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?

- A thread is an independent execution context, with its own registers, program counter, and stack. A thread is defined by an OS object called the Thread Control Block (TCB) which stores all this information.
- A process is comprised of one or more threads and enjoys OS level isolation from other processes. For example, it has its own address space. The process is defined by an OS object called the Process Control Block which stores process level information like a pointer to a page table, a list of open files, and process metadata.
- Different threads in the same process may each have their own stacks, but all share the process' address space and so can access each other's memory. Threads encapsulate concurrency while processes encapsulate isolation.

1.2 Context Switch

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

If the threads belong to the same process: CPU registers, Program Counter, Stack Pointer If they belong to different processes: The same as above, plus the Page Table Base Register

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.

- "Threads are cheap, but they're not free." Too many threads will cause a system to spend lots of time context switching and not doing useful work. Each thread requires memory for stack space and TCBs. Too many threads and these memory uses will dominate overall memory use.
- Having many threads makes synchronization more complicated as you have more and more simultaneous tasks. Also, debugging is more difficult due to non-deterministic and non-reproducible execution.

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?

Yes it can get context switched. Locks (especially user-level locks) are independent of the scheduler. (Note that threads running in the kernel with interrupts disabled would not get context-switched preemptively.)

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?

- The user program can execute a trap instruction (for a system call)
- The user program can perform a synchronous exception (bad address, bad instruction, etc)
- The processor transitions into kernel mode when responding to an interrupt.

The user cannot execute arbitrary code because entry to kernel mode is through a restricted set of routines in the kernel – not in the user's program.

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");
}
```

Three possibilities:

```
hello
world

world
hello

world
```

2 Synchronization

2.1 Locking via Disabling Interrupts

Consider the following implementation of Locks:

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

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

   - Does not work for multiple locks
   - Process holding lock may not release for a long time, effectively halting the machine. Can deadlock if program holds the lock and waits for some I/O, which requires an interrupt.
   - Does not maintain the "acquired" state of the lock across a context switch (`yield()`).

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

Does not block other processors from accessing the critical section.

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.

- Set the initial value of the semaphore to N (number of concurrent accesses allowed).
- P() decrements the semaphore's counter and either causes the process to wait until the resource is available or allocates the process the resource.
- V() increments the semaphore's counter, releasing a waiting process (if any is waiting).

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

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()
}                                }
```

This code is incorrect. It can lead to deadlock. What is happening here is the producers and consumers can go to sleep while holding the lock, preventing others from even touching the queue. Consider the case where the buffer is full. Suppose a Producer comes and grabs the mutex and then waits for emptyBuffers. A consumer then hangs on mutex and no one will ever dequeue to empty the buffer.

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()
}                                   }
```

This code is incorrect. It can lead to deadlock. The problem is exactly as the first case of deadlock mentioned in part a. The procedure can wait on the queue and sleep with the lock, but the consumer will never get to emptyBuffers.V() since that requires having held the lock. The lesson here is that a thread should never sleep while holding a shared lock.

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()
}                                   }
```

This code is correct. It *may* have reduced concurrency, as you incur the possibility of waking up a consumer with fullBuffers.V() only for it to sleep on the lock.

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.

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.

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(file, O_WRONLY);

  char *write_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);
  int bytes_written = 0;

  while (bytes_written < buf_size) {
    bytes_written += write(write_fd, &write_buf[bytes_written], buf_size - bytes_written);
  }
  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(file, O_RDONLY);

  char *read_buf = (char *) &a[0];
  int buf_size = size * sizeof(int);

  int bytes_read = 0;
  int total_read = 0;
  while ((bytes_read = read(read_fd, &read_buf[total_read], buf_size - total_read)) > 0) {
    total_read += bytes_read;
  }
  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.

```
Currently, we are reading and writing the contents of memory directly to disk.
This is convinient for us, because we do not have to any parsing of the input.
However, the memory representation of an int array is unlikely to be human readable.
```