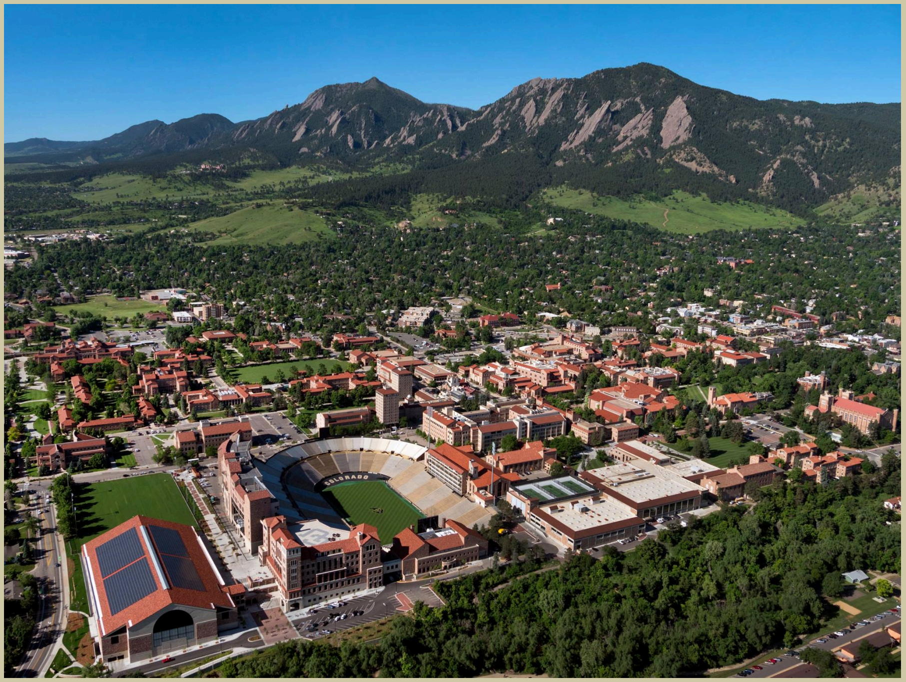
Design and Analysis of Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado Boulder

Monitors and Conditional Variables



Design and Analysis of
Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado
Boulder

University of Colorado
Boulder

Deadlock when using Semaphores

- It can easily occur
  - two tasks, each desires a resource locked by the other process
  - circular dependency
  - programming errors
    - switching order of P() and V()
    - calling wait multiple times
    - forgetting a signal
- Semaphores provide mutual exclusion, but can introduce deadlock

Monitors

- Semaphores can result in deadlock due to programming errors
  - forgot to add a P() or V(), or misordered them, or duplicated them
- To reduce these errors, introduce high-level synchronization primitives, e.g. *monitors with condition variables*
  - essentially automates insertion of P() and V() for you
  - As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#
  - underneath, the OS may implement monitors using semaphores and mutex locks

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
     // shared local variables

     function f1(...) {
     ...
     }
     ...
     function fN(...) {
     ...
     }
     init_code(...) {
     ...
     }
  }
```

- A monitor ensures that only 1 process/thread at a time can be active within a monitor
  – simplifies programming, no need to explicitly synchronize
- Implicitly, the monitor defines a mutex lock

  semaphore mutex = 1;

- Implicitly, the monitor also defines mutual exclusion around each function
  – Each function's critical code is effectively:

```
function fj(...) {
  P(mutex)
  // critical code
  V(mutex)
}
```

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

```
monitor monitor_name {
     // shared local variables
       int count;
       data_type data[MAX_COUNT];

     function add_item(...) {
     ...
     }

     ...
     function remove_item(...) {
     ...
     }

     init_code(...) {
     ...
     }
  }
```

- The monitor's private local variables can only be accessed by local monitor functions
- Each function in the monitor can only access variables declared locally within the monitor and its parameters

Monitors and Condition Variables

- Previous definition of a monitor achieves
  - mutual exclusion
  - hiding of wait() and signal() from user
  - loses the ability that semaphores had to enforce order
    - wait() and signal() are used to provide mutual exclusion
    - but have lost the unique ability for one process to signal another blocked process using signal()
    - there is no way to have a process sleep waiting on the signal

- In general, there may be times when one process wishes to signal another process based on a condition, much like semaphores.
  - Thus, augment monitors with *condition variables*.

Condition Variables

- Augment the mutual exclusion of a monitor with an ordering mechanism
  - Recall: Semaphore P() and V() provide both mutual exclusion *and* ordering
  - Monitors alone only provide mutual exclusion
- A condition variable provides ordering
  - Used when one task wishes to wait until a condition is true before proceeding
    - Such as a queue being full enough or data being ready
  - A 2nd task will signal the waiting task, thereby waking up the waiting task to proceed

Monitors and Condition Variables

condition y;

A condition variable *y* in a monitor allows three operations

- y.wait()
  - blocks the calling process
  - can have multiple processes suspended on a condition variable, typically released in FIFO order
  - textbook describes another variation specifying a priority p,   i.e. call y.wait(p)
- y.signal()
  - resumes exactly 1 suspended process.
  - If no process is waiting, then function has *no effect*.
- y.queue()
  - Returns true if there is at least one process blocked on y

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = TRUE;
   }

   void release() {
            busy = FALSE;
            x.signal();
   }

   initialization code() {
    busy = FALSE;
   }
}
```

Condition Variables Example

Block Task 1 until a condition holds true, e.g. queue is empty

```
condition wait_until_empty;
```

Task 1:

```
wait_until_empty.wait();
…  // proceed after queue
         empty
```

Task 2:

```
…
// queue is empty so signal
wait_until_empty.signal();
```

Problem: if Task 2 signals before Task 1 waits, then Task 1 waits *forever* because CV *has no state*!

