ECE 254 F15 Midterm Solutions

J. Zarnett & C. Moreno

November 5, 2015

(1A)

In deferred cancellation, a thread is notified that it is cancelled and the thread is responsible for checking if it has been cancelled and ceasing what it is doing. In asynchronous cancellation, the thread that is to be cancelled is immediately terminated.

(1B)

The fork bomb is when a malicious program repeatedly calls `fork()` to overwhelm the system by spawning so many processes. Two strategies to prevent or mitigate it are limiting the number of processes a user can spawn, or limiting the rate at which processes may be spawned.

(1C)

There is no one-to-many model because it makes no sense to have multiple kernel threads waiting around to serve a single user thread (which can be associated with at most one kernel thread at a time).

(1D)

Running task: $\frac{8 \times 4}{320} = 10\%$

Non-Running task: $\frac{16 \times 4}{320} = 20\%$

(1E)

Small error in the question: where it says `fork`, `read`, and `pipe` could all fail, instead of `read` it should actually be `malloc`. Oops. No deduction if you did not check for `malloc` failure and lines checking if `read` failed are ignored in marking.

```
int fd[2];
pid_t pid;


int main( void ) {

  int fileSize =  fileLength( PRINT_FILE ); /* Can also be declared elsewhere */
  int pipeReturn = pipe( fd );

  if ( pipeReturn < 0 ) {/* Pipe Failed */
      return -1;
  }

  pid = fork();
  if ( pid < 0 ) {/* Fork failed */
       return -1;
   }

  if (pid == 0) { /* Child Process */
      close( fd[0] ); /* Close read end */

      void* readBuffer = malloc( fileSize );
      if ( readBuffer == 0 ) { /* Checking for == NULL also okay */
          return -1;
      }
      open( PRINT_FILE );
      read( PRINT_FILE, readBuffer, fileSize );
      close( PRINT_FILE );

      write( fd[1], readBuffer, fileSize ); /* Write into the pipe */
      free( readBuffer );

      close( fd[1] ); /* Close write end */

  } else { /* Parent Process - Can also compare pid > 0 but not necessary here */

      close( fd[1] ); /* Close write end */

      void* pipeBuffer = malloc( fileSize );
      if ( pipeBuffer == 0 ) {
          return -1;
      }
      read( fd[0], pipeBuffer, fileSize ); /* Read from the pipe */
      print( pipeBuffer );

      free( pipeBuffer );
      close( fd[0] ); /* Close read end */
  }

  return 0;
}
```

(2A)

No, this does not work. One possible solution: `lock++;` statement is not atomic so a thread switch can take place in between its steps. Remember that an increment statement requires reading the variable, increasing its value by 1, and writing the changed value back. So imagine this order:

1. Thread A gets to the `lock++` statement.
2. Thread A reads 0.
3. Thread A increments `lock` (value now 1).
4. Untimely thread switch: Thread B gets chosen to run.
5. Thread B gets to the `lock++` statement.
6. Thread B reads 0.
7. Thread B increments `lock` (value now 1).
8. Thread B writes 1 to `lock`.
9. Thread B evaluates the if statement and enters the critical section.
10. Untimely thread switch back to A.
11. Thread A writes 1 to `lock`.
12. Thread A evaluates the if statement and enters the critical section.
13. Threads A and B are now in the critical section!

(2B)

This does not work. The semaphore's `wait` and `post/signal` *operations* need to be atomic, and not just the update to the internal counter.

There are many examples illustrating the flaw — one of them is as follows:

1. At a certain point in time, the internal state (counter value) of semaphore S is 1.
2. Task A calls wait on semaphore S.
3. "Simultaneously", Task B calls wait on semaphore S. Only one of them should block.
4. Task A's `wait` function succeeds in locking the mutex, and decrements S's counter
5. As soon as Task A's execution of `wait` releases the mutex, Task B wakes up (with the mutex locked), and decrements the counter before Task A had time to execute any further.
6. When Task A's execution of `wait` resumes, it will check the value of counter, and because it is now -1, it will block.
7. Task B's execution of `wait` resumes, and it will also block because the counter is -1.

(2C)

Writer

```
 1. wait( writeMutex )
 2. writers++
 3. if writers == 1
 4.     wait( noReaders )
 5. end if
 6. signal( writeMutex )
 7. wait ( noWriters )
 8. [write data]
 9. signal( noWriters )
10. wait( writeMutex )
11. writers--
12. if writers == 0
13.    signal( noReaders )
14. end if
15. signal( writeMutex )
```

Reader

```
 1. wait( noReaders )
 2. wait( readMutex )
 3. readers++
 4. if readers == 1
 5.    wait( noWriters )
 6. end if
 7. signal( readMutex )
 8. signal( noReaders )
 9. [read data]
10. wait( readMutex )
11. readers--
12. if readers == 0
13.     signal( noWriters )
14. end if
15. signal( readMutex )
```

(2D)

- Timer interrupts.

  This mechanism is essential for time-slicing / time sharing operation of the system, as it constitutes the main mechanism used for preemption. It is also relevant for programmers, since it is related to the aspect that a task switch causing "unpredictable" interleavings can indeed occur at any point during the execution of a task/thread.

- Multicore architectures.

  Multicore architectures provide true parallelism (true simultaneous execution of multiple tasks). This is relevant, for example, to understand that atomicity cannot be achieved by disabling interrupts (since interleaving by switching among tasks when interrupts occur is not the only source of race conditions when we have multicores).

- Assembly level atomic instructions.

  These provide a robust mechanism for the operating system to achieve atomicity (for example, to implement mutexes — mutexes are the very tools that we use to achive atomicity, but their operations need to be atomic; hardware-provided atomic operations solve the "chicken-or-the-egg" sort of situation that arises in this case).

(2E)

The memory allocation function `malloc` is both nondeterministic and it has the potential to block. Making a blocking call inside a critical section can get the whole system stuck in the same way as having nested `wait` semaphore calls.

(3A)

1. Mutual Exclusion
2. Hold and Wait
3. No Pre-emption
4. A cycle in the resource allocation graph

(3B)

1. If a process has been running for a short period of time, less work has to be repeated. So killing it is “less expensive” than killing and older process.

2. There is a risk if we kill “old” processes that no process will ever finish, because as soon as a process gets to be the oldest, it gets killed.

(3C)

We can get livelock with 2PL if every process attempts to get resources, none gets all of them, they each release their resources, and then try again. Example: dining philosopher’s problem. Suppose every philosopher picks up the left chopstick and fails to get the right chopstick. They will all then put down the left chopstick and try again from the beginning. The next time the philosophers attempt, each picks up the left chopstick and fails to get the right chopstick, so they decide to put them down and put them down and start again... and this cycle continues indefinitely.