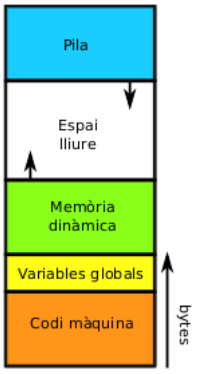
Processos : Sistemes Operatius I - Universitat de Barcelona

Lluis Garrido

1. El concepte de procés

Definició: Procés: Un procés és un programa en execució, una instància de un programa. En POO és equivalent a un objecte, que és una instància de una classe.



Estructura d'un procés a memòria

El Sistema Operatiu manté una llista dels processos en execució internament mitjançant el denominat **Control de Processos** (BCP, en anglès, Process Control Block).

Per a cada procés, el BCP guarda la següent informació bàsica:

1. En quina part de la memòria física resideix.
2. Quin és el fitxer executable associat.
3. Quin és l'usuari que l'executa.
4. Quins privilegis té.
5. Altres elements secundaris depenent dels sistema operatiu.

Els ordinadors actuals poden fer moltes coses alhora. En quant als processos.

Tot i que només hi hagi una única CPU, el sistema operatiu s'encarrega d'executar múltiples processos al mateix ordenador al mateix temps.

El sistema operatiu ho aconsegueix executant cada procés només per unes desenes o centenars de milisegons, donant la sensació a l'usuari de que s'està produïnt una execució múltiple (anomenat il·lusió de paral·lelisme). En acabar aquesta "llesca de temps"per al procés, el SO s'encarrega de canviar de procés.

De fet, com hem dit, en cada instant de temps només s'executa un únic procés, però en un segon es poden executar múltiples processos (inclús el mateix procés múltiples vegades).

En sistemes multiprocessador, el paral·lelisme és real, tot i que el sistema operatiu hi aplica la mateixa idea: en un segon múltiples processos s'executen en una determinada CPU.

Cal destacar que el sistema operatiu utilitza algorismes específics per decidir quin procés s'ha d'executar en cada moment a cada CPU. Aquesta és la base de la multiprogramació.

2. La interfície de programació

Una pregunta recurrent és quin tipus d'interfície ha de proveir un sistema operatiu als processos, entre altres coses,

- **Gestió de processos**: Pot un procés crear un altre procés? Pot un procés esperar que un altre procés acabi d'executar? Aturar o continuar l'execució d'un procés?
- **Entrada-sortida**: Com poden els processos comunicar-se amb dispositius connectats a l'ordinador? Poden els processos comunicar-se entre sí?

Respecte a aquestes dues preguntes, cal destacar que:

- La funcionalitat està implementada a nivell de nucli i els processos hi poden accedir mitjançant crides a sistema. En total hi ha una dotzena de crides associades.
- Pel que fa a sistemes de base UNIX, la interfície pràcticament no ha canviat respecte el seu disseny inicial del 1973 i continua essent molt utilitzada avui en dia. La interfície de UNIX és senzilla, potent i portable, no ha sigut necessari canviar la interfície del SO i,per tant, els desenvolupadors s'han pogut concentrar en desenvolupar aplicacions d'usuari.

És necessari que la interfície de programació sigui segura en front a qualsevol ús maliciós de la mateixa.

Antigament, en els sistemes de processament per lots, el nucli s'encarregava de crear els processos. Realment no hi havia problemes de "seguretat.en crear processos ja que era el SO mateix qui ho controlava tot.

Tanmateix a l'actualitat els mateixos processos poden crear i gestionar altres processos. Açò ha permès un munt d'innovació, essent una de les primeres

aplicacions l'intèrpret de comandes. Altres aplicacions que creen i gestionen processos són els navegadors web, els gestors gràfics de finestres, els servidors web, les BBDD, els compiladors, els editors de text... Per tant cal gestionar la seguretat d'alguna manera.

De fet, permetre que un procés pugui crear altres processos té beneficis importants:

