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 9 questions, everyone has its score specified.**

---

1. A computer has a workload consisting of long CPU intensive processes, long processes in which I/O predominates and short processes that mainly do CPU use. Do throughput and mean turnaround time improve when going from monoprogramming to multiprogramming with FCFS scheduling? And what happens when going from multiprogramming with FCFS to time-sharing with round-robin scheduling? Justify your answer by filling in the following table

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

1	Throughput	Mean turnaround time
From MONOprogramming a MULTIprogramming	(0,25) Throughput IMPROVES as multiprogramming allows overlapping CPU and input-output activity between multiple processes running concurrently, with this both the CPU and the input-output devices have greater utilization and therefore the system delivers more jobs per time unit..	(0,25) In a multiprogramming system jobs don't need necessarily to wait on the system input queue, waiting happens only in queues associated with the CPU and input-output devices. The corresponding waiting times will be lower than the ones we have on a monoprogramming system due to the simultaneous use of the CPU and input output devices that multiprogramming allows. Therefore the mean turnaround time time IMPROVES.
From MULTIprogramming to time sharing	(0,25) From the point of view of throughput a timesharing system is equivalent to a multiprogramming system, because limiting the time that a process can use the CPU once assigned to it, will not change the CPU utilization value so throughput DOESN'T IMPROVE	(0,25) Limiting the time that a process can use the CPU when it gets it avoids jobs with long CPU bursts monopolizing the CPU, giving the chance to short jobs to enter the CPU before, and therefore reducing a lot their waiting time. The waiting time that this causes on jobs with long CPU accesses is relatively short so that the mean turnaround time IMPROVES

2. Be a computer system having a multiprogramming operating system. Explain why there is a relationship between the following concepts:

(0,5 points=0,25+0,25)

2	**a)** Interrupt and system call The relationship between interrupts and system calls is that system calls are an specific type of interrupts called TRAPs
	b) System call and processor operation modes The processor operating modes allow that only the operating system is able to run privileged instructions that access to system resources. User processes can access resources but they must ask access to the operating system through the system call interface, that generate a software interrupt (TRAP) that, through an operation mode change, assures that is is the operating system the one that runs the code associated with the task.

3. Given the following code that generates the executable file "Test ":

```
1  /*** Test.c ***/
2  #include "all required headers"
3  #define N 4
4  main() {
5    int i = 0;
6    int pid;
7
8    for (i = 0; i < N; i++) {
9      pid = fork();
10     if (pid == 0) {
11       printf("Value of i = %d \n",i);
12       sleep(i);
13       exit(1);
14     }
15     sleep(2*i);
16   }
17   exit(0);
18 }
```

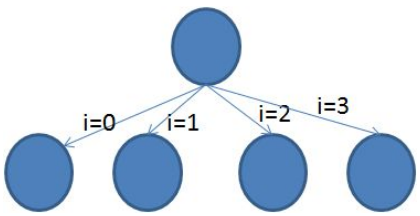
Considering that Test executes without errors answer the following items:

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

3

a) Explain the number of processes that are generated and the existing parentship between them

A total of 5 processes are created, four children and a single parent. After printf the children do sleep and then exit (1), therefore the children don't become parents.



b) Indicate for each of the generated processes if it could be orphan or zombie

Children become zombies for a short time interval. When a child completes the parent is not waiting for it, instead it is executing code or suspended. The child created with i = 0 is the one that remains more time as zombie.

There are no orphans as the father ends after the children have finished.

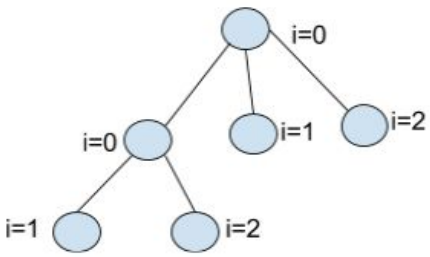
4. The following code corresponds to the executable file generated with the name "Example1".

```
/*** Example1.c ***/
#include "all required headers"
#define N 3
main() {
 int i = 0;
 pid_t pid;

 while (i < N) {
   pid = fork();
   if (pid == 0) {
     printf("Message 1: i = %d \n",i);
     if (i == N-2) {
       execl("/bin/ps","ps","-la", NULL);
       printf("Message 2: i = %d \n",i);
       exit(1);
     }
   } else {
     printf("Message 3: i = %d \n",i);
     while (wait(NULL) != -1);
   }
   i++;
 }
 printf("Message 4: i = %d \n",i);
 exit(0);
}
```

Suppose "Example1" is executed without errors : **(1,5 points=0,75+0,75)**

4a Explain the number of processes created and draw the process parentship diagram



