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, 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
1.1		20	3.1		9	5.1		7
1.2		15	3.2		5	5.2		7.5
1.3		6	3.3		11	5.3		4.5
2.1		5	4.1		15			
2.2		5	4.2		6			
2.3		4						
Total								**120**

1 Concurrency and Synchronization [41 marks total]

1.1 Accreditation is Super Inconvenient [20 marks]

Professor Derek Wright still needs your help; the department is still working on the accreditation report. Recently, Professor Aagaard has assembled a database that contains accreditation data. This data includes information about how many AUs a course has in a given term. These can change over time, as the curriculum is updated based on student feedback (and accreditation needs), so the term information is important. Professor Wright has a program that lets him find out the total number of AUs for a given set of courses by querying the database Professor Aagaard has set up. A sequence of course-term pairs is provided as input, and the total AUs is printed out.

Input is parsed using a function `course_term* get_next( )` which returns a pointer to a structure of type `course_term` which is comprised of the course number and the term (see the definition of the structure below). If we've reached the end of input, this function returns `NULL`. The implementation of this function is not shown, but assume it works correctly. This function cannot be run in parallel.

```
typedef struct {
  char course[8]; /* string representation, e.g. ECE254 */
  int term; /* integer, e.g., 1189 for Fall 2018 */
} course_term;
```

To find the number of AUs, there is the function `double query_db( course_term * ct )` . This queries the database for the number of AUs for the specified course in the specified term which is returned as a `double`. Again, the implementation is not shown, but assume it works correctly. The database query is read-only, and the database server is powerful enough, so it can be run in parallel.

Here's the simple implementation of the program:

```
course_term* get_next( );
double query_db( course_term* ct );
double total = 0.0;

int main( int argc, char** argv ) {
  while( 1 ) {
    course_term* next = get_next( );
    if ( next == NULL ) {
      break;
    }
    total += query_db( next );
    free( next );
  }
  printf( "Total AUs: %g\n.", total );
  return 0;
}
```

This program is too slow. You have agreed to rewrite the program to use pthreads to do work in parallel. Rewrite the program to use threads such that each thread carries out one query on the database. Your modified program should be free of race conditions, not have any memory leaks, and produce the same output as the serial program. For your convenience, a structure called `node` has been defined for you, so that you can keep track of the threads in the program. Complete the code below to implement the new behaviour.

```
typedef struct node {
  pthread_t t;
  struct node * next;
} node;

course_term* get_next( );
double query_db( course_term* ct );

node * head;
double total = 0.0;
/* Any other global variables you need go here */




void* query( void* arg ) {









}
```

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









}
```

1.2 The Producer-Consumer Problem [15 marks]

More Threads (3 marks). A simple way to increase performance of a system is to add more threads. Suppose the buffer size is fixed. For each part below, your answer should include a one sentence explanation.

1. Will increasing only the number of producers help?

2. Will increasing only the number of consumers help?

3. Can we increase both producers and consumers always?

Partial Consumers (12 marks). The producer/consumer problem in Lab 3 had fixed roles where a thread/process was only a producer or only a consumer. In modern frameworks, a thread/process does one step of the work and then passes it on to another thread/process that will do the next step. Having multiple consumer/producers allows for plug-and-play combinations that are easier to read and more portable across applications.

For this question, you will implement one type of consumer/producer thread that takes an item from its input queue, does some work on it with `do_work`, and then places the modified item in the output queue. Other threads are responsible for putting items into the input queue as well as taking items out of the output queue.

You should use the datatype `queue`, which is created with a maximum size (capacity) and can perform the actions `push()` and `pop()`. These three queue functions are not thread-safe. Also, you should not call `push()` when there is not enough space, and not call `pop()` when there are no items in the queue. Assume any other consumers, producers, and consumer/producer threads in the system will correctly use all concurrency constructs you have created. Hint: each of your queues needs two semaphores and a mutex.

