Concurrency - II

Recitation – 3/24
Nisarg Raval
Slides by Prof. Landon Cox and Vamsi Thummala
So far...

lock ()
if (noNote && noMilk) {
    leave note “at store”
    unlock ()
    buy milk
    lock ()
    remove note
    unlock ()
} else {
    unlock ()
}

Only hold lock while handling shared resource.
Example: thread-safe queue

```c
enqueue () {
    lock (qLock)
    // ptr is private
    // head is shared
    new_element = new node();
    if (head == NULL) {
        head = new_element;
    } else {
        node *ptr;
        // find queue tail
        for (ptr=head; ptr->next!=NULL; ptr=ptr->next){}
        ptr->next=new_element;
    }
    unlock(qLock); // Safe?
    new_element->next=0;
}
```

- **Can lock/unlock anywhere?**
- **Where should we put lock/unlock?**
- **Shared data must be in consistent state**
Data Invariant

- States of shared data
  - “always” true

- Queue Invariants?
  - Each node appears once
  - Enqueue = prior list + new node (last)
  - Dequeue removes exactly one node (first)

- When is the invariant broken?
  - Only while lock is held
  - Only by thread holding the lock
BROKEN INVARIANT
(CLOSE AND LOCK DOOR)

http://www.flickr.com/photos/jacobaaron/3489644869/
INVARIANT RESTORED

(UNLOCK DOOR)

http://www.flickr.com/photos/jacobaaron/3489644869/
Data Invariant

- *What about reading shared data?*
  - Need lock otherwise other thread can break invariant
  - e.g. thread A prints queue while thread B enqueues

- *Rule: Hold the lock while manipulating shared data*
enqueue () {
    lock (qLock)
    // ptr is private
    // head is shared
    new_element = new node();
    if (head == NULL) {
        head = new_element;
    } else {
        node *ptr;
        // find queue tail
        for (ptr=head;
            ptr->next!=NULL;
            ptr=ptr->next){}
        unlock(qLock);
        lock(qLock);
        ptr->next=new_element;
    }
    new_element->next=0;
    unlock(qLock);
}

\- All is well as I’m Always holding lock while accessing shared data!
  \- \textit{ptr may not point to tail}

\- Thinking about individual accesses is not enough

\- Must reason about dependencies between accesses
Ordering Constrain

- We want dequeue to wait while queue is empty

```c
dequeue () {
    lock (qLock);
    element=NULL;
    while (head==NULL) {} // remove head
    // remove head
    element = head;
    head = head->next;
    unlock (qLock);
    return element;
}
```
How about this?

- *Release lock before spinning*

dequeue () {
    lock (qLock);
    element=NULL;
    unlock (qLock);
    while (head==NULL) {} 
    lock (qLock);
    // remove head
    element = head;
    head = head->next;
    unlock (qLock);
    return element;
}

Head might be NULL
Does this work?

dequeue () {
    lock (qLock);
    element=NULL;
    while (head==NULL) {
        unlock (qLock);
        lock (qLock);
    }
    // remove head
    element = head;
    head = head->next;
    unlock (qLock);
    unlock (qLock);
    return element;
}
Ideal Solution

- *How about putting dequeue thread to sleep?*
  - Add self to “waiting list”
  - Enqueue thread can wake up dequeue thread

- *But what about lock?*
  - Dequeue can not sleep with lock
  - Enqueue would never be able to add
Release lock before sleeping

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueuer waiting) {
        remove from wait list
        wake up dequeueuer
    }
    release lock
}

dequeue () {
    acquire lock
    ...
    if (queue empty) {
        release lock
        add self to wait list
        sleep
        acquire lock
    }
    ...
    release lock
}
Release lock before sleeping

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueuer waiting) {
        remove from wait list
        wake up dequeueuer
    }
    release lock
}

decqueue () {
    acquire lock
    ... 
    if (queue empty) {
        release lock
        add self to wait list
        sleep
        acquire lock
    }
    ... 
    release lock
}
Release lock before sleeping

