UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

DOCUMENTACIÓ DE LABORATORI

SISTEMES OPERATIUS
GRAU EN ENGINYERIA INFORMÀTICA
Curs 2025-2026

Autors:
Professors de Sistemes Operatius del
Departament d'Arquitectura de Computadors
(UPC- BarcelonaTech)

Aquest document conté les sessions a realitzar durant les classes de laboratori.
Les sessions inclouen el treball previ i el treball a realitzar durant les sessions.

Llicència

Índex de sessions

Sessió 1. L'intèrpret de comandes: Shell

Preparació prèvia

Objectius

L'objectiu d'aquesta sessió és aprendre a desenvolupar-se en l'entorn de treball dels laboratoris. Veurem que algunes operacions es poden fer tant amb comandes interactives com utilitzant el gestor de finestres. Ens centrarem en la pràctica d'algunes comandes bàsiques i en la utilització del manual online (man) que trobareu en totes les màquines Linux.

Habilitats

- Ser capaç d'utilitzar les pàgines de man.
- Ser capaç d'utilitzar comandes bàsiques de sistema per modificar/navegar pel sistema de fitxers: cd, ls, mkdir, cp, rm, rmdir, mv, df, ln, namei, readlink, stat .
- Conèixer els directoris especials "." i "..".
- Ser capaç d'utilitzar comandes bàsiques de sistema i programes de sistema per accedir a fitxers: less, cat, grep, gedit (o un altre editor).
- Ser capaç de modificar els permisos d' accés d' un fitxer.
- Ser capaç de consultar/modificar/definir una variable d'entorn.
- Ser capaç d'utilitzar alguns caràcters especials de la Shell (intèrpret de comandes):
    - & per executar un programa en segon pla (executar en background).
    - > per guardar la sortida d'un programa (redireccionar la sortida).

Coneixements previs

En aquesta sessió no es requereixen coneixements previs.

Guia per al treball previ

Accés al sistema

En SO, podem fer servir dos tipus de laboratori de la facultat. Un d'ells, l'anomenarem "laboratori de sistemes" en aquesta documentació, i té instal·lat Ubuntu 20.04 LTS 64 bits a les màquines. En aquest cas, aquesta instal·lació té diversos usuaris creats perquè es puguin fer proves que involucrin diversos usuaris. Els usernames dels usuaris són:"alumne", "so1", "so2", "so3", "so4" i "so5". El password és "sistemes" per a tots ells. L'altre tipus de laboratori, l'anomenarem "laboratori general", i té instal·lada OpenSuse 15.4 64 bits. En aquestes màquines el teu usuari i el teu password són les teves credencials UPC.

Per començar, executarem el que anomenem una *Shell* o un intèrpret de comandes. Una *Shell* és un programa que el S.O. ens ofereix per poder treballar en un mode de text interactiu. Aquest entorn pot semblar menys intuïtiu que un entorn gràfic, però és molt senzill i potent.

Hi ha diversos intèrprets de comandes, al laboratori utilitzareu Bash (GNU-Bourne Again Shell), però en general ens hi referirem com Shell. La majoria de les coses que explicarem en aquesta sessió es poden consultar al manual de Bash (executant la comanda **man bash**).

Per executar una Shell n'hi ha prou amb executar l'aplicació "Terminal" o "Konsole". Amb aquestes aplicacions, s'obre una nova finestra (similar a la de la imatge) on s'executa una nova Shell.

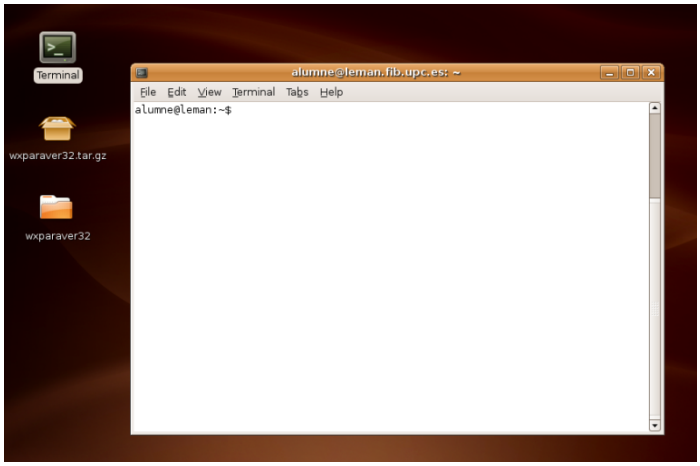

figura 1 Finestra de la shell

El primer que hauries de fer és comprovar que la Shell és Bash. Per a això pots executar la comanda:

```
#echo $SHELL
```

I et mostrarà l'executable que està en marxa. Si no és el Bash, posa'l en execució:

```
#bash
```

El text que apareix a l'esquerra juntament amb el cursor que parpadeja és el que es coneix com a *prompt* i serveix per indicar que la Shell està llesta per rebre noves ordres o comandes. Nota: en la documentació de laboratori utilitzarem el caràcter # per representar el prompt i indicar que el que ve a continuació és una línia de comandes (per provar la línia NO HEU D'ESCRIURE #, només la comanda que apareix a continuació).

El codi de la Shell es podria resumir de la següent manera:

```
while(1){
      comanda =llegir_comanda();
      executar_comanda(comandamen);
}
```

Existeixen dos tipus de comandes: comandes externes i comandes internes. Les comandes **externes** són qualsevol programa instal·lat a la màquina i les comandes **internes** són funcions implementades per l'intèrpret de comandes (cada intèrpret implementa les seves, n'hi ha de comunes a tots ells i n'hi ha de pròpies).

Comandes per obtenir ajuda

A Linux, hi ha dos comandes que podem executar de forma local a la màquina per obtenir ajuda interactiva: la comanda **man**, que ens ofereix ajuda sobre les comandes externes (com a part de la instal·lació, s'instal·len també les pàgines del manual que podrem consultar a través del man), i la comanda **help**, que ens ofereix ajuda sobre les comandes internes.

- **Llegeix la guia sobre com utilitzar el man** de Linux que tens al final d'aquesta secció ("Utilització del manual"). A continuació, **consulta el man** (`man nombre_comanda`) de les següents comandes. En concret, per a cada comanda has de llegir i entendre perfectament: la SYNOPSIS, la DESCRIPTION i les opcions que apareixen a la columna "Paràmetres" de la taula.

Per llegir amb el manresà	Descripció bàsica	Paràmetres
`man`	Accedeix als manuals on-line	
`ls`	Mostra el contingut del directori	`-l, -a`
`àlies`	Defineix un nom alternatiu a una comanda	
`mkdir`	Crea un directori	
`rmdir`	Elimina un directori buit	
`mv`	Canvia el nom d' un fitxer o el mou a un altre directori	`-i`
`cp`	Còpia fitxers i directoris	`-i`
`rm`	Esborra fitxers o directoris	`-i`
`echo`	Visualitza un text (pot ser una variable d'entorn)	
`less`	Mostra fitxers en un format apte per a un terminal.	
`cat`	Concatena fitxers i els mostra en la seva sortida estàndard	
`grep`	Busca text (o patrons de text) en fitxers	
`gedit`	Editor de text per a GNOME	
`env`	Executa una comanda en un entorn modificat, si no se li passa comanda, mostra l' entorn	
`chmod`	Modifica els permisos d' accés a un fitxer.	
`which`	Localitza una comanda	
`df`	Retorna informació sobre el sistema d' arxius	`-h -T -l -i`
`ln`	Crea enllaços entre fitxers	`-s`
`namei`	Segueix una ruta fins a trovar un punt final	
`readlink`	Imprimeix enllaços simbòlics resolts o noms d' arxiu canònics	
`stat`	Mostra l' estat del fitxer o del sistema de fitxers	`-f`
`mount`	Monta un sistema de fitxers	

- **Utilitzeu la comanda help** per consultar l'ajut sobre les següents comandes internes:

Per consultar amb l'help	Descripció bàsica
`help`	Ofereix informació sobre comandes internes de la Shell
`export`	Defineix una variable d' entorn
`cd`	Canvia el directori (carpeta) actual
`àlies`	Defineix un nom alternatiu a una comanda

- Accedeix a la pàgina del man per al bash (executant la comanda man bash) i busca el significat de les variables d'entorn PATH, HOME i PWD (nota: el caràcter "/" serveix per buscar patrons a les pàgines del man. Utilitzeu-lo per trobar directament la descripció d'aquestes variables).

Utilització del manual

Saber utilitzar el manual és bàsic ja que, tot i que durant el curs us explicarem explícitament algunes comandes i opcions, la resta (inclòs crides a sistema) hauràs de buscar-lo tu mateix al manual. El mateix man és auto contingut en aquest sentit, ja que per veure totes les seves opcions pots executar:

```
#man man
```

La informació del manual està organitzada en seccions. La secció 2, per exemple, és la de crides a sistema. Les seccions que podem trobar són:

1. comandes
2. crides a sistema
3. crides a llibreries d' usuari o del llenguatge
4. etc.

La informació proporcionada en executar el man és el que es coneix com a "pàgina de man". Una "pàgina" sol ser el nom d'una comanda, crida a sistema o crida a funció. Totes les pàgines de man segueixen un format semblant, organitzat en una sèrie de parts. En la figura 2 tens un exemple de la sortida del man per a la comanda ls (hem esborrat alguna línia perquè es vegin les principals parts). A la primera part pots trobar tant el nom de la comanda com la descripció i un esquema (SYNOPSIS) de la seva utilització. En aquesta part pots observar si la comanda accepta opcions, si necessita algun paràmetre fix o opcional, etc.

LS(1) User Commands LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [OPTION]... [FILE]...
DESCRIPTION
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuSUX nor --sort.
Mandatory arguments to long options are mandatory for short options to.
-a, --all
do not ignore entries starting with .
SEE ALSO
The full documentation for ls is maintained as a Texinfo manual. If the info and ls programs are properly installed at your site, the command
`info ls`
should give you access to the complete manual.
ls 5.93 November 2005 LS(1)

figura 2 man ls (simplificat)

La següent part és la descripció (DESCRIPTION) de la comanda. Aquesta part inclou una descripció més detallada de la seva utilització i la llista d'opcions que suporta. Depenent de la instal·lació de les pàgines de man també pots trobar aquí els codis de finalització de la comanda (EXIT STATUS). Finalment hi sol haver una sèrie de parts que inclouen els autors de l'ajuda, la forma de reportar errors, exemples i comandes relacionats (SEE ALSO).

En la figura 3 tens el resultat d'executar "man 2 write", que correspon amb la crida a sistema write. El número que posem abans de la pàgina és la secció en la qual volem buscar i que incloem en aquest cas ja que existeix més d'una pàgina amb el nom write en altres seccions. En aquest cas la SYNOPSIS inclou els fitxers que han de ser inclosos en el programa C per poder utilitzar la crida a sistema en concret (en aquest cas unistd.h). Si fos necessari "*linkar*" el teu programa amb alguna llibreria concreta, que no fossin les que utilitza el compilador de C per defecte, el

normal és que aparegui també en aquesta secció. A més de la DESCRIPTION, en les crides a funció en general (sigui crida a sistema o a llibreria del llenguatge) podem trobar la secció RETURN VALUE (amb els valors que retorna la funció) i una secció especial, ERRORS, amb la llista d'errors. Finalment, també trobem diverses seccions on aquí destaquem també la secció de NOTES (aclariments) i SEE ALSO (crides relacionades).

```
WRITE(2) Linux Programmers Manual WRITE(2)
NAME
       write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>
       ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
       write() writes up to count bytes to the file referenced by the file
       descriptor fd from the buffer starting at buf.  POSIX requires that a
       read() which can be proved to occur after a write() has returned
       returns the new data.  Note that not all file systems are POSIX con-
       forming.
RETURN VALUE
       On success, the number of bytes written are returned (zero indicates
       nothing was written).  On error, -1 is returned, and errno is set
       appropriately.   If count is zero and the file descriptor refers to a
       regular file, 0 will be returned without causing any other effect.  For
       a special file, the results are not portable.
ERRORS
       EAGAIN Non-blocking I/O has been selected using O_NONBLOCK and the
                 write would block.
       EBADF fd is not a valid file descriptor or is not open for writing.
       ….

       Other errors may occur, depending on the object connected to fd.
NOTES
       A successful return from write() does not make any guarantee that data
       has been committed to disk.  In fact, on some buggy implementations, it
       does not even guarantee that space has successfully been reserved for
       the data.   The only way to be sure is to call fsync(2) after you are
       done writing all your data.

SEE ALSO
       close(2), fcntl(2), fsync(2), ioctl(2), lseek(2), open(2), read(2),
       select( 2), fwrite( 3), writev( 3)
```

figura 3 man 2 write (simplificat)

El man és simplement una eina del sistema que interpreta unes marques en fitxers de text i les mostra per pantalla seguint les instruccions d'aquestes marques. Les quatre coses bàsiques que has de saber són:

- Normalment una pàgina de man ocupa diverses pantalles, per anar avançant simplement cal apretar la barra espaiadora.
- Per anar una pantalla cap enrere pots prémer la lletra **b** (back).
- Per buscar un text i anar directament pots fer servir el caràcter "/" seguit del text. Per exemple "/SEE ALSO" us porta directe a la primera aparició del text "SEE ALSO". Per anar a la següent aparició del mateix text simplement utilitza el caràcter **n** (next).
- Per sortir del manat el caràcter **q** (quit).

Bibliografia

La documentació que us donem en aquest quadern normalment és suficient per realitzar les sessions, però en cada sessió us donarem alguna referència extra.

- Guia de BASH shell:
  - De l'assignatura ASO (en català): http://docencia.ac.upc.edu/FIB/grau/ASO/files/lab/aso-lab-bash-guide.pdf

- En anglès: http://tldp.org/LDP/abs/html/index.html

Exercicis a realitzar al laboratori

- Les pràctiques es realitzaran en un sistema Ubuntu 20.04 LTS o en un sistema OpenSuse 15.4 64 bits
- Tens a la teva disposició una imatge del sistema igual a la dels laboratoris per poder preparar les sessions des de casa. La imatge es pot utilitzar des de VirtualBox:
  - Imatge Ubuntu: https://softdocencia.fib.upc.edu/software/Ubuntu22v3r1.ova
  - Imatge OpenSuse (aquesta imatge té una mida molt més gran que la imatge disponible d'Ubuntu): https://softdocencia.fib.upc.edu/software/suse156v1r3.ova
  - VirtualBox es pot descarregar de:
    - Windows: https://download.virtualbox.org/virtualbox/7.0.20/VirtualBox-7.0.20-163906-Win.exe
    - Linux: https://www.virtualbox.org/wiki/Linux_Downloads
    - MAC/OS (basats en processador Intel)[1]: https://download.virtualbox.org/virtualbox/7.0.20/VirtualBox-7.0.20-163906-OSX.dmg
  - En l'apartat de documentació de la pàgina web de l'assignatura tens una guia ràpida sobre com instal·lar l'entorn: https://docencia.ac.upc.edu/FIB/grau/SO/

- Contesta en un fitxer de text anomenat entrega.txt totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s' ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 
- Les línies de l'enunciat que comencen pel caràcter "#" indiquen comandes que has de provar. **NO** has d'escriure el caràcter #.
- **Per lliurar: fitxer sessio01.tar.gz**

  #tar zcfv sessio01.tar.gz entrega.txt

Accés al sistema i execució d' un terminal

La informació sobre com accedir al sistema i llançar una Shell la tens en el treball previ. Consulta la secció del treball previ i llança una Shell.

Navegar pels directoris (carpetes en entorns gràfics)

Podràs observar que la gran majoria de les comandes bàsics a Linux són 2 o 3 lletres que sintetitzen l'operació a realitzar (en anglès per suposat). Per exemple, per canviar de directori (change directory) tenim la comanda cd. Per veure el contingut d'un directori (list directory) tenim la comanda ls, etc.

A UNIX els directoris estan organitzats de forma jeràrquica. El directori base és l'arrel (representada pel caràcter /) i a partir d'aquí pengen la resta de directoris del sistema, en el qual se situen arxius i directoris comuns per a tots els usuaris. A més, dins d'aquesta jerarquia, cada usuari sol tenir assignat un directori (*home directory*), pensat perquè actuï com a base de la resta dels seus directoris i fitxers. Quan un usuari inicia un terminal, el seu directori *home* pasa a ser

[1] Els portàtils Mx de Mac (M1, M2,... basats en processadors ARM) no són compatibles amb aquest programari. Si tens un portàtil d'aquest tipus envia un email al teu professor perquè et doni alternatives.

el seu directori actual de treball. Per modificar el directori actual de treball pot fer servir la comanda **cd**, que li permet navegar per tota la jerarquia de fitxers.

A continuació, realitza els següents exercicis utilitzant les comandes que consideris més oportunes:

1. Dins de la carpeta Documents que hi ha en el teu directori home[2], crea els directoris per a les 5 primeres sessions de l'assignatura amb els noms S1, S2, S3, S4 i S5.

PREGUNTA 1. Quines comandes heu utilitzat per crear els directoris S1... S5?

2. Si obres un *File Browser*, i vas a la mateixa "carpeta" en què estàs a la Shell, hauries de veure una imatge que mostra el contingut del directori (els 5 directoris que acabes de crear).
3. A la Shell, canvia el directori actual de treball al directori S1.
4. Llista el contingut del directori. Aparentment no hi ha res. No obstant això, hi han dos "**fitxers ocults**". Tots els fitxers que en UNIX comencen pel caràcter "." són **fitxers ocults**, i solen ser especials. Consulta quina opció has d'afegir a la comanda per veure tots els fitxers. Els fitxers que es veuen ara són:
    - Fitxer de tipus directori ".": Fa referència al mateix directori en el qual estàs en aquell moment. Si executes (`cd .`) veuràs que segueixes en el mateix directori. Més endavant veurem quina utilitat té.
    - Fitxer de tipus directori "..": Fa referència al directori de nivell immediatament superior al que estem. Si executes (`cd ..`) veuràs que canvies al directori d'on venies.
    - Fixeu-vos que aquests fitxers ocults no apareixen en l'entorn gràfic: si entres a la carpeta S1 apareix buida (en l'entorn gràfic, també hi ha una opció de configuració per a les carpetes que permet visualitzar els fitxers ocults).

PREGUNTA 2. Quina comanda utilitzes per llistar el contingut d'un directori? Quina opció cal afegir per veure els fitxers ocults?

5. Les opcions de les comandes normalment es poden acumular. Mira en el manual quina opció cal incloure per veure informació estesa sobre els fitxers i prova-ho.

PREGUNTA 3. Quina opció cal afegir a ls per veure informació estesa dels fitxers? Quins camps es veuen per defecte amb aquesta opció? (si no trobes la informació al pregunta al teu professor)

6. Quan utilitzem molt sovint una configuració específica d'una comanda se sol fer servir el que es coneix com a "àlies". Consisteix a definir una pseudo-comanda que la Shell coneix. Per exemple, si veiem que la comanda `ls` sempre l'executem amb les opcions "`-la`", podem redefinir "ls" com un àlies de la següent manera:

    ```
    #alias ls='ls -la'
    ```

    Executa aquesta comanda àlies i a continuació executa ls sense arguments. Comprova que la sortida és la de la comanda `ls` amb les opcions `-la`.

[2] Els ordinadors dels "laboratoris generals" de la facultat (amb OpenSuse) estan configurats perquè l'únic directori que conservi les dades un cop apagades sigui el directori *dades*, **la resta no està garantit que es conservi**. Els directoris *Documents* i *Downloads* que teniu en el vostre home són en realitat uns enllaços als directoris *dades/Documents* i *dades/Downloads* per tant es conservaran. Els ordinadors de les aules dels "laboratoris de sistemes" de la facultat (amb Ubuntu) no conserven cap dada un cop apagats per tant si voleu conservar els fitxers els haureu de guardar en algun altre lloc.

7. Podem veure una informació similar a la de la comanda ls –la en l'entorn gràfic. Cerca al File Browser l'opció de configuració que modifica el format de la informació que mostra per a cada fitxer i la quantitat d' informació que mostra.

PREGUNTA 4. Quines opcions de menú has activat per estendre la informació que mostra el File Browser?

8. Des de la Shell esborra alguns dels directoris que has creat, comprova que realment no apareixen i torna a crear-los. Mira com es fa el mateix en l'entorn gràfic.

PREGUNTA 5. Quina seqüència de comandes has executat per esborrar un directori, comprovar que no hi és i tornar a crear-lo?

Comandes bàsiques del sistema per accedir a fitxers

1. Crea un fitxer. Per crear un fitxer que contingui qualsevol text tenim diverses opcions, per exemple, obrir l'editor i escriure qualsevol cosa:

```
#gedit test
```

2. Per poder executar qualsevol comanda veuràs que has d'obrir una altra Shell perquè la que tenies està bloquejada per l'editor (això succeeix només en obrir el primer fitxer, no si ja tens obert l'editor). Això és així perquè la Shell per defecte espera fins que la comanda actual acabi abans de mostrar el prompt i processar la següent comanda. Per evitar tenir una Shell oberta per a cada comanda que volem executar simultàniament, podem demanar-li al Shell que executi les comandes en **segon pla** (o en *background*). Quan fem servir aquest mètode, la Shell deixa en execució la comanda i immediatament mostra el prompt i passa a esperar la següent ordre (sense esperar fins que la comanda acabi). Per executar una comanda en segon pla afegim al final de la línia de comandes el caràcter especial "&". Per exemple:

```
#gedit test &
```

3. Per veure d'una forma ràpida el contingut d'un fitxer, sense obrir una altra vegada l'editor, tenim diverses comandes. Aquí esmentarem 2: cat i less. Afegeix al fitxer test 3 o 4 **pàgines** de text (qualsevol cosa). Prova les comandes cat i less.
    - Nota: si el fitxer que crees no és prou gran no veuràs cap diferència.

PREGUNTA 6. Quina diferència hi ha entre la comanda cat i less?

4. Copia el fitxer test diverses vegades (afegint un número diferent al final del nom de cada fitxer destí, p.ex. "test2") Què passaria si el fitxer origen i destí tinguessin el mateix nom? Mira en el man l'opció "–i " de la comanda cp. Què fa? Crea un àlies de la comanda (anomena'l cp) que inclogui l'opció –i.

PREGUNTA 7. Per a què serveix l'opció -i de la comanda cp? Quina és la comanda per fer un àlies de la comanda que inclogui l'opció -i?

5. Prova a esborrar algun dels fitxers que acabes de crear i a canviar-li el nom a d'altres. Fes un àlies amb l'opció –i de la comanda rm (anomena'l rm). Comprova també l'opció -i de la comanda mv.

PREGUNTA 8. Què fa l'opció -i de la comanda rm? I l'opció -i del mv? Escriu la comanda per fer un àlies de la comanda rm que inclogui l'opció -i.

6. Una altra comanda que és molt útil és la comanda grep. La comanda grep ens permet buscar un text (explícit o mitjançant un patró) en un o més arxius. Afegeix una paraula en un dels fitxers que has copiat i prova la comanda grep per buscar aquesta paraula. Per exemple, afegim la paraula "hola" a un dels fitxers i fem la prova:

```
#grep hola test test1 test2 test3 test4
```

7. La comanda ls –l també permet veure els permisos que té un fitxer. A UNIX, els permisos s'apliquen a tres nivells: el propietari del fitxer (u), els usuaris del mateix grup (g), i la resta d'usuaris (o). I fan referència a tres operacions o modes d'accés: lectura (r), escriptura (w) i execució (x). Per exemple, si en el directori actual només tenim el fitxer f1, i aquest fitxer té permís de lectura i escriptura per al propietari del fitxer, només lectura per als membres del seu grup i només lectura per a la resta d' usuaris de la màquina, l' execució de la comanda donaria la sortida següent:

```
#ls -la
drwxr-xr-x 26 alumne      alumne       884    2011-09-15 14:31 .
drwxr-xr-x 3  alumne      alumne       102    2011-09-15 12:10 ..
-rw-r--r-- 1  alumne      alumne       300    2011-09-15 12:20 f1
```

La primera columna de la sortida indica el tipus de fitxer i els permisos d' accés. El primer caràcter codifica el tipus de fitxer (el caràcter 'd' significa directori i el caràcter '-' significa fitxer de dades). I a continuació el primer grup de 3 caràcters representen, en aquest ordre, si el propietari té permís de lectura (mitjançant el caràcter 'r') o no el té (i llavors apareix el caràcter '-'), si té permís d'escriptura (caràcter 'w') o no pot escriure (caràcter '-') i si té o no permís d'execució (caràcter 'x' o caràcter '-'). El segon grup de 3 caràcters són els permisos que tenen els membres del grup de propietari i el darrer grup de 3 caràcters són els permisos d' accés que tenen la resta d' usuaris de la màquina.
Aquests permisos es poden modificar mitjançant la comanda chmod. La comanda chmod ofereix diverses maneres per especificar els permisos d'accés, una manera molt senzilla consisteix a indicar primer els usuaris que es veuren afectats pel canvi de permisos, com volem canviar aquests permisos (afegir, treure o assignar directament) i l'operació afectada. Per exemple la comanda:

```
#chmod ugo + x f1
```

Modificaria els permisos de f1, activant el permís d' execució sobre f1 per a tots els usuaris de la màquina.

La comanda:

```
#chmod o-x f1
```

Modificaria els permisos de f1 treient el permís d' execució per als usuaris que no són el propietari ni pertanyen al seu grup.

I la comanda:

```
#chmod ug=rwx f1
```

Faria que els permisos d' accés a f1 fossin exactament els indicats: lectura, escriptura i execució per al propietari i els membres del seu grup.

Si estàs treballant en els "laboratoris generals" de la facultat (amb OpenSuse), comprova en quin directori tens el fitxer test. Si està dins de la carpeta *dades* (o en *$HOME/Documents* o en *$HOME/Downloads*) fes una còpia d'aquest fitxer en el teu directori home per executar aquests comandes[3]. Modifica els permisos del fitxer de test per deixar només els d'escriptura per al propietari del fitxer, el seu grup i la resta d' usuaris, i intenta fer un cat. Torna a modificar els permisos de test deixant només els de lectura per al propietari del fitxer, el seu grup i la resta d'usuaris i intenta esborrar-lo.

[3] El motiu per fer aquesta còpia és que en aquests laboratoris el directori està gestionat amb un sistema de fitxers de Windows i no és compatible amb la comanda chmod. Si fas servir chmod per canviar fitxers que pengen d'aquesta carpeta no té efecte. Tanmateix el vostre directori home en aquestes màquines sí que forma part d'un sistema de fitxers Linux.

PREGUNTA 9. Quines opcions de chmod has utilitzat per deixar només els permisos d'escriptura? Quin resultat ha tornat a intentar veure el fitxer test? Quines opcions de chmod has utilitzat per deixar només els permisos de lectura? Has aconseguit esborrar-lo?

El sistema de fitxers

Per a cada fitxer d'un sistema tipus UNIX (Linux, FreeBSD, macOS, Android, etc.) es manté una estructura dades anomenada inode (inode). Pots veure el contingut d'un inode amb la comanda stat. Cada inode s'identifica unívocament amb un número (l'índex a la taula d'inodes del sistema). L' inode conté les dades del fitxer en bytes, però la unitat mínima d' assignació és el bloc. La mida del bloc la decideix el creador del sistema de fitxers. Per exemple, 512 bytes. En el cas d'un directori, les seves dades són parells (nom del fitxer, número d'inode) per a cada fitxer del directori. L'inode també conté les metadades del fitxer (informació de gestió): el propietari, la mida, els permisos, la data de l'última modificació, etc.

8. Executa una línia de comandes per mostrar la informació del fitxer test. En concret, hauries de saber quants blocs de dades té assignats, la mida d'un bloc de dades, el seu nombre d'inodo i el nombre de camins (*pathnames*) diferents per arribar a aquest fitxer.

PREGUNTA 10. Quina línia de comandes has executat per mostrar la informació de l'inode del fitxer test?

9. El número d'inode també apareix a la sortida la comanda ls, amb l'opció apropiada. Executa ls -lia i assegura't d'entendre totes i cadascuna de les dades que apareixen.

PREGUNTA 11. Quants enllaços (links) té el directori S1?

10. Ara podem reinterpretar la comanda mkdir. En crear un directorio es creen automàticament dos fitxers nous. El '.' és un nou camí al nou directori. El '..' és un nou camí des del nou directori al seu directori pare. Per tant, el nou directori tindrà dos enllaço. L'altre és el nom del nou directori, que es guarda dins del directori pare. A aquest tipus d'enllaços se'ls anomena hard links i també es pot crear entre fitxers regulars (els que tenen un '-' a la primera columna d'un ls -l) amb la comanda ln.
11. Executa els següents comandes:

```
#echo "això és una prova" > pr.txt

#ln -s pr.txt sl_pr

#ln pr.txt hl_pr
```

PREGUNTA 12. De quin tipus és cadascun de links creats, sl_pr i hl_pr?

12. Executa la comanda stat sobre pr.txt, sobre sl_pr i sobre hl_pr. Busca en la sortida de stat la informació sobre l' innode, el tipus de fitxer i el nombre de links.

PREGUNTA 13. Quin és el nombre de links que té cada fitxer? Què significa aquest valor? Quin inode té cada fitxer?

13. Executa les comandes cat, namei i readlink sobre sl_pr i sobre hl_pr.

PREGUNTA 14. Observes alguna diferència en el resultat d'algun de les comandes quan s'executen sobre sl_pr i quan s'executen sobre hl_pr? Si hi ha alguna diferència, explica el motiu.

14. Elimina ara el fitxer pr.txt i torna a executar les comandes stat, cat, namei i readlink tant sobre sl_pr com hl_pr.

PREGUNTA 15. Observes alguna diferència en el resultat d'algun de les comandes quan s'executen sobre sl_pr i quan s'executen sobre hl_pr? Si hi ha alguna diferència, explica el motiu.

PREGUNTA 16. Observes alguna diferència respecte a l'execució d'aquests comandes abans i després d'esborrar el fitxer pr.txt? Si hi ha alguna diferència, explica el motiu.

UNIX, a diferència de MS Windows, manté un sol arbre de sistema de fitxers, on pots arrelar (mount) altres sistemes de fitxers. Cada sistema de fitxers està associat a una partició lògica (/dev/sda1, per exemple). Per tant, per saber de tot el que espai lliure disposes en el teu sistema de fitxers hauràs de fer una ullada al que tens muntat i a on.

15. Prova les següents comandes i consulta el seu *man page* detingudament.

```
#df -h

#mount
```

PREGUNTA 17. Com podeu saber els sistemes de fitxers muntats en el vostre sistema i de quin tipus són? Indica, a més, en quins directoris estan muntats.

PREGUNTA 18. Com es pot saber el nombre d'inodes lliures d'un sistema de fitxers? Quina comanda has utilitzat i amb quines opcions?

PREGUNTA 19. Com es pot saber l'espai lliure d'un sistema de fitxers? Quina comanda has utilitzat i amb quines opcions?

Variables d'entorn

Els programes s'executen en un determinat entorn o context: pertanyen a un usuari, a un grup, a partir d' un directori concret, amb una configuració de sistema quant a límits, etc. S' explicaran més detalls sobre el context o entorn d' un programa en el tema de processos.

En aquesta sessió introduirem les **variables d' entorn**. Les variables d'entorn són similars a les constants que es poden definir en un programa, però estan definides abans de començar el programa i normalment fan referència a aspectes de sistema (directoris per defecte per exemple) i marquen alguns aspectes importants de la seva execució, ja que algunes d'elles són utilitzades per la Shell per definir el seu funcionament. Se solen definir en majúscules, però no és obligatori. Aquestes variables poden ser consultades durant l' execució dels programes mitjançant funcions de la llibreria de C. Per consultar el significat de les variables que defineix la Shell, pots consultar el man de la Shell que estiguis utilitzant, en aquest cas bash (man bash, apartat Shell Variables).

16. Executa la comanda "env" per veure la llista de variables definides en l'entorn actual i el seu valor.

Per indicar-li a la Shell que volem consultar una variable d'entorn hem de fer servir el caràcter $ davant del nom de la variable, perquè no el confongui amb una cadena de text qualsevol. Per veure el valor d'una variable en concret utilitza la comanda echo:

```
#echo $USERNAME
#echo $PWD
```

17. Algunes variables les actualitza la Shell dinàmicament, per exemple, canvia de directori i torna a comprovar el valor de PWD. Què creus que significa aquesta variable?
18. Comprova el valor de les variables PATH i HOME.

PREGUNTA 20. Quin és el significat de les variables d'entorn PATH, HOME i PWD?

PREGUNTA 21. La variable PATH és una llista de directoris, quin caràcter fa de separador entre un directori i un altre?

També podem definir o modificar una variable d'entorn utilitzant la següent comanda (per a modificacions no es fa servir el $ abans del nom):

```
export NOM_VARIABLE=valor (sense espais)
```

19. Defineix dues variables noves amb el valor que vulguis i comprova el seu valor.

PREGUNTA 22. Quina comanda has fet servir per definir i consultar el valor de les noves variables que has definit?

20. Baixa't el fitxer S1.tar.gz i copia'l a la carpeta S1. Descomprimeix-lo executant la comanda *tar xvfz S1.tar.gz* per obtenir el programa "ls" que utilitzarem a continuació.
21. Situa't a la carpeta S1 i executa les següents comandes:

```
#ls
#./ls
```

Fixeu-vos que, amb la primera opció, s'executa la comanda del sistema en lloc de la comanda ls que hi ha al teu directori. No obstant això, amb la segona opció has executat el programa ls que t'acabes de baixar en lloc d'usar la comanda ls del sistema.

22. Afegeix el directori "." al principi de la variable PATH mitjançant la comanda (fíxa't en el caràcter separador de directoris):

```
#export PATH=.:$PATH
```

Mostra el nou valor de la variable PATH i comprova que, a banda del directori ".", encara conté els directoris que tenia originalment (no volem perdre'ls).
Executa ara la comanda:

```
#ls
```

PREGUNTA 23. Quina versió del ls s'ha executat?: El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.

PREGUNTA 24. El directori en el qual estàs, està definit en la variable PATH? Què implica això?

23. Modifica la variable PATH per eliminar el directori ".". No és possible modificar parcialment la variable pel que has de redefinir-la. Mostra el contingut actual de la variable PATH i redefineix-la usant tots els directoris que contenia excepte el ".".

Executa ara la comanda:

```
#ls
```

PREGUNTA 25. Quin binari ls s'ha executat en cada cas dels anteriors:El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.

24. Afegeix el directori "." al final de la variable PATH (fíja't en el caràcter separador de directoris):

```
#export PATH = $PATH:.
```

Comprova que, a banda del directori ".", la variable PATH encara conté els directoris que tenia originalment (no volem perdre'ls).
Executa ara la comanda:

```
#ls
```

PREGUNTA 26. Quin binari ls s'ha executat en cada cas dels anteriors: El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.

Mantenim els canvis: fitxer .bashrc

Els canvis que hem fet durant aquesta sessió (excepte els que fan referència al sistema de fitxers) es perdran en finalitzar la sessió (definició d'àlies, canvis en el PATH, etc). Perquè no es perdin, hem d'inserir aquests comandes en el fitxer de configuració de sessió que utilitza la Shell. El nom del fitxer depèn de la Shell que estem utilitzant. En el cas de Bash és $HOME/.bashrc. Cada vegada que iniciem una sessió, la Shell es configura executant tots les comandes que trobi en aquest fitxer.

25. Edita el fitxer $HOME/.bashrc i afegeix la modificació del PATH que t'hem demanat en l'apartat anterior. Afegeix també la definició d'un àlies perquè cada vegada que executem la comanda ls es faci amb l'opció –m. Per comprovar que has modificat bé el .bashrc executa les següents comandes:

```
#source $HOME/.bashrc
#ls
```

I comprova que la sortida del ls es correspon amb la de l'opció –m. La comanda source provoca l'execució de tots les comandes que hi ha el fitxer que li passem com a paràmetre (és una manera de no haver de reiniciar la sessió per fer que aquests canvis siguin efectius).

- **Nota:** en l'entorn de treball els "laboratoris de sistemes" de la facultat (amb Ubuntu), el sistema s'arrenca carregant una nova imatge del sistema i per tant tots els teus fitxers es perden i els fitxers de configuració del sistema es reinicialitzen mitjançant una còpia remota. Això significa que si reinicies la sessió començaràs a treballar amb el fitxer .bashrc original i no es conservaran els teus canvis.

