ECE 252 S19 Midterm Solutions

J. Zarnett

August 1, 2019

(1)

There are multiple distinct errors. Any three of which you needed; no points for repetition.

1. Line 39: The PNG signature is only partially checked. The first byte is not examined, but it needs to be, so you should actually be checking against `sig` and `'\211PNG'`. It was there for a reason!
2. Line 54, 70 and 71: not converting chunk length or width to host order. Fix by using `ntohl()`. Because it asks for distinct, whether you identified one or multiple of these lines, it counts as one. (You could also convert width and height in the `printf` statement, that's fine as well). Also, we were extra nice and as long as you recognized that there was a conversion error you got some points here, even if the direction is backwards.
3. The chunk `p_ck_IHDR->p_data` memory is not allocated. Fix with `p_ck_IHDR->p_data = malloc(DATA_IHDR_SIZE)` or `malloc( sizeof( struct data_IHDR ) )` statement that goes somewhere after line 44 but before line 68.
4. If you allocate data for the header (item 3 in this list) we'll also be nice and count deallocation as something here as well.
5. Missing `fclose( fp )` after the last `fread()` on line 68 but before the `return` statement at the end of `main()`.
6. On line 54/'68 the macro `DATA_IHDR_SIZE` is used but it isn't properly defined. This wasn't intended as an answer, but we'll take it. It should be `sizeof( struct data_IHDR )` instead or a define directive that says.
7. We were also nice and accepted if you said there's no error checking the arguments and added some checks there.

(2.1) The signal handler can have any function name as long as the same name is used in main to register it. Creating the signal handler is worth 2, registering it 1.

```
void handle_sigint( int signo ) {
  quit = 1;
}
```

Then the signal handler is registered as:

```
signal( SIGINT, handle_sigint );
```

(2.2)

```
int main( int argc, char** argv ) {
  /* Create Socket, IPv4 / Stream */
  int sockfd = socket( AF_INET, SOCK_STREAM, 0 );

  /* Create the sockaddr in on port 1701 */
  struct sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_port = htons( 1701 );
  addr.sin_addr.s_addr = htonl( INADDR_ANY );
  char* rcv_buf = malloc( 65 );

  bind( sockfd, (struct sockaddr*) &addr, sizeof( addr ));
  listen( sockfd, 20 ); /* Any value for the backlog size is OK, even 0 */

  while( true ) {
    int newsockfd = accept( sockfd, NULL, NULL );
    memset( rcv_buf, 0, 65 );
    int r = recv( newsockfd, rcv_buf, 64 );
    if ( authenticate( rcv_buf ) ) {
      logdata* l = load_logdata( );
      sendall( newsockfd, l->data, l->length );
      free( l->data );
      free( l );
    }
    close( newsockfd );
  }

  free( rcv_buf );
  return 0;
}
```

(2.3)

```
void process_unit( struct work_unit* wu ) {

  int shmid = shmget( IPC_PRIVATE, sizeof( struct result ),  IPC_CREAT | 0666 );
  pid_t p = fork();
  if ( p > 0 ) { /* Parent */
    wait( NULL );
    void* mem = shmat( shmid, NULL, 0 );
    save_result( (struct result*) mem );
    shmdt( mem );
    shmctl( shmid, IPC_RMID, NULL );
  } else { /* Child */
    void* mem = shmat( shmid, NULL, 0 );
    struct result r = do_work( wu );
    memcpy( mem, &r, sizeof( struct result ) );
    shmdt( mem );
    exit( 0 );
  }
}
```

Grading notes:

- Using `waitpid()` instead of `wait` is fine, as long as you use the correct PID (`p`).
- The parent must be the one to delete the shared memory
- Instead of using `memcpy` you can also cast `mem` to be a `struct mem*` (let's call it `a`) and then do `*a = *r;` which will copy the data too. The data does need to be copied into the shared memory, one way or another.
- There's no requirement to copy the data out of shared memory though, you can just use it as is (cast the pointer though)
- The `exit` call in the else block is a big hint that this is the child!

(3.1)

```
void * foo_thr( void * arg ) {
  struct bar* b = NULL;

  pthread_cleanup_push( foo_handler, b );
  b = malloc(sizeof( struct bar ));

  /* Code to initialize b and do something with
       it not shown*/

   pthread_cleanup_pop( 0 );

   pthread_exit( b );
}
```

```
void foo_handler( void * ptr ) {
  if ( ptr != NULL ) {
    free( ptr );
  }
}
```

2 for the correct use of push and pop; 1 for the handler function.

(3.2)

```
/* Assume these are initialized/cleaned up
    elsewhere */
/* Queues ARE thread-safe */
Queue * input_queue;
Queue * processing_queue;
Queue * output_queue;

void enqueue( Queue q, void * i );
widget* dequeue( Queue q );
```

```
/* Declare Global Variables Here */
sem_t input_q_sem;
sem_t processing_q_sem;
sem_t output_q_sem;
```

```
/* Initialize your semaphores & mutexes */
void init_sync(  ) {
  sem_init( &input_q_sem, 0, 0 );
  sem_init( &processing_q_sem, 0, 0 );
  sem_init( &output_q_sem, 0, 0 );
}

void cleanup_sync( ) {
  sem_destroy( &input_q_sem );
  sem_destroy( &processing_q_sem );
  sem_destroy( &output_q_sem );
}

void* parser_thread( void * ignore ) {
  while( !quit ) {
    sem_wait( &input_q_sem );
    input* i = (input*) dequeue( input_queue );
    data * d = parse( i );
    enqueue( processing_queue, d );
    sem_post( &processing_q_sem );
  }
  pthread_exit( 0 );
}
```

```
/* Create num_parser detached parser threads */
void start_parsers ( int num_parsers ) {
  for ( int i = 0; i < num_parsers; i++ ) {
    pthread_t t;
    pthread_create( &t, NULL, parser_thread,
        NULL );
    pthread_detach( t );
  }
}

void* processor_thread( void * ignore ) {
  while( !quit ) {
    sem_wait( &processing_q_sem );
    data * d = (data*) dequeue(
        processing_queue );
    processed * p = process( p );
    enqueue( output_queue, p );
    sem_post( &output_q_sem );

  }
  pthread_exit( 0 );
}
```

Grading notes:

- Creating threads: 1 for loop, 2 for pthread create, 1 for detach (4 total)
- Global vars: 1 for the first, 0.5 subsequent (2 total)
- Init: 2 for the first, 0.5 subsequent (1 total) (note initial value!)
- Cleanup: 1 for the first, 0.5 subsequent (2 total)
- Use of semaphores: 1 for each wait and post call (4 total)