ECE 570/670

Prof. Irwin

Midterm Exam

- Wednesday, 3/29, 7pm - 9pm
  - Location ILCS 131

Exam Overview

- Mix of conceptual and problem-oriented questions
  - 15-20 short answer questions
  - Closed book/closed notes/no electronic devices
  - Questions on papers are in scope
- Covers up through last lecture
- May need to interpret and/or write pseudocode

Exam Overview

- Process and threads: differences and similarities.
- Atomic actions: what are they and why do we need them.
- Mutexes/Locks: general use and purpose.
- Monitors: general use and purpose.
- Semaphores: general use and purpose.
- Compare and contrast use of monitors versus semaphores.
- Classic synchronization problems covered in class: Bounded buffer (soda machine), got milk, dining philosophers.
- Producer-consumer and reader-writer synchronization problems and solutions.
- Implementing threads and locks: context switching, shared and private state, implementing synchronization primitives, test and set, interrupts.
- Deadlock: what conditions are required for deadlock.
- Mechanisms for deadlock avoidance.
- Scheduling algorithms: FCFS, RR, STCF, STCF-P, EDF, Lottery scheduling
- Tradeoffs between scheduling algorithms

Question #1

- OS design is driven by hardware advances. Name three recent advances that are influencing OS design.

- Multicore systems (due to stalled increases in clock speed), mobile/energy-constrained smartphones, distributed/networked systems, unreliable memory, specialized computing (ASICs, FPGAs, etc.)

Question #2

- What are three key characteristics of an OS process?

- Run independently of each other; don’t know about each other (i.e., isolated); interprocess communication is explicit; execute one or more threads; have their own isolated address space

Question #3

- What is the difference between a process and a thread?

- Process has its own address space. A process is an address space plus one or more threads executing within the address space.

Question #4

- What process state for a running process is not stored in the address space?
- General purpose registers and the program counter(s) and stack pointer(s)

Question #5

- What prevents complete thread independence? Give one specific example of a dependency.

- Sharing hardware and software resources creates dependencies. One thread might have to wait to use a hardware resource, like the CPU or disk I/O.

Question #6

- What are the advantages/disadvantages of designing a concurrent program using threads versus events?

- Events are fast but hard to program because everything is asynchronous and you have to maintain your own state per request. With threads, you can encapsulate state within each thread, and the threading package manages it for you.

Question #7

- Name three "Hints" for computer systems design.

- Keep it simple; Make it fast, rather than general; Use procedure arguments to provide flexibility; Leave it to the client; keep basic interfaces stable; keep a place to stand; Plan to throw one away; divide and conquer; use a good idea again; handle normal and worst case separately; split resources in a fixed way rather than sharing them; use static analysis; cache answers to expensive computations; use hints to speed up normal execution; when in doubt use brute force; compute in the background when possible; use batch processing if possible; use end-to-end error recovery; log updates to record the truth about an object; make actions atomic or restartable

Question #8

- What are the two primary goals of concurrency?

- Ensure correctness, i.e., all inter leavings are correct, while constraining the program as little as possible.

Question #9

- What is the problem with defining large critical sections, such as the one below, in multi-threaded code?

- Results in a lot of waiting by the threads. You want to release the lock as much as possible. Constrains concurrency too much and slows the program down.

Question #10

- Who do you think is “right": the SEDA paper or the “Events are a Bad Idea" paper and why? Justify your answer.

- ?

Question #11

- What is the difference between Mesa and Hoare Monitor semantics? Which is simpler from the perspective of the application programmer? Which is simpler from the perspective of the OS developer? Why?

- With Mesa semantics, you need to recheck the condition after wakeup. With Hoare semantics it is guaranteed to be true. Mesa semantics are more complex for the application programmer, but simpler from the perspective of the OS designer.

Question #12

- Do you need to be holding a lock when you call wait()?

- Yes

Question #13

- With condition variables, can you always use broadcast() instead of signal()?

- Yes

Question #14

- When might you use signal() instead of broadcast() with condition variables?

- For performance reasons, to prevent everyone from waking up at one time and then everyone going back to sleep.

Question #15

- Does the order of the up() calls matter with semaphores in terms of causing/preventing deadlock? Does the order of the down() calls matter?

- The order of the up() calls doesn’t matter. The order of the down() calls does matter.

Question #16

- Name three internal events a thread may execute to return control back to the OS?

- yield(), lock(), Read/write to a disk, or wait().

Question #17

- Name the most common external event that transfers control back to the OS from a running thread?

- Operating system timer interrupt

Question #18

- What are the basic steps involved with creating a new thread?

- Allocate and initialize a new TCB, allocate a new stack, make it look like thread was going to call a function, add thread to ready queue.

