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

fSO

**EEE2 Exam**
December 16th, 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 writing. Answer in a brief and precise way.**
- **The exam has 10 questions, every one has its score specified.**

---

**1.** Consider a program that contains a function *f* that prints the address of visible objects (local, global, and dynamic variables or functions) and the process memory map. Along its execution it displays a memory map made of 12 regions as shown below:

| /* Source code */ | Memory map at program comment /* Show map */ |
|---|---|
| `#include <…`<br>`float x;`<br>`void * p;`<br><br>`void f(int k){`<br>` int L;`<br>`/* Show map */`<br>`  ...`<br>`  }`<br><br>`main() {`<br>`  p=malloc(8);`<br>`  f(100);`<br>`}` | a 08048000-08049000 r-xp 00000000 00:14 5374410 /home/m/programa<br>b 08049000-0804a000 r--p 00000000 00:14 5374410 /home/m/programa<br>c 0804a000-0804b000 rw-p 00001000 00:14 5374410 /home/m/programa<br>d 09cbe000-09cdf000 rw-p 00000000 00:00 0 [heap]<br>e b7524000-b76c8000 r-xp 00000000 08:06 291538 /lib/i386-linux-gnu/libc-2.15.so<br>f b76c8000-b76ca000 r--p 001a4000 08:06 291538 /lib/i386-linux-gnu/libc-2.15.so<br>g b76ca000-b76cb000 rw-p 001a6000 08:06 291538 /lib/i386-linux-gnu/libc-2.15.so<br>h b76e3000-b76e4000 r-xp 00000000 00:00 0 [vdso]<br>i b76e4000-b7704000 r-xp 00000000 08:06 291528 /lib/i386-linux-gnu/ld-2.15.so<br>j b7704000-b7705000 r--p 0001f000 08:06 291528 /lib/i386-linux-gnu/ld-2.15.so<br>k b7705000-b7706000 rw-p 00020000 08:06 291528 /lib/i386-linux-gnu/ld-2.15.so<br>l bfa0f000-bfa30000 rw-p 00000000 00:00 0 [stack] |

Write and explain the letter corresponding to the region where every one of the following objects will be allocated:

0,75 points

| 1 | Objet | Region | Explenation |
|---|---|---|---|
| | x | C | x is a global variable → it is allocated in the variables (rw) region with support in the executable file |
| | k | l | k is a parameter → it is allocated in the stack region with funtion f is going to be executed |
| | f | a | f es a program function → it is allocated in the code region (r-x) with support in the executable file |
| | L | l | x is a local variable → it is allocated in the stack region while f is being executed |
| | p | c | p is a global variable → it is allocated in the variables (rw) region with support in the executable file |
| | *p | d | *p is allocated dynamically in the heap region when calling to malloc |

---

**2.** Let consider a system with 512 MB of logical addressing space, with 4 KB page size and with 64 MB of physical memory.

a) If memoy is managed by paging, with a maximum number of 4 pages per process, the following translation between logical and physical addresses has been done along the execution of process A:.

| Logical addr | 10185 | 28 | 14050 | 7248 |
|---|---|---|---|---|
| Physical addr | 1845193 | 102428 | 5858 | 27728 |

From the former information fill the page table of process A.

b) Describe the logical address format if **segmentation with paging** is applied with a maximum of 32 segments per process.

c) In the previous **segmentation with paging** model, and considering the following page table of segment 1 for a given process, obtain the logical addresses that correspond to physical addresses 8220 and 25096 in that process.

| | Segment 1 page table | |
|---|---|---|
| | *Frame* | *Valid bit* |
| 0 | 4 | Valid |
| 1 | 5 | Valid |
| 2 | 6 | Valid |
| 3 | 2 | Valid |

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

**2**

**a)**

| | Process A page table | |
|---|---|---|
| | *Frame* | *Valid bit* |
| 0 | 25 | valid |
| 1 | 6 | valid |
| 2 | 450 | valid |
| 3 | 1 | valid |

**b)** Logical address format in segmentation with paging

*Logical space is 512MBytes= $2^{29}$ bytes → 29 bit logical address.*
*Page size 4Kbyte = $2^{12}$, so the offset is 12 bit (in both pages and frames)*
*There are 32 segments per process → 5 bits segment id*