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueer waiting) {
        remove from wait list
        wake up dequeueer
    }
    release lock
}

decqueue () {
    acquire lock
    ...
    if (queue empty) {
        release lock
        add self to wait list
        sleep
        acquire lock
    }
    ...
    release lock
}
Release lock before sleeping

```
enqueue () {  
  acquire lock
  find tail of queue
  add new element
  if (dequeuer waiting) {
    remove from wait list
    wake up dequeuer
  }
  release lock
}

dequeue () {  
  acquire lock
  ...
  if (queue empty) {
    release lock
    add self to wait list
    sleep
    acquire lock
  }
  ...
  release lock
}
```

Thread can sleep forever!
Does this work?

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeuer waiting) {
        remove from wait list
        wake up dequeuer
    }
    release lock
}

dequeue () {
    acquire lock
    ...
    if (queue empty) {
        add self to wait list
        release lock
        sleep
        acquire lock
    }
    ...
    release lock
}

Waitlist is a shared resource!
Use while instead of if

enqueue () {
    acquire lock
    find tail of queue
    add new element
    if (dequeueer waiting) {
        remove from wait list
        wake up dequeueer
    }
    release lock
}

depqueue () {
    acquire lock
    ...
    if (queue empty) {
        add self to wait list
        release lock
        sleep
        acquire lock
    }
    ...
    release lock
}
Raise the level of abstraction

- **Mutual Exclusion**
  - Ensures one thread access the critical section
  - Use *locks*

- **Ordering constraints**
  - “before-after” relationship
  - One thread wait for/signal another thread
  - Use *Monitors: lock + conditional variable*
Monitor: Lock + CV

- Conditional Variable: Maintains state
  - Queue of waiting threads on a lock

- Internal atomic actions

```
// begin atomic
release lock
put thread on wait queue
go to sleep
// end atomic
```
CV – Operations

```
Lock always held
wait (lock){ //Also, wait() in Java
    release lock
    put thread on wait queue
    go to sleep
    // after wake up
    acquire lock
}
```

```
Lock always held
signal (){ //notify() in Java
    wakeup one waiter (if any)
}
```

```
Lock usually held
Broadcast (){ //notifyAll() in Java
    wakeup all waiters (if any)
}
```
Read/Write Lock