Alguns caràcters especials útils de la Shell

A la secció anterior ja hem introduït el caràcter &, que serveix per executar una comanda en segon pla. Altres caràcters útils de la Shell que introduirem en aquesta sessió són:

- *: La Shell el substitueix per qualsevol grup de caràcters (excepte el "."). Per exemple, si executem (`grep prova t*`) veurem que la Shell substitueix el patró t* per la llista de tots els fitxers que comencen per la cadena "t". Els caràcters especials de la Shell es poden utilitzar amb qualsevol comanda.
- >: La sortida de dades de les comandes per defecte està associada a la pantalla. Si volem canviar aquesta associació, i que la sortida es dirigeixi a un fitxer, podem fer-ho amb el caràcter ">". A aquesta acció se l'anomena "redireccionar la sortida". Per exemple, `ls > salida_ls`, guarda la sortida de ls al fitxer salida_ls. Prova a executar la comanda anterior. A continuació, prova-ho amb una altra comanda però amb el mateix fitxer de sortida i comprova el contingut del fitxer salida_ls.

- >>: Redirecciona la sortida de dades d'una comanda a un fitxer però en lloc d' esborrar el contingut del fitxer s'afegeix al final del que hi hagués. Repeteix l'exemple anterior però amb ">>" en lloc de ">" per comprovar la diferència.

PREGUNTA 27. Quina diferència hi ha entre utilitzar > i >>?

Fer una còpia de seguretat per a la següent sessió

Si estàs treballant en els "laboratoris de sistemes" de la facultat, llavors cada vegada que arrenquem l'ordinador es carrega una nova imatge que esborra el contingut anterior (estan configurats d'aquesta manera). Per tant, cal fer una còpia de seguretat dels canvis realitzats si volem conservar-los per a la següent sessió. Per guardar els canvis pots utilitzar la comanda tar. Per exemple, si vols generar un fitxer que contingui tots els fitxers del directori S1, a més del fitxer .bashrc, pots executar la següent comanda des del teu directori HOME:

```
#tar zcvf carpetaS1.tar.gz S1/* .bashrc
```

Finalment has de guardar aquest fitxer en un lloc segur com per exemple un pendrive o enviar-te'l per correu electrònic o en algun emmagatzematge al núvol.

En cas que estiguis en els "laboratoris generals" de la facultat, si tots els fitxers i directoris que has creat estan dins del directori *dades* aleshores no és necessari que et guardis els teus fitxers en un altre lloc, es conservaran per a la pròxima vegada que iniciïs sessió. El mateix passa amb el fitxer *$HOME/.bashrc* o qualsevol fitxer o directori creat dins de *$HOME/Documents* o *$HOME/Descarregues*, perquè els tres casos són enllaços a directoris i fitxers de *dades* i estan allà guardats en realitat. Si has creat fitxers en algun altre lloc (com en *$HOME* o en *$HOME/Escriptori*) llavors si vols conservar-los hauries de moure'ls a una altra ubicació dins del directori.

Si estàs treballant amb una màquina virtual al teu propi ordinador llavors no cal que facis cap còpia, en reiniciar la màquina les teves dades seguiran a la imatge.

Sessió 2. El llenguatge C

Preparació prèvia

Objectius

En aquesta sessió l'objectiu és practicar amb tot allò relacionat amb la creació d'executables. En aquest curs utilitzarem llenguatge C. Practicarem la correcció d' errors tant de makefiles com de fitxers C i la generació d' executables a partir de zero: fitxers font, makefile i llibreries.

Habilitats

- Ser capaç de crear executables donat un codi en C:
  - Creació d' executables i utilització de makefiles senzills.

- Ser capaç de crear programes en C des de zero:
  - Definició de tipus bàsics, taules, funcions, condicionals i bucles.
  - Utilització de punters.
  - Formatejat en pantalla dels resultats dels programes.
  - Programes ben estructurats, clars, robustos i ben documentats.
  - Creació i modificació de makefiles senzills: afegir regles noves i afegir dependències.
  - Aplicació de sangries a codi font en C.

Coneixements previs

- Programació bàsica: tipus de dades bàsiques, taules, sprintf.
- Programació mitjana: punters en C, accés als paràmetres de la línia de comandes. Fases del desenvolupament d'un programa en C: Preprocessador/Compilador/Enllaçador (o linkador).
- Ús de makefiles senzills.

Guia per al treball previ

- Llegir les pàgines de man de les següents comandes. Aquestes comandes tenen múltiples opcions, llegiu amb especial detall les que us comentem a la columna "Paràmetres" de la següent taula. No cal llegir les pàgines senceres de cada comanda sinó el seu funcionament bàsic que podeu ampliar a mesura que ho necessiteu.

Per llegir al man	Descripció bàsica	Paràmetres
`make`	Utilitat per automatitzar el procés de compilació/linkatge d'un programa o grup de programes	
`gcc`	Compilador de C de GNU	`-c,-o, -I (i majúscula), -L,-l (ele minúscula)`
`sprintf`	Conversió de format emmagatzemant-la en un búffer	
`atoi`	Converteix un string a un número enter	
`indent`	Indentació de fitxers font	

- Consultar el resum sobre programació en C que està disponible a la pàgina web de l'assignatura: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf
- Consultar el resum sobre la comparació entre C i C ++ que està disponible a la pàgina web de l' assignatura: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx
- Crea la carpeta $HOME/Documents/S2 i situa't-hi en ella per realitzar els exercicis.
- Baixa't el fitxer S2.tar.gz, descomprimeix-lo amb *tar zxfv S2.tar.gz* per obtenir els fitxers d' aquesta sessió.

Durant el curs, utilitzarem directament les crides a sistema per escriure en pantalla. El que en C++ era un cout, aquí és una combinació de **sprintf** (funció de la llibreria de C) + **write** (crida a sistema per escriure).

La funció **sprintf** de la llibreria estàndard de C s'utilitza per formatejar un text i emmagatzemar el resultat en un búffer. El primer paràmetre és el búffer, de tipus char*, el segon paràmetre és una cadena de caràcters que especifica el text a guardar així com el format i tipus de totes les variables o constants que es volen incloure a la cadena, i a partir d'aquí aquestes variables o constants (mireu els exemples que us donem al resum de C). Tot el que escrivim a la pantalla ha de ser ASCII, per la qual cosa prèviament haurem de formatejar-ho amb sprintf (excepte en el cas que ja sigui ASCII).

Posteriorment s'ha d'utilitzar la crida al sistema **write** per escriure aquest búffer en un dispositiu. Durant aquestes primeres sessions escriurem en la "sortida estàndard", que per defecte és la pantalla, o per la "sortida estàndard d'error", que per defecte també és la pantalla. A UNIX/Linux, els dispositius s'identifiquen amb un número, que se sol anomenar canal. La sortida estàndard és el canal 1 i la sortida estàndard d'error és el canal 2. El **número de canal** és el primer paràmetre de la crida a sistema write. La resta de paràmetres són l'**adreça** on comencen les dades (el búffer que hem generat amb sprintf) i el nombre de bytes que es volen escriure a partir d'aquesta direcció. Per conèixer la longitud del text utilitzarem **strlen** (funció llibreria de C).

Un exemple d'utilització de sprintf podria ser (utilitza el man per consultar com especificar diferents formats):

```
char buffer[256];
sprintf(buffer, "Aquest és l'exemple número %d\n", 1);
write(1, buffer, strlen(buffer));
```

Que mostra per la sortida estàndard: "Aquest és l'exemple número 1".

Solucionant problemes amb el makefile

1. Modifica el fitxer makefile perquè funcioni.
   - El makefile és el fitxer que utilitza per defecte l'eina make. Si executes "make" sense paràmetres, el fitxer per defecte és makefile.
   - Perquè el format d'un fitxer makefile sigui correcte ha de seguir un format com aquest (TAB significa que necessita un tabulador).

   ```
   Target: dependència1 dependència2... dependènciaN
   [TAB]com generar Target a partir de les dependències
   ```

   Per exemple:

```
P1: P1.c
      gcc -o P1 P1.c
```

2. Modifica el makefile perquè la regla llistaParametres tingui ben definides les seves dependències. Els fitxers executables depenen dels fitxers font a partir dels quals es generen (o fitxers objecte si separem la generació de l'executable en compilació i muntatge).
3. Modifica el makefile perquè el fitxer de sortida no sigui a.out sinó llistaParametres.
4. Crea una còpia del makefile, anomenant-la makefile_1, per lliurar-la.

Solucionant problemes de compilació

- Soluciona tots els errors de compilació que apareguin. Sempre és recomanable solucionar els errors en l'ordre en què apareixen.

  El compilador sempre dóna missatges indicant on s'ha produït l' error, en quina línia de codi, i quin és l' error. En aquest cas concret tenim el següent codi:

```
1. void main(int argc,char *argv[])
2. {
3.  char buf[80];
4.       for (i=0; i<argc; i + +){
5.            sprintf(buf,"L'argument %d es %s\n",i,argv[i]);
6.            write(1,buf,strlen(buf));
7.       return 0;
8. }
```

  I els errors són (són errors molt típics, per això els hem posat):

```
listaParametros.c: In function "main":
listaParametros.c:4: error: "i" undeclared (first use in this
function)
listaParametros.c:4: error: (Each undeclared identifier is
reported only once
listaParametros.c:4: error: for each function it appears in.)
listaParametros.c:5: warning: incompatible implicit declaration
of built-in function "sprintf"
listaParametros.c:6: warning: incompatible implicit declaration
of built-in function "strlen"
listaParametros.c:7: warning: "return" with a value, in
function returning void
listaParametros.c:8: error: syntax error at end of input
```

El primer indica que hi ha una variable (`i` en aquest cas) sense declarar a la **línia 4**. La línia 4 és on s'utilitza. Si et fixes, fem servir la `i` al bucle però no està declarada. En les **línies 5 i 6** utilitzem unes funcions (`sprintf` i `strlen`) que no hem declarat. El compilador no les troba i no pot saber si són correctes. Aquestes funcions estan definides a la llibreria de C que s'afegeix en generar el binari, però el compilador necessita la capçalera per veure si es fa servir correctament. Consulta el man per veure quins fitxers de capçalera (.h) cal incloure. La **línia 7** ens indica que tenim una funció (el main) que acaba amb un return 0 quan l'havíem declarat com un void. El normal és definir el main com una funció que retorna un int. L'últim error sol aparèixer quan hi ha un error que s'ha propagat. En aquest cas, faltava tancar una clau (la del for).

- Assegura't que el directori "." apareix en la variable PATH de manera que es trobin els executables que estiguin en el directori actual.

Analitza i entén el fitxer llistaParametres.c.

Si no saps programar a C llegeix els documents recomanats abans de fer aquests exercicis.

- La primera funció d'un programa escrit en C que s' executa és la rutina `main`.
- La rutina main té dos paràmetres que s'anomenen `argc` i `argv`. El paràmetre `argc` és un enter que conté el nombre d' elements de l'array `argv`. I `argv` és un array de strings que conté els paràmetres d'entrada que el programa rep en posar-se en execució. Existeix la convenció de tractar el nom de l' executable com a primer paràmetre. Per tant si se segueix aquesta convenció `argc` sempre serà major o igual que un i el primer element d'`argv` serà el nom d'aquest executable (la Shell segueix aquesta convenció per a tots els programes que posa en execució).
- Compila llistaParametres.c per obtenir l'executable llistaParametres, i executeu-lo amb diferents paràmetres (tipus i número), per exemple:

```
#./llistaParametres
#./llistaParametres a
#./llistaParametres a b
#./llistaParametres a b c
#./llistaParametres 1 2 3 4 5 6 7
```

Quins valors contenen argc i el vector argv en cadascun dels exemples? Tot i que passem números com a paràmetre, els programes els reben SEMPRE com una cadena de caràcters.

El sistema operatiu és l'encarregat d'omplir ambdós paràmetres (`argc` i `argv`) usant les dades introduïdes per l'usuari (ho veurem en el tema 2 de teoria).

Analitza detingudament i entén el fitxer punters.c.

L'objectiu és que assimilis el que significa declarar una variable de tipus punter. Un punter s'utilitza per guardar simplement una direcció de memòria. I en aquesta direcció de memòria hi ha un valor del tipus indicat en la declaració del punter. Aquí tens un exemple amb el cas concret de punters a enters.

```
char buffer[128];
int A;      Això és un enter
int *PA;    Això és un punter a enter (aquesta sense inicialitzar, no es
            pot utilitzar)
PA =&A;     Inicialitzem PA amb la direcció d'A
*PA = 4;    Posem un 4 a la posicion de memòria que apunta PA
if (*PA==A) {
      sprintf(buffer,"Aquest missatge ha de sortir sempre!\n");
}else{
      sprintf(buffer,"Aquest missatge NO ha de sortir MAI!\n");
}
write(1,buffer,strlen(buffer));
```

- Crea un fitxer nombres.c i afegeix una funció que comprovi que un string (en C es defineix com un "char *" o com un vector de caràcters, ja que no existeix el tipus "string")[4] que rep com a paràmetre només conté caràcters ASCII entre el '0' i el '9' (a més del '\0' al final i potencialment el '-' al principi per als negatius). La funció ha de comprovar que el

[4] La definició i utilització de "strings" és bàsica. Si no entenen com funciona consulta amb el professor de laboratori

paràmetre punter no sigui NULL. La funció retorna 1 si l'string representa un número i té com a molt 8 xifres (defineix una constant que anomenarem MAX_SIZE i utilitza-la), i 0 en qualsevol altre cas. La funció ha de tenir el prototip següent:
    - int esNombre(char *str);
- Per provar la funció esNombre, afegeix la funció main al fitxer nombres.c i fes que executi la funció esNombre passant-li tots els paràmetres que rebi el programa. Escriu un missatge per a cadascú indicant si és un número o no.
- Crea un Makefile per a aquest programa
- Utilitza indent per indentar el fitxer nombres.c (#indent nombres.c).
- **Per lliurar: previ02.tar.gz**
  #tar zcfv previ02.tar.gz nombres.c Makefile listaParametros.c makefile_1

Bibliografia

- Tutorial de programació en C: http://www.elrincondelc.com/cursoc/cursoc.html
- Resum sobre programació en C: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EsquemaProgramaC.pdf
- Comparació entre C i C + +: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/Laboratorios/C++vsC.ppsx

Bibliografia complementària

- Programació en C:
    - The C Programming Language. Kernighan, Brian W.; Ritchie, Dennis M. Prentice Hall
- Makefiles:
    - http://es.wikipedia.org/wiki/Make
    - http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/
- GNU Coding Standards
    - http://www.gnu.org/prep/standards/standards.html, especialment el punt: http://www.gnu.org/prep/standards/standards.html#Writing-C

Exercicis a realitzar al laboratori

- Per a tots els exercicis, s'assumeix que es modificarà el makefile quan sigui necessari i es provaran tots els exercicis que es demanen.
- Contesta en un fitxer de text anomenat entrega.txt totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s' ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 
- **Per lliurar: sessio02.tar.gz**

#tar zcfv sessio02.tar.gz entrega.txt makefile_1 listaParametros.c makefile_4 mis_funciones.h mis_funciones.c suma.c punters.c makefile_5 words.c makefile

Comprovant el nombre de paràmetres

A partir d' aquest exercici, farem que tots els nostres programes siguin robustos i usables comprovant que el nombre de paràmetres que reben sigui correcte. En cas que no ho sigui, els programes (1) imprimiran un missatge que descrigui com usar-los, (2) una línia que descrigui la seva funcionalitat, i (3) acabaran l'execució. És el que es coneix com afegir una funció Usage().

1. Implementa **una funció** que s'encarregui de mostrar el missatge descriptiu d'utilització del programa listaParametros i acabi l'execució del programa (crida a aquesta funció Usage()). **Modifica el programa listaParametros** perquè comprovi que almenys hi ha 1 paràmetre i en cas contrari invoqui a la funció Usage(). Recorda que el nom de l'executable es considera un paràmetre més i així es reflecteix en les variables argc i argv.

   Exemple de comportament esperat:

```
#listaParametros a b          <-- Exemple d' ús correcte
L'argument 0 és listaParametros
L'argument 1 és a
L'argument 2 és b
#listaParametros              <-- Exemple d' ús incorrecte
Usage:listaParametros arg1 [arg2.. argn]
Aquest programa escriu per la seva sortida la llista
d'arguments que rep
```

Processament de paràmetres

2. Crea una còpia de nombres.c (treball previ) anomenada suma.c.
3. Afegeix una altra funció al suma.c que converteixi un caràcter a número (1 xifra). La funció assumeix que el caràcter es correspon amb el caràcter d' un nombre.

```
unsigned int char2int(char c);
```

4. Modifica suma.c afegint una funció mi_atoi que rebi com a paràmetre un string i retorni un enter corresponent a l'string convertit a número. Aquesta funció assumeix que l'string no és un punter a NULL, però pot representar un nombre negatiu. Si l'string no conté un nombre correcte el resultat és indefinit. El comportament d'aquesta funció és el mateix que el de la funció atoi de la llibreria de C. Utilitza la funció char2int de l'apartat anterior.

```
int mi_atoi(char *s);
```

5. Modifica suma.c perquè es comporti de la següent manera: Després de comprovar que tots els paràmetres són números, els converteix a int, els suma i escriu el resultat. Modifica també el Makefile perquè creï el fitxer executable suma. La següent figura mostra un exemple del funcionament del programa:

```
#suma 100 2 3 4 100
La suma és 209
#suma -1 1
La suma és 0
#suma 100 a
Error: el paràmetre "a" no és un número
```

6. Crea una còpia del makefile, anomenant-la makefile_4, per lliurar-la.

Fem servir el preprocessador de C: Dividim el codi (#include)

Volem separar les funcions auxiliars que anem creant dels programes principals, de manera que puguem reutilitzar-les quan les necessitem. En C, quan volem encapsular una funció o un conjunt de funcions, el normal és crear dos tipus de fitxers:

- Fitxers **capçalera** ("*include*"). Són fitxers de text amb extensió ".h" que contenen **prototips de funcions** (capçaleres), definicions de **tipus de dades** i definicions de **constants**. Aquests fitxers són "inclosos" pel preprocessador mitjançant la directiva #include <fitxer.h> en el lloc exacte en què apareix la directiva, per la qual cosa la posició

és important. Afegir un fitxer ".h" **és equivalent a copiar i enganxar el codi del fitxer** en el lloc on hi ha la directiva #include i el seu únic objectiu és facilitar la llegibilitat i modularitat del codi. El correcte és posar en els fitxers capçalera aquells prototips de funcions, tipus de dades i constants que vulguem fer servir en més d'un fitxer.

- Fitxers auxiliars, **fitxers objecte o llibreries**. Aquests fitxers contenen les **definicions de variables** globals que necessitin les funcions auxiliars (si en necessiten alguna) i la implementació d'aquestes funcions. Podem oferir el fitxer ".c" directament, el fitxer objecte ja compilat (si no volem oferir el codi font) o, en cas de tenir més d'un fitxer objecte, ajuntar-los tots en una llibreria (archive amb la comanda `ar`).

7. Separa les funcions realitzades en els exercicis anteriors (excepte el main) en un fitxer a part (mis_funciones.c) i crea un fitxer de capçaleres (mis_funciones.h), on defineixis les capçaleres de les funcions que ofereixes, i incloïu-lo en el programa suma.c. Afegeix una petita descripció de cada funció al costat del prototip (afegeix-la com a comentari). Modifica el makefile afegint una nova regla per crear el fitxer objecte i modifica la regla del programa suma perquè ara es creï utilitzant aquest fitxer objecte. Afegeix les dependències que creguis necessàries.
8. Modifica la funció Usage de manera que com a mínim el programa suma 2 paràmetres.
9. Crea una còpia del makefile, anomenant-la makefile_5, per lliurar-la

PREGUNTA 28. Quina opció has hagut d'afegir al gcc per generar el fitxer? Quina opció has hagut d'afegir al gcc perquè el compilador trobi el fitxer mis_funciones.h?

Treballant amb punters en C

10. Mira el codi del programa punteros.c. Modifica el makefile per compilar-lo i executar-lo. Analitza el codi per entendre tot el que fa.
11. Crea un programa anomenat words.c que accepti un únic paràmetre. Aquest programa compta el nombre de paraules que hi ha a l'string passat com a primer paràmetre. Considerarem que comença una nova paraula després de: un espai, un punt, una coma i un salt de línia (\n). La resta de signes de puntuació no es tindran en compte. Un exemple de funcionament seria:

```
#words hola
1 paraules
#words "Aquesta és una frase."
4 paraules
#words "Aquest paràmetre el tracta" "aquest paràmetre no el
tracta"
4 paraules
```

12. Modifica el makefile per compilar i muntar words.c

Sessió 3. Processos

Preparació prèvia

Objectius

Els objectius d'aquesta sessió són practicar amb les crides a sistema bàsiques per gestionar processos, i les comandes i mecanismes bàsics per monitoritzar informació de kernel associats als processos actius del sistema.

Habilitats

- A nivell d' usuari BÀSIC:
  - Ser capaç de fer un programa concurrent utilitzant les crides a sistema de gestió de processos: fork, execlp, getpid, exit, waitpid.
  - Entendre l'herència de processos i la relació pare/fill.
- A nivell d' administrador BÀSIC:
  - Ser capaç de veure la llista de processos d'un usuari i algun detall del seu estat mitjançant comandes (ps, top).
  - Començar a obtenir informació a través del pseudo-sistema de fitxers /proc.

Guia per al treball previ

- Consulta el vídeo sobre creació de processos que tens a la pàgina web de l'assignatura: http://docencia.ac.upc.edu/FIB/grau/SO/enunciados/ejemplos/EjemploCreacionProcesosVideo.wmv
- Llegeix les pàgines de manual de les crides getpid/fork/exit/waitpid/execlp. Entén els paràmetres, valors de retorn i funcionalitat bàsica associada a la teoria explicada a classe. Fixeu-vos també en els *includes* necessaris, casos d' error i valors de retorn. Consulta la descripció i les opcions indicades de la comanda ps i del pseudo-sistema de fitxers /proc.

Per llegir al man	**Descripció bàsica**	**Paràmetres**
`getpid`	Retorna el PID del procés que l'executa	
`fork`	Crea un procés nou, fill de qui l'executa	
`exit`	Acaba el procés que executa la crida	
`waitpid`	Espera la finalització d'un procés fill	
`execlp`	Executa un programa en el context del mateix procés	
`perror`	Escriu un missatge de l'últim error produït	
`ps`	Retorna informació dels processos	`-a, -u, -o`
`proc`	Pseudo-file system que ofereix informació de dades del kernel	`cmdline,cwd, environ exe, status`

- Crea el directori de l'entorn de desenvolupament per a aquesta sessió ($HOME/Documents/S3).
- Descarrega el fitxer S3.tar.gz i descomprimeix-lo en el directori que has creat per obtenir els fitxers d'aquesta sessió (tar zxvf S3.tar.gz).
- Crea un fitxer de text anomenat previ.txt i escriu la resposta a totes les preguntes.

- **Analitza el codi** dels programes d'exemple i el fitxer Makefile.exemples
    - El fitxer Makefile.exemples està preparat per compilar tots els programes excepte ejemplo_fork7.c
- **Compila tots els programes,** excepte ejemplo_fork7, **usant el fitxer Makefile.ejemplos** (veure fitxer README_S3).
- **Executa ejemplo_fork1**
    - Escriu al fitxer previ.txt els missatges que apareixen en pantalla i explica quin procés mostra cadascú (pare o fill) i per què.
- **Executa ejemplo_fork2**
    - Escriu al fitxer previ.txt els missatges que apareixen en pantalla i explica quin procés mostra cadascú (pare o fill) i per què.
- **Executa ejemplo_fork3**
    - Escriu al fitxer previ.txt els missatges que apareixen en pantalla i explica quin procés mostra cadascú (pare o fill) i per què.
- **Executa ejemplo_fork4**
    - En quin ordre apareixen en pantalla els missatges? Quin procés acaba abans l'execució?
    - **Modifica el codi d'aquest programa** perquè el pare no escrigui l'últim missatge del seu codi fins que el seu fill hagi acabat l' execució.
- **Executa ejemplo_fork5**
    - Escriu al fitxer previ.txt els missatges que apareixen en pantalla i explica quin procés mostra cadascú (pare o fill) i per què.
    - **Modifica el codi d' aquest programa**, perquè el procés fill modifiqui el valor de les variables variable_local i variable_global abans d' imprimir-hi el valor. Comprova que el pare continua veient el mateix valor que tenien les variables abans de fer el fork.
- **Executa ejemplo_fork6,** redireccionant la seva sortida estàndard a un fitxer
    - Descriu el contingut del fitxer de sortida
    - Podem assegurar que si executem diverses vegades aquest programa el contingut del fitxer serà exactament el mateix? Per què?
- **Modifica el fitxer Makefile.ejemplos** per afegir la compilació de ejemplo_fork7.c i utilitza-lo per compilar ara.
    - Per què no compila el programa ejemplo_fork7.c? Té alguna cosa a veure amb el fet de crear processos? Com es pot modificar el codi perquè escrigui el valor de la "variable_local"?
- **Executa ejemplo_exec1**
    - Descriu el comportament d'aquest programa. Què veus en pantalla? Quantes vegades apareix en pantalla el missatge amb el pid del procés? A què es deu?
- **Executa ejemplo_exec2**
    - Descriu el comportament d' aquest codi. Quins missatges apareixen en pantalla? Quants processos s'executen?

- **Consulta al man**: a quina secció pertanyen les pàgines del man que heu consultat? A més, apunta aquí si s'ha consultat alguna pàgina addicional del manual a més a més de les que s'han demanat explícitament.

- **Per lliurar: previ03.tar.gz**
  `#tar zcfv previ03.tar.gz Makefile.ejemplos ejemplo_fork4.c ejemplo_fork5.c ejemplo_fork7.c previ.txt`

Exercicis a realitzar al laboratori

- Per a tots els exercicis, s'assumeix que realitzes tots els passos involucrats.
- Contesta en un fitxer de text anomenat entrega.txt totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s'ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol:
- **Per lliurar: sessio03.tar.gz**

```
#tar zcfv sessio03.tar.gz entrega.txt makefile myPS_v0.c myPS.c
myPS2.c myPS3.c parsExec.c.
```

Comprovació d' errors de les crides a sistema

1. A partir d'ara s'inclourà sempre, **per a TOTES les crides a sistema**, la comprovació d'errors. Utilitza la crida perror (man 3 perror) per escriure un missatge que descrigui el motiu que ha produït l'error. A més, en cas que l'error sigui crític, com per exemple que falli un fork o un execlp, ha d' acabar l' execució del programa. La gestió de l' error de les crides a sistema es pot fer de forma similar al següent codi:

```
int main (int argc,char *argv[])
{
…
        if ((pid=fork())<0) error_y_exit("Error en fork",1);
…
}
void error_y_exit(char *msg,int exit_status)
{
        perror(msg);
        exit(exit_status);
}
```

Creació i mutació de processos: myPS.c

L'objectiu d' aquesta secció és practicar amb les crides a sistema de creació i de mutació de processos. Per això crearàs dos codis diferents: myPS.c i myPS_v0.c. Aquests codis ens serviran com a base per als exercicis de les seccions següents.

2. Crea un programa anomenat myPS.c que rebi un paràmetre (que serà un nom d'usuari: el nom d'usuari amb el qual has iniciat sessió, root, examen, etc.) i que creï un procés fill. El procés pare escriurà un missatge indicant el seu PID. El procés fill escriurà un missatge amb el seu PID i el paràmetre que ha rebut el programa. Després d'escriure el missatge, ambdós processos executaran un bucle "while(1);" perquè no acabin (aquest bucle l'afegim perquè farem servir aquest codi en la següent secció sobre la consulta d'informació dels processos, i en aquesta secció ens interessa que els processos no acabin l'execució).
3. Crea un makefile, que inclogui les regles "all" i "clean", per compilar i muntar el programa myPS.c.

PREGUNTA 29. Com pot saber un procés el pid dels seus fills? Quina crida poden utilitzar els processos per consultar el seu propi PID?

4. Copia el codi de myPS.c en una versió myPS_v0.c. Modifica el Makefile per compilar myPS_v0.c
5. Modifica myPS.c perquè el procés fill, després d'escriure el missatge, executi la funció muta_a_PS. Aquesta funció mutarà al programa ps. Afegeix també el codi de la funció muta_a_PS:

```
/* Executa la comanda ps -u username mitjançant la crida al sistema execlp */
/* Retorna: el codi d'error en el cas que no s'hagi pogut mutar */
void muta_a_PS(char *username)
{
  execlp("ps", "ps", "-u", username, (char*)NULL);
  error_y_exit("Ha fallat la mutació al ps", 1);
}
```

PREGUNTA 30. En quins casos s'executarà qualsevol codi que aparegui just després de la crida execlp (En qualsevol cas/En cas que l'execlp s'executi de forma correcta/En cas que l'execlp falli)?

Consulta de la informació dels processos en execució: myPS.c

L'objectiu d'aquesta secció és aprendre a utilitzar el /proc per consultar alguna informació sobre l'execució dels processos. Per a això utilitzarem els codis myPS.c i myPS_v0.c que has desenvolupat en la secció anterior.

6. Per a aquest exercici utilitzarem dos terminals de la Shell. En una executa myPS amb un sol username com a paràmetre. A la segona finestra situa't al directori /proc i comprova que apareixen diversos directoris que coincideixen amb els números de PIDs dels processos. Entra en el directori del pare i del seu fill i mira la informació estesa (permisos, propietari, etc.) dels fitxers del directori.

PREGUNTA 31. Quins directoris hi ha dins de /proc/PID_PADRE? Quina opció de ls has fet servir?

PREGUNTA 32. Per al procés pare, apunta el contingut dels fitxers status i cmdline. Compara el contingut del fitxer environ amb el resultat de la comanda env (fixa't per exemple en la variable PATH i la variable PWD) Quina relació hi ha? Cerca en el contingut del fitxer status l'estat en el qual es troba el procés i anota'l al fitxer de respostes. Anota també el temps de CPU que ha consumit en mode usuari que pots trobar-lo en el fitxer stat del procés (veuràs que stat conté una sèrie de números, consulta en el manual el format què representa cadascun d'aquests números: man proc).

PREGUNTA 33. En el cas del procés fill, a quins fitxers "apunten" els fitxers cwd i exe? Quin creus que és el significat de cwd i exe?

PREGUNTA 34. En el cas del procés fill, ¿pots mostrar el contingut dels fitxers environ, status i cmdline del procés fill? En quin estat es troba?

7. Repeteix l'experiment utilitzant el programa myPS_v0.c en lloc de myPS.c i respon de nou a les preguntes per al procés fill. Observa les diferències que hi ha entre ambdues versions del codi. Recorda que a la v0 el procés fill no mutava.

PREGUNTA 35. En el cas del procés fill, a quins fitxers "apunten" els fitxers cwd i exe? Quin creus que és el significat de cwd i exe? Quines diferències hi ha amb la versió anterior? A què es deuen?

PREGUNTA 36. En el cas del procés fill, ¿pots mostrar el contingut dels fitxers environ, status i cmdline del procés fill? En quin estat es troba? Quines diferències hi ha amb la versió anterior? A què es deuen?

Execució seqüencial dels fills: myPS2.c

L'objectiu d'aquesta secció és practicar amb la crida a sistema waitpid i entendre com influeix en la concurrència dels processos creats. En particular la utilitzaràs per crear processos que s'executin de manera seqüencial.

Aquesta crida serveix perquè el procés pare esperi que els seus processos fills acabin, perquè comprovi el seu estat de finalització i perquè el kernel alliberi les estructures de dades que els representen internament (PCBs). La posición a on es produeix l'espera és determinant per generar un codi seqüencial (tots els processos fills es creen i executen d'1 en 1) o concurrent (tots els processos fills es creen i s'executen de forma potencialment paral·lela, depenent de l'arquitectura en la qual l'executem). En aquesta secció volem fer un **codi seqüencial.** Per a això utilitzarem la crida al sistema waitpid entre una creació de procés i la següent, de manera que assegurem que no tindrem 2 processos fills executant-se alhora.

```
// Exemple esquema seqüencial
for (i=0; i<num_hijos; i + +){
       pid =fork();
       if (pid==0) {
              // codi fill
              exit(0); // Només si el fill no muta i volem que acabi
       }
       // Esperem que acabi abans de crear el següent
       waitpid(...); // els paràmetres depèn del que vulguem
}
```

8. Crea una còpia de myPS.c, anomenada myPS2.c, amb la qual treballaràs en aquest exercici. Modifica, també, el makefile per poder compilar i muntar myPS2.c.
9. Modifica myPS2.c perquè, en lloc de rebre un paràmetre (username), pugui rebre N. Fes que el programa principal creï un procés per a cada username i que s'executin de manera seqüencial. Pots eliminar el bucle infinit del final de l'execució del procés.

Execució concurrent dels fills: myPS3.c

En aquesta secció es continua treballant amb la crida a sistema waitpid. Ara la utilitzaràs per crear un esquema d'execució concurrent.

```
// Exemple esquema concurrent
for (i=0; i<num_hijos; i + +){
       pid =fork();
       if (pid==0) {
              // codi fill
              exit(0); // Només si el fill no muta i volem que acabi
       }
}
// Esperem a tots els processos
while (waitpid(...)>0); // els paràmetres depèn del que vulguem
```

Aprofitarem també per comprovar els possibles efectes que pot tenir la concurrència sobre el resultat de l' execució.

10. Crea una còpia de myPS2.c, anomenada myPS3.c, amb la qual treballaràs en aquest exercici. Modifica també el makefile per poder compilar i muntar myPS3.
11. Modifica el programa myPS3.c perquè els fills es creïn de forma concurrent. Per poder consultar la informació dels processos, fes que el pare es quedi esperant una tecla després d'executar el bucle de waitpid. Per esperar una tecla pots fer servir la crida a sistema read (`read(0,&c,sizeof(char))`, on c és un char).

12. Executa myPS3 amb diversos usernames i deixa el pare bloquejat després del bucle de waitpids. En una altra finestra comprova que cap procés fill té un directori en /proc. Comprova també l'estat en què es troba el procés pare.

PREGUNTA 37. Comprova el fitxer status de /proc/PID_PADRE. En quin estat està el procés pare?

13. Per comprovar l'efecte de l'execució concurrent, i veure que la planificació del sistema genera resultats diferents, executa diverses vegades la comanda myPS3 amb els mateixos paràmetres i guardar la sortida en diferents fitxers. Comprova si l'ordre en què s'executen els ps's és sempre el mateix. Pensa que és possible que diversos resultats siguin iguals.

PREGUNTA 38. Què has fet per guardar la sortida de les execucions de myPS3?

Pas de paràmetres als executables a través de l' execlp

L'objectiu d'aquesta secció és entendre la relació entre els paràmetres rebuts pel main d'un executable i els paràmetres passats a la crida a sistema execlp. Recorda: la rutina main rep dos paràmetres: argc, de tipus enter, i argv de tipus array de strings. El sistema operatiu omple el paràmetre argc amb el nombre d'elements que té argv, i omple argv amb els paràmetres que l'usuari indica en la crida a sistema execlp (que han de ser de tipus string). Per seguir la mateixa convenció que utilitza la Shell, s'ha de fer que el primer paràmetre sigui el nom de l'executable al qual es mutarà.

14. El programa listaParametros que treballem en la sessió 2, simplement mostra per pantalla els paràmetres que rep. Escriu un programa parsExec.c que creï 4 processos fills. Un cop creats, el procés pare només ha d'esperar fins que tots els fills acabin. Cada procés fill només ha de mutar a l'executable listaParametros, cadascun passant uns paràmetres diferents, de manera que, en executar parsExec, a la pantalla s'han de veure els següents missatges (l'ordre dels missatges dona igual i pot ser diferent per a cada execució). Cada grup de missatges correspon amb la sortida d'un procés dels que muten:

```
El argumento 0 es listaParametros
El argumento 1 es a
El argumento 2 es b

Usage:listaParametros arg1 [arg2..argn]
Este programa escribe por su salida la lista de argumentos que recibe

El argumento 0 es listaParametros
El argumento 1 es 25
El argumento 2 es 4


El argumento 0 es listaParametros
El argumento 1 es 1024
El argumento 2 es hola
El argumento 3 es adios
```

15. Modifica el makefile per poder compilar i muntar parsExec.c.

Sessió 4. Comunicació de processos

Preparació prèvia

Objectius

Durant aquesta sessió introduirem la gestió d' esdeveniments entre processos com a mecanisme de comunicació i sincronització entre processos. També es treballaran aspectes relacionats amb la concurrència de processos.

Habilitats

- Ser capaç de reprogramar/esperar/enviar esdeveniments utilitzant la interfície d'UNIX entre processos. Practicarem amb: sigaction/sigsuspend/alarm/kill.
- Ser capaç d'enviar esdeveniments a processos utilitzant la comanda kill.

