ECS 251: Monitors

Sam King

Administrative

- Please don’t copy solutions from the internet, especially if you don’t cite
  - If we catch you, you will get an F for this class and I will turn you in to the academic integrity people

Administrative

- HW2 due today
- HW3 due in one week

Administrative

- How we’re going to handle the rest of the quarter
  - Lecture time will be for group meetings
  - Discussion time will be for quizzes on the advanced reading
  - I will also talk about research, the papers, technical topics, etc

Administrative

- Project proposals due on 2/7, first big milestone for the project
- We’ll talk about it in more detail on Thursday
- It’s going to be a bit challenging because we haven’t been reading papers, but I’m assuming that you have been reading papers in other classes
- Suggestion: read papers from this class ahead of time and try to reverse engineer the outline of the introduction

Administrative

- Updating grading, now have 6 quizzes, no sprint planning and lowered the points for your final presentation

```
enqueue() {
    lock(queueLock);
    // find tail of queue
    for(ptr=head; ptr->next
                  != NULL;
        ptr = ptr->next)
        ;

    // add new element
    ptr->next = new_element
    new_element->next =
        NULL
    unlock(queueLock);
}
```

```
dequeue() {
    lock(queueLock);
    element = NULL;
    while(head->next ==
          NULL) {
         unlock(queueLock);
         lock(queueLock);
    }
    element = head->next;
    head->next =
        head->next->next;

    unlock(queueLock);
    return element;
}
```

- 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(queue is empty) {
        release lock
        add to wait list
        go to sleep
    }
}
```

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

Does this work?

- What if we release lock after adding dequeuer to waiting list, but before going to sleep

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

Does this work?

Two types of synchronization

- Mutual exclusion
  - Only one thread can do a certain operation at one 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

- 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
- **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?

Condition variables

- J Crew shirt example

Thread-safe queue w/monitors

```
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
}
```

Multi threaded queue

- Note: natural to hold lock when calling wait
  - Also natural (but not required) to hold it when signaling
- Is there any problem with the “if” in the dequeue()?

Multi threaded queue

- Note: natural to hold lock when calling wait
  - Also natural (but not required) to hold it when signaling
- Is there any problem with the “if” in the dequeue()?
  - Must reason about wait properly:
    - Release lock
    - Sleep and wait for wakeup
    - Re-acquire lock

lockOwner

lockQueue

condQueue

Thread 1 Thread 2 Thread 3

lockOwner Thread 1

lockQueue

condQueue

Thread 1 (deq) Thread 2 Thread 3

lock

if(queue empty)

lockOwner

lockQueue

condQueue Thread 1

Thread 1 (deq) Thread 2 Thread 3

lock

if(queue empty)

wait

lockOwner	Thread 2
lockQueue	
condQueue	Thread 1

Thread 1 (deq)	Thread 2 (enq)	Thread 3
lock	lock	
	add item	
if(queue empty)		
wait		

lockOwner Thread 2

lockQueue

condQueue

Thread 1 (deq)	Thread 2 (enq)	Thread 3
	lock	
lock		
	add item	
if(queue empty)	signal	
wait		

lockOwner

lockQueue

condQueue

Thread 1 (deq)	Thread 2 (enq)	Thread 3
	lock	
lock		
	add item	
if(queue empty)	signal	
wait		
	unlock	

lockOwner	Thread 3
lockQueue	Thread 1
condQueue	

Thread 1 (deq)	Thread 2 (enq)	Thread 3 (deq)
	lock	lock
lock		
	add item	
if(queue empty)	signal	
wait		
	unlock	
		remove item

lockOwner Thread 1

lockQueue

condQueue

Thread 1 (deq)	Thread 2 (enq)	Thread 3 (deq)
	lock	lock
lock		
	add item	
if(queue empty)	signal	
wait		
	unlock	
		remove item
		unlock

lockOwner Thread 1

lockQueue

condQueue

Thread 1 (deq)	Thread 2 (enq)	Thread 3 (deq)
	lock	lock
lock		
	add item	
if(queue empty)	signal	
wait	unlock	
		remove item
		unlock
remove item (bug)		

Tips for prog. w/ monitors

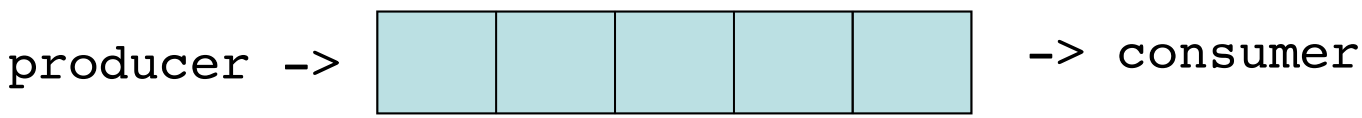
- List the shared data needed to solve the problem
- Decide which locks will protect which data
  - More locks allows different data to be accessed simultaneously, more complicated
  - One lock usually enough in this class
- Put lock…unlock calls around the code that uses shared data

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

Producer-consumer (bounded buffer)

- Problem: producer puts things into a shared buffer, consumer takes them out.
  - Synchronization for coordinating



  - Unix pipeline (gcc calls cpp | cc1 | cc2 | as)
  - Buffer between allows them to operate independently
  - What would execution be like without buffer?
- Coke machine
  - Delivery person (producer)
  - Students (and professors) buy cokes (consumer)
  - Coke machine has finite space

Producer-consumer using monitors

- Operations
  - Add coke to machine
  - Take coke out of machine
- Variables
  - Shared data for the coke machine
    - Assume can hold “max” (maxCokes) cokes
  - numCokes (number of cokes in machine)
- One lock (cokeLock) to protect shared data
  - Fewer locks easier to program, less concur.
- Ordering constraints
  - Consumer must wait for producer to fill buffer if all buffers are empty (hasCoke)
  - Producer must wait for consumer to empty buffer if buffer is completely full (hasRoom)

