Winter 2025

EECS 482: Introduction to Operating Systems

Lecture 5: Monitors

Prof. Ryan Huang

Administration

Project 1 due this Wednesday

Reminder: honor code

- You may use online resources including ChatGPT, Copilot **only** for general programming help
- You may NOT use them for help specific to EECS 482 projects

Monitors

Two mechanisms: one for each type of synchronization

- Locks for mutual exclusion
- Condition variables for ordering constraints

A monitor = a lock + the condition variables associated with that lock

Mesa vs. Hoare monitors

Mesa

```
while (!condition)
    wait(cond_var);
```

- When waiter is woken, the condition might have changed
  - another thread got the lock first
- So it must re-check the condition it was waiting for

Hoare

```
if (!condition)
    wait(cond_var);
```

- Special priority to the woken-up waiter
- Signal thread immediately gives up lock
  - reacquires lock after waiter unlocks

We (and most/all Oses) use Mesa monitors

- Waiter is solely responsible for ensuring condition is met
- Always use while(...) {wait}

Monitor programming: overall design

Step 1: Think about each thread independently

- Each thread should try to make as much forward progress as possible (be greedy)
- A thread should wait only when it is unable to proceed

Step 2: List the shared data needed for the problem (e.g., w, x, y, z)

Monitor programming: mutex

Step 2: List the shared data needed for the problem (e.g., $\underbrace{w,\ x}_{\text{lock A}},\ \underbrace{y,\ z}_{\text{lock B}}$)

Step 3: Assign a lock per group of related, shared data

- lock…unlock around accesses to shared data
- Coarse-grained versus fine-grained locking

Monitor programming: ordering

Step 4: List before-after conditions and assign condition variables to each condition

When is a thread unable to make progress?

Use one condition variable for each before-after condition

- Condition variable belongs to the lock
  that protects the shared data
  that is used to evaluate the condition

Waiting thread uses `while(condition){cv.wait}`

Signaling thread calls `signal` or `broadcast` when it changes state that may allow another thread to proceed

Typical monitor code

```
lock

// wait if needed

while (condition) {

    wait

}

do stuff

signal or broadcast about the stuff you did

unlock
```

Producer-consumer (bounded buffer)

Producers fill a shared buffer; consumers empty it

Need to synchronize actions of producers and consumers

Why use a shared buffer?

- Allows producers and consumers to operate somewhat independently (more concurrency)

Used in many situations

- Unix pipes (grep “keyword” foo.txt | wc -l)
- Project 1

Coke machine with monitors

Problem definition

- Coke machine can hold at most MAX cokes
- Delivery person (producer) adds one coke to machine
- Consumer buys one coke

Step 1: Think about threads independently

- Delivery threads add cokes, consumer threads remove cokes

Step 2: Identify shared state

- State of coke machine: `int numCokes`

Step 3: Assign locks

- One lock (`mutex cokeLock`) to protect all shared data

Coke machine with monitors

Step 4: List before-after conditions and assign a condition variable to each condition

- Consumers wait while there are no cokes (`numCokes == 0`)
  - `cv waitingConsumers`
- Delivery persons wait while the cokes are full (`numCokes == MAX`)
  - `cv waitingProducers`

Coke machine with monitors

```
Consumer()

// take coke out of machine
```

```
Producer()

// add coke to machine
```

Coke machine with monitors

```
int numCokes
mutex cokeLock
cv waitingConsumers, waitingProducers
```

Consumer()

```
cokeLock.lock()

while (numCokes == 0) {
    waitingConsumers.wait(cokeLock)
}

// take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer()

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait(cokeLock)
}

// add coke to machine
numCokes++

waitingConsumers.signal()

cokeLock.unlock()
```

Reducing number of signals?

```
int numCokes
mutex cokeLock
cv waitingConsumers, waitingProducers
```

Consumer()

```
cokeLock.lock()

while (numCokes == 0) {
    waitingConsumers.wait(cokeLock)
}

// take coke out of machine
numCokes--

waitingProducers.signal()

cokeLock.unlock()
```

Producer()

```
cokeLock.lock()

while (numCokes == MAX) {
    waitingProducers.wait(cokeLock)
}

// add coke to machine
numCokes++

if (numCokes == 1)
    waitingConsumers.signal()

cokeLock.unlock()
```

Would this still work?

Reducing number of signals

`numCokes = 0`

C1 and C2 waiting on `waitingConsumers`

P1 increments `numCokes` to 1 and signals

P2 increments `numCokes` to 2, but does not signal

Only one of C1 and C2 wakes up!

Reducing condition variables?

```
int numCokes
mutex cokeLock
cv waiting
```

```
Consumer()

cokeLock.lock()

while (numCokes == 0) {
    waiting.wait(cokeLock)
}

// take coke out of machine
numCokes--

waiting.signal()

cokeLock.unlock()
```

```
Producer()

cokeLock.lock()

while (numCokes == MAX) {
    waiting.wait(cokeLock)
}

// add coke to machine
numCokes++

waiting.signal()

cokeLock.unlock()
```

Would this still work?

Reducing condition variables

Say `MAX = 1`, and `numCokes = 0`

C1 and C2 wait

P1 increments `numCokes` to `1` and signals

- Wakes up C1

P2 waits because `numCokes = MAX`

C1 decrements `numCokes` to `0` and signals

- May wake up C2!

Why is this bad?

Reader-writer locks

Example: multiple threads would like to read or write Wolverine Access course catalog

Threads need to acquire lock to read or write shared data

Student:
```
lock()
browse course catalog
unlock()
```

Administrator:
```
lock()
change course catalog
unlock()
```

Any concern with this code?

How to safely allow more concurrency?

How to safely allow more concurrency?

Idea: threads disclose whether they intend to read or write the shared data through separate read/write lock methods

Student:

~~lock()~~ readerLock()
browse course catalog
~~unlock()~~ readerUnlock()

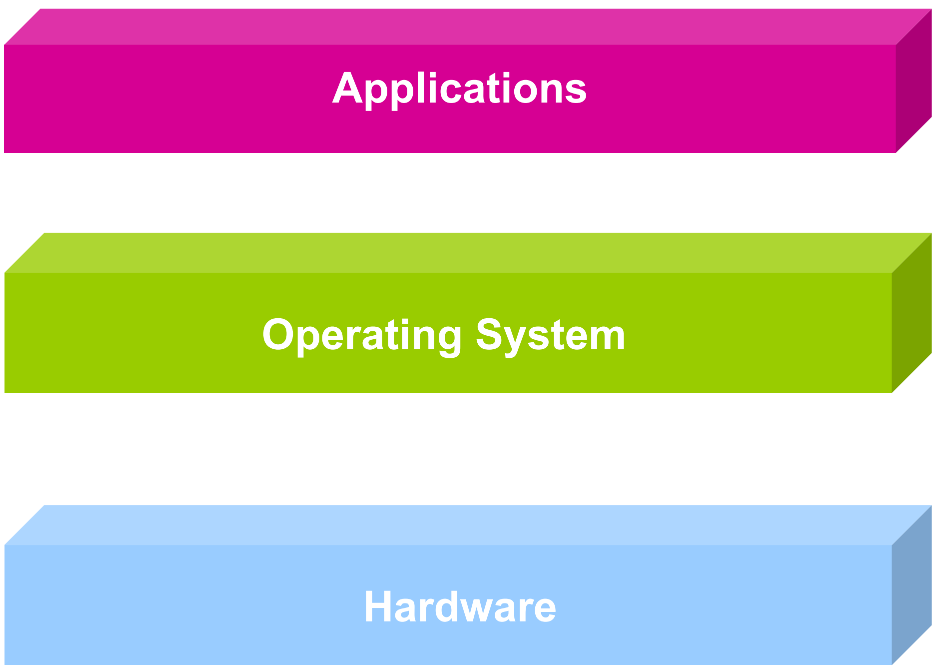
Administrator:

~~lock()~~ writerLock()
change course catalog
~~unlock()~~ writerUnlock()

Allow multiple concurrent readers, if no threads are writing data

Allow a single writer, if no other threads are reading or writing

Another level of abstraction



Even higher-level synchronization operations

Concurrent programs

(readerLock, readerUnlock, writerLock, writerUnlock)

Higher-level synchronization operations

(lock, monitor, semaphore)

Atomic operations

(load/store, interrupt enable/disable, test&set)

Reader-writer lock with monitors

Step 1: Think about threads independently

Step 2: Identify shared state (to implement reader-writer lock methods)

- `numReaders, numWriters`

Step 3: Assign locks to shared state:

- `mutex rwLock`

Step 4: List the before-after conditions and assign a condition variable for each condition:

- `readerLock` waits while `numWriters>0`
  - `cv waitingReaders`
- `writerLock` waits while `numReaders>0 || numWriters>0`
  - `cv waitingWriters`

Reader-writer lock with monitors

```
int numReaders, numWriters
mutex rwLock
cv waitingReaders, waitingWriters
```

```
readerLock () {

}
```

```
writerLock() {

}
```

```
readerUnlock() {

}
```

```
writerUnlock() {

}
```

Reader-writer lock with monitors

```
int numReaders, numWriters
mutex rwLock
cv waitingReaders, waitingWriters
```

```
readerLock () {
    rwLock.lock()
    while (numWriters>0) {
        waitingReaders.wait(rwlock)
    }
    numReaders++
    rwLock.unlock()
}
```

```
writerLock() {
    rwLock.lock()
    while (numReaders>0 || numWriters>0) {
        waitingWriters.wait(rwLock)
    }
    numWriters++
    rwLock.unlock()
}
```

```
readerUnlock() {
    rwLock.lock()
    numReaders--
    waitingWriters.signal()
    rwLock.unlock()

}
```

```
writerUnlock() {
    rwLock.lock()
    numWriters--
    waitingReaders.broadcast()
    waitingWriters.signal()
    rwLock.unlock()
}
```

Reader-writer lock with monitors

What will happen if a writer finishes and there are several waiting readers and writers?

How long will a writer wait?

How to give priority to a waiting writer?