Grau en Enginyeria Informàtica (GEI)

Sistemes Operatius

2024-2025 (1er Semestre)

Examen Parcial Novembre 2024

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	
Punts	2	2	2	2	2	10
Correcció						

1: Part A: 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.

V	F	
□	■	… Si un procés intenta obrir i llegir un fitxer el qual no tenim permis d'accés, la crida **read** fallarà amb error *EACCES* (Permission denied).
■	□	… La dualitat permet al kernel accedir a la memòria dels processos d'usuari, però impedeix als processos d'usuari accedir a la memòria del kernel.
□	■	… La senyal SIGTERM es pot generar amb **Ctrl+C** i permet finalitzar un procés de forma controlada.
□	■	… La crida a sistema `execve()` permet crear un nou procés.
□	■	… Quan un procés reserva memòria amb `malloc` però finalitza sense alliberar-la, la memòria reservada romandrà inaccessible fins que el sistema sigui reiniciat.
□	■	… Qualsevol procés a un sistema Linux és fill d'un procés pare.
□	■	… Quan la crida `fork()` retorna un número negatiu, l'error s'ha de tractar al procés pare i fill.
■	□	… L'arquitectura monolítica integra totes les funcionalitats de sistema, inclosos serveis i controladors, en un espai d'adreçament compartit.
□	■	… Els sistemes de microkernel ofereixen generalment un rendiment més elevat que els sistemes monolítics perquè distribueixen els serveis en espai d'usuari.
□	■	… Les excepcions es poden manejar mitjançant crides a sistema en el codi d'usuari, permetent així als processos interceptar errors com la divisió per zero.

****valor 0.25****

Aquesta part no requereix justificació, però si consideres que has de justificar alguna resposta, fes-ho a continuació.

Solution:

- **Fals**: La crida `read` fallarà amb error `EACCES` si el fitxer no té permisos de lectura, però no si el procés no té permisos d'accés.
- **Cert**: La dualitat permet al kernel accedir a la memòria dels processos d'usuari, però impedeix als processos d'usuari accedir a la memòria del kernel.
- **Fals**: La senyal SIGTERM es pot generar amb `Ctrl+C` però no permet finalitzar un procés de forma controlada, sinó que envia una senyal de terminació al procés.
- **Fals**: La crida a sistema `execve()` no crea un nou procés, sinó que substitueix la imatge del procés actual per una nova imatge.
- **Fals**: La memòria reservada amb `malloc` romandrà inaccessible fins que sigui alliberada, però no cal reiniciar el sistema.
- **Fals**: No tots els processos són fills d'un procés pare, ja que hi ha processos orfes que no tenen un pare. El proces init o systemd sol ser el pare de tots els processos però ell mateix no té pare.
- **Fals**: Quan la crida `fork()` retorna un número negatiu, l'error s'ha de tractar al procés pare, ja que significa que la crida ha fallat.
- **Cert**: L'arquitectura monolítica integra totes les funcionalitats de sistema, inclosos serveis i controladors, en un espai d'adreçament compartit.
- **Fals**: Els sistemes de microkernel ofereixen generalment un rendiment més baix que els sistemes monolítics perquè distribueixen els serveis en espai d'usuari.
- **Fals**: Les excepcions es poden manejar mitjançant crides a sistema en el codi d'usuari, però no permeten interceptar errors com la divisió per zero, ja que aquest tipus d'errors són gestionats pel sistema operatiu.

2: Part B: Preguntes curtes — 2 Punts

- Justifica per que les `pipes` són un mecanisme de comunicació més ràpid que els fitxers. **[0,5 punts]**

Solution:
Les pipes són un mecanisme de comunicació més ràpid que els fitxers perquè són implementades en memòria, mentre que els fitxers són implementats en disc. Això significa que les pipes no requereixen accés a disc, el que redueix la latència i el temps d'accés. A més, les pipes són més eficients per a la comunicació entre processos ja que permeten la comunicació directa entre dos processos sense necessitat d'escriure i llegir a disc, com en el cas dels fitxers.

- El procés A, i el procés B tenen un descriptor de fitxer obert igual a 10. És possible que aquests descriptors de fitxer corresponguin a fitxers diferents? Explica les condicions en què això podria ocórrer, o explica per què això és impossible. **[0,5 punts]**

Solution:
Sí, és possible que dos descriptors de fitxer oberts en dos processos diferents corresponguin a fitxers diferents. Això es deu al fet que els descriptors de fitxer són únics per a cada procés, de manera que el mateix número de descriptor de fitxer pot correspondre a fitxers diferents en dos processos diferents. Això és possible perquè els descriptors de fitxer són locals a cada procés i no es comparteixen entre processos.

- Analitza el següent codi i justifica quina serà la sortida del programa. **[0,5 punts]**

```
1  pid_t pid;
2  int c = 2;
3
4  void h(int sig){
5      c=c-1;
6      printf("%d",c); fflush(stdout);
7      exit(0);
8  }
9
10 int main(){
11     signal(SIGUSR1,h);
12     printf("%d",c); fflush(stdout);
13     if (( pid =fork()) == 0){
14         while(1);
15     }
16     kill(pid,SIGUSR1);
17     waitpid(-1,NULL,0);
18     c=c+1;
19     printf("%d",c); fflush(stdout);
20     exit(0);
21 }
```

Solution:
El programa crea un procés fill que entra en un bucle infinit. El procés pare envia un senyal SIGUSR1 al procés fill, que fa que el procés fill executi la rutina de tractament de senyals `h()`. Aquesta rutina decrementa el valor de `c` en 1 i finalitza el procés fill. A continuació, el procés pare incrementa el valor de `c` en 1 i finalitza. La sortida del programa serà "213".

- És possible dissenyar un sistema operatiu que mai desactivi les interrupcions? En cas contrari, en quines situacions és necessari desactivar-les i per què? **[0,5 punts]**

Solution:
No, no és possible. En un sistema operatiu, cal desactivar les interrupcions en situacions en què el nucli està manipulant estructures de dades crítiques o gestionant altres interrupcions. Això evita condicions de competència i garanteix la coherència en l'execució del sistema.

3: Part C: Preguntes de C — 2 Punts

- Explica breument què fa el codi i comenta quin serà el resultat del codi següent en C, els problemes que pot tenir, assumint que l'usuari introdueix el nom **Pere** i l'edat **25** i en la segona iteració introdueix el nom **Anna** i l'edat **30**. **[0,5 punts]**

```c
struct usuari {
    char *nom;
    int edat;
};

usuari* usuaris[2];
char* name;
int edat;

for (int i = 0; i < 2; i++) {
    usuaris[i] = (usuari*) malloc(sizeof(usuari));
    printf("Nom: "); scanf("%s", name);
    usuaris[i]->nom = name;
    printf("Edat: "); scanf("%d", &edat);
    usuaris[i]->edat = edat;
}

for (int i = 0; i < 2; i++) {
    printf("Usuari %d: %s, %d anys\n", i, usuaris[i]->nom, usuaris[i]->edat);
}
```

Solution:
En aquet codi es declara una estructura `usuari` amb dos camps, un per al nom i un altre per a l'edat. Després es declara un array de punters a `usuari` de dos posicions. Seguidament es declaren dues variables, `name` i `edat`, per a llegir les dades de l'usuari. Es fa un bucle de dues iteracions on es reserva memòria per a un `usuari`, es llegeixen les dades de l'usuari i es guarden a l'array d'usuaris. Finalment es recorre l'array d'usuaris i es mostren les dades de cada usuari. El problema és que el nom de l'usuari es guarda en la mateixa variable `name` per a tots dos usuaris, de manera que el nom de l'usuari 1 es sobreescriu amb el nom de l'usuari 2. Això fa que en la segona iteració, el nom de l'usuari 1 sigui "Anna" en lloc de "Pere". Per tant, el resultat serà: `Usuari 0: Anna, 25 anys` i `Usuari 1: Anna, 30 anys`. A més a més, el programa té una fuita de memòria ja que no es fa `free` de la memòria reservada amb `malloc`.

- Analitza el codi següent i explica què fa. Comenta també els possibles problemes que podrien sorgir durant la seva execució. **[0,5 punts]**

```c
int *array = (int *)malloc(5 * sizeof(int));
for (int i = 0; i < 5; i++) {
    printf("array[%d] = %d\n", i, array[i]);
}
array[5] = 42;
free(array);
```

Solution:
Aquest codi reserva memòria per a un array de 5 enters. Després intenta imprimir els valors de l'array, però com que no s'han inicialitzat, els valors són indeterminats, per tant, el resultat serà un SEGMENATION FAULT. A continuació, intenta assignar el valor 42 a la posició 5 de l'array, que està fora dels límits de l'array i provocarà un heap overflow. Finalment, allibera la memòria reservada amb `free`.

- Analitza el codi C següent. Explica breument què fa i comenta quin serà el resultat del codi assumint que l'usuari introdueix per teclat (**stdin**) els valors 2 i 1. Identifica també els problemes que poden sorgir en executar aquest codi. **[0,5 punts]**

```c
enum options { HAPPYCODING, HAPPYHACKING, HAPPYEXAM };
const char *options_names[] = { "Happy Coding!", "Happy Hacking!", "Happy Exam!" };

void imprimir(int i){
    printf("%s\n", options_names[i]);
}

int main(){
    char buf[2];
    int nbytes;
    for (int i = 0; i < 2; i++){
        nbytes = read(0, buf, sizeof(buf)-1);
        imprimir(atoi(buf));
    }
}
```

Solution:
Aquest programa llegeix dues entrades numèriques de l'usuari a través de l'estàndard d'entrada (**stdin**), les converteix a enters mitjançant `atoi()`, i mostra per pantalla (`stdout`) el missatge corresponent de l'array **options_names**. La sortida esperada depèn dels valors llegits i la conversió a enter.
En executar el codi amb les entrades 2 i 1, sorgeix un problema: només es processa el primer número **(2)**, ja que cada entrada inclou un salt de línia `\n`. Això implica que en la primera iteració es llegeix `2\n` (el caràcter 2), de manera que la sortida serà "Happy Exam!". En la segona iteració, **buf** conté únicament `\n` (el salt de línia restant al buffer), i com que `\n` no es converteix en un número vàlid, `atoi()` retorna 0. Això provoca que la sortida sigui "Happy Coding!".

- L'ús de read i buf[2] només permet llegir un caràcter cada vegada, però no descarta el salt de línia `\n`, cosa que impedeix que cada iteració processi correctament un número. S'hauria de llegir en aquest cas un caràcter més per a descartar el salt de línia.
- No es comprova si el valor introduït és vàlid (dins dels límits de l'array **options_names**), el que podria provocar errors d'execució si l'usuari introdueix un valor fora de l'interval 0-2.
- Tampoc es contrala la longitud de la cadena llegida, el que podria provocar un desbordament de buffer si l'usuari introdueix una cadena molt llarga.

- Analitza el codi següent i explica què fa, si presenta un comportament determinista i, en cas contrari, com es podria solucionar. **[0,5 punts]**

```c
void my_func(const char* src, char* dest) {
    char* buffer = malloc(1000 * sizeof(char));
    int fd1 = open(src, O_RDONLY);
    lseek(fd1, 500, SEEK_SET);
    read(fd1, buffer, 150*sizeof(char));
    int fd2 = open(dest, O_WRONLY);
    lseek(fd2, 200, SEEK_SET);
    write(fd2, buffer, 100*sizeof(char));
    close(fd1);
    close(fd2);
}
```

Solution:
Aquest codi llegeix 150 caràcters del fitxer d'origen a partir de la posició 500 i escriu 100 caràcters al fitxer de destí a partir de la posició 200. El comportament no és determinista perquè no es comprova si les crides a `read` i `write` s'han completat correctament. Per solucionar-ho, es podria comprovar el valor de retorn de les crides a `read` i `write` per assegurar-se que s'han llegit i escrit la quantitat desitjada de caràcters. Hauriam de tenir un bucle `while` per a assegurar-nos que llegim i escrivim la quantitat desitjada de caràcters.

