ECE 254 F18 Midterm Solutions

J. Zarnett

October 25, 2018

(1.1)

```
int main( int argc, char** argv ) {
  int course = atoi( argv[1] );
  int fd = open( "courses.dat", O_RDONLY );
  char* buffer = malloc( sizeof( ece_course ) );
  bool printed = false;

  int bytes_read = read( fd, buffer, sizeof( ece_course ) )
  while( bytes_read != 0 ) {
     ece_course* c = (ece_course*) buffer; /* Interpret buffer as ece_course struct */
     if ( c->number == course ) {
       printf( "AUs: %g.\n", c->au );
       printed = true;
       break;
     }

    bytes_read = read( fd, buffer, sizeof( ece_course ) );
  }

  close( fd );
  free( buffer );

  if ( !printed ) {
    printf( "No data found for course ECE %d.\n", course );
    return -1;
  }

  return 0;
}
```

(1.2) When `fork()` is called, the child process is created and the file descriptors representing both ends of the pipe are open in both the parent and the child. Thus, to completely close the pipe both parent and child need to close both ends.

(1.3)

In the OS_TCB for that task (stored in the os_active_TCB global variable), the state variable is updated to the new state. The current state/value of the CPU registers are stored on the top of the stack (the top of the stack, OS_TCB.tsk_stack, is updated to point to the top of these registers).

Notes: The exact list of registers is required for the lab-answers, but memorizing which ones is not necessary: The Status Register (PSR), Program Counter (R15), and R0-R12 and LR (R14) are stored on the task stack. The student may reference os_rdy/os_dly which are lists of ready and delayed tasks, but this is talked about in Lab 2.

The `MAGIC_WORD` represents the top of the stack. If this has been overwritten, it indicates a stack overflow error.

(1.4)

```
void* worker( void* arg ) {
 pthread_t *watchdog_thread = (pthread_t*) arg;
  /* Might get stuck! */
  result_t * res = long_running_task( );
  pthread_cancel( *watchdog_thread );
  pthread_exit( res );
}
```

```
void* watchdog( void* arg ) {
  pthread_t * worker_thread = (pthread_t*) arg;
  sleep( 30 );
  pthread_testcancel();
  /* If we got here we are not cancelled */
  pthread_cancel( *worker_thread );
  pthread_exit( NULL );
}
```

(2.1)

1. No, it does not work. Between the time of calling `sem_check` and then calling `sem_wait` the value of the semaphore could change, so you might have read a value that says your thread will not get blocked, but then it does get blocked when `sem_wait` is called.

2. No. It is possible to send a signal from one thread to another in UNIX like posting on a semaphore, but it is not possible to wait on a signal. You may register a signal handler in your program but you don't wait for it.

(You could also argue yes: If statement A1 happens before the signal is sent and then statement B2 happens in the signal handler, meaning it doesn't run until the signal arrives, that also is possible. In real life it's limited because there are some things you cannot do in signal handlers, but that's not something we talked about).

3.

```
void foo( void* arg ) {
  free( arg );
}
```

```
void* bar( void * argument ) {
  task* t = malloc( sizeof( task ) );
  pthread_cleanup_push( foo, t );
  /* Do something useful */
  pthread_cleanup_pop( 0 ); /* Must be 0 */
  pthread_exit( t );
}
```

4. Yes, it could happen that readers could starve – it's just tremendously unlikely. It's just like how a writer starves: if there are many writers then writers could be selected and readers wait indefinitely.

(2.2)

Part 1 Deadlock. Suppose at the start of the contest, sauce and dough are put out on the table. Then Contestant A takes the sauce and Contestant B takes the dough. All three contestants remain blocked and are unable to reach the signal statements. You could also argue that Contestant C takes the sauce instead of Contestant A; that's equivalent.

Also if sauce and cheese are put out, it's possible that contestant C takes the sauce and is blocked, and nobody can take the cheese.

But dough and cheese being put out will not cause a deadlock, so that answer is wrong(!).

Part 2 `contestantA, contestantB, contestantC` all start at 0. And `mutex` starts at 1.

Part 3

Contestant A	**Contestant B**	**Contestant C**
`wait( contestantA )`	`wait( contestantB )`	`wait( contestantC )`
`get_sauce()`	`get_dough()`	`get_sauce()`
`get_cheese()`	`get_cheese()`	`get_dough()`
`make_pizza( )`	`make_pizza( )`	`make_pizza( )`
`post( host )`	`post( host )`	`post( host )`

The contestant code is pretty much trivial now: wait until a helper signals, then go take your ingredients and make a pizza. Once the pizza is in the oven, indicate that you are ready for more ingredients.

(3.1)

1. This eliminates the hold-and-wait condition. Since there is only one mutex, if a thread is holding it, it cannot be waiting for a different one. (You could also argue this eliminates circular wait.)

2. This also eliminates the hold-and-wait condition. If only one mutex can be held at a time, it means we are not allowed to wait for a different one while holding a mutex. (You could also argue this eliminates circular wait.)

3. This eliminates the circular-wait condition: if mutexes are always acquired in some specified order then a circular wait cannot occur.

4. This eliminates the mutual exclusion condition: if every thread has its own data then there is no shared data and no need for mutual exclusion.

5. This eliminates the hold-and-wait condition: if a thread uses trylock functionality, then a thread is not blocked if unsuccessful in acquiring a mutex which means it's not waiting.