Six processes are generated. The original process generates a child i = 0. From there each process (parents and children) generated a son in each iteration (i = 1), (i = 2). Only processes generated at iteration (i = 1) does not create any child processes that run an exec and run the ps command

4 b Indicate what values of "i" will be printed for every message every time it is printed

Message 1	i=0, i=1, i=1, i=2, i=2
Message 2	-
Message 3	i=0, i=1, i=1, i=2, i=2
Message 4	i=3, i=3, i=3, i=3

5. In a computer two processes A and B start simultaneously, with the following execution profiles:

A: CPU (2 seconds)

B: CPU (1 second) + E/S (1 second) + CPU (1 second)

Considering that the workload due to other processes is negligible, what is the turnaround time and the waiting time of every process under the following scheduling policies?

(0,8 points)

5					
		Turnaround time		Waiting time	
		A	B	A	B
	Preemptive priorities B>A	4	3	2	0
	Round Robin (q=1 ms)	3	4	1	1

6. Indicate what state or states: New (N), Ready (P), Execution (E), Suspended (S) and Terminated (T) can be a process when the following situations occur :

(0,7 points)

6		N	P	E	S	T
	A process that is going to do a system call			X		
	A process that is accessing an I/O device				X	
	A process that has just ended a CPU quantum		X			
	A process that has just completed its first CPU burst				X	
	A process that has just executed a wait () call			X	X	
	A process that has just executed an exit() call					X
	A process that has just been created with a fork() call		X	X		

7. The short-term scheduler of a time sharing operating system manages interactive processes (PI) and process started through the net (PR). This scheduler is based on two queues: QueuePI and QueuePR. Scheduling between queues is preemptive priorities, being QueuePI the more priority one. Every queue has the following own scheduling policy: QueuePI is FCFS and QueuePR is round-robin with q = 1ut. PI processes always go to QueuePI and PR processes always go to QueuePR. You have to consider that the order of arrival to process queues is: first new process, then process coming from I/O and finally process coming from CPU. All I/O accesses are done on a single I/O device with FCFS scheduling policy.

This system executes the following jobs: :

Process	Arrival time	Type	Execution profile
AI	0	PI	2CPU+4I/O+1CPU
BR	1	PR	4CPU+1I/O+ 5CPU
CR	3	PR	6CPU+2I/O+ 1CPU
DI	5	PI	3CPU+3I/O+1CPU

Obtain the execution timeline filling the following table: . **(1,5 points)**

T	QueuePR	QuejePI	CPU	I/O queue	I/O	Comments
0		(A)	A			A arrives
1	B		A			B arrives
2	(B)		B		A	
3	B, (C)		C		A	C arrives
4	C, (B)		B		A	
5	B,C	(D)	D		A	D arrives
6	B,C	A	D			
7	B,C	A	D			
8	B,C	(A)	A		D	
9	B, (C)		C		D	A ends
10	C, (B)		B		D	
11	B,C	(D)	D			
12	B, (C)		C			D ends
13	C, (B)		B			
14	(C)		C		B	
15	C, (B)		B			
16	B, (C)		C			
17	C, (B)		B			
18	B, (C)		C			
19	(B)		B		C	
20			B		C	
21	B, (C)		C			
22	(B)		B			C ends
23						B ends

8. The following program (incomplete) creates an image "img" as a two-dimensional array (the value of each element in the array corresponds to the brightness of each pixel) and it applies processing to get the negative version of the image (using function "Negative") and it applies another processing that convert the image into a binary one (using function "Binarize"). Negative and Binarize are thread functions that are written in such a way that they have to receive a pointer to the first pixel of the half image to be transformed (that is, index [0][0] to the upper half of the image and index [ROWS / 2][0] to the bottom half), in order to transform the half image that starts from the passed pixel.

```
 1 #include <stdio.h>
 2 #include <pthread.h>
 3 #define COLUMNS 1920
 4 #define ROWS 1080
 5 #define byte unsigned char
 6
 7 byte img[ROWS][COLUMNS];
 8 pthread_attr_t attrib;
 9 pthread_t thread_negative[2];
10 pthread_t thread_binarize[2];
11
12 void *Negative(void *ptr){
13   byte *p= (byte*)ptr;
14   int i;
15   for(i=0;i<COLUMNS*ROWS/2;i++,p++){
16     //do negative of pixel pointed by p
17     *p= 255-*p;
18   }
19 }
20 void *Binarize(void *ptr){
21   byte *p= (byte*)ptr;
22   int i;
23   for(i=0;i<COLUMNS*ROWS/2;i++,p++){
24     //binarize pixel pointed by p
25     *p= *p>127? 255 : 0;
26   }
27 }
```

```
28 int main()
29 {
30  int x,y,i;
31  //Init image
32  for(y=0;y<ROWS;y++)
33    for(x=0;x<COLUMNS;x++)
34      img[y][x]=
35 255*(x+y)/(ROWS+COLUMNS);
36
37  pthread_attr_init(&attrib);
38
39  //Start concurrent Negative threads
40  for(i=0;i<2;i++)
41    //{ ... }
42
43  //Wait for Negative threads ending
44  for(i=0;i<2;i++)
45    //{ ... }
46
47  //Start concurrent Binarize threads
48  for(i=0;i<2;i++)
49    //{ ... }
50
51  // ...
   }
```

Note. Some POSIX threads functions templates

int pthread_attr_init(pthread_attr_t *attr); int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_create (pthread_t *thread**, const pthread_attr_t** *attr**, void *(**start_routine**)(void*), void** *arg**);**
int pthread_exit(void *exit_status**); int pthread_join(pthread_t** thread**, void **** exit_status**);**

Answer the following items, using variables already declared in the program:

(1,5 points=0,25+0,25+0,25+0,25+0,5)

8

a) Enter the required line of code on line 40 to concurrently launch Negative threads necessary to perform the negative transformation of the full image .

```
pthread_create(&thread_negative[i], &attrib, Negative,
(void*)&img[i*ROWS/2][0]);
```

b) Write the line of code needed on line 44 to await for Negative threads .

```
pthread_join(thread_negative[i], NULL);
```

c) Enter the necessary code on line 50 to acheive the proper program ending .

```
pthread_exit(NULL);
```

or

```
for(i=0;i<2;i++)
 pthread_join(thread_binarize[i], NULL);
```

d) What is the maximum number of threads belonging to this process that can get to run concurrently?

3 threads: the main thread (función main) and two threads running function Negative (or two threads running function Binarize, when the ones running Negative have finished).

e) Will the transformation of the image be properly completed if the waiting code on lines 47 and 48 was missing? Note that it is the same to do first the negative of a pixel and then the binarization or in reverse order , the resulting image would be the same.

Waiting is required to make sure threads running Binarize and Negative aren't concurrently working on the same half of the picture, as they may try to operate simultaneously on the same element of the image array, allowing race conditions to happen. Furthermore, if c) is solved with the second option (waiting for Binarize threads), removing waits for threads Negative may drive those threads to not finishing and so the image transformation would not be properly done.

9. This code adds the elements of an array of dimension NxM, and for that purpose N threads are created. Each thread adds a row of the matrix and after ending the addition the value of variable "total_add" is updated . **(1,5 points=0,75+0,75)**

```
1   float Mat[N][M];
2   float total_add = 0;
3   int key = 0;
4
5   void *add_row (void *arg) {
6     int col;
7     int row = (int)arg;
8     float add;
9     while(test_and_set(&key));
10
11    for (col =0;col < M; col++) {
12      add = add+Mat[row][col];
13    }
14
15    total_add = total_add + add;
16    key = 0;
17
18  }
```

```
19  int main() {
20    pthread_attr_t attr;
21    pthread_attr_init(&attr);
22    pthread_t t[N];
23    int i;
24
25    for (i = 0; i < N; i++) {
26      pthread_create(&t[i],&attr,add_row,i);
27    }
28    for (i = 0; i < N; i++) {
29      pthread_join(t[i], NULL);
30    }
31    printf("Addition = %f\n", total_add);
32  }
```

a) Indicate which of the following statements are true (T) and which are false (F) (Note: One error voids one correct answer) .

9a		**T/F**
	The proposed solution ensures that the code is free of race conditions	T
	To avoid race conditions, it is essential to protect lines 11 and 13 to prevent concurrent access to variable Mat	F
	The execution of this program will create N threads, and it will concurrently perform the addition of the rows	F
	The variable key indicates the number of threads that are on the critical section	T
	The test_and_set (& key) function checks and changes the value of variable key atomically	T

b) Modify the code above in order to use a semaphore S to synchronize activities, instead of Test_and_set. The resulting code has to compute the addition of the rows concurrently. Indicate first what lines have to commented, to remove the test_and_set based solution. Then, you have to indicate the instructions to be added and at what lines. The semaphore has to be declared and initialized properly. You can choose using Dijkstra notation or POSIX for semaphore operations.

9 b	Lines 3, 9 and 16 are commented The following lines are added: Line 4: Sem_t S; ó Sem_t S(1) Line 14: sem_wait(S) ó P(S) Line 17: sem_post(S) ó V(S) Líne 24: sem_init(&S, 0,1) ó S=1; or omitted if it is initialized in to 1 on líne 4.