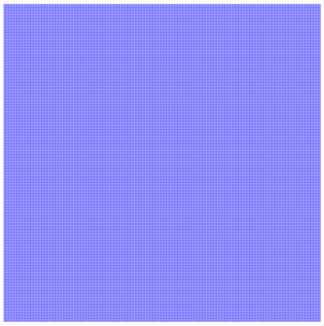


LABORATORI D'ARQUITECTURA I SISTEMES OPERATIUS

Material d'estudi

Enric Morancho, Pau Bofill, Maribel March, Montse Farreras
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Introducció

Aquest document descriu el treball a desenvolupar a cada sessió de laboratori de l'assignatura Laboratori d'Arquitectura i Sistemes Operatius (Labariso). Durant el curs, l'alumne portarà a la pràctica els coneixements que estigui adquirint a l'assignatura Arquitectura i Sistemes Operatius (Ariso).

A cada mòdul de l'assignatura Ariso li corresponen 2 o 3 sessions de laboratori de l'assignatura Labariso. En general, una d'aquestes sessions servirà per provar i/o modificar petits programes d'exemple, i les altres per a que l'alumne desenvolupi un projecte al llarg del curs.

Sessió 1: Shell (1/2)

L'objectiu d'aquesta sessió (i de la següent) és descriure les característiques de Unix quan hi interaccionem des de l'intèrpret de comandes (*shell*). La primera sessió es centra en com entrar al sistema, les comandes bàsiques, el sistema del fitxers i com sortir del sistema.

Entrada al sistema

Com Unix és un sistema operatiu multiusuari, per iniciar una sessió de treball heu d'identificar-vos i autenticar la vostra identitat. Això es realitza indicant quin és el vostre identificador d'usuari (*login*) i la vostra paraula clau (*password*). Un cop introduïts, el sistema verifica que siguin correctes i, en cas afirmatiu, s'inicia la sessió de treball.

Si treballeu en mode text, automàticament es posa en marxa una aplicació anomenada intèrpret de comandes (*shell*). Aquesta aplicació és la responsable de llegir les comandes que tecleja l'usuari i d'invocar els programes corresponents.

Si treballeu en mode gràfic, es posa en marxa una aplicació anomenada gestor de finestres i, normalment, no s'inicia cap intèrpret de comandes. Per a iniciar un intèrpret de comandes cal utilitzar el ratolí i prémer la icona convenient.

Hi ha diversos intèrprets de comandes disponibles als sistemes Unix: sh (*Bourne shell*), csh (*Cshell*), ksh (*Korn shell*), tcsh (*TCshell*), zsh (*Zshell*), bash (*Bourne again shell*),... A aquesta assignatura utilitzarem l'intèrpret que estigui configurat per defecte al sistema.

Quan l'intèrpret de comandes està preparat per a rebre les ordres de l'usuari, mostra un indicador de disponibilitat (*prompt*). Als exemples d'aquesta sessió assumirem que és prompt%.

Algunes comandes bàsiques

A continuació es presenten algunes comandes típiques de tot sistema Unix:

- **date**: mostra l'hora i la data actual del sistema
- **man**: permet accedir al manual del sistema. Espera com a paràmetre el nom de la comanda o de la crida al sistema de la que voleu obtenir informació. Per a navegar dins d'aquesta informació podem utilitzar la barra d'espai per avançar pàgina, la tecla b (*backward*) per retrocedir una pàgina i la tecla q per tornar a l'intèrpret de comandes. Per veure altres possibilitats, cal prémer la tecla h (*help*).
- **passwd**: permet canviar el *password*.
- **echo**: escriu el missatge especificat
- **who**: mostra els usuaris connectats a la nostra màquina
- **finger**: mostra els usuaris connectats a la nostra màquina o a una màquina remota

Sistema de fitxers

Estructura

El sistema de fitxers Unix té una estructura jeràrquica. Existeix un **directori arrel** (***root directory***) representat amb el símbol /. En aquest directori podem trobar fitxers ordinaris (fitxers de text, executables, ...) i també fitxers de tipus directori. Igualment, en aquests fitxers de tipus directori també poden contenir fitxers ordinaris i fitxers de tipus directori. D'aquesta manera, l'estructura jeràrquica es pot estendre fins on sigui necessari.

Cada fitxer de tipus directori conté dos fitxers especials (. i ..). El primer (.) fa referència al propi directori i el segon (..) fa referència al directori que conté a aquest.

Navegació al sistema de fitxers

Tota sessió de treball té associat un **directori de treball** (o **actual**) dins del sistema de fitxers. Per saber quin és, heu d'executar la comanda pwd (*print working directory*). Aquesta comanda mostra el camí que hi ha des del directori arrel fins el directori actual.

Per modificar el directori de treball associat a una sessió de treball cal utilitzar la comanda cd (*change directory*). Per exemple, la comanda cd .. ens permet pujar un nivell dins de l'estructura (executeu pwd per veure quin és el nou directori de treball), o anar directament a un directori (per exemple, cd /).

Quan iniciem una sessió de treball, el sistema associa un directori de treball per defecte a aquesta sessió. D'aquest directori en direm **directori d'usuari** (***home directory***) i cada usuari en té assignat un. Per tornar al directori d'usuari cal executar la comanda cd sense paràmetres (executeu la comanda pwd per comprovar-ho).

Gestió de directoris

Les comandes que permeten crear i eliminar directoris són mkdir i rmdir. Consulteu quins paràmetres necessiten al manual del sistema.

Visualització del contingut d'un directori

La comanda la és la responsable de mostrar quins fitxers estan emmagatzemats a un directori. Per exemple, situeu-vos al directori arrel i executeu la comanda ls. Apareixerà la llista de fitxers del directori arrel.

```
prompt% cd /
prompt% ls
bin boot dev etc home lib lost+found mnt nsr opt proc root sbin tmp
users usr var
prompt%
```

La comanda ls admet paràmetres com ara -l que permeten obtenir més informació sobre els fitxers. Un resultat possible de la comanda ls -l podria ser:

```
prompt% ls -l
-rw-r--r-- 1 root root 2434 mar 8 2000 DIR_COLORS
-rw-r--r-- 1 root root 9 dic 10 16:34 HOSTNAME
drwxr-xr-x 4 root root 4096 dic 18 2001 X11
...
prompt%
```

Podeu observar que per cada fitxer hi ha una línia d'informació. La interpretació d'aquestes dades és la següent:

- El primer caràcter indica si es tracta d'un fitxer ordinari (-) o d'un fitxer de tipus directori (d).
- Els següents 9 caràcters indiquen les proteccions del fitxer. Més endavant s'explicarà con interpretar-les i modificar-les.
- A continuació apareix un nombre que no interpretarem.
- Tot seguit hi ha dos identificadors. Indiquen qui és l'usuari propietari del fitxer i a quin grup d'usuaris pertany.
- El següent nombre representa la mida (en bytes) del fitxer.
- Posteriorment trobem la data de darrera modificació del fitxer.
- Finalment figura el nom del fitxer.

Per defecte, la comanda ls no mostra informació sobre els fitxers ocults. A Unix, els fitxers ocults són els que el seu nom comença per punt (.). Per a que la comanda ls mostri informació sobre aquests fitxers cal passar-li el paràmetre -a.

Noms relatius i absoluts

Sempre que invoqueu una comanda que necessiti com a paràmetre un nom de fitxer, podeu escollir entre dos formes d'especificar el nom:

- Nom absolut: camí des del directori arrel fins el fitxer (sempre començarà per / i el caràcter / separarà els directoris).
- Nom relatiu: camí des del directori actual fins el fitxer (mai començarà per /).

Per exemple, si el vostre directori actual és /home/alumnes/aaaa0000 i voleu accedir al fitxer passwd que es troba al directori /etc, podeu fer-ho especificant el nom absolut del fitxer (/etc/passwd) o el nom relatiu al directori actual (../../../etc/passwd).

Operacions sobre fitxers

A continuació s'indiquen quines són les comandes que permeten fer les operacions més comuns sobre els fitxers:

- **rm** : esborra (*remove*) un fitxer. Cal indicar el nom del fitxer a esborrar.
- **cp** : copia (*copy*) un fitxer. Cal indicar el(s) nom(s) del(s) fitxer(s) a copiar i el directori destí
- **mv**: canvia el nom (*move*) d'un fitxer. Cal indicar el nom antic i el nom nou.
- **cat**: mostra el contingut d'un fitxer
- **more**: mostra el contingut d'un fitxer per pàgines

A mode d'exemple, es mostra com copiar un fitxer a un directori acabat de crear, a continuació es canvia el nom del fitxer copiat, posteriorment es veu el seu contingut i finalment s'esborra.

```
prompt% cd
prompt% mkdir prova
prompt% cd prova
prompt% cp /etc/passwd .
prompt% ls
passwd
prompt% mv passwd passwd2
prompt% ls
passwd2
prompt% cat passwd2
...
prompt% rm passwd2
prompt% ls
prompt%
```

Proteccions

Tots els usuaris d'un sistema Unix estan treballant sobre el mateix sistema de fitxers. Per a possibilitar que alguns fitxers només siguin accessibles per determinats usuaris apareix el concepte de proteccions dels fitxers.

Les proteccions determinen quines operacions pot fer cada tipus d'usuari sobre el fitxer.

• Es consideren 3 operacions: lectura (r), escriptura (w) i execució (x).
• Es consideren 3 tipus d'usuaris: el propietari del fitxer, els usuaris que pertanyen al seu mateix grup d'usuaris i la resta d'usuaris de la màquina.

Per tant, són necessaris 9 bits d'informació per a codificar totes aquestes possibilitats. Els 9 caràcters que mostra la comanda ls -l representen aquesta informació, els primers 3 caràcters fan referència al propietari del fitxer i els permisos estan ordenats seguint l'ordre rwx; si apareix un - indica que el permís no està concedit. Els següents 3 caràcters fan referència als usuaris del mateix grup i els darrers 3 caràcters fan referència a la resta d'usuaris de la màquina. Per exemple, les proteccions rwxr----- indiquen que el propietari pot fer totes les operacions sobre el fitxer, que els membres del seu grup només poden fer lectures i que la resta d'usuaris no pot fer cap operació sobre el fitxer.