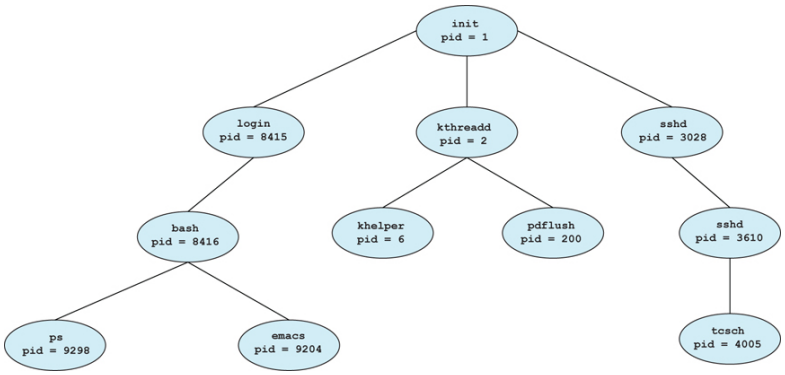
- En comptes de crear un únic programa que fa de tot, es creen múltiples programes petits i especializats per a cada tasca. L'usuari els pot conseguir per dur a terme els seus objectius.
- A la línea de comandes existeixen un munt d'aplicacions especialitzades. Es poden combinar de forma molt senzilla per aconseguir funcionalitats molt potents.
- Un navegador web acostuma a fer servir aplicacions externes per tal de dibuixar una pàgina a pantalla. Això permet aconseguir funcionalitats molt complexes.
- El mateix passa amb el servidor web que és qui .entrega"la pàgina al navegador: abans d'entregar-la al navegador es poden invocar a processos externs per tal de formatar-la adequadament perquè pugui ser visualitzada al navegador.

2.1. La gestió de processos

Definició: El procés que crea el procés s'anomena pare mentre que el procés que es crea s'anomena fill. Les tasques que ha de realitzar el nucli en crear un nou procés són:

1. Crear i inicialitzar el Bloc de Control de Processos.
2. Crear i inicialitzar l'espai d'adreces a memòria.
3. Avisar al nucli que hi ha un nou procés a la llista de processos a executar.
4. Preparar el nucli per començar a executar al "main".

Com els processos poden crear altres processos es crea una **jerarquia de processos.** El sistema operatiu emmagatzema informació sobre aquesta jerarquia:



A un sistema UNIX podem veure la jerarquia de processos mitjançant aquesta instrucció a la línea de comandes:

$ pstree

Un procés pare, en crear un procés fill, pot controlar el **context del fill**:

- Els privilegis
- Quin és el temps màxim d'execució
- Quina és la memòria màxima que pot ocupar
- Quina prioritat té per executar-se sobre la CPU respecte altres processos.
- On s'envia tot el que s'imprimeix amb "printf"(i.e. la sortida)
- D'on rep el que es captura amb "scanf"(i.e. l'entrada)
- etc.

2.1.1. La gestió de processos a Windows

A Windows la funció que permet crear un procés s'anomena CreateProcess:

```
// Start the child process
if (!CreateProcess(NULL,
// No module name (use command line)
argv[1],
// Command line
NULL,
// Process handle not inheritable
NULL,
// Thread handle not inheritable
FALSE,
// Set handle inheritance to FALSE
0,
// No creation flags
NULL,
// Use parent's environment block
NULL,
// Use parent's starting directory
&si,
// Pointer to STARTUPINFO structure
&pi )
// Pointer to PROCESS_INFORMATION structure
)
```

Els primers dos arguments permeten especificar el programa a executar i els arguments a passar al main. Tota la resta estan associats a restringir l'entorn en què s'executa el procés fill.

2.1.2. La gestió de processos a UNIX

Al sistemes UNIX s'utilitza un altre enfoc. En comptes d'utilitzar una única funció, s'utilitzen dues funciones, fork i exec

La funció fork crea un nou procés, una còpia completa del procès pare. La funció fork retorna dues vegades, una pel pare i una altra pel fill. Pel pare, retorna el número que identifica al fill; pel fill retorna 0.

```
int main(void)
{
int ret;
ret = fork();
if (ret == 0) { // fill
printf( " Soc el fill i el meu id es %d\n " , getpid());
return 0;
} else { // pare
printf( " Soc el pare del proces %d\n " , ret);
return 0;
}
}
```

El procés fill creat amb fork és independent del procés pare. Al codi de l'assignatura es dóna el codi fork-variables.c, executeu-lo.

Un cop creat un procés fill amb fork, podem executar un nou programa amb exec.

- La (família de) funcions exec reemplaçen la imatge del procés amb l'especificada a exec. No es crea cap procés nou!
- La funció exec no retorna en acabar l'execució del programa especificat a exec.

La pregunta és, per què dues funcions en comptes d'una per crear processos?

