Review Session: Threads, Synchronization, and File I/O

CS 162

September 25, 2019

Contents

1 Threads, Processes

1.1 Process vs Thread

How is a process different from a thread?

1.2 Context Switch

What state information do you need to save/restore about threads when performing a context switch?

1.3 Overuse of Threads

List two reasons why overuse of threads is bad (i.e., using too many threads for different tasks). Be explicit in your answers.

1.4 Critical Section

Suppose a thread is running in a critical section of code, meaning that it has acquired the locks through proper arbitration. Can it get context switched? Why or why not?

1.5 Mode Switch

Name three ways in which the processor can transition from user mode to kernel mode. Can the user execute arbitrary code after transitioning?

1.6 Threads

What are the possible outputs of this program?

```
void* threadfun(void* arg) {
    print("hello\n");
}

int main(int argc, char** argv) {
    pthread_t t;
    pthread_create(&t, NULL, threadfun, NULL);
    sleep(1);
    printf("world\n");
}
```

2 Synchronization

2.1 Locking via Disabling Interrupts

Consider the following implementation of Locks:

```
Lock::Acquire() {
    disable_interrupts();
}
```

```
Lock::Release() {
    enable_interrupts();
}
```

1. For a single-processor system state whether this implementation is incorrect.

2. For a multiprocessor system, explain what additional reason(s) might make this implementation incorrect?

2.2 Counting Semaphores

We have a limited number of resources such that only N threads can use them at any given time. Explain how we can use a "counting semaphore" to control access to these resources.

2.3 Bounded Buffer with Locks and Semaphores

In class we discussed a solution to the bounded-buffer problem for multiple producers and multiple consumers using semaphores and locks. In this problem, there is a fixed size buffer. Producers add to the buffer, but only if there is room in the buffer. Consumers remove from the buffer, but only if there are items. Producers should sleep until there is space in the buffer and consumers should sleep until there is something in the buffer.

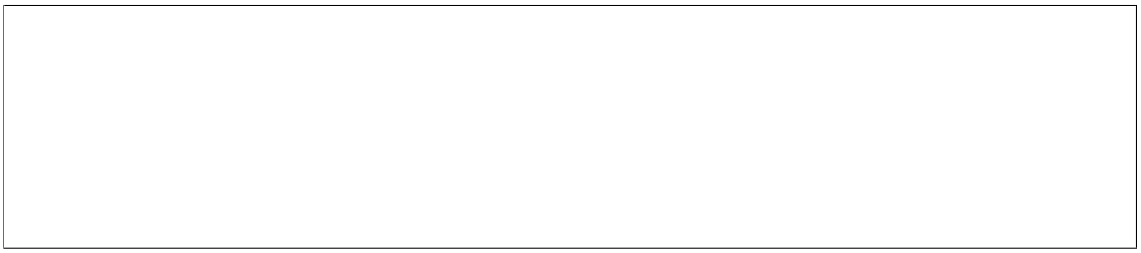
1. Reproduce this solution from lecture. Think about how we can use semaphores track how many empty and full spots are currently in the buffer. Do not go onto the next part until you are done with this, as it will give away the answer.

```
Producer () {
  ------------------------------
  ------------------------------
  queue.Enqueue()
  ------------------------------
  ------------------------------
}
```

```
Consumer () {
  ------------------------------
  ------------------------------
  queue.Dequeue()
  ------------------------------
  ------------------------------
}
```

2. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  mutex.Acquire()                  mutex.Acquire()
  emptyBuffers.P()                 fullBuffers.P()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  emptyBuffers.V()
  mutex.Release()                  mutex.Release()
}                                }
```



3. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  mutex.Acquire()                  fullBuffers.P()
  emptyBuffers.P()                 mutex.Acquire()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  mutex.Release()
  mutex.Release()                  emptyBuffers.V()
}                                }
```

4. Examine the following implementation where the lock operations have been moved. Explain it is correct. If not, explain why.

```
Producer () {                    Consumer () {
  emptyBuffers.P()                 fullBuffers.P()
  mutex.Acquire()                  mutex.Acquire()
  queue.Enqueue()                  queue.Dequeue()
  fullBuffers.V()                  emptyBuffers.V()
  mutex.Release()                  mutex.Release()
}                                }
```

3 File I/O

3.1 Benefits of Concurrency

On a single processor machine, 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.

3.2 Storing Integers

You are working for BigStore and your boss has tasked you with writing a function that takes an array of ints and writes it to a specified file for later use. He also wants you to use file descriptors (no `fopen`, etc.). Fill in the following function:

```
void write_to_file(const char *file, int *a, int size) {

  int write_fd = open(__________________, __________________);

  char *write_buf = __________________

  int buf_size = __________________
  int bytes_written = 0;
  // Write a to file.

  ____________________________________________________

  ____________________________________________________

  ____________________________________________________
  close(write_fd);
}
```

Now, write the function that retrieves previously saved integers and places them in a int array.

```
void read_from_file(const char *file, int *a, int size) {

  int read_fd = open(__________________, __________________);
```

```
  char *read_buf = ___________________

  int buf_size = ___________________
  // Read a from a file.

  ____________________________________________________

  ____________________________________________________

  ____________________________________________________

  ____________________________________________________

  ____________________________________________________
  close(read_fd);
}
```

Your coworker opens up one of the files that you used to store ints on his text editor and complains its full of junk! Explain to him why this might be the case.