Sistemes Operatius I

Pau Soler Valadés

Primavera 2019

Índex

1 Sistemes Operatius: Introducció

Definició 1.1. Un **Sistema Operatiu** (SO) és una capa de programari que gestiona els recursos d'un ordinador per als usuaris i les aplicacions que s'hi executen.

Un sistema operatiu el podem veure com una màquina estesa:

- **Oculta a l'usuari** tots els detalls escabrosos que han de ser realitzats per accedir als dispositius.
- Ofereix a l'usuari una **màquina virtual** i una **interfície** molt més senzilla d'utilitzar que el sistema de comandes.

1.1 Funcions d'un SO

Les funcions d'un SO són les següents, que fan que el poguem descriure també com una màquina virtual.

- **Àrbitre:** El SO gestiona els recursos d'una màquina real (Ordinador) per aïllar les aplicacions entre elles per a evitar fallades o atacs maliciosos a la vegada que permet la comunicació entre aplicacions.
- **Il·lusionista:** Fa creure a totes les aplicacions que tenen memòria il·limitada.
- **Interfície Comuna:** Proveeix una interfície comuna a totes les aplicacions. Mitjançant la interfície, les aplicacions poden utilitzar el maquinari i permet que les aquestes es disenyin tenint en compte el SO, que és uniforme, i no el maquinari, que és diferent en cada dispositiu.

1.2 Criteris d'Avaluació d'un SO

Fiabilitat: El sistema fa exactament allò pel qual ha estat dissenyat.

Seguretat: Si s'executés en un entorn hostil on altre programari vol pendre control del sistema, s'ha de saber protegir.

Privacitat: Certes dades son accesibles només pels usuaris autoritzats.

Portabilitat al SO: El SO proveeix a les aplicacions una interfície virtual. Les crides a sistema en són una part imporant ja que les apliacions les usen per accedir a una part del sistema operatiu.

Portabilitat a les aplicacions: El SO s'implementa independentment del maquinàri usant el Hardware Abstraction Layer (HAL). Gràcies a això es pot millorar el SO de manera senzilla a mesura que evoluciona el maquinàri.

Rendiment: es mesura de múltiples maneres:

- Temps necessàri per completar una operació i quantes operacions per unitat de temps.
- Operacions necessaries per a accedir al maquinàri.
- Repartició dels recursos entre les diverses aplicacions/usuaris d'un ordinador.
- Variació del rendiment en funció del temps.

2 El Nucli

Definició 2.1. El **Nucli** o **Kernel** és una peça de programari de total confiança que té accés total a les capaçitats del maquinari per realitzar les funcions que ha de fer un SO [*1.1 Funcions d'un SO*]. És el cor del SO.

El Kernel del SO té la tasca de **protegir** el SO, les dades que conté i els seus usuaris. Per a determinar-ho ens basarem en els criteris Fiabilitat, Seguretat, Privacitat i Eficiencia d'un SO i els adaptarem a les funcions del Kernel. [*1.2 Criteris d'avaluació d'un Sistema Operatiu*]

- Fiabilitat: El SO ha de funcionar correctament independentment del que faci la aplicació d'usuari. Per exemple: Un error de programació o atac maliciós en una aplicació determinada no ha d'afectar a la resta d'aplicacions.
- Seguretat: Cal impedir que una aplicació pugui escriure a disc i modificar el codi del sistema operatiu. Això s'aconsegueix limitant les accions que les aplicacions individuals poden realitzar.
- Privacitat: Cada usuari només pot accedir a les dades a les que està autoritzat.
- Eficiència: El SO ha de repartir els recursos entre les aplicacions. És a dir, cap aplicació ha de poder consumir tots els recursos de memòria.

El Kernel s'executa directament al processador amb drets il·limitats, només tenint limitacions a operacions possiblement perilloses controlades a nivell de hardware.

En canvi, totes les aplicacions que no siguin de confiança al SO s'executaran en un espai restringit, no tenint accés a les capacitats reals de la màquina. A aquestes aplicacions se les anomena processos.

Llavors, com protegeix el Kernel al SO? Per poder contestar aquesta pregunta hem de coneixer el concepte de procés (què és un procés i en què es diferència d'un programa), el mode dual d'operació (com implementa un SO un procés i el maquinari que fa falta per executar de forma eficient un procés en mode restringit) i la transferència del control de seguretat (com travessar la frontera entre els processos, no fiables, i el nucli que si ho és.

