

— 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	3	2	2	2	1	10
Correcció						

1: Part A: Teoria

3 Punts

Indica si l'afirmació es **certa** o **falsa** i raona la resposta:

- Analitza el següent codi i indica si es correcte o incorrecte:

```
1 struct node {  
2     int data;  
3     struct node *next;  
4 };
```

Solution:

Cert. Una *struct* pot tenir un apuntador a ella mateixa. Ja que structs són tipus de dades, es poden fer referència a elles mateixes. En aquest cas, un exemple de llista enllaçada.

- Analitza el següent codi i indica si el valor que s'imprimirà per pantalla serà el mateix per a tots els processos:

```
1 int a = 10;  
2 fork();  
3 printf("Direcció de a: %p\n", &a);
```

Solution:

Cert. Cada procés té la seva pròpia memòria, per tant, es possible que el procés pare i fill imprimeixin la mateixa direcció de memòria. Aquesta és la direcció virtual de la variable que a través del mecanisme de traducció d'adreces es mapeja a una adreça física diferent per a cada procés.

- Quan una funció retorna un punter a una variable local de la mateixa funció, el punter retornat serà vàlid fora de la funció. Per exemple:

```
1 int *func() {  
2     int a = 10;  
3     return &a;  
4 }  
5  
6 int main() {  
7     int *p = func();  
8     printf("%d\n", *p);  
9     return 0;  
10 }
```

Solution:

Fals. La variable *a* és local a la funció *func()*, per tant, quan la funció retorna, la variable *a* deixa de ser vàlida i el punter retornat apuntarà a una posició de memòria que ja no conté el valor original.

- Si un punter `int *p` apunta a una regió de memòria assignada dinàmicament amb `malloc()`, i al acabar utilitzem `free()`. S'elimina el valor de `p` i també l'espai de memòria que apunta.

Solution:

Fals. La crida a `free()` allibera l'espai de memòria assignat per `malloc()`, però no elimina el valor de `p`. `p` seguirà apuntant a la mateixa direcció de memòria, però aquesta ja no pertany al programa. Es a dir `free()` allibera la memòria que apunta `p` però no modifica `p` mateix. El punter `p` continua apuntant a l'antiga adreça de memòria fins que es reinicialitza.

- Una llibreria escrita en l'espai d'usuari implementa les crides a sistema realitzant una instrucció explícita de canvi a mode nucli per executar la crida de sistema, i després es dirigeix a una subrutina apropiada dins del codi del nucli.

Solution:

Fals. Les crides a sistema no permeten que una llibreria d'usuari canviï directament a mode nucli, ja que això comprometria la seguretat del sistema. En lloc d'això, les crides a sistema són controlades a través de trampes (traps) o instruccions `syscall`, que realitzen el canvi segur a mode nucli per a gestionar la crida, permetent al sistema operatiu controlar i restringir les operacions privilegiades.

- Un procés pot tenir dos descriptors de fitxer diferents (a la taula de descriptors de fitxer del procés) que apunten a la mateixa estructura de descripció de fitxer oberta al nucli.

Solution:

Cert. Per exemple, la crida a sistema `dup()` crea un nou descriptor de fitxer que apunta a la mateixa estructura de descripció de fitxer oberta.

- Un Microkernel pot millorar la resiliència d'un sistema contra errors en el sistema operatiu.

Solution:

Cert. Un MicroKernel pot millorar la resiliència d'un sistema contra errors en el sistema operatiu. Aïllant components del sistema en els seus propis espais d'adreces, obtenim resiliència contra errors perquè els components defectuosos es veuen impeditos d'interferir amb altres components per l'aïllament de l'espai d'adreces.

- La crida a sistema `pipe()` crea un fitxer al disc i retorna un descriptor de fitxer associat a la lectura i un altre a l'escriptura. Raonar la resposta.

Solution:

Fals. Els pipes s'implementen en un buffer del kernel a la memòria, no es creen fitxers al disc.

- Qualsevol procés en un sistema linux té un procés pare.

Solution:

Fals. El procés `init/systemd` és el procés pare de tots els processos en un sistema Linux. La resta de processos tenen un procés pare que els ha creat.

- Assumeix la següent instrucció: `write(fd, "Hola Mon", 9)`. Si el descriptor de fitxer `fd` apunta a un fitxer buit que s'ha obert amb permisos d'escriptura, es garanteix que el fitxer ha estat modificat quan la crida a `write` retorna.

Solution:

Fals. La crida `write` no garanteix que s'escriguin tots els bytes al fitxer. La crida `write` pot ser bufferitzada pel kernel i el fitxer pot no ser modificat quan la crida retorna.

- Analitza el seüent codi i indica si es pot produir un **heap overflow**:

```
1 int main() {  
2     int *p = (int *)malloc(10 * sizeof(int));  
3     p[10] = 10;  
4     free(p);  
5     return 0;  
6 }
```

Solution:

Cert. L'accés a `p[10]` està fora dels límits de l'espai de memòria reservat per `malloc()`, per tant, es produeix un desbordament de memòria.

- La crida a sistema `exit()` permet desasignar un descriptor de fitxer obert pel procés.

Solution:

Cert. La crida `exit()` tanca tots els descriptors de fitxer oberts pel procés de forma indirecta ja que el sistema operatiu allibera tots els recursos associats al procés.

2: Part B: Teoria Kernel i Cridres a Sistema

2 Punts

1. Anomena el mecanisme d'accés al kernel en les situacions següents: **(0,75 punt)**

- El planificador de processos decideix que un altre procés s'executi ja que el temps de CPU ha acabat.

Solution:

Interrupció de rellotge. El rellotge del sistema genera una interrupció que fa que el processador canviï de context.

- Un procés intenta accedir a una regió de memòria prohibida.

Solution:

Excepció. Una excepció és una condició inesperada causada pel programa de l'usuari. Això fa que el procés s'aturi i entri al nucli en el manegador d'excepcions.

- Un programa executa la funció `write()`.

Solution:

Trampa de sistema. La crida a sistema `write()` fa una trucada al kernel per escriure dades a un fitxer.

2. Quina és la importància de tenir una taula de crides a sistema indexada per un número de crida a sistema en lloc de permetre a l'usuari especificar una adreça de funció per ser cridada pel nucli un cop es faci el canvi de context? **(0,75 punts)**

Solution:

Si l'usuari pogués especificar una adreça de funció arbitrària per ser executada en el nucli, podria evitar la comprovació de seguretat.

1. Imagina que el nostre sistema operatiu té la següent crida a sistema. Indica quin és el seu propòsit i quins perills pot comportar: **(0,5 punts)**

```
1 void my_syscall(unsigned long *addr, const char *msg) {  
2   copy_to_user(addr, msg);  
3 }
```

Solution:

La crida a sistema `my_syscall()` copia el contingut de la cadena `msg` a la memòria de l'usuari apuntada per `addr`. Aquesta crida a sistema pot ser perillosa si no es comprova que la memòria de l'usuari és vàlida i que té prou espai per emmagatzemar la cadena `msg`. A més a més, si passem una adreça vàlida d'un altre usuari, aquesta crida podria retornar dades no autoritzades.

3: Part C: Gestió de processos

2 Punts

1. Analitza el codi següent i raona quin serà el seu comportament contestant de forma justificada les qüestions següents. Assumeix que cap esdeveniment asíncron afectarà la seva execució i que el procés s'executarà amb normalitat: **(1 punt)**

- Què fa el codi?
- Quina sortida trindrà?
- Quants processos es generen?
- La seva execució pot causar problemes? Per què? En cas afirmatiu, com es podrien solucionar?
- Quines són les principals crides a sistema involucrades?

```
1 int main(int argc, char *argv[]) {
2     pid_t pid = fork();
3     int status;
4     if (pid == 0) {
5         execvp(argv[1], &argv[1]);
6         while(1);
7     }
8     waitpid(pid, &status, 0);
9     if (WIFEXITED(status)) {
10        printf("El procés %d ha executat %s\n", getpid(), argv[1]);
11    }
12    exit(0);
13 }
```

Solution:

Aquest codi generarà 2 processos: un procés pare i un procés fill. El procés fill executarà el programa indicat pel primer argument passat a la línia de comandes. El procés pare esperarà a la finalització del procés fill i escriurà un missatge a la sortida estàndard indicant que el procés fill ha finalitzat. La sortida del programa serà: El procés <pid> ha executat <programa>. El procés fill pot quedar en execució indefinidament si la crida a `execvp()` falla. Les crides a sistema involucrades són `fork()`, `execvp()`, `waitpid()` i `exit()`. Per evitar que el procés fill quedi en execució indefinidament, es podria afegir un `exit(-1)` enlloc del bucle `while(1);` D'aquesta manera, el procés fill finalitzarà si la crida a `execvp()` falla. I el -1 indicarà que el programa no s'ha executat correctament.

2. Analitza el següent fragment i argumenta quina serà la sortida del programa. Assumeix que cap esdeveniment asíncron afectarà la seva execució i que el procés s'executarà amb normalitat. **(1 punt)**

```
1 int main(){
2     int i,n,p1[2];
3     char c, buf[80];
4     pipe(p1);
5     int a=10, codret, pid;
6     if (fork() == 0) {
7         a = a + 2;
8         for (c='1'; c<='5'; c++) {
9             write(p1[1], &c, 1);
```

```

10     }
11     exit(a);
12 } else {
13     close(p1[1]);
14     while ((n = read(p1[0], buf, sizeof(buf))) > 0) {
15         write(1, buf, n);
16         write(1, "-", 1);
17     }
18     pid = wait (& codret );
19     sprintf (buf, "%d\n", a+WEXITSTATUS ( codret ));
20     write( 1,buf,strlen(buf));
21 }
22 exit(0);
23 }

```

Solution:

El programa crea un pipe `p1` i un procés fill. El procés fill escriu els caràcters de '1' a '5' al pipe `p1` i finalitza amb un codi de sortida `a`. El procés pare llegeix els caràcters del pipe `p1` i els escriu a la sortida estàndard. Després, espera a la finalització del procés fill i escriu a la sortida estàndard el valor de `a` més el codi de sortida del procés fill. En aquest cas no es pot determinar la sortida exacta del programa ja que depèn de l'ordre d'execució dels processos. La sortida més probable és: **12345-22**. Però **12-345-22** també és una sortida vàlida. Si el procés fill únicament fa 2 iteracions del bucle i després executa el pare, el buffer llegirà 2 caràcter i escriurà 12- i després el fill pot continuar i escriurà 345- i finalment el pare escriurà 22. Si el fill acaba abans que el pare comenci a llegir, la sortida serà 12345-22. Per assegurar la sortida 12345-22 s'hauria de moure la crida a `wait()` abans de la crida a `read()`.

