



Midterm Review

[Syllabus]

- Everything up to and including Lecture 18 (deadlock)
- Main Topics: C, OS, System Calls, Processes, Threads, Scheduling, Synchronization, Classical Synchronization Problems, Deadlock



[Problem 1: Concurrency]

Thread A

```
for (i=0; i<5; i++) {  
    x = x + 1;  
}
```

Thread B

```
for (j=0; j<5; j++) {  
    x = x + 2;  
}
```

■ Assume

- A single-processor system
- Load and store are atomic
- **x** is initialized to 0 before either thread starts
- **x** must be loaded into a register before being incremented (and stored back to memory afterwards)



[Problem 1: Concurrency]

Thread A

```
for (i=0; i<5; i++) {  
    x = x + 1;  
}
```

Thread B

```
for (j=0; j<5; j++) {  
    x = x + 2;  
}
```

- Why is $x \leq 15$ when both threads have completed?
 - If the $x=...$ statements are not interleaved, $x = 15$.
 - If they are ever interleaved, this can only cause a smaller or equal value.

[Problem 2: Concurrency]

Thread A

```
for (i=0; i<5; i++) {  
    x = x + 1;  
}
```

Thread B

```
for (j=0; j<5; j++) {  
    x = x + 2;  
}
```

- Give a concise proof why $x \neq 1$ when both threads have completed
 - Every store into x from either Thread A or B is ≥ 0 , and once x becomes ≥ 0 , it stays ≥ 0 .
 - The only way for $x = 1$ is for the last $x=x+1$ statement in Thread A to load a zero and store a one. However, there are at least four stores from Thread A previous to the load for the last statement, meaning that it couldn't have loaded a zero.

[Problem 3: Concurrency]

Thread A

```
for (i=0; i<5; i++) {  
    x = x + 1;  
}
```

Thread B

```
for (j=0; j<5; j++) {  
    x = x + 2; atomicIncr2(x);  
}
```

- Suppose we replace ' $x = x + 2$ ' in Thread B with an atomic double increment operation `atomicIncr2(x)` that cannot be preempted while being executed. What are all the possible final values of x ? Explain.

[Problem 3: Concurrency]

Thread A

```
for (i=0; i<5; i++) {  
    x = x + 1;  
}
```

Thread B

```
for (j=0; j<5; j++) {  
    atomicIncr2(x);  
}
```

- What are all the possible final values of **x**? Explain.
 - Final values are 5, 7, 9, 11, 13, or 15.
 - The **x=x+2** statements can be “erased” by being between the load and store of an **x=x+1** statement.
 - However, since the **x=x+2** statements are atomic, the **x=x+1** statements can never be “erased” because the load and store phases of **x=x+2** cannot be separated.
 - The final value is at least 5 (from Thread A) with from 0 to 5 successful updates of **x=x+2**.



[Problem 4: Context Switching]

- What needs to be saved and restored on a context switch between two threads in the same process? What if the two threads are in different processes?
 - Save the thread's registers, stack pointer, and program counter in the TCB of the swapped out thread
 - Reload the same things from the TCB of the new thread
 - Different Processes:
 - Save and restore above
 - Load the pointer for the address space of the new process
 - Don't need to save the old pointer, since this will not change and is already stored in the PCB.



Problem 5: Threading Pros and Cons

- Under what circumstances can a multithreaded program complete more quickly than a non-multithreaded program? Keep in mind that multithreading has context-switch overhead associated with it.
 - When there is a lot of blocking that may occur (such as for I/O) and parts of the program can still make progress while other parts are blocked.
 - When there are multiple cores/processors.



[Problem 6: Bounded Buffer]

- Given the three following variations, explain whether it is correct or incorrect.
 - If correct, explain any of the advantages and disadvantages of the new code.
 - If incorrect, explain what could go wrong (i.e., trace through an example where it does not behave properly).