```
typedef struct {
  /* Implementation not shown */
} item;
tyewdef struct {
  /* Implementation not shown */
} queue;

int quit = 0; /* Will changed to 1 by Ctrl-C Handler */
/* Initialize the queue with a maximum size */
void queue_init( queue *q, int capacity );
/* pushes an item onto a queue */
void push( queue* q, item* i );
/* Returns the oldest item in the queue */
item* pop( queue* q );
/* Perform some work on an item */
item* do_work( item* i );
```

```
/* Input items queue */
queue in_queue;
int in_queue_size = 32;
/* Output items queue */
queue out_queue;
int out_queue_size = 64;

/* Put any other global variables you need here */
```

```
int main( int argc, char** argv ) {
   /* Initialize global variables here */

  /* main() creates threads of different kinds */
  /* main() joins all created threads */
  /* This area runs after all threads have been joined
     Clean up Global Variables */

  pthread_exit( 0 );
}
```

```
void* proconsumer( void* arg ) {
  while( !quit ) {

  }
  return 0;
}
```

1.3 Another Slice of Pizza? [6 marks]

Recall from the midterm exam the Pizza Problem: Each contestant has an unlimited supply of one ingredient. Contestant A has an unlimited supply of dough, Contestant B has an unlimited supply of sauce, and Contestant C has an unlimited supply of cheese. Each contestant needs to get the two ingredients they do not have and then can make a pizza. They will continue to (try to) make pizza in a loop until time is up. At the beginning of the episode, the host places two different random ingredients out. Contestants can signal the host to ask for more ingredients, but they should not do so unless they actually need some. Each time the host is woken up (signalled), he again places two different random ingredients out. When an ingredient is placed on the table, the host signals the associated semaphore. For example, if the host puts out cheese and sauce, then the host signals both `cheese` and `sauce`.

This solution was inadequate; it had the risk of deadlock:

Contestant A

```
wait( sauce )
get_sauce()
wait( cheese )
get_cheese()
make_pizza( )
signal( host )
```

Contestant B

```
wait( dough )
get_dough()
wait( cheese )
get_cheese()
make_pizza( )
signal( host )
```

Contestant C

```
wait( sauce )
get_sauce()
wait( dough )
get_dough()
make_pizza( )
signal( host )
```

But the host's behaviour can also change! For **each** of the following three changes, analyze the situation: can deadlock still occur? If your answer is yes, show a sequence of events that lead to deadlock. If your answer is no, explain why deadlock will no longer occur.

One Ingredient Only (2 marks). In the first change, suppose the host instead puts out only one ingredient at a time. That means that after each ingredient is acquired, a contestant signals the host. Consider the changed code:

Contestant A

```
wait( cheese )
get_cheese()
signal( host )
wait( sauce )
get_sauce()
make_pizza( )
signal( host )
```

Contestant B

```
wait( dough )
get_dough()
signal( host )
wait( cheese )
get_cheese()
make_pizza( )
signal( host )
```

Contestant C

```
wait( sauce )
get_sauce()
signal( host )
wait( dough )
get_dough()
make_pizza( )
signal( host )
```

The Host Just Does Whatever They Want (2 marks). Suppose that now instead of needing to be signalled to put out ingredients, the host will just put out one random ingredient at a time, at random intervals. There is space on the table for up to two ingredients at a time. If there is no space right now, the host waits until space becomes available. The contestant behaviour is the original solution with all `signal( host )` statements removed.

The Host Is Also Hungry (2 marks). Consider a minor change to the previous description. instead of needing to be signalled to put out ingredients, the host will just put out one random ingredient at a time, at random intervals. There is space on the table for up to two ingredients at a time. If there is no space right now, the host will remove, at random, one of the ingredients currently on the table. The contestant behaviour is the original solution with all `signal( host )` statements removed.

2 Deadlock [14 marks total]

2.1 Do Or Do Not [5 marks]

You encounter the following code, below on the left, in a program. While it does avoid deadlock, it obviously does not work as expected. Rewrite the code to use `pthread_mutex_trylock()` properly in the space on the right.

