Grau en Enginyeria Informàtica (GEI)

Sistemes Operatius

2025-2026 (1er Semestre)

# Examen Primer Parcial Novembre 2025

# Solució

Jordi Mateo jordi.mateo@udl.cat

---

**— Informació important —**

**Temps**: 2 h

- Aquest examen avalua els coneixements teòrics i pràctics de l'assignatura.
- Només es permet l'ús d'una fulla d'apunts en format A4; preparada prèviament per l'estudiant. Altres materials de consulta no estan permesos.
- L'examen s'ha de respondre directament en el mateix document proporcionat i amb bolígraf; les respostes escrites amb llapis no seran vàlides.
- Qualsevol sospita de plagi o còpia comportarà sancions d'acord amb la normativa acadèmica.

**Happy coding!!**

**— Omplir per l'estudiant —**

Nom: .....

Cognoms: .....

DNI: .....

- Declaro que jo he entès les instruccions.

Signatura: .....

---

**— Omplir pel professor —**

| **Exercisis** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| **Punts** | 2 | 1 | 1 | 1 | 1 | 3 | 1 | 10 |
| **Correcció** | | | | | | | | |

## 1: Teoria (V/F) — 2 Punts

Indica si les següents afirmacions són certes o falses. No requereix justificar les respostes.

- Cada encert suma **0.2 punts**.
- Cada error resta **0.1 punts**.
- La resposta en blanc no suma ni resta punts.
- La resposta doble (marcar tant cert com fals) resta **0.1 punts**.

| V | F | |
|---|---|---|
| ■ | □ | … Els arguments d'una funció `char *argv[]` es passen a la pila d'usuari primer els valors i després els punters. |
| □ | ■ | … Un procés pot capturar `SIGKILL` abans que el nucli el finalitzi. |
| □ | ■ | … Quan un procés fa `exec()`, tots els seus fills desapareixen excepte el que ha cridat `exec()`. |
| □ | ■ | … Si un **fill tanca un descriptor de fitxer**, el **pare** ja **no pot escriure-hi**. |
| □ | ■ | … Les crides `read()` o `write()` sempre retornen després de completar la transferència de dades. |
| □ | ■ | … Una **syscall** provoca una *interrupció de maquinari* que passa el control al kernel. |
| □ | ■ | … Un procés en estat zombie pot rebre senyals. |
| □ | ■ | … Una *page fault* sempre finalitza el procés que l'ha provocat. |
| ■ | □ | … Els processos *orfes* són adoptats pel procés **init/systemd**. |
| □ | ■ | … Les **variables locals** d'un programa main.c **no estan a la pila**. |

****valor 0.2****

Tot i que no es demana justificar les respostes, si creus convenient, pots escriure el teu raonament a continuació:

**Solution:**

## 2: Dualitat — 1 Punt

- Identifica quin mecanisme de transferència de mode s'està utilitzant en cada una de les línies marcades amb (1), (2), (3) i (4) del següent codi en C. Justifica la teva resposta breument.

```c
int main() {
    alarm(2);                // (1)
    int *p = NULL;
    *p = 42;                 // (2)
    write(1, "Hola!\n", 6); // (3)
    while (1);               // (4)
    return 0;
}
```

**Solution:**

(1) Crida a sistema: `alarm()` és una syscall que configura una alarma per enviar un senyal després d'un temps especificat. El procés sol · licita al nucli que configuri un temporitzador, cosa que requereix accés privilegiat al maquinari.

(2) Excepció: L'accés a una adreça nul · la (`*p = 42;`) provoca una excepció (segmentation fault), que és un tipus d'excepció que s'ha de gestionar al nucli.

(3) Crida a sistema: `write()` és una syscall que escriu dades en un descriptor de fitxer. Això també implica una transferència de mode de l'espai d'usuari al nucli.

(4) Interrupció: Quan l'alarma configurada amb `alarm(2)` expira, el nucli envia un senyal (SIGALRM) al procés, provocant una interrupció que fa que el procés canviï del mode usuari al mode nucli per gestionar el senyal.

