Please print in pen:
Waterloo Student ID Number:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

WatIAM/Quest Login Userid:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | |

UNIVERSITY OF WATERLOO

# Examination
# Final
# Fall 2017
# ECE 254

Times: Monday 2017-12-11 at 09:00 to 11:30
Duration: 2 hours 30 minutes (150 minutes)
Exam ID: 3596173
Sections: ECE 254 LEC 001,002
Instructors: Carlos Moreno, 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 six (6) questions, 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. A reference sheet is attached as the last page of the examination.
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 |
|---|---|---|---|---|---|---|---|---|
| 1a | | 6 | 3a | | 4 | 5a | | 19 |
| 1b | | 6 | 3b | | 12 | 5b | | 5 |
| 2a | | 5 | 3c | | 9 | 5c | | 6 |
| 2b | | 8 | 4a | | 4 | 6a | | 3 |
| 2c | | 8 | 4b | | 9 | 6b | | 12 |
| | | | 4c | | 4 | | | |
| **Total** | | | | | | | | **120** |

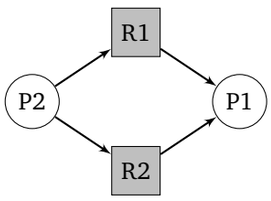


Please initial:

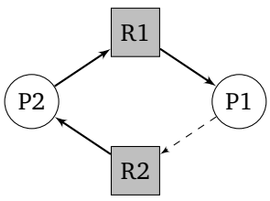
# Question 1: Deadlock [12 marks total]

## 1A: Deadlock Detection [6 marks]

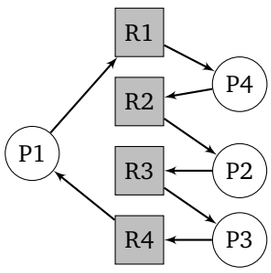
Consider the four (4) resource allocation graph diagrams below. In each of them evaluate: Is there a deadlock (0.5 marks)? If there is, list the processes involved the deadlock; If not, indicate what single resource request, if any, would cause a deadlock (1 mark).



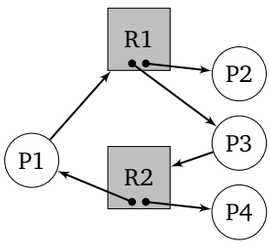

(a)




(b)




(c)




(d)

(a)

(b)

(c)

(d)

## 1B: Stop Deadlock Before it Starts [6 marks]

You have discovered a deadlock in your multithreaded code. You could solve it using one of the following three strategies that we have discussed in class. Describe each strategy and what the advantages and disadvantages are.

1. Data Copying

2. Lock Ordering

3. Two-Phase Locking



Please initial:

# Question 2: Memory [21 marks total]

## 2A: Virtual Memory (1) [5 marks]

Explain how a logical address is transformed into a physical address when the system uses virtual memory and a page table, and has a one-level translation lookaside buffer (TLB).

## 2B: Virtual Memory (2) [8 marks]

Consider a virtual memory scheme that uses segmentation combined with paging. We have a physical address space of 32 bits and a virtual address space of 40 bits. Page size is 4 KB, and we want a maximum segment size of 16 MB.

**Part 1.** Show and explain the layout of a virtual address — that is, the breakdown of the 40 bits into segment and/or page numbers, offsets, etc.

**Part 2.** For this virtual memory scheme, at some point in time, we have the following segment table and page tables for the process currently executing:

| Segment # | Segment Length | Base Address |
|---|---|---|
| 0 | 0x3F00 | 0x1000 |
| 1 | 0x27F0 | 0x2000 |

**Page Table for Segment 0**

| Page # | Present | Frame # |
|---|---|---|
| 0 | 0 | – |
| 1 | 1 | 9 |
| 2 | 1 | 7 |
| 3 | 1 | 4 |

**Page Table for Segment 1**

| Page # | Present | Frame # |
|---|---|---|
| 0 | 0 | – |
| 1 | 0 | – |
| 2 | 1 | 3 |
| 3 | 1 | 5 |

Assuming both page tables are in memory, complete the table below. To do so, determine the physical address corresponding to the virtual addresses, and indicate whether the given virtual address causes a segmentation fault or a page fault.

| Virtual Address | Physical Address | Seg. Fault? | Page Fault? |
|---|---|---|---|
| 0x00000020FF | | | |
| 0x00000032FF | | | |
| 0x0001003A00 | | | |
| 0x00010007FF | | | |



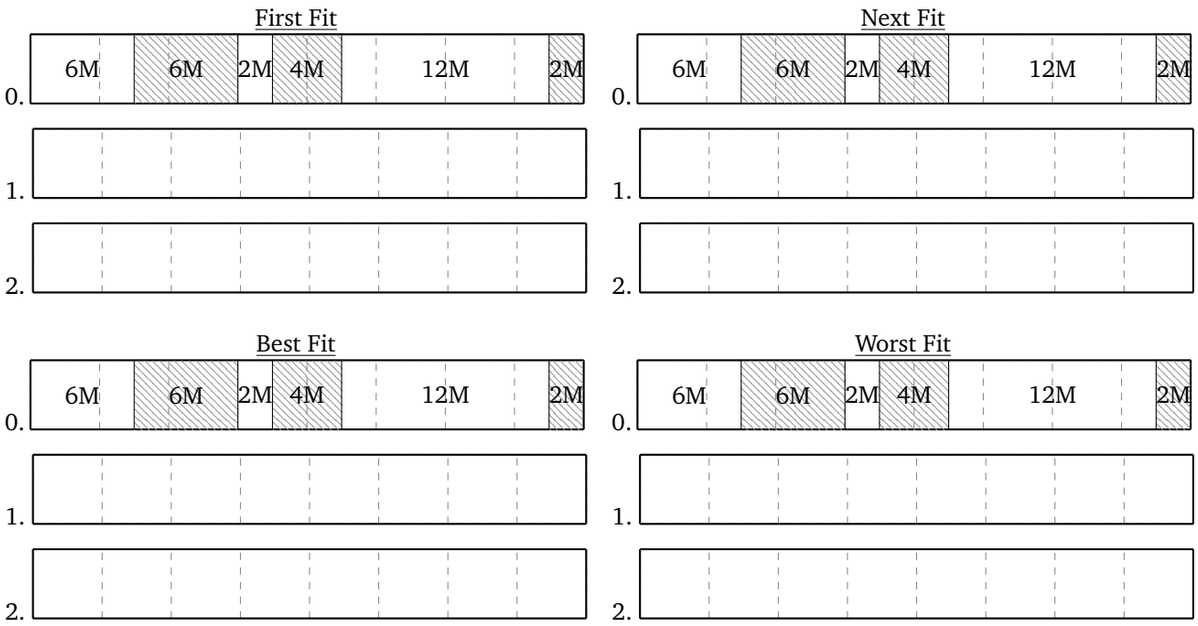
Please initial:

## 2C: Dynamic Memory Allocation [8 marks]

The diagrams below show a 32 MB block of memory used to fulfill memory allocations in this question. Use shading to indicate allocated blocks and also show the size of each block. Grey dashed lines are guidelines in increments of 4 MB to assist you in drawing the blocks in the correct sizes. The initial state of the system is shown as step 0.

Perform the following allocations using the algorithms *first fit*, *next fit*, *best fit*, and *worst fit* below under their appropriate labels. If an allocation cannot be performed, write that in the box and cease execution of the algorithm. The most recently allocated block is the 2 MB block.

1. Allocate 2 MB
2. Allocate 6 MB






Please initial:

# Question 3: Scheduling [25 marks total]

## 3A: The Linux Completely Fair Scheduler [4 marks]

Explain the functionality of the following elements in the Linux Completely Fair Scheduler:

1. The `vruntime` value of a process

2. Red-black tree

3. The `nice` value of a process

4. Target latency

## 3B: Scheduling Implementation [12 marks]

Consider the following short-term scheduling algorithm (dispatcher) described in C:

```
struct task_descriptor {
  int          task_id;
  unsigned int priority;   /* lower values represent higher priority */
  int          ready;      /* a value of 0 means blocked (not ready);
                                  any non-zero value means ready */
  /* ... additional data ... */
};

void dispatcher (struct task_descriptor tasks[], int num_tasks) {
  int i;
  int selected = 0;
  for (i = 1; i < num_tasks; ++i) {
    if (tasks[i].ready && tasks[i].priority < tasks[selected].priority) {
      selected = i;
    }
  }
  dispatch (&tasks[selected]);
}
```

Explain why this scheduling algorithm is flawed. There are two distinct flaws; you should name and explain both.