```
void do_something( ) {
  /* Some Code */
  pthread_mutex_trylock( lock1 );
  pthread_mutex_trylock( lock2 );
  /* Critical Section */
  pthread_mutex_unlock( lock2 );
  pthread_mutex_unlock( lock1 );
  /* Other code */
}
```

```
void do_something( ) {
  /* Some Code */

  /* Critical Section */

  /* Other code */
}
```

2.2 Get the DeLorean up to 88 mph... [5 marks]

When the OS does not provide automatic rollback, you have to do it yourself. We have some structures `struct thing`, `struct widget` and `struct user`. There exists a function:

```
int foo( struct user * u, struct thing * t, struct widget * w );
```

This function modifies all of the data structures it receives as parameters. It is possible that function `foo` is involved in a deadlock. A watchdog thread is set up; if the thread running `foo()` takes too long, then it will be terminated. This will leave the structures provided as arguments in an inconsistent state. Explain, using point form, what you need to add to this logic to implement "time travel" (rollback + retry n times):

2.3 Change the Rules [4 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? Explain your answer.

1. Remove a chair (maximum 4 philosophers can sit at the table at a time).

2. 80% of the time, philosophers take the chopstick to their left first; 20% of the time, they take the chopstick to their right first.

3 Memory [25 marks total]

3.1 Virtual Memory [9 marks]

We have virtual memory with paging; physical and virtual address space are 32 bits. The size of a page is 4 KB. The data you need is shown below:

Page Table for Process A

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

Page Table for Process B

Page#	Present	Frame#
0	0	–
1	0	–
2	1	58
3	1	31

Request Table

Process	Virtual Address	Result
A	0x0000051A	
B	0x0000342A	
A	0x00001100	
A	0x00001BFF	
B	0x0000CAFE	

Part 1. (1 mark) Show a breakdown of a virtual address in this system, indicating what it is composed of.

Part 2. (2 marks) Suppose there is a Translation Lookaside Buffer (TLB); what data to do we send to it and what do we get back? And how do we keep the TLB up to date?

Part 3 (1 mark). Based on the page tables above, two processes can have the same page number. But can two processes have the same frame number? Explain.

Part 4 (5 marks). Assuming both Page Tables are in memory, complete the "Result" column of the Request Table. To do so, determine the physical address corresponding to the virtual addresses. If this is not possible due to a segmentation fault, write "Seg Fault"; if it is not possible due to a page fault, write "Page Fault". Hint: frames are the same size as pages.

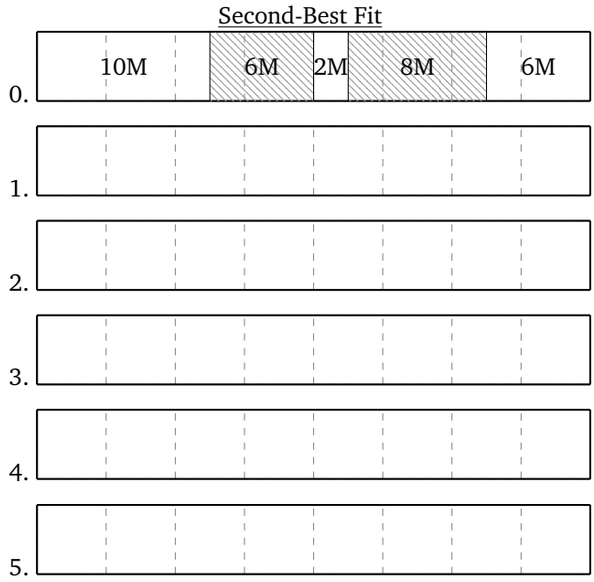
3.2 Silver Medal [5 marks]

One criticism of the Best Fit memory allocation routine is that it leaves tiny shards of unallocated memory. So, a variant of this is *second-best fit*: a memory request of size x is allocated into the second-smallest space at least as large as x. If, however, there's only one space available at least as large as x, then choose that space.

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 write 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 allocations and deallocations below using the second-best fit algorithm. If an allocation cannot be performed, write that in the box and cease execution of the algorithm. The most recently allocated block is the 8 MB block. Remember to perform coalescence immediately when it is appropriate.

1. Allocate 2 MB
2. Allocate 8 MB
3. Deallocate 6 MB
4. Allocate 4 MB
5. Allocate 4 MB



3.3 A Feast for Dwarves [11 marks]

Recall from the in-class exercise the feast that Bilbo was hosting for the dwarves. Bilbo used the "least frequently used" (LFU) replacement algorithm, taking away dishes that were unpopular and replacing them with ones that had been requested (and dishes never ran out). This is a reasonable plan, except it doesn't account for one thing: a dish that was popular in the beginning might accumulate a high count but then not be used again for a long time. So an appetizer (such as a salad) would take up table space throughout the whole meal. The obvious solution to this is aging: if a dish has not been requested in a while, it is a good candidate for replacement.

Below on the left is an implementation of LFU replacement without aging. Actually aging the counters will be done by a different thread that you will write. Below the code, you will write a list of changes that you would make to the sample code. Clearly indicate what you would change and what line(s) would be affected or where your new code would be inserted. Concurrency control is needed, so a `pthread_mutex_t` global variable called `lock` has been created for you and you may assume it has been properly initialized and will be properly destroyed before the program exits.

To age a counter, do a bitwise shift right such as `x = x >> 1;`. To set the leftmost (valid) bit in an integer, use the bitwise OR operator, such as `y = y | LEFTMOST;` (assume that `LEFTMOST` is a constant defined for you and is correct for the size of the `int` type in this system).

```
1  int serve_dishes( ) {
2    int fetches = 0;
3    int rep_idx = 0;
4    while( 1 ) {
5      int next = get_next_dish_order();
6      if ( next == -1 ) {
7        quit = true;
8        break;
9      }
10     /* Check if it's already in there */
11     bool found = false;
12     for( int j = 0; j < TABLE_SPACE; ++j ) {
13       if (platters[j].dish == next) {
14         found = true;
15         platters[j].uses++;
16         break;
17       }
18     }
19     if (!found) {
20       ++fetches;
21       for( int k = 0; k < TABLE_SPACE; ++k ) {
22         /* Empty platter has uses of -1 */
23         if ( platters[k].uses == -1 ) {
24           rep_idx = k;
25           found = true;
26           break;
27         }
28       }
29       if (!found) {
30         for ( int k = 0; k < TABLE_SPACE; ++k ) {
31           if (platters[k].uses < platters[rep_idx].uses ) {
32             rep_idx = k;
33           }
34         }
35       }
36       platters[rep_idx].dish = next;
37       platters[rep_idx].uses = 0;
38     }
39     print_state( );
40   }
41   return fetches;
42 }
```

Complete the aging thread (5 marks):

```
void* aging_thread( void* a ) {
  int * sleep_time = (int*) a;
  while( !quit ) {









    sleep( *sleep_time );


  }
  pthread_exit( NULL );
}
```

Write your changes here (6 marks):

4 Scheduling [21 marks total]

4.1 Short Answer [15 marks]

Answer each of the following questions in max. 3 sentences. 3 marks each.

1. When would it be acceptable to allow processes to starve? Explain with an example.

2. In Round-Robin scheduling, the priority of a process plays no role in how threads are selected from the ready queue. Name three ways that a higher priority process could still get an advantage over lower priority processes.

3. We have a system with many CPUs, including CPU0 and CPU1. From the point of view of scheduling, does it matter whether CPU1 is a hyperthreading core or a completely separate core from CPU0?

4. What is the advantage of the Completely Fair Scheduler's *Group Scheduling* functionality for systems like `ecelinux.uwaterloo.ca`?

5. You are evaluating the efficiency of the scheduling routine on your system. A process runs, on average, for 2 ms before being interrupted by something (time slice, being blocked, etc.). You have a 100 MHz processor and the scheduling routine takes 2500 CPU cycles to execute. What percentage of the CPU's cycles are spent performing the scheduling routine?

4.2 Battery is Here To Stay... [6 marks]

Laptops, tablets, phones, and other mobile computing devices have batteries of limited capacity. Therefore, it makes sense for for the scheduler to try to maximize battery life. Battery is usually consumed by using either the CPU intensively or using I/O devices (e.g., wireless network). Give three (3) examples of things the scheduler can do that would improve battery life, and explain your answers (2 marks each).

5 I/O Devices and Files [19 marks total]

5.1 Disk Scheduling [7 marks]

Consider a disk that has 500 cylinders labelled 0 through 499. Suppose the incoming disk reads waiting to be serviced are: `20, 302, 396, 105, 397, 280, 380`. The starting position of the disk read head is 389 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. Buffer size C means that requests are done in a group of (up to) C before proceeding on to anything else.

Algorithm	**Service Order**	**Cylinders Moved**	**Improvement over FCFS**
First-Come First Serve (FCFS)	20, 302, 396, 105, 397, 280, 380		1.000
LOOK			
SSTF + Double Buffering, Buffer size 4			

5.2 I/O Device Type [7.5 marks]

Consider a typical desktop computer keyboard as an I/O device. How would you categorize this device with respect to the following criteria? Justify your answers (1.5 marks each):

1. Data transfer mode

2. Access method

3. Transfer schedule

4. Dedication

5. Transfer direction

5.3 File Permissions [4.5 marks]

1. In the UNIX Permission scheme, what do the permissions `540` on a file mean (2 marks)?

2. You have a UNIX system with 5000 users. There is a particular file that should be accessible for reading by everyone except 10 users. How could you implement this policy efficiently? Explain (2.5 marks).

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 options ) /* 0 for options OK */
int kill( pid_t pid, int signal ) /* returns 0 returned if signal sent, -1 if an error */
void signal( int signal_number, void (*handler)(int signal) );


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 num_bytes); /* Returns number of bytes read */
ssize_t write(int file_descriptor, const void *buffer, size_t num_bytes); /* Returns number of bytes written */
int rename(const char *old_filename, const char *new_filename); /* Returns 0 on success */
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 **return_value )
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_cleanup_push( void (*routine)(void*), void *argument ); /* Register cleanup handler, with argument */
pthread_cleanup_pop( int execute ); /* Run if execute is non-zero */

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 )
```

Linux Atomic integer operations:

Function	Description
`ATOMIC_INIT( int i )`	At declaration, initialize an `atomic_t` to `i`
`int atomic_read( atomic_t *v )`	Read the integer value of `v`
`void atomic_set( atomic_t *v, int i )`	Set `v` equal to `i`
`void atomic_add( int i, atomic_t *v )`	Add `i` to `v`
`void atomic_sub( int i, atomic_t *v )`	Subtract `i` from `v`
`void atomic_inc( atomic_t *v )`	Add 1 to `v`
`void atomic_dec( atomic_t *v )`	Subtract 1 from `v`
`int atomic_sub_and_test( int i, atomic_t *v )`	Subtract `i` from `v`; return true if 0; otherwise false
`int atomic_add_negative( int i, atomic_t *v )`	Add `i` to `v`; return true if negative; otherwise false
`int atomic_dec_and_test( atomic_t *v )`	Decrement `v` by 1; return true if 0; otherwise false
`int atomic_inc_and_test( atomic_t *v )`	Increment `v` by 1; return true if 0; otherwise false