Concurrency

What is concurrency?

> *In computer science, concurrency is a property of systems which consist of computations that execute overlapped in time, and which may permit the sharing of common resources between those overlapped computations.*
>
> (Wikipedia page on concurrency)

On multiprocessor machines, several threads can execute simultaneously, one on each processor (true parallelism).

On uniprocessor machines, only one thread executes at a given time. However, because of preemption and timesharing, threads appear to run simultaneously (fake parallelism).

⇒ Concurrency is an issue even on uniprocessor machines!

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Synchronization

Concurrent threads can interact with each other in a variety of ways:

- Threads share access to system devices (through OS).
- Threads in the same process share access to data (program variables) in their process' address space.

Common solution when multiple threads access the same data structures: Mutual exclusion.

> Mutual exclusion (MutEx): The shared resource is accessed by at most one thread at any given time.

The part(s) of the program in which the shared object is accessed is called “critical section”.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Critical Section – Example

Suppose we have a data structure (C++ class) `IntList` that can be used to manage a list of integers, by using a linked list.

```
int IntList::RemoveFront() {
  ListElement *element = this->first;
  assert(!IsEmpty());
  int num = this->first->item;
  if (this->first == this->last)
    this->first = this->last = NULL;
  else
    this->first = element->next;
  this->numInList--;
  delete element;
  return num;
}
```

`RemoveFront` is (part of) a critical section. It might not work properly if executed by more than one thread at the same time. – **Why?**

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Race Condition

Definition

Suppose multiple threads are running within the same process.

A *race condition* (or *race hazard*) is an instruction sequence whose outcome depends on the order in which it is executed by the threads.

Example:

```
     int *array[256]; int arrayLen = 0;
        void addToArray(int n) {
(1)       array[arrayLen] = n;
(2)       arrayLen++;
        }
```

Two threads (T1, T2) are executing `addToArray` concurrently.
Possible execution orders:

T1(1)-T1(2)-T2(1)-T2(2), T2(1)-T2(2)-T1(1)-T1(2),
T1(1)-T2(1)-T2(2)-T1(2), T1(1)-T2(2)-T1(2)-T2(2), ...

The `RemoveFront` method on the previous slide contains a race condition.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Critical Section – Example

Suppose we have a data structure (C++ class) `IntList` that can be used to manage a list of integers, by using a linked list.

```
void IntList::Append(int item) {
  ListElement *element = new ListElement(item);
  assert(!IsInList(item));
  if (IsEmpty())
    this->first = this->last = element;
  else {
    this->last->next = element;
    this->last = element;
  }
  numInList++;
}
```

`Append` is part of the same critical section as `RemoveFront`. It may not work properly if 2 threads execute it at the same time or if one thread executes `Append`, while another one executes `RemoveFront`.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion

Peterson's Mutual Exclusion Algorithm (Gary Peterson, 1981)

Two threads executing the same function:

```
bool flag[2] = {false, false};
int turn;

void doSomething(int threadID) {
  // threadID can be 0 or 1, indicating which thread
  int otherThread = 1 - threadID;
  flag[threadID] = true;
  turn = otherThread;
  while ((flag[otherThread]) && (turn == otherThread)) {
    // busy wait
  }

  // critical section...

  flag[threadID] = false;
}
```


Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion

Properties of Peterson's Mutual Exclusion Algorithm

Mutual exclusion – T0 and T1 can never be in the critical section at the same time.

Progress requirement – If T0 does not want to enter the critical section, T1 can enter without waiting (and vice versa).

Starvation-freeness – If T0 is in the critical section and T1 wants to enter, then T1 is guaranteed to enter the critical section before T0 enters is the next time (and vice versa).

Only works for 2 threads (but can be generalized to N threads).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion

Peterson's algorithm is a form of mutual exclusion referred to as *spin lock*.

If thread T0 is in the critical section, and T1 wants to enter, then T1 executes the busy loop (“keeps spinning”) until T0 leaves the critical section.

Spin locks are only advisable if:

- the computer has more than 1 CPU;
- the wait time is usually very short;
- no other (unblocked) threads are waiting for the CPU.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion Using Special Instructions

Peterson's algorithm assumes only two atomic operations: *load* and *store.*

```
flag[threadID] = true;
turn = otherThread;
while ((flag[otherThread]) && (turn == otherThread)) { }
// critical section...
flag[threadID] = false;
```

Simpler solutions are possible if more complex atomic operations are supported by the hardware: test-and-set (set the value of a variable and return the old value), swap (swap the values of two variables).

On uniprocessor machines, mutual exclusion can also be achieved by disabling interrupts (*why?*). But needs to be done by the kernel.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion: Test-and-Set

Suppose we have a function `testAndSet` that atomically (without the possibility of being interrupted)

- checks the value of the given variable
- changes the value of the variable to some new value
- returns the original value

```
bool lock = false;  // shared global variable

void doSomething() {
  while (testAndSet(&lock, true) == true) { }  // busy wait
  // critical section; do something important...
  lock = false;  // atomic store operation
}
```

This mutual exclusion algorithms works for an arbitrary number of threads, but starvation is possible.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion: Swap

How can we realize mutual exclusion if the hardware provides an atomic swap instruction (swapping the values of two variables atomically)?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Semaphores

Problems with mutual exclusion so far:

- Peterson's algorithm only works for 2 threads.
- Spin locks (busy waits) are wasteful of CPU resources.
- Atomic operations like test-and-set are not available on every hardware.

Solution:

- Provide a synchronization primitive as a kernel service.
- Give it a fancy name: Semaphore.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Semaphores

A semaphore is an object with an integer value (having some initial value). It supports two operations:

P – If the semaphore's value is greater than zero, decrement. Otherwise, wait until the value is greater than zero and then decrement.

V – Increment the value of the semaphore.

Semaphores were invented by Edsger Dijkstra. V stands for “verhoog” (increase). P stands for “probeer te verlagen” (try and decrease). Both P and V have to be atomic operations.

Two types of semaphores: counting semaphores and binary semaphores. A binary semaphore can only have value 0 or 1.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Mutual Exclusion Using a Binary Semaphore

```
BinarySemaphore s(1);  // semaphore, initial value: 1
void doSomething() {
  s.P();
  // critical section; do something important...
  s.V();
}
```

Very convenient! The implementation of the semaphore class (usually in the OS kernel) takes care of everything.

...but how does the kernel realize the atomic operations P and V?

– see previous slides!

Producer-Consumer Scenario

Suppose we have a process with N threads, sharing an array with some data. Out of the N threads, M are

producers, adding data to the array,

while the other N-M threads are

consumers, removing data from the array.

How can we make sure that a consumer thread only removes an item from the array when there is actually something in the array?

Do we also need to make sure that two consumer threads do not try to remove the same item?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Producer-Consumer Scenario

CountingSemaphore s(0); // initial value: 0
Item buffer[SOME_REALLY_LARGE_NUMBER]; // infinite buffer
int itemCount = 0;

```
// producer code:
void addItem(Item item) {
  buffer[itemCount++] = item;
  s.V();
}

// consumer code:
Item removeItem() {
  s.P();
  return buffer[--itemCount];
}
```

– Is this all? –

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Producer-Consumer Scenario

What if the shared buffer does not have infinite capacity? How do we make sure the producers do not try to put more stuff into the buffer than there is space?

```
Item buffer[N];  // a shared array with space for N items
CountingSemaphore full(0);
CountingSemaphore empty(N);

void addItem(Item item) {
  empty.P();
  buffer[itemCount++] = item;  // assume this is atomic
  full.V();
}

Item removeItem() {
  full.P();
  Item result = buffer[--itemCount];  // assume this is
  empty.V();                          // atomic
  return result;
}
```


Stefan Buettcher
<sbuettch@uwaterloo.ca>

Implementing Semaphores

```
void Semaphore::P() {
  // start critical section
  while (this->value == 0) {
    // end critical section – this is important!
    // start critical section
  }
  this->value = this->value - 1;
  // end critical section
}

void Semaphore::V() {
  // start critical section
  this->value = this->value + 1;
  // end critical section
}
```

⇒ In order to implement a semaphore, we need to realize mutual exclusion for the methods `P` and `V`.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Implementing Semaphores

Semaphores can be implemented at the user level (as part of a user-level thread library) or in the kernel.

User-level vs. kernel-level semaphores basically have the same advantages/disadvantages as user-level vs. kernel-level threads.

As an optimization, semaphores can interact with the thread scheduler (easy when inside the kernel):

- threads can be blocked (instead of busy wait) when calling P
- performing the V operation on a semaphore can make blocked threads ready

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Semaphore Class in Nachos

```
class Semaphore {
 public:
  Semaphore(char *debugName, int initialValue);
  ~Semaphore();
  char *getName() { return name; }
  void P();
  void V();
  void SelfTest();
 private:
  char *name;
  int value;
  List<Thread*> *queue;
}
```


Stefan Buettcher
<sbuettch@uwaterloo.ca>

Semaphore Class in Nachos

```
void Semaphore::P() {
  Interrupt *interrupt = kernel->interrupt;
  Thread *currentThread = kernel->currentThread;
  IntStatus oldLevel = interrupt->SetLevel(IntOff);
  if (value <= 0) {
    queue->Append(currentThread);
    currentThread->Sleep(FALSE);
  }
  else
    value--;
  interrupt->SetLevel(oldLevel);
}
```

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Semaphore Class in Nachos

```
void Semaphore::V() {
  Interrupt *interrupt = kernel->interrupt;
  IntStatus oldLevel = interrupt->SetLevel(IntOff);
  if (!queue->Empty())
    kernel->scheduler->ReadyToRun(queue->RemoveFront());
  else
    value++;
  interrupt->SetLevel(oldLevel);
}
```

Monitors

A *monitor* is a construct in a programming language that explicitly supports synchronized access to data.

A monitor is an object for which
- the object state is accessible only through the object's methods;
- at most one method may be active at the same time.

If two threads attempt to execute methods of the object at the same time, one will be blocked until the other one is finished.

Java example:

```
class C {
  public synchronized foo() {
    // do something
  }
}
```

Condition Variables

Inside a monitor, *condition variables* may be declared and used.

A condition variable is an object with support for two operations:

- **wait** – the calling thread blocks and releases the monitor;
- **signal** – if there are any blocked threads (**wait**ing), then unblock one of them; otherwise, do nothing.

A thread being signalled does not automatically give it back the monitor. It first has to wait till the thread that currently has the monitor releases it.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Bounded Buffer with Monitor

```
Item buffer[N];
int count;
Condition notfull, notempty;

void produce(Item item) {  // monitor method
  while (count == N) { wait(notfull); }
  buffer[count++] = item;
  signal(notempty);
}

Item consume() {  // monitor method
  while (count == 0) { wait(notempty); }
  Item result = buffer[--count];
  signal(notfull);
  return item;
}
```

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Simulating Monitors with Semaphores

- Use a single binary semaphore (e.g., the Nachos Lock) to realize mutual exclusion.
- Each method must start by acquiring the mutex semaphore and must release it an all return paths.
- Signal only while holding the mutex.
- Re-check the wait condition after each wait.
- Return only constants or local variables.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlocks

Suppose there are two processes running on the same machine. The machine has 64 MB of memory. The following events occur:

- Process A allocates 30 MB.
- Process B allocates 30 MB.
- Process A tries to allocate another 8 MB and gets blocked by the kernel because the 8 MB are currently not available.
- Process B tries to allocate another 5 MB and gets blocked by the kernel because the 5 MB are currently not available.

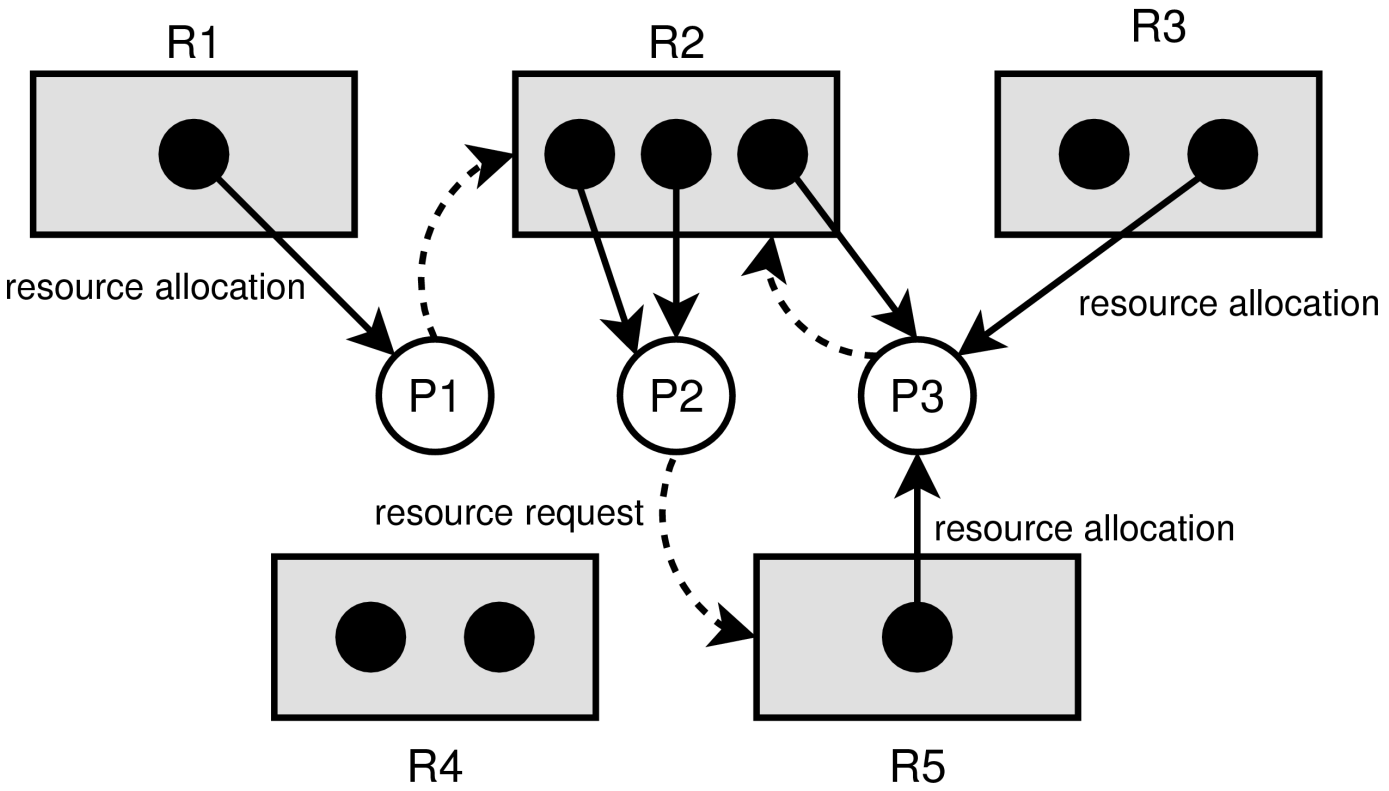
The two processes are deadlocked – neither process can make any progress because it is waiting for the other process to release some resources. The processes are permanently stuck.

