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 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):

Q	Mark	Weight	Q	Mark	Weight	Q	Mark	Weight	Q	Mark	Weight
1a		30	2a		5	3a		4	4a		5
1b		10	2b		10	3b		6	4b		21
			2c		10	3c		10	4c		4
			2d		5						
Total											**120**

Question 1: Memory [40 marks total]

1A: Caching Design Problem [30 marks]

Read all parts of question 1A before beginning. You will need to make design decisions and if you know everything the program needs to do, you will not have to backtrack on some decisions. There is NO requirement for your solution to be optimal: simple, slower, and correct is better than complicated, fast, and incorrect.

In this question, you will write a program to simulate cache replacement algorithms, both First-In-First-Out (FIFO) and Least-Recently-Used (LRU). The system you are simulating is an embedded system that has a 32 bit architecture, so it has a 32 bit memory address space, and has page size of 4 Kilobytes (4096 Bytes). A program argument will set the mode to `0` for FIFO and `1` for LRU. The program takes further arguments for memory addresses where accesses (reads or writes) take place.

The regular integer type (`int`) is not guaranteed to be anything more than 16 bits so use of `unsigned long` is appropriate in the program. The C99 standard specifies that `long`, a "long integer", is at least 32 bits (and you want it to be unsigned to handle a memory address), so you will be working mostly with this type.

The section below is for global variables and definitions. As you build up your solution you can add more things to this section. You can assume any appropriate standard header file (e.g., stdlib) exists and has been included.

```
/* Place global variables and definitions here */
int mode;
int num_frames;
```

Part 1: Working out page number (1 mark). Create a function that, given a memory address, computes the page number. For example, if the input is `19986`, the return value of the function you create would be `4`. The offset (where inside the page the memory access takes place) is not relevant.

```
unsigned long get_page_num( unsigned long address ) {



}
```

Part 2: Parse the arguments (5 marks). Create a function that takes arguments `int argc, char** argv` where `argc` is the count of arguments and `argv` is the actual program arguments. If fewer than four arguments are provided, print an error message and call `abort()` to terminate the program immediately. Remember the first argument in `argv` is the name of the executable called (and you can ignore it). Use the second argument to set the mode (global variable `mode`). Use the third to set the number of frames (global variable `frames`). After that there are multiple (at least one) additional integer arguments that are the memory addresses. An example invocation may look like this: `./a.out 0 3 19959 258285 29527258 25782572 258258292 575765 29572 102`.

You may convert the input arguments to `unsigned long` using the `strtoul` function. This function takes 3 parameters and returns an unsigned long. The first parameter is the string (element in the `argv` array to convert); the second argument should be `NULL` and the third argument `10` (for base 10 conversion). Assume this function will always succeed and that users will not provide invalid input. You can also use `atoi()` to convert an argument to an integer (and you may assume this will also succeed and do not have to deal with invalid input).

This function should return a pointer to an array of memory addresses that have been successfully parsed out of the command line arguments. That memory will be deallocated later when you are finished with it.

```
unsigned long* parse_args ( int argc, char** argv ) {



}
```

Part 3: Printing the current state (4 marks). This function prints to the console the current state of the frames. The output should be reported in parenthesis with commas separating the values, and each set of outputs on a new line. If a frame does not contain a valid page (e.g., it is the start of execution), print a dash (–) instead of a number. If, for example, there are 3 frames and they currently contain the pages of `2958`, `11`, and `385` respectively, the output should look like: `(2958, 11, 385)`. If there are 3 frames and they contain the pages `4`, `42`, and the last frame has never had any page assigned to it, the output should look like: `(4, 42, -)`.

```
void print_state(                    ) {









}
```

Part 4 Implementing FIFO (8 marks) and LRU (8 marks). In this part of the question you will implement the simulation of the FIFO and LRU algorithms for page replacement. You may design the function signatures (arguments, return type) for the functions `simulate_fifo` and `simulate_lru` as you like. The functions should, obviously, simulate the FIFO and LRU algorithms, respectively. After each step of the simulation, print out the current state of the frames.

```
           simulate_fifo(                              ) {









}

           simulate_lru(                               ) {









}
```

Part 5: Writing `main` (5 marks) In this final part of the question, complete `main` to use all the functions you have written above, so that the program completes the simulation as expected. Initialize anything that needs to be initialized, then, based on the selected mode, run either the FIFO or LRU simulation. When the simulation is done, print a message to the console telling the user what the total number of page faults was. Make sure to deallocate any resources that have been allocated in the program before the final `return` statement.

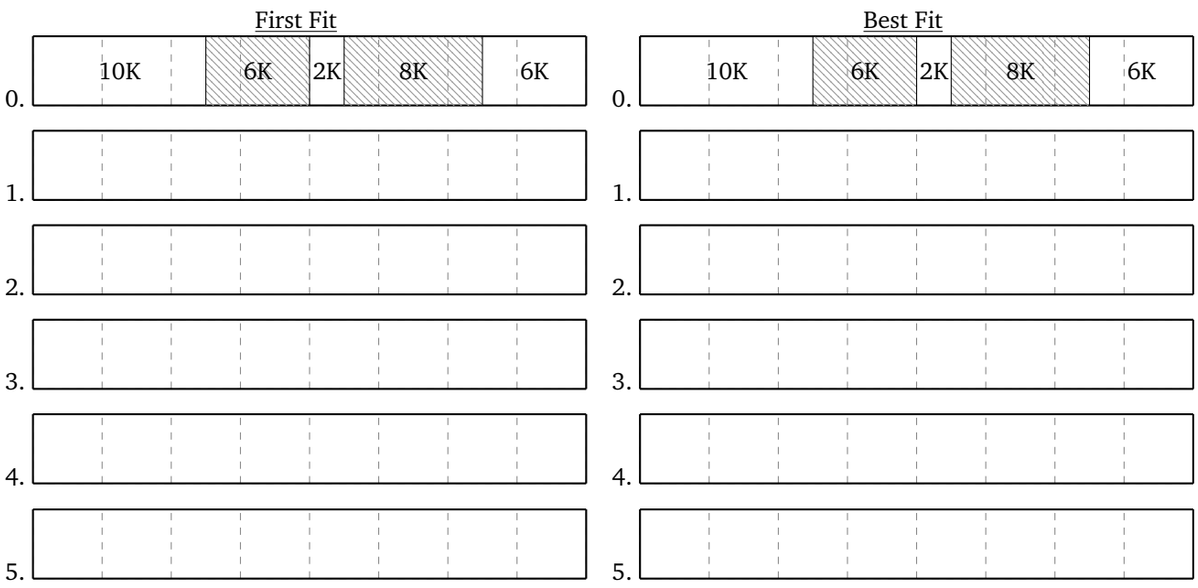
```
int main( int argc, char** argv ) {









  return 0;
}
```

1B: Dynamic Memory Allocation [10 marks]

The diagrams below show a 32 KB 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 KB 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 *first fit* algorithm on the left side and using the *best 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 KB block. Remember to perform coalescence immediately when it is appropriate.

1. Allocate 2 KB
2. Allocate 3 KB
3. Allocate 6 KB
4. Deallocate 8 KB
5. Allocate 10 KB



Question 2: Scheduling [30 marks total]

2A: Scheduling Algorithm Evaluation [5 marks]

On your co-op work term, a colleague suggests a new scheduling algorithm to you: each process starts at a default priority level and has a time slice length of 5 ms. Every time the process reaches the end of a time slice without being blocked, the length of its time slice is increased by 1 ms. If the process gets blocked before reaching the end of the time slice, its time slice is reduced in length by 1 ms to a minimum of 1 ms. Analyze the advantages and disadvantages of such a scheduling algorithm, and make a recommendation as to whether it should be used.

