Concurrency

- On multiprocessors, several threads can execute simultaneously, one on each processor.
- On uniprocessors, only one thread executes at a time. However, because of preemption and timesharing, threads appear to run concurrently.

Concurrency and synchronization are important even on uniprocessors.

Thread Synchronization

- Concurrent threads can interact with each other in a variety of ways:
  - Threads share access (though the operating system) to system devices.
  - Threads in the same process share access to program variables in their process's address space.
- A common synchronization problem is to enforce *mutual exclusion*, which means making sure that only thread at a time uses a shared object, e.g., a variable or a device.
- The part of a program in which the shared object is accessed is called a *critical section*.

Critical Section Example (Part 1)

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

The `RemoveFront` method is a critical section. It may not work properly if two threads call it at the same time on the same `IntList`. (Why?)

Critical Section Example (Part 2)

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

The `Append` method is part of the same critical section as `RemoveFront`. It may not work properly if two threads call it at the same time, or if a thread calls it while another has called `RemoveFront`

Peterson's Mutual Exclusion Algorithm

```
boolean flag[2];  /* shared, initially false */
int turn;         /* shared */

flag[i] = true;   /* in one process, i = 0 and j = 1 */
turn = j;         /* in the other, i = 1 and j = 0 */
while (flag[j] && turn == j) { }  /* busy wait */

 critical section    /* e.g., call to RemoveFront */

flag[i] = false;
```

Ensures mutual exclusion and avoids starvation, but works only for two processes. (Why?)

Mutual Exclusion Using Special Instructions

- Software solutions to the critical section problem (e.g., Peterson's algorithm) assume only atomic load and atomic store.
- Simpler algorithms are possible if more complex *atomic* operations are supported by the hardware. For example:

  Test and Set: set the value of a variable, and return the old value

  Swap: swap the values of two variables
- On uniprocessors, mutual exclusion can also be achieved by disabling interrupts during the critical section. (Normally, user programs cannot do this, but the kernel can.)

Mutual Exclusion with Test and Set

```
boolean lock;  /* shared, initially false */

while (TestAndSet(&lock,true)) { }  /* busy wait */

 critical section /* e.g., call to RemoveFront */

lock = false;
```

Works for any number of threads, but starvation is a possibility.

Semaphores

- A semaphore is a synchronization primitive that can be used to solve the critical section problem, and many other synchronization problems too
- A semaphore is an object that has an integer value, and that support two operations:

  P: if the semaphore value is non-zero, decrement the value. Otherwise, wait until the value is non-zero and then decrement it.

  V: increment the value of the semaphore
- Two kinds of semaphores:

  counting semaphores: can take on any non-negative value

  binary semaphores: take on only the values 0 and 1. (`V` on a binary semaphore with value 1 has no effect.)

By definition, the `P` and `V` operations of a semaphore are *atomic*.

Mutual Exclusion Using a Binary Semaphore

```
binarySemaphore s;  /* initial value is 1 */

P(s);

 critical section /* e.g., call to RemoveFront */

V(s);
```


Producer/Consumer Using a Counting Semaphore

```
countingSemaphore s;  /* initial value is 0 */
item buffer[infinite];  /* huge buffer, initially empty */

Producer's Pseudo-code:
  add item to buffer
  V(s);

Consumer's Pseudo-code:
  P(s);
  remove item from buffer
```

If mutual exclusion is required for adding and removing items from the buffer, this can be provided using a second semaphore. (How?)

Producer/Consumer with a Bounded Buffer

```
countingSemaphore full;    /* initial value is 0 */
countingSemaphore empty;   /* initial value is N */
item buffer[N];            /* buffer with capacity N */

Producer's Pseudo-code:
  P(empty);
  add item to buffer
  V(full);

Consumer's Pseudo-code:
  P(full);
  remove item from buffer
  V(empty);
```


Implementing Semaphores

```
void P(s) {
   start critical section
   while (s == 0) {   /* busy wait */
       end critical section
       start critical section }
   s = s - 1;
   end critical section }

void V(s) {
   start critical section
   s = s + 1;
   end critical section }
```

Any mutual exclusion technique (e.g., Dekker, Lamport, test and set) can be used to protect the critical sections. However, starvation is possible with this implementation.

Nachos Semaphore Class

```
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();
    char* getName() { return name;}
    void P();
    void V();
    void SelfTest();
  private:
    char* name;    // useful for debugging
    int value;     // semaphore value, always >= 0
    List<Thread *> *queue;
   };
```


Nachos Semaphore P()

```
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--; }
    (void) interrupt->SetLevel(oldLevel);
}
```

Nachos Semaphore V()

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


Monitors

- a monitor is a programming language construct that supports synchronized access to data
- a monitor is essentially an object for which
  - object state is accessible only through the object's methods
  - only one method may be active at a time
- if two threads attempt to execute methods at the same time, one will be blocked until the other finishes
- inside of a monitor, so called *condition variables* can be declared and used

Condition Variable

- a condition variable is an object that support two operations:

  wait: causes the calling thread to block, and to release the monitor

  signal: if threads are blocked on the signaled condition variable then unblock one of them, otherwise do nothing
- a thread that has been unblocked by *signal* is outside of the monitor and it must wait to re-enter the monitor before proceeding.
- in particular, it must wait for the thread that signalled it

---

This describes Mesa-type monitors. There are other types on monitors, notably Hoare monitors, with different semantics for `wait` and `signal`.

Bounded Buffer Using a Monitor

```
item buffer[N]; /* buffer with capacity N */
int count; /* initially 0 */
condition notfull,notempty;

Produce(item) {
  while (count == N) { wait(notfull); }
  add item to buffer
  count = count + 1;
  signal(notempty);
}
```

`Produce` is implicitly executed atomically, because it is a monitor method.

Bounded Buffer Using a Monitor (cont'd)

```
Consume(item) {
  while (count == 0) { wait(notempty); }
  remove item from buffer
  count = count - 1;
  signal(notfull);
}
```

`Consume` is implicitly executed atomically, because it is a monitor method. Notice that `while`, rather than `if`, is used in both `Produce` and `Consume`. This is important. (Why?)

Simulating Monitors with Semaphores and Condition Variables

- Use a single binary semaphore (or Nachos "Lock") to provide mutual exclusion.
- Each method must start by acquiring the mutex semaphore, and must release it on all return paths.
- Signal only while holding the mutex semaphore.
- Re-check the wait condition after each wait.
- Return only (the values of) variables that are local to the method.

Producer Implemented with Locks and Condition Variables (Example)

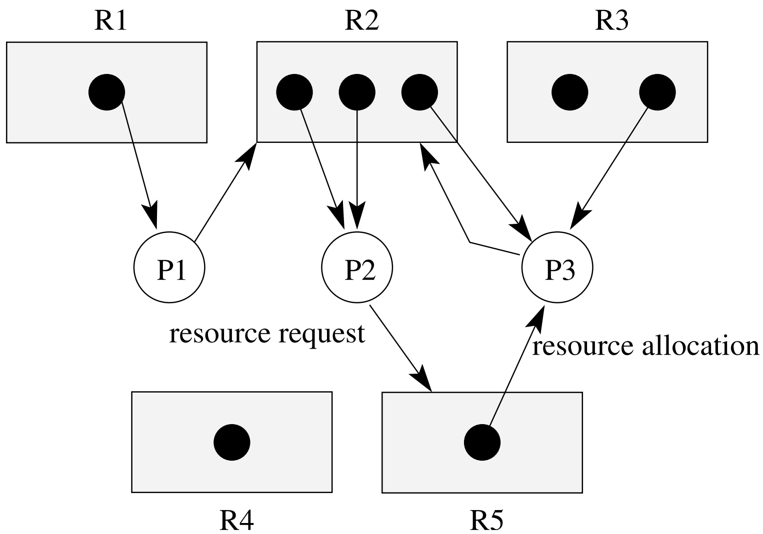
```
item buffer[N]; /* buffer with capacity N */
int count; /* initially 0 */
Condition notfull,notempty;
Lock mutex;     /* for mutual exclusion */
Produce(item) {
  mutex.Acquire();
  while (count == N) {
     notfull.Wait(mutex);  }
  add item to buffer
  count = count + 1;
  notempty.Signal(mutex);
  mutex.Release();
}
```


Deadlocks

- A simple example. Suppose a machine has 64MB of memory. The following sequence of events occurs.
  1. Process A starts, using 30MB of memory.
  2. Process B starts, also using 30MB of memory.
  3. Process A requests an additional 8MB of memory. The kernel blocks process A's thread, since there is only 4 MB of available memory.
  4. Process B requests an additional 5MB of memory. The kernel blocks process B's thread, since there is not enough memory available.