Please initial:

## 3C: Scheduling Implementation (Continued) [9 marks]

Despite being flawed, the algorithm described in 3B can work for certain cases For each of the following cases (pairs of tasks — assume that these two are the only tasks/processes executing), explain whether the above scheduling algorithm will work or explain what the problem will be. If a function is called, the name of the function indicates what the function does (make any reasonable assumptions about how the function works; state any assumptions you feel need to be made explicit):

**Case 1**

Task #1 (index 0 in array tasks; priority = 1)

```
void task1() {
  /* seed is a global variable*/
  int pseudorandom = seed;
  while (1) {
    /* A and B are some #defined constants */
    pseudorandom = pseudorandom * A + B;
    /* random_source is a global variable */
    random_source = pseudorandom;
  }
}
```

Task #2 (index 1 in array tasks; priority = 0)

```
void task2() {
  while (1) {
    char data[256];
    int conn = connect_to_server("x.uwaterloo.ca");
    read_data (conn, data, 256);
  }
}
```

**Case 2**

Exactly like case 1, except that Task #1 has priority 0 and Task #2 has priority 1.

**Case 3**

Task #1 (index 0 in array tasks; priority = 0)

```
void task1() {
    /* code identical to task 1 in Case 1; only the
       code is identical: notice that the priority
       is different w.r.t. Task #1 priority
       in Case 1 */
}
```

Task #2 (index 1 in array tasks; priority = 0)

```
void task2() {
  int x = 0;
  while (1) {
    ++x;
    if (x == 0) {
      flag = 1;   /* flag is a global variable */
    }
  }
}
```



Please initial:

# Question 4: I/O & Files [17 marks total]

## 4A: Opening and Closing Files [4 marks]

Some systems automatically open a file when it is referenced for the first time, and automatically close the file when the process terminates. Give two (2) advantages and two (2) disadvantages of this approach compared to the UNIX approach with the open and close system calls.

## 4B. Disk Scheduling [9 marks]

Consider a disk that has 500 cylinders labelled 0 through 499. Suppose the incoming disk reads waiting to be serviced are: `490, 142, 163, 19, 168, 167, 36`. The starting position of the disk read head is 52 and it is moving in the ascending direction. Complete the table below. Improvement is measured as FCFS cylinders moved divided by the alternative strategy's cylinders moved. When reporting the improvement over FCFS, round to 3 decimal places.

| **Algorithm** | **Service Order** | **Cylinders Moved** | **Improvement over FCFS** |
|---|---|---|---|
| First-Come First Serve (FCFS) | 490, 142, 163, 19, 168, 167, 36 | | 1.000 |
| Shortest Seek Time First (SSTF) | | | |
| SSTF + Double Buffering, Buffer size 3 | | | |
| SSTF + Double Buffering, Buffer size 5 | | | |
| SCAN | | | |

## 4C: NTFS File System [4 marks]

Recall from lectures the NTFS file system. Explain how journalling is used to ensure that the file system is always in a consistent state, even if a crash or power outage occurs.



Please initial:

# Question 5: Concurrency & Synchronization [30 marks total]

## 5A: Search-Insert-Delete [19 marks]

This is an extension of the readers-writers problem called the search-insert-delete problem. Instead of two types of thread, reader and writer, there are three types of thread: searchers, inserters, deleters. They operate on a shared linked list of data. Your code does not need to manipulate the list directly; instead, you must write the synchronization code and call the list-manipulation functions (named in the paragraphs below) at the right times.

*Searchers* merely examine the list; hence they can execute concurrently with each other. Searcher threads must call `void search( void* target )` where the argument to the searcher thread is the element to be found. These most closely resemble readers in the readers-writers problem.

*Inserters* add new items to the end of the list; only one insertion may take place at a time. However, one insert can proceed in parallel with any number of searches. Inserter threads call `void insert( void* to_insert )` where the argument to the inserter thread is the element to be inserted. Assume `insert` is written so the insertion can be done in parallel with the searches. Inserters resemble readers with an additional rule that only one of them can manipulate the list at a time.

*Deleters* remove items from anywhere in the list. At most one deleter process can access the list at a time, and when the deleter is accessing the list, no inserters and no searchers may be accessing the list. Deleter threads call `void delete( void* to_delete )` where the argument to the deleter thread is the element to be deleted. These most closely resemble writers in the readers-writers problem.

Hint: below is some pseudocode giving an outline (but not all the details) of the readers-writers solution:

```
wait( roomEmpty )
// write
signal( roomEmpty )
```

```
if first reader
  wait( roomEmpty )
// read
if last reader
  signal( roomEmpty )
```

Complete the code below, including the `init` function to initialize the global variables to the correct values.

```
/* Global Variables are Below */
pthread_mutex_t searcher_mutex;
pthread_mutex_t inserter_mutex;
pthread_mutex_t perform_insert;
sem_t no_searchers;
sem_t no_inserters;
int searchers;
int inserters;

void init( ) {









}

void* searcher_thread( void *target ) {









}
```

```
void* deleter_thread( void* to_delete ) {









}

void* inserter_thread( void *to_insert ) {









}
```



Please initial:

### 5B: Swap Meet! [5 marks]

The structure `container_t` is defined as follows:

```
typedef struct container {
   pthread_mutex_lock lock;
   double data;
} container_t;
```

A swap function is defined below:

```
1  void swap( container_t *x, container_t  *y ) {
2    pthread_mutex_lock( &x->lock );
3    double temp = x->data;
4    pthread_mutex_lock( &y->lock );
5    x->data = y->data;
6    pthread_mutex_unlock( &x->lock );
7    y->data = temp;
8    pthread_mutex_unlock( &y->lock );
9  }
```

At lease one concurrency problem exists in this code. Pick one, explain what it is, and describe a scenario that could cause it.

### 5C: The Dining Philosophers [6 marks]

Recall the dining philosophers problem from the lectures. The standard example scenario has five philosophers and five chopsticks and has the possibility of deadlock. We have also seen, however, that sometimes deadlock will not occur if we change the rules. For each of the changes below, determine if it individually is enough to make sure a deadlock cannot occur, and explain why.

1. Philosophers choose at random which chopstick to get first

2. One (1) additional chopstick is added to the table

3. Philosophers may steal chopsticks out of the hands of another



Please initial:

# Question 6: Security [15 marks total]

## 6A: Isolation [3 marks]

One of the computer security principles is that of isolation / compartmentalization; this is the practice of breaking an application into "isolated" sections or modules to help minimize or contain the damage if and when some attack is successful in hijacking the functionality of some section the application. For each of the following approaches, explain whether or not it constitutes an effective application of the above principle:

1. Object-oriented design with emphasis on encapsulation (assuming that the application is written in C++)

2. Multithreading

3. Multiple processes

## 6B: Disk Encryption [12 marks]

In a system with user files encryption, we saw that we need to use the login password to encrypt the files, but not directly: the system encrypts the files with a key $K$, and $K$ is encrypted using the user password as the encryption key. This encrypted $K$ is stored somewhere on the disk as file `encrypted-key`. When the user changes password, the system decrypts the file `encrypted-key` with the old password, re-encrypts it with the new password and saves it.

The above scheme, however, is not robust with respect to interruption of the procedure. For example, if the power fails and the machine turns off half-way through the procedure of changing the password and re-encrypting — since the system also needs to update the user's password in the file /etc/shadow, the following sequence of events could cause the files to be effectively lost: the system stores the new password and then the procedure is interrupted before it re-encrypts the file encrypted-key; or, the system re-encrypt first and power goes off before it updates the file /etc/shadow. In both cases, the login password and the encryption password for the file encrypted-key are different.

To solve this, we implement the following password-changing procedure:

- Create new file `encrypted-key.new` with the key $K$ encrypted with the new user password.
- Change password in the `/etc/shadow` file.
- Erase old file (encryption key encrypted with the old user password).
- Rename file `encrypted-key.new` to `encrypted-key`.

The above procedure is not atomic, and thus fails if there is more than one instance of the user logged in and concurrently changing the password (you should assume that this can happen).