Sidenote: On Linux, this would not lead to a deadlock. Linux' optimistic allocation strategy would lure one of the processes into a segmentation fault.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlocks

Resource allocation graph (example)

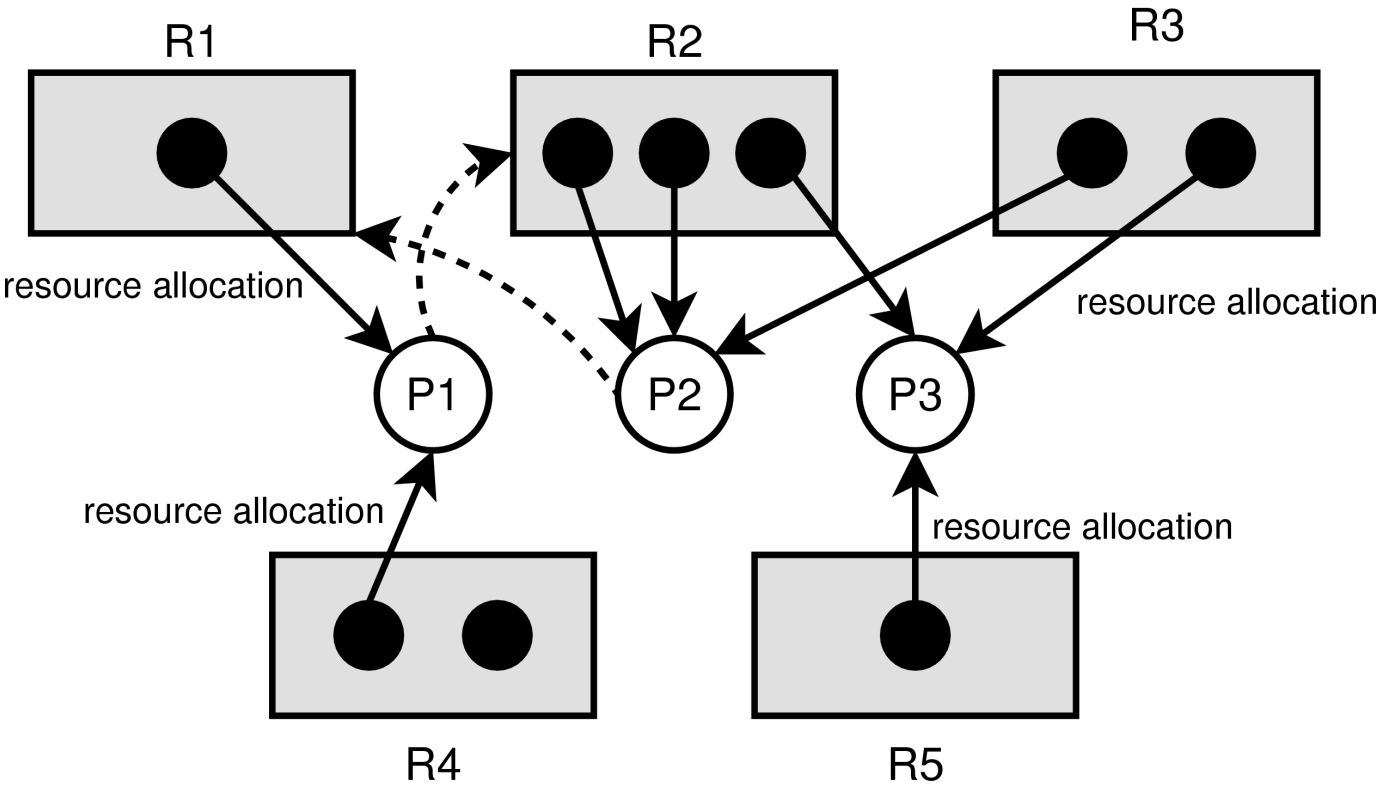


Is there a deadlock in this system?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlocks

Resource allocation graph (example)



Is there a deadlock in this system?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlock Detection

How to detect whether there is a deadlock in the system?

A thread *T* is *blocked indefinitely*
- if it is blocked and
- if all threads that *T* is waiting for are blocked indefinitely.

Equivalent statement

A thread T is *not blocked indefinitely*
- if it is not blocked or
- if it is waiting for a thread that is not blocked indefinitely.

⇒ Use this for an algorithm.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlock Detection (Algorithm)

Notation

- R_i: request vector for process P_i
- A_i: current allocation vector for process P_i
- U: unallocated resource vector (1 for each unallocated resource)
- T: scratch resource vector – resources that can become available
- f_i: flag indicating whether the algorithm is finished with P_i or not;
  if algorithm is finished with P_i, this means P_i is not indefinitely blocked

Algorithm

$T := U$ // initialize T

$f_i :=$ true if $A_i = 0$, false otherwise // if no allocation, then done with P_i

while $\exists\ i\ (\ \neg\ f_i \wedge (\ R_i \leq T\ )\ )$ do // find P_i that is not blocked indefinitely

  $T := T + A_i$ // because P_i is not b.i., all its resources can become available

  $f_i :=$ true // P_i is not blocked indefinitely, so we are done with it

if $\exists\ \ i\ (\ \neg\ f_i\ )$ then “Deadlock!”

  else “No deadlock.”

Deadlock Detection (Examples)