## 3: Miscel · lània en C — 1 Punt

Assumint que `sizeof(int)== 4` i `sizeof(int*)== 8`, considera el següent codi en C i respon les preguntes següents:

```c
int* t(int n, int *b) {
    int a[4] = {1,2,3,4};
    for (int i = 0; i < n; i++){
        a[i] = b[i] + 1;
    }
    return a;
}

int main(void) {
    int* b = malloc(4 * sizeof(int));
    for (int i = 0; i < 4; i++) {
        b[i] = i * 10;
    }
    int *p = t(4, b);
    for (int i = 0; i < 4; i++) {
        printf("%d ", p[i]);
    }
    free(b);
    return 0;
}
```

- Dibuixa el mapa de memòria d'aquest programa just abans d'executar la línia `return a;` dins de la funció `t()`. Cal indicar la zona de mèmoria, les variables i funcions, el seu contingut, la mida en bytes. [**0.5 punts**]

**Solution:**

| Zona de memòria | Variables/Funccions | Contingut | Mida (bytes) |
|---|---|---|---|
| Text (codi) | main(), t() | Codi de les funcions | Variable |
| Heap | b | [0, 10, 20, 30] | 16 (4 ints) |
| Stack | p | NULL | 8 (punter) |
| Stack | a | [11, 21, 31, 41] | 16 (4 ints) |
| Stack | n | n=4 | 4 (int) |
| Stack | b | b=adreça de b | 8 (punter) |

- Indica quin problema té aquest codi i què pot passar en temps d'execució. [**0.25 punts**]

**Solution:**
La funció `t()` retorna l'adreça d'un array local (a) que està emmagatzemat a la pila (stack). Quan `t()` retorna, la seva pila s'allibera i a deixa d'existir. Per tant, `p` (a `main()`) apunta a una zona de memòria invàlida → comportament indefinit (pot imprimir valors aleatoris o provocar segfault).

- Proposa una modificació mínima per evitar el problema i fer que el programa funcioni correctament. [**0.25 punts**]

**Solution:**
Es pot modificar la funció `t()` perquè utiltizi la heap en lloc de la pila per a l'array `a`, d'aquesta manera, els valors romandran accessibles després de sortir de la funció:

```
1 int* t(int n, int *b) {
2     int *a = malloc(4 * sizeof(int));
3     for (int i = 0; i < n; i++) {
4         a[i] = b[i] + 1;
5     }
6     return a;
7 }
```

i fer `free(p);` a `main()` després d'utilitzar `p`.

## 4: Gestió de processos i E/S — 1 Punt

Considera el següent problema: Escriure un programa en C que faci eco de l'entrada estàndard (STDIN) a la sortida estàndard (STDOUT) exactament dues vegades. A continuació es mostra una proposta de solució. Assumeix que la crida a `fork()` és exitosa i no falla mai.

```c
#define INPUT_LEN 64
int main(void) {
    pid_t pid = fork();
    char buffer[INPUT_LEN + 1];
    buffer[INPUT_LEN] = '\0';
    read(STDIN_FILENO, buffer, INPUT_LEN);
    printf("%s\n", buffer);
    exit(0);
}
```

- Aconseguiex l'objectiu d'imprimir l'entrada estàndard dues vegades? Justifica la teva resposta. [**0.4 punts**]

**Solution:**
No. Tot i que el programa crea dos processos, la lectura de l'entrada estàndard no es duu a terme de manera independent:

- Tant el pare com el fill comparteixen el descriptor de fitxer de l'entrada (STDIN_FILENO), que manté un offset comú.
- Això vol dir que, si un dels processos ja ha llegit l'entrada, l'altre trobarà l'entrada buidada i no imprimirà res.

Així, el comportament és indeterminat: de vegades només imprimeix una vegada, o fins i tot pot no imprimir res si la sincronització falla. No garanteix dos ecos complets de l'entrada.

- Identifica 4 problemes i proposa 4 solucions en la execució del codi. [**0.6 punts**]

**Solution:**

- **Manca de sincronització entre pare i fill**: Ambdós processen la mateixa entrada sense coordinar-se. Possible solució: Utilitzar `wait(NULL);` al procés pare per esperar que el fill acabi abans de llegir i escriure.
- **Només un procés realitza la lectura**: L'offset compartit de l'entrada fa que només un procés pugui llegir correctament. Possible solució: Fer que **un procés llegeixi** i enviï el text al **segon procés mitjançant una** `pipe()`.
- **Fer la lectura abans del** `fork()`: Això faria que ambdós processos tinguessin la mateixa dada llegida. Per assegurar que ambdós processos tinguin la mateixa entrada, es podria llegir abans del `fork()`. Imprimir després del `fork()`.
- **Mida fixa del buffer (64 bytes)**: Si l'entrada supera 64 caràcters, es perden dades. Possible solució: Llegir dins d'un bucle fins que `read()` retorni 0.
- **No es comprova el nombre de bytes llegits**: Pot imprimir contingut invàlid o incomplert. Possible solució: Guardar el valor retornat per `read()` i utilitzar-lo a `write()` o `printf()` per limitar la sortida.

