Please print in pen:
Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination
Final
Spring 2019
ECE 252

Times: Wednesday 2019-08-14 at 19:30 to 22:00 (7:30 to 10PM)
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 4128158
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 7 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. Do not fail this city.
10. After reading and understanding the instructions, sign your name in the space provided below.

Signature

Please initial:

1 Interprocess Communication with CURL [20 marks]

When using webservices, it is common to use your username and password only when logging in, and afterwards all subsequent requests use a "token". The token is a sequence of characters that contains some encrypted data. For the purpose of this question, assume the length of the token is known to be `TOKEN_LENGTH` characters long, not including the null terminator.

The behaviour of `authenticate` is: perform a HTTP POST operation to `https://example.com/login/{id}` where `{id}` is replaced with the `username` and the body is the provided `password`. So if the user who wants to login has the username `jzarnett`, the HTTP POST request URL is `https://example.com/login/jzarnett` . The response code is HTTP 200 and body of the response is the token, if the username and password combination is valid. Otherwise, the server sends back HTTP code 401 (Unauthorized), with an empty (0-byte) body. If the HTTP response code is anything but 200, the return value of `authenticate` should be a NULL pointer; otherwise, return a pointer to the token.

Complete the `authenticate` function below to implement the behaviour described above using cURL (the synchronous version).

Hints: you can have both a read and a write callback on the same `curl_easy_handle`. You can also use `strlen()` to find the length of a string. Assume that the global initialization and cleanup functions are done elsewhere in the program. You may also assume that the length of the password and length of the token are both small enough that each callback will run only once and all data will be processed in that invocation.

```
const char * BASE_URL = "https://example.com/login/";

size_t read_cb( void *buffer, size_t size, size_t nmemb, void *inputdata ) {









}

size_t write_cb( char *ptr, size_t size, size_t nmemb, void *userdata ) {









}

long get_http_code( curl* eh ) {
   long code;
   curl_easy_getinfo( eh, CURLINFO_RESPONSE_CODE, &code );
   return code;
}

char * authenticate( char* username, char* password ) {
```

```
}
```


Please initial:

2 Tests are for Cowards [13 marks]

Code you write should have unit tests. They are executed using a framework of some sort. Let's consider a super-simple framework. To be generic, tests are defined just by their function signature (see the definition below). Here, the execution function `run_tests` receives an array of function pointers representing the tests, and the execution function just invokes the tests directly. A test returns `true` if it passes; `false` otherwise. Finally, the number of failed tests is returned to the user.

```
typedef bool (*test_fn)( ); /* Define function pointer type for test function */

int run_tests( test_fn * tests, int num_tests ) {
 int failed_tests = 0;
 for ( int i = 0; i < num_tests; i++ ) {
     test_fn f = tests[i];
     if( !f() ) {
       failed_tests++;
     }
  }
  return failed_tests;
}
```

Well-written tests can be run in parallel, because their behaviour should not depend on any external state, including the order in which they run or how many run in parallel. Complete below the `run_tests_parallel` function, which runs the provided tests in parallel. For simplicity, we will write a naive implementation that creates a new thread for each test and leaves it to the operating system to figure out how to schedule it.

```
typedef bool (*test_fn)( ); /* Define function pointer for test function */

void* executor( void* arg ) {









}

int run_tests_parallel( test_fn * tests, int num_tests ) {
    int failed_tests = 0;
    pthread_t * threads = malloc( num_tests * sizeof( pthread_t ) );
```

```
    free( threads );
    return failed_tests;
}
```


Please initial:

3 Synchronization [31 marks total]

3.1 Go Away; We're Closed! [17 marks]

Below is the pseudocode describing the behaviour of the customers and barber in the Barbershop problem.

Customer

```
1. wait( mutex )
2. if customers == n
3.     signal( mutex )
4.     return
5. end if
6. customers++
7. signal( mutex )
8. signal( customer )
9. wait( barber )
10. get_hair_cut()
11. wait( mutex )
12. customers--
13. signal( mutex )
```

Barber

```
wait( customer )
signal( barber )
cut_hair()
```

Remember that the Barber executes in an implicit loop. This behaviour does not cover the possibility that the the shop can sometimes be closed. So, we will add some new rules that need implementing:

1. The shop is initially closed.
2. When the barber starts their day, they open the shop.
3. If a customer shows up when the shop is closed, the customer should leave immediately (using `return`).
4. The barber can tell if it is the end of day by calling the function `bool is_end_of_day()`.
5. If it is the end of the day, the barber closes the shop and kicks out all the waiting customers (rude). He does so by calling `cancel_customers()` (which you can assume resets the state of the `customer` and `barber` semaphores as well as the `customers` variable).

Hint: use `trywait()`, which returns 0 if the semaphore was successfully decremented; nonzero otherwise.

Part 1: The Customer [4 marks] Describe the changes you need to make to the customer to support the new rules as described above. Rather than copying out the pseudocode above, just write any new lines you need and indicate clearly where they are supposed to go.

