UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática de Sistemas y Computadores (DISCA)

fSO

FINAL exam

Januray 24th, 2019

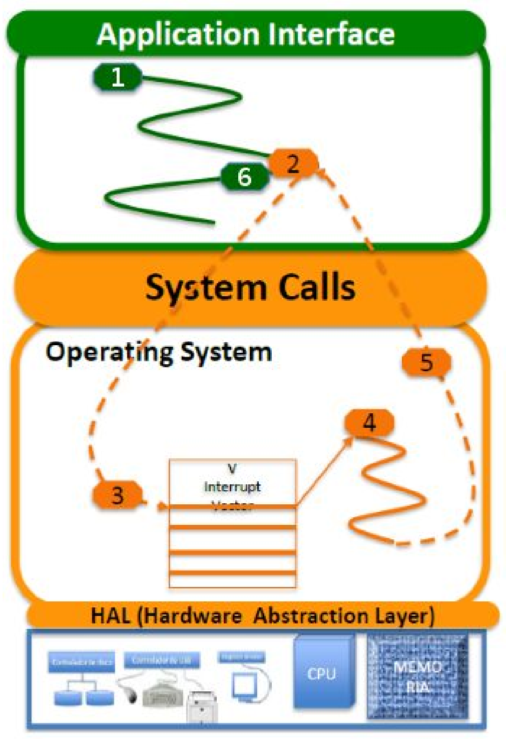
etsinf Escola Tècnica Superior d'Enginyeria Informàtica

| | | | | |
|---|---|---|---|---|
| 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.**
- **Remember that you have to appropriately explain your answers to get the full corresponding score**

**1.** Describe the events numbered from 3 to 6 in the figure, related to the request of a system call by a user process, and indicate in "Mode:" the CPU operating mode that corresponds to the execution of each one of the numbered events .

**(0,9 points)**

| | | |
|---|---|---|
| 1 |  Application Interface<br>1<br>2<br>6<br>System Calls<br>Operating System<br>5<br>4<br>3<br>Interrupt Vector<br>HAL (Hardware Abstraction Layer)<br>CPU<br>MEMORIA | **EVENT 1**<br>**Description:** User program in execution<br><br>**Mode:** User<br><br>**EVENT 2**<br>**Description:** System call<br><br>**Mode:** User<br><br>**EVENT 3**<br>**Description:** Identify system call<br><br>**Mode:** Kernel<br><br>**EVENT 4**<br>**Description:** Execute system call service routine<br><br>**Mode:** Kernel<br><br>**EVENT 5**<br>**Description:** Transfering results<br><br>**Mode:** Kernel<br><br>**EVENT 6**<br>**Description:** Execute next instruction<br><br>**Mode:** User |

**2.** Given the following code whose executable file is *example1*:

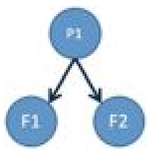
```
/*** example1***/
#include "head required.h"

int main() {
  int i=0, delay1=1, delay2=1;
  pid_t pid, pid_x;

  for (i=0; i<2; i++) {
    pid=fork()
    if (pid==0) {
      sleep(delay1);
      if(execlp("wc","wc","-l","example1.c",NULL)<0) {
        printf("Error running wc\n");
        exit(0); }
      pid_x=fork();
    }
  }
  sleep(delay2);
  while (wait(NULL) =! -1);
  exit(0);
}
```

Supposing that example1 runs without errors, answer to the following: **(1,0 points = 0,5 + 0,5)**

**2**

**a)** Number of processes that are generated when executing example1 and draw the processes tree .

The number of processes created when executing example1 is three, one parent and two children. Both children will execute the execlp ("wc", "wc", "- l", "example1.c", NULL) <0. The scheme would be the following:



**b)** Indicate with the corresponding explanation the possibility of generating zombie or orphan processes for each of the following pairs of values of *delay1* and *delay2*:

***b1)*** *delay1=1000; delay2=10*;

Normal operation, when the children end sentence "sleep(delay1)", and therefore their suspended state, the parent will be already waiting for them in the "wait()" loop.

***b2)*** *delay1=10: delay2=1000*;

The greater the difference, the longer the children can stay in zombie state, since delay2 delays the parent process to make the "wait()" call.

**3.** A time-sharing system has a Round Robin short-term process scheduler, with quantum **q = 2 tu** and a single ready queue. When several events occur simultaneously then the order of insertion in the ready queue is: new process, coming from I/O and coming from CPU (quantum end). The following table shows an scheduling for processes A, B and C, that arrive at instants t = 0, t = 3 and t = 8 respectively, and whose standing alone processing descriptions are:

| | | | | | |
|---|---|---|---|---|---|
| A (t=0) | 4CPU+ 2I/O + 4CPU + 3I/O + 1CPU | B (t=3) | 6CPU + 1I/O + 2CPU | C(t=8) | 3CPU+2I/O+1CPU |

Due to an implementation failure, the scheduler does not work according to the proposed specifications and so it makes mistakes. Detect in the table the instant when an error occurs for the first time, indicate the error reason and do again the scheduling from that instant in the available empty columns on the table.

**(1,2 points = 0,9 + 0,3)**

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

**3**

**a)** Error time instant T = 8

Reason why: Queue insertion is not done correctly, C (new process) must be inserted before A (process that ends the quatum)

**b)** Indicate the turnaround time and the waiting time for every process:

| | A | B | C |
|---|---|---|---|
| Turnaround time | 18 | 17 | 13 |
| Waiting time | 4 | 7 | 6 |

**4**. Given the following string of 52 characters corresponding to a DNA gene:
CACTCAGCACGAA GGGCAGAGGAATG CTTACCGTCCTGA GCCACCCACCAGC

We want to find in what positions the CAG amino acid is referenced, with a design based on 4 concurrent threads. Each thread analyzes a single fraction of 13 characters of the gene, so that one thread analyzes the first 13 characters, another the 13 following, etc. The function "find_amino" receives a pointer to the position of the first character of the fraction that the assigned thread has to process and, in case of finding the amino acid in that fraction, marks the first character where it appears by overwriting the corresponding character 'C' by '*'. The main program, after making sure that all the occurrences of the amino acid have been marked, looks for the marked positions and sends them to the standard output in order, separated by an space. **(1,0 points = 0,5 + 0,25 + 0,25 )**

```
 1  #include <all_needed.h>
 2  #define NFRAC 4
 3  #define GENSIZE 52
 4
 5  char gen[] = "CACTCAGCACGAAGGGCAGAGGAATG
 6  CTTACCGTCCTGAGCCACCCACCAGC";
 7  char amino[] = "CAG";
 8
 9  void *find_amino(void *ptr) {
10    char *pgen= (char*) ptr;
11    int i;
12    for (i=0; i<GENSIZE/NFRAC-2; i++) {
13      if (pgen[i] == amino[0] &&
14          pgen[i+1] == amino[1] &&
15          pgen[i+2] == amino[2])
16        pgen[i] = '*';
17  }
```

```
18  int main() {
19    int i;
20    char *pfrac;
21    pthread_attr_t attrib;
22    pthread_t thread[NFRAC];
23    pthread_attr_init(&attrib);
24
25    for (i = 0; i<NFRAC; i++) {
26      pfrac= gen + i * GENSIZE/NFRAC;
27
28  /**complete**/
...
      for (i = 0; i<GENSIZE; i++) {
        if (gen[i]=='*')
           printf("%d ", i);
     }
```

**4**

**a)** Use the variables already declared in the program and fill in the missing lines of code in the space marked as "/ ** complete ** /", so that the creation and waiting of the 4 threads "find_amino" according to the problem solution proposed.
**NOTE**. Functions definition:
*int pthread_join(pthread_t thread, void **retval);*
*int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);*

```
25  for (i = 0; i<NFRAC; i++){
26    pfrac= gen + i * GENSIZE/NFRAC;
      pthread_create(&thread[i], &attrib, find_amino, pfrac);
    }

    for (i = 0; i<NFRACTIONS; i++) {
      pthread_join(thread[i], NULL);
    }
```

**b)** Explain what is the maximum number of threads belonging to this process that will run concurrently.
Five, the main one (main) + the 4 threads "find_amino"

**c)** Explain if this program execution may lead to any race condition. It is not possible, because the 4 threads access different positions of the gene chain, which is the only variable they share. On the other hand, the main thread does not concurrently access this chain either, but waits until the other 4 threads have finished their accesses.

**5**. Given the following code of three concurrent processes whose initial values of variable X and semaphores S1, S2 are the following: X = 1, S1 = 1, S2 = 1.

| A | B | C |
|---|---|---|
| P(S1);<br>X = X + 1;<br>V(S1); | P(S1);<br>V(S1);<br>P(S2);<br>X = X - 1;<br>V(S2); | P(S1);<br>P(S2);<br>X = X + 1;<br>V(S2); |

These three processes are executed in a system where the order in which the processor is assigned for the first time is ALWAYS determined by the order of arrival in the ready queue. After getting the CPU for the first time, the first instruction (P (S1)) is ALWAYS executed, but from that point on, context changes can happen at any time. Indicate for every given sequence: **1.** If the three processes can end; **2.** If it is possible for a race condition to happen and; **3.** The final value of X, if all processes end.

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

**5**

**a)** A, B, C

The three processes end, there is NO race condition and X = 2. SEQUENCE: Enter A first, decrement S1, so that B can not enter, increase X, and increase S1. Then B is executed, which subtracts and allows to pass to C and finally C adds 1. Therefore, the final value will be X = 2, NO deadlock can occur, since while A is being executed, even if a context change occurs on line X = X + 1, processes B and C are blocked in the first lines, and then B and C with semaphore S2 are protecting the critical section.

**b)** B, A, C

All three processes end and a race condition is possible. P2 starts, put semaphore S1 to 0 and puts it back to one, so A can awake when a possible context change occurs executing the critical section..

**6**. A system with 4 GBytes of main memory uses demand paging with an LRU as replacement algorithm, with **LOCAL** scope. The logical address space is 4 GBytes and the page size is 4 KBytes. At a given moment the execution of processes A and B begins, and the system assigns 4 frames (numbered from 0 to 3) for A and B that are distributed according to fair allocation. All frames are initially empty .

**(1,6 points = 0,2 + 0,8 + 0,6)**

**6**

**a)** Obtain the reference string from the following sequence of logical addresses (in hexadecimal):
(A:1A407125) (A:1A4072C9) (A:4FB30190) (B:00400000) (A:1A407C41) (B:1000458F) (A:24B71AFF) (B:1E861100) (B:1E861ABC) (A:1A407642) (B:0040093C) (A:4FB30456)

A:1A407; A:4FB30; B:00400; A:1A407; B:10004; A:24B71; B:1E861; A:1A407; B:00400; A:4FB30

**b)** Indicate the evolution of main memory content for the reference string obtained in section a). Frames are assigned in increasing order. Also indicate where page faults happen.

| **Frame** | A:1A407<br>t=0 | A:4FB30<br>t=1 | B:00400<br>t=2 | A:1A407<br>t=3 | B:10004<br>t=4 | A:24B71<br>t=5 | B:1E861<br>t=6 | A:1A407<br>t=7 | B:00400<br>t=8 | A:4FB30<br>t=9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | A:1A407<br>*(0) | A:1A407<br>(0) | A:1A407<br>(0) | A:1A407<br>(3) | A:1A407<br>(3) | A:1A407<br>(3) | A:1A407<br>(3) | A:1A407<br>(7) | A:1A407<br>(7) | A:1A407<br>(7) |
| **1** | | A:4FB30<br>0*(1) | A:4FB30<br>(1) | A:4FB30<br>(1) | A:4FB30<br>(1) | A:24B71<br>*(5) | A:24B71<br>(5) | A:24B71<br>(5) | A:24B71<br>(5) | A:4FB30<br>* (9) |
| **2** | | | B:00400<br>*(2) | B:00400<br>(2) | B:00400<br>(2) | B:00400<br>(2) | B:1E861<br>*(6) | B:1E861<br>(6) | B:1E861<br>(6) | B:1E861<br>(6) |
| **3** | | | | | B:10004<br>*(4) | B:10004<br>(4) | B:10004<br>(4) | B:10004<br>(4) | B:00400<br>* (8) | B:00400<br>(8) |

TOTAL PAGE FAULTS :

8, 4 of them with replacement

**c)** After the reference string in section b), two new logical addresses are generated: (A: 1A407FFF) and (B: 1E861008). Explain what physical addresses the MMU will generate for each of them.

A: 1A407FFF → Hit: page A:1A407 is allocated into frame 0 so the physical address is 00000FFF

B: 1E861008 → Hit: page B:1E861 is allocated into frame 2 so the physical address is 00002008

7. Considering the inheritance mechanism in Unix processes and POSIX calls, answer the following sections:

**(1,3 points = 0,6 + 0,7)**

7

**a)** Fill the file descriptor tables, indicating the content of the non-empty descriptors, during the execution of each of the processes involved in the following command:

```
$ ls -l 2> err | tee f1 | grep ".c" > f2
```

The command has to run and end correctly. Remember that **tee** command writes its standard input in both the file that is passed to it as a parameter and the standard output.

**NOTE**. Name the first pipe "pipeA" and the second one "pipeB"

| **ls** | |
|---|---|
| 0 | STDIN |
| 1 | pipeA[1] |
| 2 | "err" |
| 3 | |
| 4 | |

| **tee** | |
|---|---|
| 0 | pipeA[0] |
| 1 | pipeB[1] |
| 2 | STDERR |
| 3 | "f1" |
| 4 | |

| **grep** | |
|---|---|
| 0 | pipeB[0] |
| 1 | "f2" |
| 2 | STDERR |
| 3 | |
| 4 | |

**b)** Supposing that system calls execute without errors, complete the following C program with the required sentences and system calls (one in each line with its number underlined) so that its execution be equivalent to the following command line : `$ ls -l 2> err | tee f1 | grep ".c" > f2`

```
 1. #include <all_needed.h>
 2. #define newfile (O_RDWR | O_CREAT | O_TRUNC)
 3. #define mode644 (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
 4. int main() {
 5.    int pipeA[2], pipeB[2], fd;
 6.     pipe(pipeA);
 7.    if (fork()) {
 8.        fd = open("err", newfile, mode644);
 9.        dup2(fd,STDERR_FILENO);
10.        dup2(pipeA[1], STDOUT_FILENO);
11.        close(pipeA[0]); close(pipeA[1]); close(fd);
12.        execlp("ls", "-l", NULL);
13.     } else {
14.        pipe(pipeB);
15.        if (fork()) {
16.            dup2(pipeA[0], STDIN_FILENO);
17.            dup2(pipeB[1], STDOUT_FILENO);
18.            close(pipeA[0]); close(pipeA[1]);
19.            close(pipeB[0]); close(pipeB[1]);
20.            execlp("tee", "tee", "f1", NULL    ); /*complete*/
21.        } else {
22.            close(pipeA[0]); close(pipeA[1]);
23.            fd = open("f2", newfile, mode644   ); /*complete*/
24.            dup2(fd,STDOUT_FILENO);
25.            dup2(pipeB[0], STDIN_FILENO);
26.            close(pipeB[0]); close(pipeB[1]); close(fd);
27.            execlp("grep", "grep", ".c", NULL);
28.        }
29.   }
30.   return 0;
31. }
```

**8**. Given the following listing corresponding to the contents of a directory in a POSIX system:

```
 i-node     permissions links   user   group    size    date          name
32448485    drwxrwxr-x    2     pep   alumne    4096 ene  8  11:57  .
       1    drwxr-xr-x   11     pep   alumne  236871 ene  8  12:02  ..
32448793    -rw-rw----    1     pep   alumne     310 ene  9  10:37  f1
32448802    --w-rw-r--    3     pep   alumne     343 ene  9  11:15  f3
32448805    -rw----r--    3     pep   alumne     343 ene  9  10:33  f4
32448752    -rwxrwxrwx    1     pep   alumne    5824 ene  9  15:17  f5
33373385    -r-sr-xr-x    1     ana   alumne     706 ene  9  10:35  cp1
32448804    lrwxrwxrwx    1     ana   alumne       8 ene  9  10:36  cp2 ->cp1
32448803    lrwxrwxrwx    1     ana   profes       8 ene  9  10:40  f2 ->f1
```

Commands `cp1` and `cp2` require the name of two files as arguments: `cp1(or cp2) file1 file2`, result in the content of `file1` being copied to `file2` and if `file2` does not exist then it is created.

**(1,0 points = 0,75 + 0,25)**

**8**

a) Rellene la tabla e indique en caso de éxito cuales son los permisos que se comprueban y, en caso de error, cuál es el permiso que falla y porqué.

| **User** | **Group** | **Command** | **Does it work? (YES OR NO)** |
|---|---|---|---|
| **ana** | **profes** | **./cp1 f1 f3** | NO |
| **Explain** Process with identity (ana, profes) has execution permission for ./cp1. The process maintains its identity (ana, profes) so it will not have permission to read file f1, since it checks the third permission triplet. | | | |
| **pep** | **alumne** | **./cp2 f3 f2** | YES |
| **Explain** As ./cp2 is a link to ./cp1 permissions checked at process (pep, alumne) are those of ./cp1. You can execute it by having the identity of (ana, alumne). Check the second triplet of f3 and you have read permission. And to write in f2, since it is a link to f1, the second triplet of f1 is consulted and it has permission to write. | | | |
| **pep** | **disca** | **./cp1 f4 f6** | NO |
| **Explain** Process with identity (pep, disk) checks permissions of the third triplet and yes it can execute ./cp1, changing its identity to (ana, disc). It has, therefore, read permissions in f4 (third triplet) but does not have write permissions on the directory, to create file f6 (third triplet). | | | |

**b)** Explain what type of file is **f2** and indicate the numbers of the i-nodes that will be required to access in order to read the contents of that file .

File f2 is a symbolic link to f1 since they do not share the same i-node.
For being a symbolic link, to read the content of f2 we have to first access the data block of f2, where it tells us the path to get to the content of f1.

The i-nodes accessed will be: 1, 32448803, 32448485, 32448793

**9**. A 512 MByte Minix file system has the following features:

- 32-Byte i-nodes, with 7 direct pointers to zones, 1 indirect and 1 double indirect
- 16-bit (2-Byte) pointers to zone
- 16-Byte directory entries (14 Bytes for name and 2 Bytes for i-node id)
- **1 Block = 1 Zone = 2 KBytes**

**(1,0 points = 0,6 + 0,4)**

**9**

**a)** Obtain the sizes for each header element, considering the file system formated for 16384 (16K) i-nodes.

| Boot | Super block | i-node bit map | Zone bit map | i-nodes | Data area |
|---|---|---|---|---|---|
| 1 Block | 1 Block | 1 Block | 16 Blocks | 256 Blocks | |

i-nodes bit map → $16384 / 2K * 8 = 2^4\,2^{10} / 2^1\,2^{10}\,2^3 = 1$ → 1 Block

Maximum number of zones = $512MB / 2KB = 2^9\,2^{20} / 2^1\,2^{10} = 256$ K zones

Zones bit map → Nº Zones / block size in bits = $256K / 2K * 8 = 2^8\,2^{10} / 2^1\,2^{10}\,2^3 = 2^4 = 16$ Blocks

i-nodes → $16384 * 32Bytes / 2KBytes = 2^4\,2^{10}\,2^5 / 2^1\,2^{10} = 256$ Blocks

**b)** Calculate and justify the minimum zone size with which a 512-MByte partition should be formatted if the i-node is designed with only **eight** 32-bit direct pointers to zone, and you want to have files of size up to 256 MBytes. Justify the zone size and what advantages and disadvantages this design has for (large) 256-MByte files and (small) 1-Byte files .

With 32-bit pointers to zone, partitions of up to 4 Gibazones could be addressed, but if only 8 direct pointers are available in each i-node, a file can only have a maximum of 8 zones assigned. If we want 256-MByte files, each zone will have to be 32Mbytes ($2^8\,2^{20} / 2^3 = 2^5\,2^{20}$).

Large files: Quick access since all zone pointers are in the i-node and provide direct access.

1-Byte Files: There is a lot of internal fragmentation because if you assign at least one zone you are wasting a lot of partition space → 32MBytes-1Byte