[Problem 6a: Bounded Buffer]

```
Producer () {  
    mutex_lock(m);  
    1 → sem_wait(emptyBuffers);  
    put 1 coke in machine;  
    sem_signal(fullBuffers);  
    mutex_unlock(m);  
}
```

```
Consumer () {  
    2 → mutex_lock(m);  
    sem_wait(fullBuffers);  
    take 1 coke from machine;  
    sem_signal(emptyBuffers);  
    mutex_unlock(m);  
}
```

- Suppose coke machine is initially full.
- Suppose a Producer comes and grabs mutex and then waits for emptyBuffers.
- A consumer then waits on mutex -- **deadlock!**
- Is there a different way to deadlock?
- Yes: Consumer grabs mutex and waits for fullBuffers.

[Problem 6b: Bounded Buffer]

```
Producer () {  
    mutex_lock(m);  
    1 → sem_wait(emptyBuffers);  
    put 1 coke in machine;  
    sem_signal(fullBuffers);  
    mutex_unlock(m);  
}
```

```
Consumer () {  
    sem_wait(fullBuffers);  
    2 → mutex_lock(m);  
    take 1 coke from machine;  
    sem_signal(emptyBuffers);  
    mutex_unlock(m);  
}
```

- This code is incorrect - it can lead to **deadlock**.
 - The problem is exactly as the previous example.
 - Consider the case where the coke machine is initially full.
 - Suppose a Producer comes and grabs **m** and then waits for **emptyBuffers**.
 - A consumer then hangs on **mutex** and no one will ever consume a coke to empty a buffer.

[Problem 6c: Bounded Buffer]

```
Producer () {  
    sem_wait(emptyBuffers);  
    mutex_lock(m);  
    put 1 coke in machine;  
    sem_signal(fullBuffers);  
    mutex_unlock(m);  
}
```

```
Consumer () {  
    sem_wait(fullBuffers);  
    mutex_lock(m);  
    take 1 coke from machine;  
    sem_signal(emptyBuffers);  
    mutex_unlock(m);  
}
```

- This code is correct.
 - This code allows more concurrency
 - Since **mutex** immediately surrounds both the Producer and Consumer actions of putting a coke and taking a code from the machine, releasing **mutex** quickly achieves better concurrency



[Problem 7]

- Most round-robin schedulers use a fixed size quantum. Give an argument in favor of a small quantum, and another in favor of a larger one.
- Compare the contrast the types of systems and jobs to which the arguments apply.



[Problem 8]

- Classify the relation between each of the following pairs into either “if one increases, the other always increases” or “if one decreases, the other always increases” or “no such relation exists always.”
 1. Throughput and Waiting Time.
 2. Waiting Time and Turnaround Time.
 3. Response time and Turnaround Time.
 4. Throughout and Turnaround Time.



[Problem 9]

- Suppose there are two types of dining philosophers. One type always picks up his left fork first and the other type always picks up his right fork first – call these a lefty and a righty. Each type executes consecutive “wait”s on their forks (left followed by right for lefties, and the other way around for righties), eats, then does “signal”s on the forks in reverse order of the waits (right followed by left for lefties, and the other way around for the righties).



[Problem 9 (continued)]

- Does every seating arrangement of lefties and righties with at least one of each avoid deadlock? Why?
- Does it prevent starvation? Why?



[Problem 10]

- Some Linux atomic operations do not involve two accesses to a variable, such as `atomic_read(atomic_t *v)`. A simple read operation is obviously atomic in any architecture. Therefore, why is this operation added to the repertoire of atomic operations?
 - It's not simple. Reading one variable may involve multiple reads in the physical memory.



[Problem 11]

You are designing a data structure for efficient dictionary lookup in a multithreaded application. The design uses a hash table that consists of an array of pointers each corresponding to a hash bin. The array has 1001 elements, and a hash function takes an item to be searched and computes an entry between 0 and 1000. The pointer at the computed entry is either null, in which case the item is not found, or it points to a doubly linked list of items that you would search sequentially to see if any of them matches the item you are searching for. There are three functions defined on the hash table: Insertion (if an item is not there already), Lookup (to see if an item is there), and deletion (to remove an item from the table). Considering the need for synchronization, would you:

1. Use a mutex over the entire table?
2. Use a mutex over each hash bin?
3. Use a mutex over each hash bin and a mutex over each element in the doubly linked list?



