Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Midterm
Spring 2019
ECE 252

Times: Thursday 2019-06-20 at 13:45 to 14:45 (1:45 to 2:45PM)
Duration: 1 hour (60 minutes)
Exam ID: 4128259
Sections: ECE 252 LEC 001
Instructors: 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

Marking Scheme (For Examiner Use Only):

Question	Mark	Weight	Question	Mark	Weight	Question	Mark	Weight
1		9	2.1		3	3.1		3
‘			2.2		12	3.2		15
			2.3		8			
Total								**50**

Please initial:

1 Files [9 marks]

Consider the code below that prints out the length and width of the provided PNG file. This code contains multiple errors and does not work. Find three (3) distinct semantic (i.e., not syntax) issues with the code. Indicate clearly where each error is, and explain what code changes you would make (and where) to fix that problem. Use the space to the right of the code for your answer.

```
1  #define PNG_SIG_SIZE    8 /* number of bytes of png image signature data */
2  #define CHUNK_LEN_SIZE  4 /* chunk length field size in bytes */
3  #define CHUNK_TYPE_SIZE 4 /* chunk type field size in bytes */
4  const char FIRST_BYTE = '\211';
5
6  typedef unsigned char U8;
7  typedef unsigned int U32;
8  struct chunk {
9      U32 length;  /* length of data in the chunk */
10     U8  type[4]; /* chunk type */
11     U8  *p_data; /* pointer to location where the actual data is */
12     U32 crc;     /* CRC field  */
13 };
14 struct data_IHDR {    /* IHDR chunk data */
15     U32 width;        /* width in pixels, big endian   */
16     U32 height;       /* height in pixels, big endian  */
17     U8  bit_depth;    /* num of bits per sample or per palette index. valid values are: 1, 2, 4, 8, 16 */
18     U8  color_type;   /* 0: Grayscale; 2: Truecolor; 3: Indexed-color 4: Greyscale+alpha; 6: Truecolor+alpha */
19     U8  compression;  /* only method 0 is defined for now */
20     U8  filter;       /* only method 0 is defined for now */
21     U8  interlace;    /* =0: no interlace; =1: Adam7 interlace */
22 };
23
24 int main (int argc, char **argv) {
25     struct chunk *p_ck_IHDR = NULL;
26     struct data_IHDR *p_data_IHDR = NULL;
27     char sig[PNG_SIG_SIZE];
28
29     FILE* fp = fopen( argv[1], "rb" );
30     if ( fp == NULL ) {
31         exit(1);
32     }
33
34     int nbytes = fread( sig, 1, PNG_SIG_SIZE, fp );
35     if ( nbytes != PNG_SIG_SIZE ) {
36         exit(2);
37     }
38
39     if ( strncmp (sig+1, "PNG", 3 ) ) {
40         fprintf( stderr, "Not a PNG file!\n" );
41         exit(3);
42     }
43
44     p_ck_IHDR = malloc( sizeof( struct chunk ) );
45     if ( p_ck_IHDR == NULL ) {
46         exit(4);
47     }
48
49     nbytes = fread( &p_ck_IHDR->length, 1, CHUNK_LEN_SIZE, fp );
50     if ( nbytes != CHUNK_LEN_SIZE ) {
51         exit(5);
52     }
53
54     if ( p_ck_IHDR->length != DATA_IHDR_SIZE ) {
55         fprintf( stderr, "Incorrect IHDR chunk length!\n" );
56         exit(6);
57     }
58
59     nbytes = fread( &p_ck_IHDR->type, 1, CHUNK_TYPE_SIZE, fp );
60     if ( nbytes != CHUNK_TYPE_SIZE ) {
61         exit(7);
62     }
63     if ( strncmp( (char *)p_ck_IHDR->type, "IHDR", 4 ) ) {
64         fprintf( stderr, "Incorrect Chunk Type!\n" );
65         exit(8);
66     }
67
68     nbytes = fread( p_ck_IHDR->p_data, 1, DATA_IHDR_SIZE, fp );
69     p_data_IHDR = (struct data_IHDR *) p_ck_IHDR->p_data;
70     U32 width = p_data_IHDR->width;
71     U32 height = p_data_IHDR->height;
72     printf( "The dimensions of the image: %d X %d.\n", width, height );
73
74     free( p_ck_IHDR );
75     return 0;
76 }
```


Please initial:

2 Processes & Interprocess Communication [23 marks total]

2.1 Signals [3 marks]

Complete the code below to register the signal handler to react to the `SIGINT` signal and set `quit` to be `1`. Assume the functions `set_up()`, `clean_up()`, and `do_useful_work()` are defined elsewhere and work as intended.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

volatile int quit = 0;

