# EECE.4810/EECE.5730: Operating Systems

Spring 2017

Homework 2 Solution

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.*

**Solution:** Since the file must be accessed sequentially, the work of reading and writing it cannot reasonably be parallelized, and only a single thread should be used for each of these tasks.

   - *(8 points) How many threads will you create for the CPU-bound portion of the application? Briefly explain.*

**Solution:** The CPU-bound portion should be divided evenly among the four processors, so four threads should be created for this part of the program. Fewer than four would waste processor resources, while more threads would be unable to run simultaneously.

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;

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

**Solution:** According to the textbook solution to this problem, this code segment creates a total of six processes (including the original one) and two threads, as described below:

- The initial call to `fork()` (labeled "1" above) creates a copy of the original process (2 processes at this point)
- The second call to `fork()` (labeled "2" above) is executed only by the child process from the first `fork()` call. (3 processes at this point)
- There are now two processes executing the code inside the if statement, meaning both of those processes call `thread_create()` (3 processes, 2 threads)
    - What should have been made clear in the original assignment is that each newly created thread starts a different function than the one currently executing—one of the arguments to a `thread_create()` function is a pointer to the function to begin executing. The threads therefore don't move on to the last `fork()` call.
- All three processes execute the final call to `fork()` (labeled "4" above), so each process copies itself at that point (6 processes, 2 threads)

However, Peter brought to my attention the fact that the above solution does not account for the fact that each process starts as a single thread of execution. A better solution would be to say that this code segment creates a total of six processes and eight threads (two threads created by calls to `thread_create()` and six threads corresponding to the six single-threaded processes).

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.*

**Solution:** Accesses to `number_of_processes` constitute a race condition. The increment and decrement operations shown are actually three separate operations: load the shared variable, modify it, and store it back in memory. Interleaving these operations produces wrong results.

**Solution to 4a (continued):** For example: say `number_of_processes` initially equals 4. One process (P1) calls `allocate_process()`, while another (P2) calls `release_process()`. The variable should therefore be 4 when both functions are complete—but say the following happens:

- P1 reads 4
- P2 reads 4
- P1 increments `number_of_processes`: 4 → 5
- P2 decrements `number_of_processes`: 4 → 3

At this point, it doesn't matter which process writes its value of `number_of_processes` back to memory—the value will be wrong.

*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).*

**Solution:** Each function should start with a call to `lock()` and end with a call to `unlock()` (prior to returning).

*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.*

**Solution:** The server can use a single semaphore initialized to the maximum number of allowed connections. The server will call `down()` on the semaphore for each connection request, decrementing the semaphore. Each time a connection is released, the server can call `up()` on the semaphore, incrementing it. Once the maximum number of connections has been reached, all requests will trigger a call to `down()` that will block until a connection is released and the `up()` function is called.

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);
```

**Solution:** The algorithm satisfies the three conditions as follows:

- Mutual exclusion: The `flag` and `turn` variables satisfy this condition. If both processes set their `flag` variable to true, only the process whose turn it is will move forward. The other process will be forced to wait until the first process updates `turn`.
- Progress: The mechanism for allowing progress is as described above—`turn` chooses the process to move forward if both attempt to enter the critical section simultaneously.
- Bounded waiting: Since one process sets `turn` to the value of the other process, it ensures that the waiting process will be allowed to enter its critical section next.

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.*

**Solution:** Whether or not multiple threads improve performance depends largely on whether the work in a program can be parallelized. Programs that perform the same operation across a large set of data (for example, matrix arithmetic) can be easily parallelized and therefore will perform better with multiple threads. Programs that are largely sequential (for example, a program reading the contents of a file in order) would not run faster with multiple threads.