Operating Systems Design (CS 423)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

Dequeue if empty?

- What if you wanted to have `dequeue()` wait if the queue is empty?
- Could spin in a loop:

```
dequeue() {
lock(queuelock);
element = NULL;
while (head-next == NULL) {wait;};
if(head->next != NULL) {
element = head->next;
head->next = head->next->next;}
unlock(queuelock);
return element;
}
```

Problem

```
lock(queuelock);
…
while (head-next == NULL) {wait;};
```

- Holding lock while waiting
- No one else can access list (if they observe the lock)
- Wait forever

Dequeue if empty – Try 2

- Could release the lock before spinning:

```
lock(queuelock);
…
unlock(queuelock);
while (head-next == NULL)
 {wait;};
```

- Will this work?

Dequeue if empty – Try 3

- Could release lock and acquire lock on every iteration

```
lock(queuelock);
…
while (head-next == NULL)
 {unlock(queuelock);
  lock(queuelock);};
```

- This will work, but very ineffecient.

Dequeue if empty

- Busy waiting is inefficient, instead you would like to "go to sleep"
  - Waiting list shared between enq and deq
  - Must release locks before going to sleep

Dequeue if empty – Try 4

- Does this work?

```
enqueue() {
 lock();
 find tail;
 add new element;
 if(waiting deq) {
   rem deq from wait;
   wake up deq;
   }
 unlock();
}
```

```
dequeue(){
…
if(queue is empty) {
  release lock();
  add to wait list;
  go to sleep;
  }
…
}
```

Dequeue is empty – try 5

- What if we release lock after adding dequeue() to waiting list, but before going to sleep
- Does this work?

```
...
if(queue is empty) {
 add myself to waiting list;
 release lock;
 go to sleep and wait; }
```

Two types of synchronization

- Mutual exclusion
  - Only one thread can do given operation at a time (e.g., only one person goes shopping at a time)
  - Symmetric
- Ordering constraints
  - Mutual exclusion does not care about order
  - Are situations where ordering of thread operations matter
    - E.g., before and after relationships
  - Asymmetric

Monitors

- Note: Differs from Tanenbaum's treatment
- Monitors use separate mechanisms for the two types of synchronization
  - Use **locks** for mutual exclusion
  - Use **condition variables** for ordering const.
- A monitor = a lock + the condition variables associated with that lock

Condition variables

- Main idea: let threads sleep inside critical section by **atomically**
  - Releasing lock
  - Putting thread on wait queue and go to sleep
  - Each cond var has a queue of waiting threads
- Do you need to worry about threads on the wait queue, but not asleep

Operations on cond. variables

- **Wait():** atomically release lock, put thread on condition wait queue, go to sleep
  - release lock
  - Go to sleep
  - Request lock as part of waking up

- **Signal():** wake up a thread waiting on this condition variable; causes awoken program to request the lock

Operations on cond. variables

- **Broadcast():** wake up all threads waiting on this condition variable; all will request lock
- Note: thread must hold lock when it calls wait()
- Should thread re-establish the invariant before calling wait? How about signal?

Example: Cell phone case at AT&T Store

- Bought a Cell Phone from AT&T store
- "Came" with choice of case
- I looked (lock(case)) and didn't find one I wanted
- They told me they would call me when the next shipment was in
- I left (wait (case, case_arrives))
- They call mid-day to say shipment in (signal(case, case_arrives))
- I go at end of day to get one (lock(case))
- Cases are sold out (wait(case, case_arrives))

Thread-safe queues with monitors

- Is this good enough?

```
enqueue() {
 lock(queueLock);
 find tail;
 add elem to tail;

 signal(queueLock,
        queueCond);

unlock(queueLock);
}
```

```
dequeue() {
 lock(queueLock);

 if(queue empty) {
   wait(queueLock,
        queueCond)};

remove from queue;
unlock(queueLock);
return item;
```

Problem Scenario

lockOwner: Lock Queue: Cond Queue:

Thread1(deq) Thread2(enq) Thread3(deq)

Problem Scenario

lockOwner:T1 Lock Queue: Cond Queue:

Thread1(deq) Thread2(enq) Thread3(deq)

lock();

Problem Scenario

lockOwner: Lock Queue: Cond Queue:T1

Thread1(deq) Thread2(enq) Thread3(deq)

lock();
if(queue empty) {wait()};

Problem Scenario

lockOwner:T2 Lock Queue: Cond Queue:T1

Thread1(deq) Thread2(enq) Thread3(deq)

lock();

if(queue empty) {wait()};

lock();

Problem Scenario

lockOwner:T2 Lock Queue: Cond Queue:T1

Thread1(deq) Thread2(enq) Thread3(deq)

lock();
if(queue empty) {wait()};

lock();
find tail;
add elem;

Problem Scenario

lockOwner:T2 Lock Queue:T3, Cond Queue:T1

Thread1(deq) Thread2(enq) Thread3(deq)

lock();
if(queue empty) {wait()};

lock();
find tail;
add elem;

lock();

Problem Scenario

lockOwner:T2 Lock Queue:T3,T1 Cond Queue:

Thread1(deq) Thread2(enq) Thread3(deq)

```
lock();
if(queue empty) {wait()};
                    lock();
                    find tail;
                    add elem;        lock();
                    signal();
```

Problem Scenario

lockOwner: Lock Queue:T3,T1 Cond Queue:

Thread1(deq) Thread2(enq) Thread3(deq)

```
lock();
if(queue empty) {wait()};
                    lock();
                    find tail;
                    add elem;        lock();
                    signal();
                    unlock();
```

Problem Scenario

lockOwner:T3	Lock Queue:T1	Cond Queue:
Thread1(deq)	Thread2(enq)	Thread3(deq)
lock(); if(queue empty) {wait()};		

	lock();
	find tail;
	add elem;
	signal();
	unlock();	

		lock();
		if(queue empty) { … }. remove from queue;

Problem Scenario

lockOwner:T3	Lock Queue:T1	Cond Queue:
Thread1(deq)	Thread2(enq)	Thread3(deq)
lock(); if(queue empty) {wait()};		

	lock();
	find tail;
	add elem;
	signal();
	unlock();	

		lock();

		if(queue empty) { … }.
		remove from queue;
		unlock();

Problem Scenario

lockOwner:T1 Lock Queue: Cond Queue:

Thread1(deq) Thread2(enq) Thread3(deq)

```
lock();
if(queue empty) {wait()};
                          lock();
                          find tail;
                          add elem;          lock();
                          signal();
                          unlock();
                                             if(queue empty) { … }.
                                             remove from queue;
                                             unlock();
remove from queue;  (Bug!)
```

Thread-safe queues with monitors

- Is this good enough?

```
enqueue() {
 lock(queueLock);
 find tail;
 add elem to tail;

 signal(queueLock,
        queueCond);

unlock(queueLock);
}
```

```
dequeue() {
 lock(queueLock);

 while(queue empty) {
   wait(queueLock,
        queueCond)};

remove from queue;
unlock(queueLock);
return item;
```

Mesa vs. Hoare monitors

- So far have described Mesa monitors
  - When waiter is woken, must contend for the lock with other threads
  - Must re-check condition
- What would be required to ensure that the condition is met when the waiter returns from the wait and start running again?

Mesa vs. Hoare monitors

- Hoare monitors give special priority to woken-up waiter
  - Signaling thread gives up lock
  - Woken-up waiter acquires lock
  - Signaling thread re-acquires lock after waiter unlocks
- We'll stick with Mesa monitors
- Mesa most common, why?

Programming with Monitors

- List shared data needed to solve problem
- Decide which locks protect which data
  - More locks allows different data to be accessed simultaneously, more complicated
  - One lock usually enough in this class
- Put lock...unlock calls around code that uses shared data

Programming with Monitors

- List ordering constraints
  - One condition variable per constraint
  - Condition variable's lock should be the lock that protects the shared data used to eval condition
- Call wait() when thread needs to wait for a condition to be true
  - Use a while loop

Programming with Monitors

- Call signal when a condition changes
- Make sure invariant is established whenever lock is not held
  - E.g., before you call wait