```
consumer() {
  lock(cokeLock);



  take one coke out
    of machine


  unlock(cokeLock)
}
```

```
producer(){
  lock(cokeLock)



  add one coke to
    machine


  unlock(cokeLock)
}
```

```
consumer() {
  lock(cokeLock);

  while(numCokes = 0)
  {
    wait(cokeLock,
         hasCoke)
  }

  take coke out
    of machine

  signal(hasRoom)

  unlock(cokeLock)
}
```

```
producer(){
  lock(cokeLock)

while(numCokes = max)
  {
    wait(cokeLock,
         hasRoom);
  }

  add coke to
    machine

  signal(hasCoke)

  unlock(cokeLock)
}
```

- What if we wanted to have producer continuously loop?

```
Producer() {

  lock(cokeLock);
  while(1) {

    while(numCokes == max) {
      wait(cokeLock, hasRoom);
    }
    add coke to machine
    signal(hasCoke);
  }
  unlock(cokeLock);

}
```

- What if we added a sleep?

```
Producer() {

  lock(cokeLock);
  while(1) {
       sleep(1 hour);
    while(numCokes == max) {
      wait(cokeLock, hasRoom);
    }
    add coke to machine
    signal(hasCoke);
  }
  unlock(cokeLock);

}
```

```
consumer() {
  lock(cokeLock);

  while(numCokes = 0)
  {
    wait(cokeLock,
         hasCokeRoom)
  }

  take coke out
    of machine

  signal(hasCokeRoom)

  unlock(cokeLock)
}
```

```
producer(){
  lock(cokeLock)

while(numCokes = max)
  {
    wait(cokeLock,
         hasCokeRoom);
  }

  add coke to
    machine

  signal(hasCokeRoom)

  unlock(cokeLock)
}
```

- Multiple conditions for a single condition variable is probably a bad idea
  - Hard to reason about changes in conditions mean
- Can we always use broadcast() instead of signal()?