# ECE 254 F16 Midterm Solutions

J. Zarnett

September 13, 2016

**(1A)**

```
int main( void ) {

  pid_t pid;
  int child_result;
  int parent_result;

  pid = fork();

  if ( pid < 0 ) { /* Fork Failed */
    return -1;
  } else if ( pid == 0 ) { /* Child */
    return execute_B();
  } else { /* Parent */
    parent_result = execute_A();
    wait( &child_result );
  }

  if ( child_result == 0 && parent_result == 0 ) {
    printf( "Completed.\n" );
    return 0;
  }

  if ( child_result != 0 ) {
    printf( "Error %d Occurred.\n", child_result);
  }
  if ( parent_result != 0 ) {
    printf( "Error %d Occurred.\n", parent_result);
  }

  return -1;
}
```

**(1B)**

**Asynchronous**: the thread is terminated immediately upon the call to `pthread_cancel`.

**Deferred**: the thread is notified that it has been cancelled and is responsible for cleaning itself up and exiting.

**(1C)**

In general I'll accept synonyms for the official names used in the lecture notes as long as there is no ambiguity about what the state means.

- New (or created)
- Ready
- Running (or executing)
- Blocked
- Terminated (or finished, or zombie)

**(1D)**

The Serial part $S$ is 24s out of 120 $\rightarrow$ 20% or 0.2.

1. Speedup with 4 CPUs = $\frac{1}{0.2+\frac{0.8}{4}}$ = 2.5.

2. As $N$ goes to infinity the equation reduces to speedup = $\frac{1}{0.2}$ = 5.

3. 3.5 = $\frac{1}{0.2+\frac{0.8}{N}}$, solve for $N$. The value is $9.33333$ which should be rounded up to 10.

**(2A)**

Indicate how you will name and initialize your variables here:

Integer counter for customers waiting is called `customers`; starts at 0.
Mutex for controlling access to `customers` is called `mutex`; starts at 1.
Binary semaphores `customer` and `barber` both start at 0.

**Customer**

```
wait( mutex )
if customers == n
    signal( mutex )
    return
end if
customers++
signal( mutex )

signal( customer )
wait( barber )
get_hair_cut()

wait( mutex )
customers--
signal( mutex )
```

**Barber**

```
wait( customer )
signal( barber )
cut_hair()
```

**(2B)**

```
sem_t a_ready;
sem_t b_ready;

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

    sem_init( &a_ready, 0, 0 );
    sem_init( &b_ready, 0, 0 );

    pthread_t thread_a;
    pthread_t thread_b;

    pthread_create(&thread_a, NULL, execute_a, NULL);
    pthread_detach( thread_a );
    pthread_create(&thread_b,  NULL, execute_b, NULL);
    pthread_detach( thread_b );
    pthread_exit(0);
}


void *execute_a( void* ignore ) {
    prepare_a();
    sem_post( &a_ready );
    sem_wait( &b_ready );
    run_a();
sem_destroy( &a_ready );
    pthread_exit(0);
}

void *execute_b( void* ignore ) {
    prepare_b();
    sem_post( &b_ready );
    sem_wait( &a_ready );
    run_b();
    sem_destroy( &b_ready );
    pthread_exit(0);
}
```

**(3A)** The options in class were:

1. The priority of the process.
2. How long the process has been executing.
3. How long is remaining in execution, if known.
4. What resources the process has (number and type).
5. Future resource requests, if known.
6. Whether the process is user-interactive or in the background.
7. How many times, if any, the process has been selected as a victim.

Any four of those (1 mark for noting the criterion; 1 for how it is used).

Sample answer:

1. Priority: we prefer to kill processes with lower priority.
2. Execution time so far: we prefer to kill processes that have executed for less time (as less work needs to be repeated).
3. User-Interactivity/Background: we prefer to kill background processes so that users are less likely to notice the disruption.
4. Number of times selected: to prevent a situation where one process never finishes, we prefer not to kill the same process multiple times.

**(3B)**

1. Mutual Exclusion
2. Hold & Wait
3. No Preemption
4. Cycle in the resource allocation graph

**(3C)** Yes. `pthread_mutex_trylock` is a nonblocking call and thus a thread that uses it will not get blocked if the call does not succeed in acquiring the lock. A thread that fails to acquire a mutex may release other mutexes it holds before trying again, for example, or can go on to do something else entirely. Either way, it is not blocked and the deadlock is avoided.