Part 2: The Barber [5 marks] Now you will need to change the barber's behaviour. Instead of the loop for the barber being implied, now you will need to make it explicit. The barber does not mind sleeping through closing time; if they do, a customer who shows up after closing time wakes the barber up. That customer can get their hair cut and then the barber will do the end-of-day procedure. The semaphores used here should correspond with those used in part 1.

Please initial:

Part 3: Less Rude Barber [8 marks] It is sort of rude for the barber to send people home, hair uncut, after they have waited. Let's replace rule 5 above with this: if it is the end of the day, the barber closes the shop, but will cut the hair of any customer(s) who are still waiting at the time of closing, if any. Once the last waiting customer has had their hair cut, the barber is finally done.

3.2 Synchronization and Concurrency Short Answer [9 marks]

Answer each of the following questions using no more than three sentences. 3 marks each.

1. Signalling for Synchronization: a UNIX signal can be used as a mechanism of synchronization. Assume process B has a signal handler that ignores SIGHUP. Does this code work as intended? Explain your answer.

Process A

```
void do_stuff( pid_t proc_b ) {
  foo( );
  kill( proc_b, SIGHUP );
  bar( );
}
```

Process B

```
void work( ) {
  do_stage( 1 );
  pause( );
  do_stage( 2 );
}
```

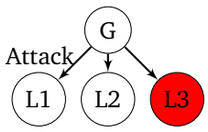
2. What is wrong with the following code? Show a sequence of steps that could cause the problem, and explain how would you fix it.

```
void atomic_increment( int * i ) {
  int old = __sync_add_and_fetch( i, 0 );
  __sync_lock_test_and_set( i, old + 1 );
}
```

3. Helgrind uses the memory address of a `pthread_mutex_t` as its unique identifier. Is this a valid strategy? Explain your answer.

Please initial:

3.3 Generally Speaking [5 marks]



Note: to account for black-and-white printing of the exam, the loyal participants have no shading; the shaded participants are traitors.

Part 1: The simple case [2 marks] Suppose we have the scenario above where we have one disloyal participant. The Emperor has commanded the General to attack. The default action is to retreat. Using one round of vectors, as discussed in the lecture, show how each of the lieutenants arrives at the decision about what to do.

Part 2: Different, Other Treason [3 marks] Suppose the scenario is instead that all 3 lieutenants are loyal, but the general is a traitor. Is one round still enough to decide on a course of action? Justify your answer, considering all possibilities.

4 Shared Memory Producer/Consumer Lab [8 marks]

In lab 3, you implemented a multi-processes program with shared memory to fetch 50 PNG segments from servers in parallel and concatenate the image segments together. The program has four user-specified parameters. B is the maximum number of buffers the program can have to store data received from the server. P is the number of processes that download the data from the servers; C is the number of processes that process the data. The procedure to process data involves having the process sleep for X milliseconds and then inflate the received PNG image IDAT chunk. Assume that for each image segment request, it takes at least 300 ms for the server to respond.

Part 1. [1 mark] If $(B, P, C, X) = (1, 1, 1, 500)$, what is the lower bound of your program execution time?

Part 2. [3 marks] You ran your program with different input parameter values and got the timing results shown in the table on the right, when the server load was light and steady. You find out from Piazza discussions that if the server response time is 200ms per image segment, then the maximum speed up you could achieve is around 3 times. Consider each of the runs above: are they plausible? If not, explain why not.

L#	B	P	C	X	Time (s)
L1	1	1	1	0	15.4
L2	1	5	1	0	5.1
L3	1	1	5	0	15.6
L4	5	1	5	0	15.3
L5	5	5	1	0	5.1
L6	5	5	5	0	5.2

Part 3. [2 marks] During development, you notice that though you put `shmctl()` to remove the shared memory segments in your code, you still have many un-cleaned shared memory segments left on the server after your program terminates. What could be the problem? How do you clean these leftover shared memory segments?

Part 4. [2 marks] You notice that you have many zombie processes running on the server. What could be the reason and how would you prevent this from happening?

Please initial:

5 Deadlock [10 marks total]

5.1 Changing Other Rules [6 marks]

Recall the dining philosophers problem from the lectures: there are five philosophers and five chopsticks. Sometimes, deadlock will not occur if we change the rules. For each of the changes below: can deadlock still occur? Your answer should begin with yes or no, followed by an explanation of why that references the four conditions necessary for deadlock to occur (2 marks each).

1. Philosophers can grab chopsticks out of the hands of colleagues.

2. One of the five philosophers has endless patience and will try to eat rice using only one chopstick.

3. A philosopher can pick up a free chopstick no matter where it is on the table.

5.2 Resource Allocation Graph [4 marks]

Draw the two resource allocation graphs as specified below. Each should have 3 processes and 3 resources.

(i) A graph with a deadlock involving all processes.

(ii) A graph with a cycle, but no deadlock.

