ECE 252 S19 Final Exam Solutions

(1)

```
size_t read_cb( void *dest, size_t size, size_t nmemb, void *inputdata ) {
  char* pw = (char*) inputdata;
  int len =  strlen( pw );
  memcpy( dest, pw, len );
  return len;
}

size_t write_cb( char *ptr, size_t size, size_t nmemb, void *userdata ) {
  memcpy( userdata, ptr, nmemb );
  return nmemb;
}

char * authenticate( char* username, char* password ) {
  CURL* c = curl_easy_init();
  char * url = malloc( strlen( BASE_URL ) + strlen( username ) + 1 );
  char * token = malloc( TOKEN_LENGTH + 1 );
  memset( token, 0, TOKEN_LENGTH + 1 );
  sprintf( url, "%s%s", BASE_URL, username );

  curl_easy_setopt( c, CURLOPT_URL, url );
  curl_easy_setopt( c, CURLOPT_POST, 1L );
  curl_easy_setopt( c, CURLOPT_WRITEFUNCTION, write_cb );
  curl_easy_setopt( c, CURLOPT_WRITEDATA, token );
  curl_easy_setopt( c, CURLOPT_READFUNCTION, read_cb );
  curl_easy_setopt( c, CURLOPT_READDATA, password );

  CURLcode ret = curl_easy_perform( c );
  if ( ret != CURLE_OK || get_http_code( c ) != 200 ) {
    free ( token );
    token = NULL;
  }
  curl_easy_cleanup( c );
  free( url );
  return token;
}
```

- readcb: 1 for copying the data in, 1 for return length of password [2]
- writecb: 1 for copying the data, 1 for return length [2]
- 1 for easy init, 1 for cleanup, 1 for perform [3]
- 1 for check of CURLCode, 1 for check of http code; 1 for free of token, 1 for setting NULL [4]
- Setting all six options, half mark each [3]
- 1 for malloc of token, 1 for malloc of url [2]
- 1 for sprintf of url, 1 for memset of token [2]
- 1 for freeing url, 1 for return [2]

Other notes: you can use calloc for token. Note sprintf adds the null terminator for you. But the +1 is needed on the malloc of url.

(2)

```
typedef bool (*test_fn)( ); /* Define function pointer for test function */

void* executor( void* arg ) {
    test_fn f = (test_fn) arg;
    bool * res = malloc( sizeof( bool ) );
    *res = f();
    pthread_exit( res );
}

int run_tests_parallel( test_fn * tests, int num_tests ) {
    int failed_tests = 0;
    pthread_t * threads = malloc( num_tests * sizeof( pthread_t ) );
    for ( int i = 0; i < num_tests; i++ ) {
        pthread_create( &threads[i], NULL, executor, tests[i] );
    }
    void * rv;
    for ( int i = 0; i < num_tests; i++ ) {
        pthread_join( threads[i], &rv );
        bool * b = (bool *) rv;
        if ( ! (*b) ) {
            failed_tests++;
        }
        free( rv );
    }
    free( threads );
    return failed_tests;
}
```

Grading notes executor (4 total):

- Creating a return value 1 (half if they abuse the `pthread_exit` function for this so no malloc is needed)
- Invoking the function 1
- Assigning the return value 1
- return or `pthread_exit` with the result 1

Grading notes in the run function (9 total):

- for loop for create 1
- `pthread_create` 3
- for loop for join 1 (-3 if loops are combined)
- join return value goes somewhere 1
- `pthread_join` 2
- increment failure count only if the return value is appropriate 1

(3.1) I've chosen `open` as the semaphore, but it's okay to think of it as `closed` instead. Just be consistent.

For part 1, add the following to the customer, before statement 1 (and of course post instead of signal is perfectly fine).

```
if trywait( open ) != 0
    return
end if
signal( open )
```

For part 2, the barber:

```
signal( open )
while ! is_end_of_day()
  wait( customer )
  signal( barber )
  cut_hair()
end while
wait ( open )
cancel_customers()
```

For part 3, the barber

```
signal( open )
while ! is_end_of_day()
  wait( customer )
  signal( barber )
  cut_hair()
end while
wait ( open )
while trywait( customer ) == 0
  signal( barber )
  cut_hair()
end while
```

(3.2) 1. This does not work. If the `kill` call is sent before the `pause()` call, then Process B will be stuck waiting.

2. Each operation is atomic, but the whole thing is not. Suppose the value of `i` is 42. Thread 1 calls the function concurrently with Thread 2. Thread 1 reads 42. Then Thread 2 reads 42. Then Thread 1 writes 43. Then Thread 2 writes 43. The answer is wrong; it should be 44. Fix by: changing the 0 parameter to 1 in the first function and deleting the second line.

Also okay is using a different function to do this (any other valid one). Half marks if you said use a mutex; it defeats the purpose of atomic...

3. Yes, this works. Two distinct mutexes cannot be at the same location; therefore we are certain every mutex has a unique identifier.

(3.3)

Vector for L1		
A	A	?

Vector for L2		
A	A	?

Vector for L3		
A	A	?

Therefore they decide to attack.

Part 2: Yes. If the general sends everyone the same order, then the lieutenants carry it out. That is the easy case.

If the general sends mixed messages, because there are only two actions (attack/retreat) and three lieutenants, there will always be a majority. Two lieutenants will receive action 1 (whatever it is) and one will receive action 2. The lieutenants are all honest, so a majority is formed for action 1 and that is what happens.

