SURNAME		NAME		Group
ID		Signature		

- **Keep the exam sheets stapled.**
- **Write your answer inside the reserved space.**
- **Use clear and understandable writing. Answer briefly and precisely.**
- **The exam has 7 questions, everyone has its score specified.**

1. In each section, briefly define the first term that is cited and relate to it the sentence in C language or the shell command given below.

(1,5 points = 0,5 + 0,5+0,5)

1	**a)** System call (define)
	System calls are a mechanism for requesting services to the operating system. It is an interface between the applications and the kernel that provides protected access to the system hardware and software resources.
	printf (" This message is printed on the standard output \n"); // in a program (relate)
	To invoke systems calls in a convenient and portable way, in programs written in C, the language provides a collection of functions such as printf() that are part of a collection of standardized libraries.

	b) User and kernel execution modes (define)
	There are different operating modes of the processor. When the system executes a user program, it does so in user mode, in which the execution of certain instructions that provide access to some critical resources is restricted. Accessing these resources is the sole responsibility of the kernel that runs in kernel mode, in which these restrictions do not apply. Kernel mode is entered whenever a kernel service is invoked as a result of an exception or interrupt.
	b= a / 0; // division by zero in a user program (relate)
	The execution of this statement causes a division by zero. It is an error that will cause an exception that will suspend the process, the processor will exit user mode, entering kernel mode and will execute the service routine corresponding to the treatment of this exception.

	c) Timesharing (define) Timesharing systems are multiprogrammed systems in which user-machine interaction is supported, limiting the length of every CPU access, quickly switching between different ready jobs (processes). $ ls -la | wc -l (relate) This command line launches two processes simultaneously, in foreground or interactively. While running they also exchanges a flow of information between them. These processes progress concurrently, alternating in the use of the CPU thanks to the timesharing mechanism.

2. Given the program ***experiment.c*** whose executable file is ***experiment***.

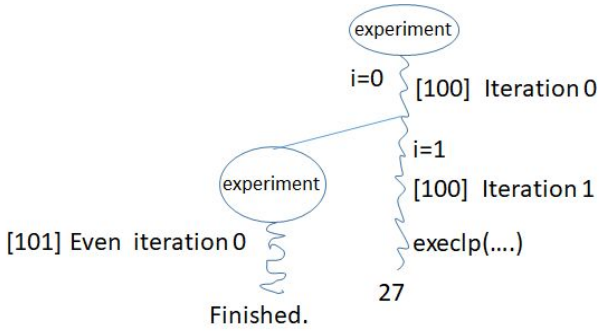
```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <unistd.h>
 4 #include <sys/wait.h>
 5
 6 int main(int argc, char *argv[]) {
 7     pid_t pid;
 8     int i, N=4;
 9
10     for (i=0; i<N; i++) {
11         printf("[%d] Iteration %d\n", getpid(),i);
12         if (i%2 == 0) {
13             pid = fork();
14             if (pid == 0) {
15                 printf("[%d] Even iteration (%d)\n", getpid(),i);
16                 break;
17             } else if (pid == -1) {
18                 printf ("Error creating the process\n");
19                 exit(-1);
20             }
21         } else if (execlp("wc","wc","-l","experiment.c",NULL)<0)
22              printf("Error running wc\n");
23     }
24    while (wait(NULL)!=-1);
25    printf("[%d] Finished.\n", getpid());
26    exit(0);
27 }
```

Nota: ***a % b*** returns the remainder of the integer division ***a / b***.

(1,4 points = 0,7 + 0,7)

a) Supposing that the system calls used do not produce any error, justify the number of processes created and the relationship between them when executing *experiment*.

In the first iteration, i = 0, being i even the condition is met and the parent will fork () creating a child process that will exit the loop and end. In the second iteration, i = 1, the condition is not met and the parent calls to execlp() changing its code for the one take from executable file wc in this case, and then it will end after executing wc. Therefore, there will be only two processes: the initial process and an additional process, the latter's child.



[100] Iteration 0
[101] Even iteration 0
[101] Finished.
[100] Iteration 1
27 // Result from executing command “wc -l experiment.c”

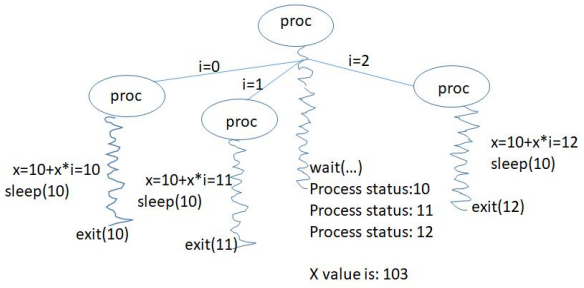
3. Given the ***pro.c*** program whose executable file is ***proc***.

```
 1 #include <stdio.h>
 2 #include <sys/types.h>
 3 #define N 3
 4 int X=1;  //global variable
 5
 6 int main(int argc,char *argv[]) {
 7     pid_t pid;
 8     int i,status;
 9
10     for (i=0; i<N; i++) {
11         pid = fork();
12         if (pid==0) {
13             X=10+X*i;
14             sleep(10);
15             exit(X);
16         }
17     }
18     X=100+i;
19     while (wait(&status)>0) {
20         printf("Process status: %d.\n", status/256);
21     }
22     printf("X value is: %d \n",X);
23     return(0);
24 }
```

(1,2 points = 0,6 + 0,4 + 0.2)

3 **a)** Explain the processes that are created when executing proc, the relationship between them and the message they print in the terminal.

When executing proc, 4 processes are created, one parent and 3 children. Since the children end up doing exit(X) they don't create new processes, while the parent waits for them all.
What is shown in the terminal is:
Process status: 10
Process status: 11
Process status: 12
X value is: 103



b) Given the instructions in lines 13, 14, 15 and 19, mark the state(s) in which the process(es) could be after executing the line sentence and before executing the following line sentence.

Line	Suspended	Ready	Execution	Ended	Zombie	Orphan
13		X	X			
14	X					
15				X		
19	X	X	X			

c) Explain if there may or may not be a *race condition* in the access to global variable X.
When a new process is created with fork (), the new process will have its own copy of variable X, they do not share variable X, and therefore there will be no possible race condition accessing X

4. In a timesharing system its short-term scheduler has two queues, one managed with SRTF (QueueS), in which in case of a tie the arrival order is used as tie-breaking criteria, and another managed with Round Robin (QueueR) with a quantum of 2 time units (q = 2 ut). New processes and those coming from I/O always go to QueueR which is the one with the highest priority and a process is degraded to QueueS after consuming a time quantum in the CPU. Interqueue scheduling relies on preemptive priorities. All I/O operations are performed on a single device with FCFS queue.

This system is requested to execute the following set of jobs:

Process	Arrival time	Execution profile
A	0	5 CPU + 2 I/O + 1 CPU
B	1	4 CPU + 4 I/O + 4 CPU
C	2	3 CPU + 1 I/O + 5 CPU

(2,0 points = 1,4 + 0,6)

4 a) Obtain the execution time line, filling the following table. The processes in the queues have to be sorted according to the scheduling criteria, in such a way that the one on the right is the first. At the instant when a process goes to the CPU don't remove it from the incoming queue just put it between parenthesis, to make clear from where it comes from.

T	Queue-->	QueueS-->	CPU	QueueIOS-->	E/S	Event
0	(A)		A(4)			A arrives
1	B(4)		A(3)			B arrives
2	C(3) (B)	A(3)	B(3)			C arrives
3	C(3)	A(3)	B(2)			
4	(C)	A(3)B(2)	C(2)			
5		A(3)B(2)	C(1)			
6		A(3)B(2) (C)	C(0)			C is the shortest
7		A(3) (B)	B(1)		C(0)	
8	(C)	A(3)B(1)	C(4)			
9		A(3)B(1)	C(3)			
10		C(3)A(3) (B)	B(0)			
11		C(3) (A)	A(2)		B(3)	Tie
12		C(3)	A(1)		B(2)	
13		C(3)	A(0)		B(1)	
14		(C)	C(2)	A(2)	B(0)	
15	(B)	C(2)	B(3)		A(1)	
16		C(2)	B(2)		A(0)	
17	(A)	B(2)C(2)	A(0)			
18		B(2) (C)	C(1)			A ends, Tie
19		B(2)	C(0)			
20			B(1)			C ends
21			B(0)			
22						B ends
23						

4 b) Obtain for every process its waiting time and its turnaround time.

	Waiting time	Turnaround time	
A	A	**9**	18-0 = **18**
B	B	**9**	22-1 = **21**
C	C	**9**	20-2 = **18**

---

5. Given the following program ***endth.c*** which executable is ***endth***.

(1,1 points = 0,8 + 0,3)

```
#include <... all headers...>
pthread_t  th1, th2;
pthread_attr_t atrib;
int N=0;

void *Func2(void *arg) {
    int j=2;
    sleep(5);
    printf("END THREAD 2 \n");
    pthread_exit(&j);
}
```

```
void *Func1(void *arg) {
    int j=1;
    pthread_create(&th2, &atrib,Func2,NULL);
    N=N+j;
    sleep(20);
    printf("END THREAD 1 \n");
    pthread_exit(&j);
}
```

```
int main (int argc, char *argv[]) {
    int b;

    printf("START MAIN \n");
    pthread_attr_init(&atrib);
    pthread_create(&th1, &atrib, Func1,NULL);
    N= N+10;
    sleep(10);
    pthread_join(th2,(void**) &b);
    printf("END MAIN \n");
    exit(0);
}
```

5 **a)** Indicate the strings printed by the program in the Terminal after its execution. Explain your answer.

START MAIN
END THREAD 2
END MAIN

The main thread ends executing exit() and so the whole process ends. Therefore, the process ends before thread th1 finishes after executing printf(). So this printf() from th1 is not printed.

b) Explain whether or not there is a possibility of a race condition on variables N, j and b declared in the program.

Among these variables the only one shared is N, it is shared by threads th1 and the main thread and therefore it is the only variable that could produce a race condition. Variables j and b are local to the threads and therefore each one has its copy.

6. Indicate for every one of the following statements about the critical section if it is true or false. Explain your answer:

(1,2 points = 0,4 + 0,4 + 0,4)

6

a) If a semaphore-based protocol is used to solve the critical section problem, as shown below, the initial value of the semaphore S must be 0.

```
Input protocol → P(S)
                 Critical section();
Output protocol → V(S)
```

FALSE

To solve the critical section problem the semaphore must be initialized to 1, in order to meet the condition of mutual exclusion, progress and limited waiting. If the semaphore is initialized to 0 then there is a deadlock.

b) The hardware solution based on Disable/Enable interrupts, shown in the following code, can not always be used.

```
Input protocol → DI (Disable)
                 Critical section();
Output protocol → EI (Enable)
```

TRUE

The hardware solution based on the DI / EI instructions is only valid for kernel activities, since these instructions are privileged and can only be executed in kernel mode. Furthermore, this solution would not be suitable for systems with more than one CPU, since they should be disabled in all the cores.

c) The solution based on test_and_set instruction, shown next, fulfills the conditions of mutual exclusion, progress and limited waiting, for any scheduling policy.

```
// key is a global variable initialized to 0
Input protocol → while (test_and_set(&key));
                 Critical section();
Output protocol → key = 0;
```

FALSE

Test & Set solution if based on active waiting in the input protocol to the critical section, and so it is very dependent on the scheduler policy. This can produce even deadlocks in unfair policies. This solution does not meet the limited waiting requirement, although it does meet the mutual exclusion and progress.

7. The following code is a variation of the "addrows" program seen in threads lab session (5). In this version, the adder threads (based on function AddRow) are the ones that perform the addition of row partial additions into **total_addition**, instead of the main thread. To achieve this the the following changes have been done on the code:

- The **addition** element has been removed from the **row** structure .
- Variable **total_addition** has been declared global .
- Local variable **local_addition** has also been declared inside **AddRow** .
- Mutex **m** has been declared .

```
1   #include <...all_headers...>
2   #define DIMROW 1000000
3   #define NUMROWS 20
4   typedef struct row{
5        int vector[DIMROW];
6   } row;
7   row matrix[NUMROWS];
8   long total_addition = 0;
9   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
11
12  void *AddRow( void *ptr ) {
13     int k;
14     int local_addition;
15     row *fi;
16     fi=(row *)  ptr;
17     local_addition=0;
18     for(k=0;k<DIMROW;k++) {
19          total_addition += fi->vector[k];
20     }
21  }
22  int main() {
23     int i,j;
24     pthread_t  threads[NUMROWS];
25     for(i=0;i<NUMROWS;i++) {
26          for(j=0;j<DIMROW;j++) {
27              matrix[i].vector[j]=1;
28          }
29     }
30     pthread_attr_init( &atrib );
31     for(i=0;i<NUMROWS;i++) {
32         pthread_create(&threads[i],NULL,AddRow,(void *)&matrix[i]);
33     }
34     for(i=0;i<NUMROWS;i++) {
35        pthread_join(threads[i],NULL);
36     }
37     printf(  "Total addition is: %ld\n",total_addition);
38  }
```

Answer the following questions:

(1,6 points = 0,4 + 0,6 + 0,6)

7	**a)** Identify in the program the line(s) corresponding to the critical section. The line with critical section is 19, since in it a shared global variable is accessed by several threads: `(total_addition += fi->vector[k];)`

b) Use mutex **m** to protect the critical section and explain what disadvantage may arise in relation to the concurrency level with this solution.

```
for(k = 0; k < DIMROW; k++) {
    pthread_mutex_lock(&m);
    total_addition += fi->vector[k];
    pthread_mutex_unlock(&m);
}
```

This solution is not very efficient since the critical section is continuously accessed for each element to be added, which severely limits the concurrence in the threads execution.

c) Regardless of the changes made in section b), write the required code to modify the program so that: continue to make the addition within the loop but using the variable **local_addition**, instead of total_addition, and after the loop local_addition is added to total_addition. If necessary, in your solution, protect the critical section and explain what advantage this solution has over the one in section b .

```
for(k = 0; k < DIMROW; k++) {
    local_addition += fi->vector[k];
}
pthread_mutex_lock(&m);
total_addition += local_addition;
pthread_mutex_unlock(&m);
```

It is a more efficient solution than the one in section b). By accessing only the shared variable after the addition, the number of accesses to the critical section is drastically reduced and therefore there is a greater amount of concurrence between the threads that add up.