6 Files & File Systems [10 marks total]

6.1 Change is Inevitable [6 marks]

Many programs, especially those without a graphical interface, use configuration files to control various options. These are usually plain-text (or XML) formatted on disk. One thing you might like to do is use `inotify` to observe changes to the file and, if the file is written to, reload the configuration file. Complete the code below so that you watch for modification (use the constant `IN_MODIFY` for the `mask`). For the sake of simplicity, assume there will never be more than one event at a time, but note that if the thread is going to terminate then a read will return 0 as no data was successfully read.

```
void* file_watcher( void* arg ) {
  char* config_filename = (char*) arg;
  int buffer_size = sizeof( struct inotify_event ) + strlen( config_filename ) + 1;




  while( !quit ) {




  }
  /* cleanup goes here */


  free( buffer );
  pthread_exit( 0 );
}
```


Please initial:

6.2 Lock Your Vehicle [4 marks]

Complete the code below to write lock the entire file using the `struct flock` approach. Hint: if you set the length of the segment to 0, that locks the whole file. Return -1 if an error occurs; otherwise return the file descriptor.

```
int open_and_lock( char* filename ) {
  int fd = open ( filename, O_RDWR );
  struct flock fl;









}
```

7 Asynchronous I/O [28 marks total]

7.1 Sockets and Asynchronous I/O [25 marks]

While users do not like them very much, ads are a very common part of the user experience on the internet. The company hosting the content (e.g., music or video) gets paid some small fee for each ad they annoy their users with. When your company's service is preparing to show an ad, it asks a set of advertisers to bid within a certain timeframe. The highest bid is chosen. If no bids are received before the timeout elapses, then the company will play an ad for the ad-free version of its service. The ad content is already hosted by your company, so all the function needs to return is the unique integer ID of the advertiser whose ad is selected (0 for the company's own ad).

The function you will implement gets an array of `struct advertiser` as defined below, as well as a parameter `length` telling you the length of this array.

```
struct advertiser {
  struct addrinfo * addr; /* The address information (returned from getaddrinfo()) for this advertiser */
  int id; /* This advertiser's unique ID */
  /* Other properties not needed in this question not shown */
}
```

Your function should establish socket communication with each advertiser (your code acts as the client). If a connection can be established (i.e., there's no error in creating the socket) then `send()` the `service_id` value (identifies your program) and the `user_id` (identifies the user) in that order; you may safely assume the two integers can be sent without having to use `sendall()` functionality. Sending may be done sequentially.

After sending the request to each advertiser, they will send back responses. Responses may come in any order, or not come at all. Therefore, you should use `select()` to manage the sockets established earlier. If a socket is ready, `recv()` an integer value from it – this represents that company's bid in 1000ths of a cent. If you got an answer from a particular advertiser, you may close that socket immediately.

The overall timeout is 1 second. When select returns, this system's implementation of `select()` has updated the `struct timeval` to show the time remaining before the timeout. So if it has an initial value of 1 second and an answer comes in after 50 ms, then the `struct timeval` will contain the value of 950 ms.

After 1 second of waiting for responses has elapsed (i.e., the returned `timeval`'s values are both zero), you may close the sockets and clean up. Return the unique ID of the advertiser with the winning bid.

Hint: remember that byte order is important. Assume integers are 32 bits (4 bytes) in this system.

Use the function below to create a socket given an address.

```
int open_socket( struct addrinfo * addr ) {
  int socket = socket( addr->ai_family, addr->ai_socktype, addr->ai_socktype );
  if ( socket == -1 ) {
    int status = connect( socket, addr->ai_addr, addr->ai_addrlen );
    if ( status != 0 ) {
      socket = -1;
    }
  }
  return socket;
}
```


Please initial:

```
int determine_ad_id( int service_id, int user_id, struct advertiser * advertisers, int length ) {
  int * sockets = send_requests( service_id, user_id, a, length );
  int result = get_responses( sockets, a, length );
  return result;
}

/* Send outgoing requests; returns an array of sockets (with capacity length) */
/* To preserve the size, put -1 in the array if there's a problem with the socket */
/* Caller responsible for closing sockets and deallocating the returned array */
int* send_requests( int service_id, int user_id, struct advertiser * a, int length ) {









}

/* Receive incoming responses: takes the socket array from the send_requests function */
int get_responses( int* sockets, struct advertiser * a, int length ) {









}
```

7.2 AIO [3 marks]

Consider the two implementations of AIO control blocks below. Assume all arrays and variables have been defined and initialized correctly. Which approach is more efficient, and why?

```
for ( int i = 0; i < length; i++ ) {
  aiocbs[i].aio_nbytes = 1024;
  aiocbs[i].aio_filedes = fds[i];
  aiocbs[i].aio_offset = 0;
  aiocbs[i].aio_buf = buffers[i];
  aio_read( &aiocbs[i] );
}
aio_suspend( aiocbs, length, NULL );
```

```
for ( int i = 0; i < length; i++ ) {
  aiocbs[i].aio_nbytes = 1024;
  aiocbs[i].aio_filedes = fds[i];
  aiocbs[i].aio_offset = 0;
  aiocbs[i].aio_buf = buffers[i];
  aiocbs[i].aio_lio_opcode = LIO_READ;
}
lio_listio( LIO_WAIT, aiocbs, length, NULL );
```


Please initial:

Please initial: