UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática de Sistemas y Computadoras (DISCA)

fSO

EEE1: Ejercicio de Evaluación

November, 3rd 2014

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 writing. Answer in a brief and precise way.**
- **The exam has 9 questions, everyone has its score specified.**

---

1. Considering the concept of system calls and the Unix shell that works with internal and external commands, answer to the following sections as brief and precise as possible:
   a) Describe what are internal and external command in the Unix shell, and how they are implemented.
   b) Explain what a system call is, how they are used by applications and if they are necessary to implement internal and external shell commands.
   c) Suppose you are working with the Unix Shell (bash), indicate what are the name of the file/s that contain executable code for every one of the following commands:

```
cd /usr/bin
ls –la | more
./MyCopy  fic1  fic2
```

NOTE: **cd** is an internal command; **`cp, ls, more`** are external commands; and ***`MyCopy`*** is an scripts with execution permissions that contains one line “`cp $1 $2`”.

(0.4+0.3+0.3=1.0 points)

1

a)
The Shell deals with to command types:

- **Internal: Their code are inside the self shell**
- **External: The command is performed by another process created by the Shell, that process relies on another executable file with binary code or Shell script.**

b)
System calls are the mechanism to ask for OS services that allow access to protected system resources (i.e. hardware devices). These services are offered as library functions.
The Shell is a system utility that provides a user interface base don text commands, so all commands use one or more system call, independently if commands are internal or external

c)

- **“cd /usr/bin”: It is an internal command, its code is inside the “bash” executable file**
- **“ls –la | more” : “ls” and “more” are executable files usually located at “/bin”**
- **“./MyCopy fic1 fic2”: “MyCopy” file is located at the working directory, it is a Shell script that is executed by the Shell (“bash”) executing the executable file “/bin/cp”**

---

2. Given the following C code, that produces the executable file named “prog” and considering that **COND** can be defined as “= =” or “>”, answer the following items:

```
/***** Source code for prog.c ***/
#include <all required headers>

int main() {
  pid_t pid;
  int i;

  for (i=0; i<3; i++){
    pid = fork();
    if (pid COND 0){
      printf("PID_IF = %d, PPID_IF = %d \n", getpid(), getppid());
      sleep(5);
      break;
    }
    printf("PID_FOR = %d, PPID_FOR = %d \n", getpid(), getppid());
  }
  sleep(5);
  return(0);
}
```

a) Suppose that “prog” is executed in such a way that its parent PID is 4000, and its own PID is 4001. Along its execution the system assigns to the processes that are created the sequence of PIDs: 4002, 4003, 4004, etc. Fill the following tables with the values that are shown in the screen with the strings PID_IF, PPID_IF, PID_FOR and PPID_FOR, considering that **COND** is defined as:

a1) `#define COND ==`

a2) `#define COND >`

b) Considering “`#define COND ==`” explain if there can be zombie or orphan processes, and if so how many of each type can appear.

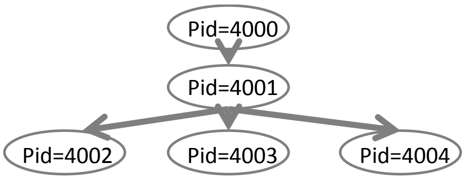
(0.4+0.4+0.4=1,2 points)

2

a1) `#define COND ==`

It is generated a process fan

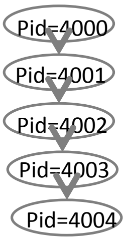
PID_IF	PPID_IF	PID_FOR	PPID_FOR
4002	4001	4001	4000
4003	4001	4001	4000
4004	4001	4001	4000



a2) `#define COND >`

It is generated a process chain

PID_IF	PPID_IF	PID_FOR	PPID_FOR
4001	4000	4002	4001
4002	4001	4003	4002
4003	4002	4004	4003



b) Zombie and orphan processes with “`#define` **`COND`** `==`”

With this condition a process fan is generated, so the sentences inside “if (pid==0)” are only executed by the children that go into suspension after 5 seconds and then they go out of the loop to enter again into suspension for another 5 seconds, 10 seconds in all.

The initial process with PID 4001 executes all the for loop iterations, and before ending goes into suspension for 5 seconds. Furthermore the parent doesn’t call to wait so it will end before its children that become **3 orphan processes**.

There will be **no zombie processes** because the children execution time is greater that the parent.

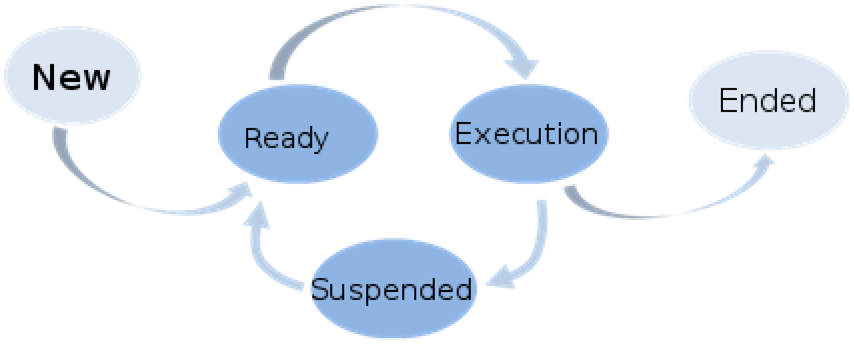
3. Indicate for each of the following items if it is a component of the operating system or an operating system utility

NOTE. Two errors void a correct answer.

(0.7 points)

3	**OS component**	**OS utility**	**Element**
	X		Process scheduler
		X	Command Shell
		X	Text editor
		X	Compiler
	X		Virtual memory manager
		X	System monitorization tools
	X		I/O device driver

4. A given operating system has the following diagram of possible states and transitions for processes:



Explain if this operating system could be working with a preemptive scheduler.

(0.5 points)

4

A preemptive scheduler allows a process to be expelled from the CPU before ending its CPU burst, to permit another more priority process entering the CPU.

The state transition diagram in the figure cannot correspond to a preemptive scheduler because the transition from execution to ready is missing.

5. Consider two executable files named “compare” and “same” located in the current working directory and whose source codes are shown below:

```
// Program compare.c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char * argv[]){
  int pid;int status;
  if (argc != 3){
    printf("compare: wrong arguments\n");
    exit(-1);
  }
  pid=fork();
  if (pid==0){
    execl("./same","same",argv[1],argv[2],NULL);
    exit(-2);
  }
  wait(&status);
  if(status==0){
    printf("Text A\n");
    exit(0);
  }
  printf("Text B\n");
  exit(0);
}
```

```
// Program same.c

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char * argv[]){
  int a,b;
  if (argc != 3){
    printf("same: wrong \n");
    exit(255);
  }
  a=atoi(argv[1]);
  b=atoi(argv[2]);

  if (a==b) exit(0);

  exit(1);
}
```

The *atoi* function returns a numerical value of type “*int*” from a string (*char **), so *atoi("215")* returns the integer value 215.

Indicate:

- the total number of created processes, and
- the text lines shown in the terminal after every one of the following executions:

a) `./compare 3 3`
b) `./compare 6 3`

(0.5+0.5=1.0 point)

5

a) ./compare 3 3

```
$ ./compare 3 3
Text A
```

2 processes are created.
The first process with code from “./compare” and inside this code it creates a child calling to fork().
The child process performs the call:

```
    execl("./same","same",argv[1],argv[2],NULL);
```

so its code becomes “./same”
The child termination code is 0 (exit(0)) the parent waits after performing “wait(&status)” and prints “Text A”.

b) ./compare 3 6

```
$ ./compare 6 3
Text B
```

It happens the same as in a) but now (a == b) is false so the termination code in the child is 1 (exit(1)) and the parent prints “Text B” after waiting in “wait(status)”.

6. A CPU scheduler has 3 queues (S, U1, U2), managed with an inter queue policy of preemptive priorities, being S the one with the highest priority and U2 the one with the lowest priority. Queue S receives only server processes that remain in this queue until ending, but user processes are assigned to queues U1 and U2. When a user process enters the system it is located on queue U2 and it remains in this queue until its first I/O operation is completed, after that it promotes to queue U1 where it will remain until ending its execution. The system has a single I/O device managed with a FCFS policy. Every CPU queue is managed with the following policies:

S:FCFS **U1: Round Robin with q=1** **U2: Round Robin with q=2**

Five processes arrive to the system with the following features:

Process type	Process	Execution profile	Arrival instant
Server	Sa	1 CPU + 4 I/O + 1 CPU+1 I/O + 1 CPU	5
Server	Sb	1 CPU + 2 I/O + 1 CPU	7
User	Uc	4 CPU + 2 I/O + 2 CPU	0
User	Ud	4 CPU + 2 I/O + 3 CPU	1
User	Ue	3 CPU + 1 I/O + 1 CPU	3

a) Obtain the time line of CPU, I/O and queues process allocation (1.3 points)

T	FCFS Queue S	RR (1) Queue U1	RR (2) Queue U2	CPU	Queue I/O	I/O	Comments
0				Uc(3)			Uc arrival
1			Ud(4)	Uc(2)			Ud arrival
2			Uc(2)	Ud(3)			
3			Ue(3) Uc(2)	Ud(2)			Ue arrival
4			Ud(2) Ue(3)	Uc(1)			
5			Uc(1) Ud(2) Ue(3)	Sa(0)			Sa arrival
6			Uc(1) Ud(2)	Ue(2)		Sa	
7			Ue(2) Uc(1) Ud(2)	Sb(0)		Sa	Sb arrival
8			Ue(2) Uc(1)	Ud(1)	Sb	Sa	
9			Ue(2) Uc(1)	Ud(0)	Sb	Sa	
10			Ue(2) Uc(1)	Sa(0)	Sa Ud	Sb	
11			Ue(2)	Uc(0)	Sa Ud	Sb	
12			Ue(2)	Sb(0)	Uc Sa	Ud	
13				Ue(1)	Uc Sa	Ud	Sb ends
14			Ue(1)	Ud(2)	Uc	Sa	
15		Ud(2)	Ue(1)	Sa(0)		Uc	
16			Ue(1)	Ud(1)		Uc	Sa ends
17		Ud(1)	Ue(1)	Uc(1)			
18		Uc(1)	Ue(1)	Ud(0)			
19			Ue(1)	Uc(0)			Ud ends
20				Ue(0)			Uc ends
21						Ue	
22				Ue(0)			
23							Ue ends

b) Compute the mean waiting time, the mean turnaround time and the CPU utilization.

(0.75 points)

6	**b)** Mean waiting time = (0+0+9+7+15) / 5=31 / 5 = 6,2 ut Mean turnaround time = ((16-5)+(13-7)+(20-0)+(19-1)+(23-3)) / 5=(11+6+20+18+20)/ 5 = 75 / 5 = 15 ut CPU utilization = 22 / 23 = 0, 956 = 95,6 %

7. Given the following code, use semaphores with the notation proposed by Dijkstra (P and V operations) to ensure that following running threads execute functions whose name begins with "sequence-" according to the order that suggest the numbers employed in such names. If there are multiple functions with the same name (including the number at the end), none should start before the end of the functions with the previous name that must be completed (all of them) before the start of any function with the following name. Furthermore, calls to function "sc()" must be done in mutual exclusion. Declare and initialize the semaphores required to accomplish the described behavior.

1.0 point

7	*Declare and initialize the required semaphores* *mutex=1, S2 =0, S3=0, S4=0*		
	THREAD 1	**THREAD 2**	**THREAD 3**

	`P(mutex)`
	
	`sc();`
	`V(mutex)`
	
	`sequence1();`
	`V(s2);`
	`P(mutex);`
	`sc();`
	`V(mutex);`
	
	`P(s3);`
	`sequence3();`
	`V(S4);`
		`P(mutex)`
		`sc();`
		`V(mutex)`
		
		`sequence1();`
		`V(S2);`
		
		`P(mutex);`
		`sc();`
		`V(mutex);`
		
		`P(S3);`
		`sequence3();`
		`V(S4);`
			
			`P(mutex)`
			`sc();`
			`V(mutex)`
			
			`P(s2); P(S2);`
			`sequence2();`
			`V(S3); v(S3);`
			
			`P(S4); P(S4);`
			`sequence4();`

8. The following program, whose executable code has been generated with name "threads", processes the strings passed as arguments, showing the result on the standard output.

```
/** Program threads.c **/
#include <all required headers>
#define MAX_THREADS 100
pthread_t thread_p[MAX_THREADS];
pthread_t thread_m[MAX_THREADS];

pthread_attr_t atr;

void *Print(void* ptr){
  char *str= (char*) ptr;
  int longx;
  longx= strlen(str); //String length
  sleep(longx);
  printf("%s\n",str);
}

void *UpperC(void* ptr){
  char *str= (char*) ptr;
  int longx;
  int i;
  longx= strlen(str); //String length
  for(i=0;i<longx;i++)
    str[i]= str[i]-32; //Uppercase conv.
  sleep(2); }
```

```

int main(int argc, char *argv[]){
  int i, h=0;

  pthread_attr_init(&atr);

  for (i=1; i<argc; i++){
    pthread_create(&thread_p[i],&atr,Print,argv[i]);
  }
  for (i=1; i<argc; i++){
    pthread_create(&thread_m[i],&atr,UpperC,argv[i]);
    pthread_join(thread_m[i], NULL);
  }
  pthread_exit(NULL);
}
```

Considering that the execution of "threads" is done with the following command line:

```
$./threads dont unstaple the sheets
```

Answer the following:

a) What is the maximum number of threads that will run concurrently?

b) Why is it necessary to call *pthread_exit(NULL)* at the end of `main` ? What can happen if it is not called?

c) Suppose that the time consumption of "threads" is determined exclusively by the waiting introduced by sleep() calls, that is, consider that the rest of the instructions do not consume a significant amount of time. Considering the command below, indicate at what time instants printing is completed and at what time instants is completed the conversion to uppercase of each one of the 4 strings that are passed as arguments, naming them as "arg1" to "arg4". Suppose that "threads" execution begins at t = 0.

(0.4+0.4+0.5=1.3 points)

8	a) 4(Print) + 4(UpperC) + 1 (main) threads are created The first 4 threads correspond to each of the program calling parameters ("dont" "unstaple" "the" "sheets") the execute function Print, these threads a executed concurrently with the main thread. Later a thread is created for every parameter, that execute function UpperC but every one of these threads (4 in all) waits to the former one calling to pthread_join before starting the next one, so these 4 threads are executed sequentially. So there are up to 4 + 1 + 1 = 6 threads running concurrently.
	b) pthread_exit call guaranties that if the main thread ends before the other threads, they will not be suddenly ended so they will be able to end their execution. If pthread_exit() is not called some of the string may not be printed.

	c) **$./threads dont unstaple the sheets** // Execution starts at t = 0
	All the printing threads start at t = 0, with a printing delay equal to the string length, so the printing behaves as following:
	
	T = 3 arg1 (the)
	T = 4 arg3 (dont)
	T = 6 arg4 (sheets)
	T = 8 arg2 (unstaple)
	
	The uppercase conversion is done sequentially in such a way that the every UpperC thread waits 2 seconds after ending, and then the next UpperC thread is created, so we got the following conversion times:
	
	T = 0 arg1 (dont)
	T = 2 arg2 (unstaple)
	T = 4 arg3 (the)
	T = 6 arg4 (sheets)

9. Answer as much briefly and precisely as possible to the following questions:

(1.25 points)

9

a) How do you define a concurrent program?

It is a single program that, when running its code, several concurrent activities are created that evolve independently. These activities cooperate to solve a problem synchronizing and communicating between them.

b) What is a race condition?

A race condition is an undesirable situation that makes a sequential execution become incorrect when done concurrently. It happens when the concurrent execution of several activities over a shared variable leaves the variable in an inconsistent state, depending on the relative execution speed of the involved activities.

c) What is a critical section?

It is an area of code where several threads or processes perform reading and writing operation on shared variables. So they are code zone where execution must be synchronized.

d) With what requirements must comply a protocol that protects critical sections?

The protocol must comply with three conditions: mutual exclusion, progress and limited waiting.

Mutual exclusion: If one activity is inside the critical section it must be the only one.
Progress: If any activity is inside the critical section and other activities want to enter it then the decision about which process will enter only relies on the processes that are willing to enter.
Limited waiting: After one activity has ask to enter the critical section there is a limit in the number of times that other activities will enter the critical sections, so there a finite waiting bound.

e) What is busy waiting synchronization?

It a mechanism on which some solutions to race condition rely, it consists of avoiding one activity entering the critical section when it is busy by forcing the execution of an **empty loop that consumes CPU time.**

An example is **test-and-set()** instruction, that only lacks guarantying limited waiting.