## 5: Senyals i Processos — 1 Punt

Considera el següent fragment d'un programa (que té línies de codi que falten). Omple els espais en blanc del codi per assegurar que la sortida d'aquest programa sigui sempre les següents dues línies en el següent ordre (sota totes les intercalacions o planificacions):

```
1 val = 0
2 val = 1
```

No més d'una crida a funció (o crida a sistema) per espai en blanc! La teva solució no pot fer assignacions a la variable **val**. Tampoc es permet l'ús de `printf()` o altres instruccions que imprimeixin a la sortida estàndard.

```
1  void rectangle () {
2      pid_t oval = getppid();
3      ________________________; //1
4  }
5  void cercle(int i) {
6      val = 1;
7  }
8  int val = 0;
9  int main() {
10     pid_t pid = fork();
11     ____________________________; //2
12     if (pid == 0) {
13         ________________________; //3
14     } else {
15        ________________________; //4
16     }
17     printf"(val=%d\"n,val);
18 }
```

**Solution:**
Aquí tens una possible solució per omplir els espais en blanc del codi:

```
1 //1 kill(oval, SIGCONT);
2 //2 signal(SIGCONT, cercle);
3 //3 rectangle();
4 //4 wait(NULL);
```

## 6: Communicació i Recobriments — 3 Punts

Assumeix que les següents comandes estan disponibles al sistema i si inspecciones la seva pàgina de manual (`man`) et mostren la següent informació:

`man cat`

```
1 cat - concatenate files and print on the standard output
```

`man tee`

```
1 tee - read from standard input and write to standard output and files
2 Copy standard input to each FILE, and also to standard output.
```

`man grep`

```
1 grep - print lines that match patterns
2 grep searches for PATTERNS in each FILE.  PATTERNS is one or more patterns
      separated by newline characters, and grep prints each line that matches a
      pattern.  Typically PATTERNS should be quoted when grep is used in a shell
      command.
3 A FILE of ""- stands for standard input.  If no FILE is given, recursive searches
      examine the working directory, and nonrecursive searches read standard input.
```

`man wc`

```
1 wc - print newline, word, and byte counts for each file
2 Print newline, word, and byte counts for each FILE, and a total line if more than
      one FILE is specified.  A word is a non-zero-length sequence of printable
      characters delimited by white space.
3 With no FILE, or when FILE is -, read standard input.
4 -l print only the newline counts
```

Considera el següent codi en C i *assumeix que funciona correctament, en cas de necessitat pots assumir que els fills es creen en ordre d'execució.*

```c
int fd[2];
pipe(fd);

if (fork() == 0) {
    int f = open("file.txt", O_RDONLY);
    dup2(f, STDIN_FILENO); close(f);
    dup2(fd[1], STDOUT_FILENO); close(fd[0]); close(fd[1]);
    execlp("cat", "cat", NULL);
}
if (fork() == 0) {
    dup2(fd[0], STDIN_FILENO);
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]); close(fd[1]);
    execlp("tee", "tee", "file_copy.txt", NULL);
}
if (fork() == 0) {
    dup2(fd[0], STDIN_FILENO);
    dup2(fd[1], STDOUT_FILENO);
    close(fd[0]); close(fd[1]);
    execlp("grep", "grep", "pattern", NULL);
}
if (fork() == 0) {
    dup2(fd[0], STDIN_FILENO);
    int devnull = open("/dev/null", O_WRONLY);
    dup2(devnull, STDERR_FILENO);
    close(fd[0]); close(fd[1]);
    execlp("wc", "wc", "-l", NULL);
}
close(fd[0]); close(fd[1]);
wait(NULL); wait(NULL); wait(NULL);wait(NULL);
exit(0);
```

- Explica què fa aquest codi i quina és la comanda equivalent. [**0.5 punt**]

**Solution:**
Aquest codi crea una canalització (pipe) entre tres processos fills que executen les comandes `tee`, `grep` i `wc`. La comanda equivalent seria:

```
cat < file.txt | tee file_copy.txt | grep pattern | wc -l 2> /dev/null
```

- Hi ha algun problema amb aquest codi? *Justifica la teva resposta.* [**0.5 punt**]

**Solution:**
El problema amb aquest codi és que tots els processos fills comparteixen el mateix extrem de lectura i escriptura de la pipe (fd[0], fd[1]). Això pot causar problemes perquè `grep` i `wc` poden competir per llegir les dades, i això pot portar a comportaments inesperats o pèrdua de dades.

- Mostra la taula de descriptors oberts sobre cada procés just abans d'acabar (`exit()`). Mostra l'evolució, tatxant els tancats i escrivint al costat els nous. [**0.5 punt**]

**Solution:**

| | Pare | Fill 1 `cat` | Fill 2 `tee` | Fill 3 `grep` | Fill 4 `wc` |
|---|---|---|---|---|---|
| 0 | 0 | ~~0~~ file.txt | ~~0~~ fd[0] | ~~0~~ fd[0]] | ~~0~~ fd[0] |
| 1 | 1 | ~~1~~ fd[1] | ~~1~~ fd[1] | ~~1~~ fd[1] | 1 |
| 2 | 2 | 2 | 2 | 2 | /dev/null |
| 3 | fd[0] | ~~fd[0]~~ | ~~fd[0]~~ file_copy.txt | ~~3~~ fd[0] | ~~3~~ fd[0] |
| 4 | fd[1] | ~~fd[1]~~ | ~~fd[1]~~ | ~~4~~ fd[1] | ~~4~~ fd[1] |
| 5 | | ~~file.txt~~ | | | ~~/dev/null~~ |

- Com podriem gestionar de forma **concurrent** diferents patrons **patterns**. [**0.5 punt**]

**Solution:**

```
.\prog pattern1 || .\prog pattern2 || .\prog pattern3
```

- Imagina que vols passar el patró a cerca així `echo "pattern" | ./my_program` i que el teu programa faci el `grep` amb aquest patró. Com ho faries? [**0.5 punt**]

**Solution:**
Podríem llegir el patró des de l'entrada estàndard al principi del programa i després passar aquest patró com a argument a la comanda `grep` dins del procés fill corresponent. Utilitzaríem `fgets()` o `read()` per llegir el patró.

- Imagina que vols passar el patró a cerca així `echo "pattern" > myfifo` i que el teu programa faci el `grep` amb aquest patró. Com ho faries? [**0.5 punt**]

**Solution:**
Podríem crear un fitxer FIFO (named pipe) abans d'executar el programa i després llegir el patró des d'aquest fitxer FIFO dins del programa. Utilitzaríem `open()` per obrir el FIFO i `read()` per llegir el patró. Posteriorment, passaríem aquest patró a `grep` abans de fer el `execlp()`.

## 7: Crides a sistema i Creació de processos — 1 Punt

Consulta la pàgina de manual de la crida a sistema `mmap()` i respon les següents preguntes:

```
mmap() creates a new mapping in the virtual address space of the calling process.
    The starting address for the new mapping is specified in addr.  The length
    argument specifies the length of the mapping (which must be greater than 0). If
    addr is NULL, then the kernel chooses the (page-aligned) address at which to
    create the mapping; this is the most portable method of creating a new mapping.
     If addr is not NULL, then the kernel takes it as a hint about where to place
    the mapping; on Linux, the kernel will pick a nearby page boundary (but always
    above or equal to the value specified by /proc/sys/vm/mmap_min_addr) and
    attempt to create the mapping there. If another mapping already exists there,
    the kernel picks a new address that may or may not depend on the hint.  The
    address of the new mapping is returned as the result of the call. The contents
    of a file mapping (as opposed to an anonymous mapping; see MAP_ANONYMOUS below)
    , are initialized using length bytes starting at offset offset in the file (or
     other object) referred to by the file descriptor fd.  offset must be a multiple
     of the page size as returned by sysconf(_SC_PAGE_SIZE). After the mmap() call
    has returned, the file descriptor, fd, can be closed immediately without
    invalidating the mapping.

The prot argument describes the desired memory protection of the mapping (and
    must not conflict with the open mode of the file).  It is either PROT_NONE or
    the bitwise OR of one or more of the following flags:

PROT_EXEC  Pages may be executed.
PROT_READ  Pages may be read.
PROT_WRITE Pages may be written.
PROT_NONE  Pages may not be accessed.

The flags argument determines whether updates to the mapping are visible to other
    processes mapping the same region, and whether updates are carried through to
     the underlying file. This behavior is determined by including exactly one of
    the following values in flags:

MAP_SHARED
Share this mapping.  Updates to the mapping are visible to other processes mapping
    the same region, and (in the case of file-backed mappings) are carried through
    to the underlying file. (To precisely control when updates are carried through
    to the underlying file requires the use of msync(2).)
MAP_SHARED_VALIDATE (since Linux 4.15)
This flag provides the same behavior as MAP_SHARED except that MAP_SHARED mappings
    ignore unknown flags in flags.  By contrast, when creating a mapping
    using MAP_SHARED_VALIDATE, the kernel verifies all passed flags are known and
    fails the mapping with the error EOPNOTSUPP for unknown flags. This mapping
    type is also required to be able to use some mapping flags (e.g., MAP_SYNC).
MAP_PRIVATE
Create a private copy-on-write mapping.  Updates to the mapping are not visible to
     other processes mapping the same file, and are not carried through to the
    underlying file. It is unspecified whether changes made to the file after
    the mmap() call are visible in the mapped region.

RETURN VALUE
On success, mmap() returns a pointer to the mapped area.  On error, the value
    MAP_FAILED (that is, (void *) -1) is returned, and errno is set to indicate the
    error.
```