Question #19

- Why is it ok for the OS to disable interrupts, but not application programs?

- We can trust the OS to not stall the machine.

Question #20

- How does the test&set(X) function work? Implement a simple busy-waiting lock for multi-processors using the test&set(X) function.

- Atomically sets value equal to 1 but returns original value
- lock () { while (test&set(value) == 1) { }
- unlock () { value = 0 }

Question #21

- What are the options for ensuring atomicity on a uniprocessor within the OS?

- Disabling interrupts, atomic load-store, test&set

Question #22

- What are the four conditions for deadlock? How can the OS prevent deadlock in user programs? Should the OS prevent deadlock in user programs?

- Limited resources, hold-and-wait, no pre-emption, and circular chain of requests. Eliminate one of the conditions. Probably not its job.

Question #23

- Implement producer-consumer using both monitors and semaphores.

With monitors

```
consumer () {
  lock (sodaLock)

  while (numSodas == 0)
{
    wait
(sodaLock,hasSoda)
  }

  take soda out of
machine

  signal (hasRoom)

  unlock (sodaLock)
}
```

```
producer () {
  lock (sodaLock)


while(numSodas==MaxSodas)
{
    wait (sodaLock,
hasRoom)
  }

  add soda to machine

  signal (hasSoda)

  unlock (sodaLock)
}
```

With semaphores

```
Semaphore mutex(?),fullBuffers(?),emptyBuffers(?)
```

```
consumer () {






}
```

```
producer () {






}
```

- Remember to define/initialize semaphores

With semaphores

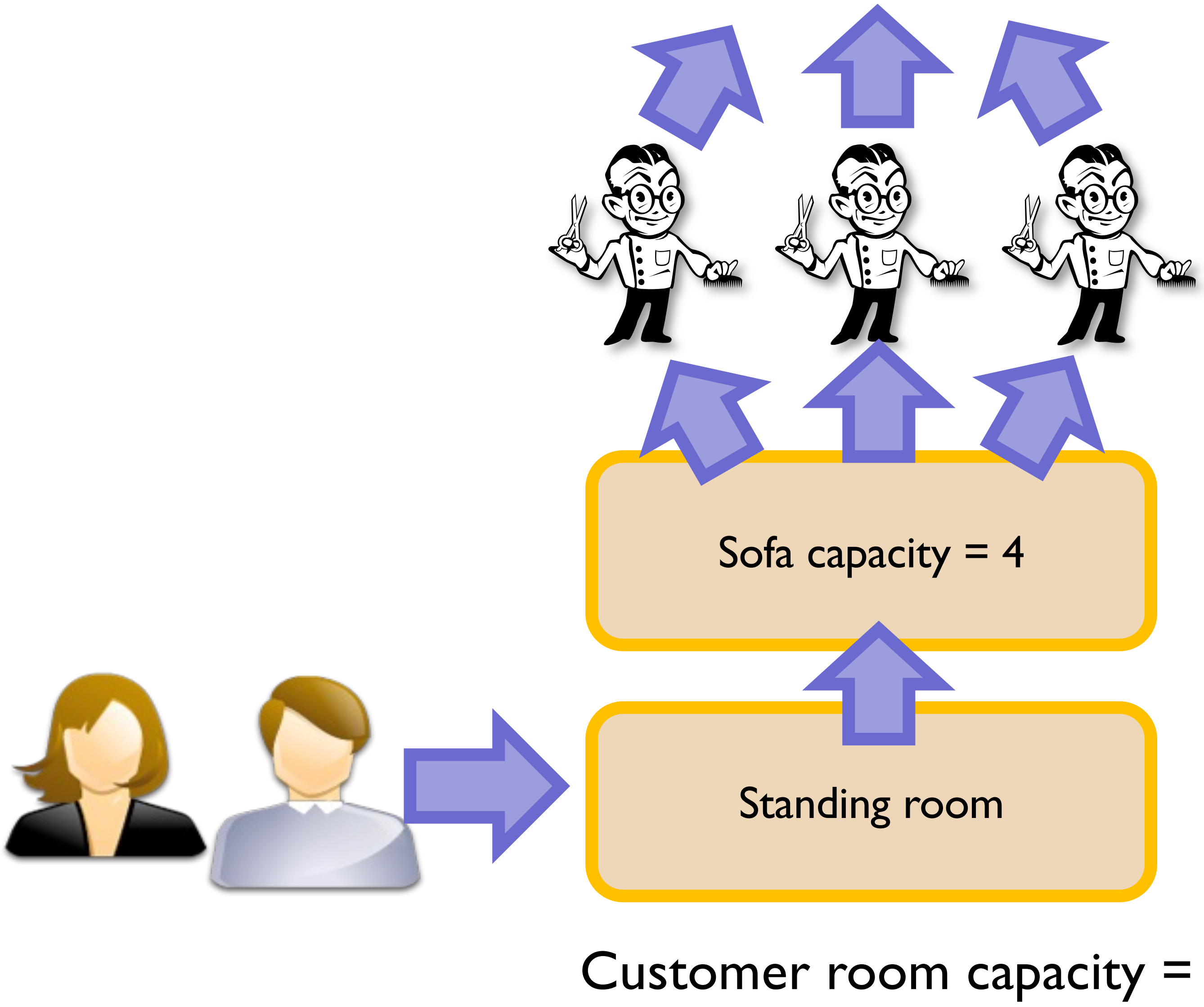
```
Semaphore
mutex(1),fullBuffers(0),emptyBuffers(MaxSodas)
```

```
consumer () {
  down (fullBuffers)

  down (mutex)

  take soda out

  up (mutex)

  up (emptyBuffers)
}
```

```
producer () {
  down (emptyBuffers)

  down (mutex)

  put soda in

  up (mutex)

  up (fullBuffers)
}
```

- Remember to define/initialize semaphores

Question #24



Question #24

- A barbershop consists of a waiting room that has 4 chairs and can hold up to 9 people (standing room only). The barbershop also contains 3 barber chairs, which a customer sits in while one of the 3 barbers cuts their hair. *If there are no customers to be served*, the barber goes to sleep. *If a customer enters the barbershop and there is no space to stand in the waiting room*, then the customer must wait until there is space. *If there is space to stand in the waiting room*, but no chairs, then the customer must wait to sit in a chair. *If there is an open chair*, but the barber chairs are occupied, then the customer must wait until a barber finishes cutting hair and the barber chair becomes free. *If a barber chair is available for a customer*, but the barber is asleep, then the customer should wake up the barber.

- Write pseudocode to synchronize the Customer() thread functions and the Barber() thread functions using both monitors and semaphores. For monitors, define condition variables for all the conditions the Barbers and Customers must wait on. Remember that for every wait() there must be a signal(), and that every wait() should be inside the body of a while loop that rechecks its condition upon exiting the wait() function. For semaphores, first define the semaphores and variables you are using, and their starting values. Barber threads should invoke a function cutHair() when actually cutting a customers hair, and Customer threads should always wait while getting their hair cut.

Barbershop

```
lock;
roomCV;
numInRoom=0;
sofaCV;
numOnSofa=0;
chairCV;
numInChair=0;
customerCV;
cutCV;
```

```
Customer () {
  lock.acquire ();
  while (numInRoom == 9)                  // Enter room
    roomCV.wait (lock);
  numInRoom++;

  while (numOnSofa == 4)                  // Sit on sofa
    sofaCV.wait (lock);
  numOnSofa++;

  while (numInChair == 3)                 // Sit in chair
    chairCV.wait (lock);
  numInChair++;
  numOnSofa--;
  sofaCV.signal (lock);
  customerCV.signal (lock);               // Wake up barber
  cutCV.wait (lock);                      // Wait for cut to finish
  numInChair--;                           // Leave the shop
  chairCV.signal (lock);
  numInRoom--;
  roomCV.signal (lock);
  lock.release ();
}
```

Enter room

Sit on sofa

Sit in chair

Wake up barber

Wait for cut to finish

Leave the shop

Barbershop

```
Barber () {
  lock.acquire ();
  while (1) {
    while (numInChair == 0)
      customerCV.wait (lock);
    cutHair ();
    cutCV.signal ();
  }
  lock.release ();
}
```

```
Customer () {
  lock.acquire ();
  while (numInRoom == 9)
    roomCV.wait (lock);
  numInRoom++;

  while (numOnSofa == 4)
    sofaCV.wait (lock);
  numOnSofa++;

  while (numInChair == 3)
    chairCV.wait (lock);
  numInChair++;
  numOnSofa--;
  sofaCV.signal (lock);
  customerCV.signal (lock);
  cutCV.wait (lock);
  numInChair--;
  chairCV.signal (lock);
  numInRoom--;
  roomCV.signal (lock);
  lock.release ();
}
```

Barbershop with semaphores

```
Semaphore room = 9,
  sofa = 4, chair = 3,
  customer = 0, cut = 0;
```

```
Barber () {
  while (1) {
    customer.down()
    // cut hair
    cut.up ()
  }
}
```

```
Customer () {
  room.down ()
  // enter room
  sofa.down ()
  // sit on sofa
  chair.down ()
  // sit on chair
  sofa.up ()
  customer.up ()
  cut.down ()
  // leave chair
  chair.up ()
  // leave room
  room.up ()
}
```