29 bits

| Segment (5 bits) | Segment offset (24 bits) |
|---|---|
| B28 B24 | B23 B0 |

29 bits

| Segment (5 bits) | Page id (12 bits) | Offset (12 bits) |
|---|---|---|
| B28 B24 | B23 B12 | B11 B0 |

**c)**

| Physical addresses | Formula to compute the logical addr. | Logical address |
|---|---|---|
| 8220 | S1,3*4096 + 28 | S1,12316 |
| 25096 | S1,2*4096 + 520 | S1,8712 |

**3.** In relation to page faults in a system with virtual memory, indicate if every one of the following statements are true(T) or false(F). (**Note.** One error voids one correct answer)

0,75 points

| **3** T/F | |
|---|---|
| V | When a process requests a new page to expand the stack always happens a page fault, regardless of having or not free frames to allocate it. |
| V | There is a page fault when is accessed a page that is still in the frame reservation pool. |
| V | A page fault happens when a process access the first time to a page of code or data that is not allocated in physical memory |
| F | Page faults only happen when there are no free frames in physical memory to allocate a new page. |
| F | A page fault always involves writing one page from memory to disk and reading one page from disk to memory. |

**4.** Let consider a virtual memory system with **two levels of paging** and the following features:

- 24 bit logical addresses and 22 bit physical addresses
- 4 Kbyte page size
- 16 entries in the first level page table
- OPTIMAL replacement algorithm and LOCAL scope
- 4 frames is the máximum number of frames that a porcess can get

Now there is process P in memory that uses frames 0 to 3, as shown in the following figure, being p1 and p2 the page indexes to first and second level, repectively.

| Frame | p1, p2 |
|---|---|
| 0 | (0x8,0x18) |
| 1 | (0x8,0xDA) |
| 2 | (0x3,0x85) |
| 3 | (0xB,0x15) |

**Note**. Notation 0x, means hexadecimal

a) Obtain the logical address and physical address formats, specifying the total number of bits and the number of bits per field.

b) Suppose that from the time instant shown in the table, the CPU references the following sequence of process P addresses: 0x154891 0x385F94 0x8DA122 0xB15679 0xB15A8C, 0x3851E9, 0x36C98A, 0x154917 0x1541CB, 0x385A03, 0x385545, 0x2F223C, 0x2F2B21, 0x1546F5. Show the evolution of the physical memory content and obtain the number of page faults.

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

**4**

**a)** Logical and physical addresses structure

*Page size is 4Kbytes = $2^{12}$, so the offset is 12 bits (both for frames and pages). Physical addresses have 22 bits, so they remain 10 bits for frame id. So physical addresses have 22 bits, 10 for the frame id and 12 for the offset.*

22 bits

| Frame (10 bits) | Offset (12 bits) |
|---|---|
| B21 B12 | B11 B0 |

*Logical addresses have 24 bits: 4 for p1 (first level page id), 8 for p2 (second level page id) and 12 for the offset.*

24 bits

| P1 (4 bits) | P2 (8 bits) | Offset (12 bits) |
|---|---|---|
| B23 B20 | B19 B12 | B11 B0 |

**b)** *Reference string (1,54), (3,85), (8,DA), (B,15), (3,85), (3,6C), (1,54), (3,85), (2,F2), (1,54)*

| Frame | | **(1,54)** | (3,85) | (8,DA) | (B,15) | (3,85) | **(3,6C)** | (1,54) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | (8,18) | (1,54) | (1,54) | (1,54) | (1,54) | (1,54) | (1,54) | (1,54) | | |
| 1 | (8,DA) | (8,DA) | (8,DA) | (8,DA) | (8,DA) | (8,DA) | (3,6C) | (3,6C) | | |
| 2 | (3,85) | (3,85) | (3,85) | (3,85) | (3,85) | (3,85) | (3,85) | (3,85) | | |
| 3 | (B,15) | (B,15) | (B,15) | (B,15) | (B,15) | (B,15) | (B,15) | (B,15) | | |

| Frame | (3,85) | **(2,F2)** | (1,54) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | (1,54) | (1,54) | (1,54) | | | | | | | |
| 1 | (3,6C) | (2,F2) | (2,F2) | | | | | | | |
| 2 | (3,85) | (3,85) | (3,85) | | | | | | | |
| 3 | (B,15) | (B,15) | (B,15) | | | | | | | |

Total number of page faults = *3 page faults with replacement*

**5.** Let consider a system with 16 Mbyte of physical memory, 20 bit logical addresses, pages of 4 Kbytes and **demand paging**. It uses SECOND CHANCE with LOCAL SCOPE as replacement algorithm. The system assigns 3 frames to every process. Now the frame assigment and the physical memory state for processes A and B, that are being executed, are the ones shown in the following tables:

| Frame | Process | Page | Reference bit |
|---|---|---|---|
| 0x003 | A | 0xA5 | 1 |
| 0x004 | A | 0x24 | 0 |
| 0x005 | A | 0x6E | 1 |

| Frame | Process | Page | Reference bit |
|---|---|---|---|
| 0x009 | B | 0x9A | 0 |
| 0x00A | B | 0x27 | 1 |
| 0x00B | B | 0x3F | 0 |

From that instant the following sequence of logical addresses are accessed (A, 0x24350) (A, 0x9A000) (B, 0x3A120) (A, 0x99050) (B, 0x3A650) (B, 0x28495) (A, 0x6E350). In the beginning the searching for a victim, for every process, matches the increasing order of the frame id allocated to every processes. Show the evolution of the memory frames involved filling the following table:

1,0 point

**5**

<table>
<tr><th rowspan="2">Frame</th><th rowspan="2">Init page<br>| Ref bit</th><th colspan="7">Referenced page id</th></tr>
<tr><th>A,24</th><th>A,9A</th><th>B,3A</th><th>A,99</th><th>B,3A</th><th>B,28</th><th>A,6E</th></tr>
<tr><td>0x003</td><td>-> A5<br>1</td><td>-> A5<br>1</td><td>9A<br>1</td><td>9A<br>1</td><td>9A<br>1</td><td>9A<br>1</td><td>9A<br>1</td><td>9A<br>1</td></tr>
<tr><td>0x004</td><td>24<br>0</td><td>24<br>1</td><td>-> 24<br>0</td><td>-> 24<br>0</td><td>99<br>1</td><td>99<br>1</td><td>99<br>1</td><td>99<br>1</td></tr>
<tr><td>0x005</td><td>6E<br>1</td><td>6E<br>1</td><td>6E<br>0</td><td>6E<br>0</td><td>-> 6E<br>0</td><td>-> 6E<br>0</td><td>-> 6E<br>0</td><td>-> 6E<br>1</td></tr>
<tr><td>0x009</td><td>-> 9A<br>0</td><td>-> 9A<br>0</td><td>-> 9A<br>0</td><td>3A<br>1</td><td>3A<br>1</td><td>3A<br>1</td><td>-> 3A<br>1</td><td>-> 3A<br>1</td></tr>
<tr><td>0x00A</td><td>27<br>1</td><td>27<br>1</td><td>27<br>1</td><td>-> 27<br>1</td><td>-> 27<br>1</td><td>-> 27<br>1</td><td>27<br>0</td><td>27<br>0</td></tr>
<tr><td>0x00B</td><td>3F<br>0</td><td>3F<br>0</td><td>3F<br>0</td><td>3F<br>0</td><td>3F<br>0</td><td>3F<br>0</td><td>28<br>1</td><td>28<br>1</td></tr>
<tr><td></td><td>Initial</td><td>Hit</td><td>Fault</td><td>Fault</td><td>Fault</td><td>Hit</td><td>Fault</td><td>Hit</td></tr>
</table>

Total number of page faults = 4

**6.** In a time sharing system where processes A and B have been executed, a meassurement was done of the sequence of page references that the CPU has performed, getting the following result:

| Time | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Proc,page | A1 | B2 | A3 | B4 | A2 | B1 | A5 | B6 | A2 | B1 | A2 | B3 |

| Time(cont) | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|
| Proc,page | A7 | B6 | A3 | B2 | A1 | B2 | A3 | B6 |

Considering a working set window of 4, obtain the working set for A and B in t = 6, t = 10, t = 14 and t = 19.

0,5 points

**6**

| Time | Working set |
|---|---|
| t=6 | A={1, 2, 3, 5}<br>B={1, 2, 4} |
| t=10 | A={2, 5}<br>B={1, 4, 6} |
| t=14 | A={2, 3, 7}<br>B={1, 3, 6} |
| t=19 | A={1, 3, 7}<br>B={2, 6} |

---

**7.** The following listing correspond to the content of a directory in a UNIX system:

```
drwxr-xr-x     2 peter    users    4096 sep  8     2012  .
drwxr-xr-x     8 peter    users    4096 dec 10     14:39 ..
-rwsrw-r-x     1 peter    users    9706 sep  9     2012  append
-rw-rw-r--     1 peter    users    4310 sep  9     2012  f1
-r--rw-r--     1 peter    users    4157 sep  9     2012  f2
lrwxrwxrwx     1 peter    users       6 sep  9     2012  new->append
```

Where program "append" appends the content of a file specified as the first argument to another file specified as the second argument. Let's consider the following command:

```
$ append f1 f2
```

Indicate if the following statements about it are true(T) of false(F):
**Note**. One error voids a correct answer.

(0,75 points)

| 7 | T/F | |
|---|---|---|
| | F | User *john* that belongs to group *students* can not start the execution of file "append" |
| | F | User *mary* that belongs to group *users* can start the execution of file "append" but she will get an error when the process "append" will try to write in file *f2* |
| | F | The two users mentioned above, *john* and *mary*, will execute the command successfully |
| | F | User *peter* that belongs to group *users* will execute the command successfully |
| | V | If the following permissions are set for file "append": `-rwxrwsr-x` (the SETGUID bit is set) then user *john* will execute the command succesfully |

**8.** Given the following C code, obtain the content of the file descriptor tables in the code points marked as /* Point 1 */, /* Point 2 */ and /* Point 3 */ for every active process in each point. Obtain also the values of variables fd1, fd2, fd3, fd[0] and fd[1] if they are meaningful.

```
/* C code */
………
fd1=open("f1",…);
close(STDOUT_FILENO);
fd2=open("f2",...);
dup2(fd1,STDERR_FILENO); /* Point 1 */
dup(STDERR_FILENO);
dup(STDIN_FILENO);
fd3=open("f3",...); /* Point 2 */
if (fork()==0) {
   dup2(fd3, STDIN_FILENO);
   close(fd1);
}
close(fd2);
pipe(fd); /* Point 3 */
……
```

0,75 points

**8**

Point = /*Point 1*/
Variable values

fd1=3, fd2=1

File descriptor table

| | |
|---|---|
| 0 | stdin |
| 1 | f2 |
| 2 | f1 |
| 3 | f1 |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

Point = /*Point 2*/
Variable values

fd1=3, fd2=1, fd3=6

File descriptor table

| | |
|---|---|
| 0 | stdin |
| 1 | f2 |
| 2 | f1 |
| 3 | f1 |
| 4 | f1 |
| 5 | stdin |
| 6 | f3 |
| 7 | |

Point = /*Point 3*/
Variable values

fd1=3, fd2=1, fd3=6
fd[0]=1, fd[1]=7

File descriptor table

| | |
|---|---|
| 0 | stdin |
| 1 | pipe[0]/*parent*/ |
| 2 | f1 |
| 3 | f1 |
| 4 | f1 |
| 5 | stdin |
| 6 | f3 |
| 7 | pipe[1]/*parent*/ |

Point = /*Point 3*/
Variable values

fd1=3, fd2=1, fd3=6
fd[0]=1, fd[1]=3

File descriptor table

| | |
|---|---|
| 0 | f3 |
| 1 | pipe[0]/*child*/ |
| 2 | f1 |
| 3 | pipe[1]/*child*/ |
| 4 | f1 |
| 5 | stdin |
| 6 | f3 |
| 7 | |

**9.** The following program sorts a list of numbers by calling program *sort* and using redirections and pipes. The column on the right shows the result of a successful execution.

<table>
<tr><th>/* C code */</th><th>/* Execution result */</th></tr>
<tr><td><pre>
#include “the required ones”
int main(int argc, char *argv[]) {
  int backup, fd[2];
  char list[]=" 4 \n 3 \n 1 \n 2 \n";

  pipe(fd);
  printf("Before:\n %s After:\n", list);

  if (fork()==0){
    dup2 (______,____________);
    close (______); close (______);
    execlp("sort", "sort", NULL);
  } else {
    backup=dup(STDOUT_FILENO);
    dup2 (______, ____________);
    close (______); close (______);

    printf("%s", list);
    dup2 (backup, STDOUT_FILENO);
    wait(NULL);
    printf("That’s all \n");
  }
  return 0;
}
</pre></td><td><pre>
Before:
 4
 3
 1
 2
After:
 1
 2
 3
 4
That’s all
</pre></td></tr>
</table>

a) Complete the parameters of *dup2* and *close* to get the desired program behaviour.
b) Explain what is the purpose of variable *backup*.
c) Explain what will happend if lines are changed in the proposed code as follows:

