ECE 254 Midterm Exam Solutions

J. Zarnett

October 26, 2017

(1a)

```
void* run( void* argument ) {
  int* arg = (int*) argument;

  int* count = malloc( sizeof( int ) );
  *count = 0;

  for( int i = 0; i < *arg; i++ ) {
    *count += compute( i );
  }

  pthread_exit( count );
}
```

Grading notes:

1. Part 1 is five marks:

   - 1 for correct use of `argument`
   - 1 for allocating the partial sum for this thread
   - 1 for initializing the partial sum
   - 1 for running compute in a loop
   - 1 for correct use of the `pthread_exit` function.

2. Part 2 is five marks:

   - 2 for spawning the threads correctly
   - 2 for joining the threads correctly
   - 1 for computing and printing the value

In all cases, -1 for failing to deallocate anything allocated; -1 for pointer problems.

```
int main( int argc, char** argv ) {

  if (argc < 3) {
    printf("Too few arguments provided.\n");
    return -1;
  }

  int num_threads = atoi( argv[1] );
  if (num_threads < 1) {
    printf("Invalid number of threads.\n");
    return -1;
  }

  int num_iterations = atoi( argv[2] );
  if ( num_iterations < 1 ) {
    printf("Invalid number of iterations.\n");
    return -1;
  }

  pthread_t threads[num_threads];

  for( int i = 0; i < num_threads; i++ ) {
    pthread_create( &threads[i], NULL, run, &num_iterations);
  }

  int sum = 0;
  void* rval;
  for( int j = 0; j < num_threads; j++ ) {
    pthread_join( threads[j], &rval );
    int* r = (int*) rval;
    sum += *r;
    free( r );
  }

  int total_points = num_threads * num_iterations;
  float pi = 4 * (float) sum / total_points;

  printf("Pi = %f", pi);
  return 0;
}
```

(1b)

(a) Efficient I/O is achieved through interrupts. Possible explanations (required, but a brief explanation suffices) include: 1) to avoid polling for the I/O device to complete the operation. 2) set up and get started the operation, then switch to another task and when the device completes the operation it produces an interrupt.

Alternative valid answers can be: for user input such as keyboard and mouse, interrupts make the process more efficient since the OS does not need to do polling to check for user input.

(b) Timer interrupts allow time slicing and responsive multi-tasking.

Alternative valid answers: 1) They provide the guarantee that the OS will at some point regain control of the processor, even if the currently running process uses the CPU without stopping. 2) they allow the OS to keep track of real-time by counting "ticks"

(c) Traps allow user-level programs to invoke the OS and automatically trigger a switch to privileged mode. Alternative valid answer (although not as important as the previous one): they are like hardware interrupts in that they return control to the OS, only that the software explicitly produces the interrupt.

(d) Execution mode allows the OS to protect itself from malicious (if they mention malicious, good, but it's not a requirement for the answer to be considered correct) or "buggy" processes, and it allows the OS to enforce schemes that protect one process from another.

Alternative valid answer: to control parameters that control the hardware, we need this mechanism to ensure that normal programs do not have the ability to manipulate the hardware's operation.

(e) Test-and-set is useful for the OS to implement a basic mutual exclusion mechanism (alternatively, to rely on a basic atomic operation). Ideally, they should also mention that based on this mechanism, the OS now can provide higher-level mechanisms such as mutexes, semaphores, etc.

(1c)

(a) Interrupts — both timer interrupts and I/O interrupts. For cases where the processes are CPU-bound with no I/O, timer interrupts guarantee that each time slice (or each millisecond) the OS will regain control and can switch to a different task. For normal processes that execute I/O, invoking the I/O already yields control to the OS, so the interrupt is not as essential (-0.5 marks if this detail is not mentioned at all); still, when the I/O operation completes, the device will produce an interrupt and the OS will regain control.

(b) Traps, since they explicitly invoke the OS. Alternative valid answer: The OS abstraction layer itself (the API). Since programs need to invoke some OS facilities to interact with the hardware, perform I/O, etc., then they will use these APIs to give control of the CPU back to the OS.

(2a)

Global variables:

```
int tasks;
pthread_mutex_t mutex;
sem_t empty_list;
sem_t full_list;
```

Initialize global variables in `init` and perform cleanup of anything allocated or initialized in `cleanup` (4 marks):

```
void init() {

  tasks = 0;
  pthread_mutex_init( &mutex, NULL );
  /* Dobby can proceed at the start of time */
  sem_init( &empty_list, 0, 1 );
  /* Elves have to wait for Dobby */
  sem_init( &full_list, 0, 0 );

}
```

```
void cleanup() {

  pthread_mutex_destroy( &mutex );
  sem_destroy( &empty_list );
  sem_destroy( &full_list );

}
```

Complete the C functions below for the dobby and the house elves (8 marks):

```
void* dobby( void * ignore ) {
  /* while true is fine also */
  while ( 1 ) {
    /* Dobby is blocked if the list is not empty */
    sem_wait( &empty_list );
    post_tasks();
    tasks += 20;
    sem_post( &full_list );
    /* On the next iteration of the loop Dobby
       will probably be  blocked because the
       list is likely to not be empty.
       The wait could be at the end of the
       loop, but not in both places. */
  }
}
```

```
void* house_elf( void * ignore ) {
  while( 1 ) {

    pthread_mutex_lock( &mutex );
    if ( tasks == 0 ) {
      /* Nothing is available to do; must wake up Dobby */
      sem_post( &empty_list );
      /* Now wait for Dobby to finish posting tasks */
      sem_wait( &full_list );
    }
    /* At least one task is available because either
       tasks was not zero or we waited for Dobby
       to post more */
    tasks--;
    take_task();
    pthread_mutex_unlock( &mutex );
    do_work();
  }

}
```

(2b)

```
wait( chopsticks[ id ] )
wait( chopsticks[ (id + 1) % n )
eat()
signal( chopsticks[ id ] )
signal( chopsticks[ (id + 1) % n )
```

The order of the wait statements may vary; the order of the signal statements can also vary.

(2c)

Although a semaphore has some internal integer counter, using an integer is not equivalent to a semaphore for two reasons:

- Operations incrementing and decrementing an integer are not guaranteed to be atomic; a semaphore wait or signal (post) operation manipulates the counter atomically.

- A thread or process that decrements an integer would not be blocked if the value is negative after the decrement; when waiting on a semaphore if the value becomes negative the thread or process is blocked.

(2d)

a) 0: In the signalling synchronization pattern, the initial value of the semaphore must be zero so that the thread to receive the signal is blocked until the thread sending the signal has sent it.

b) 1: When using a semaphore to implement mutual exclusion, the initial value should be 1 so that the first thread that wants to enter into the critical section can actually do so; otherwise all threads might get blocked.

c) 10: In the producer consumer example, we initialized two semaphores to represent the number of items and spaces; at the beginning of time the number of spaces is the size of the buffer which in this specific example is 10. (Alternative answer: multiplex).