ECS 251: Thread synchronization

Sam King

Administrative

- Project groups due today
- Project ideas due on Thursday

Administrative

- Quizzes moving to Tuesdays to line up more closely with the homework
- Cancel class on Thursday
  - Art will have extra office hours

Administrative

- Last time: Too Much Milk using atomic loads and stores
- This time: Locks
- Next time: Condition variables, lock implementation

Too much milk (solution #3)

- Idea: have a way to decide who will buy milk when both leave notes at the same time. Have Sam hang around to make sure job is done.

Sam:

```
leave noteSam
while (noteAnne) {
    do nothing
}
if (noMilk) {
    buy milk
}
remove noteSam
```

Anne:

```
leave noteAnne


if (no noteSam) {
    if( noMilk ) {
        buy milk
    }
}
remove noteAnne
```

Too much milk (solution #3)

- Sam’s “while(noteAnne)” prevents him from running his critical section at the same time as Anne’s
- Proof of correctness
  - “Exercise to the reader”
- Correct, but ugly
  - Complicated
  - Asymmetric
  - Inefficient
    - Sam consumes CPU time while waiting (**Busy Waiting**)

Higher-level synchronization

- Problem: could solve “too much milk” using atomic loads/stores, but messy
- Solution: raise the level of abstraction to make life easier for the programmer

Concurrent programs
High-level synchronization provided by software
Low-level atomic operations provided by hardware

Locks (mutexes)

- A lock is used to prevent another thread from entering a critical section
- Two operations
  - Lock(): wait until lock is free, then acquire

```
do {
    if (lock == LOCK_FREE) {
        lock = LOCK_SET
        break
    }
} while(1)
```

  - Unlock(): `lock = LOCK_FREE`

Locks (mutexes)

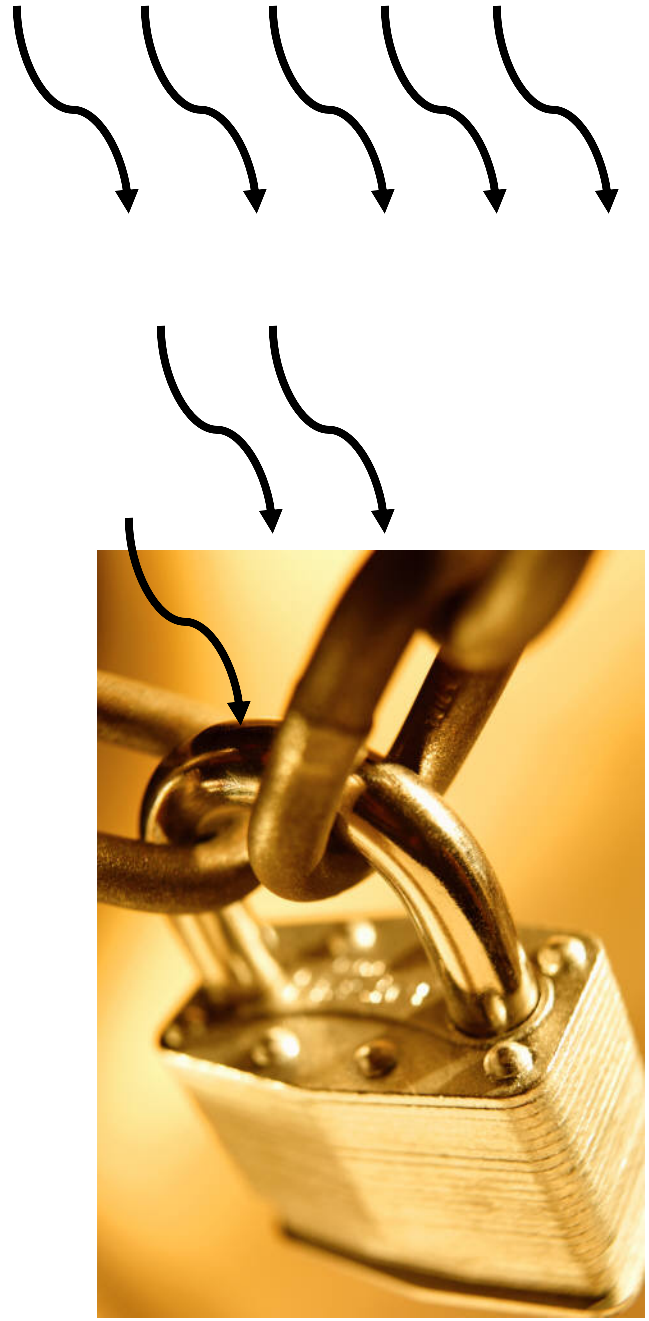
- Why was the “note” in Too Much Milk solutions #1 and #2 not a good lock?

- For elements of locking
  - Lock is initialized to be free
  - Acquire lock before entering a critical section
  - Wait to acquire lock if another thread already holds
  - Release lock after exiting critical section

- All synchronization involves waiting
- Thread can be **running**, or **blocked** (waiting)

Locks

- Locks -- shared variable among all thread
- Multiple threads share locks
  - Only affects threads that try to acquire locks
  - Like putting a padlock on the fridge



Lock variables

- **Critical section** -- part of the program where threads access shared (global) state
- Locks -- shared variables used to enforce **mutual exclusion**
  - Can have multiple lock variables

Locks (mutexes)

- Locks make “Too Much Milk” really easy to solve!

```
Sam:
Lock(fridgeLock)
If (noMilk) {
    buy milk
}
Unlock(fridgeLock)
```

```
Anne:
Lock(fridgeLock)
If (noMilk) {
    buy milk
}
Unlock(fridgeLock)
```

- Correct, but inefficient
- How to minimize the time the lock is held?

Too Much Milk Solution

- Does the following solution work

```
lock()
if(noMilk && noNote) {
    leave note "I'm buying milk"
    unlock()
    buy milk
    remove note
} else {
    unlock()
}
```

Too Much Milk Solution

- Does the following solution work

```
lock()
if(noMilk && noNote) {
     leave note “I’m buying milk”
     unlock()
     buy milk
     lock()
     remove note
     unlock()
} else {
     unlock()
}
```

Thread-safe queue w / locks

```
enqueue() {

    // 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

}
```

Thread-safe queue w / locks

```
dequeue() {

    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?

```
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;
    if(head->next != NULL){
        element =
            head->next;
        head->next =
          head->next->next;
    }
    unlock(queueLock);
    return element;
}
```

Invariants for multi-threaded queue

- Can enqueue() unlock anywhere?
- Stable state called an **invarient**
  - 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

- What about the following locking scheme:

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

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

- What if you wanted to have dequeue() wait if the queue is empty?

- Could spin in a loop

```
Dequeue() {
    …
    while(head->next == NULL)
        ;
    …
}
```

- Could release the lock before spinning

```
unlock();
while(head->next == NULL)
    ;
```

Too Much Milk Solution

- Does the following solution work

```
lock()
If(noNote && noMilk) {
    leave note “I’m buying milk”
    unlock()
    buy milk
    remove note
} else {
    unlock()
}
```

```
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