4: Cas d'estudi 1 — 2 Punts

Analitza el codi següent i explica quin serà el seu comportament, responent de forma justificada a les qüestions següents. *Assumeix que cap esdeveniment asíncron afectarà la seva execució i que el procés s'executarà amb normalitat.*

```
1    int fd = mkfifo("myfifo", 0666);
2    char msg[200];
3    for(int i = 0; i < N; i++){
4        fork();
5        sprintf(msg, "Process %d\n", getpid());
6        write(fd, msg, strlen(msg));
7        wait(NULL);
8    }
9    exit(0);
```

- Què fa el codi? **[0,3 punts]**

Solution:
Aquest codi crea un fitxer FIFO anomenat *myfifo* amb permisos de lectura i escriptura (0666). A continuació, en un bucle que s'executa **N** vegades, es crida a `fork()` per crear un nou procés fill en cada iteració. Cada procés fill escriu un missatge al FIFO que conté el seu identificador de procés (PID) mitjançant la funció `write()`. Al final de cada iteració, el procés pare espera a que el seu fill acabi amb la funció `wait()`.

- Quin resultat produirà el programa? *Comenta quin serà el resultat d'executar el programa i indica si és determinista o no.* **[0,3 punts]**

Solution:
La sortida esperada del codi serà una sèrie de missatges escrits al fitxer FIFO *myfifo*, amb cada missatge indicant el PID del procés que l'ha escrit. L'ordre dels missatges al FIFO dependrà de l'ordre en què els processos fills escriguin al FIFO, així com de la programació del sistema operatiu.

- Quants processos es generen? *Argumenta en funció de N.* **[0,4 punts]**

Solution:
El nombre de processos generats serà 2^N. Això es deu al fet que cada crida a `fork()` crea un nou procés fill. En cada iteració del bucle, es criden `fork()` i, per tant, s'afegeixen processos fills de manera exponencial. Si N = 3, el total de processos generats serà 2^3 = 8 (1 procés pare + 7 fills).

- Quines són les principals crides a sistema involucrades? **[0,25 punts]**

Solution:
Les principals crides al sistema involucrades en aquest codi són: `mkfifo()`, `fork()`, `write()` i `wait()`. `mkfifo()` es fa servir per crear un fitxer FIFO anomenat *myfifo* amb permisos de lectura i escriptura. `fork()` es crida per crear un nou procés fill en cada iteració del bucle. `write()` s'utilitza per escriure un missatge al FIFO amb el PID del procés fill. `wait()` s'utilitza per esperar a que el procés fill finalitzi abans de continuar amb el procés pare.

- Indica una comanda que permeti visualitzar totes les crides a sistema. **[0,25 punts]**

Solution:
Per visualitzar totes les crides a sistema del programa, es pot utilitzar la comanda `strace` amb l'executable del programa. Per exemple, si l'executable es diu `cas`, la comanda seria `strace ./cas`.

- La seva execució pot causar problemes? Per què? *En cas afirmatiu, com es podrien solucionar?* **[0,5 punts]**

Solution:
Sí, la seva execució pot causar problemes, principalment bloqueig i consum excessiu de recursos. Quan `fork()` és cridat múltiples vegades, es pot produir un nombre molt alt de processos, resultant en un bloqueig del sistema operatiu si s'excedeixen els límits de processos. A més, el procés pare espera a cada fill a finalitzar amb `wait()`, que pot causar un bloqueig si els processos fills no s'executen correctament o si el sistema és lent a crear o gestionar els processos.
Per evitar problemes de bloqueig i consum excessiu de recursos, es podria limitar el nombre de processos fills creats, per exemple, mitjançant una comprovació de la variable `i` abans de cridar `fork()`. Això permetria controlar el nombre de processos fills generats i evitar un nombre excessiu de processos.
Els missatges escrits al FIFO es poden llegir utilitzant la funció `read()`. No obstant això, és important tenir en compte que, si no hi ha un procés llegint del FIFO, la funció `write()` pot bloquejar-se perquè el FIFO està ple (en el cas de FIFO no de lectura). Això pot causar un bloqueig del procés fill si el procés pare no llegeix els missatges.