Condition Variables vs Semaphores

Both have wait() and signal(), but semaphore's signal()/V() *preserves state* in its integer value

```
Semaphore wait_until_empty=0;
```

Task 1:

```
wait(wait_until_empty);
…  // proceed after queue
         empty
```

Task 2:

```
…
// queue is empty so signal
signal(wait_until_empty); //V()
```

if Task 2 signals before Task 1 waits, then Task 1 does *not* wait forever because semaphore has state and remembers earlier signal()

Complex Conditions

- Suppose you want task T1 to wait until a complex set of conditions set by T2 becomes TRUE
  - use a condition variable

```
condition x;
int count=0;
float f=0.0;
```

T1

```
…
while(f!=7.0 && count<=0) {
   x.wait();
}
… // proceed
```

T2

```
…
f=7.0;
count++;
x.signal();
```

Problem: could be a race condition in testing and setting shared variables

Complex Conditions (2)

- Surround the test of the conditions with a mutex to atomically test the set of conditions

```
lock mutex;
condition x;
int count=0;
float f=0.0;
```

T1

```
…
Acquire(mutex);
while(f!=7.0 && count<=0) {
    Release(mutex);
    x.wait();
    Acquire(mutex);
}
Release(mutex);
… // proceed
```

T2

```
…
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```

Complex Conditions (3)

```
lock mutex;
condition x;
int count=0;
float f=0.0;
```

- pthreads replaces complex sequence of release/wait/acquire() with:

```
pthread_cond_wait(&cond_var,&mutex)
```

T1

```
…
Acquire(mutex);
while(f!=7.0 && count<=0) {
  pthread_cond_wait(&x,&mutex);
}
Release(mutex);
… // proceed
```

T2

```
…
Acquire(mutex);
f=7.0;
count++;
Release(mutex);
x.signal();
```

Broadcast Signals

- In some cases, you may want to wake all tasks blocked on a condition variable x, not just one
  - x.signal() only wakes one task
- *x.broadcast()* is a 3rd operation provided for CVs on some systems
  - Wakes all tasks blocked on a CV
  - In pthreads, `pthread_cond_broadcast()` wakes all waiting threads

Monitors

- Declare a monitor as follows (looks somewhat like a C++ class):

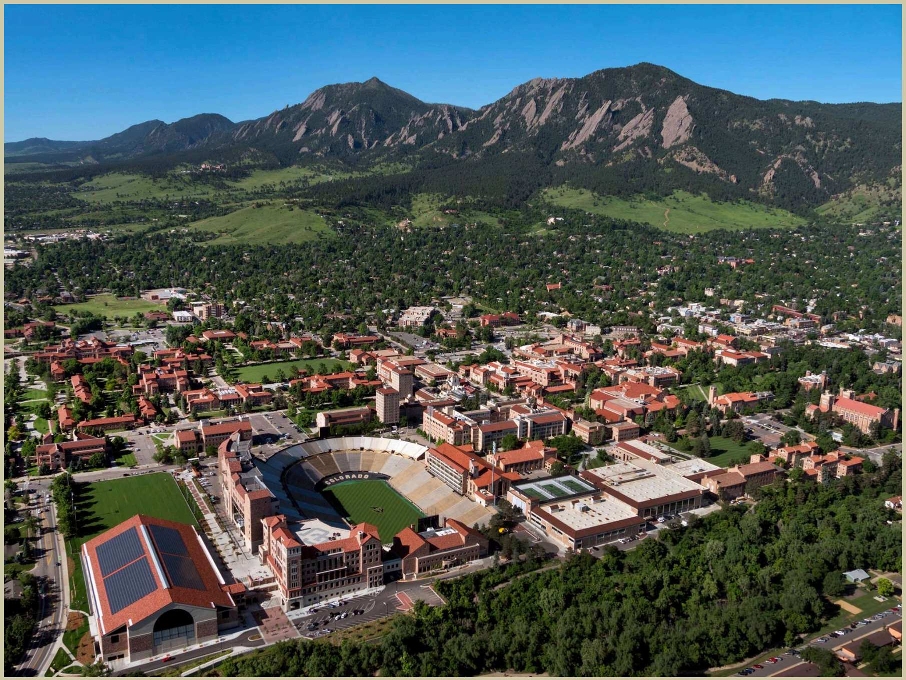
```
monitor monitor_name {
     // shared local variables
        int count;
        data_type data[MAX_COUNT];
        condition y;

     function add_item(...) {
     ...
        y.wait();
     ...
     }

     ...
     function remove_item(...) {
     ...
         y.signal();
     ...
     }

     init_code(...) {
     ...
     }
  }
```

- The monitor's local variables can only be accessed by local monitor functions
- Each function in the monitor can only access variables declared locally within the monitor and its parameters

Monitors and Condition Variables

- Problem: A monitor ensures that only 1 process/thread at a time can be active within a monitor
  - if a process P1 calls y.signal() it would wake another process P2 blocked on y.wait()
  - But we must avoid having two processes at the same time in the monitor
  - Need "wake-up" semantics on a y.signal()
- Two solutions proposed to only have one process active:
  - Hoare semantics, also called signal-and-wait
    - The signaling process P1 waits for P2 to either leave the monitor or wait on another condition, before resuming
  - Mesa semantics, also called signal-and-continue
    - The signaled process P2 continues to wait until the signaling process P1 leaves the monitor or wait on another condition

Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox

University of Colorado
Boulder