Instructions:

1. No aids are permitted except non-programmable calculators.
2. Turn off all communication devices.
3. There are four (4) questions, some with multiple parts. Not all are equally difficult.
4. The exam lasts 150 minutes and there are 100 marks.
5. 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.
6. 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		4	2a		8	3a		13	4a		9
1b		6	2b		2	3b		2	4b		3
1c		3	2c		8	3c		8	4c		3
1d		6	2d		2				4d		15
1e		3	2e		5						
Total											**100**

Question 1. Memory [22 marks total]

1A. Locale [4 marks]

Recall from class that caching works because of the existence of temporal locality and spatial locality. Explain what those two concepts mean.

1B. Caching and Hard Drives [6 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.

You have the budget necessary to make one of the following improvements (but only one): (1) increase the size of cache so the cache hit rate goes to 99%; or (2) buy a better hard disk so that it takes only 9 ms to read from hard disk. Which of these should you choose if your goal is to minimize the average access time? Explain which you would choose, and justify your answer quantitatively.

1C. Dynamic Memory Allocation [3 marks]

When memory allocation requests come in, they may not exactly match the size of a block, so after the allocation we may be left with some fragmentation. This can be internal (inside the allocated block) or external (outside of any allocated block). Either way the memory is "wasted". Give a theory as to why we might prefer internal fragmentation to external (under some reasonable circumstances).

1D. Dynamic Memory Allocation II [6 marks]

In addition to `malloc()` to allocate memory and `free()` to deallocate memory, some systems support the idea of `realloc()`. This is used to take a block of memory and expand it. Suppose you had allocated an array of 10 elements and you would now like to expand the array to have a capacity of 20. You can call `realloc` with a pointer to the original array and the new, larger size. The returned memory will be contiguous and still contain the first ten elements of the array exactly as they were. Explain how `realloc` might fulfill this request efficiently.

1E. Best Fit Algorithm [3 marks]

In the labs for this class, you were asked to implement the "Best Fit" and "Worst Fit" dynamic memory allocation algorithms to allocate memory. Describe how the best fit memory allocation algorithm works as it was implemented in your lab (keeping in mind that memory requests are rounded up to the nearest 4 bytes).

2. Scheduling [25 marks total]

2A. Evaluation Criteria [8 marks]

Recall from class there were nine (9) criteria that could be used to evaluate a given scheduling algorithm. Name and describe briefly four (4) of those criteria.

2B. Real-Time Scheduling [2 marks]

Describe one set of circumstances where a real time operating system will not schedule a task with a hard deadline.

2C. Scheduling Order [8 marks]

Consider this table of information about processes:

Process ID	Service Time	Arrival Time
A	1058	0
B	581	1586
C	691	404
D	104	0
E	256	95

Give the scheduling order for these processes according to the *Highest Response Ratio Next* algorithm. Note that not all processes arrive at the same time and so a process cannot yet run if it has not arrived. Assume, however, this is non-preemptive. Remember that the response ratio, R, is calculated as: $\frac{w+s}{s}$ where w is the waiting time and s is the service time. And remember that time keeps ticking as processes run, so re-evaluate after each process is finished what should be next.

2D. Virtual Round Robin [2 marks]

Recall that the Virtual Round Robin scheduling algorithm has a ready queue and also an auxiliary queue. What is the purpose of the auxiliary queue?

2E. The Constant Time Scheduler [5 marks]

Recall from the lecture about Linux scheduling the existence of the $O(1)$ or *Constant Time Scheduler*. List five (5) distinct points about how it operates.

3. I/O and Disk [23 marks total]

3A. Disk Scheduling [13 marks]

Consider a disk that has 500 cylinders labelled 0 through 499. Suppose the incoming disk read requests are currently waiting to be serviced: $66, 356, 297, 268, 461$. The starting position is 283 and the disk head is moving in a descending direction (towards 0).

Complete the table below, showing the service order, total cylinders moved, and the improvement compared to First-Come First Served (that is, the speedup as calculated by cylinders moved in FCFS divided by cylinders moved in the chosen strategy) rounded to three decimal places.

Strategy	Service Order	Total Cylinders Moved	Improvement over FCFS
First Come First Served	66, 356, 297, 268, 461		1.000
Shortest Seek Time First			
SCAN			
LOOK			

3B. Files [2 marks]

What is the purpose of the open and close system calls?

