CS350: Operating Systems
Lecture 4: Concurrency

Ali Mashtizadeh

University of Waterloo

Review: Processes and Threads

- A process is an instance of a running program
    - Process can have one or more threads

- A thread is an execution context
    - Share address space (code, data, heap), open files
    - Have their own CPU registers and stack (local variables)

- POSIX Thread APIs
    - `pthread_create()` – Create a new thread
    - `pthread_exit()` – Destroy the current thread
    - `pthread_join()` – Waits for a thread to exit

Outline

Critical Sections

```
int total = 0;
void add() {
   for (int i=0; i<N; i++) {
      total++;
   }
}

void sub() {
   for (int i=0; i<N; i++) {
      total--;
   }
}
```

Critical Sections

```
int total = 0;
void add() {
   /* r8 := &total */
   for (int i=0; i<N; i++) {
      lw r9, 0(r8) /* total++ */
      add r9, 1
      sw r9, 0(r8)
   }
}

void sub() {
   for (int i=0; i<N; i++) {
      lw r9, 0(r8) /* total-- */
      sub r9, 1
      sw r9, 0(r8)
   }
}
```

Critical Sectionsi: Schedule 1

```
Thread #1                          Thread #2
------------                       ------------
lw r9, 0(r8) /* total++ */
add r9, 1
sw r9, 0(r8)
                                   lw r9, 0(r8) /* total-- */
                                   sub r9, 1
                                   sw r9, 0(r8)
------------                       ------------
```

Critical Sectionsi: Schedule 1

```
Thread #1                            Thread #2

-------------                        -------------
lw r9, 0(r8) /* total++ */
add r9, 1
sw r9, 0(r8)
                                     lw r9, 0(r8) /* total-- */
                                     sub r9, 1
                                     sw r9, 0(r8)
-------------                        -------------
```

- Increment completed then decrement
- Result: total $= 0$

Critical Sections: Schedule 2

Thread #1

```
------------
lw r9, 0(r8) /* total++ */
add r9, 1
sw r9, 0(r8)

------------
```

Thread #2

```
------------

lw r9, 0(r8) /* total-- */
sub r9, 1
sw r9, 0(r8)
------------
```

Critical Sections: Schedule 2

Thread #1

```
------------
lw r9, 0(r8) /* total++ */
add r9, 1
sw r9, 0(r8)
------------
```

Thread #2

```
------------
lw r9, 0(r8) /* total-- */
sub r9, 1
sw r9, 0(r8)
------------
```

- Both load zero, then stores clobber one another
- Result: total = -1

Critical Sections: Schedule 3

Thread #1

```
-------------
lw r9, 0(r8) /* total++ */
add r9, 1

sw r9, 0(r8)


-------------
```

Thread #2

```
-------------
lw r9, 0(r8) /* total-- */
sub r9, 1
sw r9, 0(r8)



-------------
```

Critical Sections: Schedule 3

```
Thread #1                          Thread #2
------------                       ------------
lw r9, 0(r8) /* total++ */         lw r9, 0(r8) /* total-- */
add r9, 1                          sub r9, 1
                                   sw r9, 0(r8)
sw r9, 0(r8)


------------                       ------------
```

- Both load zero, then store clobbers the other
- Result: total = 1

Need for Synchronization

- Problem: Data races occur without synchronization

- Options:
  - Atomic Instructions: instantaneously modify a value
  - Locks: prevent concurrent execution

- ... it gets worse!

Program A

```
int flag1 = 0, flag2 = 0;

void p1(void *ignored) {
  flag1 = 1;
  if (!flag2) { critical_section_1(); }
}

void p2(void *ignored) {
  flag2 = 1;
  if (!flag1) { critical_section_2(); }
}

int main() {
  pthread_t tid;
  pthread_create(tid, NULL, p1, NULL);
  p2(); pthread_join(tid);
}
```

- Can both critical sections run?

Program B

```
int data = 0, ready = 0;

void p1(void *ignored) {
  data = 2000;
  ready = 1;
}

void p2(void *ignored) {
  while (!ready)
    ;
  use(data);
}
```

- Can `use` be called with value 0?

