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		16	2.4		10	4.1		5
1.2		10	3.1		10	4.2		7.5
1.3		5	3.2		5	4.3		7.5
2.1		10	3.3		10			
2.2		4	3.4		5			
2.3		10	3.5		5			
Total								**120**

1 Concurrency, Synchronization, Deadlock [31 marks total]

1.1 Producer-Consumer Problem [16 marks]

Consider the below producer-consumer code, very similar to the coding example from the lectures.

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

typedef struct node {
  pthread_t thread;
  struct node * next;
} node;

item* buffer[BUFFER_SIZE]; /* array of pointers */
int pindex = 0;
int cindex = 0;
sem_t spaces;
sem_t items;
pthread_mutex_t mutex;
node * consumers;

item* produce( ) {
  /* Implementation not shown */
}

void consume( item * i ) {
  /* Implementation not shown */
}
```

```
void* producer( void* arg ) {
  while( 1 ) {
    item* i = produce();
    sem_wait( &spaces );
    pthread_mutex_lock( &mutex );
    buffer[pindex] = i;
    pindex = (pindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &items );
  }
  pthread_exit( NULL );
}

void* consumer( void* arg ) {
  while( 1 ) {
    sem_wait( &items );
    pthread_mutex_lock( &mutex );
    item* i = buffer[cindex];
    buffer[cindex] = NULL;
    cindex = (cindex + 1) % BUFFER_SIZE;
    pthread_mutex_unlock( &mutex );
    sem_post( &spaces );
    consume( *item );
  }
  pthread_exit( NULL );
}
```

Your goal is to complete the code below to automatically create 3 new consumers if the buffer is above 75% full, and destroy 3 consumers if the buffer is below 25% full. However, there should always be at least one consumer, so if there's only one consumer left, do not destroy it. You may need to make changes to the `consumer` function, but rather than copying those out again, there is space below to indicate what you would change in them. Hint: use a linked list to keep track of the consumers.

```
int main( int argc, char** argv ) {
    pthread_t b_thread;

    /* Initialize global Variables Here */
    sem_init( &items, 0, 0 );
    sem_init( &spaces, 0, BUFFER_SIZE );
    pthread_mutex_init( &mutex, NULL );
    memset( &buffer, 0, BUFFER_SIZE * sizeof( item* ));



  /* Creation of producer threads not shown */

  pthread_create( &b_thread, NULL, balancer, NULL );
  pthread_exit( NULL );
}

/* Computes usage of the buffer as a fraction of 1 */
double buffer_usage( ) {









}
```

```
void* balancer ( void* arg ) {
   while( 1 ) {









    sleep( 30 ); /* Sleep this thread for 30 seconds */
   }
}
```

Consumer Function Explain what you would change in the `consumer` function, being specific about what lines of code you would add/remove/change, and where.

1.2 Wait Your Turn! [10 marks]

As we have discussed in class, when signalling on a semaphore, we have no guarantee about which thread will be unblocked (and this makes the semaphore "unfair"). Suppose, that you want to have fair first-in-first-out behaviour in your program, so the first thread that waits on a semaphore is the first one to get woken up when a signal occurs. You are in luck because the `pthread_mutex_t` in your system is fair (first-in-first-out) but the `sem_t` is not. Complete the implementation below of the `sem_queue` type to provide the desired behaviour.

```
typedef struct {
  sem_t sem;
  pthread_mutex_t mutex;
  /* Put any additional variables below */