- **Improve standard lock for multiple readers**
- **Read: Shared Access**
  - Can assign locks to multiple readers only when no threads are requesting write access
- **Write: Exclusive Access**
  - No other threads are reading or writing
- **Terminology**
  - Mutex: $m$
  - Conditional Variable: $c$
  - Number of readers: $i$
  - $i = -1$ to represent writer
Read/Write Lock

```java
AcquireShared() {
    lock(m);
    while (i<0) {
        wait(m, c);
    }
    i = i + 1;
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while (i!=0) {
        wait(m, c);
    }
    i = -1;
    unlock(m);
}

ReleaseShared() {
    lock(m);
    i = i - 1;
    if (i==0) {
        signal(c);
    }
    unlock(m);
}

ReleaseExclusive() {
    lock(m);
    i = 0;
    signal(c);
    unlock(m);
}

How to handle multiple Waiting Readers?
Using Broadcast

AcquireShared() {
    lock(m);
    while (i<0) {
        wait(m,c);
    }
    i = i + 1;
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while (i!=0) {
        wait(m,c);
    }
    i = -1;
    unlock(m);
}

ReleaseShared() {
    lock(m);
    i = i - 1;
    if (i==0) {
        signal(c);
    }
    unlock(m);
}

ReleaseExclusive() {
    lock(m);
    i = 0;
    broadcast(c);
    unlock(m);
}
Using Broadcast

AcquireShared() {
    lock(m);
    while (i<0) {
        wait(m, c);
    }
    i = i + 1;
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while (i!=0) {
        wait(m, c);
    }
    i = -1;
    unlock(m);
}

ReleaseShared() {
    lock(m);
    i = i - 1;
    if (i==0) {
        signal(c);
    }
    unlock(m);
}

ReleaseExclusive() {
    lock(m);
    i = 0;
    broadcast(c);
    unlock(m);
}

Spurious Wakeups?
Two CV – Read and Write

AcquireShared() {
    lock(m);
    Rwait = Rwait + 1;
    while (i<0) {
        wait(m, cR);
    }
    Rwait = Rwait - 1;
    i = i + 1;
    unlock(m);
}

ReleaseShared() {
    lock(m);
    i = i - 1;
    if (i==0) {
        signal(cW);
    }
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while (i!=0) {
        wait(m, cW);
    }
    i = -1;
    unlock(m);
}

ReleaseExclusive() {
    lock(m);
    i = 0;
    if (Rwait > 0) {
        broadcast(cR);
    } else {
        signal(cW);
    }
    unlock(m);
}
Spurious Lock Conflicts

ReleaseShared() {
    lock(m);
    i = i - 1;
    if(i==0){
        signal(cW);
    }
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while(i!=0){
        wait(m,cW);
    }
    i = -1;
    unlock(m);
}

ReleaseShared() {
    sendSignal = false;
    lock(m);
    i = i - 1;
    if(i==0){
        sendSignal = true;
    }
    unlock(m);
    if(sendSignal)
        signal(cW);
}
Starvation?

AcquireShared() {
    lock(m);
    Rwait = Rwait + 1;
    while(i<0){wait(m,cR);}
    Rwait = Rwait - 1;
    i = i + 1;
    unlock(m);
}

ReleaseShared() {
    lock(m);
    i = i - 1;
    if(i==0) {
        signal(cW);
    }
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while(i!=0){wait(m,cW);}
    i = -1;
    unlock(m);
}

- Reader A calls AcquireShared  \( \rightarrow i = 1 \)
- Reader B calls AcquireShared  \( \rightarrow i = 2 \)
- Writer calls AcquireExclusive  \( \rightarrow \text{Blocked} \)
- Reader A calls ReleaseShared  \( \rightarrow i = 1 \)
- Reader C calls AcquireShared  \( \rightarrow i = 2 \)
- ...
Starvation?

AcquireShared() {
    lock(m);
    Rwait = Rwait + 1;
    while(i<0){wait(m,cR);}
    Rwait = Rwait - 1;
    i = i + 1;
    unlock(m);
}

AcquireShared() {
    lock(m);
    Rwait = Rwait + 1;
    if(Wwait > 0){
        signal(cW);
        wait(m,cR);
    }
    while(i<0){wait(m,cR);}
    Rwait = Rwait - 1;
    i = i + 1;
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    while(i!=0){wait(m,cW);}
    i = -1;
    unlock(m);
}

AcquireExclusive() {
    lock(m);
    Wwait = Wwait + 1;
    while(i!=0){wait(m,cW);}
    Wwait = Wwait - 1;
    i = -1;
    unlock(m);
}
Coding Practices

- **(Almost) Never sleep()**

- **(Always) Loop before you leap!**
  - While(CV is true){ Wait() }

- **Avoid using synchronized(this)**
  - Lock is held and released in between method
  - Hard to read/follow
  - Instead divide the code into modules and synchronize on methods
Coding Practices

- **Pool of threads**
  - e.g. client threads in web server

- **Careful while accessing data packed tightly**
  - e.g. different mutex for different fields in Union