2.1 El Concepte de Procés

Per poder definir bé què és un procés anem a entendre quina és la estructura d'un procés a memòria.

1. Fem un programa amb un llenguatge d'alt nivell (C per exemple).
2. Fem servir un compliador per convertir el codi de C a llenguatge màquina. El sistema sap com traduïr aquest a llenguatge màquina. Ara el SO el pot executar.
3. El SO copia les instruccions de llenguatge màquina del disc a la memòria física. A partir d'allà crea la memòria dinàmica i col·loca la pila d'execució a on hi hagi espai lliure.
4. Quan cridem al "main" (al executar el codi) ja tenim el **procés** que voliem executant-se!

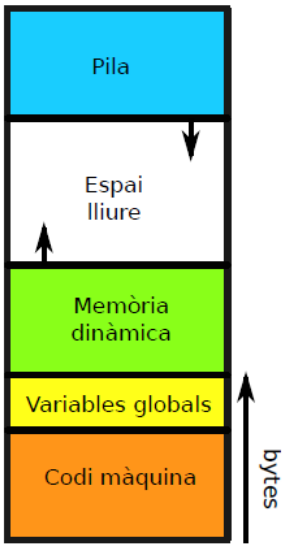

Figura 1: *Com funciona un procés*

Ara amb aquest exemple fet, anem a definir què és un procés:

Definició 2.2. Un **Procés** és una instància d'un programa que s'executa amb drets restringits.

Observacions. Distingim:

- Es donen permisos (es controla l'accés) per a crear la pila i la zona de memòria dinàmica.
- Un procés es pot executar múltipes instàncies d'un mateix programa.

- El compilador és un **programa** que crea el **procés** corresponent.
- Usarem el concepte de Procés en comptes del terme aplicació:

$$\text{Gestiono aquest Procés} \equiv \text{Gestiono aquesta aplicació}$$

Ara suposem que el procés de l'apartat anterior comença a executar-se. Com ens podem assegurar que les instruccions que executa un procés no perjudiquen a altres procesos o al nucli? Hem d'assegurar **protecció** entre processos.

2.2 Mode Dual d'Operació

Definició 2.3. El **Mode Dual d'Operació** és el fet de que el màquinari pot executar dos modes:

1. Mode Nucli: El processador pot executar qualsevol instrucció.
2. Mode Usuari: El procesador comprova cada instrucció abans d'executar-la. Si es detecta alguna infracció, es notifica al nucli de la incidència.

El nucli sempre s'executa en el primer, mentres que els processos sempre s'executen amb el segon.

Perquè el nucli pugui protegir els processos entre sí he de controlar com a mínim (a) les *Intruccions Privilegiades* (es prohibeixen les instruccions perilloses en mode usuari) (b) la *Protecció de Memòria* (prohibir els accesos fora de l'espai de memòria assignat a un procés) i (c) les *Instruccions de Temporitzador* (independentment del que faci un procés, el nucli ha de pendre el control per supervisar el que està passant).

2.2.1 Instruccions Privilegiades

Els processos d'usuari només poden utilitzar un subconjunt de totes les instruccions disponibles (mode d'usuari d'execució) mentre que el nucli té tota la potència de la màquina (mode nucli d'execució).
Llavors, com determinem el subconjunt al que el mode d'usuari d'execució pot accedir? Les instruccions no poden:

- Modificar el seu nivell de privilegi d'execució.
- Modificar les interrupcions del maquinari.
- Canviar la regió de memòria a la que pot accedir.

- Obtenir privilegis de manera directa només mitjançant una crida a sistema (transfereix el control al sistema operatiu per fer la operació demanada).

Quan un procés executa una instrucció no autoritzada es produeix una excepció del processador. Aquesta transfereix el control a una funció del nucli. Normalment quan això succeeix es "mata"el procés (aturar la excepció).

2.2.2 Protecció de Memòria

Al executar un procés d'usuari, el procés ha d'accedir a memòria física per poguer ser executat, però això vol dir que el nucli també ha d'estar a memòria per gestionar les crides a sistema del procés, interrupcions de temporitzador i les possibles excepcions.
Hi poden haver múltiples processos executant i per cada un s'ha de configurar el maquinari perquè cada procés només pugui accedir a la seva parcel·la de memòria. Tal com en les Instruccions Privilegiades, si un procés intenta accedir a una posició de memòria no permesa, es produeix una excepció.