} sem_queue;

void sem_queue_init( sem_queue * q, int init_value ) {









}

void sem_queue_destroy( sem_queue * q ) {









}
```

```
void sem_queue_wait( sem_queue * q ) {









}

void sem_queue_signal( sem_queue * q ) {









}
```

1.3 Dealing with Deadlock [5 marks]

During your co-op term at Cyberdyne Systems, you are working on a the control software for the T-1 (forerunner of the T-800, obviously) that occasionally gets deadlocked. A coworker has decided to solve this by replacing all `pthread_mutex_t` structures with one single global variable mutex. Assume there are no other concurrency or synchronization constructs in the program.

Part 1. (3 marks) Will this solve the problem? Explain your reasoning.

Part 2. (1 mark) Name one major downside to this plan.

Part 3. (1 mark) Is there an alternative approach to solve the problem that might be better?

2 Memory [34 marks total]

2.1 Determining Page Number [10 marks]

You have a system has a 32-bit virtual address, with a 4 KB page size and the size of an int is 32 bits as well. Write a C program that takes virtual address (in decimal) on the command line as an argument, and produces as output, the page number and offset for the given address. As an example, your program would run as follows:

```
./address X
The page number is Y and the offset is Z.
```

Where of course X, Y, and Z are replaced with actual values. You do not need to do any error checking.

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









  return 0;
}
```

2.2 Caching [4 marks]

In class we have discussed the idea that in caching we can compare against the optimal algorithm, even though the optimal algorithm cannot be implemented. Explain the steps you would need to take to evaluate the effectiveness of a particular cache replacement algorithm against the optimal, for a given program. Use point form.

2.3 Virtual Memory [10 marks]

Imagine you have a 32-bit virtual memory system with 4KB pages, as discussed in the lectures, using a translation lookaside buffer, and a regular page table.

How would you detect an invalid memory access from a user program (2 marks)?

What does the operating system do when a user program does reference an invalid location (1 mark)?

What steps take place to transform a logical address into a physical address (3 marks)?

What are the tradeoffs in choosing the size of the translation look aside buffer (2 marks)?

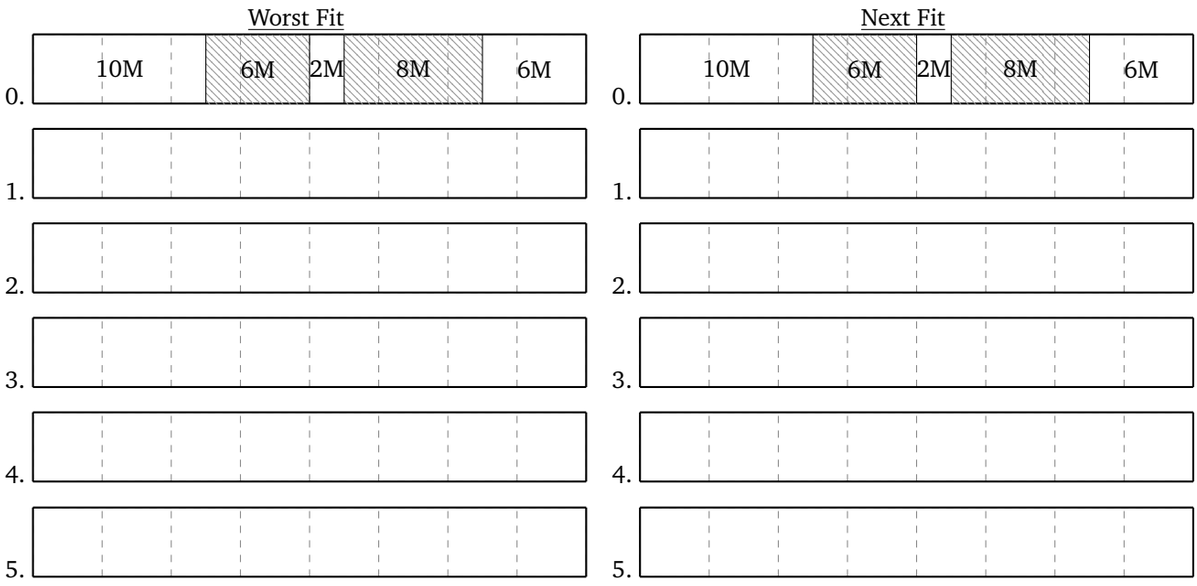
Would changing to a hierarchical page table with two levels be better? Justify your answer (2 marks).

