Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Final
Fall 2019
ECE 252

Times: Wednesday 2019-12-18 at 12:30 to 15:00 (3PM)
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 4259110
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 four (4) questions, some with multiple parts. Not all are equally difficult.
5. The exam lasts 150 minutes and there are 120 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. If you need more space, use the last page of the exam. Clearly indicate on the page with the question that your answer continues on the last page.
10. Do not fail this city.
11. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Please initial:

1 Interprocess Communication [32 marks total]

1.1 Maybe we should have written it in C? [20 marks]

Your company's product (written in Java) sometimes runs out of memory. When that happens, information about memory usage is written to a file and then uploaded to a server. Other developers want to analyze this file, but first they need to get it. You will write a micro-service in C that is used to retrieve files by their ID number. Because the files could be large and sending can take a long time, the server will be multi-threaded so that multiple requests can be in progress at once.

The main thread should set up its socket and be ready for incoming connections. Allow a backlog of 10 pending connections. In the infinite loop, the server should handle incoming connections as follows: when a client connects, then a new (detached) thread should be created to handle that client's request. The server does not need to pay attention to the client's address. Once the new thread is created, the main thread of the server can go back to awaiting connections.

When the connection is open, the client sends a `uint32` (32-bit integer) that identifies the file it wants. If the file does not exist, the server can just close the connection without sending data. Otherwise, the server should read the file into memory in `BUF_SIZE` chunks and send those to the client. You can assume that `BUF_SIZE` is configured so that it can be sent all at once. When the server is done sending the file, it should close the connection.

Complete the code below to implement the server's functionality as described.

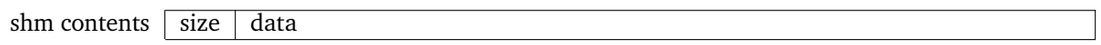
```
/* Takes a file number (32-bit integer) and returns at file descriptor for that file (opened read-only) if it
     exists. If the file does not exist (ie, fileno is invalid), -1 is returned */
int open_file( uint32 fileno );

void* serve_file( void* arg ) {
    char* buf = malloc( BUF_SIZE );









    free( buf );
    return NULL;
}

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( 2520 );
    addr.sin_addr.s_addr = htonl( INADDR_ANY );









    while( true ) {









    }
    return 0;
}
```


Please initial:

1.2 Making a File, and Checking it Twice [12 marks]

Checksums are used to verify file integrity (i.e., make sure they haven't been corrupted). Your program will calculate the checksum of a file twice: once in a child process and again in the parent process. You will then compare the results to make sure both have the same value. The processes will communicate using shared memory.

The parent process will create a shared memory segment. The child process will read the file specified by `argv[1]` and put the file size followed by the file contents into shared memory (see diagram below).



Files will be no bigger than `MAX_FILE_BYTES`. The child process will then use `calculate()` to calculate the checksum and return that value. The parent process will use `wait()` to wait for the child process and get its return value. When the child process ends, its `return` statement adds other information to the return value, so the parent process needs to use `WEXITSTATUS(return_status)` to extract the returned checksum (it produces an unsigned integer value). The parent process will then recalculate the checksum from the data in shared memory. It returns 0 if both checksums are the same and -1 if they are not.

The file should be closed and shared memory released before returning. Complete the code below to implement the behaviour described above.

```
uint8_t calculate(void *mem, size_t bytes);

int main(int argc, char **argv) {
  uint8_t childsum;
  uint8_t checksum;

  int shmid = shmget(IPC_PRIVATE, sizeof(size_t) + MAX_FILE_BYTES, IPC_CREAT | 0600);

  int pid = fork();
  if (pid < 0) {
      printf("Error␣occurred:␣pid␣=␣%d.\n", pid);
      return -2;
  } else if (pid == 0) {









  } else {









    return childsum == checksum ? 0 : -1;
 }
}
```


Please initial:

2 Concurrency & Synchronization [54 marks total]

2.1 Tests are for Cowards, but Wise Cowards [15 marks]

Code you write should have unit tests. They are executed using a framework of some sort. To be generic, tests are defined just by their function signature (see the definition below). A test returns `true` if it passes; `false` otherwise. Finally, the number of failed tests is returned to the user. The previous co-op student wrote the following code:

```
typedef bool (*test_fn)( ); /* Define function pointer for test function */
int failed_tests = 0;

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 ) {
    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;
}
```

This runs your tests, but your program has 30 000 unit tests. This framework creates 30 000 threads at once, which is not wise. You will modify this program so that `run_tests_parallel` takes a third parameter, `int threads`, which tells you the number of threads to run in parallel. Hint: you should use detached threads.

```
typedef bool (*test_fn)( ); /* Define function pointer for test function */
int failed_tests = 0;
int completed = 0;
int total_tests;
sem_t next;
sem_t done;
pthread_mutex_t lock;

void* executor( void* arg ) {









  pthread_exit( NULL );
}

int run_tests_parallel( test_fn * tests, int num_tests, int threads ) {
    total_tests = num_tests;









    return failed_tests;
}
```


Please initial:

2.2 The Slice is Nice [14 marks]

A pizza requires three ingredients: dough, sauce, and cheese. In the cooking competition show, each contestant still has an unlimited supply of one ingredient (e.g., Contestant A has an unlimited supply of dough). To make a pizza, they need the other two ingredients which the host will place on the table when signalled.