Per modificar les proteccions d'un fitxer s'utilitza la comanda chmod. Aquesta comanda codifica les proteccions seguint el codi r=4, w=2 i x=1 i sumant els drets de cada tipus d'usuari. És a dir, les proteccions rwxr----- es representen com 740 (4+2+1, 4+0+0, 0+0+0). Per exemple, chmod 751 file faria que file passés a tenir les proteccions rwxr-x--x.

Directoris i fitxers habituals

A continuació mostrem els directoris més habituals de tot sistema de fitxers Unix:

- **/bin** : conté els executables de la majoria de comandes (/bin/ls, /bin/cp, ...)
- **/dev** : conté els fitxers que representen els dispositius del sistema (terminal /dev/tty, ...)
- **/etc** : conté dades de configuració del sistema (fitxer de *passwords* /etc/passwd, ...)
- **/home** : conté els directoris d'usuari dels usuaris de la màquina
- **/lib** : conté les llibreries de funcions
- **/proc**: conté informació sobre els programes en execució a la màquina
- **/tmp** : conté arxius temporals dels processos
- **/usr/include** : conte els fitxers de capçalera de les crides al sistema

Utilitzeu les comandes cd i ls per a explorar el contingut d'aquests directoris.

Metacaràcters per enumerar fitxers

L'intèrpret de comandes permet utilitzar metacaràcters, amb els quals podeu fer referència a varis fitxers de cop o a directoris d'ús comú.

- * : representa qualsevol seqüència de caràcters. Per exemple, `ls /bin/d*e` mostraria els fitxers amb nom que comença per d i acaba en e.
- ? : representa un caràcter qualsevol. Per exemple, `ls /bin/d??e` mostraria els fitxers amb nom que comença per d, acaba en e, i té exactament 4 caràcters.
- [ ] : representa un caràcter dins d'un rang. Per exemple, `ls /bin/[a-m]*[n-z]` mostraria els fitxers amb nom que comença amb un caràcter entre a i m, i acaba en un caràcter entre n i z.
- ~ : representa el directori d'usuari. `echo ~` mostraria quin és el nostre directori d'usuari.

➢ Activitats

1. Creeu un directori al vostre *home directory*, copieu-hi el fitxer executable `/bin/ls`, i feu que s'anomeni `lsmeu`.

2. Situeu-vos al directori arrel (`cd /`). Sense canviar de directori, copieu el fitxer /etc/passwd al directori que heu creat abans.

3. Esborreu la vostra còpia del fitxer `/etc/passwd`. Busqueu al manual alguna forma per a que abans de borrall els fitxers demani confirmació.

4. Hi ha alguna diferència entre un fitxer amb els drets de protecció `rwxrwxrwx` i un altre amb els drets `------rwx` ? Proveu-ho.

5. Quines són les proteccions mínimes necessàries per a poder esborrar un fitxer? Comproveu-ho.

6. Com s'interpreten les proteccions `rwx` sobre un fitxer de tipus directori?

Edició de textos

L'editor de textos més característic de Unix és el `vi`. De totes formes, a la majoria de màquines Unix també es troben instal·lats altres editors com ara pico, `joe, emacs, nedit, vim, ...`

Tot i que no són els editors més potents, a aquesta assignatura es recomana utilitzar els editors `pico` (mode text) o `gedit` (mode gràfic) ja que resulten els més senzills d'utilitzar.

Per a editar un fitxer heu d'indicar el nom de l'editor a utilitzar i el nom del fitxer a editar. Si el fitxer no existeix, es crearà un fitxer buit. Un cop fetes les modificacions desitjades, podem actualitzar el contingut del fitxer.

Sortida del sistema

Per a finalitzar l'execució de l'intèrpret de comandes heu d'escriure la comanda `logout`. Si treballeu en un entorn en mode text, això provocarà l'acabament de la sessió de treball amb el sistema. Si treballeu en un entorn gràfic, normalment també caldrà que busqueu la icona o opció de menú que us permeti finalitzar la sessió de treball.

➢ Activitats extres

- Consulteu al manual els paràmetres de la comanda `ls`.
- Consulteu al manual el funcionament de la comanda `du`.
- Consulteu al manual el funcionament de la comanda `df`.

Sessió 2: Shell (2/2)

L'objectiu d'aquesta sessió és continuar descrivint les característiques de Unix des de l'intèrpret de comandes (*shell*). Concretament, la noció de procés, les modalitats d'execució de programes, la redirecció d'entrada/sortida, la comunicació de processos mitjançant *pipes* i les variables d'entorn.

Noció de procés

Un procés no és més que un programa en execució. Normalment, l'intèrpret de comandes crea una procés per cada comanda que llegeix i aquest procés s'encarrega d'executar la comanda indicada. Quan aquest procés finalitza, l'intèrpret de comandes torna a mostrar el *prompt* del sistema.

La comanda que dóna informació sobre els processos a la màquina és ps. Per defecte, mostra informació bàsica dels processos en execució a la sessió actual, però utilitzant alguns paràmetres podem fer que doni tota la informació disponible sobre aquests processos (paràmetre -f o -l), informació sobre tots els processos d'un usuari (paràmetre -u i el nom de l'usuari) o sobre tots els processos de la màquina (paràmetre -e).

Per exemple:

```
prompt% ps
PID TTY TIME CMD
4976 pts/49 00:00:00 bash
7913 pts/49 00:00:00 ps
prompt% ps -l
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
100 S 503 4976 4973 0 66 0 - 455 wait4 pts/49 00:00:00 bash
000 R 503 7914 4976 21 72 0 - 678 - pts/49 00:00:00 ps
prompt% ps -f
UID PID PPID C STIME TTY TIME CMD
enricm 4976 4973 0 Jan08 pts/49 00:00:00 -bash
enricm 7915 4976 0 16:38 pts/49 00:00:00 ps -f
prompt% ps -ef
...
```

A continuació es descriu el significat de les columnes més importants:

- **PID** identificador del procés. És un nombre enter.
- **TTY** terminal associat al procés. El nom del fitxer corresponent s'obté afegint a /dev/ el contingut de la columna (en aquest cas, /dev/pts/49).
- **TIME**: indica el temps de CPU que porta consumit el procés. Observeu que aquest temps pot ser sensiblement inferior al temps transcorregut des de que es va iniciar l'execució del programa ja que alguns processos fan poc ús de la CPU.
- **CMD**: comanda responsable de la creació del procés.
- **UID:** identificador d'usuari propietari del procés (pot ser l'identificador d'usuari o el seu equivalent numèric).
- **PPID**: identificador del procés pare. A Unix, tot procés (excepte el primer procés del sistema) és fill d'algun altre procés.
- **SZ** : quantitat de memòria ocupada pel procés.

- **STIME**: instant en el que es va iniciar l'execució del procés.

Una altra possibilitat per veure els processos que hi ha a la màquina és utilitzar la comanda top. Aquesta comanda mostra periòdicament informació de quants processos hi ha al sistema, l'ocupació global de la memòria i quins són els processos que estan fent més ús del processador. Per acabar l'execució d'aquesta comanda cal prémer la tecla q.

L'identificador de procés (*pid*) resulta necessari per a efectuar algunes operacions sobre els processos, per exemple, per matar-los. La comanda que ens permet matar un procés és la comanda kill; cal parametritzar-la amb -9 i el *pid* del procés a matar.

➢ Activitats

1. • Llisteu tots els processos que té en execució l'usuari `root`.

2. • Executeu la comanda `ps` i intenteu matar al procés que executa la comanda `ps` fent `kill -9 pid_procés`. A què és degut el missatge d'error que apareix?

3. • Sabent que el `PPID` és l'identificador del procés que ha creat a un altre procés, obtingueu tota la genealogia del procés que executa el vostre intèrpret de comandes.

4. • Intenteu interpretar el significat de la informació mostrada per la comanda `top`.

Modalitats d'execució de programes

Per defecte, un cop llegida una comanda, l'intèrpret de comandes no torna a mostrar el *prompt* fins que la comanda hagi finalitzat la seva execució. Aquesta forma d'executar els programes rep el nom de ***foreground*** (**en primer pla**).

Els intèrprets de comandes també permeten iniciar l'execució d'una comanda de forma que el *prompt* es mostri abans que la comanda finalitzi la seva execució. Aquesta forma d'executar les comandes rep el nom d'execució en ***background*** (**en segon pla**). A Unix, podeu executar programes en *background* utilitzant el metacaràcter &.

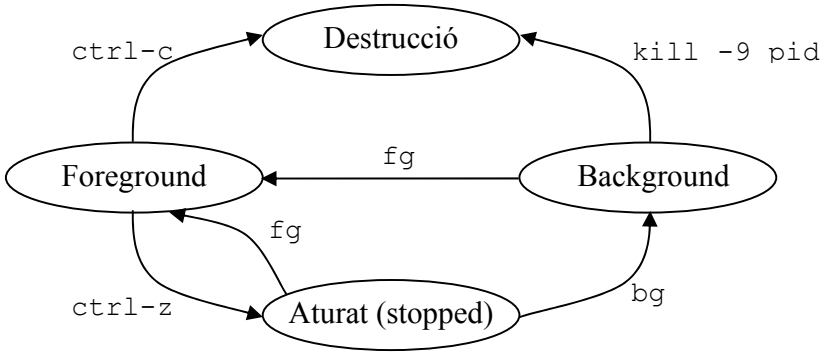
A mode d'exemple, inicialment s'invoca la comanda `sleep 5` en *foreground* i tot seguit la comanda `echo` (el metacaràcter ; és el separador entre comandes). Podeu veure que el missatge s'escriu un cop han transcorregut els 5 segons. En canvi, després s'invoca `sleep` en *background*, per tant el missatge s'escriu abans que passin els 5 segons.

```
prompt% sleep 5; echo hola
hola
prompt% sleep 5 & echo hola
[1] 3333
hola
prompt% ps
...
3333 pts/14 00:00:00 sleep
...
prompt%
...
[1] Done sleep 5
```

Els nombres que apareixen a l'iniciar l'execució d'una comanda en *background* representen un identificador de comanda i l'identificador de procés. Quan la comanda en *background* finalitza la seva execució, apareix un missatge indicant-ho.

Canvi de modalitat d'execució d'un programa

El següent diagrama mostra com canviar el mode d'execució d'un programa que estigui en execució. Per aturar una comanda que s'executi en *foreground* cal prémer les tecles `Ctrl i Z` (prement `Ctrl i C` provocaríem la finalització de la comanda, tot i que algunes comandes són immunes al `Ctrl-C`). La resta de transicions s'efectuen mitjançant comandes.