2.4 Dynamic Memory Allocation [10 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 and deallocations using the *worst fit* algorithm on the left side and using the *next fit* algorithm on the right side. 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 Scheduling [35 marks total]

3.1 Real Time Scheduling [10 marks]

You have a real time system where all tasks are hard real time, and the scheduling algorithm is earliest deadline first, and time slicing is used. At the end of each time slice, the scheduler runs, evaluates the current state, and decides what task to run next. Assume no tasks will ever get blocked for any reason.

The table below gives the breakdown of the tasks. A task that arrives during time slice n may be scheduled during time slice $n + 1$ (but not sooner). A task that has an execution length of k requires k time slices to complete. A task with a deadline of D must complete during or before time slice D.

Task ID	Arrival Time Slice	Execution Length (time slices)	Deadline Time Slice
A	0	5	10
B	4	3	8
C	9	6	20
D	10	4	14
E	15	2	17
F	17	2	20

Each block represents a time slice. In each box below, write the ID of the task that will be executed in that time slice. If there is nothing executing in a time slice, write a dash (–) in the box.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

3.2 Linux Constant Time Scheduler [5 marks]

Demonstrate that the Linux Constant Time ($O(1)$) Scheduler actually has $O(1)$ characteristics in all its major operations. You need to consider separately each thing that the scheduling routine does.

3.3 Scheduling Parameters [10 marks]

Discuss the impact of changes in size of the following parameters in scheduling, considering what happens when the value is too small and when it is too large (2 marks each):

1. Length of a time slice

2. The amount of time it takes to choose the next thread to run

3. The importance of process priority in decision-making

4. The priority increase given to threads that have just completed an I/O operation

5. The priority given to I/O-bound processes over CPU-bound ones

3.4 Scheduling Starvation [5 marks]

How do the following scheduling algorithms prevent starvation (1 mark each)?

1. Highest Response Ratio Next

2. Virtual Round Robin

3. Shortest Job Next

4. The Linux Constant Time ($O(1)$) Scheduler

5. The Linux Completely Fair Scheduler

3.5 Process Politeness [5 marks]

In the "lottery" scheduling algorithm, a process can "give" its tickets to another process to increase the share of resources that the recipient is likely to receive. What are two (2) pros and two (2) cons of this approach, and give an example of where it might not be possible to give tickets to another process.

4 I/O and Files [20 marks total]

4.1 NTFS Journalling [5 marks]

In NTFS (the Windows NT File System) there is journalling which is used to stop metadata from being in an inconsistent state, but it is possible user data ends up in an inconsistent state. What would have to be changed to prevent user data from being in an inconsistent state? Give three reasons why (you think) Microsoft has not implemented this.

4.2 Disk Access Time [7.5 marks]

Suppose you have a system with one level of cache, main memory, and a magnetic hard disk. It takes 7 ns to read from cache, 1 microsecond to read from main memory, and 10 ms to read from the hard disk. The cache hit rate is 98%. A cache miss will be found 99% of the time in memory. Remember the effective access time formula:

$$\text{Effective Access Time} = h \times t_c + (1 - h)(p \times t_m + (1 - p) \times t_d)$$

Part 1 (0.5 marks): Compute the effective access time for the system as-is.

Part 2 (3 marks): A hard drive manufacturer offers you an upgrade: a hard drive with a built-in cache. This cache uses the principle of spatial locality to guess what block of the hard drive the program is likely to access next. If it is correct, the data is ready in 2 ms; otherwise 12 ms (so it is slower, if it gets it wrong). The sales representative tells you that the cache predicts correctly, on average, 32% of the time. Suppose that is true; what will the effective access time be?

Part 3 (4 marks): Calculate the break-even point for this upgraded system. That is, what does the hit rate (in percentage) of the hard drive built-in cache have to be equivalent to the system without this new hard drive?

4.3 I/O Device Type [7.5 marks]

Consider the network as an I/O device. How would you categorize this device along 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

Additional Free Space

Use this space if needed. Be sure to clearly indicate which question is being worked on.

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 */
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 )

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 )
```