To cut costs, all the assistants got fired. Therefore, each contestant has to check the state of ingredients themselves. They can use the functions `dough_present()`, `sauce_present()`, and `cheese_present()` (which return `bool`) to test for the presence of ingredients, and `get_dough()`, `get_sauce()`, and `get_cheese()` to take the ingredients. Only one instance of an ingredient is available at a time. When they take their ingredients they post on a semaphore to tell the host to put out more ingredients. They will continue to try to make pizza in a loop until the show is over (`show_over == true`).

The host waits on a semaphore and then places two different random ingredients out. He uses `place_ingredient()` which puts out a random ingredient, and never the same one twice in a row. Because there are no assistants, the host has to yell (broadcast) that ingredients are available, so he uses a condition variable for synchronization. The host repeats this process until the show is over. Both the contestants and host use a mutex to protect accesses to the ingredients.

Implement the code of `host()` and `contestantA()` below (the other contestants' behaviour can be inferred from contestant A, so for time reasons you do not have to implement it).

Hint: the host has to yell one more time when the show is over, so contestants know they have to stop cooking.

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
sem_t sem;
bool show_over = false;

int main( int argc, char **argv ) {
    sem_init( &sem, 0, 1 );
    pthread_t tid[4];
    pthread_create( tid+0, NULL, host, NULL );
    pthread_create( tid+1, NULL, contestantA, NULL );
    pthread_create( tid+2, NULL, contestantB, NULL );
    pthread_create( tid+3, NULL, contestantC, NULL );
    sleep( SHOW_TIME );
    show_over = true;
    for( int i = 0; i < 4; i++ ) {
      pthread_join( tid[i], NULL );
    }
    pthread_exit( NULL );
}

void* contestantA( void* arg ) {
  while(!show_over) {









  }
  pthread_exit( NULL );
}

void* host( void *arg ) {
  while(!show_over) {









  }

  pthread_exit( NULL );
}
```


Please initial:

2.3 Our First Catch of the Day... [12 marks]

The Rebel Alliance is trying desperately to escape the ice planet Hoth before Darth Vader can put an end to the rebellion once and for all. The rebels want to evacuate the base with transports; transports are escorted by a pair of X-Wing fighters. Transports have no hope of surviving if they are not escorted, so the fighters must stay with their assigned transport. Normally, central command would tell the fighters and transports when to launch, but the base is under assault. Thus, the transports and fighters must organize themselves. Only one group of fighters and transports can launch at a time. In this problem, you can treat the pair of fighters as a single unit.

If a transport is ready, but fighters are not, the transport has to wait for the next fighters. If the fighters are ready but the transport is not, then the fighters have to wait for the next transport. When the fighters and transport are ready, they can both use the `launch()` function and they are off! The transport is the slower one to launch, so the next group should not proceed until the transport has finished the `launch()` process.

Write pseudocode below to implement the functionality described above. You will need an integer counter for the number of ready fighters, called `fighters`, an integer counter for the number of ready transports called `transports`, and three semaphores: `mutex`, `fighter_queue`, `transport_queue`.

Initial Values. [1.5 marks] Indicate the initial values of each of the semaphores here. The integer counters `fighters` and `transports` are both initialized to `0`.

Implementation. [10.5 marks]

Fighters **Transport**

2.4 Fly the Friendly Skies... Or don't. [13 marks]

At an airport, there are always lots of displays around that show you the flights for today. This is something that is updated rarely, but is queried frequently (from many places). It is therefore a good place to use readers-writers locks. There are 3 functions supported in this (simplified) system: display current flights in the schedule, update a flight in the schedule, and add a flight to the schedule. A flight and a schedule have the following structure:

```
typedef struct {
  char airline[21];
  int flight_no;
  char time_24hr[6];
  int gate;
} flight;

typedef struct {
  flight *flights;
  int num_flights;
  int max_flights;
  pthread_rwlock_t * lock;
} schedule;
```

The `display()` function should make a copy of the provided `schedule`'s `flights` array and return it to the caller.

The `update()` function should search the list of flights in the schedule for the provided flight. If a flight with the same airline and number is found, overwrite that flight with the new data and return true. If it is not found, change nothing and return false.

The `add()` function is to add a flight (obviously). If the schedule array is already full, it should increase the size of the array with `realloc` to double its current size. Then, add the flight to the array after the last flight in it.

You may assume that all provided arguments have been correctly allocated and initialized, and that deallocation of these structures is the responsibility of the caller. Complete the code on the next page.