Podem evitar amb el maquinari del SO (amb el nucli concretament) que un procés accedeixi a parts no permeses a memòria física. Hi ha dues maneres de solucionar el problema: mitjançant un esquema amb dos registres maquinari anomenat Base i Límit i la Memòria Virtual.

2.2.2.1 Base i Límit:

Tenim dos registres: Base i Limit. Aquest només poden ser modificats per funcions privilegiades. **Cada cop que s'accedeix a memòria, se suma la base a la adreça.** Llavors es compara el valor amb el límit. Si és menor, deixarem accedir a memòria, si no ho és, es produirà una excepció.

Observacions. Notem que:

- Cada accés a memòria comporta una suma i una comparació, però val la pena el cost addicional, ja que ens garantitza protecció.
- El SO opera directament sobre memòria real.

Aquest esquema té importants defectes com:

- Previsió de memòria màxima: es necessita calcular la memòria màxima que ocuparà el procés a memòria, previsió que no és fàcil de fer.

- Incapacitat de compratir memòria: No podem compartir intruccions entre processos, ni tan sols les instruccions del codi màquina.
- No Fragmentació de Memòria: tot el procés ocupa un espai continuu infragmentable.

2.2.2.2 Memòria Virtual

El processador genera adreces virtuals que es tradueixen pel maquinari específic a una adreça física.

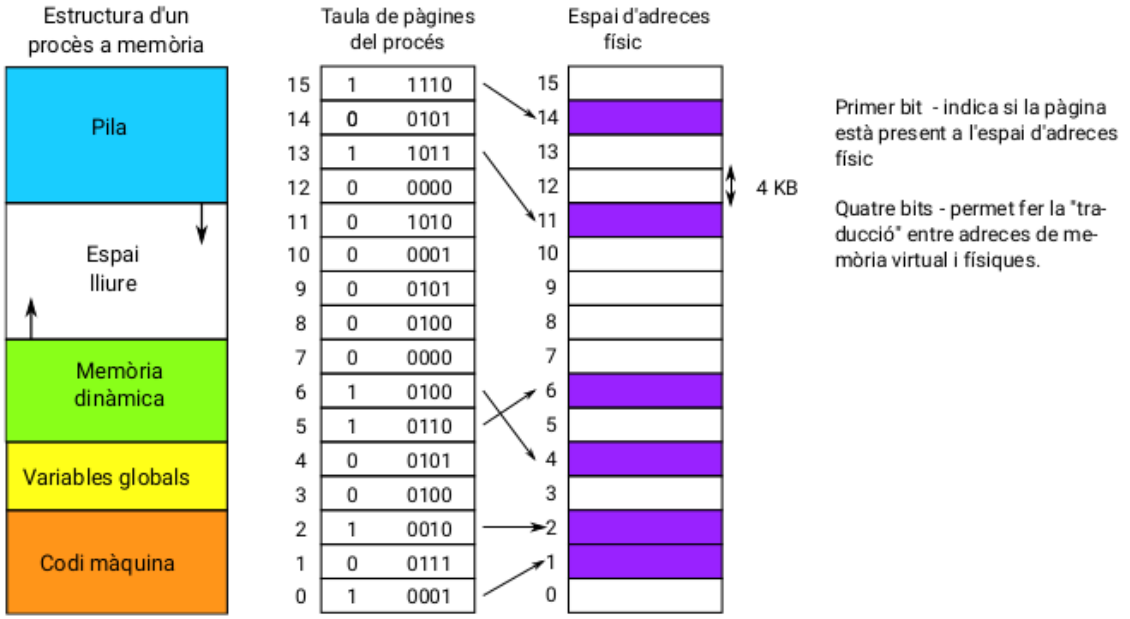

Figura 2: *Amb la memòria virtual podem aconseguir espai aparentment il·limitat i lineal, no com ens queda a la memòria física.*

Les característiques principals de la memòria virtual són:

- El mapat d'una adreça virtual a una física es fa a nivell de maquinari.
- El nucli s'encarrega de gestionar les taules dels processos.
- Cada procés té disponible tot l'espai de memòria possible tot i que no hi sigui físicament.
- Posicions contigues a l'espai virtual no tenen perquè ser contigues a l'espai físic.