2B: 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

2C: Scheduling Algorithm Relations [10 marks]

Many CPU-scheduling algorithms take a parameter. For example, the Round-Robin algorithm requires a parameter to indicate the length of a time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithm for each queue, the criteria used to move processes between queues, and so on. It is possible sometimes to describe one algorithm as being another with some specific parameters. What (if any) relation holds between the following pairs of algorithm sets (2 marks each)?

1. First-Come-First-Serve and Round-Robin

2. Shortest Process Next and Shortest Job First

3. Round Robin and Lottery

4. Highest Priority Period and Multilevel Feedback Queue (FCFS in each queue)

5. Lottery and Guaranteed Scheduling

2D: CPU-Bound & I/O-Bound Processes [5 marks]

One piece of information we are interested in when deciding how to schedule a process is whether that process is CPU-Bound or I/O bound. Making this determination is, however, not trivial. Complete the table below to list one advantage and one disadvantage for each of the five potential approaches listed in the leftmost column.

Approach	**Advantage**	**Disadvantage**
OS Guesses Randomly		
Developer Specifies at Compile-Time		
User Specifies at Launch-Time		
OS Scans Binary and Counts System Calls (many = I/O Bound)		
OS Assumes CPU-Bound; Modifies if High I/O Usage		

Question 3: I/O Devices [20 marks total]

3A: Disk Scheduling Queue [4 marks]

In lecture we discussed the idea of the Shortest Seek Time First (SSTF) Algorithm and that it is vulnerable to starvation if the disk is busy. Double buffering was a solution to prevent this situation where each buffer can contain C requests. Discuss the tradeoffs involved in choosing the size of configuration parameter C.

3B: File Allocation [6 marks]

A very small USB drive is formatted such that it has 32 blocks numbered 0 through 31. The table below lists the files currently allocated on that drive and their respective block(s).

File ID	Allocated Block(s)
A	0
B	2, 3, 4
C	28, 29, 30
D	16, 17, 18, 19, 20, 21, 22
E	11, 12, 13

Then two more files are allocated: first file F with a size of 4 blocks, then file G with a size of 2 blocks. Complete the table below to show where the blocks of files F and G would be allocated, according to the algorithm listed in the leftmost column.

File Allocation Method	Allocated Blocks for File F	Allocated Blocks for File G
Contiguous Allocation		
Contiguous Allocation, Best Fit		
Linked Allocation		

3C. Caching and Hard Drives [10 marks]

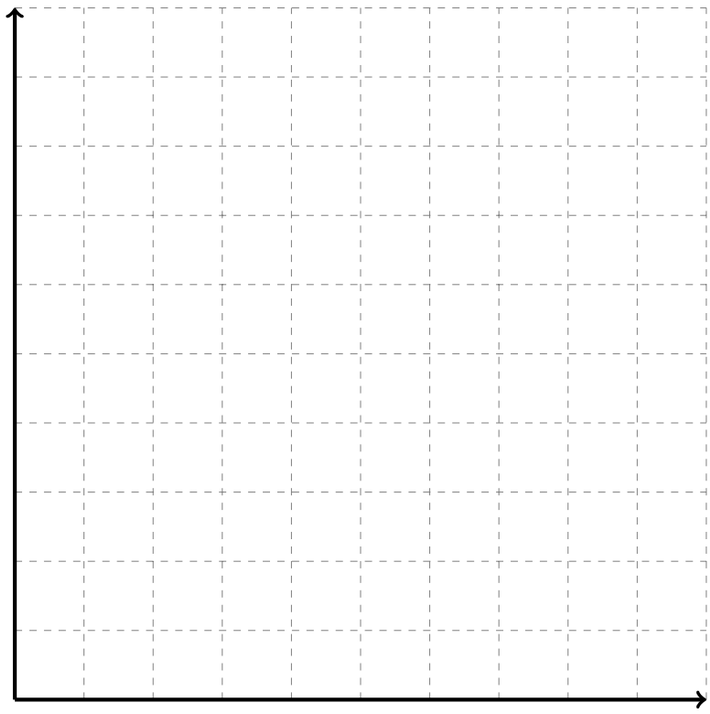
Suppose you have a system with one level of cache, main memory, and a magnetic hard disk. It takes 7 ns to read from cache, 1 microsecond to read from main memory, and 10 ms to read from the hard disk. The cache hit rate is 98%. A cache miss will be found 99% of the time in memory. Remember the effective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

