EECE.4810/EECE.5730: Operating Systems

Spring 2017

Homework 2

Due **3:15 PM**, **Wednesday, 2/15/17**

Notes:

- These problems were taken or adapted from Chapters 4 & 5 of our course text (Silberschatz, et al, *Operating Systems Concepts*, 9th edition)
- While typed solutions are preferred, handwritten solutions are acceptable.
- Any electronic submission must be in a single file. Archive files will not be accepted.
  - As noted in the syllabus, you will lose 10 points if you fail to follow this rule.
- Electronic submissions should be e-mailed to Dr. Geiger at Michael_Geiger@uml.edu. Please include your name as part of your filename (for example, mgeiger_hw2.pdf).
- EECE.4810 students must complete problems 1-4, for a total of 50 points.
- EECE.5730 students must complete problems 1-6, for a total of 75 points.

1. (15 points) A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application (e.g., a program that spends most of its time on computation, not I/O or memory accesses) is running on this system. All input is performed at program start-up, when a single file must be opened and read sequentially. Similarly, all output is performed just before the program terminates, when the program results must be written sequentially to a single file. Between startup and termination, the program is entirely CPU-bound (e.g., only executing instructions). Your task is to improve the performance of this application by multithreading it. Specifically, you must determine:

   - (7 points) How many threads will you create to handle input and output? Briefly explain.
   - (8 points) How many threads will you create for the CPU-bound portion of the application? Briefly explain.

2. (10 points) Consider the code segment below. The `thread_create()` function starts a new thread in the calling process. (For the purpose of this problem, you can ignore the lack of arguments to that function.) How many unique processes are created? How many unique threads are created?

```
pid_t pid;

pid = fork();
if (pid == 0) {                    // Child process
   fork();
   thread_create( . . . );
}
fork();
```

3. (15 points) Consider the code example for allocating and releasing processes shown below. Note that the functions are not complete—comments show where detailed code would be required to allocate and release process resources. This code would be part of the kernel.

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* An implementation of fork() would call this function */
int allocate_process() {
   int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {

      // Code to allocate process resources and assign new_pid

      ++number_of_processes;
      return new_pid;
   }
}

/* An implementation of exit() would call this function */
void release_process() {

   // Code to release process resources

   --number_of_processes;
}
```

a. (7 points) Identify the race condition(s) in these two functions, assuming each function could be called by one (or more) kernel threads.. (In other words, determine what operation(s) involving shared data depend on the timing or ordering of different threads.) Give an example of the race condition causing incorrect results to support your answer.

b. (8 points) Given a lock, `mutex`, with operations `lock()` and `unlock()`, indicate where these operations would have to be used to prevent the race condition(s).

4. (10 points) Servers are often designed to limit the maximum number of connections, accepting up to *N* connections but forcing other requests to wait if that maximum number has been reached. Explain how semaphores can be used to limit the number of simultaneous connections.

5. (15 points, *EECE.5730 only*) The first known correct software solution to the critical-section problem for two processes was developed by the Dutch mathematician Theodorus Dekker. The processes share the following variables:

```
boolean flag[2];       // Both initially false
int turn;
```

The structure of process P_i (i == 0 or 1) is shown below. The other process is P_j (j == 1 or 0). Prove the algorithm shown below satisfies the following requirements for a critical section:

1) Mutual exclusion: At most one process can execute its critical section at a time.
2) Progress: If multiple processes attempt to enter their critical sections at the same time, one process is guaranteed to be selected and move forward.
3) Bounded waiting: Once a process P has requested access to its critical section, there exists a bound on the number of times other processes will be allowed to access their critical sections before P is given access to its critical section.

```
do {
   flag[i] = true;

   while (flag[j] == true) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ;       // do nothing
         flag[i] = true;
      }
   }

      /* critical section would be placed here */

   turn = j;
   flag[i] = false;

} while (true);
```

6. (10 points, *EECE.5730 only*) Multithreading does not necessarily guarantee improved performance over single-threaded versions of the same program. Provide one example of a type of program that would run faster with multiple threads than it would with one thread, and one example of a type of program that would not run faster with multiple threads.