(4)

Part 1: 0.5 X 50 = 25 seconds

Part 2: Only L2 is incorrect. We only have one buffer and increasing the number of producers should not bring any timing improvement. Most likely the student accidentally created more buffers in this case to produce this type of performance gain. Rest of the timing data are reasonable. The problem is producer speed bounded, hence increasing the producers number when buffer size is 5 will bring improvement. But increasing the consumer numbers won't change anything.

Part 3: The shmctl cleanup routine will not be reached if the program crashes before it gets there. Use the "ipcrm" command to remove the left over shared segments.

Part 4: The child process terminates, but the parent process does not call wait/waitpid to collect the child process's exit status. Let parent call wait/waitpid to collect the exit status of the child process.

(5.1)

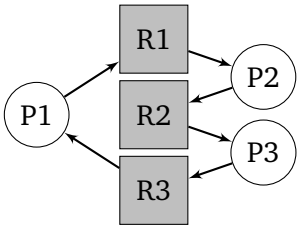
1. This will prevent deadlock. It eliminates the no-preemption condition which is necessary for deadlock to occur.

2. This will prevent deadlock. This reduces the number of resource requests by 1, meaning a cycle in the resource allocation graph will not form and deadlock will not occur. (You can also argue that if this philosopher gets a chopstick, he will eventually finish and leave; allowing at least one other philosopher to have 2, etc.; if he gets none than pigeonhole principle says someone has 2...)

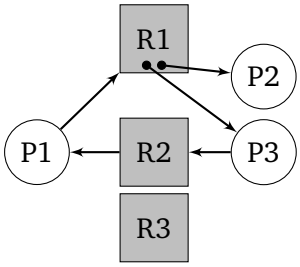
3. This will not prevent deadlock. It is still possible to be in a scenario where each philosopher has only 1 chopstick, no matter where on the table they took it from. If that is the case, a deadlock still occurs.

(5.2)

(i)



(ii) The question asks you to have a resource allocation graph with 3 processes and 3 resources; there is no requirement that all resources have to be involved. All you need is a cycle, but no deadlock.



(6.1)

```
void* file_watcher( void* arg ) {
  char* config_filename = (char*) arg;
  int buffer_size = sizeof( struct inotify_event ) + strlen( config_filename ) + 1;
  struct inotify_event * buffer = malloc( buffer_size );

  int ifd = inotify_init( );
  inotify_add_watch( ifs, config_filename, IN_MODIFY );

  while( !quit ) {
    int bytes_read = read( ifd, buffer, buffer_size );
    if ( bytes_read != 0 ) {
      reload_config_file( );
    }
  }

  close( ifd );

  free( buffer );
  pthread_exit( 0 );
}
```

(6.2)

```
int open_and_lock( char* filename ) {
  int fd = open ( filename, O_RDWR );
  struct flock fl;

  fl.l_whence = SEEK_SET; // 0.5
  fl.l_type = F_WRLOCK;  // 0.5
  fl.l_start = 0; // 0.5
  fl.l_len = 0; // 0.5

  int rv = fcntl( fd, F_SETLKW, &fl );  // 1

  return rv == 0 ? fd: -1; // 1
}
```

(7.1) 9 marks first part; 16 in the second part

```
int* send_requests( int service_id, int user_id, struct advertiser * advertisers, int length ) {
  int sid = htonl( service_id );
  int uid = htonl( user_id );

  int * sockets = malloc( length * sizeof(int) );

  for ( int i = 0; i < length; i++ ) {
    sockets[i] = open_socket( advertisers[i].addr );
    if ( sockets[i] == -1 ) {
      continue;
    }
    send( sockets[i], &sid, 4, 0 );
    send( sockets[i], &uid, 4, 0 );
  }
  return sockets;
}

int get_responses( int* sockets, struct advertiser * a, int length ) {
  int maxfd = 0;
  int winner = 0;
  int winning_bid = 0;
  fd_set set;
  struct timeval tv;
  tv.tv_sec = 1;
  tv.tv_usec = 0;

  bool done = false;
  while( !done ) {
    FD_ZERO( &set );
    for ( int i = 0; i < length; i++ ) {
      if ( sockets[i] == -1 ) {
        continue;
      }
      FD_SET( sockets[i], &set );
      if ( sockets[i] > maxfd ) {
        maxfd = sockets[i];
      }
    }

     select( maxfd + 1, &set, NULL, NULL, &tv );
     for ( int i = 0; i < length; i++ ) {
```

```
       if ( FD_ISSET( sockets[i], &set ) {
         int currentbid;
         recv( sockets[i], &currentbid, 4), 0 );
         currentbid = ntohl( currentbid );
         if ( currentbid > winning_bid ) {
            winning_bid = currentbid;
            winner = advertisers[i]->id;
          }
          close( sockets[i] );
          sockets[i] = -1;
        }
      }
      if ( tv.tv_sec == 0 && tv.tv_usec == 0 ) { /* Time's up */
        done = true;
      }
    }
  for ( int i = 0; i < length; i++ ) {
    if ( sockets[i] != -1 ) {
      close ( sockets[i] );
    }
  }
  free( sockets );
  return winner;
}
```

(7.2) The approach on the left is arguably more efficient. On the right, you wait until all `aiocb` structures are initialized before enqueueing them en masse. On the left, while you create control block `i`, previously-created control blocks may have already started running.

You could also make the complete opposite argument, that the right side is more efficient, if you argue that the cost of the `aio_read` system call being repeatedly invoked outweighs the benefits of starting some requests early.