2.2.3 Interrupcions de Temporizador

Quan el SO comença a executar un procés, aquest és lliure d'executar qualsevol instrucció no privilegiada, cridar a una funció, fer un bucle...

Com que cada procés creu que té tota la CPU disponible per ell sol, el SO ha d'indicar periòdicament quin procés s'està executant a la CPU per assegurar que tots els processos s'executen (els va alternant i que tots tinguin una estona d'execució) i algun procés pot encallar-se en un bucle infinit. Per això necessitem que el SO prengui el control de la CPU periòdicament.

S'utilitza un temporitzador controlat pel hardware que interromp la execució del procés cada determinat temps. Quan aquest comptador arriba a 0, es transfereix el control al SO i es passa a **mode Nucli**. Llavors el SO decideix quin procés s'executa a la CPU.
Es poden produir altres interrupcions en la execució d'un procés que no tinguin a veure amb el comptador, com per exemple, un disc que ha llegit dades o quan l'usuari mou el ratolí.

2.3 Transferència del Control de Seguretat

En aquest apartat parlarem de com el SO passa entre mode nucli (fiable) i el mode usuari (no fiable), de quan es fa i de les diferències d'una a l'altre.

2.3.1 Mode Usuari a Mode Nucli

Hi ha tres raons per les quals es pot realitzar la transferènca de mode usuari a mode nucli: les excepcions, les interrupcions i les crides a sistema.
Cal mencionar que les dues primeres són síncrones mentre que la última és asíncrona.

2.3.1.1 Excepcions

Definició 2.4. Una **Excepció** és una condició inesperada de l'execució del procés.

Per exemple, si un procés intenta executar una instrucció privilegiada, accedir a una posició de memòria fora de l'espai permès, dividir per 0 o es topa amb un breakpoint d'un debugger farà que salti una excepció.

Les excepcions són molt útils en

- **Virtualització:** Emular maquinari que no existeix per fer alguna cosa que no pot fer. Per exemple:
  - Hi ha processadors que no suporten operacions en coma flotant. Si un procés demanda una operació d'aquest tipus, es produeix una excepció, el SO emula la operació i es continua la execució de manera normal.

- Quan una màquina virtual s'executa en mode usuari, aquesta intentarà accedir al maquinari com si tingués privilgis totals. Llavors salta una excepció que capturarà el SO amfitrió i realitzarà la operació demanada.

- **Gestió de la Memòria Virtual:** Quan un procés accedeix a una posició de memòria no permesa o a una posició vàlida però la informació no és a la RAM, salta una excepció.

2.3.1.2 Interrupció

Definició 2.5. Una **Interrupció** és un senyal asíncron de maquinari produït per un esdeveniment (és a dir, no produït per la execució del procés).

Es comporta com una excepció però: el procesador atura l'execució del procés actual i es passa a executar una determinada funcionalitat del SO depenent del cas.

Cada tipus d'interrupció té una funció gestor associada: Interrupcions de temporitzador, interrupcions sobre operacions d'entrada-sortida...

2.3.1.3 Crides a Sistema

Definició 2.6. Una **crida a sistema** és un conjunt de funcions que ofereix el sistema operatiu als processos per accedir als dispositius o realitzar qualsevol altre operació que requereixi privilegis. Mirar

Son les comandes, sent aquestes les interfícies que ofereix el SO als processos per accedir als dispositius o realitzar una altra operació a la que només hi té accés el nucli.

Aquestes s'impementen gràcies a una instrucció màquina específica (trap instruction) que fa que la CPU passi de mode usuari a mode nucli i es cridi una funció determinada del SO. Llavors, el SO realitza l'operació demanada; quan acaba, torna al mode usuari continuant la execució després del trap instrucion.

Linux ofereix més de 250 crides a sistema que poden fer crides a sistema, controlar processos i comunicar-los fins a la informació de manteniment.

2.3.2 Mode Nucli a Mode Usuari

Quan volem anar del mode nucli al mode usuari serà per una de les següents causes:

1. **Reestablir l'execució després d'una excepció, interrupció o crida a sistema**: quan el Nucli acaba la operació restabliex la operació passant a mode usuari.