  ```c
  Union{ int i; int j; } u;
  ```
  - lock(u.i) & lock(u.j) – Not a good idea
  - lock(u)
Metrics for Elevator Scheduling

- **Service Time**
  - Time between pushing the button and exit the elevator
  - Can be approximated by wait time

- **Efficiency**
  - Amount of total work done (Energy)
  - Number of floors visited by the elevator

- **Fairness**
  - Variation in the service time
Metrics for Elevator Scheduling

- **Service Time** - **Minimize**
  - Time between pushing the button and exit the elevator
  - Can be approximated by wait time

- **Efficiency** - **Minimize**
  - Amount of total work done (Energy)
  - Number of floors visited by the elevator

- **Fairness** - **Maximize**
  - Variation in the service time
FCFS – First Come First Served

- Service in the order of request
- Simple
- No Starvation
- How good it is?
Example

- The elevator is currently servicing the 10th floor
- Order of requests from riders at the 10th floor:
  5 (down), 35 (up), 2 (down), 14 (up), 12 (up), 21 (up),
  3 (down), 9 (down), 22 (up), 20 (up)
- To simplify, let us assume everyone gets in

```
20  2  22  12  9  6  3  18  21  9  12  2  14  12  2  33  35  30  5
```

- Total service time (assuming 1 unit time per floor serviced):
  5 + 30 + 33 + 12 + 2 + 9 + 18 + 6 + 12 + 2 = 129, Avg: 12.9
- Can we do better?
  - Service the closest floor
SSTF – Shortest Seek Time First

- Go to the closest floor in the work queue
- Reduces total seek time compared to FCFS
- Order of requests from riders at the 10th floor:
  5 (down), 35 (up), 2 (down), 14 (up), 12 (up), 21 (up),
  3 (down), 9 (down), 22 (up), 20 (up)

- Total service time (assuming 1 unit time per floor serviced):
  \[1 + 3 + 2 + 6 + 1 + 1 + 13 + 30 + 2 + 1 = 60, \text{ Avg: 6}\]
- Disadvantages:
  - Starvation possible
  - Switching directions may slow down the actual service time
- Can we do better? Reorder the requests w.r.t direction
• Start servicing in a given direction to the end
  • Change direction and start servicing again
• Order of requests from riders at the 10th floor:
  5 (down), 35 (up), 2 (down), 14 (up), 12 (up), 21 (up),
  3 (down), 9 (down), 22 (up), 20 (up)

• Total service time (assuming 1 unit time per floor serviced):
  \[1 + 4 + 2 + 1 + 10 + 2 + 6 + 1 + 1 + 1 + 13 = 41, \text{ Avg: 4.1}\]
• Advantages
  • Reduces variance in seek time
• Can we do better?
Circular SCAN (C – SCAN)

- Start servicing in a given direction to the end
  - Go to the first floor without servicing any requests;
  - Restart servicing
- Order of requests from riders at the 10th floor:
  5 (down), 35 (up), 2 (down), 14 (up), 12 (up), 21 (up),
  3 (down), 9 (down), 22 (up), 20 (up)

- Total service time (assuming 1 unit time per floor serviced):
  \[1 + 4 + 2 + 1 + 1 + 11 + 2 + 6 + 1 + 1 + 1 + 13 = 43, \text{ Avg: 4.3}\]
- Advantages
  - More fair compared to SCAN
  - Is this what you expect in a real-world elevator?
Elevator Scheduling

- At least one difference from C-SCAN
  - Direction of pick up
- Order of requests from riders at the 10th floor:
  5 (down), 35 (up), 2 (down), 14 (up), 12 (up), 21 (up),
  3 (down), 9 (down), 22 (up), 20 (up)

- Total service time (assuming 1 unit time per floor serviced):
  \[ 1 + 4 + 2 + 1 + 1 + 9 + 2 + 2 + 6 + 1 + 1 + 1 + 13 = 43, \text{ Av: } 4.3 \]
- Can you do better?
  - We look forward to your lab submissions
Disk Scheduling

- Similar to Elevator Scheduling

- Queue of jobs waiting to access disk
  - Read jobs
  - Write jobs

- Queue Entry
  - Pointer to memory location to read/write from/to
  - Sector number to access
  - Pointer to the next job