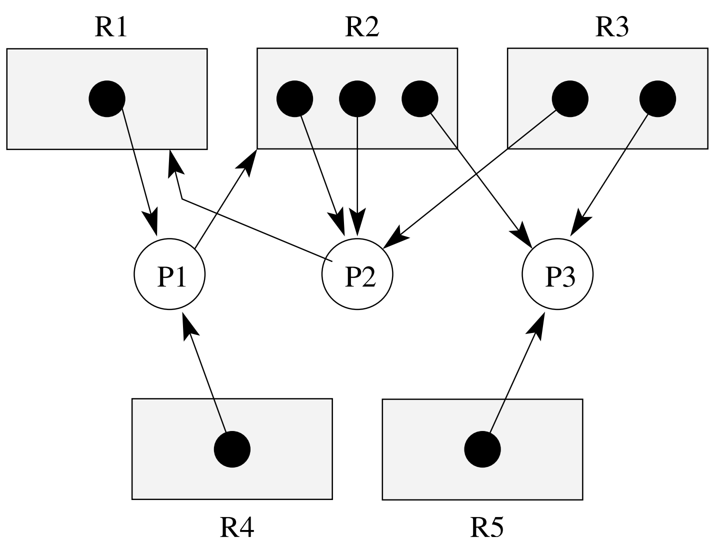
These two processes are *deadlocked* - neither process can make progress. Waiting will not resolve the deadlock. The processes are permanently stuck.

Resource Allocation Graph (Example)



Is there a deadlock in this system?

Resource Allocation Graph (Another Example)



Is there a deadlock in this system?

Deadlock Prevention

No Hold and Wait: prevent a process from requesting resources if it currently has resources allocated to it. A process may hold several resources, but to do so it must make a single request for all of them.

Preemption: to wait for a resource, a process must release and (after waiting) re-acquire any resources it currently holds.

Resource Ordering: Order (e.g., number) the resource types, and require that each process acquire resources in increasing resource type order. That is, a process may make no requests for resources of type less than or equal to i once the process has requested resources of type i.

Deadlock Detection and Correction

- main idea: the system maintains the resource allocation graph and tests it to determine whether there is a deadlock. If there is, the system must recover from the deadlock situation.
- deadlock recovery is usually accomplished by terminating one or more of the processes involved in the deadlock
- when to test for deadlocks? Can test on every blocked resource request, or can simply test periodically. Deadlocks persist, so periodic detection will not "miss" them.

Deadlock detection and deadlock correction are both costly. This approach makes sense only if deadlocks are expected to be infrequent.

Detecting Deadlock in a Resource Allocation Graph

- System State Notation:
  - R_i: request vector for process P_i
  - A_i: current allocation vector for process P_i
  - U: unallocated (available) resource vector
- Additional Algorithm Notation:
  - T: scratch resource vector
  - f_i: algorithm is finished with process P_i? (boolean)

Detecting Deadlock (cont'd)

```
/* initialization */
T = U
f_i is false if A_i > 0, else true
/* can each process finish? */
while ∃ i ( ¬ f_i ∧ R_i ≤ T ) {
      T = T + A_i;
      f_i = true
}
/* if not, there is a deadlock */
if ∃ i ( ¬ f_i ) then report deadlock
else report no deadlock
```

Deadlock Detection, Positive Example

- $R_1 = (0, 1, 0, 0, 0)$
- $R_2 = (0, 0, 0, 0, 1)$
- $R_3 = (0, 1, 0, 0, 0)$
- $A_1 = (1, 0, 0, 0, 0)$
- $A_2 = (0, 2, 0, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 1, 1, 0)$

The deadlock detection algorithm will terminate with $f_1 == f_2 == f_3 ==$ `false`, so this system is deadlocked.

Deadlock Detection, Negative Example

- $R_1 = (0, 1, 0, 0, 0)$
- $R_2 = (1, 0, 0, 0, 0)$
- $R_3 = (0, 0, 0, 0, 0)$
- $A_1 = (1, 0, 0, 1, 0)$
- $A_2 = (0, 2, 1, 0, 0)$
- $A_3 = (0, 1, 1, 0, 1)$
- $U = (0, 0, 0, 0, 0)$

This system is not in deadlock. It is possible that the processes will run to completion in the order P_3, P_1, P_2.