2. **Canviar l'execució a un altre procés:** ha de recuperar la informació del procés antic i restablir l'execució en el punt en què es va aturar.

3. **Executar un nou procés:** carrega en memòria les instruccions, executa la pila, estableix la primera instrucció a executar i passa a mode usuari.

4. **Upcall (crida del nucli als processos):** mecanisme perquè els processos puguin rebre notificacions asíncrones d'esdeveniments.

2.3.3 Implementació Tècnica

Les transferències del mode nucli al mode usuari i viceversa són fruit de la col·laboració entre el maquinari i el programari.

2.3.3.1 Vector d'Interrupcions

Quan es produeix una interrupció, excepció o crida a sistema, es passa a mode Nucli, el maquinari determina si és una de les tres nombrades i per acabar les identifica amb un número sencer.

Aquest nombre sencer està associat a una entrada en un **vector d'interrupcions**. Aquesta entrada apunta a la funció que gestionarà l'esdeveniment (interrupció, excepció o crida a sistema).

2.3.3.2 Crides a Sistema

Definició 2.7. Una **Crida a Sistema** és una crida per la que es transfereix el control al SO. A nivell de programació d'usuari és igual que una crida a qualsevol altre funció.

Els passos per fer una crida a sistema són:

1. L'usuari fa una crida:

2. S'executa la "trap instruction". Es passa a mode nucli i s'executa una determinada funció del SO.

3. Dins de cada SO cada crida a sistemes s'implementa en funcions diferents.

4. S'executa la crida a sistema i se segueix el camí invers per continuar l'execució en el punt en què s'ha realitzat la crida. Es torna a passar a mode usuari.

3 Els Processos

En aquest capítol explorarem les crides a sistema que ens ofereix el SO per crear processos i per gestionar-ne la seva seguretat, i les eines bàsiques de comunicació entre processos.

3.1 El concepte de Procés

En aquest apartat continuarem des de l'apartat del Nucli [*2.1 El concepte de Procés*], ja que necessitavem la idea intuitiva i ara hem de profunditzar-hi.

Definició 3.1. Un **Procés** és un programa en execució, una instància d'un programa.

El SO manté mitjançant el **Bloc de Control de Processos** o BCP (Process Control Block), una llista dels processos que s'executen. Per cada procés s'emmagatzema:

- L'identificador de procés.
- Quin usuari l'executa
- Quins privilegis té
- Quines restriccions té
- En quina part de la memòria fisica resideix (memòria virtual)

Tot i que només hi hagi una CPU, els ordinadors d'avui en dia poden fer moltes coses a la vegada. El SO s'encarrega d'executar múltiples processos al mateix ordinador al mateix temps. Ho fa executant cada procés durant unes desenes o centenes de milisegons. Quan acaba aquesta llesca de temps el SO s'encarrega de canviar de procés.
A cada instant de temps només s'executa un únic procés, però en un segon es poden executar múltiples processos, donant la il·lusió a l'usuari de paral·lelisme. Aquest paral·lelisme és real en sistemes multiprocessador, encara que s'usi la mateixa idea: en un segon múltiples processos poden executar en una determinada CPU. El propi SO decideix quin procés ha d'executar a cada moment a cada CPU mitjançant algorismes refinats i concrets.

3.2 La Interfície de Programació

Definició 3.2. Diem **Interfície de programació** al codi que es dedica a comunicar processos entre ells o amb el nucli. Molt relacionat amb el concepte de *crides a sistema*

El SO ha de proveir a un procés diverses opcions a realitzar que es poden englobar en les categories següents:

- Gestió de processos: Pot un procés crear un altre procés? Pot esperar a que un altre acabi d'executar? Pot aturar o continuar l'execució d'un altre?
- Entrada-Sortida: Com poden els processos comunicar-se amb els dispositius connectats a l'ordinador? Poden comunicar-se entre si dos processos?
- Altres: La gestió de memòria, la xarxa i altres més variats.

Tan la gestió de processos com l'entrada-sortida estan implementades a nivell de nucli i els processos hi poden accedir mitjançant crides a sistema, havent-hi una dotzena. Aquest sistema no ha cambiat pràcticament des de 1973, sent encara molt utilitzada avui en dia.

La interfície de programació ha de ser segura contra qualsevol ús maliciós. Per a fer-ho, els pròpis processos poden crear i gestionar altres processos. Aquest fet és una gran innovació, generant per exemple, l'intèrpret de comandes. Un altre exemple molt clar són els navegadors web, que acostumen a fer servir aplicacions externes per dibuixar una pàgina a pantalla.

3.2.1 La Gestió de Processos

El procés que crea el procés s'anomena **pare**, mentre que el procés creat s'anomena **fill**. Les tasques que ha de fer el nucli al crear un nou procés són:

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

Com que un procés pot crear altres processos és necessàri crear una jerarquia de processos. A linux la comanda "pstreel·la crea.

Als sistemes UNIX quan es crea un procés s'executen dues funcions, el fork i exec.

Fork crea un nou procés, una còpia del procés pare. Aquesta funció retorna dues vegades: una pel pare, i retorna el número que identifica al fill; i una altre pel fill, que retorna 0.
El procés fill creat amb fork és independent del procés pare.

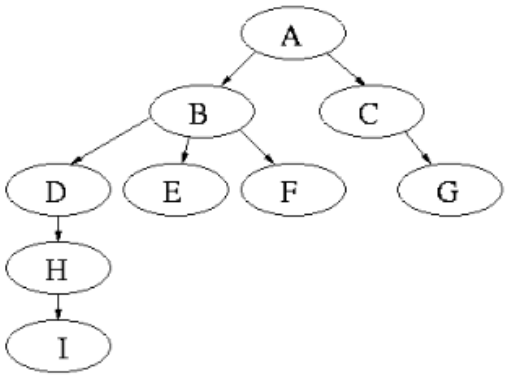

Figura 3: *Graf que representa una possible jerarquia de processos.*

Ja havent creat un procés fill amb fork, podem executar un programa amb exec. La família de funcions exec reemplaçen la imatge del procés nou amb la expecificada a exec. No es crea cap procés nou! i la funció exec no retorna en acabar l'execució del programa especificat.

Llavors, per què utilitzem dues funcions en comptes d'una?

1. **Herència del context:** Al fer el fork el procés fill hereta el context del pare (privilegis, fitxers obrets...)

2. **Canvi del context:** Entre el fork i l'exec, es pot establir el nou context del programa a executar amb crides a sistema.

3. **Nombre d'arguments:** Fork no té cap argument i exec només dos. En canvi la comanda CreateProcess (el seu equivalent en Windows) té 10 arguments!

Com hem dit al punt 2. al paràgraf anterior, un procés pare, al crear un procés fill, pot controlar el context del fill. Com a context entenem els privilegis, el temps màxim d'execució, la memòria màxima que pot ocupar, la prioritat per executar-se sobre la CPU, on s'envia tot el que s'imprimeix al fer un "printf", d'on es rep el que es captura amb "scanf"...

Per exemple, suposem un programa que només tingui while(1) dins del seu cos i el cridem desde línia de comandes. Mai s'aturaria, però si posem fork-exec-setrlimit, executarà el programa anterior limitant el temps màxim d'execució a 1 segon.

Setrlimit permet establir múltiples límits a diversos aspectes del context, com el temps de CPU, l'ús de la pia o el nombre de fitxers que es poden obrir, per exemple. El seu equivalent en comandes de UNIX és ulimit que limita els recursos assignats als processos que s'executen.

Tot i anomenar els processos pare i fill pugui denotar una jerarquia d'execució, no té perquè ser cert, depen del procés. Si el pare necessita el resultat del fill per continuar la execució, usarà la comanda wait, però si no fa falta, els dos processos es poden executar en paral·lel.

Cal mencionar també que no és necessàri usar el fork i l'exec en parella. Per exemple, Google Chrome a linux usa fork per generar les pestanyes noves.

3.2.2 Entrada-Sortida

Un ordinador té una gran diversistat de dispositus d'entrada i sortida: teclat, ratolí, port USB, ethernet, wifi, pantalla, micròfon, càmera i un llarg etcètera. Antigament hi havia una interfície específica per a cada dispositiu, cosa que feia que quan s'inventava un nou dispositiu s'hagués d'actualitzar la interfície. Amb la arribada de UNIX es van unificar totes les petites interfícies. Ara aquesta idea és universal i implementada a tots els dispositius.

