Instructions:

1. No aids are permitted. No calculators of any type are permitted.
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 80 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		4	2a		10	3c		3
1b		2	2b		5	3d		5
1c		8	3a		15			
1d		16	3b		12			
Total								**80**

Question 1: Processes and Threads [30 marks total]

1A: Fork [4 marks]

In UNIX, to spawn a new process, the system call is `fork()`. This creates an exact duplicate of the parent. Both the parent and child continue executing at the line after the `fork` system call. At this point, we might want them to do different things. How can we, inside the program, determine if the current process is the parent or child?

1B: Process State [2 marks]

What does it mean for a process to be in a *zombie* state?

1C: Searching with pthreads [8 marks]

There is a linear array of integers and you wish to search the array to find the index of a specific value (any location of it will do). Linear searches are slow, but they are a parallelizable task. You would therefore like to break up this job into 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.

The structure containing the parameters to be passed to the search function is defined as follows:

```
typedef struct parameter {
    int startIndex;
    int endIndex;
    int searchValue;
} parameter_t;
```

Assume there is a globally defined array of integers called `array` that is filled with appropriate values. It may or may not contain the desired search value (passed in the parameters as `searchValue`). You may also assume that the start and end index values are valid and no error checking is needed.

Complete the `search` function below to perform a linear search of the `array` from `startIndex` to `endIndex` as specified in the parameters, to find the index of the search value. If the desired value is not found in this section of the array, then -1 should be the result.

To return the result, we will (ab)use the `pthread_exit` routine. This function takes one parameter: a pointer to an integer (which is normally the status code, but we will use it for the index of the search value).

Hint 1: you will need to allocate a new integer variable to store the result and give a pointer to it as input to the `pthread_exit` system call.

Hint 2: remember that 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 ) );`

Remember, before returning with `pthread_exit`, to deallocate the `parameter_t` memory with `free()`.

```
void *search( void *void_arg ) {









}
```

1D: Completing the Search [16 marks]

Now that we have created the background search function, it is time to define and create the search routine. There is still the globally defined array of integers called `array` that is populated 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. Finally, the value we are searching for is `SEARCH_VALUE`.

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 `search` function defined in the previous part of this question. Each thread must be provided with a `parameters_t` structure indicating the start, end, and search value, for its section. You will also need to decide how to break up the array into more or less equal parts (having one thread's section larger if `ARRAY_SIZE` does not divide evenly by `NUM_CPUS` is expected).

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 later cast to an `int*` pointer. After casting it, assess if the value from dereferencing the pointer is -1; if it is not then the value from dereferencing that pointer is the index of `SEARCH_VALUE`.

If you find the desired search value, print a message to the console with `printf` indicating the index. Example: `printf("Found at %d", *rv);` where `rv` is an integer pointer to the result. If not found, produce no output.

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

Hint 2: the `search` 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 below to make this happen.

```
void *search( void *void_arg );

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

    pthread_t threads[NUM_CPUS];
    void* returnValue;




    pthread_exit(0);
}
```

Question 2: Concurrency and Synchronization [15 marks total]

2A: Mutual Exclusion [10 marks]

The pseudocode below represents a candidate solution to the producer-consumer problem:

Producer

```
[produce item]
added = false
wait( mutex )
while added is false
   if count < BUFFER_SIZE
       [add item to buffer]
       count++
       added = true
   end if
end while
signal( mutex )
```

Consumer

```
removed = false
wait( mutex )
while removed is false
   if count > 0
       [remove item from buffer]
       count--
       removed = true
   end if
end while
signal( mutex )
[consume item]
```

Assess this code to determine whether it is vulnerable to deadlock or starvation. Consider each of those separately. Justify your answers.

2B: Hardware Support [5 marks]

Recall from the lectures the *Test-and-Set* instruction, which, with hardware support, executes atomically (indivisibly). This construct can be used as a building block to accomplish mutual exclusion. Describe, using C or C-like pseudocode, the semantics of this instruction.

Question 3: Deadlock [35 marks total]

3A: Banker's Algorithm [15 marks]

Using the (general) Banker's Algorithm, determine if the state depicted below is safe or unsafe. Show your work.

Resources in Existence

$[15, 6, 9, 10]$

Resources Available

$[6, 3, 5, 4]$

Current Allocations

$$\begin{bmatrix} P_0 & 2 & 0 & 2 & 1 \\ P_1 & 0 & 1 & 1 & 1 \\ P_2 & 4 & 1 & 0 & 2 \\ P_3 & 1 & 0 & 0 & 1 \\ P_4 & 1 & 1 & 0 & 0 \\ P_5 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Maximum Requests

$$\begin{bmatrix} P_0 & 9 & 5 & 5 & 5 \\ P_1 & 2 & 2 & 3 & 3 \\ P_2 & 7 & 5 & 4 & 4 \\ P_3 & 3 & 3 & 3 & 2 \\ P_4 & 5 & 2 & 2 & 1 \\ P_5 & 4 & 4 & 4 & 4 \end{bmatrix}$$

3B: Deadlock Detection [12 marks]

In the real world, the banker's algorithm as executed on the previous page is usually only useful for detecting deadlock rather than avoiding it. In a real computer system, over time, resources may be added or removed, and processes will start and and finish. The algorithm is time consuming (in class we said $\Theta(m \times n^2)$ where m is the number of resources and n the number of processes) so we want to run it only when we absolutely must. If an alteration can be made safely, we do not need to run the deadlock detection algorithm. For each of the following situations, indicate if that change could be made safely, and under what circumstances.

1. A new resource is added to the system.

2. An available resource is permanently removed from the system.

3. Increase the maximum requests for one process (a process requests additional resources).

4. Decrease the maximum requests for one process (a process indicates it will not need some resource).

5. Process creation.

6. Process termination.

3C: Deadlock Recovery [3 marks]

Assume a system's deadlock recovery strategy is *rollback*. Describe, in three (3) points, this strategy.

3D: Two-Phase Dining [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. Suppose the philosophers change their policy to use *two-phase locking*. Explain how it works and demonstrate that it prevents deadlock and allows all philosophers to (eventually) eat.