/* Write your signal handler here */
```

```
int main( int argc, char** argv ) {
  set_up( );




  while( quit == 0 ) {
    do_useful_work( );
  }
  clean_up( );
  return 0;
}
```

2.2 Hailing Frequencies Open, Captain [12 marks]

The United Federation of Planets uses listening devices (beacons) to detect whether Romulan ships are attempting to cross the neutral zone and enter Federation territory. Periodically, Federation vessels (e.g., the USS *Enterprise*) that are in the area will check on these beacons. The Federation vessel will open a connection to the device, and authenticate itself. If authentication is successful, then the beacon will send its log data for analysis. The beacon should continue to listen for (future) incoming connections after the connection has been closed.

Complete the code below to receive incoming connections and respond to them. Your implementation will handle all requests sequentially. You may use `NULL` for both client address parameters when establishing a connection. When a connection is established, receive the 64-character secret sent by the other party. Use `authenticate()` to check if it is valid. If it is not, close the connection. If the secret is valid, send back the result, and close the connection. Use the provided `sendall()` function to send back the result to the *Enterprise*. After this, the program should continue awaiting incoming connections (use an infinite loop). Assume all calls will succeed.

```
typedef struct {
  char* data;
  int length;
} logdata;

/* Returns true if secret is valid; false otherwise */
bool authenticate( char* secret );

int main( int argc, char** argv ) {
  int sockfd = socket( AF_INET, SOCK_STREAM, 0 );

  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 );
```

```
/* Returns a logdata structure containing this sensor
     device's data. Deallocation of the struct and its
     contents is responsibility of the caller */
logdata* load_logdata( );

/* Send all bytes of buf via the socket; len is number
     of bytes to send */
int sendall( int socket, char *buf, int len );
```

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


Please initial:

2.3 Shared Memory [8 marks]

Your program processes work units and turns them into results, which are then saved. You will implement the functionality below so that the work units are done by a child process and the result is returned using shared memory. Assume that all system calls (e.g., `fork()`) will succeed.

There is a structure `struct work_unit` that contains the package of work to do and `struct result` to represent the result. The definitions of these structures are not shown.

The child process should do the work using the function `do_work()` (see definition below) and then return the result via shared memory to the parent. The parent process should wait while the child completes the processing. Once the result is returned, the parent should save it using `save_result()` (definition also shown below).

```
struct result do_work( struct work_unit wu );
void save_result( struct result *r );

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 ) {









    } else {









      exit( 0 );
    }
}
```

3 Threads & Concurrency [18 marks total]

Complete the following code to register the cancellation handlers to perform cleanup in case the thread is cancelled before the function completes:

3.1 Thread Cancellation [3 marks]

```
void * foo_thr( void * arg ) {

  struct bar* b = NULL;


   b = malloc(sizeof( struct bar ));


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


   pthread_exit( b );
}
```

```
void foo_handler( void * ptr ) {



}
```


Please initial:

3.2 You Had One Job! [15 marks]

One way that a program can be structured is by using threads: each thread has a specific job to do and it does that job on some unit of work before putting it into the next step. In this program there will be two kinds of thread: parser and processor threads. Parser threads take work out of the input queue, parse it, and put it into the processing queue. Processing threads take work out of the processing queue, process it, and put it into the output queue. The main idea is to allow work to be done in parallel to speed up operations.

Each thread should be blocked until there is some work for it to do and woken up. Each thread is responsible for signalling the next when there is an item in the input queue. You can assume that queues are effectively infinite in size, but that trying to dequeue if the queue is empty will crash your program. Queues are thread-safe, and so are the parse and process functions.

You may assume that items appear in the input queue by parts of the program not shown. You may also assume that another part of the program takes things out of the output queue. Other parts of the program will use any synchronization constructs that you create correctly, including waking up parser threads when appropriate.

Define the synchronization constructs you need in the global variables area. Then initialize them in the `init_sync` function. Clean up your synchronization constructs in `cleanup_sync()`. Implement the function `start_parsers` to create `num_parsers` detached parser threads (assume the same operation is done for processor threads). Finally, complete the parser and processor thread functionality to use the synchronization constructs correctly.

```
/* 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 );
void* dequeue( Queue *q );

/* Initialize your semaphores & mutexes */
void init_sync(  ) {









}

void cleanup_sync( ) {









}

/* Create num_parser detached parser threads */
void start_parsers ( int num_parsers ) {









}
```

```
/* Declare Global Variables Here */

void* parser_thread( void * ignore ) {
  while( !quit ) {

    input* i = (input*) dequeue( input_queue );

    data * d = parse( i );

    enqueue( processing_queue, d );

  }
  pthread_exit( 0 );
}

void* processor_thread( void * ignore ) {
  while( !quit ) {

    data * d = (data*) dequeue( processing_queue );

    processed * p = process( p );

    enqueue( output_queue, p );

  }
  pthread_exit( 0 );
}
```


Please initial:

Please initial: