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. Answer the following questions in a short but clear way:

(1.4 points = 4 x 0.35)

1	a) ¿What is understood by CPU and I/O concurrency?. There may be a process running on the CPU and simultaneously another or other processes may be serviced by the I / O devices. These hardware elements can operate in parallel.
	b) ¿Why processors have two execution modes?¿which are they? Processors have to provide at least two modes of operation. One mode allow executing all available instructions and it is usually called kernel or protected mode. In the other mode certain instructions are restricted (interrupts control, access to protected memory areas, some processor registers, etc), it is the so-called user or regular mode. This allows the Operating System to have full machine control when running in protected mode, restricting access to hardware to user programs that run in regular mode.
	c) ¿Which of the following elements belong to the operating system kernel and which don't? ***process manager*** / ***shell*** / ***memory manager*** / ***device handler*** / ***ls command*** / ***system calls interface*** / ***internet navigator*** ***Belong to OS kernel***: *process manager / memory manager / device handler / system call interface* ***Don't belong to OS kernel***: *shell / ls command / internet browser*
	d) ¿What is CPU utilization? write the formula that computes it CPU utilization, along a given time interval T, is defined as the percentage of time that the CPU has been executing instructions (not idle), named T_e. So, CPU utilization is computed as follows: $U = T_e / T$

2. Consider that there is a file "hello.txt" in the current working directory, containing the text "hello \n", so that command "cat hello.txt" prints a line with the word hello. Assume that the following program is compiled and executed, for every value of X = 1, 2, 3 and 4. Expose for each of 4 cases (values of X), what is displayed on the terminal when the program is executed and explain your answers.

```
 1 #include <...all headers...>
 2 #define X 1    //1,2,3,4
 3
 4 int main(int argc, char *argv[]) {
 5     int val = 0;
 6     int parent_pid = getpid();
 7
 8     if (X >= 3) val = fork();
 9     if (val == 0) {
10         if (X%2 == 1) // X odd
11             execl("/bin/cat", "cat", "hello.txt", NULL);
12         else
13             execl("/cat/bin", "cat", "hello.txt", NULL);
14     }
15
16     if (getpid() == parent_pid)
17         printf("parent\n");
18     else
19         printf("child\n");
20     return 0;
   }
```

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

(1.4 points = 0,35+0,35+0,35+0,35)

	a) Case X=1 (#define X 1) `hello` No children are created. The cat program runs successfully. The execl call does not return.
	b) Case X=2 (#define X 2) `parent` No children are created. The execution of cat fails because the path is not correct. The execl call returns with error -1. The process finally prints "parent"

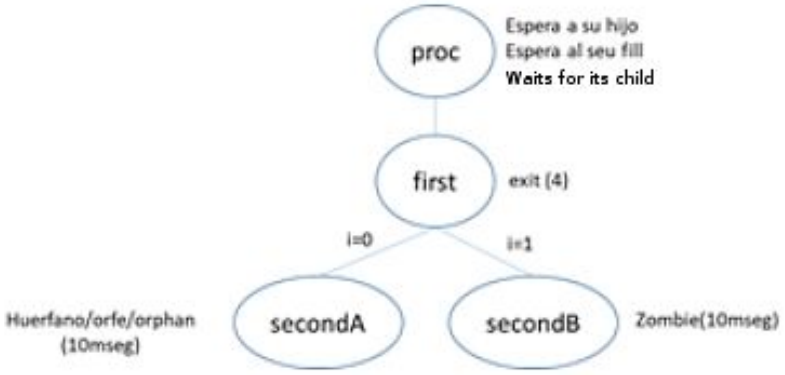
	c) Case X=3 (#define X 3)
	`hello`
	`parent`
	
	A child is created that executes cat successfully. The execl call does not return. Only the parent reaches line 15 and prints "parent".

	d) Case X=4 (#define X 4)
	`parent`
	`child`
	
	A child is created. The execution of cat fails because the path is not correct. The execl call returns with error -1. Both parent and child reach line 15.

3. Complete the following code for *proc.c* which executable file will be named ***proc,*** in such a way that:

- When executing ***proc*** a new process ***first*** will be created, which in turn will create two children ***secondA*** and ***secondB***.
- Process ***proc*** has to wait for its child ***first*** and ***first*** has to finish with ***exit(4)***.
- Processes ***secondA*** will endup orphan and ***secondB*** zombie along, at least, 10 miliseconds. To achieve this it is suggested to use ***sleep(),*** with ***sleep(10)***, ***sleep(20)*** y ***sleep(30),*** when considered appropriate.

The following diagram shows the parent-child relations and processes actions.



(1.1 points)

3.

```
//Program proc.c
#include <all required.h>
int main(int argc, char *argv[]) {
    int val1, val2, status;
    int i, pid;
    printf("Process proc\n");
    val1 = fork();

    if (val1 == 0) {
        printf("I am the child \n");
        for (i=0; i<2; i++) {
            printf("Second generation \n");
            val2 = fork();
            if (val2 == 0) {
                if (i==0) {
                    printf("Orphan child \n");
                    sleep(30); exit(0);
                }
                if (i == 1) {
                    printf("Zombie child \n");
                    sleep(10); exit(0);
                }
            }
         }//for

         sleep(20);
         exit(4);
    } //if val1



    while ((pid=wait(&status)) > 0)
        printf("hijo esperado %d, estado %d\n", pid, status/256);
    exit(0);
}
```

4. In a timesharing system there is a single I/O device that is managed by a FCFS queue. To this system 3 processes arrive: A, B and C. Their arrival instants, priority (being 1 the highest priority, 3 the lowest) and standing alone processing sequences are on the following table:

Process	Arrival	Priority	Standing alone processing sequence
A	0	3 (-)	4 CPU + 1 I/O + 3 CPU + 1 I/O + 4 CPU
B	1	2	2 CPU + 2 I/O + 3 CPU + 1 I/O + 3 CPU
C	2	1 (+)	1 CPU + 5 I/O + 1 CPU + 5 I/O + 1 CPU

(2.1 points = 1.1+ 0.3+0.3+0.4)

4 a) Fill the following table with the processing evolution considering the ready queue scheduled by **preemptive priorities**. In every visit to CPU and I/O indicate the remaining time for that burst.

T	Ready	CPU	I/O queue	I/O	Comment
0		A(4)			A arrives
1	A	B(2)			B arrives
2	B,A	C(1)			C arrives
3	A	B(1)		C(5)	
4		A(3)	B	C(4)	
5		A(2)	B	C(3)	
6		A(1)	B	C(2)	
7			A,B	C(1)	
8		C(1)	A	B(2)	
9			C,A	B(1)	
10		B(2)	C	A(1)	
11	A	B(2)		C(5)	
12	A	B(1)		C(4)	
13		A(3)	B	C(3)	
14		A(2)	B	C(2)	
15		A(1)	B	C(1)	
16		C(1)	A	B(1)	
17		B(3)		A(1)	Fin C
18	A	B(2)			
19	A	B(1)			
20		A(4)			Fin B
21		A(3)			
22		A(2)			
23		A(1)			
24					Fin A
25					

4 b) Indicate the waiting time (CPU queue) and the turnaround time for every process (**Table PRIO_1**)

PRIO_1	Waiting time	Turnaround time
A	**7**	24-0 = **24**
B	**1**	20-1 = **19**
C	**0**	17-2 = **15**

4 c) Using the same preemptive priority policy on the ready queue scheduler, on the same workload, the following waiting times and turnaround times are obtained. Determine what priorities have been assigned to the processes to obtain these values. . (**Table PRIO_2**)

PRIO_2	T.espera	T.retorno
A	0	13
B	7	26
C	14	28

Priorities have to be inverted: A = 1; B = 2, C = 3

4 d) The following table shows the results obtained for the same workload using a Round-Robin policy with quantum q = 2ut. **(Table RR)**

RR	T.espera	T.retorno
A	8	22
B	8	19
C	6	21

Relying on tables PRIO_1, PRIO_2 and RR, compute the throughput obtained on every one of the three scheduling policies.

Througput:
PRIO1 = 3/24 = 0,125 jobs/tu;
PRIO2 = 3/30 = 0,1 jobs/tu; (*)
RR = 3/23 = 0,13 jobs/tu;(*)

From the results on tables PRIO_1, PRIO_2 and RR, what type of processes have benefited on every policy?

Type of processes optimized by every policy:
PRIO1: I/O bound
PRIO2: CPU bound
RR: Balanced CPU&I/O

(*) Note: Process C is last ending job for both PRIO2 and RR. It has a turnaround time of 28 with PRIO2 and 21 with RR. So, the workload processing times are: 28+2 = 30 and 21+2 = 23, respectively

5. Given the following program ***Threads.c*** whose executable file is ***Threads***, answer the following items:

(1.5 points = 0.6 +0.3+0.3+0.3)

```
#include <... all headers...>
#define NTHREADS 3
pthread_t Th[NTHREADS];
pthread_attr_t atrib;
int N=0;

void *Func(void *arg)
{ int i= (int) arg;
  N=N+i;
  printf("Thread %d,N = %d\n",i ,N);
  pthread_exit(0);
}
```

```
int main (int argc, char *argv[]) {
  pid_t pid;
  int i;
  printf("START MAIN \n");
  pthread_attr_init(&atrib);

  pid=fork();
  for (i=0;i<NTHREADS;i++)
   {pthread_create(&Th[i],&atrib, Func,(void *)i);
    pthread_join(Th[i],NULL); }

  if (pid == 0) pthread_exit(0);
  else exit(0);
  printf("END MAIN \n");
}
```

5	**a)** Indicate the sequence (one of the possible ones) that the program prints on the Terminal when executing it. Explain your answer. START MAIN START MAIN Thread 0,N=0 Thread 1,N=1 Thread 2,N=3 Thread 0,N=0 Thread 1,N=1 Thread 2,N=3 The threads created by both processes are launched sequentially, because until one isn't finished the next one is not created. This is why all threads end correctly. The two processes run concurrently so what each one prints could be interleaved with what the other one prints.
	b) Indicate the maximum number of threads that could be running concurrently when executing the ***Threads*** program is executed. Explain your answer. The launching of threads by each process is carried out sequentially, since each time a thread is created it is joined, so execution does not continue until that thread ends. The two processes are executed concurrently and each of them will execute the main thread and one of the threads created within the loop concurrently. Therefore, the maximum number of threads that will be executed concurrently will be 4, 2 for each process.
	c) Indicate if there is a risk of race condition when running the ***Threads*** program. Explain your answer. There is no risk of race condition since access to the global variable N by each thread is done sequentially. And as for the two processes generated each of them has its own copy of variable N.
	d) Assume that the threads on the ***Threads*** process are user level (runtime) threads, what the system's scheduler should manage in that case?. Explain your answer. If the threads are not supported by the kernel then the short-term scheduler, that manages the CPU, will only see two processes competing for the CPU.

6. On a computer with a single CPU running an operating system with a Round-Robin scheduler, it is desired to solve the problem of accessing to a critical section with ONLY TWO THREADS INVOLVED, relying on **variants of the test_and_set()** approach. Explain for every case proposed if the limited waiting condition is met and if it can be considered as active waiting or non-active waiting:

(1.2 points = 0.4+0.4+0.4)

6

a)

Input protocol:	Output protocol:
`while (test_and_set(&key)) /*bucle vacio*/ ;`	`key = 0;`

Does it always verify limited waiting?

It does not meet limited waiting, when a thread leaves its critical section it could re-enter before its quantum expires and it maybe it will expire again being within the critical section. This could be repeated indefinitely.

Is it active waiting?

It is clearly active waiting since there is a wait in a loop consuming all the whole quantum assigned.

b)

Input protocol:	Output protocol:
`while (test_and_set(&key)) usleep(100) ;`	`key = 0;`

Does it always verify limited waiting?

It still does not meet limited waiting, when a thread leaves its critical section it could re-enter before its quantum expires and it maybe it will expire again being within the critical section. This could be repeated indefinitely.

Is it active waiting?

It is not active waiting as the waiting thread waits in suspended state, letting the remaining threads to use the CPU.

c)

Input protocol:	Output protocol:
`while (test_and_set(&key)) usleep(100);`	`Key = 0;` `usleep(105);`

Does it always verify limited waiting?

It complies with limited waiting since while one thread is in the critical section and the other thread arrives, it will be suspended for 100 useg and when the thread in the critical section leaves it it will be suspended for 105 useg, guaranteeing that the one who is waiting to enter will “wake up” before and therefore will be able to enter the critical section. The problem is that we unnecessarily suspend a thread that is no longer in the critical section.

Is it active waiting?

It is not active waiting as the waiting thread waits in suspended state, letting the remaining threads to use the CPU.

7. Describe a possible sequence of the concurrent execution of threads ThA, ThB and ThC.

```
// Initial values
 int x=0, y=0;
 Semaphore: S1=0, S2=1, S3=0;
```

ThA	ThB	ThC

P(S3);
P(S2);
x = x + 1;
y = 2*x + y;
V(S2);
V(S1);
	P(S1);
	P(S2);
	x = 2*x;
	y = x + y;
	V(S2);
		P(S2);
		x = x - 2;
		y = x - y:
		V(S2);
		V(S3);

Use the following table to keep track of the threads operations, as well as the values of variables and semaphores.

(1.3 points)

7

	ThA	ThB	ThC	S1=0	S2=1	S3=0	x=0	y=0
1	P(S3)	P(S1)	P(S2)	-1	0	-1	0	0
2			x=x-2				-2	
3			y=x-y					-2
4			V(S2)		1			
5			V(S3)			0		
6	P(S2)				0			
7	x=x+1						-1	
8	y=2x+y							-4
9	V(S2)				1			
10	V(S1)			0				
11		P(S2)			0			
12		x=2*x					-2	
13		y=x+y						-6
14		V(S2)		0	1	0	-2	-6
15								
16								