Instructions:

1. No aids are permitted except for 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):

Question	Mark	Weight	Question	Mark	Weight	Question	Mark	Weight
1a		2	2a		4	3d		2
1b		2	2b		4	3e		6
1c		2	2c		14	4a		4
1d		4	2d		2	4b		6
1e		3	3a		13	4c		18
1f		4	3b		3			
1g		5	3c		2			
Total								**100**

Question 1: Scheduling [22 marks total]

1A: Time Slicing [2 marks]

Choosing the length of a time slice is a design decision; identify the tradeoffs of longer vs. shorter time slices.

1B: Real-Time Scheduling [2 marks]

Consider the processes as described below.

Process ID	Deadline	Execution Time
A	2059	50
B	311	120
C	512	122
D	584	79

Give the scheduling order for these processes according to the *Least Slack Time First* algorithm.

1C: Shortest Job First [2 marks]

Consider the processes as described below.

Process ID	Predicted Total Process Time	Predicted Next CPU Burst
A	240	8
B	109	12
C	42	42
D	2066	33

Give the scheduling order for these processes according to the *Shortest Job First* algorithm.

1D: Round-Robin Scheduling [4 marks]

Recall that the (basic) Round-Robin scheduling algorithm tends to favour CPU-bound processes. Give two (2) reasons why it is generally desirable for a scheduling algorithm to give some priority to I/O-bound processes.

1E: Priority Inheritance [3 marks]

Explain in three (3) points the concept of *priority inheritance* and the problem this technique solves.

1F: CPU Affinity [4 marks]

Processor affinity is the ability to assign a process such that it will run on only one (or one set of) specified CPU(s). Explain two (2) situations where processor affinity would result in better execution performance and why it improves performance.

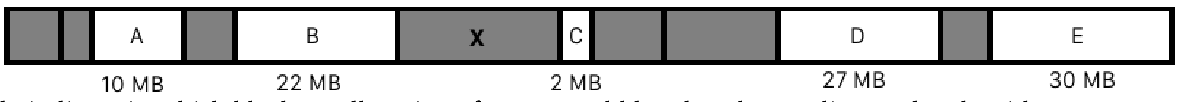
1G: Linux Scheduling [5 marks]

Recall that in Linux the CTS – Constant Time or $O(1)$ Scheduler – was replaced with the CFS – Completely Fair Scheduler. Provide a five (5) point technical explanation of **either** the $O(1)$ scheduler **or** the CFS (but not both). Clearly indicate which one you are explaining. Point form is acceptable.

Question 2: Memory [24 marks total]

2A: Dynamic Memory Allocation [4 marks]

Suppose the current state of memory is as shown as below. The shaded areas indicate memory that is currently allocated. The area marked with an X indicates where the most recent memory allocation took place.



Clearly indicate in which block an allocation of 8 MB would be placed according to the algorithm:

1. First Fit:
2. Next Fit:
3. Best Fit:
4. Worst Fit:

2B: Caching [4 marks]

The system you are operating on has one level of cache, main memory, and a magnetic hard disk. It takes 7 ns to read something from cache, 1 μs to read from main memory, and 10 ms to read from disk. Assume that 99% of the time a cache miss will be found in memory. What is the cache hit rate (expressed between 0 and 1, rounded to 3 decimal places) necessary such that the average access time is at most 750 ns? Show your work.

2C: Caching Algorithms [14 marks]

A system has a cache that can contain three pages and it has pages 0 through 7. The cache starts out completely empty. Suppose the page references occur in this order: $1, 2, 3, 4, 2, 1, 0, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6$. Complete the tables below to show the state of cache after each reference and sum up the total page faults.

First-In First-Out

Step	`cache[0]`	`cache[1]`	`cache[2]`	**Fault?**
1	1	—	—	✓
2	1	2	—	✓
3	1	2	3	✓
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
FIFO Page Faults:				

Least Recently Used

Step	`cache[0]`	`cache[1]`	`cache[2]`	**Fault?**
1	1	—	—	✓
2	1	2	—	✓
3	1	2	3	✓
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
LRU Page Faults:				

2D: Thrashing [2 marks]

Define *thrashing*.

Question 3: Disk [26 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: $356, 9, 472, 375, 302$. The starting position is 309 and the disk head is moving in an ascending direction (towards 499).

