Please print in pen:

Waterloo Student ID Number:

WatIAM/Quest Login Userid:

UNIVERSITY OF WATERLOO

Examination Final Fall 2022 ECE 252

Times: Wednesday 2022-12-21 at 09:00 to 11:30
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 4992771
Sections: ECE 252 LEC 001,002
Instructors: Jeff Zarnett

Special Materials

Candidates may bring only the listed aids.

· Calculator - Non-Programmable

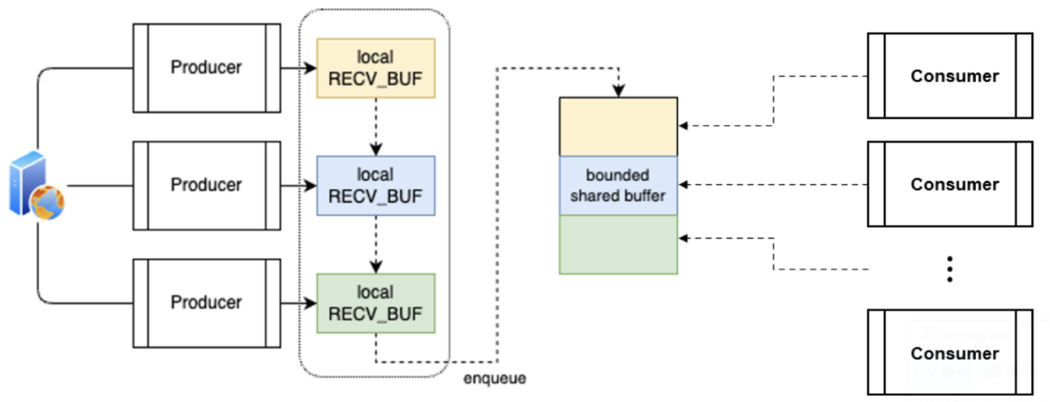
Instructions:

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 four (4) questions, some with multiple parts. Not all are equally difficult.
5. The exam lasts 150 minutes and there are 100 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. After reading and understanding the instructions, sign your name in the space provided below.

Signature

1 About Lab 3... [10 marks]

In Lab 3, you are asked to solve a producer-consumer problem by using shared memory. To recap, the producers fetch image strips concurrently from a web server and the consumers process those image strips. The producers and consumers communicate through shared memory. One approach is to make producers to fetch image strips in parallel and save them into local buffers as shown below.



Identify one memory inefficiency of this design and propose a better approach. Use a diagram to illustrate your design and pseudocode to explain the use of synchronization constructs to control access to the shared buffer.

2 Concurrency and Synchronization [33 marks total]

2.1 I have to go... in person? [18 marks]

Recently, I needed to go to a Service Ontario location. This resembles the barbershop problem, except instead of getting my hair cut, I'm doing paperwork. Instead of Customer and Barber, we'll call the parties here Requester and Worker, and instead of `get_hair_cut()` it will be `request_help()` and rather than `cut_hair()` the worker will execute `help_requester()`. When I went there, if the lineup was too long I would have left, so that part remains. Here's the pseudocode for the behaviour, modified with the new names:

Requester

```
wait( mutex )
if requesters == n+1
    signal( mutex )
    return
end if
requesters++
signal( mutex )

signal( requester )
wait( worker )
request_help()

wait( mutex )
requesters--
signal( mutex )
```

Worker

```
wait( requester )
signal( worker )
help_requester()
```

While I was waiting in line, I observed a difference from the barbershop behaviour: an elderly person arrived with some mobility assistance. When that happened, the workers prioritized that person over everyone else waiting in line, meaning there exists a concept of priority here. In this problem, we'll expand on the barbershop solution to have two kinds of requesters: Priority Requesters and Other Requesters. The behaviour of the different thread types is similar as above, but note the following:

Workers will choose to help a Priority Requester first if one is present. If there are none, then they help one of the Other Requesters.

Priority Requesters behave like Other Requesters, except they are unusually patient and will not leave no matter how many people are currently waiting.

The behaviour of Other Requesters is the same as in the solution where there is no priority given to Priority Requesters. Priority Requesters count towards the number of requesters when the Other Requester decides if they will leave. If Priority Requesters arrive while they are already waiting, Other Requesters will continue to wait.

Write down the pseudocode for all three types of thread below to implement the above behaviour.

Hint: It makes sense to reverse some of the usages of the semaphores when compared to above. For example, the worker thread could wait on a semaphore called `worker` which is used by other kinds of threads to wake up a worker.

Other Requester **Priority Requester** **Worker**

2.2 Snooze Button [6 marks]

Recall this code from lectures (slightly modified for space) where we discussed the usage of the `inotify` strategy to use the existence of a file as a lock. The code as written here suffers from the possibility of the *lost wakeup* problem, which is to say that there is a possibility that a process that is waiting for the deletion of the file might be waiting indefinitely. Analyze the code below, and explain (1) a scenario where a process can get stuck indefinitely, and (2) how you would fix it. The explanation should reference specific line numbers to explain the problem and where any added/changed/removed code should be. You do not need to write any code or pseudocode, just give a written explanation of how to address it.

```
 1  int lockFD;
 2  bool our_turn = false;
 3  while( !our_turn ) {
 4      lockFD = open( filename, O_CREAT | O_EXCL | O_TRUNC );
 5      if ( lockFD == -1 ) {
 6          printf( "The␣lock␣file␣exists␣and␣process␣%ld␣will␣wait␣its␣turn...\n", getpid() );
 7          int notifyFD = inotify_init( );
 8          uint32_t watched = inotify_add_watch( notifyFD, filename, IN_DELETE_SELF );
 9
10          int buffer_size = sizeof( struct inotify_event ) + strlen( filename ) + 1;
11          char* event_buffer = malloc( buffer_size );
12          printf("Setup␣complete,␣waiting␣for␣event...\n");
13          /* Read the file descriptor for the notify -- we get blocked here until there's an event that we want */
14          read( notifyFD, event_buffer, 1 );
15
16          struct inotify_event* event = (struct inotify_event*) event_buffer;
17          /* Here we can look and see what arrived and decide what to do. In this example, we're only watching one
18          file and one type of event, so we don't need to make any decisions now */
19          printf("Event␣occurred!\n");
20
21          free( event_buffer );
22          inotify_rm_watch( lockFD, watched );
23          close( notifyFD );
24      } else {
25          char* pid = malloc( 32 );
26          memset( pid, 0, 32 );
27          int bytes_of_pid = sprintf( pid, "%ld", getpid() );
28
29          write( lockFD, pid, bytes_of_pid );
30          free ( pid );
31          close( lockFD );
32          our_turn = true;
33      }
34  }
```