```
flights* display( schedule* s ) {









}
bool update( schedule* s, flight* f) {









}
void add(schedule* s, flight *f ) {









}
```

3 Deadlock [8 marks]

Dead or Alive? [2 marks] Give two (2) symptoms of livelock that allow us to differentiate it from deadlock.

Deal with it [3 marks] If you have detected livelock, what should you do about it?

Everyone gets a signal? [3 marks] Suppose the operating system is capable of detecting deadlock and telling you what semaphores are involved. Your proposed recovery strategy is to `sem_post` on all semaphores involved in the deadlock. Is this a good strategy? Your answer should begin with yes or no, followed by your reasoning.

Please initial:

4 Asynchronous I/O [26 marks total]

4.1 Mmmmmmulti-cURL! [15 marks]

Consider the cURL multi code below that contains some errors. Find five (5) errors and for each of those, briefly explain (1) what is wrong, (2) the consequences of this issue, and (3) how to fix it. Use point form.

```
1  size_t callback( char *d, size_t s, size_t n, void *p ) {
2    memcpy( p, d, s * n );
3    return s * n;
4  }
5
6  int main( int argc, char** argv ) {
7    int still_running = 0;
8    int msgs_left = 0;
9    int outputfd = open( "output.txt", O_APPEND );
10   char** buffers = malloc( (argc - 1) * sizeof( char* ) );
11
12   curl_global_init( CURL_GLOBAL_ALL );
13   CURLM* cm = curl_multi_init( );
14   CURL *eh = curl_easy_init();
15
16   for ( int i = 1; i < argc; ++i ) {
17     buffers[i-1] = malloc( BUF_SIZE );
18     memset( buffers[i-1], 0, BUF_SIZE );
19     curl_easy_setopt( eh, CURLOPT_WRITEFUNCTION, callback );
20     curl_easy_setopt( eh, CURLOPT_WRITEDATA, buffers[i-1] );
21     curl_easy_setopt( eh, CURLOPT_PRIVATE, buffers[i-1] );
22     curl_easy_setopt( eh, CURLOPT_URL, argv[i] );
23     curl_multi_add_handle( cm, eh );
24   }
25
26   curl_multi_perform( cm, &still_running );
27
28   CURLMsg *msg = NULL;
29   while ( ( msg = curl_multi_info_read( cm, &msgs_left ) ) ) {
30     if ( msg->msg == CURLMSG_DONE ) {
31       eh = msg->easy_handle;
32
33       CURLcode return_code = msg->data.result;
34       if ( return_code == CURLE_OK ) {
35         fprintf( stderr, "Error: %d.\n", msg->data.result );
36         curl_multi_remove_handle( cm, eh );
37         curl_easy_cleanup( eh );
38         continue;
39       }
40
41       char* b;
42       curl_easy_getinfo( eh, CURLINFO_PRIVATE, &b );
43       write( outputfd, b, BUF_SIZE );
44     }
45     curl_multi_remove_handle( cm, eh );
46     curl_easy_cleanup( eh );
47   }
48
49   for (int i = 0; i < argc - 1; i++) {
50     free( buffers[i] );
51   }
52   free( buffers );
53   curl_multi_cleanup( cm );
54   return 0;
55 }
```

4.2 Glory to the Emperor! [11 marks]

Recall that in Byzantium, lieutenants and generals are sometimes... unreliable. If our system has three lieutenants and one general, it can handle up to one traitor (failed component) and decision-making takes one round. You are to implement the code for a loyal lieutenant in that scenario. The lieutenant function takes these arguments:

1. The command that it received from the general as a `char('a'` or `'r')`,
2. The file descriptor for an open socket to one other lieutenant, and
3. The file descriptor for an open socket to the second other lieutenant.

The lieutenant should send the command it received to both other lieutenants, receive their copies of the command, and then return the majority value of these commands. The other lieutenants might fail to send the command they received, so you need to use `poll()` with a timeout of 1 second to wait to receive from the other lieutenants.

You may assume that both sockets stay open at all times and there there are no errors on the sockets. An unreceived value defaults to `'r'`. If you want `poll()` to skip a `pollfd` structure (for example, because you already got an answer from them), then you can set that structure's `fd` field to `-1`. The event for reading from a socket is `POLLIN`.

```
char lieutenant(char command, int sockfd1, int sockfd2) {




}
```

Extra Space. You may use this area as extra space for any question. Clearly indicate in the original question that your answer continues here. Also indicate here which question is being continued.

Please initial:

Please initial: