Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination Midterm Fall 2019 ECE 252

Times:	Monday 2019-10-21 at 17:45 to 18:45 (5:45 to 6:45PM)
Duration:	1 hour (60 minutes)
Exam ID:	4288816
Sections:	ECE 252 LEC 001,002
Instructors:	Andrew Morton, Jeff Zarnett

Special Materials

Candidates may bring only the listed aids.

· Calculator - Non-Programmable

Instructions:

1. No aids are permitted except non-programmable calculators with no persistent memory.
2. Turn off all communication devices. Communication devices must be stored with your personal items for the duration of the exam. Taking a communication device to a washroom break during this examination is not allowed and will be considered an academic offence.
3. Place all bags at the front or side of the examination room, or beneath your table, so they are inaccessible.
4. There are three (3) questions, some with multiple parts. Not all are equally difficult.
5. The exam lasts 60 minutes and there are 50 marks.
6. Verify that your name and student ID number is on the cover page and that your examination code appears on the bottom of each page of the examination booklet.
7. If you feel like you need to ask a question, know that the most likely answer is “Read the Question”. No questions are permitted. If you find that a question requires clarification, proceed by clearly stating any reasonable assumptions necessary to complete the question. If your assumptions are reasonable, they will be taken into account during grading.
8. An API reference document is provided to accompany this exam. This will not be collected, so do not use it as extra space.
9. Do not fail this city.
10. After reading and understanding the instructions, sign your name in the space provided below.

Signature

1 Interprocess Communication

1.1 Message Queues [11 marks]

On the *Enterprise*, every three months, Commander Riker and Counselor Troi perform crew evaluations. Commander Riker needs to send the results to the head of each division; he's asked you to write a program to help. Completed evaluations are stored in a file on the main computer, sequentially, but in no particular order. Evaluations need to be read into memory and sent to the division heads using a message queue.

Crew members are evaluated on different criteria depending on their division, and accordingly, the evaluation data is a different size for each division. Each evaluation in the file is preceded by its type (which is 4 bytes long). Thus, the first four bytes of the file are the type of the first evaluation. This type tells you how many bytes the first evaluation is. It is followed by the first evaluation. Then, the next four bytes will be the type of the second evaluation, followed by the data of second evaluation. This pattern continues until the end of the file.

See the table for the sizes and types:

Division	Size of Evaluation	Evaluation Type
Command (Red)	CMD_SIZE	CMD_TYPE
Operations (Gold)	OPS_SIZE	OPS_TYPE
Science (Blue)	SCI_SIZE	SCI_TYPE

The structure of a message to be put on the queue is below. Regardless of the type of evaluation, the `struct eval_data` is large enough to hold it. The `mtype` field should be assigned the type of evaluation.

```
typedef struct {
  long mtype;
  struct eval_data data;
} eval;
```

Once an evaluation is ready, send it into the message queue. When creating the queue, use the provided `key_t` to create it with the flags `0640 | IPC_CREAT`. Complete the `process()` function below to implement the behaviour described above. For the sake of time, you may assume all system calls will succeed.

```
void process( char* filename, key_t key ) {









}

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

1.2 Ping... Pong... [12 marks]

The `ping` command is used to check if a remote system is online and reachable. We will use sockets to simulate this command. The client creates a streaming connection with the server on port 5000. Whenever the client sends the string “ping” to the server, the server sends back the current time. The time sent is of type `time_t` and it is a 4-byte numeric type. The server will respond an unspecified number of times before it closes the connection.

Complete the code for the client below. The client should repeatedly ping the server by sending the string “ping” to the remote server. After each ping, the client should print the time received from the server using

```
char *ctime(const time_t *timep)
```

which returns a null-terminated string of the form

```
"Tue Oct  8 17:59:56 2019\n"
```

After the server closes the connection, the client should close the socket and free any dynamically allocated memory. Assume all necessary header files are included. Make sure to check return values of your system calls; if an error occurs, and return -1 from `main` after cleaning up.

```
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( sockfd, res->ai_addr, res->ai_addrlen );
    if ( result != 0 ) {
      return -1;
    }











}
```

2 Concurrency & Synchronization

2.1 Where's the lamb sauce?! [14 marks]

On his well-known cooking competition show, Gordon Ramsay orders around a group of chefs. The chefs work in a system where each chef is assigned a station (appetizer, meat, fish, garnish, or dessert). At the start of service, Gordon creates the chef threads and assigns them to stations. When an order arrives at the kitchen, Gordon activates the relevant chefs, waits for their food to be ready, and then checks it. If the food is okay, it is served. If not, he swears a lot and the order must be restarted. Once all orders are successfully served, Gordon terminates all chef threads. Complete the `gordon()` function below to implement this behaviour. You can assume that the relevant `station` and `order` structures have been created and initialized properly.

```
typedef struct {
  sem_t call;
  sem_t food;
} station;

typedef struct
  dish * dishes; /* Array of dishes */
  int size; /* Size of the array */
} order;

/* Used by a chef to cook the current order */
void cook( );

/* Returns a pointer to the station that the chef with
     the given array index is supposed to take for
     tonight's dinner service */
station* get_station( int i );

/* Get the next order from the waitstaff; returns NULL
     if there are no more orders */
order* get_next( );

/* Reads the order and updates the provided pointer s
     to point to an array of the relevant stations for
     this order; num is assigned the size of that array
     . Deallocation of the returned array is the
     responsibility of the caller.  */
void read_order( order* o, stations ** s, int * num );

/* Returns true if the food is made correctly; false
     otherwise */
bool check_food( );

/* Used by Gordon if the dish is not correct */
void swear( );

/* Serves the dish (use only if dish is correct)
   Deallocates the order when it has been served */
void serve( order* order );

void* chef( station* s ) {
  while( true ) {
    sem_wait( &(s->call) );
    cook();
    sem_post( &(s->food) );
  }
}
```

```
void gordon( pthread_t* chefs, int num_chefs ) {
  /* Create chef threads, assign stations */









  /* Start cooking orders */



























  /* All orders complete */






}
```

2.2 Coordination is Complicated [4 marks]

Three (3) threads, all created simultaneously, each execute the pseudocode below. Write down all possible outputs. Initial values are: `mutex` = 1, `count` = 0, `semA` = 0, `semB` = 0.

```
wait( mutex )
count++
if count == 1
    post( mutex )
    wait( semA )
    wait( mutex )
    print A
    post( mutex )
else if count == 2
    post( mutex )
    wait( semB )
    wait( mutex )
    print B
    post( mutex )
else
    post( semA )
    post( semB )
    print C
    post( mutex )
end if
```

3 PNG File Manipulation [9 marks]

In lab1 and lab2, we worked on simple PNG format files. An important step in PNG file concatenation is to read the data field from an IDAT chunk, and then inflate the data to a buffer. You will need to allocate enough buffer space to hold the inflated data. Assume the PNG image is RGBA. That is: for each pixel it contains red, green, blue, and alpha samples. Recall bit depth means how many bits are needed to represent a sample. The image dimensions are described in terms of horizontal and vertical pixels.

Part 1. [5 marks] Complete the function below calculate the inflated data buffer size.

```
/**
 * @brief returns the number of bytes a uncompressed filtered RGBA PNG image needs
 * @param int width, width of the image, should be positive
 * @param int height, height of the image, should be positive
 * @param int bit-depth, number of bits per sample needs, only allows 8-bit or 16-bit samples
 * @return long, number of bytes the uncompressed filtered RGBA PNG image needs on success; -1 on failure
 */
long get_size_inflated_data_RGBA( int width, int height, int bit_depth ) {









}
```

Part 2. [4 marks] Assume the above function is implemented correctly. Now you want to use it in your code to inflate an image (only has one idat chunk) that has dimensions of 400 x 6000 and 8-bit depth.

```
void inflate_one_idat( void *args ) {
    long size_inf = get_size_inflated_data_RGBA( 400, 6000, 8 );
    char buf_inf[size_inf];
    /* call mem_inf here and buf_inf is the destination buffer to hold the inflated data */
    /* do the rest of stuff here, assume args contain output params */
}
```

Identify two (2) problems with the above code and explain how you would fix them.