2.3 Yes, Supervisor [9 marks]

In this question, you will use a condition variable to make it so that `main()` will not proceed to call `clean_up()` until `tasks_done` has reached `total_tasks`. You can assume that the condition variables are initialized correctly and that worker threads are created correctly as well. You may ignore error handling. Complete the code below.

```
pthread_cond_t cv;
pthread_mutex_t lock;
int tasks_done = 0;
int total_tasks = 10;

int main( int argc, char** argv ) {
  init( ); /* Initialize cv, lock */
  start_workers( ); /* Start N worker() threads */




  clean_up();
  return 0;
}
```

```
void* worker( void* task ) {
  do_task( task );




  pthread_exit( NULL );
}
```

3 This Exam brought to you by... [17 marks]

Users don't like them, but ads are part of the user experience on the internet. The company hosting the content (e.g., video) gets paid a fee for each ad they annoy their users with. When the 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. In this question, you will implement code for the ad company receiving requests to quote a price for the ad.

Implement the function `loop` to have the following behaviour. When you get an incoming connection, create a new detached thread to handle this request. The new thread receives the 4-byte integer `service_id` from the client using a blocking receive call. Using the ID, it will call `get_current_price()` to get the price we are willing to pay. Then, send the price to the requester and close the socket. You may skip error checking in this question.

```
void* handle_request( void* a ); /* Function Prototype */

/* Assume that the socket is the socket we have already done bind() and listen() on and it is ready */
void loop( int socket ) {
  while( !quit ) {










  }
}

/* Returns the current price, in 1000ths of a cent, that we are willing to pay  to the service identified by the
     service_id parameter */
int get_current_price( int service_id );

void* handle_request( void* a ) {










}
```

4 Lumberjack as a Service [40 marks total]

In this multi-part question, we will be building up a service that handles logging. Logging, in the computer sense, is an important part of understanding the execution of the program. We are making a service that observes events, and generates log events, then writes them to a log. This will combine inter-process communication, the producer-consumer problem, and asynchronous I/O. It is best to read the whole question before starting any subparts to prevent needing to go back and revise things already written. In this question, you may skip error checking.

There will be one producer and one consumer thread in the program. The producer will observe events, generate log messages of the type `log_msg` (see definition below), and put them in the shared memory array (if there's space). The consumer will take those messages and then write them to a log, asynchronously.

```
typedef struct {
  char message[MAX_LENGTH];
  int message_length;
} log_msg;
```

4.1 Setup of IPC [15 marks]

We are going to create some shared memory, and it will have the synchronization constructs we need as well as the shared array. Define below your structure `shared` that will contain any shared synchronization constructs and an array of size `ARRAY_SIZE` which you can assume is a global constant.

Then, complete the function `create_and_attach_shmem()` which, as it sounds, creates (with `IPC_PRIVATE` and permissions `0666`) and attaches to shared memory. Then, complete `initialize_shared` to initialize the shared memory as needed for future use, including an empty array.

```
#define ARRAY_SIZE 100
typedef struct {







} shared;

typdef struct {
  int shmid;
  void* mem;
} shmem;


shmem* create_and_attach_shmem( ) {                    void initialize_shared( shared* s ) {




                                                       }
}
```

4.2 Producing and Consuming [15 marks]

Now, implement the code for the producer and consumer threads. Remember that producers generate events from the call to get_next_event() and then add it to the shared memory. Consumers take items from shared memory and should call the function:

```
void write_to_log_file( int log_file_descriptor, char* log_line, int log_line_length );

void* producer( void* arg ) {
  shared* s = (shared*) arg;
  int current = 0;

  while( !quit ) {
    log_msg message = get_next_event( );









  }
  pthread_exit(NULL);
}

void* consumer ( void* arg ) {
  shared* s = (shared*) arg;
  int output_file = get_output_file_descriptor();




  while( !quit ) {









   }
  pthread_exit(NULL);
}
```

4.3 Writing the Log File with AIO [10 marks]

Finally, there is the implementation of writing the log line to the the log file. For this, you should use the POSIX AIO approach (`struct aiocb`) to write the content to the file. The function signature below provides the file descriptor to write to (assume the file is already open for appending), the log line to write, and the length of the log line. Use AIO to append the log to the file.

When finished writing, the provided log line is no longer needed and can be deallocated. Remember that you need to call `aio_return` after the AIO is done. You should configure the AIO so that a new thread is created that runs the function `cleanup` below that cleans up after the operation is done. You do not need to close the log file after the write; the log remains open for writing as long as the process executes.

```
void cleanup( union sigval arg ) {








}


void write_to_log_file( int log_file_descriptor, char* log_line, int log_line_length ) {








}
```