**Part 1.** Write the code to avoid race conditions *without using mutexes, semaphores, or busy-waiting*. Instead, you should use atomic file operations. In addition to avoiding the race condition, you want to cancel the attempt that arrives second. That is: when attempting to change the password, if the program detects that a password-changing procedure is already taking place, the program should cancel the operation and inform the user that the password is already being changed by someone else.



Please initial:

For the sections of your answer that require reading from a file or writing to it, you may use pseudocode (including writing data to `/etc/shadow`). For encryption, assume you have a function encrypt and a function decrypt with prototypes as shown below:

```
void encrypt (const char * plaintext, const char * key, char * ciphertext);
void decrypt (const char * ciphertext, const char * key, char * plaintext);
```

In both cases, the first two parameters are input parameters and the last one is where the function writes the result. The functions assume that the given pointer points to an area in memory with sufficient space to write the result. You may assume that the encryption has the property that the ciphertext and plaintext are equally long (i.e., if you pass $N$ characters of plaintext, the output will be $N$ characters of ciphertext and vice-versa).

For erasing and renaming files, see description for functions `remove` and `rename` on the reference sheet.

**Part 2.** Write the pseudocode for the algorithm that handles the file decryption when a given user logs in, taking into account the possibility of an interrupted session (e.g., the machine was powered off half-way through a password change for that user).



Please initial:

# Reference Sheet

Assume always the C99 standard (e.g., you can declare an integer in the same line as the for statement).

Memory is allocated in C with `malloc()` and to get the size of memory you want to allocate, there is `sizeof`, normally used in conjunction with `malloc`. Example: `int* p = malloc( sizeof( int ) );`

Memory is deallocated using `free`. Example: `free( p );`

An argument can be converted to an integer using the function `int atoi( char* arg )`.

Printing is done using `printf` with formatting. `%d` prints integers; `%lu` prints unsigned longs; `%f` prints double-precision floating point numbers. A newline is created with `\n`.

Some UNIX functions you may need:

```
pid_t fork( )
pid_t wait( int* status )
pid_t waitpid( pid_t pid, int status )
int kill( pid_t pid, int signal ) /* returns 0 returned if signal sent, -1 if an error */

int open(const char *filename, int flags);  /* Returns a file descriptor if successful, -1 on error */
ssize_t read(int file_descriptor, void *buffer, size_t count); /* Returns number of bytes read */
ssize_t write(int file_descriptor, const void *buffer, size_t count); /* Returns number of bytes written */
int rename(const char *old_filename, const char *new_filename); /* Returns 0 on success , operates atomically */
int remove(const char *filename) ; /* Deletes a file or directory, returns 0 on success, operates atomically */
int close(int file_descriptor);
```

When opening a file the following flags may be used for the `flags` parameter (and can be combined with bitwise OR, the `|` operator):

| Value | Meaning |
|---|---|
| `O_RDONLY` | Open the file read-only |
| `O_WRONLY` | Open the file write-only |
| `O_RDWR` | Open the file for both reading and writing |
| `O_APPEND` | Append information to the end of the file |
| `O_TRUNC` | Initially clear all data from the file |
| `O_CREAT` | Create the file |
| `O_EXCL` | If used with `O_CREAT`, the caller MUST create the file; if the file exists it will fail |

For your convenience, a quick table of the various pthread and semaphore functions we have discussed:

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **returnValue )
pthread_detach( pthread_t thread )
pthread_cancel( pthread_t thread )
pthread_testcancel( ) /* If the thread is cancelled, this function does not return (thread terminated) */
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_trylock( pthread_mutex_t *mutex ) /* Returns 0 on success */
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )

pthread_cond_broadcast( pthread_cond_t * condition_variable );
pthread_cond_destroy( pthread_cond_t * condition_variable );
pthread_cond_init( pthread_cond_t * condition_variable, const pthread_condattr_t * attributes );
pthread_cond_signal( pthread_cond_t * condition_variable );
pthread_cond_wait( pthread_cond_t * condition_variable, pthread_mutex_t * mutex );
pthread_condattr_init( pthread_condattr_t * attributes );
pthread_condattr_destroy( pthread_condattr_t * attributes);

sem_init( sem_t* semaphore, int shared, int initial_value); /* 0 for shared OK */
sem_destroy( sem_t* semaphore )
sem_wait( sem_t* semaphore )
sem_post( sem_t* semaphore )
```



Please initial: