UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática de Sistemas y Computadoras (DISCA)

fSO

Ejercicio de Evaluación

05 de Noviembre de 2012

etsinf Escola Tècnica Superior d'Enginyeria Informàtica

SURNAMES		NAME		Group
DNI		Signature		

- **Keep exam sheets stapled.**
- **Answer only in the space reserved for this purpose.**
- **Use clear and readable writting. Answer briefly and accurately.**
- **The exam has 9 questions, everyone indicates its grade.**

1) Can you create a user application that does not do any system call? Explain the answer. **0,75 points**

1

It won't be possible.

User applications need accessing disk data (read/write sytem calls) and shown information on the screen (write call and default output device). Even a "computation only" application has to finish and so it has to call to system call "exit()" in order to aware the OS about releasing system resources taken by the process. So it is impossible to implement a useful application that doen't comply the former restrictions.

2) During initialization or boot of the operating system of a modern desktop computer, may the processor be found in user mode? Explain the answer. **0,75 points**

2

It can't.

During the system startup it is required to load the OS kernel in memory, this requires accessing disk or secondary storage and so executing I/O instructions that are privileged and then only executable in kernel mode.

When the switch on button is pressed a resident program in ROM starts execution, it has to allocate in memory another program that has to find the OS kernel code in the hard disk, allocate it in main memory and finally transfer the CPU control to it. Since then the kernel takes charge of drivers initialization and of creating the root process (init) from where all the other process will be created. Process creation is done in kernel mode.

3) The following code refers to the generated executable file named "Example1".

```
/*** Ejemplo1***/
#include "all_required_header_files.h"

main()
{ int i=0;
  pid_t pid, pid2;

  while (i<2)
  { pid=fork()
      switch(pid)
       {case (-1):{printf("Error creating child\n");
                   break;}
        case (0):{pid2=fork();
                  printf("Child %i created\n",i);
```

```
                        sleep(10);
                        exit(0);
                       }
           default: {printf("Parent\n");
                        sleep(5);}
        }
     i++;
   }
   exit(0);
}
```

Suppose that "Example1" runs successfully

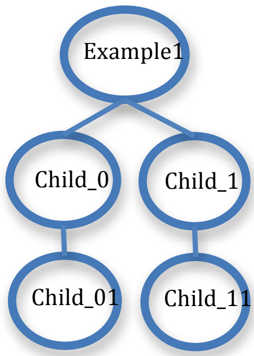
a) Indicate in a reasoned way, the number of processes that will create and draw the generated process tree.

b) Indicate and explain, if zombie processes and/or orphans can appear.

1.5 puntos

3

a) *5 processes are created in all when Example1 code is executed.*
Process Example1 creates two children: with pid=fork(), when i=0 creates Child_0 next executes "default" and it does i++, executed the loop code again with i=1 and creates Child_1.
Furthermore both Child_0 and Child_1 become parents because they execute code in "case (0)" where there is another fork() call. These new children (Child_01 and Child_11) and their respective parents (Child_0 and Child_1) execute exit(), and so finish, after sleep(10).



b) *In Example1 code, there no wait() call, so parent processes do not wait to their children and then orphan processes can appear, if parent processes end before their children.*
sleep() suspends a process by its will.
The initial process from Example1 is suspended along 5 seconds (sleep(5)) after creating Child_0 and another 5 seconds after creating Child_1, 10 seconds in all. Child_0 and e Child_1 are suspended along 10 seconds (sleep(10)), there is a high probability that Example1 end before Child_1. In this case Child_1 becomes orphan and it is adopted by "init".

It is very improbable that zombies will appear, because all the parent processes (Example1, Child_0 and Child_1) seem to finish before their children.

---

4) Indicate and explain if the following statements, about process states, are true or false:

a) Only the processes that are *running* can be passed to the *finished* state, when the execution of their instructions ends.

b) The *suspended* state can be reached only by processes that have requested or are performing a blocking I/O operation.

c) The state change from *running* to *ready* is only possible in systems with preemptive schedulers.