Program C

```
int a = 0, b = 0;

void p1(void *ignored) {
  a = 1;
}

void p2(void *ignored) {
  if (a == 1)
    b = 1;
}

void p3(void *ignored) {
  if (b == 1)
    use(a);
}
```

- Can `use()` be called with value 0?

Correct answers

- Program A: I don't know
- Program B: I don't know
- Program C: I don't know
- Why don't we know?
  - It depends on what machine you use
  - If a system provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency
- Note: Examples and other slide content from [Adve & Gharachorloo]

Outline

1. Critical Sections
2. CPU and Compiler Consistency
3. Peterson's Algorithm
4. Mutexes and Condition Variables
5. Semaphores
6. Data Races

Sequential Consistency

Sequential consistency

The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program. [Lamport]

- Boils down to two requirements:
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*
- Without SC, multiple CPUs can be “worse” than preemptive threads
  - May see results that cannot occur with any interleaving on 1 CPU
- Why doesn’t all hardware support sequential consistency?

SC thwarts hardware optimizations

- Complicates write buffers
  - E.g., read flag$[n]$ before flag$[2-n]$ written through in Program A
- Can't re-order overlapping write operations
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- Complicates non-blocking reads
  - E.g., speculatively prefetch `data` in Program B
- Makes cache coherence more expensive
  - Must delay write completion until invalidation/update (Program B)
  - Can't allow overlapping updates if no globally visible order (Program C)

SC thwarts compiler optimizations

- Code motion
- Caching value in register
    - Collapse multiple loads/stores of same address into one operation
- Common subexpression elimination
    - Could cause memory location to be read fewer times
- Loop blocking
    - Re-arrange loops for better cache performance
- Software pipelining
    - Move instructions across iterations of a loop to overlap instruction latency with branch cost

x86 consistency [Intel SDM 3A, §8.2]

- x86 supports multiple consistency/caching models
  - Memory Type Range Registers (MTRR) specify consistency for ranges of physical memory (e.g., frame buffer)
  - Page Attribute Table (PAT) allows control for each 4K page
- Choices include:
  - **WB**: Write-back caching (the default)
  - **WT**: Write-through caching (all writes go to memory)
  - **UC**: Uncacheable (for device memory)
  - **WC**: Write-combining – weak consistency & no caching
    (used for frame buffers, when sending a lot of data to GPU)
- Some instructions have weaker consistency
  - String instructions (written cache-lines can be re-ordered)
  - Special "non-temporal" store instructions (`movnt*`) that bypass cache and can be re-ordered with respect to other writes

x86 WB consistency

- Old x86s (e.g, 486, Pentium 1) had almost SC
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected?

x86 WB consistency

- Old x86s (e.g, 486, Pentium 1) had almost SC
  - Exception: A read could finish before an earlier write to a different location
  - Which of Programs A, B, C might be affected? *Just A*
- Newer x86s also let a CPU read its own writes early (store-to-load forwarding)

```
volatile int flag1 = 0, flag2 = 0;
```

```
int p1 (void)                    int p2 (void)
{                                {
  register int f, g;               register int f, g;
  flag1 = 1;                       flag2 = 1;
  f = flag1;                       f = flag2;
  g = flag2;                       g = flag1;
  return 2*f + g;                  return 2*f + g;
}                                }
```

  - E.g., *both* `p1` and `p2` can return 2:
  - Older CPUs would wait at "`f = ...`" until store complete

x86 atomicity

- `lock` – prefix makes a memory instruction atomic
  - Usually locks bus for duration of instruction (expensive!)
  - Can avoid locking if memory already exclusively cached
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered w. locked ones
- `xchg` – Exchange instruction is always locked (without the prefix)
- `cmpxchg` – Compare and exchange is also locked (without the prefix)
- Special fence instructions can prevent re-ordering
  - `lfence` – can't be reordered w. reads (or later writes)
  - `sfence` – can't be reordered w. writes
    (e.g., use after non-temporal stores, before setting a *ready* flag)
  - `mfence` – can't be reordered w. reads or writes

Assuming sequential consistency

- Often we reason about concurrent code assuming S.C.
- But for low-level code, know your memory model!
  - May need to sprinkle barriers instructions into your source
- For most code, avoid depending on memory model
  - Idea: If you obey certain rules (discussed later) ...system behavior should be indistinguishable from S.C.
- Let's for now say we have sequential consistency
- Example concurrent code: Producer/Consumer
  - `buffer` stores `BUFFER_SIZE` items
  - `count` is number of used slots
  - `out` is next empty buffer slot to fill (if any)
  - `in` is oldest filled slot to consume (if any)

```
void producer (void *ignored) {
   for (;;) {
      item *nextProduced = produce_item ();
      while (count == BUFFER_SIZE)
         /* do nothing */;
      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
   }
}

void consumer (void *ignored) {
   for (;;) {
      while (count == 0)
         /* do nothing */;
      item *nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;
      consume_item (nextConsumed);
   }
}
```

- What can go wrong in above threads (even w. S.C.)?

Data races

- `count` may have wrong value
- Possible implementation of `count++` and `count--`

```
register←count                register←count
register←register + 1         register←register − 1
count←register                count←register
```

- Possible execution (count one less than correct):

```
register←count
register←register + 1
register←count
register←register − 1
count←register
count←register
```

Data races (continued)

- What about a single-instruction add?
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe?

Data races (continued)

- What about a single-instruction add?
  - E.g., i386 allows single instruction `addl $1,_count`
  - So implement `count++/--` with one instruction
  - Now are we safe?
- Not atomic on multiprocessor!
  - Will experience exact same race condition
  - Can potentially make atomic with `lock` prefix
  - But `lock` very expensive
  - Compiler won't generate it, assumes you don't want penalty
- Need solution to *critical section* problem
  - Place `count++` and `count--` in critical section
  - Protect critical sections from concurrent execution

Desired properties of solution

- *Mutual Exclusion*
  - Only one thread can be in critical section at a time
- *Progress*
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in
- *Bounded waiting*
  - Once a thread T starts trying to enter the critical section, there is a bound on the number of times other threads get in
- Note progress vs. bounded waiting
  - If no thread can enter C.S., don't have progress
  - If thread A waiting to enter C.S. while B repeatedly leaves and re-enters C.S. *ad infinitum*, don't have bounded waiting

Outline

1. Critical Sections
2. CPU and Compiler Consistency
3. Peterson's Algorithm
4. Mutexes and Condition Variables
5. Semaphores
6. Data Races

Peterson's solution

- Still assuming sequential consistency
- Assume two threads, T_0 and T_1
- Variables
  - `int not_turn;` // not this thread's turn to enter C.S.
  - `bool wants[2];` // `wants[i]` indicates if T_i wants to enter C.S.
- Code:

```
for (;;) { /* assume i is thread number (0 or 1) */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

Does Peterson's solution work?

```
for (;;) { /* code in thread i */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn */;
  Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

- Mutual exclusion – can't both be in C.S.
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from C.S.
- Progress – If T_{1-i} not in C.S., can't block T_i
  - Means `wants[1-i] == false`, so T_i won't loop
- Bounded waiting – similar argument to progress
  - If T_i wants lock and T_{1-i} tries to re-enter, T_{1-i} will set `not_turn = 1 - i`, allowing T_i in

Outline

1. Critical Sections
2. CPU and Compiler Consistency
3. Peterson's Algorithm
4. Mutexes and Condition Variables
5. Semaphores
6. Data Races

Mutexes

- Peterson expensive, only works for 2 processes
  - Can generalize to n, but for some fixed n
- Must adapt to machine memory model if not S.C.
  - Ideally want your code to run everywhere
- Want to insulate programmer from implementing synchronization primitives
- Thread packages typically provide *mutexes*:

```
void mutex_init (mutex_t *m, \ldots );
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

  - Only one thread acuires `m` at a time, others wait

Thread API contract

- All global data should be protected by a mutex!
  - Global = accessed by more than one thread, at least one write
  - Exception is initialization, before exposed to other threads
  - This is the responsibility of the application writer

Compiler/Runtime Contract (C, Java, Go, etc.)

Assuming no data races the program behaves sequentially consistent.

- If you use mutexes properly, behavior should be indistinguishable from Sequential Consistency
  - Responsibility of the threads package & compiler
  - Mutex is broken if you use properly and don't see S.C.
- OS kernels also need synchronization
  - Some mechanisms look like mutexes
  - But interrupts complicate things (incompatible w. mutexes)

PThread Mutex API

- Function names in this lecture all based on *pthreads*
- `int pthread_mutex_init(pthread_mutex_t *m,`
  `pthread_mutexattr_t attr)`
  - Initialize a mutex
- `int pthread_mutex_destroy(pthread_mutex_t *m)`
  - Destroy a mutex
- `int pthread_mutex_lock(pthread_mutex_t *m)`
  - Acquire a mutex
- `int pthread_mutex_unlock(pthread_mutex_t *m)`
  - Release a mutex
- `int pthread_mutex_trylock(pthread_mutex_t *m)`
  - Attempt to acquire a mutex
  - Return 0 if successful, otherwise -1 (`errno == EBUSY`)

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;

void producer (void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();

        mutex_lock (&mutex);
        while (count == BUFFER_SIZE) {
          mutex_unlock (&mutex); /* <--- Why? */
          thread_yield ();
          mutex_lock (&mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        mutex_unlock (&mutex);
    }
}
```

Improved consumer

```
void consumer (void *ignored) {
    for (;;) {
        mutex_lock (&mutex);
        while (count == 0) {
          mutex_unlock (&mutex);
          thread_yield ();
          mutex_lock (&mutex);
        }

        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        mutex_unlock (&mutex);

        consume_item (nextConsumed);
    }
}
```

Condition variables

- Busy-waiting in application is a bad idea
  - Thread consumes CPU even when can't make progress
  - Unnecessarily slows other threads and processes
- Better to inform scheduler of which threads can run
- Typically done with *condition variables*
- `int pthread_cond_init(pthread_cond_t *, \ldots );`
  - Initialize with specific attributes
- `int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
  - Atomically unlock `m` and sleep until `c` signaled
  - Then re-acquire `m` and resume executing
- `int pthread_cond_signal(pthread_cond_t *c);`
  `int pthread_cond_broadcast(pthread_cond_t *c);`
  - Wake one/all threads waiting on `c`

Improved producer

```
mutex_t mutex = MUTEX_INITIALIZER;
cond_t nonempty = COND_INITIALIZER;
cond_t nonfull = COND_INITIALIZER;

void producer (void *ignored) {
   for (;;) {
      item *nextProduced = produce_item ();

      mutex_lock(&mutex);
      while (count == BUFFER_SIZE)
        cond_wait(&nonfull, &mutex);

      buffer [in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      count++;
      cond_signal(&nonempty);
      mutex_unlock(&mutex);
   }
}
```

Improved consumer

```
void consumer (void *ignored) {
   for (;;) {
      mutex_lock (&mutex);
      while (count == 0)
       cond_wait (&nonempty, &mutex);

      item *nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      count--;
      cond_signal (&nonfull);
      mutex_unlock (&mutex);

      consume_item (nextConsumed);
   }
}
```

Re-check conditions

- Always re-check condition on wake-up

```
while (count == 0) /* not if */
  cond_wait (&nonempty, &mutex);
```

- Else, breaks w. spurious wakeup or two consumers
  - Start with empty buffer, then:

C_1	C_2	P
`cond_wait (…);`		`mutex_lock (…);`
		⋮
		`count++;`
		`cond_signal (…);`
	`mutex_lock (…);`	`mutex_unlock (…);`
	`if (count == 0)`	
	⋮	
	use `buffer[out] ...`	
	`count--;`	
	`mutex_unlock (…);`	
use `buffer[out] ...`	⟵ No items in buffer	

Condition variables (continued)

- Why must `cond_wait` both release mutex & sleep?
- Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

Condition variables (continued)

- Why must `cond_wait` both release mutex & sleep?
- Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
  mutex_unlock (&mutex);
  cond_wait (&nonfull);
  mutex_lock (&mutex);
}
```

- Can end up stuck waiting when bad interleaving

```
PRODUCER                        CONSUMER

while (count == BUFFER_SIZE)
  mutex_unlock (&mutex);
                                mutex_lock (&mutex);
                                ...
                                count--;
                                cond_signal (&nonfull);
  cond_wait (&nonfull);
```

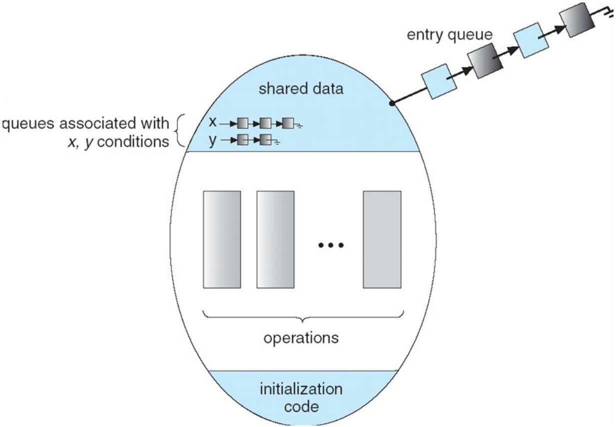
Monitors [Hoar]

- Programming language construct (e.g. Java, C#)
  - Possibly less error prone than raw mutexes, but less flexible too
  - A class where only one procedure executes at a time
  - Often provides CV like functionality

```
public class Statistics {
   private int counter;
   public synchronized int get() { return counter; }
   public synchronized void inc() { counter++; }
}
```

- Can implement mutex w. monitor or vice versa
  - But monitor alone doesn't give you condition variables
  - Need some other way to interact w. scheduler
  - Use *conditions*, which are essentially condition variables

Monitor implementation



- Queue of threads waiting to get in
- Java provides `obj.wait()`, `obj.notify()` and `obj.notifyAll()`

Outline

① Critical Sections

② CPU and Compiler Consistency

③ Peterson's Algorithm

④ Mutexes and Condition Variables

⑤ Semaphores

⑥ Data Races

Semaphores [Dijkstra]

- A *Semaphore* is initialized with an integer N
  - `int sem_init(sem_t *s, ..., unsigned int n);`
- Provides two functions:
  - `sem_wait(sem_t *s)` (originally called P)
  - `sem_post(sem_t *s)` (originally called V)
- Operation: `sem_wait` will return only N more times than `sem_post` called
  - Example: If $N == 1$, then semaphore is a mutex with `sem_wait` as lock and `sem_post` as unlock
- Semaphores give elegant solutions to some problems
- Linux primarily uses semaphores for sleeping locks
  - `sema_init`, `down_interruptible`, `up`, ...
  - Also reader-writer semaphores, `rw_semaphore` [Love]

Semaphore producer/consumer

- Initialize `full` to 0 (block consumer when buffer empty)
- Initialize `empty` to N (block producer when queue full)

```
void producer(void *ignored) {
    for (;;) {
        item *nextProduced = produce_item ();
        sem_wait(&empty);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&full);
    }
}
void consumer(void *ignored) {
    for (;;) {
        sem_wait(&full);
        item *nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        sem_post(&empty);
        consume_item(nextConsumed);
    }
}
```

Outline

1. Critical Sections
2. CPU and Compiler Consistency
3. Peterson's Algorithm
4. Mutexes and Condition Variables
5. Semaphores
6. Data Races

Benign races

- Sometimes “cheating” buys efficiency. . .
- Care more about speed than accuracy

```
hits++;  // each time someone accesses web site
```

- Know you can get away with race

```
if (!initialized) {
  lock (m);
  if (!initialized) { initialize (); initialized = 1; }
  unlock (m);
}
```

Detecting data races

- Static methods (hard)
- Debugging painful—race might occur rarely
- Instrumentation—modify program to trap memory accesses
- Lockset algorithm [eraser] particularly effective:
  - For each global memory location, keep a "lockset"
  - On each access, remove any locks not currently held
  - If lockset becomes empty, abort: No mutex protects data
  - Catches potential races even if they don't occur