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	356, 9, 472, 375, 302		1.000
Shortest Seek Time First			
SCAN			
C-SCAN			

3B: Disk Journalling [3 marks]

Explain *disk journalling* by describing what it is, why it is desirable to have it, and its limitations.

3C: B-Trees [2 marks]

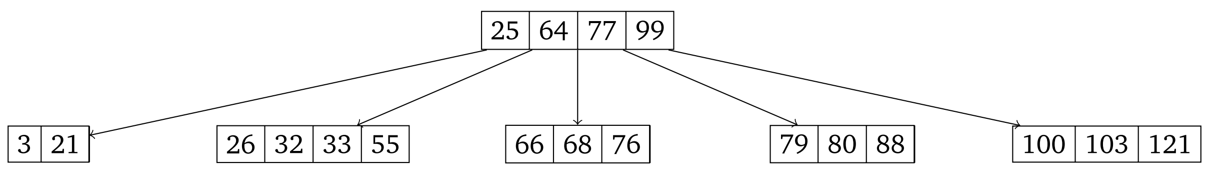
Why are directory structures in an operating system typically maintained using a B-Tree instead of some other data structure?

3D: Free Space Management [2 marks]

Consider a system where free space is kept in a free-space list. Imagine that the pointer to the free-space list is lost; can the system reconstruct the free space list? Explain.

3E: B-Tree Insertion [6 marks]

Assume we have a B-Tree structure to manage inodes which receive unique integer identifiers. The maximum number of elements that can be in a node is 4. Given the state of the B-Tree shown below, show the state of the tree after inserting, in this order, the keys: $1, 6, 84, 44$. The correct answer is worth 6 marks, but showing intermediate steps will result in part marks if your final answer is wrong.



Question 4: Concurrency & Deadlock [28 marks total]

4A: IPC Methods [4 marks]

Compare and contrast the Inter-Process Communication strategies of *Shared Memory* and *Message Passing*.

4B: Deadlock Prevention [6 marks]

Recall that three conditions are necessary for deadlock to be possible: (1) mutual exclusion, (2) hold-and-wait, and (3) no pre-emption. When attempting to perform deadlock prevention, we said that if we can eliminate one of these three conditions, deadlock is impossible. For each of the three conditions, if elimination of the condition can be applied to the dining philosophers problem to prevent deadlock, explain how; if it cannot, explain why not.

4C: Producer-Consumer Implementation [18 marks]

Recall the *Producer-Consumer Problem* from lectures. Two or more threads share a common buffer that is of fixed size. A producer thread generates data and puts it in the buffer. A consumer thread takes data out of the buffer. In this question, you will complete the code to implement the producer and consumer threads as well as add to the `main` function any code necessary to initialize global variables.

Assume there is a buffer (integer array, `int[]`) named `BUFFER` of size `BUFFER_SIZE` which is initially empty. Producers will "produce" an item by calling the function `int produce_item()`, and they place this item in the buffer at a free space. Consumers will take an item out of a filled space in the buffer. The consumer will then set that space in the buffer to `-1`, then "consume" the item by calling the function `void consume_item( int item )`.

Your solution should support multiple concurrent producers and consumers. To accomplish this, recall that you will need one (1) mutex and two (2) general (or counting) semaphores.

The function signatures you will need for mutex operations:

```
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 )
```

When calling `pthread_mutex_init` the constant `NULL` should be given as the attributes (second parameter).

The function signatures you will need for semaphore operations:

```
sem_init( sem_t *semaphore, int shared, unsigned int initialValue)
sem_wait( sem_t *semaphore )
sem_post( sem_t *semaphore )
```

When calling `sem_init`, `0` should be given as the second parameter. Remember that "post" is equivalent to "signal".

Hint: you will want to have two integer variables, one to keep track of the next index for a producer to place an item and one to keep track of the next index for a consumer to read an item.

Complete the code on the next page to implement a race condition-, deadlock-, and starvation-free solution to the producer-consumer problem.

```
int produce_item(); /* Implementation not shown */
void consume_item( int item ); /* Implementation not shown */
void *producer( void *void_arg );
void *consumer( void *void_arg );

/* Declare global variables here */









void *producer( void *void_arg ) {
    /* Implement producer code */


















    pthread_exit(0);
}

void *consumer( void *void_arg ) {
    /* Implement consumer code */















    pthread_exit(0);
}

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

/* Initialize global variables here */











   /* Initialization and creation of the producer / consumer pthreads
      is not shown, but there can be arbitrarily many of each */

    pthread_exit(0);
}
```