1.0 point

4	a) *FALSE* *From the Ready and Suspended states a process can go to Finished when it receives a signal. A process can finish in an unexpected way by means of SIGKILL, even in suspended state, and SIGKILL cannot be masked.*
	b) *FALSE* *Processes can go to Suspended state when they do blocking calls to wait() or sleep(). They also go to Suspended when they try to También pasan a SUSPENDIDO cuando solicitan un recurso SW que se encuentra ocupado (sem_wait, pthread_mutex_lock)*
	c) *TRUE* *With preemptive schedulers when one process is expelled from the CPU before ending its CPU burst, by another with higher priority, it is put in the Ready queue. In non-preemptive schedulers processes only leave the CPU to go to finished or suspended.*

5) A system has an multiqueue scheduler in the short term (STS) with three queues Queue0, Queue 1 and Queue 2, with scheduling algorithms **RR with q = 1, SRTF, and FCFS**, respectively.

Scheduling between **queues is managed with preemptive priorities** being the least priority and the higher priority the ones of Queue0 and Queue2, respectively. Each process has a counter of promotion (CountPro) that the system maintains to manage promotion/demotion between queues, so that processes can get into more priority queues. Whenever a process **goes to the suspended state** his CountPro is incremented by 1 (CountPro = CountPro+1). A process is placed in Queue0 if its CountPro is equal to 0, in Queue1 if its CountPro = 1 and in Queue2 if CountPro > = 2. When processes arrive to the system its CountPro is initialized (CountPro = 0) and they go to Queue0.

Suppose that I/O operations are carried out on the same FCFS scheduled device and that the following processes shown in the table arrive to the system:

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

a) Fill out the following table indicating at each instant in time where the processes are located.
b) Obtain the CPU utilization and the mean waiting time and mean turnaround time.

2.0 point (1.25+0.75)

5a	T	Cola 0 RR q=1	Cola 1 SRTF	Cola 2 FCFS	CPU	Cola E/S	E/S	Evento
	0	A			A			A arrives, ContPro(A)=0
	1	A B			B			B arrives, ContPro(B)=0
	2	CA			A		B	C arrives, ContPro(C)=0 ContPro(B)=1
	3	D C			C	A	B	D arrives
	4	CD	B		B		A	ContPro(A)=1
	5	CD			B		A	
	6	CD	B A		A			
	7	CD	B		B		A	ContPro(A)=2
	8	CD			B		A	
	9	CD			D	B	A	ContPro(B)=2

	10	D C			C	B	A	
	11	CD		A	A		B	
	12	CD			D		B	A ends
	13	C		B	B		D	ContPro(D)=1
	14	C	D		B			
	15	C			D			B ends
	16	C			C			D ends
	17				---		C	ContPro(C)=1
	18				C			
	19							C ends
	20							
	21							
	22							
	23							
	24							

5b

Mean waiting time = 1+1+12+9 /4 =23/4 =5.75 tu

Mean turnaround time = (12+14+17+13)/4 = 56/4 =14 tu

CPU utilization = 18/19 = 97.4%

6) Given the following code that has been generated the executable file named "Example2".

```
/*** Example2***/
#include <stdio.h>
#include <pthread.h>

void *fun_thread( void *ptr )
{ int sec;

  sec=(int)ptr;
  sleep(sec);
  printf("I waited %d seconds\n",sec);
}

int main()
{
  pthread_attr_t atrib;
  pthread_t thread1, thread2, thread3;

  pthread_attr_init( &atrib );
  pthread_create(&thread1, &atrib, fun_thread, (void *)30);
  pthread_create(&thread2, &atrib, fun_thread, (void *)1);
  pthread_create(&thread3, &atrib, fun_thread, (void *)10);
  pthread_join(thread3, NULL)
}
```

Write the strings that prints the program in the terminal after its execution. Explain your answer.

1.0 point

6	*I waited 1 seconds* *I waited 10 seconds* *The main thread only waits with pthread_join() to thread3. So the main thread will end immediately after thread3.* *When the main thread ends all the process resources are released then all running threads inside the process will finish, in whatever execution point they were.* *Given the time interval difference in the sleep calls that the threads do, thread2 finishes before thread3 and thread1 finishes after thread3, this means that message from thread2 is printed and message from thread1 will never be printed.*

---

7) The following C code corresponds to a process with two threads that share memory and have to synchronize using the **busy waiting mechanism**. In order to design the **input and output protocols** to the critical section, the shared variables **flag** and **turn** are declared.

JUMP THIS QUESTION

IT CORRESPONDS TO REMOVED CONTENT

```
//***** Example 3 **//
#include <pthread.h>
…
int flag[2];
int turn;

void* thread(void* id)
{ int i;
  i = (int)id;

  while ( 1 ) {

    remaining_section();

    /** b)Assign values to flag[i] and turn**/
    while(/** c)Input protocol condition **/);

    critical_section();

    flag[i] = 0;
  }
}
int main() {
  pthread_t th0, th1;
  pthread_attr_t atrib;
  pthread_attr_init(&atrib);
  /** a)Initialization of flag variable **/
  pthread_create(&th0, &atrib, thread, (void *)0);
  pthread_create(&th1, &atrib, thread, (void *)1);
  pthread_join(th0, NULL);
  pthread_join(th1, NULL);
}
```

Answer to the following items:

a) Replace "`/**a) Initialization of flag variable **/`" by its C code.

b) Replace "`/**b) Assign values to flag[i] and turn **/`" by its C code.

c) Replace "`/**c) Input protocol condition **/`" by its C code.

1.0 point

7	**a)** `flag[0]=0;  flag[1]=0;`

	b)
	
	`flag[i] = 1;`
	`turn =(i+1)%2;`

	c) `while (flag[(i+1)%2]==1) && (turn==(i+1)%2);`

Indicate all the possible values that can reach variable x after the concurrent execution of the processes A, B and C. Explain your answer indicating for each reached value of x the execution order of every section.

// Shared variables
int x=1;
Semaphores S1=3, S2=0, S3=0;		

// Process A	//Process B	//Process C

P(S1)
P(S3)
x = 2*x + 1; // section 1
V(S2)
	P(S1)
	P(S2)
	x = x*3; // section 2
	V(S1)
		P(S1)
		x = x + 2; // section 3
		V(S3)
		P(S1)
		x = x + 3; // section 4

1.0 point

8

There are at least four possible execution patterns:

x=1
C P(S1), section3, V(S3) -> A P(S1)P(S3) section1, V(S2) -> B P(S1), P(S2) section2 , V(S1) -> C P(S1) section4 → value of x=24
*x=3 → x= 3*2 +1 =7 → x= x*3= 21 → x= x+3=24*

x=1
C P(S1),section3,V(S3), P(S1), section4 → A P(S1), P(S3), section1 → Process B suspended in semaphore S1
x=3 → x=6 → x=13

x=1
C P(S1), section3;V(S3)→ A P(S1), P(S3), section1, V(S2)→ C P(S1)section4 → Process B suspended in semaphore S1
x=3 → x=7 → x=10

x=1
C P(S1), sección3,V(S3) → B P(S1) → C P(S1), sección4→ A P(S1)A se Suspende en S1→ B P(S2), B suspendido en el semáforo S2
x=3 → x=6

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática de Sistemas y Computadoras (DISCA)

fSO

Ejercicio de Evaluación

05 de Noviembre de 2012

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

9) In 'Example4' complete the code of functions **fth_ONE** and **fth_TWO**, with the operations on **semaphores** needed to display on the screen the first ten unsigned integers sorted (0 1 2 3 4 5 6 7 8 9).

```
/*** Example4***/
#include <stdio.h>
#include <pthread.h>

void *fth_ONE( void *ptr )
{ int i;
  for (i=1; i<10; i+=2)
  {
    printf("%d ",i);

  }
}
```

```
void *fth_TWO( void *ptr )
{ int i;
for (i=0; i<10; i+=2)
  {
    printf("%d ",i);

  }
}
```

```
int main()
{ pthread_attr_t atrib;
  pthread_t th1, th2;

  pthread_attr_init( &atrib );
  pthread_create(&th1,&atrib,fth_ONE, NULL);
  pthread_create(&th2,&atrib,fth_TWO, NULL);
  pthread_join( th1, NULL);
  pthread_join( th2, NULL);
  printf ("\n");
}
```

Thread 'th1' should display the odd numbers and "th2" the even numbers. To synchronize the threads execution use as many semaphores as needed and specify their initial values. Use the P and V names for semaphore operations.

1.0 puntos

9

Semáforo: sinc, smutex;
sinc=0; smutex=1;

```
void *fth_ONE( void *ptr )
{
  int i;
  for (i=1; i<10; i+=2) {
    P(sinc);
    printf("%d ",i);
    V(smutex);
  }
}
```

```
void *fth_TWO( void *ptr )
{
  int i;
  for (i=0; i<10; i+=2) {
    P(smutex);
    printf("%d ",i);
    V(sinc);
  }
}
```