3C. File Allocation Methods [8 marks]

Compare and contrast the file allocation methods of *Contiguous Allocation* and *Linked Allocation*. The comparison should include some points about how these methods work as well as their advantages and disadvantages.

4. Concurrency and Synchronization [30 marks total]

4A. Synchronization Problems [9 marks]

The program analysis tool Helgrind identifies three categories of error in programs using POSIX pthreads. They are: (1) incorrect use of the API, (2) lock ordering problems, and (3) data races. For each of those categories, explain, using an example, a particular instance of that programming error.

4B. Programming with pthreads [3 marks]

Imagine you are on co-op and you have been asked by a colleague to review some code. The goal of the function is to execute an algorithm in parallel based on the number of CPUs. Assume that all the variables and parameters are properly defined. After all setup is done, you encounter this section of code to launch the threads and collect their results. Explain, briefly, what is wrong with this code.

```
for ( int i = 0; i < NUM_CPUS; ++i ) {
    pthread_create( &threads[i], NULL, function, params );
    pthread_join( threads[i], &returnValues[i] );
}
```

4C. Deadlock [3 marks]

Can clever use of scheduling prevent deadlock in all situations? Your answer should begin with either yes or no, followed by a brief explanation of your reasoning.

4D. Thread Pool [15 marks]

A common programming practice is to have a pool of "worker" threads to service incoming requests. The requests are placed in a queue and when a worker is available to service the next request, it takes the next work item and executes it. An item of work to do is usually called a "work unit" or "unit of work", and in this question it is modelled as a `struct` that has been typedef'd as `workunit`. You do not need to know anything about the internals of this structure.

For the purpose of this question, the work queue is modelled as a linked list. You do not need to know the implementation of the linked list. Assume the list is created and initialized for you (and you do not need to do so). There are functions available to remove the first element of the list and to add items to the back of the list:

```
void* pop_front( )
void push_end( void* toAdd )
```

Warning: these operations are NOT thread safe.

There are four (4) functions to implement, as follows.

1. `main`**.** In the `main` function you will spawn `NUM_WORKERS` threads that are to be workers. Workers run the `fetchAndExecute` function. The setup is all `main` needs to do; the call to `pthread_exit` at the end ensures that it will wait for all spawned threads before the function actually exits.

2. `interruptHandler`**.** In this function, you should signal to each of the spawned threads that they are cancelled using the `pthread_cancel` routine.

3. `fetchAndExecute`**.** This is the function that a worker thread executes. In this function, create an infinite loop that does the following: dequeue the next unit of work with the `pop_front` function and process it using the function `int execute( workunit *toDo )`. If the execute function returns any value other than `0`, assume that something went wrong and place the unit of work back in the queue with `push_end`). Remember that the functions to manipulate the work queue are not thread safe. If there is nothing in the queue of work to do, `pop_front` returns a pointer to `0`, in which case that worker should call `sleep(1000)` before it tries again.

4. `enqueue_workunit`**.** Here, add the unit of work given as an argument to the back of the work queue.

For your convenience, a partial listing of the pthread functions that you can use:

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument )
pthread_join( pthread_t thread, void **returnValue )
pthread_cancel( pthread_t thread )
pthread_exit( void *value )
pthread_mutex_init( pthread_mutex_t *mutex, pthread_mutexattr_t *attributes )
pthread_mutex_lock( pthread_mutex_t *mutex )
pthread_mutex_unlock( pthread_mutex_t *mutex )
pthread_mutex_destroy( pthread_mutex_t *mutex )
```

You may pass in the constant `NULL` for the attributes in each call.

Complete the functions in C on the next page. Your solution should avoid deadlock, race conditions and other concurrency problems.

```
pthread_t threads[NUM_WORKERS];
void *fetchAndExecute( void *void_arg );

/* When SIGINT received, cancel all pthreads */
void interruptHandler( int ignore ) {









}

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

    /* Define the interruptHandler function to catch SIGINT */
    signal(SIGINT, interruptHandler);









    pthread_exit( 0 );
}

void fetchAndExecute( void *void_arg ) {

```

```
}


void enqueue_workunit( workunit* wu ) {

```



```
}
```