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

Part 1 (2 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 10 ms. The sales representative tells you that the cache predicts correctly, on average, 20% of the time. Suppose that is true; what will the effective access time be?

Part 3 (5 marks): You are, for obvious reasons, distrustful of marketing and sales claims. You know performance will vary based on how effective the prediction algorithm is. Complete the graph in the space below that relates the hit rate in the hard drive built-in cache (the X-axis) to effective access time (the Y-axis) over the full range of possibilities for how effective the cache could be. Remember to label your axes!



Question 4: Concurrency & Synchronization [30 marks total]

4A: Mutual Exclusion with Lock Variables [5 marks]

The following code attempts to enforce mutual exclusion to protect a critical section by attempting to lock, then checking if it was successful. If `lock` equals zero, it means it is unlocked; any non-zero value means it is locked. Assume at program startup `lock` is initialized with a value of `0`.

```
1  while( 1 ) { /* Infinite Loop */
2     lock++;
3     if ( lock > 1 ) {
4          /* lock was > 0 before we increased */
5          /* undo the increment and try again */
6         lock--;
7     } else {
8         break; /* Success! Can enter critical section */
9     }
10 }
11 /* Critical Section */
12 lock--; /* Unlock */
```

Notice that the variable `lock` is shared between all threads requiring access to the critical section. Does this technique work? If you answer yes, explain why it works; if you answer no, show an interleaving demonstrating a situation where it fails (i.e., a situation where two threads are in the critical section at a time).

4B: Readers-Writers Problem [21 marks]

Recall from lectures the Readers-Writers problem. There are two kinds of threads, readers and writers. A writer will modify the data; a reader will read it only without modification. Any number of readers may be in the critical section simultaneously, but only one writer may be in the critical section (and when it is, no readers are allowed). Or, in other words, a writer cannot enter the critical section while any other thread (whether reader or writer) is there. While a writer is in the critical section, neither readers nor writers may enter the critical section. In this question you will implement in C the solution that ensures that writers do not starve.

You may assume that readers and writers are created correctly (how this is done is not shown for space reasons). You may also assume that the `init` and `cleanup` functions to initialize global variables and perform cleanup are called at the correct points in the code. A reader reads data by calling the function `void read_data( );` a writer writes data by calling the function `void update_data( )`. The implementations of the read and update functions are not shown but you may assume they exist and work correctly. When a thread has finished its work it may terminate normally. Hint: you will want two semaphores and one mutex.

Declare any global variables here:

Initialize global variables in `init` and perform necessary cleanup in `cleanup` below (6 marks):

```
void init() {

}
```

```
void cleanup() {

}
```

Complete the C functions below for the writer side (5 marks), the reader side (10 marks):

```
void* writer( void * ignore ) {

}
```

```
void* reader( void * ignore ) {

}
```

4C: Joinable Threads [4 marks]

Recall that pthreads may be either *detached* or *joinable*. If a thread is joinable we collect a return value with `pthread_join`. Suppose you are writing code for a system where threads are always detached and cannot be made joinable. How could you simulate the behaviour of joinable threads?

POSIX System Call Reference

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 );`

Printing is done using `printf` with formatting. `%d` prints integers; `%lu` prints unsigned longs. A newline is created with `\n`.

Some UNIX functions you may need:

```
pid_t fork( )
void wait( int* status )
void waitpid( pid_t pid, int status )
kill( pid_t pid, int signal )
```

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( ) /* returns nonzero if cancelled */
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 )
```