<table>
<tr><th>/* Initial code */</th><th>/* New code */</th></tr>
<tr><td><pre>
…….
dup2 (backup, STDOUT_FILENO);
wait(NULL);
printf("That’s all \n");
….
</pre></td><td><pre>
…..
dup2 (backup, STDOUT_FILENO);
printf("That’s all \n");
wait(NULL);
…
</pre></td></tr>
</table>

d) Explain what will happend if lines are changed in the proposed code as follows:

<table>
<tr><th>/* Initial code */</th><th>/* New code */</th></tr>
<tr><td><pre>
…
dup2 (backup, STDOUT_FILENO);
wait(NULL);
printf("That’s all \n");
…
</pre></td><td><pre>
….
wait(NULL);
dup2 (backup, STDOUT_FILENO);
printf("That’s all \n");
….
</pre></td></tr>
</table>

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

**9**

**a)** To be completed

```
if (fork()==0){
  dup2 (fd[0], STDIN_FILENO);
  close (fd[0]);
  close (fd[1]);
  execlp("sort","sort",NULL);
} else {
  backup=dup(STDOUT_FILENO);
  dup2 (fd[1], STDOUT_FILENO);
  close (fd[0]);
  close (fd[1]);
}
```

**b)**
As the program changes the standard output by the pile, previously the standard output descriptor is stored in backup to be able to write again on it (last message)

**c)**
If printing before wait() then it is possible that the order be not the same, i.e., the result of the sort() could appear on screen before the message of "That's all". It will depend on the scheduling

**d)**
In this case sort() would be blocked since we keep opened the writing descriptor of the pipe (that is closed by dup2(backup,STDOUT_FILENO)) and so the parent process gets blocked in wait()

---

**10.** A 6 GBytes partition if formated with a MINIX file system with the following parameters:

- 1 block = 1 KByte
- 1 zone = 1 block
- 64 byte i-node with 32 bit pointers to zone (7 direct, 1 indirect and 1 double indirect)
- 32 bit directory entrie
- 8192 i-nodes
- File system organization as follows:

| Boot block | Super block | i-node bit map | Zone bit map | i-nodes | Data area |
|---|---|---|---|---|---|

a) Compute the number of blocks in the i-node bit map, the zone bit map, the i-nodes area and the data area.

b) In this file system the root directory contains:
   - 2 regular files *reg1* y *reg2* of 10 Kbyte each one
   - 1 directory *dir* that itself contains a regular file reg3 of 10 Kbyte

b1) Explain how may zones are used by every one of the former files and what type of information contain those zones.

b2) Explain the total number of occupied i-nodes in this file system and the value of the field "Number of links" in every occupied i-node.

1,5 points (0,75+0,5+0.25)

**10** a)

i-node bit map = 8192/(1024*8) = 1 block

Zones bit map = Nº of zones /(1024*8) = $(6*2^{30}/1024)/(1024*8)$ =

= $6*2^{20}/2^{13} = 6*2^{7}$ = 768 blocks

i-nodes = 8192 * 64 Bytes /(1024 Bytes) = 512 Blocks

File system header = 1 + 1 + 1 + 768 +512 = 1283 Blocks

Nº zone in data area = Nº total blocks – Nº header blocks =

= $6*2^{30}$ / 1024 – 1283 = $6*2^{20}$ – 1283 = 6290173

b1)

| File | Nº of busy zones | Type of information contained |
|---|---|---|
| / | 1 | 5 directory entries reg1, reg2, dir , . and .. = 5*32Bytes = 160 Bytes |
| /reg1 | 11 | 10 zones for data + 1 zona for pointers (3 pointers) |
| /reg2 | 11 | 10 zones for data + 1 zona for pointers (3 pointers) |
| /dir | 1 | 3 directory entries reg3, . and .. = 3*32Bytes = 96 Bytes |
| /dir/reg3 | 11 | 10 zones for data + 1 zona for pointers (3 pointers) |

b2)

There are 5 i-nodes busy, one for every existing file: /, reg1, reg2, reg3 and dir. The number of links for every busy i-node are:

/ -> 3 links

/reg1 -> 1 link

/reg2 -> 1 link

/dir -> 2 links

/dir/reg3 -> 1 link