# ECE 252 F19 Midterm Exam Solutions

### (1.1)

```
void process( char* filename, key_t key ) {
  int queue = mssgget( key, 0640 | IPC_CREAT );

  int fd = open( filename, O_RDONLY );
  while( true ) {
    int type;
    int b = read( fd, &type, sizeof( int ) );
    if ( b == 0 ) {
      break;
    }

    eval ev;
    memset( &ev, 0, sizeof( eval ));
    ev.mtype = type;

    int size;
    if ( type == CMD_TYPE ) {
      size = CMD_SIZE;
    } else if ( type == OPS_TYPE ) {
      size = OPS_SIZE;
    } else { /* Science */
      size = SCI_SIZE;
    }
    read( fd, &(ev.data), size );

    /* Send to queue */
    msgsnd( queue, &ev, sizeof( eval ), 0 );
  }
  close( fd );
}

int main( int argc, char** argv ) {
  key_t key = ftok( argv[1], 1 );
  process( argv[1], key );
}
```

- Open 1
- Get Queue 1
- Loop until end of file 1
- Read type 1
- create eval and memset 1
- set type 1
- read data 2
- send to queue 2
- close file 1

It's important NOT to destroy the queue since the recipient programs need to get the messages out of it. And just because we have SENT all messages into the queue does not mean all have been collected.

You can also do this with `FILE*` pointers and `fopen, fread` etc. That solution is equally valid to this one.

**(1.2)**

```
const char *ping = "ping";

int main( int argc, char **argv ) {
    struct addrinfo hints;
    struct addrinfo *res;

    memset( &hints, 0, sizeof( hints ) );
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    int r = getaddrinfo( argv[1], "5000", &hints, &res );
    if( r != 0 ) {
      return -1;
    }

    int sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if ( sockfd < 0 ) {
      return -1;
    }

    int result = connect( socked, res->ai_addr, res->ai_addrlen );
    if ( result != 0 ) {
      return -1;
    }

    bool closed = false;
    bool error = false;
    while( !closed ) {
      int s = send( sockfd, ping, strlen(ping) + 1, 0 );
      if ( s == -1 ) {
        closed = true;
        error = true;
        break;
      }
      time_t response;
      int r = recv( sockfd, &response, sizeof( response ), 0 );
      if ( recv == -1 ) {
        error = true;
        closed = true;
      } else if ( recv == 0 ) {
         closed = true;
      } else {
         response = ntohl( response );
         printf( "%s", ctime( response ));
      }
   }
   close( sockfd );
   freeaddrinfo( res );
   return error ? -1 : 0;
}
```

- Loop until closed 1
- `send`: 2
- Check error of `send` 1
- `recv`: 2
- check error of `recv` 1
- check for done 1
- `ntohl` 1
- print response 1
- `close` socket 0.5
- `freeaddrinfo` 0.5
- return value from main 1

**(2.1)**

```
void gordon( pthread_t* chefs, int num_chefs ) {
  /* Create chef threads, assign stations */
  for ( int i = 0; i < num_chefs; i++ ) {
    pthread_create( &chefs[i], NULL, chef, get_station(i) );
  }

  order* next = get_next();
  while( next != NULL ) {
    int num_stns;
    station* stns;
    read_order( o, &stns, &num_stns );

    for (int i = 0; i < num_stns; i++) {
      sem_post( &(stns[i].call) );
    }
    for (int i = 0; i < num_stns; i++) {
      sem_wait( &(stns[i].food) );
    }
    free( stns );
    if ( check_food() ) {
      serve( next );
      next = get_next();
    } else {
      swear();
    }
  }

  /* All orders complete */
  for ( int i = 0; i < num_chefs; i++) {
    pthread_cancel( chefs[i] ); /* pthread_kill would also be thematically correct */
  }
}
```

- Create chefs with `pthread_create` 2
- loop (while or for), advance + terminate 2
- `read_order` 2
- post on the relevant call semaphores 2
- wait on the relevant food semaphores 2
- if OK, serve food 1, free 1
- swear if food not okay 1
- Loop to clean up chefs 1

If you did `pthread_join` on the chefs that is fine – not wrong, but also not necessary.

**(2.2)**

```
[ "C A B", "C B A" ]
```

**(3)** Part 1:

```
long get_size_inflated_data_RGBA( int width, int height, int bit_depth ) {
    int bytes_per_sample = 0;

    /* some sanity check here */
    if ( width < 0 || height < 0 ) {
        return -1;
    }

    if ( bit_depth != 8 && bit_depth != 16 ) {
        return -1;
    }

    bytes_per_sample = bit_depth / 8;
    return ( 1 + 4 * width * bytes_per_sample ) * height;
}
```

2 marks for sanity checks; 3 marks for the calculations.

Part 2:

1. Not checking the return value of `get_size_inflated_data_RGBA`. Fix by adding an if block that exits the function if an error occurred.
2. Allocating an arbitrarily large chunk of memory on the stack is not possible. If the image is too large, the behaviour will be undetermined as stack gets overflowed. One should use `malloc` to dynamically allocate memory in this case.