Analitza el següent codi en C:

```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/mman.h>

int main() {
    char *p = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
    strcpy(p, "Hola món");

    pid_t pid = fork();

    if (pid == 0) {
        p[0] = 'h';
    } else {
        sleep(1);
        printf("%s\n", p);
    }
    return 0;
}
```

Aquesta és una sessió de depuració amb `gdb` del codi anterior.

```
(gdb) break main
Breakpoint 1 at 0xaaaaaaaaa085c: file ex4.c, line 7.
(gdb) break 19
Breakpoint 2 at 0xaaaaaaaaa08d0: file ex4.c, line 19.
(gdb) set detach-on-fork off
(gdb) run
Breakpoint 1, main () at ex4.c:7
    char *p = mmap(NULL, 4096, PROT_READ | PROT_WRITE,
(gdb) next
    strcpy(p, "Hola món");
(gdb) print p
$1 = 0xffffff7ff6000 ""
(gdb) next
     pid_t pid = fork();
(gdb) print p
$2 = 0xffffff7ff6000 "Hola món"
(gdb) next
[New inferior 2 (process 46848)]
    if (pid == 0) {
(gdb) inferior 2
(gdb) continue
Continuing.
Thread 2.1 "ex4" hit Breakpoint 2.2, main () at ex4.c:19
     return 0;
(gdb) print p
$3 = 0xffffff7ff6000 "hola món"
(gdb) continue
Continuing.
[Inferior 2 (process 46848) exited normally]
(gdb) print p
No symbol "p" in current context.
(gdb) inferior 1
```

```
33      if (pid == 0) {
34  (gdb) continue
35  Continuing.
36  Hola món
37  Thread 1.1 "ex4" hit Breakpoint 2.1, main () at ex4.c:19
38      return 0;
39  (gdb) print p
40  $4 = 0xfffff7ff6000 "Hola món"
41  (gdb) continue
42  Continuing.
43  [Inferior 1 (process 46847) exited normally]
```

- Explica si es coherent aquesta sortida obtinguda amb el debugger `gdb` i per què. [**0.4 punts**]

**Solution:**
La sortida obtinguda amb el debugger `gdb` és coherent perquè el procés fill i el procés pare tenen espais d'adreces separats a causa de l'ús de `MAP_PRIVATE` en la crida a `mmap()`. Quan el procés fill modifica la primera lletra de la cadena a 'h', aquesta modificació no afecta la memòria del procés pare. Per tant, quan el procés pare imprimeix la cadena després d'esperar un segon, encara veu "Hola món" en lloc de "hola món". Tot i això, les adreces virtuals són les mateixes en ambdós processos, ja que `mmap()` va retornar la mateixa adreça per a ambdós processos després del `fork()`. Això és possible perquè cada procés té la seva pròpia còpia privada de la memòria mapejada, i les modificacions fetes per un procés no afecten l'altre.

- Quin mecanisme del kernel permet que `fork()` sigui eficient malgrat copiar tot l'espai d'adreces del pare al fill? [**0.3 punts**]

**Solution:**
El mecanisme del kernel que permet que `fork()` sigui eficient malgrat copiar tot l'espai d'adreces és el *copy-on-write* (COW). Amb COW, el nucli no fa una còpia immediata de les pàgines de memòria del procés pare quan es crea el procés fill. En canvi, ambdós processos comparteixen les mateixes pàgines de memòria fins que un dels processos intenta escriure en una pàgina. En aquest moment, el nucli crea una còpia de la pàgina per al procés que està escrivint, permetent així una gestió eficient de la memòria.

- Què canviaria si `mmap()` s'hagués fet amb **MAP_SHARED** en lloc de **MAP_PRIVATE**? [**0.3 punts**]

**Solution:**
Si `mmap()` s'hagués fet amb **MAP_SHARED** en lloc de **MAP_PRIVATE**, les modificacions realitzades pel procés fill haurien estat visibles per al procés pare. Això significa que quan el procés fill canviés la primera lletra de la cadena a 'h', aquesta modificació es reflectiria en la memòria compartida, i quan el procés pare imprimís la cadena després d'esperar un segon, veuria "hola món" en lloc de "Hola món".