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

Locks (mutexes)

- Locks make "Too Much Milk" really easy to solve!

Elsa:
```
lock(frigdelock);
if (milk low)
   {buy milk}
unlock(fridgelock)
```

Carl:
```
lock(frigdelock);
if (milk low)
   {buy milk}
unlock(fridgelock)
```

- Correct but inefficient
- How to reduce waiting for lock?

How to reduce time lock is held?

Too Much Milk – Solution 7

- Does the following work?

```
lock();
if (milk low & no note) {
    leave note;
    unlock();
    buy milk;
    remove note; }
else { unlock() }
```


Too Much Milk – Solution 7

- Does the following work?

```
lock();
if (milk low & no note) {
    leave note;
    unlock();  buy milk;
    lock();  remove note;   unlock(); }
else { unlock() }
```


Queues without Locks

```
enqueue (new_element, head) {
// find tail of queue
for(ptr=head; ptr->next != NULL;
      ptr = ptr->next);
// add new element to tail
ptr->next = new_element;
new_element->next = NULL;
}
```


Queues without Locks

```
dequeue(head, element) {
element = NULL;
// if something on queue, remove it
if(head->next != NULL) {
element = head->next;
head->next = head->next->next;}
return element;
}
```

- What bad things can happen if two threads manipulate the queue at the same time?

Thread-safe Queues with Locks

```
enqueue (new_elt, head) {
lock(queuelock);
// find tail of queue
for(ptr=head; ptr->next !=
  NULL;
    ptr = ptr->next);
// add new element to tail
ptr->next = new_elt;
new_elt->next = NULL;
unlock(queuelock);
}
```

```
dequeue(head, elt) {
lock(queuelock);
element = NULL;
// remove if possible
if(head->next != NULL) {
elt = head->next;
head->next =
 head->next->next;}
unlock(queuelock);
return elt;
}
```


Invariants for multi-threaded queue

- Can `enqueue()` unlock anywhere?
- Stable state called an **invariant**
  - I.e., something that is "always" true
- Is the invariant ever allowed to be false?

Invariants for multi-threaded queue

- In general, must hold lock when manipulating shared data
- What if you're only reading shared data?

Enqueue – Can we do better?

```
enqueue() {
lock
find tail of queue
unlock

lock
add new element to tail of queue
unlock
}
```

- Is this better?

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
  - Re-acquire lock
- **Signal():** wake up a thread waiting on this condition variable

Operations on cond. variables

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