[Problem 12]

You are asked to implement a one-shot barrier for N threads. Each thread executes code of the following manner:

```
Barrier *b;
```

```
...
```

```
Arriveatbarrier(b);
```

Basically each of the first $N-1$ thread to call `Arriveatbarrier(b)` will block until the last (N th) thread has arrived there. Once that happens, all threads will be released simultaneously to proceed.

One-shot means this code will be executed exactly once by each thread.

1. Write an implementation using semaphores and mutexes.
2. Does your solution work if the barrier is reused multiple times by threads (i.e., it is not a one-shot barrier)?



[Problem 13]

- Implement semaphore wait and semaphore signal functions using the test and set lock primitive.



[Problem 14]

- You are given an XCHG (exchange) hardware instruction that atomically exchanges two memory addresses. Implement a test and set lock using the XCHG primitive.



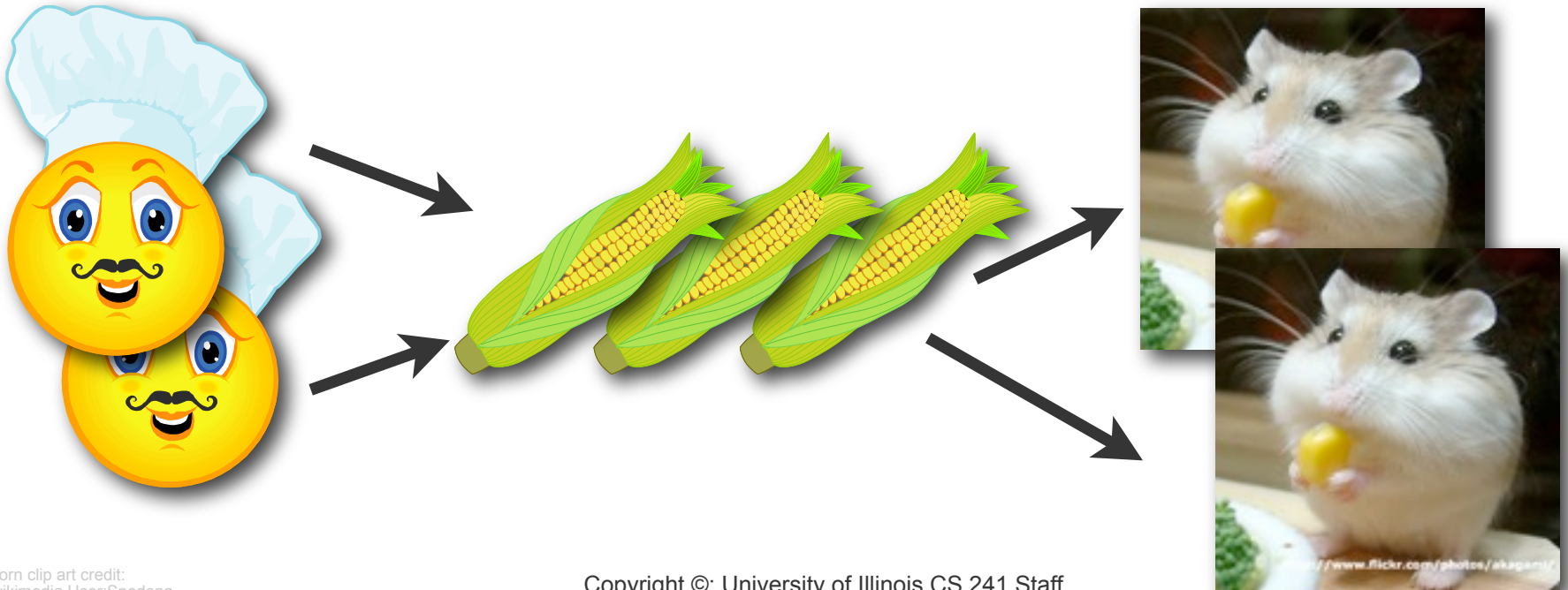
Problem 15: producer-consumer without semaphores

- Let's solve it with mutexes + condition variables instead!
- First, what was the semaphore-based solution we saw?



Recall Producer-Consumer

- Chefs cook items and put them on a conveyer belt
- Customers pick items off the belt



[Solution



- Prevent overflow: block producer when full!
Counting semaphore to count #free slots
 - 0 → block producer
- Prevent underflow: block consumer when empty!
Counting semaphore to count #items in buffer
 - 0 → block consumer
- Mutex to protect accesses to shared buffer & pointers.

[Pseudocode getItem()]

- For consumer
- *Error checking/EINTR handling not shown*

```
sem_wait(items);  
mutex_lock(mutex);  
result = buffer[ removePtr ];  
removePtr = (removePtr + 1) % N;  
mutex_unlock(mutex);  
sem_signal(slots);
```

[Pseudocode putItem(*data*)]

- For producer
- *Error checking/EINTR handling not shown*

```
sem_wait(slots);  
mutex_lock(mutex);  
buffer[ insertPtr ] = data;  
insertPtr = (insertPtr + 1) % N;  
mutex_unlock(mutex);  
sem_signal(items);
```

[And now, without semaphores]

Shared:

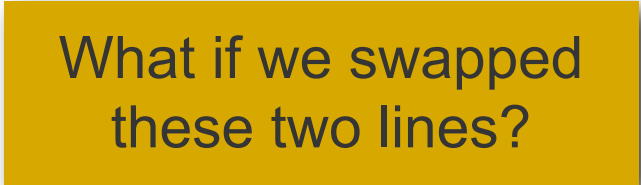
```
int buf_size;
element_t buffer[n];
cond_t not_full;
cond_t not_empty;
mutex m;
```

Reader:

```
while (1) {
    lock(m);
    while (buf_size == 0)
        cond_wait(not_empty, m);
    grab_element(buffer);
    buf_size--;
    cond_signal(not_full);
    unlock(m);
}
```

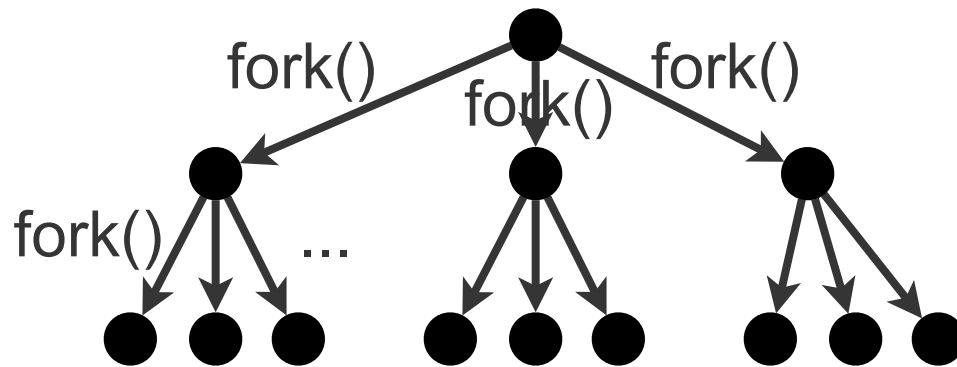
Writer:

```
while (1) {
    lock(m);
    while (buf_size == n)
        cond_wait(not_full, m);
    put_element(buffer);
    buf_size++;
    cond_signal(not_empty);
    unlock(m);
}
```



What if we swapped these two lines?

Homework 2 Question 2: Balanced ternary tree



[A solution]

C

P

```
create_ternary_tree_of_depth(int N) {  
  int i;  
  for (i = 1; i < N; i++)  
    if (fork())  
      if (fork())  
        if (fork())  
          break;  
}
```