- **Activitats**

5. Invoqueu la comanda `sleep 10` en *foreground*. Quines diferències hi ha entre prémer `Ctrl-C i Ctlr-Z` abans de que transcorrin els 10 segons?

6. Entreu al manual de la màquina (per exemple, `man ls`) i sortiu prement les tecles `Ctlr-Z`. Feu el necessari per a continuar la consulta del manual.

Redirecció d'entrada/sortida

A Unix, els processos no llegeixen directament del teclat ni escriuen directament a pantalla, sinó que ho fan sobre uns dispositius intermitjos anomenats entrada estàndard (*stdin*) i sortida estàndard (*stdout*). Per defecte, aquests dispositius intermitjos fan referència al teclat i a la pantalla, però des de l'intèrpret de comandes podeu modificar aquesta associació (redireccionar els canals estàndard).

El metacaràcter < permet redireccionar l'entrada estàndard a un fitxer; anàlogament, el metacaràcter > permet redireccionar la sortida estàndard a un fitxer.

A mode d'exemple, podeu executar la comanda ls redireccionant la seva sortida estàndard al fitxer file i després executar la comanda wc (compta el nombre de caràcters, paraules i línies) fent que la seva entrada estàndard sigui file.

```
prompt% ls /bin > file
prompt% ls
... file ...
prompt% wc < file
88 88 520
prompt%
```

Com a Unix tots els dispositius estan presents al sistema de fitxers, podríeu arribar a redireccionar la sortida estàndard d'una comanda cap a una altra finestra. Proveu-ho obrint dos intèrprets de comandes i executant comandes des d'un d'ells tot redireccionant la sortida cap al `/dev/ttyXXX` (o `/dev/pts/XXX`) de l'altre.

Comunicació de processos mitjançant *pipes*

L'intèrpret de comandes permet executar concurrentment 2 o més processos facilitant que es comuniquin. Això s'aconsegueix utilitzant el metacaràcter |. Aquest metacaràcter indica que volem comunicar la sortida estancar d'un procés amb l'entrada estàndard d'un altre, és a dir, tot el que el primer procés escrigui per la seva sortida estàndard anirà a parar a l'entrada estàndard del següent. Per tant, executant `ls /bin | wc` aconseguiríeu un resultat similar al resultat obtingut en l'exemple anterior.

Utilitzant aquest metacaràcter, podem combinar vàries comandes senzilles i aconseguir resultats complexes. Les comandes que acostumen a combinar-se utilitzant *pipes* també són conegudes com filtres; totes elles llegeixen les seves dades d'entrada de l'entrada estàndard i escriuen el seu resultat per la sortida estàndard. Les més habituals són:

- `wc`: compta el nombre de línies/paraules i caràcters
- `sort`: ordena les línies seguint el criteri indicat
- `grep`: busca línies que continguin una paraula o patró
- `cut`: selecciona columnes
- `head`: selecciona les primeres línies del fitxer
- `tail`: selecciona les darreres línies del fitxer
- `more`: paginador
- `uniq, tr, awk, sed, ...`

A continuació es mostren alguns exemples del seu ús:

- `ps -ef | grep inetd` : obtenir informació sobre el procés que executa la comanda `inetd`
- `ls -l /bin | sort -n -r +4 | more` : mostra un llistat ordenat dels fitxers del directori `/bin` ordenat per mida
- `ls -l /bin | sort -n -r +4 | head -1 | cut -c56-` : escriu el nom del fitxer més gran del directori `/bin`
- `ps -e | tail +2 | wc -l`: diu el nombre de processos a la màquina
- `ls -l / | grep ^d | wc -l` : retorna el nombre de directoris que existeixen en el directori arrel

Proveu-les i intenteu entendre el seu funcionament. El significat dels paràmetres utilitzats el podeu deduir fent proves i el podeu trobar a les pàgines de manual del sistema.

- **Activitats**

7. Escriviu una comanda que mostri per la seva sortida estàndard el nombre de fitxers del directori arrel. (Pot ser útil el filtre `wc`).

8. Escriviu una comanda que mostri per la seva sortida estàndard el nom del segon fitxer més gran del directori arrel. (Poden ser útils els filtres `sort, cut, head i tail`).

9. Escriviu una línia de comandes que retorni els 5 processos de l'usuari `root` amb identificador de procés més alt.

10. Escriviu una comanda que retorni el nombre de processos que estan executant la comanda `mingetty`. (Pot ser útil el filtre `grep`).

11. Quines diferències hi ha entre executar `ls -l /bin >tmp; wc <tmp i ls -l /bin | wc` ? (Penseu en temps d'execució, recursos utilitzats,...)

Variables d'entorn

Els intèrprets de comandes permeten definir un seguit de variables i assignar valors a aquestes variables. Els programes poden consultar el valor de les variables i, en funció d'aquests, poden modificar el seu comportament. Per exemple, acostuma a existir una variable anomenada `TERM` que indica el tipus de terminal que esteu utilitzant. En funció del valor d'aquesta variable, un programa pot presentar els seus resultats de forma més o menys sofisticada (colors, nombre de columnes disponibles, ...).

Per veure totes les variables definides actualment podem utilitzar les comandes set i `printenv`. La modificació del valor de les variables depèn de l'intèrpret de comandes que s'estigui utilitzant.

Si voleu veure el valor d'una única variable podeu utilitzar la comanda `echo` i el metacaràcter $ (indica que fem referència a una variable). Per exemple, `echo $TERM`

A continuació es detalla el significat d'algunes de les variables més usuals:

- COLUMNS: Indica el nombre de columnes de la finestra de text
- HOME: Conté el directori d'usuari. És consultada per la comanda `cd` quan l'executem sense indicar cap paràmetre; `cd` canvia al directori indicat per aquesta variable.
- LINES: Indica el nombre de línies de la finestra de text
- PATH: Conté la llista de directoris on el *shell* busca els fitxers executables corresponents a les comandes que demaneu executar. La cerca es realitza en el

mateix ordre com els directoris estan emmagatzemats a la variable; en el moment que es troba un directori que contingui l'executable, la cerca finalitza; si no se'n troba cap, el *shell* mostra un missatge d'error del tipus *Command not found*.

- TERM: Indica el tipus del terminal, per exemple `vt100, xterm,` ...

➢ Activitats

12. Podria haver alguna diferència entre executar el fitxer executable `ls` de la forma `ls` o de la forma `/bin/ls`?

13. Consulteu el manual per veure com definir noves variables i modificar el seu valor.

➢ Activitats extres

- Escriviu una línia de comandes que mostri els 5 processos que estan consumint més memòria.

- Escriviu una línia de comandes que mostri els 5 processos que estan consumint més temps de CPU (des de que es van començar a executar).

- Situeu-vos al directori `/proc` i verifiqueu quina informació pot obtenir-se dels vostres processos.

- Consulteu al manual el funcionament de la comanda `find`.

Sessió 3: Entorn de desenvolupament d'aplicacions

L'objectiu d'aquesta sessió és donar-vos a conèixer quin és l'entorn de desenvolupament d'aplicacions que ofereixen la majoria de sistemes Unix. Per entorn de desenvolupament entenem el conjunt d'eines que podem utilitzar durant el cicle de vida d'una aplicació: editors, compiladors, muntadors, *debuggers*,..

Edició de programes

L'editor de text és l'eina que ens permet escriure i modificar el codi font dels programes. Els editors presents a la majoria de sistemes Unix són `vi`, `vim`, `gedit` (gràfic), `emacs`, `pico`, `joe`,... Tot i que l'editor "estàndard" i més estès als sistemes Unix és el `vi`, a aquesta assignatura s'aconsella utilitzar editors més senzills com ara el `pico` (text) o el `gedit` (gràfic).

La majoria d'editors tenen opcions que faciliten la nostra tasca com a programadors ja que poden ajudar a cometre menys errors. Per exemple, utilitzant l'editor `gedit`, podeu configurar-lo per:

- Ressaltar la sintaxi del llenguatge. De forma automàtica, utilitzarà diferents colors per a ressaltar les paraules reservades del llenguatge, constants, comentaris, ...

- Indentar el codi automàticament. Si una línia conté el símbol {, l'inici de la següent línia serà desplaçat vàries columnes a la dreta. De la mateixa manera, si una línia conté el símbol }, l'inici de la següent línia serà desplaçat vàries posicions a l'esquerra.

- Ressaltar els parèntesi, claus,... Per exemple, quan escriviu el caràcter), l'editor ressalta durant uns instants quin és el (que es troba al mateix nivell.

Totes aquestes opcions poden ajudar a detectar errors sense passar pel compilador. Per exemple, si escriviu malament una paraula reservada veureu que no canvia de color; si us oblideu un símbol { veureu que la indentació del programa no és la que esperaríeu,...

Per configurar el `gedit` heu de posar-lo en marxa (passeu com a paràmetre un nom de fitxer amb extensió .c) i, utilitzant el menú *Preferences*, activar l'opció *Highlight Syntax*; altres opcions interessants són *Statistics Line*, *Show Matching* i *Auto Indent*. Totes aquestes opcions poden ser guardades a un fitxer per a que s'activin automàticament el proper cop que poseu en marxa l'editor.

➢ **Activitats**

1. Busqueu de quina forma podeu anar directament a una línia del fitxer que esteu editant (per exemple, en un fitxer de 1000 línies, anar a la línia 740).

2. Verifiqueu si l'editor té algun mecanisme per desfer els darrers canvis introduïts al fitxer en edició.

Indentació de programes

A mesura que anem modificant un programa, especialment programes molt llargs, és possible que el codi esdevingui incomprensible. Per evitar-ho hem de procurar mantenir el programa correctament indentat. És a dir tabulat de manera que es vegi l'estructura per blocs de codi (sovint delimitats per { }) clarament definida. També existeixen eines que fan aquesta feina automàticament, són útils si per exemple algú ens passa un programa que no esta indentat al nostre gust (una eina d'aquest tipus és `indent`).

➢ **Activitat**

3. Al directori de l'assignatura podeu trobar el fitxer `int1.c`. Aquest programa no està indentat. Edita'l i indenta'l correctament.

Compilació/Muntatge

Un cop escrit el programa en llenguatge C, cal generar el fitxer executable corresponent. Les eines que s'encarreguen d'aquesta tasca són el compilador i el muntador. Per simplificar la nostra tasca, farem que el compilador s'encarregui d'invocar automàticament al muntador, d'aquesta forma, amb una única comanda obtindreu el fitxer executable.

El compilador de llenguatge C s'anomena `cc` (*C compiler*) i rep com a paràmetres el nom(s) del(s) fitxer(s) amb codi font que composen la nostra aplicació. Si el compilador detecta algun error sintàctic, ens mostrarà un missatge d'error i el nombre de línia on el compilador es pensa que es troba l'error. Si no hi ha cap error, es genera un fitxer executable que rep el nom `a.out`. Si voleu que el fitxer executable tingui un altre nom, heu de passar al compilador els paràmetres `-o` i el nom que voleu que tingui l'executable. Com a exemple, per a obtenir un executable anomenat int1 corresponent al fitxer font `int1.c` cal executar la comanda `cc -o int1 int1.c`.

Un cop obtingut el fitxer executable, per a invocar-lo heu d'escriure el seu nom. Si el *shell* mostra l'error *Command not found*, haureu de revisar que la variable `PATH` contingui el directori actual o haureu d'invocar l'executable fent `./nom_executable` per forçar que el *shell* el busqui al directori actual.

Depuració

El *debugger* (depurador) és una eina que ens permet seguir l'execució d'un programa instrucció a instrucció, així com examinar i modificar el contingut de les variables del nostre programa.

Per defecte, el compilador genera un executable que és depurable únicament en llenguatge ensamblador. Per poder depurar en un llenguatge d'alt nivell cal utilitzar l'opció `-g` del compilador. Per exemple: `cc -g -o int1 int1.c`

Un cop generat l'executable, podeu posar en marxa el *debugger*. Utilitzareu un depurador en mode gràfic anomenat `ddd`, tot i que també existeixen depuradors en mode text com el `gdb`. Invocareu al `ddd` passant-li com a paràmetre **el nom del fitxer executable a depurar**. Per exemple: `ddd int1`

Obtindreu una finestra amb el codi font del vostre programa. Típicament, ara hauríeu de definir el(s) punt(s) d'aturada (*breakpoints*), és a dir, en quin punt (s) del vostre codi us interessa examinar el contingut de les variables o us interessa fer una execució pas a pas. Per definir un *breakpoint* només cal fer un doble *click* a l'inici de la línia de codi font en qüestió.

Ara podeu posar en marxa el programa. Podeu fer-ho prement el botó *Run* o al menú *Program* l'opció *Run* (d'aquesta manera podríeu passar paràmetres al programa). El programa començarà a executar-se i es detindrà quan arribi a algun *breakpoint*. Una fletxa marcarà el punt del codi on es troba.

En aquest moment, podeu examinar el valor de les variables. El més senzill és situar el punter del ratolí sobre alguna referència a la variable en qüestió. En breus moments apareixerà el seu valor. Una altra opció es fer doble *click* i apareixerà el seu valor a una nova finestra. Des d'aquesta finestra podeu modificar el valor de la variable.

Per continuar l'execució del programa teniu diverses opcions:

- El botó *Cont* continua l'execució fins arribar a un nou *breakpoint*.
- El botó Step executa el programa fins que arriba a una altra línia de codi (de la rutina actual o d'una altra rutina).
- El botó Next executa el programa fins que arriba a la línia següent de la rutina actual (en el cas d'estar a la darrera línia, a la línia següent on es va invocar a la rutina).

També podeu definir *breakpoints* condicionals, és a dir, que només provoquin l'aturada si una variable té un determinat valor. Altres possibilitats són veure els diferents blocs d'activació presents a la pila (opció *Backtrace* del menú *Status*) o definir *watchpoints* (com breakpoints però no associats a que el programa arribi a una línia de codi sinó associats a que una variable es modifiqui, sigui llegida, escrita o assoleixi un determinat valor).

➢ Activitat

4. Executeu amb el depurador els programes `int1.c`, `int2.c` i `int3.c`.

Pas de paràmetres

L'execució de programes en un entorn Unix facilita el pas de paràmetres. De la mateixa manera que les comandes del shell reben paràmetres, els programes que escrivim nosaltres també en poden rebre. Internament, aquests arguments els recull la funció main:

```
int main(char argc,char *argv[]){
...
exit(status);
}
```

Al moment de començar l'execució del programa el paràmetre `argc` conté el nombre de paràmetres rebuts comptant el nom del programa i `argv` és una llista d'strings amb els noms (cadenes ascii) dels arguments. Així, `argv[0]` correspon al nom del programa, `argv[1]` és el primer paràmetre, etc. Dins del main el tractament dels arguments el farem mitjançant les funcions de tractament d'strings (`strcmp, stcpy, atoi`, etc).

- **Activitat**
  5. Editeu el programa `int4.c` i intenteu predir el seu comportament. A continuació compileu-lo i executeu-lo diverses vegades amb diferent nombre de paràmetres, comproveu que el resultat correspon amb les vostres prediccions.

Implementació dels jugadors

Per a començar a practicar amb el llenguatge C, podeu començar a pensar en una part de la pràctica que haureu d'implementar al llarg del curs. Es tracta d'un jugador del joc del 7 i mig.

Haureu d'escriure un programa en llenguatge C de la forma més modular possible, fent ús de funcions externes al main del vostre programa per a fer més aclaridor el codi. El programa, que anomenarem `jugador1.c`, anirà obtenint cartes de forma aleatòria. Un cop obtinguda cada carta, el programa decidirà si ja s'ha passat de 7.5, si té exactament un 7.5, si vol obtenir una nova carta o si es planta.

En el directori de l'assignatura trobareu un fitxer executable anomenat `jug1`. El vostre programa s'hauria de comportar d'una manera similar.

En particular, en aquest programa haureu d'implementar les següents rutines, ja que després caldrà modificar aquestes parts per a que el servidor o el client les utilitzin:

a) **Obtenció de les cartes**: el codi relacionat amb generar una carta aleatòriament ha d'estar dins d'una rutina anomenada **`int generar_carta()`**, la qual retorna un número aleatori de l'1 al 10.
b) **Decidir si continuem jugant o ens plantem**: aquest codi ha d'estar dins de la rutina **`int decidir_continuar(...)`**, la qual retorna 1 si continuem jugant o 0 si ens plantem. Podeu adoptar els criteris que creieu més adients per a prendre la decisió, així com passar-li tants paràmetres com us sigui necessari.
c) **Escriptura de missatges per la sortida estàndard**: podeu utilitzar la rutina `printf`; més endavant utilitzarem la crida al sistema `write`. De totes maneres,

és molt recomanable que tot el codi relacionat amb aquest apartat s'implementi fent ús de rutines, ja que després les modificarem. Així doncs tindrem la rutina **`void escriure_msg_estat(...)`** amb tants paràmetres com necessiteu per a mostrar per la sortida estàndard, després de demanar cada carta, el valor previ que teníem acumulat, quina carta hem obtingut i la suma total que ens queda; i la rutina amb **`void escriure_msg (char *s)`** per a la resta de missatges que vulgueu mostrar.

Consideracions:

- Generació de nombres aleatoris: la rutina `rand()` genera un nombre aleatori entre 0 i un cert valor màxim (típicament 2147483647) . Si voleu obtenir un valor entre `0 i M-1`, només cal que utilitzeu la sentència `rand()%M`.
- Inicialització dels nombres aleatoris: per a que cada execució del jugador pugui generar una seqüència diferent de nombres aleatoris, és precís que al començament del programa invoqueu a la rutina `srand()` passant-li com a paràmetre un nombre "aleatori", com per exemple el resultat de la crida al sistema `getpid()`.
- Codificació de les cartes: hi ha 10 cartes possibles (as, 2, 3, 4, 5, 6, 7, sota, cavall i rei) i cadascuna té associat un valor (1, 2, 3, 4, 5, 6, 7, 1/2, 1/2 i 1/2 respectivament).
- Depuració: podeu utilitzar el `ddd` per a depurar el vostre programa.

Activitats extres

- L'eina `make` simplifica el procés de compilació i muntatge d'aplicacions. Resulta especialment útil quan les aplicacions estan formades per molts fitxers font. A la pàgina web de l'assignatura podeu trobar una breu introducció: `make.ppt`

- Al directori de l'assignatura podeu trobar una versió del joc *tetris* implementada amb un programa "ofuscat" que té menys de 1500 caràcters. Analitzeu aquest codi i intenteu entendre la seva implementació.

Sessió 4: Processos (1/3)

L'objectiu d'aquesta sessió és familiaritzar-se amb les crides al sistema de gestió de processos (`fork`, `wait`, `exec`, `exit`, `getpid`, `getppid`) tot executant i/o modificant petits programes de prova que trobareu al directori de l'assignatura. En cas de dubte respecte la funcionalitat de les crides, sempre podeu consultar el manual del sistema (comanda `man`).

Crida al sistema `fork`

Aquesta crida és la responsable de la creació de nous processos. Crea un nou procés (procés fill) que és una rèplica del procés que ha invocat a la crida `fork` (procés pare); és a dir, l'espai de memòria del procés fill (codi, pila i dades) s'inicialitzarà amb el contingut que té el pare. Un cop creat el procés fill, el procés pare i el procés fill s'executen concurrentment a partir de la instrucció següent a la invocació a la crida fork, pero cadascú treballarà amb la seva pròpia còpia de les variables.

La crida `fork` retorna un valor diferent al procés pare i al procés fill. La crida fork retorna un 0 al procés creat i l'identificador del procés creat al procés pare. Això permetrà separar el codi que executarà el pare del que executarà el fill.

➢ Activitats

1. Compileu i executeu el programa `ex01.c`. Com és que el missatge del segon `printf` apareix dos cops per pantalla? (Al programa s'ha utilitzat la crida al sistema `getpid`, que retorna l'identificador del procés que la crida).
2. Compileu i executeu el programa `ex02.c`. Comproveu que el valor de retorn és diferent per a cada procés. Per què es dóna aquest fet?
3. Compileu i executeu el programa `ex03.c`. Comproveu que el valor de retorn és diferent per a cada procés.
4. Compileu i executeu el programa `ex04.c`. Com és que el procés pare no acaba mai?
5. Compileu i executeu el programa `ex05.c`. Quants processos es creen? Quina relació pare/fill existeix entre ells? Fes un esquema de processos que reflecteixi la jerarquia familiar creada. (Al programa s'ha utilitzat la crida al sistema `getppid`, que retorna l'identificador del procés pare del que invoca a la crida).
6. Compileu i executeu el programa `ex06.c`. Quants processos es creen? Quina relació pare/fill existeix entre ells? Fes un esquema de processos que reflecteixi la jerarquia familiar creada. Què passa si elimineu la línia `sleep(1)`?
7. Compileu i executeu el programa `ex07.c` Justifiqueu el resultat obtingut, dibuixant la jerarquia de processos creada i indicant clarament qui escriu cadascun dels missatges.
8. Compileu i executeu el programa `ex08.c` Justifiqueu el resultat obtingut. Podríem haver obtingut un resultat diferent? Per què?

Crida al sistema `wait`

Aquesta crida permet sincronitzar l'execució d'un procés pare amb l'acabament d'un dels seus processos fills.

La crida al sistema `wait` retorna l'identificador del procés fill que ha acabat. Això pot ser útil si un procés té varis processos fills.

- **Activitats**

9. Compileu i executeu el programa `ex11.c`. Comproveu com es produeix la sincronització.

10. Escriu un programa anomenat `ex12.c` que creï dos processos fills i determini quin dels dos acaba primer, mostrant un missatge per la sortida estàndar. Cada fill s'adormirà el seu `PID` mòdul 10 segons, escrivint un missatge per la sortida estàndar indicant quant temps estarà adormit (podeu fer servir la crida a sistema `sleep(N)` per a resoldre l'exercici).

Crida al sistema `exec`

Aquesta crida permet canviar la imatge d'un procés, és a dir, carregar un nou programa a memòria i inicialitzar la zona de dades i la de pila per a començar la seva execució.

Per invocar aquesta crida cal especificar el nom del fitxer executable i els paràmetres de l'execució; per conveni, el nom de l'executable també és un paràmetre. Per tant, per executar "`ls -l -a`", haurieu d'especificar el nom de l'executable ("`ls`") i els paràmetres (`"ls", "-l" i "-a"`).

- **Activitats**

11. Compileu i executeu el programa `ex21.c` Comproveu com s'invoca a l'executable "`ls`". Com és que no apareix el missatge escrit amb el segon `printf`?

12. Consulteu al manual les diferències entre les crides `execl, execlp, execle, execv, execvp i execve`.

Crida al sistema `exit`

Aquesta crida permet indicar explícitament l'acabament d'un procés (fins ara, tots els processos acabaven de forma implícita ja que arribaven al final de la rutina `main`).

La crida té un paràmetre de tipus enter que es pot utilitzar per a que el procés fill es comuniqui amb el procés pare. El procés pare podrà obtenir aquest valor mitjançant la crida al sistema `wait`.

- **Activitat**

13. Compileu i executeu el programa `ex31.c` Comproveu com existeix una comunicació entre el fill i el pare.

➢ Activitats extres

- Escriu un programa que anomenarem `ex41.c` que rebi com a paràmetre un número. El programa escriurà el número i el seu PID per la `stdout`. En cas de que aquest número sigui més gran que 0, s'executarà a sí mateix, passant-se com a paràmetre el número decrementat en una unitat.
- Escriu un programa que anomenarem `ex42.c` que rebi com a paràmetre un número. El programa ha de crear tants fills com l'argument que ha rebut. Els fills s'executaran concurrentment. Cadascun d'ells multiplicarà el número pel seu `PID` mòdul 10 i al morir el passarà al seu pare. Aquest, a mesura que els seus fills vagin morint, escriurà per la `stdout` el PID del fill mort i el valor que havia calculat.

Sessions 5 i 6: Processos

L'objectiu d'aquestes sessions és que desenvolupeu una aplicació que utilitzi les crides al sistema relacionades amb la gestió de processos (`fork, exec, wait i exit`). Al llarg del curs, anirem afegint funcionalitats a aquesta aplicació, de forma que acabi utilitzant tot el repertori de crides al sistema que veureu a l'assignatura. A partir d'ara, heu de realitzar la comprovació d'errors de totes les crides a sistema.

Descripció general de l'aplicació

L'aplicació que desenvolupareu al llarg del curs simularà una partida entre N jugadors del joc de cartes del set i mig. Aquesta aplicació estarà composada per varis processos i seguirà un esquema de funcionament client/servidor on els processos clients seran els jugadors i un procés servidor s'encarregarà de distribuir les cartes i de gestionar estadístiques.

Per a desenvolupar l'aplicació us proposem un seguit de passos de forma que les noves funcionalitats a implementar a cada pas estaran molt delimitades.

Pas 1: Creació de N jugadors seqüencials

Ara us encarregareu de que es creïn fins a `N` processos jugadors. Haureu d'implementar un programa anomenat `servidor2.c` que rebrà per la línia de comandes un paràmetre numèric. Aquest paràmetre representa el nombre de processos jugador que s'han de crear. Cadascun d'aquests carregarà el fitxer executable jugador1, que vau implementar en la sessió 3. Els processos jugadors s'executaran de forma seqüencial, és a dir, el jugador `i-èssim` es crearà quan el jugador `(i-1)-èssim` hagi acabat la seva execució.

Al directori de l'assignatura trobareu un fitxer executable anomenat `serv2`. El vostre programa s'hauria de comportar d'una manera similar.

Consideracions:

- Conversió de cadena de caràcters a enter: podeu utilitzar la rutina `atoi` per a conéixer el valor numèric de la cadena de caràcters que conté el nombre de processos jugadors.
- Tractament d'errors: al fitxer `panic.c` del directori de l'assignatura hi ha una rutina de tractament d'errors que podeu utilitzar als vostres programes. La rutina s'anomena `panic` i té la mateixa interfície que el `printf`. Un cop que ha escrit el missatge indicat per l'usuari, escriu el missatge d'error propi del sistema operatiu i acaba l'execució del procés.

Pas 2: Generació de N jugadors concurrents

Ara modificareu el codi anterior de forma que els processos fills s'executin de forma concurrent. A aquest nou programa l'anomenarem `servidor3.c` Al directori de l'assignatura trobareu un fitxer executable anomenat `serv3`. El vostre programa s'hauria de comportar d'una manera similar.

Per a verificar que funciona correctament podeu posar la sentència `sleep(N)` on `N` és el nombre de segons que ens quedarem bloquejats en aquesta crida (poseu un nombre aleatori), abans de que cada fill canviï la imatge del seu programa.

Pas 3: Generació de resultats per part del procés pare

Fins ara, el procés pare no té coneixement de com han acabat els processos jugadors. En aquest pas haureu de modificar els processos jugadors de forma que, en el moment d'acabar, indiquin al procés pare a quin valor s'han plantat o si s'han passat. D'aquesta manera, el procés pare podrà indicar quin jugador ha guanyat la partida (en cas d'empat entre varis jugadors, podeu assumir que el guanyador és el procés jugador amb *pid* més baix).

Aquests programes els anomenarem `servidor4.c i jugador4.c` Al directori de l'assignatura trobareu dos fitxers executables anomenats `serv4 i jug4`. Els vostres programes s'haurien de comportar d'una manera similar.

Consideracions:

- Tingueu en compte que com l'execució és concurrent: el primer fill que es crea no té per què ser el primer fill que acaba.
- Tot el codi relacionat amb la cerca del jugador guanyador ha d'estar dins la rutina **`void decidir_guanyador(...)`** amb els paràmetres que creieu convenients. Aquesta rutina decidirà quin és el jugador guanyador i ho escriurà en el fitxer `resultats.txt` El format d'aquest fitxer ha de ser similar a:

```
Resultats finals del joc set i mig
----------------------------------
El Jugador <PID> s'ha plantat a <puntuació final>
El Jugador <PID> s'ha passat
...
...
...
El guanyador és <pid> amb <puntuació final>!
ó bé
Cap guanyador: Tots els jugadors s'han passat!
```

Sessió 7: Entrada/Sortida (1/3)

L'objectiu d'aquesta sessió és familiaritzar-se amb les crides al sistema d'entrada/sortida sobre fitxers i *pipes* (`open`, `read`, `write`, `close`, `lseek`, `dup`, `pipe`) tot executant i/o modificant petits programes de prova que trobareu al directori de l'assignatura. En cas de dubte respecte la funcionalitat de les crides, sempre podeu consultar el manual del sistema (comanda `man`).

Crides al sistema `read` i `write`

La crida `read` és la responsable de llegir dades d'un canal (descriptor de fitxer) i portar-les a memòria. La crida `write` és la responsable d'escriure dades emmagatzemades a memòria sobre un canal.

Totes dues crides estan parametritzades de la mateixa manera: el canal sobre el que volem llegir (escriure), la direcció de memòria a partir de la que volem emmagatzemar les dades que es llegeixin (o on es troben les dades a escriure) i el nombre màxim de bytes que volem llegir (escriure). La crida retorna el nombre de bytes que realment s'han llegit (escrit); si la crida `read` retorna que ha llegit 0 bytes, indica que s'ha arribat al final del fitxer. Les dues crides assumeixen que els fitxers s'accedeixen seqüencialment, és a dir, el primer `read` d'un byte sobre un canal ens retorna el primer byte del fitxer corresponent, el segon `read` ens retornaria el segon byte, i així successivament.

```
int read(int fd, char *addr, int nbytes);
int write(int fd, char *addr, int nbytes);
```

Tot procés creat des del *shell* té oberts tres canals:

- El canal 0 (també anomenat canal d'entrada estàndard). Per defecte, està associat al teclat, però utilitzant el metacaràcter < indiquem al shell que l'associï a un altre fitxer (es á dir, estaríem redireccionant el canal d'entrada estàndard).
- El canal 1 (també anomenat canal de sortida estàndard). Per defecte, està associat a la pantalla, però utilitzant el metacaràcter > indiquem al shell que l'associï a un altre fitxer (és a dir, estaríem redireccionant el canal de sortida estàndar).
- El canal 2 (també anomenat canal d'error). Per defecte, associat a la pantalla.

Inicialment, provareu un programa que llegeix i escriu sobre aquests canals estàndard. Es tracta d'un programa que converteix les lletres majúscules en minúscules i a l'inrevés.

➢ Activitats

1. Compileu el programa `es01.c`. Executeu-lo redireccionant l'entrada estàndard cap al fitxer `prova1.txt` (`es01 < prova1.txt`) i executeu-lo redireccionant l'entrada i la sortida estàndard cap a dos fitxers; comproveu que el contingut del fitxer és l'esperat.

