continues
- compile: `gcc -std=c99 -fopenmp openmp.c -lgomp`
- COBEGIN/COEND: each thread executes different section:

```

#include <omp.h>
... // declarations of p1, p2, p3
int main() {
    int i;
    #pragma omp parallel sections num_threads( 4 ) // fork "4" threads
    { // COBEGIN
        #pragma omp section
        { i = 1; } // BEGIN ... END
        #pragma omp section
        { p1( 5 ); }
        #pragma omp section
        { p2( 7 ); }
        #pragma omp section
        { p3( 9 ); }
    } // COEND (synchronize)
}

```

- **for** directive specifies each loop iteration is executed by a team of threads (COFOR)

```

int main() {
    const unsigned int rows = 10, cols = 10; // sequential
    int matrix[rows][cols], subtotals[rows], total = 0;
    // read matrix
    #pragma omp parallel for // fork "rows" threads
    for ( unsigned int r = 0; r < rows; r += 1 ) { // concurrent
        subtotals[r] = 0;
        for ( unsigned int c = 0; c < cols; c += 1 )
            subtotals[r] += matrix[r][c];
    }
    for ( unsigned int r = 0; r < rows; r += 1 ) // sequential
        total += subtotals[r];
    printf( "total:%d\n", total );
} // main

```

- In this case, sequential code directly converted to concurrent via `#pragma`.
- Variables outside section are shared; variables inside are thread private.
- Programmer responsible for sharing in vector/matrix manipulation.

- barrier

```
int main() {  
    #pragma omp parallel num_threads( 4 ) // fork "4" threads  
    {  
        sleep( omp_get_thread_num() );  
        printf( "%d\n", omp_get_thread_num() );  
        #pragma omp barrier // wait for all block threads to arrive  
        printf( "sync\n" );  
    }  
}
```

- Without omp section, all threads run same block (like omp parallel **for**).
- Barrier's trigger is the number of block threads.
- Threads sleeps for different times, but all print "sync" at same time.
- Also critical section and atomic directives.



# Index

- \_Accept**, 152, 171, 173–175
  - destructor, 176
- \_At**, 36, 82
- \_Coroutine**, 29
- \_Disable**, 41
- \_Enable**, 40, 41
- \_Exception**, 35
- \_Monitor**, 150
- \_Mutex**, 150
- \_Nomutex**, 152
- \_Resume**, 36
- \_Select**, 184
- \_Task**, 78
- \_Throw**, 36, 40, 82
- \_When**, 171, 173
- 1:1 threading, 70
- ABA problem, 203
- activation, 7
- active, 25
- actor, 75, 222
- Ada 95, 211
- adaptive spin-lock, 102
- administrator, 179
  - worker tasks, 179
- allocation
  - heap, 4
  - stack, 4
- allocation graphs, 144
- alternation, 87
- Amdahl’s law, 71
- arbiter, 96
- atomic, 84, 91, 151, 153, 225
- atomic, 198
- atomic consistent, 197
- atomic instruction
  - compare/assign, 201
  - fetch-and-increment, 99
  - swap, 98
  - test/set, 98
- automatic signal, 163
- bakery, 94
- banker’s algorithm, 143
- barge, 106
- barging, 86, 164, 224
- barging avoidance, 106, 166
- barging prevention, 106
- barrier, 114, 116, 166
- baton passing, 126
- binary semaphore, 118, 153
- BinSem, 120
  - acquire, 120
  - release, 120
- block activation, 11
- bottlenecks, 66
- bound catch clause, 20
- bound handlers, 39
- bounded buffer, 125, 152, 153, 164, 171, 173–175
- bounded overtaking, 91, 106
- break**, 4
  - labelled, 2
  - limitations, 4
- buffering, 124
  - bounded, 125
  - unbounded, 124
- busy wait, 84, 101, 103, 113
- busy waiting, 86
- C, 12
- C++11, 218
  - atomic, 198
- cache, 191
  - coherence, 193
  - consistency, 193

- eviction, 192
- flush, 192
- cache coherence, 193
- cache consistency, 193
- cache line, 191
- cache thrashing, 194
- call, 7
- call-back, 181
- catch, 14
- catch-any, 60
- channel, 216
- client side, 180
  - call-back, 181
  - future, 182
  - returning values, 180
  - ticket, 181
- COBEGIN, 73, 78, 122
- cocall, 48
- COEND, 73, 78
- coherence, 193
- collection, 161
- communication, 84
  - direct, 169
- compare-and-set(assign) instruction, 201
- concurrency, 65
  - difficulty, 65
  - increasing, 177
  - why, 65
- Concurrent C++, 213
- concurrent error
  - indefinite postponement, 139
  - live lock, 139
  - race condition, 139
  - starvation, 140
- concurrent exception, 14, 36, 82
- concurrent execution, 65
- concurrent hardware
  - structure, 66
- concurrent systems
  - explicit, 70
  - implicit, 70
  - structure, 70
- condition, 153, 161
- condition lock, 110, 113
- conditional critical region, 150
- consistency, 193
- context switch, 29, 66
- continue**
  - labelled, 2
- control dependency, 189
- cooperation, 103, 106, 107, 115
- coprocessor, 209
- coroutine, 25
  - main, 29
- coroutine main, 34, 54, 116
- counter, 122
- critical path, 72, 209
- critical region, 149
- critical section, 84, 101
  - hardware, 97
    - compare/assign, 201
    - fetch-and-increment, 99
    - swap, 98
    - test/set, 98
  - self testing, 86
- CUDA, 209
- data dependency, 189
- dating service, 155
- deadlock, 140, 141, 149
  - allocation graphs, 144
  - avoidance, 143
  - banker's algorithm, 143
  - detection/recovery, 146
  - mutual exclusion, 141
  - ordered resource, 142
  - prevention, 142
  - synchronization, 141
- declare intent, 88
- Dekker, 89, 198
- delivered, 14
- dependent, 103
- dependent execution, 73
- derived exception-types, 59
- destructor
  - \_Accept**, 176
- detach, 218
- detection/recovery, 146
- direct communication, 169
- disjoint, 195

- distributed system, 67
- divide-and-conquer, 79
- double-check locking, 196
- dynamic multi-level exit, 11, 18
- dynamic propagation, 18
- eliding, 189
- empty, 114, 122, 153
- entry queue, 154
- exception, 14
  - concurrent, 14, 36
  - handling, 13
  - handling mechanism, 13
  - hierarchy, 35
  - inherited members, 35
  - list, 61
  - nonlocal, 14, 36
  - parameters, 60
  - type, 14, 35
- exception handling, 13
- exception handling mechanism, 13
- exception list, 61
- exception parameters, 60
- exception type, 14
- exceptional event, 13
- execution, 14
- execution location, 25
- execution state, 25, 68
  - blocked, 68
  - halted, 68
  - new, 68
  - ready, 68
  - running, 68
- execution status, 25
- exit
  - dynamic multi-level, 11
  - static multi-exit, 2
  - static multi-level, 2
- explicit scheduling, 162
- explicit signal, 163
- external scheduling, 152, 170
- eye-candy, 1, 3
- failed cooperation, 159
- failure exception, 61
- false sharing, 194
- faulting execution, 14, 40
- fetch-and-increment instruction, 99
- Fibonacci, 26
- fix-up routine, 9
- flag, 130
- flag variable, 2, 159
- flickering, 193
- forward branch, 4
- freshness, 130
- fresh, 131
- front, 153
- full coroutine, 26, 49
- functor, 218
- future, 182, 220, 221
- Future\_ISM
  - available, 184
  - cancel, 184
  - cancelled, 184
  - delivery, 184
  - operator** T(), 184
  - operator**()(), 184
  - reset, 184
- garbage collection, 76
- Gene Amdahl, 71
- general-purpose GPU, 209
- generalize kernel threading, 70
- Go, 215
- goroutine, 215
- goto**, 2, 4, 11, 12
- GPGPU, 209
- graph reduction, 145
- greedy scheduling, 73
- guarded block, 15, 21, 59
- handled, 15
- handler, 14, 35
  - resumption, 22, 35, 37
  - termination, 35
- hazard pointers, 206
- heap, 4
- Heisenbug, 66
- Hesselink, 90
- immediate-return signal, 164

- implicit scheduling, 162
- implicit signal, 163
- inactive, 25
- increasing concurrency, 177
- indefinite postponement, 86, 139
- independent, 103
- independent execution, 73
- inherited members
  - exception type, 35
- internal scheduling, 153, 173
- interrupt, 67, 68, 207
  - timer, 67
- intrusive data-structures, 160
- intrusive list, 160
- invocation, 7
- isacquire, 109
- isomorphic graph, 145
- istream
  - isacquire, 109
- iterator, 43
  
- Java, 214
  - volatile**, 198
- Java monitor, 165
- jmp\_buf, 12
  
- kernel threading, 70
- kernel threads, 69
- keyword, additions
  - \_Accept**, 152
  - \_At**, 36
  - \_Coroutine**, 29
  - \_Disable**, 41
  - \_Enable**, 41
  - \_Exception**, 35
  - \_Monitor**, 150
  - \_Mutex**, 150
  - \_Nomutex**, 152
  - \_Resume**, 36
  - \_Select**, 184
  - \_Task**, 78
  - \_Throw**, 36
  - \_When**, 171
  
- label variable, 11
- lexical link, 38
  
- lifo scheduling, 73
- linear, 71
- linear speedup, 71
- livelock, 86
- liveness, 86
- local exception, 14
- lock, 87
  - taxonomy, 101
  - techniques, 126
- lock composition, 159
- lock free, 201
- lock programming
  - buffering, 124
    - bounded buffer, 125
    - unbounded buffer, 124
- lock-release pattern, 108
- longjmp, 12, 198
- loop
  - mid-test, 1
  - multi-exit, 1
  
- M:1 threading, 70
- M:N threading, 70
- main
  - coroutine, 29
  - task, 78, 169
- match, 14
- memory model, 197
- mid-test loop, 1
- modularization, 7
- monitor, 150
  - condition, 153, 161
  - external scheduling, 152
  - internal scheduling, 153
  - scheduling, 151
  - signal, 153, 161
  - simulation, 150
  - wait, 153, 161
- monitor type
  - no priority blocking, 164
  - no priority implicit signal, 164
  - no priority nonblocking, 164
  - priority blocking, 164
  - priority implicit signal, 164
  - priority nonblocking, 164



- monitor types, 162
- multi-exit
  - Multi-exit loop, 1
  - mid-test, 1
  - multi-level
    - dynamic, 18
- multi-level exit
  - dynamic, 11
  - static, 2
- multiple acquisition, 104
- multiple outcomes, 7
- multiprocessing, 66
- multiprocessor, 67
- multitasking, 66
- mutex lock, 104, 105, 120, 151
- mutex member, 150
- MutexLock, 105, 151
  - acquire, 105, 151
  - release, 105, 151
- mutual exclusion, 84, 125
  - alternation, 87
  - deadlock, 141
  - declare intent, 88
  - Dekker, 89
  - Dekker-Hesseling, 90
  - game, 85
  - lock, 87, 98
  - N-thread
    - arbiter, 96
    - bakery, 94
    - prioritized entry, 92
    - tournament, 95
  - Peterson, 91
  - prioritized retract intent, 89
  - retract intent, 88
- N:N threading, 70
- nano threads, 70
- nested monitor problem, 159
- no priority blocking, 164
- no priority implicit signal, 164
- no priority nonblocking, 164
- non-linear, 71
  - speedup, 71
- non-preemptive, 67
  - scheduling, 67
- nonlocal exception, 14, 36, 40
- nonlocal transfer, 7, 11, 16
- object
  - threading, 78
- object binding, 39
- OpenMP, 225
- operating system, 69, 70
- optimization, 189
- ordered resource, 142, 147
- ostream
  - osacquire, 109
- owner, 108
- owner lock, 104, 108
- ownership, 4, 201
- P, 118, 122, 141, 142, 149, 161
- parallel execution, 65
- park, 105
- partial store order, 197
- passeren, 118
- Peterson, 91
- precedence graph, 123
- preemptive, 67
  - scheduling, 67
- prioritized entry, 92
- prioritized retract intent, 89
- priority blocking, 164
- priority implicit signal, 164
- priority nonblocking, 164
- private semaphore, 135
- process, 65
- processor
  - multi, 67
  - uni, 66
- program order, 189
- prolagen, 118
- propagation, 14, 35
  - dynamic, 18
  - static, 17
- propagation mechanism, 14
- pthreads, 224
- race condition, 139
- race free, 197

- raise, 14, 35, 36
  - resuming, 35, 36
  - throwing, 35, 36
- readers/writer, 155, 159
  - freshness, 130
  - monitor
    - solution 3, 155
    - solution 4, 157
    - solution 8, 158
  - semaphore, 127
    - solution 1, 128
    - solution 2, 129
    - solution 3, 130
    - solution 4, 130
    - solution 5, 132
    - solution 6, 134
    - solution 7, 136
  - staleness, 130
- real time, 71
- release consistency, 197
- RendezvousFailure
  - failed cooperation, 159
- reordering, 189
- replication, 189
- reraise, 14
- resume, 36
- reservation, 207
- resume, 29, 40, 49
- resumption, 15, 22
- resumption handler, 22, 37
- rethrow, 36
- retract intent, 88
- retry, 21
- return code, 9
- return union, 9, 19
- routine
  - activation, 7
- routine abstraction, 223
- rw-safe, 90
- safety, 85
- scheduling, 67, 151, 170
  - explicit, 162
  - external, 152, 170
  - implicit, 162
  - internal, 153, 173
- scrambling, 193
- select blocked, 185
- select statement, 184
- self testing, 86
- semaphore, 141, 142
  - binary, 118, 153
  - counting, 120, 161
  - integer, 120
  - P, 118, 141, 142, 149, 161
  - private, 135
  - split binary, 126
  - V, 118, 149, 161
- semi-coroutine, 25, 49
- sequel, 17
- sequence, 161
- sequence points, 198
- sequential consistency, 197
- server side
  - administrator, 179
  - buffer, 178
- setjmp, 12
- shadow queue, 131, 157, 158
- shared-memory, 69
- signal, 161
  - automatic, 163
  - explicit, 163
  - immediate-return, 164
  - implicit, 163
- signal, 153, 174, 175
- signalBlock, 154
- single acquisition, 104
- software engineering, 7
- software pattern, 7
- software transactional memory, 208
- source execution, 14, 40
- speedup, 71
  - linear, 71
  - non-linear, 71
  - sub-linear, 71
  - super linear, 71
- spin lock, 101
  - implementation, 102
- split binary semaphore, 126
- spurious wakeup, 167

- SR, 213
- stack allocation, 4
- stack unwinding, 12, 15
- staleness, 130
- stale, 131
- START, 74, 79, 122
- starter, 48
- starvation, 73, 86, 129, 140
- state transition, 68
- static exit
  - multi-exit, 2
  - multi-level, 2
- static multi-level exit, 2
- static propagation, 17
- static variable, 85
- status flag, 9
- stream lock, 109
- strong memory ordering, 194
- sub-linear, 71
  - speedup, 71
- super linear, 71
- super-linear speedup, 71
- suspend, 29
- swap instruction, 98
- synchronization, 125, 151
  - communication, 84
  - deadlock, 141
  - during execution, 83
  - termination, 79
- synchronization lock, 110
- SyncLock, 153
- task, 65
  - exceptions, 81
  - external scheduling, 170
  - internal scheduling, 173
  - main, 78, 169
  - scheduling, 170
  - static variable, 85
- temporal order, 131
- terminate, 19
- terminated, 25
- termination, 15
- termination synchronization, 79, 115
- test-and-set instruction, 98
- thread, 65
  - communication, 73
  - creation, 73
  - synchronization, 73
- thread graph, 73
- thread object, 78
- threading model, 69
- throw, 14
- ticket, 135, 181
- time-slice, 101, 134, 135, 203, 205
- timer interrupt, 67
- times, 108
- total store order, 197
- tournament, 95
- transaction, 208
- tryacquire, 102
- TSO, 198
- uActor
  - Delete, 77
  - Destroy, 77
  - Finished, 77
  - Message, 77
  - Nodelete, 77
  - SenderMsg, 77
  - StartMsg, 77
  - StopMsg, 77
- uArray, 5
- uArrayPtr, 6
- uBarrier, 116
  - block, 116
  - last, 116
  - reset, 116
  - total, 116
  - waiters, 116
- uBaseException
  - defaultResume, 35
  - defaultTerminate, 35
  - message, 35
  - source, 35
  - sourceName, 35
- $\mu$ C++, 70
- $\mu$ C++, 20, 79, 180, 182
- uCondition, 153, 174, 175
  - empty, 153

- front, 153
- signal, 153
- signalBlock, 154
- wait, 153
- uCondLock, 113
  - broadcast, 113
  - empty, 113
  - signal, 113
  - wait, 113
- uLock, 102
  - acquire, 102
  - release, 102
  - tryacquire, 102
- unbounded buffer, 124
- unbounded overtaking, 91, 106
- unfairness, 86
- unguarded block, 15
- uniprocessor, 66
- unique\_ptr, 5
- unpark, 105
- uOwnerLock, 108
  - acquire, 108
  - release, 108
  - times, 108
  - tryacquire, 108
- uSemaphore, 122, 141, 142
  - counter, 122
  - empty, 122
  - P, 122, 141, 142
  - TryP, 122
  - V, 122
- user threading, 70
- user time, 71
- uSpinLock, 102
  - acquire, 102
  - release, 102
  - tryacquire, 102
- V, 118, 122, 149, 161
- virtual machine, 70
- virtual processors, 69
- volatile**, 198
- WAIT, 74, 79
- wait, 161
- wait, 153, 174, 175
- wait free, 201
- weak memory ordering, 194
- weak order, 197
- worker task, 178
- worker tasks, 179
  - complex, 179
  - courier, 179
  - notifier, 179
  - simple, 179
  - timer, 179
- yield, 101