4: Cas d'estudi: Comunicació entre processos

2 Punts

Analitza el següent codi i contesta les preguntes següents:

```

1  main(){
2
3      int p[2];
4      int fd;
5      pipe(p);
6
7      switch (fork()){
8          //Fill 1
9          case -1: error();
10         case 0:
11             if (dup2(p[1], 1) < 0) error();
12             fd = open("input.txt", O_RDONLY);
13             if (fd < 0) error();
14             if (dup2(fd, 0) < 0) error();
15             if (close(p[0]) < 0) error();
16             if (close(p[1]) < 0) error();
17             if (close(fd) < 0) error();
18             execlp("grep", "grep", "-R", NULL);
19             exit(0);
20     }

```

```

21
22     switch (fork()){
23         //Fill 2
24         case -1: error();
25         case 0:
26             if (dup2(p[0], 0) < 0) error();
27             fd = open("output.txt", O_WRONLY|O_APPEND|O_CREAT , 0600);
28             if (fd < 0) error();
29             if (dup2(fd, 1) < 0) error();
30             if (close(p[0]) < 0) error();
31             if (close(p[1]) < 0) error();
32             if (close(fd) < 0) error();
33             execlp("nl", "nl", NULL);
34             exit(0);
35     }
36
37     if (close(p[0]) < 0) error();
38     if (close(p[1]) < 0) error();
39     wait(NULL);
40     wait(NULL);
41 }

```

1. Indica quina comanda de bash està simulant el codi anterior.

Solution:

```
1 grep -R < input.txt | nl >> output.txt
```

2. Què passaria si el procés Fill 1 mor abans d'escriure tot el seu resultat a la pipe?

Solution:

El procés **nl** continuarà llegint de la pipe i acabarà detectant final de fitxer llegint de la pipe. **nl** no serà conscient de la mort prematura de **grep** llevat que, per exemple, sàpiga quants caràcters hauria d'haver llegit de la pipe o si del format intern de les dades llegides es pot deduir que manca informació.

3. Què passaria si el procés Fill 2 mor abans que Fill 1 acabi d'escriure tot el resultat a la pipe?

Solution:

Si **nl** mor, quan **grep** intenti escriure a la pipe el SO veurà que està escrivint a una pipe on no hi ha cap procés lector. Com això probablement indica que s'ha produït algun tipus d'error, el SO ho notificarà a **grep** mitjançant el signal **SIG_PIPE**. Si **grep** no ha definit rutina d'atenció a aquest signal, morirà immediatament.

5: Cas d'estudi: Senyals

1 Punt

Analitza el codi següent i contesta les preguntes següents:

```
1  int trets = 0;
2
3  void dispara(int pid){
4      int n = rand()%31;
5      kill(pid, n);
6  }
7
8  void mostra_trets(){
9      printf("[%d] He rebut %d trets\n", getpid() ,trets);
10 }
11
12 void handler(int n){
13     trets++;
14     if(n == SIGKILL){
15         mostra_trets();
16         exit(0);
17     }
18     if(n == SIGUSR1 || n == SIGUSR2){
19         mostra_trets();
20     }
21 }
22
23 int main() {
24
25     for(int i=0; i<32; i++){
26         signal(i, handler);
27     }
28
29     switch(fork()){
30         case -1: error();
31         case 0:
32             while(1){
33                 dispara(getppid());
34                 pause();
35             }
36         default:
37             while(1){
38                 dispara(pid);
39                 pause();
40             }
41     }
42
43 }
```

1. Quin és l'objectiu del codi?
2. Aquest codi pot acabar en un bucle infinit? Per què?
3. Té algun problema? Per què?

Solution:

En aquest codi el procés pare i el procés fill es disparen senyals aleatòries. El procés pare i el procés fill tenen un handler que compta el nombre de senyals rebuts. En el cas que el procés pare rep un SIGKILL, el procés finalitzarà ja que SIGKILL no es pot capturar. Si rep un SIGUSR1 o SIGUSR2, mostrarà el nombre de senyals rebuts i continuarà. El procés fill fa el mateix. Un dels codis (pare o fill) no pot acabar en un bucle infinit ja que tard o d'hora rebrà un SIGKILL i finalitzarà. Ara bé, l'altre procés pot acabar en un bucle infinit ja que no té cap condició de sortida. Per tant, el codi també pot generar un estat zombie ja que el pare no espera a la finalització del fill.