2. Executeu el programa `es01` sense redireccionar els canals estàndard. A mesura que introduïu línies, s'aniran convertint. Per a indicar al sistema que ja no hi ha més caràcters, cal prémer Ctrl D; això provoca que la crida al sistema read sobre tecla retorni que ha llegit 0 caràcters. Quina és la diferència que hi ha entre prémer Ctrl C i Ctrl D?

Crides al sistema `open` i `close`

Si un procés necessita llegir o escriure d'altres fitxers que no siguin els associats als canals estàndard, ha d'obrir aquests fitxers. La crida responsable d'aquesta tasca és la crida `open`. Aquesta crida està parametritzada amb el nom del fitxer i el mode amb el qual el volem obrir (`O_RDONLY`, `O_WRONLY` o `O_RDWR` o opcionalment `O_CREAT`, `O_TRUNC` o `O_APPEND`); si tot és correcte, la crida `open` retorna el canal associat al fitxer obert. Les posteriors lectures i/o escriptures sobre el fitxer hauran d'utilitzar el canal retornat per la crida al sistema `open`.

La crida `close` és la que ens permet indicar al sistema que el procés ja no vol utilitzar més el fitxer associat a un determinat canal.

➢ Activitats

3. Compileu el programa es11.c. Aquest programa és una nova versió del programa anterior; ara admet com a paràmetre la llista de fitxers a convertir (per exemple, el podríeu invocar fent es11 es01.c prova1.txt es11.c). El programa ha d'anar obrint tots els fitxers especificats per l'usuari.
4. Escriu un programa anomenat es12.c que copiï la seva entrada estàndard sobre la seva sortida estàndard i, a més a més, sobre un fitxer que li passarem com a paràmetre. Explica el tractament que hauria de fer el teu programa per permetre que des de la línia de comandes es pugui redireccionar l'entrada i/o la sortida estàndard del programa com en el següent exemple:

```
pinguino_% es12 cccc < aaaa > bbbb
```

Generarà dos fitxers, el `bbbb` i el `cccc` tots dos amb el mateix contingut que `aaaa`.

Crida al sistema `lseek`

Tot fitxer obert té associat un punter que indica la posició sobre la que es farà la següent operació de lectura/escriptura. Després de cada lectura/escriptura, aquest punter s'incrementa amb el nombre de caràcters llegits/escrits. Per tant, el funcionament d'aquest punter està pensat per accedir seqüencialment als fitxers.

La crida `lseek` ens permet modificar el valor d'aquest punter. La modificació es pot fer respecte la posició actual del punter (`SEEK_CUR`), respecte la darrera posició del fitxer (`SEEK_END`) o respecte l'inici del fitxer (`SEEK_SET`). Consulteu el manual del sistema per a veure tots els paràmetres de la crida, la seva interpretació, i quin resultat retorna.

➢ Activitats

5. Compileu i executeu el programa `es21.c`. Aquest programa escriu el segon caràcter d'un fitxer.
6. Escriviu un programa anomenat `es22.c` que, utilitzant la crida al sistema `lseek`, permeti obtenir la mida d'un fitxer.
7. Escriviu un programa anomenat `es23.c` tal que mostri per la sortida estàndard cada línia d'un fitxer d'entrada passat com a paràmetre, dos cops. No es pot fer servir un buffer de mida superior a un caràcter.

```
pinguino_% more file.txt
aaaa
bbb
pinguino_% es23 file.txt
aaaa
aaaa
bbb
bbb
```

➢ Activitats extres

- Escriviu un programa anomenat `es24.c` tal que mostri per la sortida estàndard les línies amb més de `N` caràcters (Es considera que el salt de línia forma part de la línia). El nom del fitxer i el nombre `N` els rebrem com a paràmetres. No es pot fer servir un buffer de mida superior a un caràcter.

```
pinguino_% more file.txt
aaaa
bbb
pinguino_% es24 file.txt 4
aaaa
```

Consideracions d'eficiència

Els programes que heu utilitzat fins ara llegien i escrivien caràcter a caràcter tot i que les crides `read i write` permeten llegir i escriure varis caràcters simultàniament. A alguns programes, el fet de llegir caràcter a caràcter pot tenir repercussions en el temps d'execució del programa. Penseu que cada cop que s'invoca al sistema operatiu, aquest ha de fer un seguit d'operacions que fan augmentar el temps d'execució del programa.

➢ Activitats

8. Compileu i executeu el programa `es31.c` Aquest programa llegeix vàries vegades el contingut d'un fitxer gran i l'escriu sobre un altre fitxer; la primera lectura del fitxer utilitza un buffer de 64Kb, la segona utilitza un buffer de 32 Kb, i així successivament fins a un buffer d'1 bytes.

Relació entre la crida al sistema `open` i les crides al sistema `fork` i `exec`

La crida al sistema `fork` crea un procés fill que és una còpia quasi idèntica del procés pare. El procés fill tindrà oberts els mateixos canals que el procés pare. Una qüestió important és que el pare i el fill compartiran el punter de lectura/escriptura sobre els fitxers oberts **abans** de fer la crida al sistema `fork`. Per tant, les operacions que qualsevol dels dos processos faci sobre aquest fitxer modificaran el valor del punter que utilitza l'altre procés.

➢ Activitats

9. Compileu i executeu el programa `es41.c` Justifiqueu el resultat obtingut.
10. Compileu i executeu el programa `es42.c` Justifiqueu el resultat obtingut.
11. La crida al sistema `exec` manté els canals oberts que té el procés. Escriviu un programa anomenat `es43.c` que posi aquest fet de manifest, simulant el que fa la shell al executar la comanda

    ```
    pinguino_% ls -la > tmp.txt
    ```

    Per a fer-ho primer heu de realitzar el redireccionament mitjançant les crides `open/close` i després invocar a la crida a sistema `exec` amb els paràmetres adequats.

➢ Activitats extres

- Compileu i executeu el programa `es44.c` Justifiqueu el resultat obtingut.
- Escriu un programa anomenat `es45.c` que tregui sobre la `stdout` el contingut d'un fitxer passat com a paràmetre però invertit (és a dir, començant pel final).
- Escriu un programa anomenat `es46.c` que mostri sobre la `stdout` el resultat d'alinear a la dreta el contingut d'un fitxer passat com a paràmetre; és a dir, haurà d'insertar el nombre adient d'espais en blanc a l'inici de cada línia de forma que totes les línies acabin tenint la mateixa llargada. No es pot fer servir un buffer de mida superior a un caràcter. Per exemple,

```
pinguino_% more file.txt
aaaaaa
bbb
pinguino_% es46 file.txt
aaaaaa
bbb
```

Crida al sistema `pipe`

Una *pipe* és un dispositiu d'emmagatzematge *FIFO* que permet comunicar dos o més processos. La crida al sistema `pipe` s'encarrega de crear una *pipe* ordinària. La crida ens retorna dos descriptors, un que ens permetrà llegir de la *pipe* i un altre que ens permetrà escriure-hi.

Per a que un altre procés pugui accedir a aquesta *pipe*, cal que el procés hereti l'accés mitjançant la crida al sistema `fork`.

➢ Activitats

12. Compileu i executeu el programa `es51.c`. Comproveu com el pare i el fill s'intercanvien dades utilitzant la *pipe*.
13. La crida al sistema `read` té un comportament peculiar quan intenta llegir d'una `pipe` i resulta que està buida. Llegint el manual i escrivint programes de prova, intenteu deduir quin és el comportament de la crida al sistema `read` quan llegeix d'una *pipe* buida. Contempleu dos casos: quan existeixen algun/s escriptors sobre la `pipe` i quan no n'existeix cap.
14. Volem escriure un programa que faci el mateix que la següent seqüència de comandes de UNIX:

    ```
    pinguino_% ls | wc > file
    ```

    Per a fer-ho, ens ofereixen en programa `es52.c` Comproveu que el resultat esperat és incorrecte. Busca on estan els errors i modifica el programa per a que faci el que volem.

➢ Activitats extres

- Escriu un programa anomenat `es53.c` que creï tants fills com se li indiqui en la línia de comandes (paràmetre del programa). Cadascun d'aquests fills li enviarà al seu pare el seu identificador de procés mitjançant una `pipe` i a continuació morirà. El pare mostrarà per la stdout de manera explicativa tots els identificadors que li arribin. Es valorarà l'ús mínim de recursos del sistema.
- Indiqueu justificadament quina estructura de processos crea el programa `es54.c` Mostra també com estan comunicats els processos creats, assumint que cap crida a sistema retorna error.

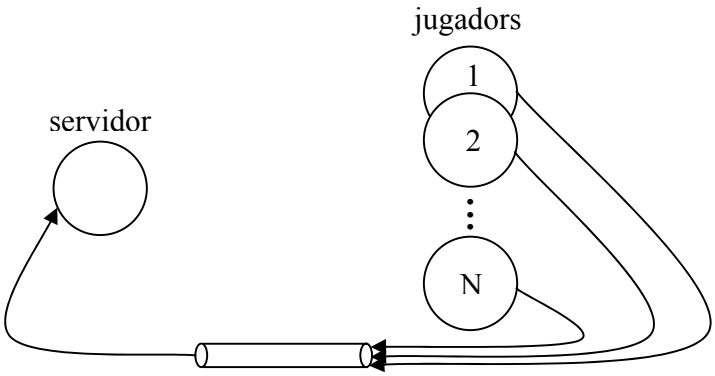
Sessions 8 i 9: Entrada/Sortida

L'objectiu d'aquestes sessions és que afegiu una sèrie de funcionalitats relacionades amb entrada/sortida al programa jugador i al programa servidor del joc del set i mig que vàreu desenvolupar a les sessions 5 i 6. Caldrà que utilitzeu les crides al sistema bàsiques d'entrada sortida (`open, read, write, close, pipe`).

Pas 1: Enviar puntuació final utilitzant una única *pipe*

Fins ara, quan un jugador moria indicava al seu procés pare a quina puntuació havia arribat (o si s'havia passat). Aquesta comunicació es feia mitjançant les crides `exit` i `wait`.

En aquest primer pas fareu que cada jugador **a més a més** enviï aquesta mateixa informació al seu pare utilitzant una *pipe*. La següent figura mostra com haurien d'estar comunicats els processos:



Abans de morir, cada jugador escriurà a la *pipe* quina ha estat la seva puntuació final. El procés pare haurà de comprovar que les dades llegides de les *pipes* coincideixen amb les rebudes per la crida `wait`, mostrant un missatge per la `stdout` indicant si són correctes o no.

Els programes s'anomenaran `servidor5.c i jugador5.c` En el directori de l'assignatura trobareu els executables `serv5 i jug5`. Els vostres programes s'haurien de comportar d'una manera similar.

Consideracions:

Utilitzar una única pipe us introdueix varis problemes:

- Caldrà identificar a quin procés pertany cada puntuació final. Per tant, cada jugador haurà d'escriure a la *pipe* el seu resultat final i el seu identificador (el seu

pid). D'aquesta manera, el procés pare podrà saber a quin jugador correspon cada puntuació.

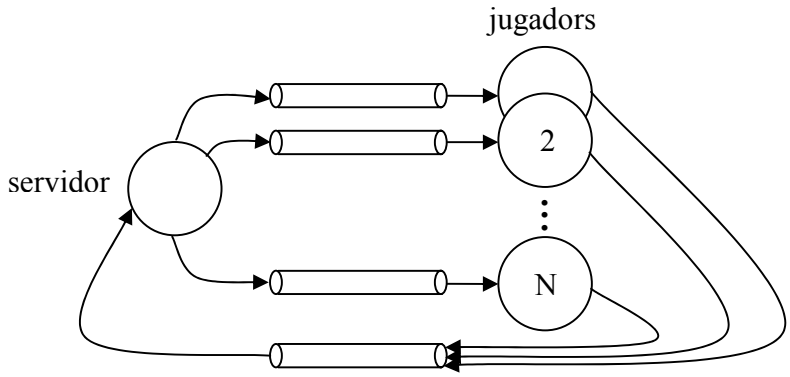
- Cal tenir present que varis processos poden estar escrivint a la *pipe* simultàniament, per tant, haureu de garantir que no hi hagi problemes de concurrència (per exemple, evitar que entre que un procés escriu el seu identificador i la seva puntuació, un altre procés escrigui alguna cosa a la *pipe*).

Raona primer com es transmetrà la informació de fills a pares: en quin format i en quants reads / writes.

- Per últim, recorda que, com ara ja coneixem les crides a sistema relacionades amb l'entrada i sortida, hauràs de substituir els `printf` per la crida a sistema `write` amb els paràmetres corresponents.

Pas 2: Distribució de cartes utilitzant N *pipes*

Fins ara, cada jugador es generava les seves pròpies cartes. Ara fareu que sigui el procés pare qui les generi i les vagi enviant als jugadors. Per comunicar les cartes entre el pare i els jugadors caldrà disposar d'una *pipe* per cada jugador. Cada `pipe` serà creada en el moment que la necessitem (és a dir, no les creeu totes al començament de cop). La següent figura mostra com haurien d'estar comunicats els processos:



A la primera ronda del joc, el pare generarà aleatòriament N cartes (una per cada jugador) i escriurà cadascuna d'aquestes cartes a la *pipe* corresponent. Per a fer-ho podeu copiar la rutina `int generar_carta()` desenvolupada en el programa `jugador1.c` tal i com està, i modificar-la en aquest per a que llegeixi la carta enviada pel pare.

Un cop que cada jugador hagi processat la seva carta, cada jugador indicarà al seu pare si vol rebre més cartes, si es planta o si s'ha passat (és a dir, heu d'afegir un codi de control a la informació que el jugador enviava en el pas1: `0,1 o 2` respectivament). Per a enviar aquesta informació utilitzareu la *pipe* que heu desenvolupat al pas anterior.

Quan el pare tingui totes les respostes, tornarà a generar aleatòriament cartes però únicament pels processos que segueixin en el joc. Aquests passos es repetiran fins que cada jugador s'hagi plantat o s'hagi passat.

Heu de tenir en compte que, a cada ronda, l'ordre de recepció de les respostes dels jugadors no ha de coincidir necessàriament amb l'ordre d'enviament de les cartes als jugadors.

Anomenarem a aquests programes `servidor6.c i jugador6.c`, els quals haurien de comportar-se com els executables `serv6 i jug6` que trobareu en el directori de l'assignatura.

Consideracions:

- Si ho preferiu, en aquest apartat podeu treure la comprovació dels resultats rebuts per la `pipe` i els obtinguts pel `wait` que heu desenvolupat en el pas anterior. Ens quedarem amb els resultats de la `pipe`.
- A partir d'ara, l'identificador de cada jugador serà el seu índex de creació (no el seu PID). Per a fer-ho haureu de modificar tant el codi del procés pare com el del jugador. Primer de tot, a l'inici del programa, el pare escriurà a cada pipe l'identificador del jugador i paral.lelament el jugador primer de tot haurà de llegir-lo. A més, tots els missatges que ell generi no començaran pel seu PID, sino enlloc d'ell, per aquest identificador. Ara el format del fitxer final de resultats serà similar al següent:

```
Resultats finals del joc set i mig
----------------------------------
El Jugador <ID> s'ha plantat a <puntuació final>
El Jugador <ID> s'ha passat
...
...
...
El guanyador és el jugador <id> amb <puntuació final>!
ó bé
Cap guanyador: Tots els jugadors s'han passat!
```

Sessió 10: Signals (1/2)

L'objectiu d'aquesta sessió és familiaritzar-se amb les crides al sistema relacionades amb signals POSIX (`sigaction, sigprocmask, kill, alarm, sigsuspend`) tot executant i/o modificant petits programes de prova. En cas de dubte respecte la funcionalitat de les crides, podeu consultar el manual del sistema (comanda `man`).

Introducció

Els signals són una facilitat oferta pel sistema operatiu pel tractament d'esdeveniments asíncrons. Conceptualment, són molt semblants a les interrupcions hardware: generació d'un esdeveniment, salvar l'estat del procés, passar el control a una rutina d'atenció al esdeveniment i restaurar l'estat. La major diferència radica en el tipus d'esdeveniments que provoquen els signals. Existeixen varis tipus de signals, cadascun d'ells:

• està identificat per un número enter i per un nom de constant que comença per `SIG` (per exemple, el signal 9 és el `SIGKILL`).
• està associat a un esdeveniment (per exemple, accés a memòria invàlid)
• té una rutina d'atenció per defecte (que normalment provoca la mort del procés) però, en la majoria dels casos, cada usuari pot definir la rutina d'atenció al signal.

Tipus de signals

L'estàndard POSIX defineix més de 30 tipus de signals, entre ells tenim el `SIGKILL, SIGALRM, SISUSR1,`... Inicialment, veureu diferents maneres d'obtenir la llista dels signals disponibles al nostre sistema i informació sobre ells.

➢ Activitats

1. Consulteu la pàgina de manual de la comanda `kill` (és la comanda que permet generar signals). Busqueu si la comanda té alguna opció que et doni aquesta informació.
2. Consulteu la pàgina de manual de la crida al sistema `sigaction`. Al final de la pàgina trobareu referències a altres pàgines relacionades. El format d'aquestes referències és nom pàgina(secció del manual). Busqueu la pàgina que conté la informació desitjada.
3. El fitxer capçalera associat als signals és el `signal.h` que es troba a `/usr/include/signal.h`. Mirant el seu contingut, i el dels fitxers capçalera inclosos en aquest fitxer, intenteu arribar al fitxer on apareix la llista de signals del sistema.

Generació de signals

Crida al sistema `kill`

La crida al sistema `kill` permet demanar al sistema operatiu que dipositi un tipus de signal sobre un procés; està parametritzada amb el tipus de signal a generar i amb l'identificador del procés destinatari del signal.

La comanda kill està implementada utilitzant la crida al sistema `kill` i ens permet dipositar signals des de l'intèrpret de comandes.

- **Activitats**

4. Utilitzant dues finestres, a un finestra proveu de crear un procés "lent" (per exemple, executeu la comanda `sleep 100`) i a l'altra proveu de generar diferents signals sobre el procés. Comproveu com el missatge que escriu el *shell* quan es diposita el signal sobre el procés depèn del tipus de signal que hagueu generat.
5. Compila i executa el programa `si01.c`, el qual necessita un paràmetre (`izquierda o derecha`). Descriu què fa el programa. Per què és necessari la utilització d'una pipe?

Crida al sistema `alarm`

La crida `alarm` programa el temporitzador associat a cada procés. Aquesta crida està parametritzada amb un nombre de segons, i provoca que es dipositi un signal `SIGALRM` quan aquest nombre de segons hagi transcorregut.

- **Activitat**

6. Compileu el programa `si02.c`. És una primera versió d'un programa que es comportaria com la comanda `sleep`. Executeu-lo indicant un nombre de segons i comproveu que es comporta com `sleep`. Qui escriu el missatge *Alarm clock*?, quan? Executeu-lo indicant un nombre de segons gran i des d'una altra finestra executeu varis cops la comanda `ps` per veure el temps de CPU que consumeix el procés; què està passant? Per què no té el mateix comportament la comanda `sleep`?

Conjunt de signals bloquejats

Per defecte, en qualsevol moment es pot dipositar qualsevol tipus de signal. En alguns casos pot ser interesant evitar temporalment el dipòsit d'un signal, típicament per a garantir la realització d'algunes operacions en exclusió mútua o per a garantir que algunes operacions acabin.

Per a cada procés, el sistema operatiu manté un conjunt de signals bloquejats, és a dir, els signals que actualment no es poden dipositar sobre el procés. La crida al sistema que ens permet obtenir/modificar aquest conjunt és la crida `sigprocmask` i està parametritzada amb l'operació a fer (bloquejar `SIG_BLOCK`, desbloquejar `SIG_UNBLOCK`

o assignar SIG_SETMASK), una variable d'entrada de tipus conjunt de signals i una variable de sortida de tipus conjunt de signals.

Per a manipular variables de tipus conjunt de signals (sigset_t) hi ha vàries rutines auxiliars: sigemptyset, sigfillset, sigaddset, sigdelset i sigismember.

- **Activitats**

7. Compileu i executeu el programa si11.c. Comproveu que bloqueja temporalment el signal associat a l'esdeveniment de prémer les tecles Ctrl C (SIGINT) i que, si premeu les tecles durant aquest temps, el procés no finalitza fins que es desbloqueja el signal.
8. Escriviu un programa anomenat si12.c que mostri per la stdout quins signals estan bloquejats en un moment donat.

Capturar un signal

La crida sigaction permet definir la rutina d'atenció a un signal. La crida està parametritzada amb el tipus de signal, una variable d'entrada de tipus struct sigaction que conté, entre altres coses, un punter a la nova rutina d'atenció, i una variable de sortida, del mateix tipus que l'anterior, on el sistema ens deixa informació sobre quin era l'anterior tractament al signal.

- **Activitats**

9. Compileu i executeu el programa si21.c. És una nova versió de la comanda sleep on es captura el signal SIGALRM. Qui escriu ara el missatge?
10. Compileu i executeu el programa si22.c. Aquest programa incrementa el valor d'una variable global en una unitat si rep un SIGUSR1 i en 2 si rep un SIGUSR2. Per què és necessari incloure els dos signals en el camp sa_mask de l'struct sigaction?

Esperar el dipòsit d'un signal

Finalment, la crida sigsuspend permet bloquejar un procés fins que es dipositi un signal concret, és a dir, el procés esperarà a que arribi un determinat signal sense consumir CPU.

- **Activitats**

11. Compileu i executeu el programa si31.c. És una nova versió de la comanda sleep on es bloqueja el procés. Per què creieu que s'ha bloquejat el SIGALRM abans de fer l'alarm?
12. Modifica el programa si22.c de manera que no fem cap mena d'espera activa. Aquest nou programa l'anomenarem si32.c.
13. Compileu i executeu el programa si33.c Justifica el resultat obtingut.

➢ Activitats extres

- Implementa un programa anomenat `si41.c` que busqui el caràcter 'z' en un fitxer durant un temps màxim. El nom del fitxer i el nombre màxim de segons durant el qual hem de realitzar la cerca, són passats com a paràmetres del programa.

Sessió 11: Signals (2/2)

L'objectiu d'aquesta sessió (i de la següent) és que afegiu una sèrie de funcionalitats relacionades amb signals al programa jugador i al programa servidor del joc del set i mig que heu desenvolupat. Caldrà que utilitzeu les crides al sistema bàsiques de signals (`sigaction, kill, sigprocmask, siguspend, alarm`).

Pas 1: Signal `SIGTERM`

Normalment, el significat associat al `SIGTERM` és que el procés ha d'acabar la seva execució tan ràpidament com sigui possible.

En el nostre cas, quan el procés pare rebi aquest signal haurà d'escriure un missatge informatiu, enviar un signal `SIGTERM` a tots els processos jugadors i invocar a la crida al sistema `exit`. Quan un procés jugador rebi el `SIGTERM` haurà d'escriure un missatge informatiu i invocar a la crida al sistema exit.

Podeu enviar un SIGTERM des de l'intérpret de comandes utilitzant la comanda

```
kill -<signal> <pid>
```

Pas 2: Signals `SIGALRM`

El signal `SIGALRM` està associat a l'expiració del temporitzador del procés. Utilitzarem el temporitzador per a que el procés pare escrigui cada N segons un missatge indicant el nombre de segons que han passat des de que s'ha iniciat la seva execució. N serà el segon paràmetre del programa.

Pas 3: Signals `SIGUSR1`

Utilitzarem el signal `SIGUSR1` per a que el procés pare comuniqui als jugadors que ha acabat d'escriure en el fitxer de resultats. Per tant, abans de finalitzar, els jugadors esperaran el dipòsit d'aquest signal sense fer cap mena d'espera activa. En tal cas, cada jugador llegirà el fitxer, el mostrarà per la sortida estàndar i finalitzaràla seva execució.

Fixeu-se que els jugadors llegeixen el fitxer de resultats concurrentment.

Consideracions:

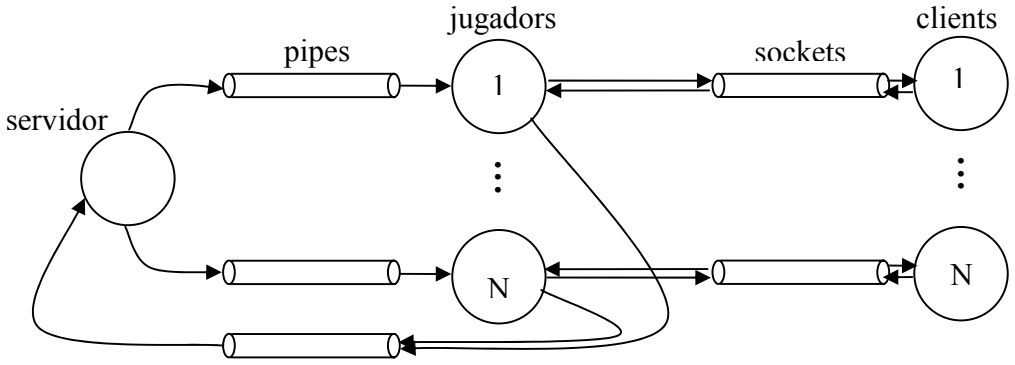
- Per a provar que el codi és correcte, feu que el procés pare invoqui a la rutina `sleep` abans o després d'acabar l'escriptura sobre el fitxer.

Als programes implementats amb les modificacions indicades en aquesta sessió, els anomenarem `servidor7.c i jugador7.c` En el directori de l'assignatura trobareu els executables `serv7 i jug7`. Els vostres programes haurien de tenir un comportament similar.

Sessió 12: Sockets

L'objectiu d'aquesta sessió és que afegiu una sèrie de funcionalitats relacionades amb sockets per a aconseguir que el joc del set i mig que heu anat desenvolupant pugui ser executat remotament seguint el model client/servidor. Caldrà que utilitzeu les crides al sistema bàsiques de sockets (`socket, bind, listen, connect, accept`).

Per a això és necessari implementar el programa `client` que l'usuari executarà remotament i que es comunicarà amb el programa `jugador` utilitzant sockets. També haureu de modificar els programes `jugador` (jug8) y `servidor` (serv8).



La nova versió del `jugador` enviarà pel socket el resultat de les jugades per a que el programa `client` les mostri per la seva sortida estàndar en la màquina remota. A més a més, haureu de modificar la rutina `decidir_continuar` per a que demani pel socket al `client` si vol continuar el joc o si el vol finalitzar. D'altra banda, la nova versió del programa `servidor` crea un nou `jugador` per a cada petició de connexió que rebi d'un `client`, durant un temps fixat i fins a arribar a un número máxim de jugadors. Si passat aquest temps no s'han connectat tots els clients possibles, s'inicia el joc només amb els clients que s'han connectat i no es permet cap altra connexió,

Al directori de l'assignatura disposeu de `jug8 i serv8`, el vostre programa s'hauria de comportar de manera similar.

Per executar-los: s'ha de llançar el servidor en una màquina (poseu-hi els executables corresponents al servidor i al jugador) i el/s client/s en d'altres (ho podeu fer també des de diverses finestres dins la mateixa màquina).

Els paràmetres del servidor són el nombre màxim de jugadors que poden estar jugant la mateixa partida, el temps (en segons) que donarem per a que els jugadors s'hi puguin apuntar (connectar-se al servidor) i finalment el port en el qual s'està executant el servidor.

D'altra banda, el client rep com a paràmetres el nom de la màquina on s'executa el servidor o la seva adreça IP (per a proves dins la mateixa màquina podeu utilitizar la addreça de loopback 127.0.0.1 `localhost`) i el port del servei.

Un cop heu comprovat el funcionament passem a l'implementació:

Client

El client haurà de crear un socket i demanar la connexió amb el servidor de setimig. Un cop acceptada la connexió, es limitarà a anar mostrant per la sortida estàndard el que llegeix del socket, excepte si rep el caràcter '$'. En tal cas, mostrarà un missatge preguntant a l'usuari si vol continuar jugant o no. Si l'usuari escriu el caràcter 's' (sí) o 'n' (no), enviarà pel socket aquest mateix caràcter. En cas contrari, mostrarà un missatge d'error i li ho tornarà a preguntar.

En el directori de l'assignatura trobareu un fitxer anomenat `client.c` on està implementada tota la part de sockets. Afegiu el codi que falta per a que el programa es comporti com s'ha descrit en el paràgraf anterior i de manera similar a l'executable `client`

Jugador

En aquesta part haureu de modificar dos punts respecte al programa jugador que heu anat implementant al llarg del curs (a aquest nou programa l'anomenarem `jugador8.c`):

- La rutina **`int decidir_continuar()`**: Aquesta rutina reflexava el criteri que adoptàvem per a decidir si continuàvem jugant o no. Ara substituirem el seu codi per a que faci el següent:
  - Primer escriurem pel socket el caràcter `'$'`. Aquest és un caràcter de control que provocarà que el `client` li pregunti a l'usuari què vol fer.
  - Acte seguit, esperarem a llegir del socket un dels dos caràcters: `'s'` (sí) o `'n'` (no), retornant 1 o 0 respectivament.
- A l'inici del programa, després d'haver llegit l'identificador del jugador com feiem en els passos anteriors, llegirem de la pipe que ens comunica amb el nostre pare el nombre total de jugadors que participen en aquesta partida del joc i escriurem un missatge explicatiu pel socket indicant que comença la partida amb aquest nombre de jugadors (podeu utilitzar la rutina `escriure_msg()` per a fer-ho).
- Un altre qüestió que cal considerar és que degut al ús de la crida `exec`, el jugador no té accés a la variable que conté els canals del socket. Per solventar aquest problema teniu dues opcions:
  - Passar el canal al jugador a través del `exec`, el jugador recull el canal a través del `argv` del `main`
  - Redirecció dels canals 0 i 1 (entrada/sortida del jugador) cap al socket abans de la crida `exec`.

Servidor

El servidor l'anomenarem servidor8.c. Al directori de l'assignatura disposeu del codi serv8.c, que implementa tota la part d'inicialització de sockets. Afegiu aquest codi al vostre programa i adapteu-lo seguint les indicacions que se us comenten.

Consideracions:

- La reprogramació del signal `SIGCHLD` és necessària per a que es reculli l'estat dels processos zombies. Heu de tenir en compte que aquest signal es pot dipositar en qualsevol punt de l'execució i, per tant, el vostre codi ha tenir en compte la possibilitat de que aquest signal es dipositi durant l'execució d'una crida a sistema bloquejant.
- Com que no sabem el nombre de clients que es connectaran en l'interval de temps que ens passen com a segon paràmetre, cal que porteu el compte dels jugadors que anem creant i que en cap cas es superi el màxim nombre de jugadors permesos que ens passen com a primer paràmetre.
- Heu de modificar la rutina de tractament del `SIGALRM` per la que se us indica en el codi `serv8.c`