UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Departamento de Informática de Sistemas y Computadoras (DISCA)

fSO

EEE1

November, 4th 2013

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

SURNAME		**NAME**		Group
IDN		**Signature**		E

- **Keep the exam sheets stapled.**
- **Write your answer inside the reserved space.**
- **Use clear and understandable writting. Answer in a brief and precise way.**
- **The exam has 9 questions, every one has its score specified.**

---

1. Suppose that you are working in a compay where you have to assist customers that do queries by phone and the reply should be on line, as quick as possible. To do the job you have to perform actions and queries on a computer system. What kind of OS should be installed on the computer: batch monoprogrammed, multiprogrammed or time sharing, explain your answer.

(0,75 points)

1

The needs of this job involves several applications running concurrently and responses in a short time.
* Batch systems will queue requests for execution and they do not have user-machine interaction, since these systems are managed by a specialized operator. The response time in a batch system depends on queued jobs and therefore would not be a good choice.
* Pure multiprogrammed systems allow multiple concurrent applications, but they not include user machine interaction.
* Time sharing systems are able to run multiple applications at the same time and delivering response from the machine in a short enough time because CPU time-slice scheduling, and so they also support terminals for interactive users.
So a time-sharing system must be used

---

2. Answer the following questions in relation to an OS:
a) What are the differences between a shell command and a system call? Explain your answer on both the concept and the use.
b) Enumarate the sequence of actions that an OS takes when a user application performs a system call, indicate which are done in kernel mode.

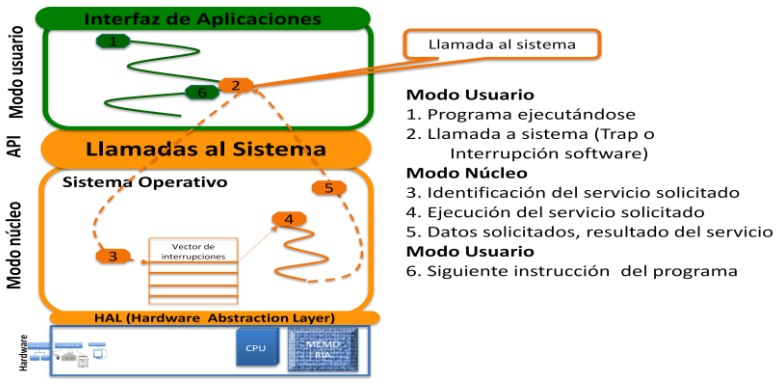
(0,75 points)

2

a)

System calls are the mechanism through which applications access the operating system services and therefore all the machine resources. Shell commands are user programs that use system calls to ask for OS services. Shell commands are commonly invoked from terminal or a shell script, while system calls are included in the applications code, libraries code and of course shell commands and the shell itself.

b)



3. Given the following code in C and POSIX corresponding to a process named Proc:

```
 1 #include …. // as required
 2 #define N 3
 3
 4 main() {
 5   int i = 0;
 6   pid_t pid_a;
 7
 8   while (i<N)
 9   { pid_a = fork();
10     switch (pid_a)
11     { case -1:
12         printf("Error creating child...\n");
13         break;
14       case 0:
15         printf("Message 1: i = %d \n", i);
16         if (i < N-1) break;
17         else exit(0);
18       default:
19         printf("Message 2: i = %d \n", i);
20         while (wait(NULL)!=-1);
21      }
22     i++;
23   }
24   printf("Message 3: i=%d\n",i);
25   exit(0);
26 }
27
```

a) Draw the process tree generated when it is running and indicate for every process at what value of "i" it has been created.

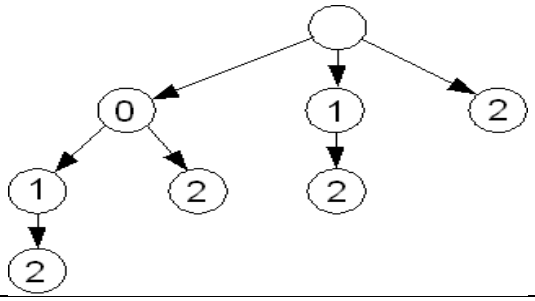
b) Explain if there is a possibility of appearing orphan and/or zombie children.

(1,0 point)

3

a)

a) A total of 8 processes are created with the structure tree that is shown in the figure, the numbers indicate the value of the variable i in the instant of its creation.



b)

There is no possibility of zombies or orphans when you run this code since the processes that act as parents always wait for children processes with the sentence:

```
while (wait(NULL) != -1));
```

Therefore waiting for the completion of all children guarantees that there are no orphaned, and doing the wait() call immediately after creating child processes ensures that there are no zombies.

4. Given the following code in C and POSIX corresponding to a process named Test, that runs successfully:

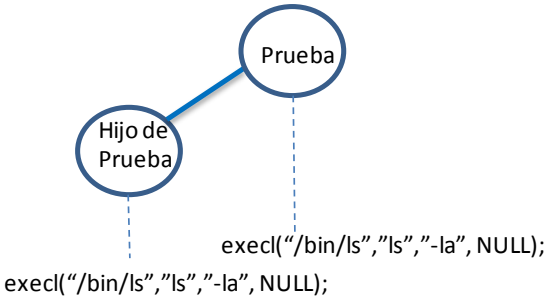
```
 1 #include …. // as required
 2
 3 int main() {
 4   pid_t val;
 5   printf("Menssage 1: before exec()\n");
 6   fork();
 7   execl("/bin/ls","ls","-la", NULL);
 8   val = fork();
 9   if (val==0) {
10     execl("/bin/ps","ps","-la", NULL);
11     printf("Menssage 2: after exec()\n");
12     exit(1)
13   }
14   printf("Message 3: before exit()\n);
15   exit(0);
16 }
```

a) Explain the number of processes that are creating when executing Test and the parent/child relationship.
b) Indicate and explain the messages and information shown on the screen when executing Test.

(1,0 point)

4

a) If this code runs successfully it creates two processes, the main process and a child process (created in line 6 with fork()). Both processes parent and child execute line 7 and therefore they both change its memory image, changing the initial code by the "ls" command code with parameter "-la", so both run the command "ls - la".



b) The following messages appear in the screen:
"Message 1: before exec()"
The output of "ls –la" command execution
The output of "ls –la" command execution

5. Given the following code corresponding to process Example:

```
 1 #include …. // as required
 2
 3 int main(void) {
 4   int val;
 5   printf("Message 1\n");
 6   val=fork();
 7   /** Write here your changes **/
 8   sleep(5);
 9   printf("Message 2\n");
10   return 0;
11 }
```

a) Indicate what changes are required in the previous code in order to make the child process orphan and to be adopted by `init`. (**Note**: Use sleep() and C senteces).
b) Indicate what chantes are required in the previous code in order to make the child process zombie for a while. (**Note**: Use sleep() and C sentences).
c) Explain in what instructions in the proposed code we can be sure that a CPU context switch will happen and in what intructions the will be an state change in the Example process.

(1,0 points)

5

a)

A process becomes orphan when his parent finishes before it. To ensure the completion of the parent process before his child, the code in line 6 could be similar to the following:
Option 1: if (val>0) exit(0);
Option 2: if (val== 0) sleep(30); /* sleep()for much longer than the parent process */

b)

A process becomes zombie when it ends before its parent calls to wait() for this child. To ensure the completion of the child process rather than its parent, the line 6 code could be similar to the following:
Option 1: if (val==0) exit(0);
Option 2: if (val> 0) sleep(30); /* sleep() for much longer than the child process */

c)

In line 8 when after calling to sleep (5) the OS will act, particularly its short-term scheduler that suspends the process, so it leaves the CPU, and the CPU is allocated to a process selected from the ready queue (context switch).
Processes are commonly suspended when they request access to I/O devices. Lines 5 and 9 could be generated context changes, although, given the type of instruction in question (display on screen) is likely that the process request is addressed without suspension.

---

6. Given the following code that tries to solve the race condition problem, and assuming that there are more than one thread executing concurrently the code of the "add" function, indicate if the following sentences are true or false (**Note**: An error voids a correct answer).

```
1   void *add (void *argument) {
2     long int count;
3     long int aux;
4     while(test_and_set(&key)) ;
5     for (count = 0; count < REPETITIONS; count = count + 1) {
6       V = V + 1;
7     }
8     key =0;
9     printf("-------> End Add (V = %ld)\n", V);
10    pthread_exit(0);
11  }
```

(1,0 points)

6

T/F	
V	Line 4 implements the input protocol to the critical section.
F	The proposed solution doesn't guarantee that it is race condition free
V	The proposed solution guarantees that the code is race condition free, because the access to the global variable V in done sequentially
V	If we interchange line 4 - 5, and 7 - 8, the resulting code is race condition free
V	Function call `test_and_set(&key)` tests and sets the value of key in an atomic way

7. Given the following code that has generated an executable file named "Threads1".

```
 1 #include … // as required
 2 #define TwoSeconds 2000000
 3 pthread_t H1,H2;
 4 pthread_attr_t atr;
 5
 6 void *WorkB (void *P)
 7 { char * text =(char *) P;
 8   usleep(TwoSeconds);
 9   printf("%s\n",text);
10 }
11
12 void *WorkA (void *T)
13 { printf("Text:\n");
14   pthread_create(&H2, &atr, WorkB, T);
15   usleep(TwoSeconds);
16   pthread_join(H2, NULL);
17 }
18
19 int main() {
20   pthread_attr_init(&atr);
21   pthread_create(&H1, &atr, WorkA, "FSO exam");
22   usleep(TwoSeconds);
23   pthread_join(H1, NULL);
24   pthread_join(H2, NULL); /* Thread created inside WorkA*/
25   return(0);
   }
```

a) Indicate and explain the strings that are printed in the terminal after Threads1 execution.
b) Indicate and explain the approximate time that Threads1 will take for execution.
c) Explain if in main() it will be correct not to wait for thread H2 that is the one created inside WorkA.
d) During the execution of Thread1, how many active threads are there when "Text:" is written in the terminal, and how many when it is written "FSO exam"? Explain your answer.

(1,0 points)

7

a)
The program prints two lines: first-line "text:", and then (after waiting two seconds) "FSO exam"

b)
Execution time is about two seconds. Three threads do a bit of work (system call to write a text or create a thread) and then are suspended for two seconds. It is quite likely that the threads are suspended at the same time, overlapping their suspension times

c)
Unlike the OS processes, threads have no parent-child relationship. The main thread (the one created from the main function) can wait for completion of a thread that it has not directly created. Anyway, the call to "pthread_join" in line 24 will have no effect, because H2 has been waited for by H1 with a blocking call to "pthread_join", so H2 is already completed when the main thread executes line 24.

d)
When "Text:" is written there are two active threads, one of them is the thread that executes the main function main and the other one is H1. When "Message of FSO" is written there are three active threads the thread that executes the function main, H1 and H2.

8. In a computer system there are three process classes: OS processes (QOS), user processes (QU) and background processes (QB). The short term scheduler in the system has three queues, one for every class of processes. The queue OS uses preemtive priorities (PP), the queue U uses STRF and the queue QB uses RR with q=1. **The inter queue scheduling is preemtive pritorities**, being QOS the one with higher priority and QB the one with lower priority. I/O operations are performed on the same device with FCFS scheduling.
Two processes of every class arrive to the system at the instants specified in the following table with the corresponding execution profiles:

Process	Execution profile	Arrival instant	Process class
A (-priority)	2 CPU	2	Sistem (QOS)
B (+priority)	3 CPU	3	Sistem (QOS)
C	1 CPU + 1 E/S + 1 CPU	6	User (QU)
D	2 CPU + 1 E/S + 1 CPU	17	User (QU)
E	4 CPU + 3 E/S + 2 CPU	0	Background (QB)
F	2 CPU + 1 E/S + 2 CPU	5	Background (QB)

a) Fill the following table with the execution time line.
b) Obtain the mean waiting time, the mean turnaround time and the CPU utilization.

2.0 points (1.25+0.75)

8a

T	QOS PP	QU SRTF	QB RR q=1	CPU	I/O Queue	I/O	Comments
0				$E^1(3)$			E arrives
1				$E^1(2)$			
2			$E^1(1)$	$A^1(1)$			A arrives
3	$A^1(0)$		$E^1(1)$	$B^1(2)$			B arrives
4	$A^1(0)$		$E^1(1)$	$B^1(1)$			
5	$A^1(0)$		$F^1(1)$ $E^1(1)$	$B^1(0)$			F arrives
6		$C^1(0)$	$F^1(1)$ $E^1(1)$	$A^1(0)$			C arrives **B ends**
7			$F^1(1)$ $E^1(1)$	$C^1(0)$			**A ends**
8			$F^1(1)$	$E^1(1)$		$C_1(0)$	
9			$E^1(0)$ $F^1(1)$	$C^2(0)$			
10			$E^1(0)$	$F^1(1)$			**C ends**
11			$F^1(0)$	$E^1(0)$			
12				$F^1(0)$		$E_1(2)$	
13					$F_1(0)$	$E_1(1)$	
14					$F_1(0)$	$E_1(0)$	
15				$E^2(1)$		$F_1(0)$	
16			$E^2(0)$	$F^2(1)$			
17			$F^2(0)$ $E^2(0)$	$D^1(1)$			D arrives
18			$F^2(0)$ $E^2(0)$	$D^1(0)$			
19			$F^2(0)$	$E^2(0)$		$D_1(0)$	
20			$F^2(0)$	$D^2(0)$			**E ends**
21				$F^2(0)$			**D ends**
22							**F ends**
23							
24							

8b

Mean waiting time = (3 + 0 + 1 + 0 + 11 + 10) / 6 = 4.17 tu

Mean turnaround time = (5 + 3 + 4 + 4 + 20 + 17) / 6 = 8.83 tu

CPU utilization = 20 / 22 = 0.91 = 91%

9. Given the following program, where there are three semaphores, with the threads "add" and "sub". Indicate and explain for every one of the values of "x" and "y" proposed, if it could happen or not a race condition, the state at what the threads will end and the final value of V that is written at line 12:

```
#include … // as required
int V = 100;
sem_t sem, add, sub;
```

```
void *Add (void *argument) {
  int count;
  for (count=0; count<100; count++) {
    sem_wait(&add);
    sem_wait(&sem);
    V = V + 1;
    sem_post(&sem);
    sem_post(&sub);
  }
}
```

```
void *Sub (void *argument) {
  int count;
  for (count=0; count<100; count++) {
    sem_wait(&sub);
    sem_wait(&sem);
    V = V - 1;
    sem_post(&sem);
  }
}
```

```
1.   int main (void) {
2.     pthread_t threadAdd, threadSub, threadInspec;
3.     pthread_attr_t attr;
4.     int x, y;
5.     pthread_attr_init(&attr);
6.     sem_init(&sem,0,1);
7.     sem_init(&add,0,x);
8.     sem_init(&sub,0,y);
9.     pthread_create(&threadAdd, &attr, Add, NULL);
10.    pthread_create(&threadSub, &attr, Sub, NULL);
11.    usleep(80000000); // enough to execute Add and Sub
12.    fprintf(stderr, "-------> FINAL VALUE: V = %d\n", V);
13.    exit(0);
14.  }
```

a) Suposse x = 5 and y = 1.
b) Suposse x = 500 and y = 1.
c) Suposse x = 20 and y = 5. (1.5 points)

9

a) The *add* semaphore is initialized to 5 and the *sub* semaphore is initialized to 1. In this code there are only *add* decrement operations (Pop(add)) and no increment operations are performed on *add* (V(add)). A P(add) operation is done before incrementing variable *V* (*V* = *V* + 1), so the number of times that you can increment *V* will be the initial value of the *add* semaphore that is 5. A increment operation is done on semaphore *sub* after increasing *V* (V (sub)).
Function *Sub* makes an operation P(sub) before decrementing *V* (*V* = *V*-1), therefore it will perform as many subtraction as addition done before plus the initial value of the *sub* semaphore which is 1, so *V* is decremented by 6. Then the final value of *V* is 99.
Furthermore the Add thread will remain suspended in the *add* semaphore and the Sub thread will be suspended at semaphore *sub*.
Semaphore *sem* guarantees mutual exclusion so there is no possible race condition on *V* and the result will always be the same.

b) The *add* semaphore is initialized to 500 and the *sub* semaphore is initialized to 1. Both for loop in Add and Sub funtions are limited to 100 turns, therefore now threads will not be blocked, both of them will do 100 operations, so the final value of *V* will be 100.

c) The *add* semaphore is initialized to 20 and the *res* semaphore is initialized to 5. It is the same case as in section a). A number of subtractions that exceeds 5 the number of sums will be done , so the final value of *V* will be 95.