Instructions:

1. No aids are permitted except non-programmable calculators.
2. Turn off all communication devices.
3. There are three (3) questions, some with multiple parts. Not all are equally difficult.
4. The exam lasts 80 minutes and there are 70 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. Do not fail this city.
7. 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		3	2a		10	3a		4
1b		4	2b		10	3b		2
1c		2	2c		5	3c		2
1d		6				3d		2
1e		10						
1f		10						
Total								**70**

Question 1: Processes and Threads [35 marks total]

1A. Fork [3 marks]

What would be printed to the console when the following program is executed? Remember, `fork()` can fail.

```
int main( void ) {
    int pid = fork() ;
    printf( "%d \n", pid );
    return 0;
}
```

1B: Inter-Process Communication [4 marks]

Give two reasons why the operating system needs to be involved when two processes want to communicate via shared memory.

1C: POSIX pthreads [2 marks]

What is the difference between a detached thread and a joinable thread, aside from the obvious answer that a detached thread is not joinable and cannot be joined?

1D: Data Races [6 marks]

Consider the following code:

```
void* run1(void* arg) {
    int* x = (int*) arg;
    *x += 1;
}

void* run2(void* arg) {
    int* x = (int*) arg;
    *x += 2;
}
```

```
int main(int argc, char *argv[]) {
    int* x = malloc(sizeof(int));
    *x = 1;
    pthread_t t1, t2;
    pthread_create(&t1, NULL, &run1, x);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, &run2, x);
    pthread_join(t2, NULL);
    printf("%d\n", *x);
    free(x);
    return EXIT_SUCCESS;
}
```

Is there a data race in this program? Your answer should begin with yes or no, followed by your reasoning.

1E: Parallel Array Sum [10 marks]

Imagine there is a very large array of integers and you wish to sum it up. This is a parallelizable task; therefore you can break up this job and run it using multiple threads. If there are n CPUs in the system, you will want to spawn n threads to get this done. In this part of the question, you will write, using C code, the function to run in a newly spawned thread.

Assume there is a globally defined array of integers called `array` that is filled with appropriate values. There are some other constants defined to make this question simpler: `NUM_CPUS` is assigned the number of available CPUs and therefore the number of threads you will spawn. There is also a constant `ARRAY_SIZE` which contains the correct size of the array.

The parameter provided will be a pointer to an integer. The integer will contain the start index for this particular thread. Each thread starts its sum at zero and adds the value at its starting index. The next index to be added to the partial sum will be the starting index plus `NUM_CPUS`. This continues to the end of the array. So, if the constant `NUM_CPUS` is 4, the first thread adds elements at index 0, 4, 8...; the second thread index 1, 5, 9....

To return the result, use the `pthread_exit` routine. This function takes one parameter: a pointer. In this case, the pointer should point to a value for the partial sum of the array that this thread has calculated. Allocate a new integer to store the result and give a pointer to it as input to the `pthread_exit` system call.

Hint 1: remember that memory is allocated in C with `malloc()` and to get the size of memory you want to allocate, there is the `sizeof` keyword. Example: `int* p = malloc( sizeof( int ) );`

Hint 2: Before returning with `pthread_exit`, deallocate the parameter memory with `free()`.

```
void *sum( void *void_arg ) {









}
```

1F: Start Me Up [10 marks]

Having implemented the sum function, it is time to write the `main` function. The array and constants defined in the previous part of the question remain the same.

Recall the pthread function signatures that you will need:

```
pthread_create( pthread_t *thread, const pthread_attr_t *attributes,
                void *(*start_routine)( void * ), void *argument );
pthread_join( pthread_t thread, void **returnValue );
```

The basic outline of how this function should work is as follows: create `NUM_CPUS` threads with `pthread_create`, starting the `sum` function defined in the previous part of this question. Each thread must be provided with a pointer to an integer containing its start index.

After all such threads have been created, `main` should use `pthread_join` on each of the threads it has created and get the return value of that thread. The second parameter of `pthread_join` should be the address of a `void*` pointer, which you can cast to an `int*` pointer. After casting it, take the value and add it to the global sum.

Once all partial sums have been collected and added to produce the global sum, print it to the console using `printf`. Example: `printf("The sum is %d.", sum);`

Hint 1: call `pthread_create` with `NULL` for the attributes (to get the defaults).

Hint 2: the `sum` routine allocated an integer variable and returned it. Remember to `free` that variable when you have finished with it. The `pthread_join` routine's second parameter is updated to point to that variable when the function is finished.

Complete the `main` function on the next page to make this happen.

```
void *sum( void *void_arg );

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

    pthread_t threads[NUM_CPUS];
    void* returnValue;









    pthread_exit(0);
}
```

Question 2: Concurrency and Synchronization [25 marks total]

2A. General and Binary Semaphores [10 marks]

You are on co-op, working on a system where there are no counting semaphores, only binary semaphores. The previous student decided to implement his own version of counting semaphores by building on binary semaphores. There are two semaphores in this system, `sem` and `delay`. Your boss tells you that there is a problem with this implementation:

```
void semWait(semaphore s) {
    wait(sem);
    s--;
    if (s < 0) {
        signal(sem);
        wait(delay);
    } else {
        signal(sem);
    }
}
```

```
void semSignal(semaphore s) {
    wait(sem);
    s++;
    if (s <= 0) {
        signal(delay);
    }
    signal(sem);
}
```

Identify the flaw in the solution and list a sequence of steps to demonstrate the problem. Hint: assume that this is an environment with at least 4 concurrent threads.

2B. General and Binary Semaphores II [10 marks]

Having identified the problem in question 2A, modify the code to fix the problem.

2C. The Starving Philosophers Problem [5 marks]

Remember our poor, unfortunate friends, the dining philosophers: they're at the worst restaurant in the world, because there is only a bowl of rice, five chairs, and five chopsticks (placed around the table). As before, deadlock arises if they all are hungry and attempt to sit down at the table and eat at the same time, because each philosopher picks up the chopstick on his/her left, and then can never proceed because each is waiting for the chopstick on his/her right.

Prove that if there is exactly one left-handed philosopher at the table, i.e., one who tries to pick up the chopstick on his/her right first, deadlock will not occur. Hint: use proof by contradiction; assume there is a deadlock and demonstrate a contradiction.

Question 3: Deadlock [10 marks total]

3A. Resource Allocation Graph [4 marks]

Recall that the fourth criterion for deadlock is a cycle in the resource graph, but a cycle in the graph does not always mean there is a deadlock. Draw a sample resource allocation graph with a cycle but no deadlock.

3B. Banker's Algorithm [2 marks]

Why is the banker's algorithm to prevent deadlock not very useful for a general purpose operating system?

3C. The Ostrich Algorithm [2 marks]

Why do common operating systems like Windows and UNIX not attempt to resolve or report deadlocks?

3D. Deadlock Resolution [2 marks]

List one pro and one con of two of the deadlock resolution strategies discussed in lectures.