1. En fer fork el procés fill hereta el context del pare (privilegis, fitxers oberts, ...).
2. Després del fork i abans de fer l'exec, es pot establir el nou context del programa a executar amb crides a sistema.
3. Fork no té arguments, exec només té dos arguments. La funció CreateProcess de Windows té 10 arguments!

```
int main(void)
{
int ret;
char *argv[3] = {"/usr/bin/ls", "-l", NULL};
ret = fork();
if (ret == 0) { // fill
printf("Soc el fill i el meu id es %d\n", getpid());
// Aqui s'estableix el nou context del proces fill!
// Es fan les crides necessaries abans de fer exec
execv(argv[0], argv);
} else { // pare
printf("Soc el pare del proces %d\n", ret);
return 0;
}
}
```

La funció setrlimit permet establir múltiples límits: temps de CPU, ús de memòria dinàmica, ús de la pila, nombre de fitxers que es poden obrir, etc. Veure el codi fork-exec-setrlimit.c

A més d'establir límits de privilegis sobre el procés fill, també es pot controlar.

Alguns cops és interessant que el pare esperi que el procés fill finalitzi la seva execució per continuar la seva execució pròpia. (*Per exemple, és possible que el pari necessiti el resultat del fill per poder continuar la execució.*). La funció wait permet fer-ho.

```
int main(void)
int main(void)
{
int ret;
char *argv[3] = {"/usr/bin/ls", "-l", NULL};
ret = fork();
if (ret == 0) { // fill
printf("Soc el fill i el meu id es %d\n", getpid());
execv(argv[0], argv);
} else { // pare
printf("Soc el pare del proces %d\n", ret);
wait(NULL);
printf("Torno a ser el pare un cop el fill ha acabat\n");
return 0;
}
}
```

Tanmateix, en alguns contextos no és interessant que el pare esperi al fill. El fet de no posar wait permet que pare i fill s'executin "paral·lelament". Pel cas de la línea de comandes això es pot especificar mitjançant un & al final de la línea, per exemple: instruccio1 & instruccio2

Usualment el fork i l'exec apareixen en parella, encara què també pot ser interessant un fork sense exec. Per exemple a Google Chrome cada pestanya és un procés creat amb un fork (*En sistemes UNIX, en sistemes Windows fork no existeix i per això s'utilitza una cua de processos que "esperen" que el procés principal els doni enllaços per poder processar*).

- Cada procés s'executa en paral·lel amb la resta de processos. Això fa que les pestanyes es "carreguin" en paral·lel.
- Si a una de les pàgines hi ha un problema només fallarà aquella pestanya però no tot el navegador ja que els processos són independents entre sí.

A altres navegadors con Firefox s'utilitza un altre mecanisme per obtenir el paral·lelisme: **els fils**. Els veurem més endavant i tenen els seus avantatges i desavantatges enfront el fork.

2.2. L'entrada-sortida

Un ordinador té una gran diversitat de dispositius d'entrada i sortida: teclat, ratolí, disc, port USB, ethernet, wifi, pantalla, micròfon, càmera, etc.

- Antigament hi havia una interfície de programació específica per a cada dispositiu. Però cada cop que sinventava un nou dispositiu, calia actualitzar la interfície per ser capaç de gestionar-lo.

- Una gran innovació als sistemes UNIX va ser la unificació amb una única interfície: utilitza la mateixa interfície per escriure i llegir de disc que per enviar i llegir dades de la xarxa. Aquesta idea va tenir tant d'exit que actualment és universal a la resta de sistemes operatius.

Les característiques de la interfície entrada-sortida de UNIX són:

- **Uniformitat**: tots els dispositius d'IO utilitzen el mateix conjunt de crides a sistema, open, close, read i write
- **Obrir abans d'utilitzar**: abans d'utilitzar el dispositiu cal obrir-lo amb la crida a sistema **open** (inclús la impressora, per exemple). El sistema operatiu pot comprovar aleshores els permisos d'accés i mantenir un diari sobre els dispositius oberts.

  En obrir un dispositiu el sistema operatiu retorna al procés un sencer (integer) anomenat **descriptor de fitxer** (file descriptor), associat al procés, que més endavant es pot utilitzar per llegir o escriure del dispositiu.
- **Orientat a bytes**: s'accedeix a tots els dispositius amb vectors de bytes, siguin fitxers de disc o un dispositiu de xarxa.
- **Lectures amb buffer al nucli**: totes les dades que el nucli llegeix es emmagatzemen a un buffer intern del nucli, vinguin de xarxa o de disc. Això permet que es faci servir una única crida a sistema, **read**, per llegir les dades i que el procés pugui llegir les dades quan les demana.
- **Escriptures amb buffer al nucli**: de forma similar, totes les dades a escriure al dispositiu s'emmagatzemen primer a un buffer intern del nucli. El procés només ha de fer servir la crida a sistema, **write**, per escriure dades. El nucli copia les dades a escriure al buffer intern i s'encarrega d'enviar les dades al dispositiu al ritme necessari.
- **Tancament explícit**: quan un procés ha finalitzat de fer servir un dispositiu ha d'utilitzar la crida a sistema **close**. Això allibera al nucli tots els recursos necessaris.

Veiem ara tècniques bàsiques de **comunicació interprocés** a un mateix ordinador: **la redirecció** i **la canonada**:

Recordem primer la funció printf: és una crida a una llibreria de l'espai d'usuari

```
printf(."El valor és %d\n", i)
```

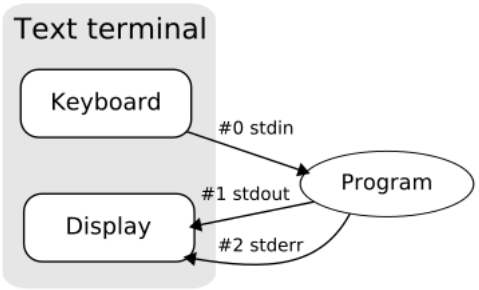
Aquesta instrucció genera a l'espai de usuari, la cadena a imprimir. Un cop generada es fa la crida al sistema operatiu amb un write

```
count = write(fd, vector, nbytes)
```

La funció write, disponible a una llibreria d'usuari, fa la crida a sistema. El primer paràmetre és el descriptor de fitxer (un sencer) i els altres dos fan referència al vector de bytes a escriure. Quin descriptor de fitxer s'utilitza en imprimir per pantalla?

Tot procés, en crear-se pel sistema operatiu (sigui Windows o UNIX) té creats 3 fitxers per defecte:

- Descriptor **0**: conegut com "**entrada estàndard**", està associat per defecte el teclat (es captura amb scanf).
- Descriptor **1**: conegut com a "**sortida estàndard**", està associat per defecte a la pantalla del terminal (s'imprimeix amb printf).
- Descriptor **2**: conegut com a "**sortida d'error**", està associat per defecte al terminal.



Les instruccions:

```
i = 25;
printf("El valor es %d\n", i);
```

Es converteix en la següent crida a sistema:

```
char *vector = "El valor es 25\n";
int nbytes = 15;
count = write(1, vector, nbytes);
```

2.2.1. Redirecció sortida estàndard a fitxer

Per defecte el descriptor 1 (printf) està associat al dispositiu "pantalla". Ara volem associar el descriptor 1 a un fitxer de disc.

Per això cal fer els següents passos:

1. Fer un fork. Després de fer el fork la situació del procés fill és la que es mostra a la figura.
2. S'obre el fitxer on volem que es bolqui tot allò que s'escriu al descriptor 1 (printf)
3. Executem el programa. Tot el que aquest escriu al descriptor 1 (imprimeix amb printf) s'escriu a disc!

Aquest és el codi C que ho permet fer:

```
int ret;
char *argv[3] = {"/usr/bin/ls", "-l", NULL};
ret = fork();
if (ret == 0) { // fill
// obrim fitxer
int fd = open("fitxer.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
dup2(fd, 1); // associem fd al descriptor 1
close(fd);
// tanquem fd
execv(argv[0], argv); // executem programa
} else { // pare
...
}
```

Tot el que s'imprimeixi al descriptor 1 estarà associat a fd, el fitxer de disc, en comptes de la pantalla. Típicament es diu que *"es redirigeix la sortida estàndard de pantalla a un fitxer"*.

La línea de comandes permet redirigir la sortida estàndard a un fitxer:
$ ls -l > fitxer.txt

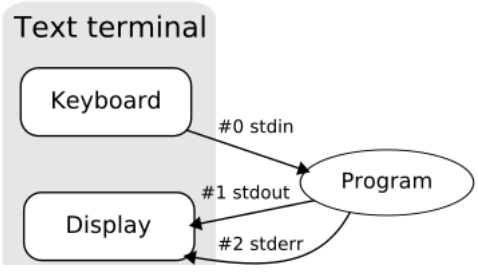
La línea de comandes interpreta la línea anterior i executa el codi anterior.

Una cosa similar passa amb
$ var=$(ls)

L'interpret de comandes redirigeix la sortida estàndard de la comanda ls (p.e a un fitxer temporal) per després llegir-ho i assignar-ho a la variable var.

De la amteixa forma que hem redirigit la sortida estàndard a un fitxer, podem redirigir un fitxer a l'entrada estàndard. Volem associar el descriptor 0 (scanf) a un fitxer de disc. Els passos són els següents:

1. Fer un fork. Després de fer el fork la situació del procés fill és la que es mostra a la figura.
2. S'obre el fitxer d'on volem que el descriptor 0 (scanf) agafi les dades.
3. Executem el programa. La funció scanf agafarà les dades de disc!



Aquest és el codi que ho permet fer:

```
int ret;
char *argv[2] = {"./scanf",
NULL};
ret = fork();
if (ret == 0) { // fill
// obrim fitxer
int fd = open("linies.txt", O_RDONLY);
dup2(fd, 0); // associem fd al descriptor 0
close(fd);
// tanquem fd
execv(argv[0], argv); // executem programa
} else { // pare
...
}
```

La línea de comandes permet redirigir un fitxer a l'entrada estàndard a un fitxer.

$ ./scanf < linies.txt

Inclús podem redirigir l'entrada i sortida estàndard de l'aplicació scanf.

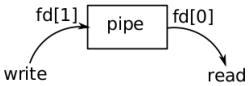
$ ./scanf < linies.txt > output.txt

2.2.2. Canonades

Definició: Canonada: La canonada és una forma bàsica de comunicació interprocés: l'objectiu és dirigir les dades que un procés A genera cap a un altre procés B.

Com es fa això? Mitjançant una crida a sistema, **pipe**, que crea:

1. Un buffer, al sistema operatiu, que gestiona la comunicació.
2. Dos descriptors de fixers, un d'escriptura i un altre de lectura.



Codi bàsic d'una canonada:

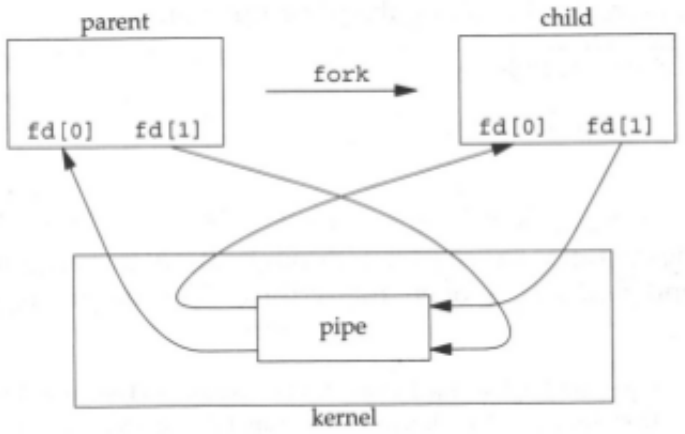
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
int fd[2];
if (pipe(fd) == -1) {
fprintf(stderr, "pipe not created");
exit(1);
}
// fd[1] per escriptura
// fd[0] per lectura
return 0;
}
```

Podem enviar qualsevol tipus de dada per una canonada, per exemple:

```
int main(void)
{
int fd[2];
char buf[30];
if (pipe(fd) == -1) {
fprintf(stderr, "pipe not created");
exit(1);
}
printf("writing to file descriptor #%d\n", fd[1]);
write(fd[1], "test", 4);
printf("reading from file descriptor #%d\n", fd[0]);
read(fd[0], buf, 4);
buf[4] = 0;
printf("read \"%s\"\n", buf);
return 0;
}
```

Les canonades s'utilitzen per comunicar diferents processos. Els processos han de tenir una relació pare-fill.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{
int fd[2];
pipe(fd);
if (fork() == 0) { // child
...
} else { // parent
...
}
return 0;
}
```



Per exemple: si el pare escriu a fd[1], el fill ho pot llegir de fd[0].

Alguns detalls sobre el funcionament de les canonades:

- El buffer és un **buffer intern** del SO. Totes les dades passen per aquest buffer.

- El buffer és relativament **petit** (al voltant de 1 a 4 KBytes)
- Si en **llegir** del buffer no hi ha cap dada, el procés que fa la crida es **bloqueja** fins que hi ha dades disponibles.
- Si en **escriure** al buffer aquest és **ple**, el procés que fa la crida es **bloqueja** fins que un altre procés el comença a buidar.
- Típicament les canonades s'utilitzen a la línea de comandes per combinar múltiples comandes "senzillesï aconseguir una funionalitat més complexa:

  ```
  $ ls | ./scanf
  ```

  La comanda anterior envia el que imprimeix ls per la sortida estàndard (descriptor 1) a l'entrada estàndard (descriptor 0) de l'aplicació scanf.
- Podem combinar les canonades i la redirecció com vulguem.

  ```
  $ ls | ./scanf > out.txt
  ```