Coneixements previs

Els signals o esdeveniments poden ser enviats per altres processos o enviats pel sistema automàticament, per exemple quan acaba un procés fill (SIGCHLD) o acaba el temporitzador d'una alarma (SIGALRM).

Cada procés té una taula en el seu PCB on es descriu, per a cada signal, quina acció cal realitzar, que pot ser: **Ignorar l'esdeveniment** (no tots poden ignorar-se), **realitzar l'acció per defecte** que tingui el sistema programada per a aquest esdeveniment, o **executar una funció que el procés hagi definit** explícitament mitjançant la crida a sistema sigaction. Les funcions de tractament de signal han de tenir la capçalera següent:

```
void nom_funcio( int numero_de_signal_rebut );
```

Quan el procés rep un signal, el sistema passa a executar el tractament associat a aquest signal per a aquest procés. En el cas que el tractament sigui una funció definida per l'usuari, la funció rep com a paràmetre el nombre de signal que n' ha provocat l' execució. Això ens permet associar una mateixa funció a diferents tipus de signal i fer un tractament diferenciat dins d'aquesta funció.

Guia per al treball previ

Amb uns exemples veurem, de forma senzilla, com programar un esdeveniment, com enviar-lo, que succeeix amb la taula de programació de signals en fer un fork, etc. Per a aquesta sessió repassa els conceptes explicats en classe de teoria sobre processos i signals. Llegeix les pàgines del man amb el tema que s'indiquen a la següent taula.

Per llegir al man	Descripció bàsica	Paràmetres
sigaction	Reprograma l'acció associada a un esdeveniment concret	
kill (anomenada a sistema)	Envia un esdeveniment concret a un procés	
sigsuspend	Bloqueja el procés que l'executa fins que rep un signal (els signals el tractament dels quals és ser ignorat no desbloquegen el procés)	
sigprocmask	Permet modificar la màscara de signals bloquejats del procés	
alarm	Programa la tramesa d'un signal SIGALRM al cap de N segons	
sleep	Funció de la llibreria de C que bloqueja al procés durant el temps que se li passa com a paràmetre	
/bin/kill (comanda)	Envia un esdeveniment a un procés	**-L**
ps	Mostra informació sobre els processos del sistema	**-o pid,s,cmd,time**
waitpid	Espera la finalització d' un procés	**WNOHANG**

Decarrega el fitxer S4.tar.gz i descomprimeix-lo (tar zxvf S4.tar.gz). Compila els fitxers i executa'ls. Al fitxer README que trobaràs hi ha una petita descripció del que fan i com compilar-los. Intenta entendre'ls i comprendre com es fan servir les crides a sistema que practicarem abans d'anar al laboratori. Els fitxers estan comentats de manera que entenguis el que s'està fent.

Els signals que principalment faremñ servir són SIGALRM (alarma, temporitzador), SIGCHLD (fi d'un procés fill), o SIGUSR1/SIGUSR2 (sense significat predefinit, perquè el programador pugui usar-los amb el significat que li convingui). Consulta les accions per defecte d'aquests signals.

Realitza les següents proves abans d' anar al laboratori. Crea un fitxer anomenat entrega.txt i escriu les respostes a les següents preguntes (indicant el seu número).

Sobre alarm1: recepció de diferents tipus de signals i enviament d'esdeveniments des de la consola

1. Executa alarm1 en una consola i observa el seu comportament. Què passa quan passen 5 segons?

2. Executa de nou alarm1 i abans que acabi el temporitzador envia-li un signal de tipus SIGKILL. Per a això, en un altre terminal executa la comanda ps per obtenir el pid del procés i a continuació utilitza la comanda kill amb l'opció "–KILL" per enviar al procés el signal SIGKILL. El comportament és el mateix que si esperes que arribi el SIGALRM? Rebes un missatge diferent al terminal?

Recorda que el Shell és l'encarregat de crear el procés que executa les comandes que introdueixes i de recollir l'estat de finalització d'aquest procés. El pseudo-codi del Shell per executar una comanda en primer pla és el següent:

```
while (1) {
        comanda = llegir_comanda();
        pid_h = fork();
        if (pid_h == 0)
                execlp(comanda,....);
        }
        waitpid (pid_h, &status, 0);
}
```

3. Quin procés s'encarrega de mostrar els missatges que apareixen en pantalla quan mor el procés alarm1? Amb quina crida a sistema el shell recull l'estat de finalització del procés que executa alarm1 quan aquest acaba?

4. És necessari l'exit que hi ha al codi d'alarm1? S'executa en algun cas?

Sobre alarm2: qualsevol signal es pot enviar des del Shell.

1. Executa alarm2 en un terminal. Obre un altre, esbrina el pid del procés alarm2 i utilitza la comanda kill per enviar-li el signal **"–ALRM"** diverses vegades. Quin comportament observes? ¿El control de temps ha funcionat com pretenia el codi?

2. Es pot modificar el tractament associat a qualsevol signal?

3. Mira en el man (man alarm) el valor de retorn de la crida a sistema alarm i pensa com podríem arreglar el codi per detectar quan un SIGALRM ens arriba sense estar relacionat amb el temporitzador.

Sobre alarm3: la taula de programació de signals s'hereta.

1. Qui rep els SIGALRM: el pare, el fill o tots dos? Com ho has comprovat? Modifica la funció "funcion_alarma" perquè en el missatge que s'escriu aparegui també el pid del procés, de manera que puguem veure fàcilment quin procés rep els signals.

Sobre alarm4: els SIGALRM són rebuts únicament pel procés que els ha generat

1. Quants SIGALRMS programa cada procés? Qui rep cada alarma: El pare, el fill, tots dos? Què li passa al pare? Com ho has comprovat? Modifica el codi de manera que la primera alarma la programi el pare abans de fork (i el fill no), i observa com el fill es queda esperant en la crida sigsuspend.

Sobre ejemplo_waitpid: tramesa de signals mitjançant crida a sistema, gestió de l'estat de finalització dels fills i desactivació del temporitzador.

1. Analitza el codi d' aquest programa i executa'l. Observa que dins del codi dels processos fills s'utilitza un temporitzador per fixar el seu temps d'execució a un segon.

2. Modifica el codi perquè el procés pare en sortir del waitpid mostri en sortida estàndard un missatge descrivint la causa de la mort del fill (ha acabat amb un exit o ha mort perquè ha rebut un signal).

3. Completa el programa per limitar el temps màxim d'espera al waitpid en cada iteració del bucle: si al cap de 2 segons cap fill ha acabat l'execució, el pare haurà d'enviar-li un SIGKILL a tots els seus fills vius. En cas que algun fill acabi a temps, el pare desactivarà el temporitzador i mostrarà un missatge indicant quant temps faltava perquè s'enviés el SIGALRM. Per desactivar el temporitzador es pot utilitzar la crida a sistema alarm passant-li com a paràmetre un 0. NOTA: teniu en compte que per fer això cal guardar tots els pids dels fills creats i anar registrant els fills que ja han mort.

- **Per lliurar: previ04.tar.gz**

```
#tar czfv previ04.tar.gz entrega.txt
```

Bibliografia

- Transparències del Tema 2 (Processos) de SO-grau.
- Capítol 21 (Linux) d'A. Silberschatz, P. Galvin i G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercicis a realitzar al laboratori

- Per a tots els exercicis, s'assumeix que es provaran tots els exercicis que es demanen i es modificarà el makefile perquè es puguin compilar i muntar els executables.
- Contesta en un fitxer de text anomenat entrega.txt totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s'ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 

- **Per lliurar: sessio04.tar.gz**

```
#tar zcfv sessio04.tar.gz entrega.txt makefile ejemplo_alarm2.c
ejemplo_alarm3.c exemple_signal.c exemple_signal2.c
esdeveniments.c esdeveniments2.c signal_perdido2.c.
```

Analitzar l'execució dels processos

Al llarg d'aquesta sessió de laboratori és important entendre pas a pas la recepció, tractament, i altres possibles gestions que veurem relacionades amb els signals. Per això, de vegades és necessari tenir disponible una mica de temps per poder fer aquesta anàlisi. Amb aquesta finalitat farem els següents passos.

En aquells llocs on vulguem aturar temporalment l'execució del procés introduirem el següent tros de codi:

```
…

len = sprintf(buf, "Punt de control X\n");
write(1, buf, len);
kill(getpid(), SIGSTOP);

…
```

La seva funció és: 1) mostrar un missatge per pantalla, per tenir una referència de l'instant d'execució al qual correspon i 2) aturar temporalment l'execució del procés mitjançant un signal SIGSTOP.

El signal SIGSTOP és peculiar, ja que, en SOs Linux, posa al procés en estat "Stopped". És una categoria de l'estat "bloquejat", vist a classe de teoria, i que és diferent de l'estat "Sleeping", que és l'estat al qual passa el procés quan fa una crida a sistema bloquejant. Per reprendre l'execució requereix del signal SIGCONT (NOTA: per defecte aquest signal només reprèn l'execució del procés si està en estat "Stopped", tret que es reprogrami el signal). Mentre no rebi aquest signal, el procés quedarà detingut. Amb això aconseguim poder fer totes les anàlisis que calen durant aquest període de temps.

Sobre alarm2: Detectem quin signal ha arribat

1. Reprograma el signal SIGUSR1 i fes que estigui associat a la mateixa funció que l'alarma. Modifica la funció "funcion_alarma" de manera que actualitzi els segons en cas

que arribi un signal SIGALRM i que escrigui un missatge en cas que arribi SIGUSR1. Hauràs de modificar també la màscara de signals que passes a la funció sigsuspend. Comprova que funciona enviant signals SIGUSR1 des de la consola utilitzant la comanda kill.

NOTA: Recorda que la funció d'atenció al signal rep com a paràmetre el nombre de signal rebut. En el treball previ teniu més detalls.

Sobre alarm3: Signals i creació/mutació de processos

Aquests exercicis estan orientats a que observeu el que hem explicat a classe de teoria relacionat amb la gestió de signals i la creació/mutació de processos: En crear un procés el fill HERETA la taula de signals del seu pare. En mutar un procés, la seva taula de signals es posa per defecte.

2. Modifica el codi perquè la reprogramació del signal SIGALRM (crida a sistema sigaction) només la faci el fill. **OBSERVA** com abans d'aquesta modificació, tant pare com fill (per mitjà de l'herència) tenien capturat el SIGALRM. Després del canvi, la modificació del fill és privada, per la qual cosa el pare té associat a SIGALRM l'acció per defecte.

PREGUNTA 39. Què li passa al pare ara quan li arriba l'esdeveniment SIGALRM?

3. Modifica el codi perquè després de programar el temporitzador el procés fill muti per executar un altre programa. Aquest programa ha de durar més de 10 segons perquè rebi el signal SIGALRM abans d'acabar (per exemple, pots implementar un programa que només executi un bucle infinit). **OBSERVA** com en mutar l'acció associada a SIGALRM es posa per defecte, ja que la taula de signals es reseteja.

PREGUNTA 40. Què passa amb la taula de tractaments de signals si fem un execlp (i canviem el codi)? Es manté la reprogramació de signals? Es posa per defecte?

Sobre ejemplo_waitpid: Esperem que acabin els fills. Impacte de la implementació dels signals

PREGUNTA 41. El programa ejemplo_waitpid, És seqüencial o concurrent?

La crida a sistema waitpid normalment s'utilitza de forma bloquejant. El procés que l'executa no continua fins que algun procés fill acaba. Si desitgem controlar la finalització dels processos fills sense bloquejar el pare, podem capturar el signal SIGCHLD i esperar que vagi arribant. En aquest cas, en ser un cas una mica forçat, haurem de realitzar una espera activa en el procés ja que d'una altra manera acabarà la seva execució abans que acabin els processos fills.

4. Crea una còpia d'aquest programa anomenada exemple_signal.c. Modifica aquest programa perquè el procés pare no es bloquegi en el main a l'espera dels fills, sinó que executi la crida waitpid en la funció d'atenció al SIGCHLD (notificació que un fill ha acabat). Això li permet continuar amb la seva execució normalment. Recorda que en aquest cas has de fer que el pare faci una espera activa per no acabar la seva execució. El normal seria utilitzar aquesta opció en cas que el pare tingui alguna cosa per executar. Pots utilitzar una variable global que et digui si encara queden fills vius.

COMPTE!: Mentre s'està executant el tractament d'un signal, el sistema bloqueja la notificació de nous signals del mateix tipus, i el procés els rebrà en sortir de la rutina.

Però, el sistema operatiu només és capaç de recordar un signal pendent de cada tipus. És a dir, si mentre s'està executant el tractament d'un signal el procés rep més, en sortir de la rutina només es tractarà un de cada tipus i la resta es perdran. Per aquest motiu, la rutina de tractament de SIGCHLD ha d'assegurar que en sortir-ne s'ha esperat tots els fills que hagin mort durant el tractament del signal però sense bloquejar-se (recorda que dins d'una rutina de tractament de signal mai has de bloquejar-te). Consulta el significat de l'opció WNOHANG de la crida a sistema waitpid (man waitpid) i utilitza-la perquè waitpid sigui no bloquejant.

5. Crea una còpia de ejemplo_signal.c amb el nom exemple_signal2.c. Modifica aquest programa perquè el pare, una vegada creats tots els processos fills, els hi enviï a tots ells el signal SIGUSR1 (l'acció del qual per defecte és acabar) i després continuï incrementant la variable comptador. A més, després d'executar la crida waitpid, el pare mostrarà el PID del fill més el motiu pel qual aquest procés ha acabat (macros WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG). Quin valor de finalització presenta cada procés fill?

    - PISTA: Tingueu en compte que la crida sigsuspend() bloqueja el procés fins que arribi un signal que el procés tingui capturat, però si el signal arriba abans que el procés executi el sigsuspend(), el procés podria bloquejar-se indefinidament. Hauràs de modificar la màscara de signals del sigsuspend perquè capturi el SIGUSR1.
    - Crea una funció que, utilitzant les macros anteriors, escrigui el PID del procés que ha acabat i un missatge indicant si ha acabat per un exit o un signal i en cada cas el exit_status o signal_code. Teniu un exemple de com usar aquestes macros en les transparències del T2.

Protecció entre processos

Els processos no poden enviar signals a processos d' altres usuaris. Per provar-ho iniciarem sessió amb diversos usuaris. En els "laboratoris de sistemes" de la facultat hi ha diversos usuaris creats (*so1*, *so2*, *so3*, *so4* i *so5*) el password de tots és "sistemes" (el mateix per a la imatge d'Ubuntu que ofereix LCFIB per usar com a màquina virtual). En els "laboratoris generals" de la facultat disposem de l'usuari *examen* amb el password "examen22". La comanda *su* permet connectar-se com un altre usuari (sense iniciar una nova sessió).

6. Executa el programa alarm2 en un terminal i comprova el seu PID. Obre una nova sessió i canvia a un altre usuari (executa #su so1 o #su examen). Intenta enviar signals al procés que està executant alarm2 des de la sessió iniciada amb l'altre usuari.

PREGUNTA 42. El segon usuari pot enviar signals al procés llançat per l'usuari inicial?, quin error dona?

Gestió de Signals

L'objectiu d'aquesta secció és que siguis capaç d'implementar per complet un programa que gestioni diversos tipus de signals utilitzant un tractament diferent al de per defecte.

7. Fes un programa, anomenat esdeveniments.c, que executi un bucle infinit i que tingui una variable global per emmagatzemar el temps que porta el procés executant-se (en segons). Aquesta variable es gestiona mitjançant signals de la manera següent:
    - Cada segon s'incrementa
    - Si el procés rep un SIGUSR1 la variable es posarà a zero
    - Si el procés rep un SIGUSR2 escriurà per pantalla el valor actual del comptador

Fes que tots els signals del programa siguin atesos per la mateixa funció. Envia els signals des de la línia de comandes i comprova que funciona correctament.

Comportament per defecte

La reprogramació d'un signal a Linux es manté durant tota la vida del procés. Per aquesta raó, de vegades, cal forçar el comportament per defecte dels signals en el cas que no ens interessi processar més esdeveniments.

8. Crea una còpia d' esdeveniments.c amb el nom esdeveniments2.c. Modifica el codi d'esdeveniments2.c perquè la segona vegada que rebi el mateix signal s'executi el comportament per defecte d' aquest signal. PISTA: consulta a la pàgina del manual del sigaction la constant SA_RESETHAND.

PREGUNTA 43. Quin missatge mostra el Shell quan s'envia per segona vegada el mateix signal?

Tractament de signals imbricat

Quan configurem la reprogramació d'un signal elegim com volem actuar davant la recepció d'un altre signal durant l' execució del nostre codi de tractament del signal reprogramat. D'una banda, podem seqüencialitzar el tractament, per la qual cosa no es tracta fins que acabi l'execució del nostre codi. Simplement hem d'incloure en la màscara "sa_mask", utilitzada per a la reprogramació del signal, tots aquells que volem bloquejar durant el tractament (és a dir, seqüencialitzar). D' altra banda, podem imbricar el tractament, per la qual cosa comença a tractar-se immediatament el tractament de l' altre signal. En aquest cas, aquest comportament el tindran aquells signals que no estan en la "sa_mask" utilitzada per a la reprogramació del signal.

1. Crea una còpia d' esdeveniments.c amb el nom esdeveniments3.c. Modifica el codi d'esdeveniments perquè la reprogramanció del signal SIGUSR1 sigui: mostrar un missatge (per exemple, "inici tractament") per pantalla; enviar-se a si mateix un SIGINT; mostrar un missatge (per exemple, "fi tractament") per pantalla. Ara, fes una prova amb el signal SIGINT en la "sa_mask" de la reprogramació de SIGUSR1 i una altra prova sense que estigui inclosa. En ambdues proves envia des de terminal un SIGUSR1 al procés per veure quin és el comportament.

PREGUNTA 44. Explica quines diferències veus en els missatges que apareixen per pantalla que mostren els dos experiments i quin és el motiu

Riscos de la concurrència

Quan es programen aplicacions amb diversos processos concurrents no es pot assumir res sobre l' ordre d' execució de les instruccions dels diferents processos. Això també aplica a l'enviament i recepció de signals: no podem assumir que un procés rebrà un signal en un moment determinat. Això s'ha de tenir en compte en utilitzar la crida sigsuspend.

El programa signal_perdido.c mostra aquesta problemàtica de forma artificial. Aquest programa crea un procés que pretén esperar en una crida sigsuspend la recepció d'un signal. El procés pare és l'encarregat d'enviar-li aquest signal. El moment en què s'envia el signal depèn del paràmetre del programa: es fa immediatament (valor de paràmetre 0) o es retarda un temps (valor de paràmetre 1).

2. Executa el programa signal_perdido passant com a paràmetre 1. Anota en el fitxer entrega.txt quin resultat observes. A continuació executa de nou el programa passant com a paràmetre 0. Anota de nou en respuestas.txt el resultat que observes.

PREGUNTA 45. Explica a què es deu el resultat de les execucions de signal_perdido amb paràmetre 1 i amb paràmetre 0

3. Modifica el programa signal_perdido (anomena'l signal_perdido2.c) perquè, abans de realitzar l'espera, bloquegi provisionalment la recepció del signal utilitzant la crida sigprocmask, de manera que s'eviti la pèrdua del signal. Recorda desbloquejar el signal després de l'espera.

Sessió 5. Gestió de Memòria

Preparació prèvia

Objectius

- Comprendre la relació entre el codi generat per l'usuari, l'espai lògic del procés que executarà aquest programa i l'espai de memòria física ocupat pel procés.
- Entendre la diferència entre enllaçar un executable amb llibreries estàtiques o dinàmiques.
- Entendre el funcionament d'algunes comandes bàsiques que permeten analitzar l'ús de la memòria que fan els processos.
- Entendre el comportament de funcions de la llibreria de C i de crides a sistema simples que permeten modificar l'espai d'adreces lògic dels processos en temps d'execució (memòria dinàmica).

Habilitats

- Ser capaç de relacionar parts d'un binari amb el seu espai d'adreces lògic en memòria.
- Saber distingir les diferents regions de memòria i en quina regió s' ubica cada element d'un procés.
- Entendre l'efecte de malloc/free/mmap/munmap/sbrk sobre l'espai d'adreces.
- Saber com reservar i alliberar memòria dinàmica i les seves implicacions en el sistema.
- Ser capaços de detectar i corregir errors en l'ús de memòria d'un codi.

Coneixements previs

- Entendre el format bàsic d' un executable.
- Programació en C: ús de punters.
- Saber interpretar i consultar la informació disponible sobre el sistema i els processos en el directori /proc.

Guia per al treball previ

- Abans de la sessió, consulteu el man (`man nom_comanda`) de les següents comandes. En concret, per a cada comanda heu de llegir i entendre perfectament: la SYNOPSIS, la DESCRIPTION i les opcions que us comentem a la columna "Paràmetres" de la taula.

Per llegir al man	**Descripció bàsica**	Paràmetres
`gcc`	Compilador de C	**`-static`**
`nm`	comanda que mostra la taula de símbols del programa (variables globals i funcions)	
`objdump`	comanda que mostra informació sobre el fitxer objecte	**`-d`**
`/proc`	Conté informació sobre el sistema i els processos en execució	**`/proc/[pid]/maps`**
`pmap`	comanda que mostra el mapa de memòria d'un procés	**`-x`**
`malloc`	Funció de la llibreria de C que valida una regió de memòria lògica	
`free`	Funció de la llibreria de C que allibera una regió de memòria lògica	
`sbrk`	Crida a sistema que modifica la mida de la regió de dades (dinàmiques)	
`mmap`	Crida a sistema que mapeja una nova regió en l'espai lògic del procés	**`MAP_SHARED`, `MAP_PRIVATE`, `MAP_ANONYMOUS`**
`munmap`	Crida a sistema que elimina la regió especificada de l'espai lògic del procés	

- A la pàgina web de l'assignatura (http://docencia.ac.upc.edu/FIB/grau/SO) teniu el fitxer *S5.tar.gz* que conté tots els fitxers font que utilitzareu en aquesta sessió. Crea un directorio en la teva màquina, copia en ell el fitxer *S5.tar.gz* i descomprimeix-lo (`tar zxfv S5.tar.gz`).
- Crea un fitxer de text anomenat *previ.txt* i hi contesta les següents preguntes.
- Practica l'ús de *nm* i *objdump* amb els següents exercicis aplicats al següent codi (pots trobar el codi complet en el fitxer *mem1_previo.c*):

```
const int cons1 = 1;
int resp = 0xDEAD2BAD;
int argn;

void proth (int k, int n, int *X) {
    *X = (k * (cons1 << n)) + cons1;
}

int main (int argc, char *argv[]) {
    int argk;
    char buff[80];

    argk = atoi(argv[1]);
    argn = atoi(argv[2]);
    proth(argk, argn, &resp);
    sprintf(buff, "El número de Proth de %d i %d es %d\n", argk, argn, resp);
    write(1, buff, strlen(buff));
}
```

- Utilitza la comanda *nm* sobre l'executable *mem1_previo* i apunta en el fitxer "*previ.txt*" la direcció assignada a les variables cons1, resp, argn i argk del programa. **És possible saber la direcció de totes aquestes variables amb la comanda *nm*? Per què?** Indica també en quina secció de l'executable estan reservades aquestes variables (busca en el man els tipus de símbols que ens mostra *nm*. Per exemple, els símbols etiquetats amb una "D" significa que són a la secció de dades). Consulta també la direcció i la secció assignada a la funció proth (i anota-la a "*previ.txt*").
- **Modifica el programa anterior (anomena'l *mem1_previo_v2.c*)** perquè la variable resp definida com a punter a enter (int *resp) i assigna-li una direcció vàlida abans de cridar a la funció proth. Per això utilitza la funció de llibreria *malloc* per reservar prou espai per emmagatzemar un enter i assigna la direcció d'aquesta regió a la variable. Adapta la resta del codi a aquest canvi perquè continuï funcionant.
- El programa *mem2_previo.c* llegeix de teclat el nombre d'elements d'un vector d'enters, inicialitza dos vectors amb aquesta mida (un amb els enters parells i un altre amb els senars) i a continuació suma els elements de cada vector. **Modifica aquest programa (llàmal *mem2_previo_v2.c*)** perquè, en lloc d'usar dos vectors estàtics, usi memòria dinàmica. Per a això declara les variables *vector1* i *vector2* com a punters a enter. Després de llegir per teclat el nombre d'elements que ha de tenir cada vector, el programa ha d'utilitzar la funció de llibreria *malloc* per reservar una regió de memòria en la qual hi càpiguen aquests elements i assignar la direcció d'aquesta regió a la variable *vector1, i* la crida a sistema *mmap* per fer l'equivalent per a la variable *vector2* (en aquest cas, la regió de memòria ha de ser anònima i privada).
- Utilitza la comanda per compilar els fitxers *mem2_previo.c* i *mem2_previo_v2.c* enllaçant-los amb llibreries estàtiques. **Indica en el fitxer "*previ.txt*" quina opció del *gcc* has utilitzat**. Per a tots dos programes fes el següent: executa el programa i abans de polsar *Return* perquè acabi, des d'un altre terminal, accedeix al directori */proc/PID_del_proceso* que conté la informació sobre el procés i observa el fitxer *maps*. Aquest fitxer conté una línia per a cada regió de memòria reservada. La primera columna ens indica la direcció inicial i final de la regió (en hexadecimal). La diferència entre ambdós valors ens dóna la mida de la regió. Cerca a la pàgina del man per a proc el format de la sortida del fitxer *maps* i el significat de la resta de camps. **Anota en el fitxer**

"*previ.txt*" la mida total de les regions de memòria on estan guardats els vectors (heap, bss+dades, o mapeig anònim) per als següents números d'elements dels vectors: 10, 10000 i 40000. La regió del heap està etiquetada com a [heap], però la regió de bss i les regions anònimes no estan etiquetades, i la regió de dades és una de les etiquetades amb el nom de l'executable (n'hi ha diverses). Hauràs de deduir quines són pels permisos de la regió i per la direcció dels símbols *vector1* i *vector2*. Hi ha alguna diferència entre els diferents valors de les execucions d'ambdós programes?

- El fitxer *mem3_previo.c* conté un codi que té un error en l'ús d'un punter. Executa el programa i comprova quin error apareix. **Modifica el codi (en un nou fitxer *mem3_previo_v2.c*)** perquè quan el programa generi un signal de tipus SIGSEGV (*segmentation fault*), la rutina d'atenció al signal mostri un missatge d'error per pantalla i acabi l'execució del programa.
- **Per lliurar: previ05.tar.gz**

```
#tar zcfv previ05.tar.gz previ.txt mem1_previo_v2.c
mem2_previo_v2.c mem3_previo_v2.c
```

Bibliografia

- Capítol 8 (Main Memory) d'A. Silberschatz, P. Galvin i G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercicis a realitzar al laboratori

- Per a cada pregunta en què es crea un nou fitxer de codi s'ha de modificar el Makefile perquè el compili i munti l'executable.

- Contesta en un fitxer de text anomenat "*entrega.txt"* totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s' ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol:

- **Per lliurar: sessio05.tar.gz**

```
#tar zcfv sessio05.tar.gz entrega.txt Makefile mem1_v2.c mem2_v2.c
mem2_v3.c mem3_v2.c
```

Espai d'adreces d'un procés i compilació estàtica i dinàmica

L'objectiu d'aquesta secció és doble: d'una banda, entendre com s'organitzen les dades i el codi d'un programa en l'espai d'adreces del procés; d'altra banda, entendre com influeix tant en l'espai d'adreces el tipus de compilació utilitzat (estàtica o dinàmica). Per a això, farem servir comandes que permeten analitzar els executables i observarem la informació sobre l'espai d'adreces guardada en el /proc.

El fitxer *mem1.c* conté un programa que estén la funcionalitat del programa *mem1_previo.c* utilitzat en el treball previ afegint reserves de memòria dinàmica durant l'execució. Analitza el seu contingut abans de respondre a les següents preguntes.

1. Compila el programa enllaçant amb les llibreries dinàmiques del sistema (compilació per defecte) i guarda l'executable amb el nom *mem1_dynamic*. Compila ara el programa, enllaçant amb les llibreries estàtiques del sistema, i guarda l'executable amb el nom *mem1_static*. Compara ara els executables *mem1_dynamic* i *mem1_static*.
    a. Fent servir la comanda *nm* per veure els símbols definits en cada executable.
    b. Fent servir la comanda *objdump* amb l'opció *–d* per veure el codi traduït.
    c. Comparant les mides dels executables resultants.

PREGUNTA 46. Per als dos executables (compilat estàtic i dinàmic) observa la seva mida i la sortida de les comandes *nm* i *objdump*. En què es diferencien?

2. Executa les dues versions de l'executable passant-li els paràmetres ['1' i '3'] i abans de polsar *Return* perquè el programa acabi, des d'un altre terminal, compara el contingut del fitxer *maps* del /proc per a cadascun dels processos (també pots fer servir la comanda *pmap* comparar el mapa de memòria dels processos).

PREGUNTA 47. Observa el contingut del fitxer *maps* del /proc per a cada procés i anota per a cada regió: adreça inicial, adreça final i etiqueta associada. Quina diferència hi ha entre el *maps* de cada procés?

PREGUNTA 48. A quina regió de les descrites en el maps pertany cada variable (`ptr`, `cons1`, `resp`, `argn` i `argk`) i cada regió reservada amb *malloc*? Anota l'adreça inicial, l'adreça final i el nom de la regió i relaciona els resultats amb les seccions de cada programa detallades per la comanda *nm*.

Gestió de la memòria dinàmica

L'objectiu d'aquesta secció és entendre com afecta l'espai d'adreces d'un procés la memòria dinàmica reservada en temps d'execució mitjançant l'interfície *malloc/free*. *Malloc/free* són les funcions de la llibreria de C, que ofereixen una gestió molt més sofisticada de la memòria dinàmica. La llibreria de C reserva una zona gran de heap i la utilitza per anar gestionant les peticions de l'usuari sense haver de modificar la mida del heap. Per a això, utilitza estructures de dades (també emmagatzemades en el heap) que serveixen per anotar quins trossos estan ocupats i quins trossos estan lliures.

3. Executa el programa *mem1_static* passant-li els paràmetres ['1' i '1'], ['1' i '3'], i ['3' i '4'] (són 3 execucions). Observa el *maps* del /proc per comparar la mida del heap en funció del nombre d'invocacions a *malloc*. Analitza els resultats en funció de l'explicat a classe sobre el funcionament de *malloc*.

PREGUNTA 49. Per a cada execució anota el nombre d'invocacions a *malloc* i les adreces inicial i final del heap que mostren el fitxer *maps* del /proc. Canvia la mida del heap segons el paràmetre d'entrada? Per què?

4. Crea una còpia del fitxer *mem1.c* anomenada *mem1_v2.c*. Allibera la memòria al final de cada iteració del bucle de reserva de regions en *mem1_v2.c* i torna a executar el programa, amb paràmetres ['3' i '4'], observant el fitxer *maps* del /proc i la mida de la zona que conté les regions reservades amb *malloc*.

PREGUNTA 50. Quina és la mida del heap en aquest cas? Explica els resultats.

Herència/compartició de la memòria dinàmica

L'objectiu d'aquesta secció és entendre l'herència/compartició de la memòria dinàmica en la creació de processos depenent de l'interfície que s'utilitzi: *malloc* o *mmap*. Executa el programa *mem2.c* i respon a les següents preguntes.

PREGUNTA 51. Quin és el valor que veu el fill en la variable al principi de la seva execució? Com justifiques aquest comportament?

5. Modifica aquest programa (anomena'l *mem2_v2.c*) perquè, en lloc d'utilitzar la funció de llibreria *malloc,* el procés pare usi la crida a sistema *mmap* per reservar una regió de memòria dinàmica anònima i privada.

PREGUNTA 52. A quina regió de les descrites en el *maps* pertany la regió reservada amb *mmap*? Indica l'adreça inicial i l'adreça final. Quin és el valor que veu el pare en la variable al final de l'execució? Com justifiques aquest comportament?

6. Modifica el programa anterior (anomena'l *mem2_v3.c*) perquè la regió de memòria dinàmica reservada amb *mmap* sigui ara anònima i compartida.

PREGUNTA 53. Quin és el valor que veu el pare en la variable al final de l'execució? Com justifiques aquest comportament?

PREGUNTA 54. Compara el contingut del fitxer *maps* del /proc per als programes *mem2_v2* i *mem2_v3*. En què es diferencien?

Accessos incorrectes

El fitxer *mem3.c* conté un codi que té un error en l'ús d'un punter (diferent de l'error que hi havia al fitxer *mem3_previo.c* vist en el treball previ). Executa el programa i comprova que realment no funciona.

7. Modifica el codi (anomena'l *mem3_v2.c*) perquè quan el procés rebi un signal de tipus SIGSEGV:

    a. Imprimeixi a la sortida estàndard un missatge indicant: i) l'adreça de la variable p; ii) el valor de la variable p (adreça a la qual apunta el punter); iii) l'adreça on finalitza el heap del procés (aquesta adreça ja no és vàlida) (mira en el man com aconseguir aquest valor amb la crida a sistema *sbrk*).

    NOTA: Imprimeix només els 3 valors que us demanem, si copies directament el write del codi pots estar repetint el mateix error que provoca el signal.

    b. Si és la primera vegada que rep el signal, utilitza la crida a sistema *sbrk* per reservar memòria addicional al heap de manera que es puguin guardar tants enters addicionals com es van guardar abans que es generés el signal. Recorda que *sbrk* simplement augmenta o redueix la mida del heap en X bytes (el que demani l'usuari). Si arriba un altre signal SIGSEGV, el programa acabarà la seva execució després d'escriure el missatge.

PREGUNTA 55. Quin error conté el codi del programa? Per què el programa no falla en les primeres iteracions?

Sessió 6. Gestió d'Entrada/Sortida (I)

Preparació prèvia

Objectius

- Entendre el concepte de independència de dispositius
- Entendre els mecanismes que ofereix la shell per a la redirecció i comunicació de processos

Habilitats

- Ser capaç d'aplicar els avantatges de la independència de dispositius
- Saber redireccionar l'entrada i la sortida d'un procés des de la shell
- Saber comunicar dos comandes a través de pipes sense nom des de la shell

Guia per al treball previ

- Abans de la sessió, consulteu el man (`man nom_comanda`) de lrs següents comandes. En concret, per a cada comanda heu de llegir i entendre perfectament: la SYNOPSIS, la DESCRIPTION i les opcions que us comentem a la columna "Paràmetres" de la taula.

Per llegir al man	Descripció bàsica	Paràmetres
`open`	Obre un fitxer o dispositiu	
`write`	Crida a sistema per escriure en un dispositiu virtual	
`read`	Crida a sistema per llegir d'un dispositiu virtual	
`grep`	comanda que busca patrons en un fitxer o a la seva entrada estàndard si no se li passa fitxer com a paràmetre	**`-c`**
`ps`	comanda que mostra informació sobre els processos en execució	**`-e, -o`**
`strace`	Llista les crides a sistema executades per un procés	**`-e, -c`**

- A la pàgina web de l'assignatura (http://docencia.ac.upc.edu/FIB/grau/SO) teniu el fitxer S6.tar.gz que conté tots els fitxers font que utilitzareu en aquesta sessió. Crea un directori a la teva màquina, copia en ell el fitxer S6.tar.gz i descomprimeix-lo.

- Contesta a les següents preguntes en el fitxer "previ.txt".

Redirecció d'entrada/sortida, ús dels dispositius lògics terminal i pipe

El fitxer es1.c conté un programa que llegeix de l'entrada estàndard caràcter a caràcter i escriu el llegit a la sortida estàndard. El procés acaba quan la lectura indica que no queden dades per llegir. Compila el programa i, a continuació, executa'l de les següents maneres per veure com es comporta en funció dels dispositius associats als canals estàndard del procés:

1. Introdueix dades per teclat per veure com es copien en pantalla. Per indicar que no queden dades polsa ^D (Control+D), que és l'equivalent a final de fitxer en la lectura de teclat. **Quin valor retorna la crida read després de polsar el ^D?**
2. Crea un fitxer amb un editor de text qualsevol i llança el programa ./es1 associant mitjançant la shell la seva entrada estàndard a aquest fitxer. Recorda (veure Sessió 1) que és possible redireccionar l'entrada (o la sortida) estàndard d'una comanda a un fitxer utilitzant el caràcter especial de la shell < (o > per a la sortida). **Apunta la comanda utilitzada al fitxer "previ.txt".**

Els Shell de Linux permeten que dos comandes intercanviïn dades utilitzant una pipe sense nom (representada pel caràcter '|' a la shell). La seqüència de comandes connectades mitjançant pipes s' anomena pipeline. Per exemple, l'execució del pipeline:

```
#comanda1 | comanda2
```

fa que el Shell creï dos processos (que executen comanda1 i comanda2 respectivament) i els connecti mitjançant una pipe sense nom. Aquest pipeline redirecciona la sortida estàndard del procés que executa la comanda1, associnant-la amb l'extrem d'escriptura d'aquesta pipe, i redirecciona l'entrada estàndard del procés que executa la comanda 2, associant-la amb l'extrem de lectura de la mateixa pipe. D'aquesta manera, tot el que el procés comanda1 escrigui en la seva sortida estàndard serà rebut pel procés comanda2 quan llegeixi de la seva entrada estàndard.

Per exemple, en el directori on has descomprimit el fitxer de la sessió, executa el pipeline:

```
#ls -l |grep es
```

Quin és el resultat? Quina operació realitza la comanda 'grep es'?

Enllaçar les dos comandes mitjançant la pipe és similar a realitzar la combinació següent:

```
#ls -l > salida_ls
#grep es < salida_ls
#rm salida_ls
```

3. Executa un pipeline que mostri en la sortida estàndard el PID, l'usuari i el nom de tots els processos bash que s'estan executant en el sistema. Per a això utilitza les comandes ps i grep combinades amb una pipe. **Anota la comanda al fitxer "previ.txt".**

Format de sortida

A Linux, la interfície d'entrada/sortida està dissenyada per a l'intercanvi de bytes **sense interpretar el contingut de la informació**.

És a dir, el sistema operatiu es limita a transferir el nombre de bytes que se l'indica a partir de la direcció de memòria que se l'indica, i és responsabilitat del programador interpretar correctament aquests bytes, emmagatzemant-los en les estructures de dades que l'interessi en cada moment. A l'hora de recuperar una dada que s'ha guardat en un fitxer, el programador haurà de tenir en compte el format en què s'ha guardat.

Per exemple, si volem llegir un número de la consola (que és un dispositiu que només accepta ASCII, tant d'entrada com de sortida), primer caldrà llegir els caràcters i després convertir-lo a número. Aquí teniu un exemple suposant que l'usuari escriu el número i després prem Ctrl-D.

```
char buffer[64];
int num,i =0;
//Quan l'usuari premi CtrlD el read retornarà 0
//Com que no coneixem quantes xifres té el número, cal llegir-lo amb un bucle
while (read(0,&buffer[i],1)>0) i++;
buffer[i]='\0';
num = atoi( buffer);
```

Per exemple, si volem escriure l'enter *10562* fent servir la seva representació interna a la màquina, estarem escrivint el nombre de bytes que ocupa un enter (4 bytes si fem servir el tipus *int* en una màquina de 32 bits).

```
int num = 10562;
write(1,&num,sizeof(int)); //Si el canal 1 és la consola, veurem escombraries, ja
que només accepta ascii.
```

Per recuperar- i interpretar correctament el seu valor haurem de llegir aquest mateix nombre de bytes i guardar-lo en una variable de tipus enter.

```
//Exemple de lectura (sense control d'errors), suposant que al fitxer dades.txt hi
ha enters faríem...
int fd, num;
fd = open("dades.txt",O_RDONLY);
read(fd,&num,sizeof(int));  //En aquest cas el nombre de bytes a llegir és fix
```

Si per contra volem escriure el mateix nombre com una cadena de caràcters (per exemple, per mostrar-lo per pantalla), el primer pas és convertir-lo a una cadena de caràcters, en la qual cada caràcter representa un dígit. Això implica que estarem fent servir tants bytes com dígits tingui el número (en aquest exemple, 5 bytes).

```
char buff[64];
int num = 10562;
sprintf(buff,"%d",num);
write(1,buff,strlen(buff));
```

4. El fitxer es7.c és un programa que escriu a la sortida estàndard un enter usant la representació interna de la màquina. Compila'l i executa'l redireccionant la seva sortida estàndard a un fitxer:

   ```
   #./es7 > foo.txt
   ```

   Escriu un programa es7_lector.c que en executar-lo de la següent manera:

   ```
   #./es7_lector < foo.txt
   ```

   sigui capaç de llegir i interpretar correctament el contingut d' aquest fitxer.

5. En el cas del dispositiu lògic **terminal (o consola)**, el **device driver que el gestiona interpreta tots els bytes que es volen escriure com a codis ASCII**, mostrant el caràcter corresponent. El fitxer es8.c és un programa que escriu dues vegades un número per sortida estàndard: una usant la representació interna de la màquina i una altra convertint abans el número a string. Compila0'l i executa'l. **Quants bytes**

s'escriuen en cada cas? Quines diferències observes en el que apareix en pantalla?

- **Per lliurar: previ6.tar.gz**

```
#tar zcfv previ06.tar.gz es7_lector.c previ.txt
```

Bibliografia

- Transparències del Tema 4 (Entrada/Sortida) de SO-grau
- Capítol 13 (I/O Systems) d'A. Silberschatz, P. Galvin i G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercicis a realitzar al laboratori

➢ Contesta en un fitxer de text anomenat entrega.txt totes les preguntes que apareixen en el document, indicant per a cada pregunta el seu número i la teva resposta. Aquest document s' ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 

➢ **Per lliurar: sessio06.tar.gz**

```
#tar zcfv sessio06.tar.gz entrega.txt es1_v2.c es6_v2.c
```

Redireccionament i buffering

En aquest primer exercici treballarem amb el fitxer es1.c vist en el treball previ. Com ja s'ha comentat, aquest fitxer conté un programa que llegeix de l'entrada estàndard caràcter a caràcter i escriu el llegit a la sortida estàndard. A continuació, realitza els exercicis següents:

1. Executa la comanda ps des d'un terminal. La columna *TTY* de la sortida del ps et dirà quin fitxer dins del directori /dev representa el terminal associat al shell que tens en execució. Obre un nou terminal i executa de nou la comanda ps. Observa ara que el fitxer que representa el terminal és diferent.
2. Executa, des del segon terminal, el programa es1 redireccionant la seva sortida estàndard per associar-la amb el fitxer que representa el primer terminal. Observa com el que s'escriu en el segon terminal apareix en el primer.
3. Escriu una comanda a la Shell que llanci dos processos que executin el programa es1 que estiguin connectats mitjançant una pipe sense nom. Introdueix uns quants caràcters mitjançant el teclat i prem ^D per finalitzar l'execució dels processos.
4. Crea una còpia del programa es1.c anomenant-la es1_v2.c. Modifica el programa es1_v2.c perquè, en comptes de llegir i escriure els caràcters d'un en un, ho faci utilitzant un búfer (char buffer[256]).
5. La comanda *strace* executa el programa que se li passa com a paràmetre i mostra informació sobre la seqüència de crides a sistema que realitza. Amb l'opció *–i* se li especifica que mostri informació sobre una única crida a sistema i amb l'opció *–o* se li especifica que guardi aquesta informació en un fitxer.

   Volem comparar el nombre de crides a sistema *read* que executen les dues versions del programa (es1 i es1_v2). Per a això executa els següents comandes:

```
#strace -o salida_v2 -e read ./es1_v2 < es2.c

#strace -o salida_v1 -e read ./es1 < es2.c
```

PREGUNTA 56. Apunta en el fitxer "entrega.txt" les comandes que has utilitzat en cada apartat. A més, lliura el fitxer "es1_v2.c" Quines diferències observes en les dues execucions de strace? Quantes crides a sistema read executa cada versió? Quina influència pot tenir sobre el rendiment d'ambdues versions de codi? Per què?

Format de sortida

Analitza en detall el codi dels fitxers es2.c, es3.c i es4.c i assegura't d'entendre el que fan abans de continuar. A continuació, utilitza la comanda make per compilar-los.

1. Executa dues vegades el programa es2, primer posant el primer paràmetre a 0 i després a 1 (utilitza el valor que vulguis per al segon paràmetre). Redirecciona també la sortida estàndard del procés per associar-la a dos fitxers diferents. Observa el contingut dels dos fitxers generats.

PREGUNTA 57. Explica les diferències observades en la sortida del programa quan el primer paràmetre val 0 o 1. Per a què creus que serveix aquest paràmetre?

2. Executa dues vegades el programa es3 redireccionant la seva entrada estàndard en cada execució per associar-la a cadascun dels fitxers generats a l'apartat anterior.

PREGUNTA 58. Explica el motiu dels resultats observats depenent del format fitxer d'entrada.

3. Executa ara dues vegades el programa es4 de la mateixa manera que has executat el programa es3 en l'apartat anterior.

PREGUNTA 59. Explica les diferències observades entre la sortida del programa es3 i es4. Per què la sortida és intel·ligible per a un dels fitxers d'entrada i no per a l'altre?

Cicle de vida

Analitza el contingut dels fitxers es5.c i es1.c, i assegura't d'entendre'n el funcionament.

1. Compila els dos programes i executa cadascun d'ells en un shell diferent. A continuació executa la següent comanda:

```
#./showCpuTime.sh ./es5 ./es1
```

showCpuTime.sh és un script que mostra el temps de consum de CPU de cadascun dels programes passats com a paràmetre cada cert temps (cada 2 segons).

2. Quan acabi l'script, mata els 2 processos es5 i es1.

PREGUNTA 60. Escriu els valors que ha mostrat l'script showCpuTime.sh per a cadascun dels processos i comenta les diferències entre ells pel que fa al consum de CPU. A què es deuen? Identifica les línies de codi de marquen la diferència entre ells

Modifiquem la gestió de signals per part del kernel desactivant el flag SA_RESTART

En les sessions anteriors, havíem activat sempre el flag SA_RESTART en la crida sigaction. D'aquesta manera, quan un procés està bloquejat en una entrada/sortida i es rep un signal, el procés es desbloqueja, es gestiona el signal i, si cal, es continua amb l'operació d'entrada/sortida de forma transparent a l'usuari. Tanmateix, aquest no és el funcionament estàndard d'UNIX, on l'entrada/sortida no continua automàticament.

Crea una còpia del fitxer es6.c anomenant-la es6_v2.c. Modifica el programa es6_v2.c per reprogramar la gestió del signal SIGINT i que mostri un missatge per sortida estàndard informant que s' ha rebut el signal. Utilitza la crida a sistema sigaction sense el flag SA_RESTART perquè la gestió sigui com en UNIX. En aquest cas, el read retorna error en rebre aquest signal. Modifica el programa principal perquè després del read es mostri un missatge en sortida estàndard indicant el resultat de l'operació: read correcte, read amb error (diferent d'interrupció per signal), o read interromput per signal. Consulta en el man els diferents valors d'**errno** per a aquests casos.

Fes la següent seqüència d'execucions per comprovar el bon funcionament del teu codi:

a) Executa el programa i prem return per desbloquejar el *read*.
b) A continuació, executa el programa, però mentre està esperant en el *read* envia-li el signal SIGINT prement ^C.

PREGUNTA 61. Anota en el fitxer entrega.txt el resultat d'ambdues execucions. Lliura el codi programat al fitxer es6_v2.c

PREGUNTA 62. Què passaria si activèssim el flag SA_RESTART al sigaction? Repeteix les execucions anteriors activant el flag SA_RESTART i anota el resultat en el fitxer entrega.txt.

Sessió 7. Gestió d'Entrada/Sortida (II)

Preparació prèvia

Objectius

- Entendre el funcionament de les pipes sense nom.

Habilitats

- Ser capaços de comunicar processos utilitzant pipes sense nom.
- Ser capaços de comunicar processos utilitzant pipes amb nom.

Coneixements previs

- Crides a sistema de gestió de processos

Guia per al treball previ

- Abans de la sessió, consulteu el man (man nombre_comanda) de les següents comandes. En concret, per a cada comanda heu de llegir i entendre perfectament: la SYNOPSIS, la DESCRIPTION i les opcions que us comentem a la columna "Paràmetres" de la taula.

Per llegir al man	Descripció bàsica	Paràmetres
`mknod`	comanda que crea un fitxer especial	`P`
`mknod (anomenada al sistema)`	Crida al sistema que crea un fitxer especial	
`pipe`	Crida a sistema per crear una pipe sense nom	
`open/creat`	Obre (crea) un fitxer	`O_NONBLOCK, ENXIO, O_CREAT,O_TRUNC, "Permisos"`
`close`	Tanca un descriptor de fitxer	
`dup/dup2`	Duplica un descriptor de fitxer	
`mkfifo`	comanda que crea fifos (named pipes)	
`lseek`	Modifica la posició de lectura/escriptura d'un fitxer	`SEEK_SET, SEEK_CUR, SEEK_END`
`mmap`	Mapea fitxers en memòria	`MAP_PRIVATE`

- Crea una pipe amb nom mitjançant la comanda mknod (o mkfifo). A continuació, llança un procés que executi el programa 'cat' redireccionant la seva sortida cap a la pipe que acabes de crear. En una shell diferent llança un altre procés que executi també el programa 'cat', però ara redireccionant la seva entrada cap a la pipe que acabes de crear. Introdueix dades per teclat, a la primera Shell, i prem ^D per indicar la fi. Anota al fitxer "previ.txt" les comandes que has executat.
- És possible comunicar les dos comandes 'cat' des de dos terminals diferents a través d'una pipe sense nom (per exemple, utilitzant un pipeline de la shell vist en la sessió anterior)? I des del mateix terminal? Raona la resposta al fitxer "previ.txt".

- Escriu en el fitxer "previ.txt" el fragment de codi que hauríem d'afegir a un programa qualsevol per redireccionar la seva entrada estàndard a l'extrem d'escriptura d'una pipe sense nom utilitzant les crides a sistema close i dup. Imagina que el descriptor de fitxer associat a l'extrem d'escriptura de la pipe és el 4.
- **Per lliurar: previ7.tar.gz**

```
#tar zcfv previ7.tar.gz previ.txt
```

Bibliografia

- Transparències del Tema 4 (Entrada/Sortida) de SO-grau.
- Capítol 13 (I/O Systems) d'A. Silberschatz, P. Galvin i G. Gagne. Operating System Concepts, 8th ed, John Wiley &Sons, Inc. 2009.

Exercicis a realitzar al laboratori

➢ A mesura que vagis realitzant els exercicis, modifica el Makefile per poder compilar i muntar els nous programes que es demanen.

➢ Totes les preguntes que se us facin les haureu de contestar en un document de text a part, anomenat entrega.txt, en el qual indicareu, per a cada pregunta, el seu número i la vostra resposta. Aquest document s'ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 

➢ **Per lliurar: sessio7.tar.gz**

```
#tar zcfv7.tar.gz entrega.txt sense_nom.c escriptor.c lector.c
escriptor_v2.c append.c invertint_fitxer.c insertax2_lseek.c
insertax2_mmap.c Makefile
```

Pipes sense nom

Escriu un programa al fitxer "sense_nom.c" que creï una pipe sense nom i a continuació un procés fill, el canal d'entrada estàndard del qual haurà d'estar associat a l'extrem de lectura de la pipe. Per fer la redirecció utilitza les crides a sistema close i dup. El procés fill haurà de mutar la seva imatge per passar a executar la comanda 'cat' vista al treball previ. Per la seva banda, el procés pare enviarà a través de la pipe el missatge de text "Inici" al seu fill, tancarà el canal d'escriptura de la pipe i es quedarà a l'espera que el fill acabi. Quan això succeeixi, el pare mostrarà el missatge "Fi" per la sortida estàndard i acabarà l'execució.

1. Executa el programa anterior fent que el Shell redireccioni la sortida estàndard del pare a un fitxer.

PREGUNTA 63. Què conté el fitxer al final de la execució? ¿Conté la sortida del pare i del fill, o només la del pare? Com s'explica aquest contingut?

2. Canvia el codi del pare perquè no tanqui l'extrem d'escriptura de la pipe després d'enviar el missatge.

PREGUNTA 64. Acaba el programa pare? i el fill? Per què?

Pipes amb nom

Escriu dos programes que es comuniquin a través d'una pipe amb nom. Un d'ells (lector.c) llegirà de la pipe fins que la lectura li indiqui que no queden més dades per llegir i mostrarà a sortida estàndard tot el que vagi llegint. L'altre procés (escriptor.c) llegirà de l'entrada estàndard fins que la lectura li indiqui que no queden dades i escriurà a la pipe tot el que vagi llegint. Quan no quedin més dades per llegir els dos programes han d'acabar.

PREGUNTA 65. Si volguéssim que el lector també pogués enviar un missatge a l'escriptor, podríem utilitzar la mateixa pipe amb nom o n'hauríem de crear una altra? Raona la resposta.

Escriu una altra versió del programa escriptor a la pipe, anomenada escriptor_v2.c. Aquest programa en intentar obrir la pipe, si no hi ha cap lector de la pipe, mostrarà un missatge per la sortida estàndard que indiqui que s'està esperant a un lector i a continuació es bloquejarà en l'open de la pipe fins que un lector obri la pipe per llegir. Consulta l'error ENXIO en el man d'open (man 2 open) per veure com implementar aquest comportament.

Control de la mida dels fitxers

Crea un fitxer "file" el contingut del qual és "12345".
Ara implementa un programa, anomenat append.c, que afegeixi al final del fitxer "file" el contingut del propi fitxer. Utilitzem el fitxer file a mode d' exemple però el programa ha de ser genèric: per a qualsevol fitxer d'entrada, després d' executar el programa append.c el fitxer tindrà el contingut original duplicat. **Pista**: si quan provis aquest programa triga més d'uns segons en acabar, mata al procés i comprova la mida del fitxer. Si aquesta mida és més que la doble de la mida original revisa en el codi la condició de fi que li has posat al bucle de lectura del fitxer.

PREGUNTA 66. Lliurament el fitxer append.c

Operacions amb lseek

Crea un programa que anomenarem "invertint_fitxer" que farà una còpia d'un fitxer que rep com a paràmetre però amb el contingut invertit. El nom del fitxer resultant serà el del fitxer original amb l'extensió ".inv".

PREGUNTA 67. Lliurament el fitxer invirtiendo_fichero.c.

Crea un fitxer de dades el contingut del qual sigui "123456". Ara implementa un codi (insertarx2_lseek.c) que, usant lseek, insereix el caràcter "X" entre el "3" i el "4", de tal manera que el contingut final del fitxer sigui "123X456". Fes aquesta prova a mode d'exemple però el programa ha de ser genèric: Donada una posició concreta del fitxer s'insereix el caràcter "X".

PREGUNTA 68. Lliurament el fitxer inserirx2_lseek.c.

Reimplementa l'exercici anterior utilitzant mmap. Anomena'l insertax2_mmap.c

PREGUNTA 69. Lliurament el fitxer insertarx2_mmap.c

Sessió 8. Multithreading

Preparació prèvia

Objectius

- Entendre les diferències entre processos i threads.
- Entendre els usos típics d'utilització de threads.
- Entendre les problemàtiques associades a l'ús de memòria compartida.

Habilitats

- Ser capaç d'identificar les diferències entre processos i threads.
- Ser capaç d'utilitzar comandes i crides al sistema bàsiques per treballar amb threads.
- Ser capaç d'identificar i solucionar condicions de carrera.

Coneixements previs

- Crides a sistema de gestió de processos.

Guia per al treball previ

- Abans de la sessió, consulteu el man (man nombre_comanda) de les següents comandes. En concret, per a cada comanda haureu de llegir i entendre perfectament: la SYNOPSIS i la DESCRIPTION.

Per llegir al man	Descripció bàsica
`pthread_create`	Crea un nou pthread
`pthread_join`	Espera que finalitzi el pthread indicat com a paràmetre
`pthread_mutex_init`	Inicialitza una variable per controlar l'accés a una regió crítica
`pthread_mutex_lock`	Controla l'entrada a una regió crítica
`pthread_mutex_unlock`	Controla la sortida d'una regió crítica
`pthread_exit`	Finalitza el pthread actual
`clock_gettime`	Llegeix el temps en una estructura de segons i nanosegons

Un dels usos dels threads és evitar que un procés es bloquegi completament en fer una operació d'E/S bloquejant. Per a això, és habitual crear un thread, que serà el que es bloquejarà, en lloc del thread principal, per la qual cosa el procés pot atendre altres peticions. En aquesta sessió us donem un exemple fent servir sockets. Socket simplement és un dispositiu de xarxa. En aquest cas utilitzarem el que s'anomena UNIX sockets que permet crear un socket però local a la màquina i que en lloc d'identificar-se per una adreça IP (direcció de xarxa) i un port s'identifica per un nom de fitxer. Durant la sessió haureu de modificar els fitxers client.c i server.c però aquests fitxers utilitzen un fitxer socketMng.c que no caldrà que modifiqueu. Aquest últim fitxer conté funcions auxiliars per crear els sockets, etc. Veureu que la part client és diferent de la part servidor. Mentre que el client només ha de crear el socket i enviar/rebre dades, el servidor ha de crear un socket "master", al qual els clients es connectaran, i es crearà un canal (FD) específic per a cada connexió concreta. Com a treball previ, heu d'entendre el codi bàsic que se us dóna i realitzar les següents modificacions:

- Modifica el programa cliente.c perquè mesuri el temps que triga a fer totes les peticions. Com que les fa una rere l'altra el normal és que trigui *num_peticiones x cost de cada petició*. El cost de cada petició es pot ajustar en l'sleep que hi ha en el servidor. Anota el temps per a 1 ... [num.coresx8] peticions.
    - NOTA: Si veus que no es nota la diferència de temps augmenta el nombre de peticions.
- Modifica el cliente, anomena'l cliente_thread.c, perquè creï un thread per a cada petició. Repeteix les proves anteriors per comprovar si té algun impacte en el temps.

- **Per lliurar: previ8.tar.gz**

```
#tar zcfv previ8.tar.gz cliente.c cliente_thread.c
```

Bibliografia

- Transparències del Tema 5 de SO-grau.

Exercicis a realitzar al laboratori

➢ A mesura que vagis realitzant els exercicis, modifica el Makefile per poder compilar i muntar els nous programes que es demanen.

➢ Totes les preguntes que se us facin les haureu de contestar en un document de text a part, anomenat entrega.txt, en el qual indicareu, per a cada pregunta, el seu número i la vostra resposta. Aquest document s' ha de lliurar a través del Racó. Les preguntes estan ressaltades en negreta enmig del text i marcades amb el símbol: 

➢ **Per lliurar: sessio8.tar.gz**

```
#tar zcfv sessio8.tar.gz entrega.txt costProcessos.c
costThreads.c suma_paralela.c suma_paralela_v2.c
suma_paralela_v3.c server_thread.c client.c cliente.c
```

Processos vs Threads

La creació d'un thread és molt menys costosa que la creació d'un procés ja que alguns recursos com la memòria o els canals són compartits. Crea dos programes en C, un anomenat "costProcessos.c" i un altre "costThreads.c". Cadascú rebrà un argument que serà el nombre de processos o threads a crear respectivament. El programa haurà d'implementar un bucle que en cada iteració crearà un procés (o un thread) i esperar que acabi. Fes que el procés o thread simplement escrigui un missatge amb el seu ID (PID/TID) i acabi (exit o pthread_exit).

PREGUNTA 70. Executa els programes amb 1000, 10.000 i 100.000 iteracions i mesura el temps d' execució. Anota els temps en el fitxer entrega.txt

Threads per oferir paral·lelisme

El programa suma_paralela.c és un exemple típic de creació de threads. El programa crea N threads (rebut com a argument) i fa la suma de les dues matrius en paral·lel. Executa el programa amb 1 ... [número de coresx4] threads. Modifica el programa per mesurar el temps d' execució i justifica els temps obtinguts. Consulta el número de cores executant la comanda lscpu.

Threads i memòria compartida

1. Modifica el programa suma_paralela.c, anomena'l suma_paralela_v2.c, perquè es calculi l'agregat de tots els valors de la matriu C. Fes que cada thread acumuli els elements de la part de la matriu C que ha calculat en una única variable global de tipus int. Per assegurar que el resultat és correcte, utilitzarem una variable de tipus pthread_mutex_t (també global). La variable la inicialitzarem en el main abans de crear els threads i la utilitzarem en la funció sum_matrix.

2. Comprova que el resultat és correcte comparant el resultat quan utilitzem 1 thread i diversos threads. Compara el temps d' execució.

3. Copia el programa suma_paralela_v2.c, anomena'l suma_paralela_v3.c. Fes una optimització en la qual cada thread utilitzi una variable local a la funció sum_matrix per sumar tots els elements que li corresponen. Com que és local, no necessitarem utilitzar exclusió mútua. Un cop tinguem el resultat, per acumular-lo en la variable global utilitzarem exclusió mútua. Repeteix els experiments anteriors, entre 1 i [nombre de coresx4], i compara el temps d'execució.

PREGUNTA 71. Anota els experiments anteriors en el fitxer entrega.txt i raona la diferència entre suma_paralela_v2 i suma_paralela_v3.

Threads per a Entrada/Sortida

Tal com s'ha vist a classe, els threads serveixen per oferir un millor rendiment en programes tipus client/servidor, permetent atendre N peticions alhora. Els programes cliente.c i server.c són un exemple de senzill d'aquesta estratègia. El programa server crea un socket i atén cada connexió de forma seqüencial (una darrere l'altra). El programa cliente fa N peticions una darrere l'altra. El codi per usar aquest tipus de dispositiu (sockets) no és necessari modificar-lo, només entendre'l per utilitzar-lo.

4. Modifica el server.c, anomena'l server_thread.c, perquè es creï un thread per atendre cada connexió dels clients. Repeteix els experiments del treball previ i anota el temps d'execució de tots els casos realitzats.

PREGUNTA 72. Anota els experiments anteriors en el fitxer entrega.txt i raona l'impacte de la modificació comparat amb la modificació del treball previ.

Apèndix: Llista de preguntes

PREGUNTA 1. Quines comandes heu utilitzat per crear els directoris S1... S5?

PREGUNTA 2. Quina comanda utilitzes per llistar el contingut d'un directori? Quina opció cal afegir per veure els fitxers ocults?

PREGUNTA 3. Quina opció cal afegir a ls per veure informació estesa dels fitxers? Quins camps es veuen per defecte amb aquesta opció? (si no trobes la informació al pregunta al teu professor)

PREGUNTA 4. Quines opcions de menú has activat per estendre la informació que mostra el File Browser?

PREGUNTA 5. Quina seqüència de comandes has executat per esborrar un directori, comprovar que no hi és i tornar a crear-lo?

PREGUNTA 6. Quina diferència hi ha entre la comanda cat i less?

PREGUNTA 7. **Per a què serveix l'opció -i de la comanda cp? Quina és la comanda per fer un àlies de la comanda que inclogui l'opció -i?**

PREGUNTA 8. **Què fa l'opció -i de la comanda rm? I l'opció -i del mv? Escriu la comanda per fer un àlies de la comanda rm que inclogui l'opció -i.**

PREGUNTA 9. **Quines opcions de chmod has utilitzat per deixar només els permisos d'escriptura? Quin resultat ha tornat a intentar veure el fitxer test? Quines opcions de chmod has utilitzat per deixar només els permisos de lectura? Has aconseguit esborrar-lo?**

PREGUNTA 10. **Quina línia de comandes has executat per mostrar la informació de l'inode del fitxer test?**

PREGUNTA 11. **Quants enllaços (links) té el directori S1?**

PREGUNTA 12. **De quin tipus és cadascun de links creats, sl_pr i hl_pr?**

PREGUNTA 13. **Quin és el nombre de links que té cada fitxer? Què significa aquest valor? Quin inode té cada fitxer?**

PREGUNTA 14. **Observes alguna diferència en el resultat d'algun de les comandes quan s'executen sobre sl_pr i quan s'executen sobre hl_pr? Si hi ha alguna diferència, explica el motiu.**

PREGUNTA 15. **Observes alguna diferència en el resultat d'algun de les comandes quan s'executen sobre sl_pr i quan s'executen sobre hl_pr? Si hi ha alguna diferència, explica el motiu.**

PREGUNTA 16. **Observes alguna diferència respecte a l'execució d'aquests comandes abans i després d'esborrar el fitxer pr.txt? Si hi ha alguna diferència, explica el motiu.**

PREGUNTA 17. **Com podeu saber els sistemes de fitxers muntats en el vostre sistema i de quin tipus són? Indica, a més, en quins directoris estan muntats.**

PREGUNTA 18. **Com es pot saber el nombre d'inodes lliures d'un sistema de fitxers? Quina comanda has utilitzat i amb quines opcions?**

PREGUNTA 19. **Com es pot saber l'espai lliure d'un sistema de fitxers? Quina comanda has utilitzat i amb quines opcions?**

PREGUNTA 20. **Quin és el significat de les variables d'entorn PATH, HOME i PWD?**

PREGUNTA 21. **La variable PATH és una llista de directoris, quin caràcter fa de separador entre un directori i un altre?**

PREGUNTA 22. **Quina comanda has fet servir per definir i consultar el valor de les noves variables que has definit?**

PREGUNTA 23. **Quina versió del ls s'ha executat?: El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.**

PREGUNTA 24. **El directori en el qual estàs, està definit en la variable PATH? Què implica això?**

PREGUNTA 25. **Quin binari ls s'ha executat en cada cas dels anteriors:El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.**

PREGUNTA 26. **Quin binari ls s'ha executat en cada cas dels anteriors:El ls del sistema o el que t'acabes de descarregar? Executa la comanda "which ls" per comprovar-ho.**

PREGUNTA 27. Quina diferència hi ha entre utilitzar > i >>?

PREGUNTA 28. Quina opció has hagut d'afegir al gcc per generar el fitxer? Quina opció has hagut d'afegir al gcc perquè el compilador trobi el fitxer mis_funciones.h?

PREGUNTA 29. Com pot saber un procés el pid dels seus fills? Quina crida poden utilitzar els processos per consultar el seu propi PID?

PREGUNTA 30. En quins casos s'executarà qualsevol codi que aparegui just després de la crida execlp (En qualsevol cas/En cas que l'execlp s'executi de forma correcta/En cas que l'execlp falli)?

PREGUNTA 31. Quins directoris hi ha dins de /proc/PID_PADRE? Quina opció de ls has fet servir?

PREGUNTA 32. Per al procés pare, apunta el contingut dels fitxers status i cmdline. Compara el contingut del fitxer environ amb el resultat de la comanda env (fixa't per exemple en la variable PATH i la variable PWD) Quina relació hi ha? Cerca en el contingut del fitxer status l'estat en el qual es troba el procés i anota'l al fitxer de respostes. Anota també el temps de CPU que ha consumit en mode usuari que pots trobar-lo en el fitxer stat del procés (veuràs que stat conté una sèrie de números, consulta en el manual el format què representa cadascun d'aquests números: man proc).

PREGUNTA 33. En el cas del procés fill, a quins fitxers "apunten" els fitxers cwd i exe? Quin creus que és el significat de cwd i exe?

PREGUNTA 34. En el cas del procés fill, ¿pots mostrar el contingut dels fitxers environ, status i cmdline del procés fill? En quin estat es troba?

PREGUNTA 35. En el cas del procés fill, a quins fitxers "apunten" els fitxers cwd i exe? Quin creus que és el significat de cwd i exe? Quines diferències hi ha amb la versió anterior? A què es deuen?

PREGUNTA 36. En el cas del procés fill, ¿pots mostrar el contingut dels fitxers environ, status i cmdline del procés fill? En quin estat es troba? Quines diferències hi ha amb la versió anterior? A què es deuen?

PREGUNTA 37. Comprova el fitxer status de /proc/PID_PADRE. En quin estat està el procés pare?

PREGUNTA 38. Què has fet per guardar la sortida de les execucions de myPS3?

PREGUNTA 39. Què li passa al pare ara quan li arriba l'esdeveniment SIGALRM?

PREGUNTA 40. Què passa amb la taula de tractaments de signals si fem un execlp (i canviem el codi)? Es manté la reprogramació de signals? Es posa per defecte?

PREGUNTA 41. El programa ejemplo_waitpid, És seqüencial o concurrent?

PREGUNTA 42. El segon usuari pot enviar signals al procés llançat per l'usuari inicial?, quin error dona?

PREGUNTA 43. Quin missatge mostra el Shell quan s'envia per segona vegada el mateix signal?

PREGUNTA 44. Explica quines diferències veus en els missatges que apareixen per pantalla que mostren els dos experiments i quin és el motiu

PREGUNTA 45. Explica a què es deu el resultat de les execucions de signal_perdido amb paràmetre 1 i amb paràmetre 0

PREGUNTA 46. **Per als dos executables (compilat estàtic i dinàmic) observa la seva mida i la sortida de les comandes *nm* i *objdump*. En què es diferencien?**

PREGUNTA 47. **Observa el contingut del fitxer *maps* del /proc per a cada procés i anota per a cada regió: adreça inicial, adreça final i etiqueta associada. Quina diferència hi ha entre el *maps* de cada procés?Observa el contingut del fitxer *maps* del /proc per a cada procés i anota per a cada regió: adreça inicial, adreça final i etiqueta associada. Quina diferència hi ha entre el *maps* de cada procés?**

PREGUNTA 48. **A quina regió de les descrites en el maps pertany cada variable (`ptr`, `cons1`, `resp`, `argn` i `argk`) i cada regió reservada amb *malloc*? Anota l'adreça inicial, l'adreça final i el nom de la regió i relaciona els resultats amb les seccions de cada programa detallades per la comanda *nm*.**

PREGUNTA 49. **Per a cada execució anota el nombre d'invocacions a *malloc* i les adreces inicial i final del heap que mostren el fitxer *maps* del /proc. Canvia la mida del heap segons el paràmetre d'entrada? Per què?**

PREGUNTA 50. **Quina és la mida del heap en aquest cas? Explica els resultats.**

PREGUNTA 51. **Quin és el valor que veu el fill en la variable al principi de la seva execució? Com justifiques aquest comportament?**

PREGUNTA 52. **A quina regió de les descrites en el *maps* pertany la regió reservada amb *mmap*? Indica l'adreça inicial i l'adreça final. Quin és el valor que veu el pare en la variable al final de l'execució? Com justifiques aquest comportament?**

PREGUNTA 53. **Quin és el valor que veu el pare en la variable al final de l'execució? Com justifiques aquest comportament?**

PREGUNTA 54. **Compara el contingut del fitxer *maps* del /proc per als programes *mem2_v2* i *mem2_v3.* En què es diferencien?**

PREGUNTA 55. **Quin error conté el codi del programa? Per què el programa no falla en les primeres iteracions?**

PREGUNTA 56. **Apunta en el fitxer "entrega.txt" les comandes que has utilitzat en cada apartat. A més, lliura el fitxer "es1_v2.c" Quines diferències observes en les dues execucions de strace? Quantes crides a sistema read executa cada versió? Quina influència pot tenir sobre el rendiment d'ambdues versions de codi? Per què?**

PREGUNTA 57. **Explica les diferències observades en la sortida del programa quan el primer paràmetre val 0 o 1. Per a què creus que serveix aquest paràmetre?**

PREGUNTA 58. **Explica el motiu dels resultats observats depenent del format fitxer d'entrada.**

PREGUNTA 59. **Explica les diferències observades entre la sortida del programa es3 i es4. Per què la sortida és intel·ligible per a un dels fitxers d'entrada i no per a l'altre?**

PREGUNTA 60. **Escriu els valors que ha mostrat l'script showCpuTime.sh per a cadascun dels processos i comenta les diferències entre ells pel que fa al consum de CPU. A què es deuen? Identifica les línies de codi de marquen la diferència entre ells**

PREGUNTA 61. **Anota en el fitxer entrega.txt el resultat d'ambdues execucions. Lliura el codi programat al fitxer es6_v2.c**

PREGUNTA 62. **Què passaria si activèssim el flag SA_RESTART al sigaction? Repeteix les execucions anteriors activant el flag SA_RESTART i anota el resultat en el fitxer entrega.txt.**

PREGUNTA 63. **Què conté el fitxer al final de la execució? ¿Conté la sortida del pare i del fill, o només la del pare? Com s'explica aquest contingut?**

PREGUNTA 64. **Acaba el programa pare? i el fill? Per què?**

PREGUNTA 65. **Si volguéssim que el lector també pogués enviar un missatge a l'escriptor, podríem utilitzar la mateixa pipe amb nom o n'hauríem de crear una altra? Raona la resposta.**

PREGUNTA 66. **Lliurament el fitxer append.c**

PREGUNTA 67. **Lliurament el fitxer invirtiendo_fichero.c.**

PREGUNTA 68. **Lliurament el fitxer inserirx2_lseek.c.**

PREGUNTA 69. **Lliurament el fitxer insertarx2_mmap.c**

PREGUNTA 70.

PREGUNTA 71.

PREGUNTA 72. **Anota els experiments anteriors en el fitxer entrega.txt i raona l'impacte de la modificació comparat amb la modificació del treball previ.**