Example 1

$R_1 = (0,1,0,0,0)$
$R_2 = (0,0,0,0,1)$
$R_3 = (0,1,0,0,0)$

$A_1 = (1,0,0,0,0)$
$A_2 = (0,2,0,0,0)$
$A_3 = (0,1,1,0,1)$

$U = (0,0,1,1,0)$

Example 2

$R_1 = (0,1,0,0,0)$
$R_2 = (1,0,0,0,0)$
$R_3 = (0,0,0,0,0)$

$A_1 = (1,0,0,1,0)$
$A_2 = (0,2,1,0,0)$
$A_3 = (0,1,1,0,1)$

$U = (0,0,0,0,0)$

Example 3

$R_1 = (1,0,0,0,0)$
$R_2 = (0,0,1,0,0)$
$R_3 = (0,0,0,0,1)$

$A_1 = (0,1,0,0,2)$
$A_2 = (1,0,0,3,0)$
$A_3 = (0,0,2,0,0)$

$U = (0,1,0,1,0)$

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlock Prevention

Deadlocks are caused by cyclic dependencies. They can be avoided by imposing the following rules on any resource allocation:

No "hold and wait": A thread may never wait for a resource while it is holding another resource. It may hold several resources, but it must request them all in a single operation.

Preemption: In order to wait for a resource, a thread must first release all its resources and then re-acquire them.

Resource ordering: Each resource type is assigned a number. A thread may not request a resource of type $j \leq k$ if it currently holds a resource of type $m \geq k$.

Which rule is the least restrictive?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Deadlock Recovery

The system can maintain a resource allocation graph and use it to detect when deadlocks occur.

Deadlock recovery can be accomplished by terminating one or more threads involved in a deadlock.

This is usually not done in the kernel, because the kernel has no idea what it is doing – *Which thread should be terminated???*

Deadlock recovery is only implemented in critical applications (e.g., real-time systems) that need to keep working under all circumstances.

In most applications, a deadlock indicates a design flaw and should be dealt with by fixing the code and recompiling.

Stefan Buettcher
<sbuettch@uwaterloo.ca>