5: Cas d'estudi 2

2 Punts

Analitza el codi següent i respon a les preguntes següents. Assumeix que el codi *s'executarà sense errors i que no hi haurà cap esdeveniment asíncron que afecti la seva execució.* **Si detectes errors són intencionats**.

```
1    int pipe1[2], pipe2[2]; pipe(pipe1); pipe(pipe2);
2    if (fork() == 0){
3        close(pipe1[0]); close(pipe2[0]); close(pipe2[1]);
4        dup2(pipe1[1], 1);
5        int fd = open("input.txt", 0); dup2(fd, 0);
6        execlp("cat", "cat", NULL);
7    }
8    if (fork() == 0){
9        close(pipe1[1]); close(pipe2[0]);
10       dup2(pipe1[0], 0); dup2(pipe2[1], 1);
11       execlp("grep", "grep", "hello", NULL);
12   }
13   if (fork() == 0){
14       close(pipe1[0]); close(pipe1[1]); close(pipe2[1]);
15       dup2(pipe2[0], 0);
16       int fd = open("output.txt", 1); dup2(fd, 1);
17       execl("wc", "wc", "-l", NULL);
18
19   }
20   close(pipe2[0]);close(pipe2[1]);
21   for (int i = 0; i < 3; i++){wait(NULL);} return 0;
```

- Quina comanda es vol implementar amb aquest codi? **[0,5 punts]**

Solution:
Aquest codi implementa la comanda `cat < input.txt | grep "hello" | wc -l > output.txt`. Els passos principals són llegir el contingut del fitxer `input.txt` amb `cat`, filtrar les línies que contenen la paraula "hello" amb `grep`, i comptar el nombre de línies amb `wc -l`, i finalment escriure el resultat al fitxer `output.txt`.

- Analitza el codi i comenta els problemes que poden sorgir en l'execució del programa. **[0,5 punts]**

Solution:
Un dels problemes principals és que no es tanquen totes les pipes correctament en cada procés fill. Això pot provocar que les pipes no es tanquin correctament i que els processos no finalitzin correctament. Per solucionar aquest problema, s'haurien de tancar totes les pipes que no s'utilitzen en cada procés fill.
El procés pare no tanca mai la pipe1[1], la qual farà que el procés lector de la pipe1 no finalitzi mai. Per solucionar aquest problema, s'hauria de tancar la pipe1[1] al procés pare després de crear els processos fills.
El execl necessita el camí complet de l'executable, per tant, s'hauria de canviar `execlp("cat", "cat", NULL);` per `execlp("/bin/cat", "cat", NULL);`, `execlp("grep", "grep", "hello", NULL);` per `execlp("/bin/grep", "grep", "hello", NULL);` i `execl("wc", "wc", "-l", NULL);` per `execl("/usr/bin/wc", "wc", "-l", NULL);`.
Els processos haurien de comprovar que les crides a sistema retorne un valor vàlid i gestionar els errors adequadament. Per exemple, es podria comprovar el retorn de `pipe()`, `fork()`, `dup2()`, `open()`, `execlp()`, `execl()`, `close()` i `wait()` per assegurar-se que les crides s'han completat correctament.

- Quina comanda ens permetria obtenir la taula de fitxers oberts per cada procés? **[0,25 punts]**

Solution:
`lsof -p <pid>` o `ls -l /proc/<pid>/fd/` ens permet obtenir la taula de fitxers oberts per un procés amb un PID específic. Podem utilitzar aquesta comanda per obtenir la taula de fitxers oberts per cada procés creat en la implementació de la comanda `cat < input.txt | grep "hello" | wc -l > output.txt`.

- Analitza el cas en què es generen dos processos fills i el procés pare fa un dels recobriments. Seria aquesta solució igualment vàlida o podria presentar problemes de funcionament? *Justifica la teva resposta.* **[0,75 punts]**

Solution:
La solució amb 2 processos fills i una tasca addicional en el pare podria ser vàlida. Únicament tindriam problemes de zombies ja que el pare fa el recobriment i no esperarà als fills.