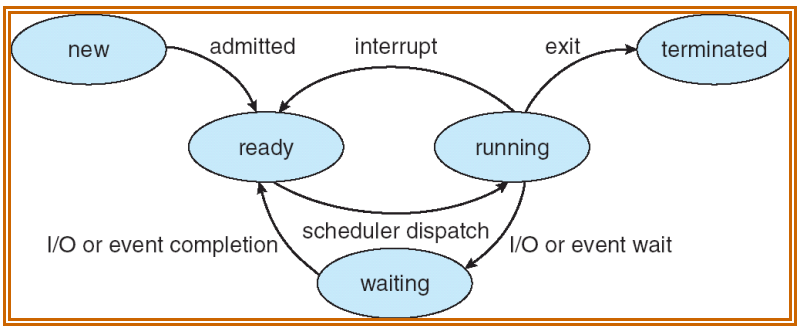
Name: Total: _____/ 67 marks
Student No:_____

1. [15 marks] (Kernel structure) Let us suppose we have a process in a multiprogrammed OS.

What are the possible events that can make a given process enter the Ready Queue? Consider the short, medium, and long term Schedulers, and any other reasons for a process to enter this queue. Explain how the OS Kernel will react to these different events (in terms of the states of the system and the transitions between these states). You can use the space below to draw a state diagram if needed.

Hint: give at least *three* kinds of events to be considered, and their corresponding transitions.

MAX: 15 MARKS



3 marks: 1 mark for recognizing each of the events (new process, interrupt, end of I/O-event)

1 mark: New process: from new to ready
1 mark: there is a PCB free
1 mark: there is memory
1 mark: all the resources needed are available
1 mark: the long-term scheduler has selected the job
1 mark: the system call is implemented as a software interrupt and it produces a context switch
1 mark: we need to update the PCB table
If they considered the mid-term scheduler, all the items above applied to the mid-term scheduler
1 mark: the mid-term scheduler put it back in the ready queue because there are resources available

1 mark: Interrupt: from running to ready
1 mark: the process has freed the CPU and it is now back again at the ready queue (for instance because of a timeout)
1 mark: interrupted when time slice expires (in time-shared system) (Timer ISR)
1 mark: a new process should be picked now (the scheduler must be invoked by the ISR)
1 mark: only in the case of preemptive algorithms (higher priority process arrives)
1 mark: the interrupt stops the process and it produces a context switch
1 mark: we need to update the PCB table

1 mark: End of I/O-event: from waiting to ready

1 mark: the I/O operation that the process is waiting for has finished and an interrupt occurred
1 mark: an event the process was waiting for (e.g. semaphore wait()) has now occurred (semaphore signal())
1 mark: a new process should be picked now (the scheduler must be invoked by the ISR)
1 mark: the interrupt stops the process and it produces a context switch
1 mark: we need to update the PCB table

Other valid answers? 1 mark

2. [12 marks] Short questions. RESPOND IN THE SPACE GIVEN.

a. [3 marks] Briefly explain how the kernel/user modes of operation work.

1 mark: A mode bit is added to the hardware of the computer to indicate the mode of operation.
1 mark: All user programs execute in user mode. When the user application requests a service from the OS, it uses a system call and the mode switches to kernel mode.
1 mark: The hardware allows privileged instructions to be executed only in kernel mode.
1 mark: If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the OS.
1 mark: to protect processes from each other
1 mark: to protect the OS from illegal accesses from user programs
1 mark: to prevent the users to access the hardware directly
1 mark: to provide security between user processes
1 mark: to prevent a process to take the i/o devices (or cpu) forever

Other valid answers? 1 mark

b. [3 marks] Briefly discuss how a Time-Shared system works.

1 mark: we use a timer to prevent processes to take the cpu forever
1 mark: the timer will give a small amount of time (time slice) to each process and will switch to the next one
1 mark: Multiple processes are loaded in main memory, and switched in and out.
1 mark: The job of a user executes within its time slice and once its time slice expires (i.e. notified by a Timer ISR), the currently running process is preempted, a context switch occurs and the next process in the ready queue starts executing. This process continues in a cycle.
1 mark: we have a single powerful cpu, and numerous interactive users
1 mark: interactive users do not use the cpu; most of the time they are working on a terminal not using the processor (I/O most of the time)

Other valid answers? 1 mark

c. [3 marks] What is the Operating System Shell? (or Command Interpreter?) What is the relationship between the Shell, and the OS API?

1.5 marks: what is the OS shell,

- *Allows direct command entry (e.g. via a keyboard input)*
- *Gets a command from the user and executes it*

1.5 marks: relation to the OS API

- *Users can use the Shell to execute commands provided by the OS API, which in turn will execute a system call and perform an OS related service (e.g. in UNIX, use the rm command to delete a file).*

Other valid answers? 1 mark

d. [3 marks]. (Scheduling jobs) What is the Long-Term Scheduler and what it is its relationship with the Job Control Interpreter?

1 mark: Long-term scheduler (or job scheduler) selects the processes that should be brought into the ready queue (loaded in main memory and put in the ready state) from a mass-storage device (e.g. disk).
1 mark: Compared to short-term scheduler executed much less frequently and thus can take more time to execute.
1 mark: Controls the degree of multiprogramming (number of processes in main memory).
1 mark: it was used in batch OS (and still used in the case of batch processing in large computer centers)
1 mark: The Job Control Language Interpreter is responsible for reading and carrying out the Job Control Language instructions from processes that are brought into the ready queue from the mass-storage device.

Other valid answers? 1 mark

3. [30 marks] (Processes)

a) [20 marks] Consider the following template for a Unix-based program:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main ()   {
   pid_t pid;
   int num;


}
```

Write a program in which two processes interchange data through shared memory (or a shared file if you do not remember how to work with shared memory). The parent process writes three numbers in the shared memory/file and the child process reads each of them, adds them, and displays the result on screen. After this, the child process ends. The parent process waits for the child to end, and then ends.

Protect the access to the shared data using semaphores (there is a summary of system calls at the end of the exam, but you can use other services if you remember them).

NOTE: You can obtain partial marks even without using semaphores (we will evaluating the use of the remaining Unix system calls and the overall program organization).

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main () {
    pid_t pid;
    int i=0, numRead=0;

    int semid = semaphoreCreate(0); //create semaphore
    // create shared memory
    int shmid = shmget(IPC_PRIVATE, sizeof(int)*3, IPC_CREAT | 0600);
    int *data = (int*) shmat(shmid, 0, 0);

    pid = fork (); //fork process

    if (pid == -1) {
        printf("Error: could not fork()");
        exit(1);
    } else if (pid == 0) {    // child process
        while (numRead < 3) {
            semaphoreWait(semid,true);
            numRead++;
            //printf("Child process has read %d values.\n",numRead);

        }
        int sum=0;
        for (i=0; i<3; i++)
            sum += data[i];
        printf("The sum is %d.\n", sum);
        exit(0);

    } else {     // parent process
        for (i=0; i < 3; i++) {
            data[i] = i +5;
            //printf("Parent produced %d.\n", data[i]);
            semaphoreSignal(semid);
        }
    }

    wait(NULL); // wait for child process

    //Remove semaphore set
    semaphoreRemove(semid);

    // detach shared memory
    if (shmdt(data) == -1) {
        perror("ERROR: Failed to detach failed memory");
    }
    //release shared memory
    if (shmctl(shmid, IPC_RMID, 0) == -1) {
        perror("ERROR: Releasing shared memory");
    }
    return 0;
}
```

2 marks: opening the file or shared memory before fork OK
2 mark: create and initialize semaphore

3 marks: selection after Fork OK: parent, child, error
2 marks: parent writing OK
5 marks: child reading and ensuring that child reads after parent writes (3 marks), adding, and printing (2 marks) OK
2 marks: use of semaphores (e.g. protecting access to critical section, or synchronizing processes) OK
1 mark: remove semaphore
1 mark: closing file or detaching/removing shared memory OK
2 marks: exiting program (wait until child finishes, e.g. invoke wait()) OK

b) [5 marks] For this example, explain what does the *fork()* system call do. Include ALL the steps carried out by the OS when this system call is executed (with detail).

1 mark: The fork() system call is used to create a new process that is a duplicate of the process that invokes fork().
1 mark: copies the PCB of the parent into a new PCB for the child
1 mark: finds space in memory for the child process
1 mark: copies the process image (e.g. program and data, process context (state, status, resources), stacks) of the parent into the child's memory image
1 mark: returns the child's PID to the parent, and 0 to the child. -1 if error.
1 mark: modifies some necessary information on the PCB (memory location, parent's PID, etc).
Other valid answers? 1 mark

c) [5 marks] Explain what would happen if the parent process would not wait for the child process to end. How is the code modified? And what could happen during the execution of both processes?

1 mark: deleting any "wait"instructions
1 mark: modifications on the semaphores/shared memory as needed
1 mark: discussion on buffer overruns
1 mark: discussion on deadlocks
1 mark: discussion on livelocks
1 mark: discussion on concurrency problems
1 mark: explanation on concurrent execution
1 mark: The parent process would not be able to safely clean up the system resources (semaphores and shared memory).

Other valid answers? 1 mark

4. [10 marks] (Threads) Assume that you need to start executing a new activity in a multithreaded OS. Compare the mechanism for **creation and termination** of such activity using as a *new thread* within the same process, against the creation and termination of the activity as a *new process*. Explain which one is more efficient and why. Explain and justify.

1 mark: Creating a new thread in a process is more efficient than creating a completely new process.

Creation:

- *1 mark: Process creation is more time-consuming and resource intensive compared to thread creation.*
- *1 mark: new process requires allocating memory for process image: program and data, process context (state, status, resources), stacks; the thread shares most of that*
- *1 mark: new process creates a complete new PCB; new thread doesn't*
  - *Initializing Process Control Block (PCB) which contains: id, state, priority, program counter and more…*
- *1 mark: shared information (code, open files, shared variables, shared memory, etc) does not need to be duplicated for threads; only non-shared information is created*
- *1 mark: the PC needs to be duplicated (and the current CPU status) and stored on the PCB for the thread*

Termination:

- *1mark: when a thread is finishing and the rest continue, there's less information to eliminate (it's faster and easier)*
- *1 mark: is this not the last thread to terminate? (different options; in most cases the other threads continue)*
- *1 mark: is this the last thread to terminate? Delete PCB, free memory, etc.*

Other:

- *1 mark: Context switching threads is also typically faster that context switching for processes because it does not require switching PCBs (saving context of one PCB and moving a new PCB from memory).*
- *1 mark: Communication overhead between threads is faster compared to processes because it does not involve the kernel (system calls).*
  - *Threads can communicate using global variables belonging to their process.*
  - *Processes need to use Inter-process communication (IPC) mechanisms such as shared memory or message queues (both involve kernel) to communicate with each other.*

Other valid answers? 1 mark