Les característiques de la interfície d'entrada-sortida a UNIX són:

- **Uniformitat:** Tots els dispositius d'entrada-sortida fan servir els mateix conjunt de crides a sistema.
- **Obrir abans d'utilitzar:** Abans de usar el dispositiu cal fer la crida a sistema "open". Quan el SO obre un dispositiu retorna al procés un integer anomenat **descriptor de fitxer** associat al procés, que es podrà usar per llegir o escriure al 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 d'emmagatzement a un buffer intern del nucli, vinguin de xarxa o de disc. Això permet que només s'hagi de fer una crida a sistema "read"per llegir les dades i que el procés les pugui llegir que les demana.
- **Escriptures amb buffer al nucli:** totes les dades a escriure al dispositiu s'emmagatxemen 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 escriue al buffer intern i s'encarrega d'enviar les dades al ritme necessari.
- **Tancament Explícit:** quan un procés ha finalitzat de fer servir un dispositiu s'utilitza la crida a sistema *close*. Això allibera al nucli de tots els recusos necessaris.

Observem que el SO ofereix proctecció entre processos, ja que aquests no poden interferir entre sí, i si es volen comunicar, necessiten el permís del SO. Ara veurem les tècniques bàsiques de comunicació interprocés: la **redirecció** i la **canonada**.

Tot procés (en crear-se pel SO) té creats 3 fitxers per defecte.
Descriptor 0: Conegut com entrada "estàndard", associat per defecte amb el teclat (es captura amb scanf).
Descriptor 1: Conegut com a sortida estàndard, associat per defecte amb la pantalla del terminal (s'imprimeix amb printf)
Descriptor 2: Conegut com sortida d'error, està asociat per defecte amb la terminal.

3.2.2.1 Redirecció Sortida Estàndar a Fitxer

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

1. Fer un 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. Cridem al sistema per a associar el descriptor 1 al fitxer.
3. Executem el programa. Tot el que s'escriu al descriptor 1 s'escriu a disc!

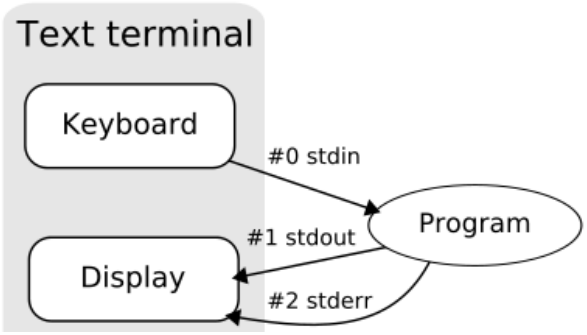

Figura 4: *Redirecció sortida estàndard.*

Tot el que s'imprimeixi al descriptor1 estarà associat al fd, el fitxer de disc, en comptes de la pantalla. Es diu que es redirigeix la sortida estàndard de pantalla a un fitxer.

3.2.2.2 Redirecció Entrada Estàndar des d'un Fitxer

Tal i com hem redirigit la sortida estàndar a un fitxer, podem redirigir un fitxer a l'entrada estàndar. Volem associar el descriptor 0 (scanf) a un fitxer de disc:

1. Fer un 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!

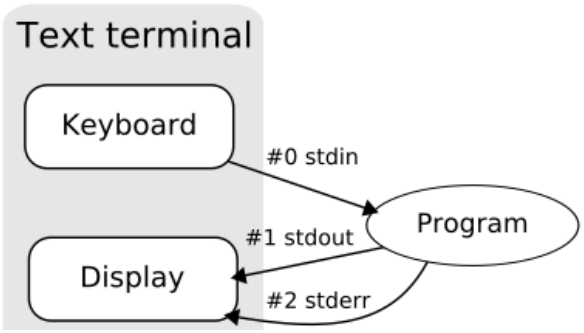

Figura 5: *Redirecció entrada estàndard.*

3.2.2.3 Canonades

Definició 3.3. La **Canonada** és una forma bàsica de comunicació interprocés. Consisteix en dirigir les dades que un procés A genera cap a un altre procés B.

La canonada es crea amb la comanda pipe, que genera:

- Un buffer al SO, que gestiona la operació.
- Dos descriptors de fitxes: un d'escriptura i un altre de lectura.

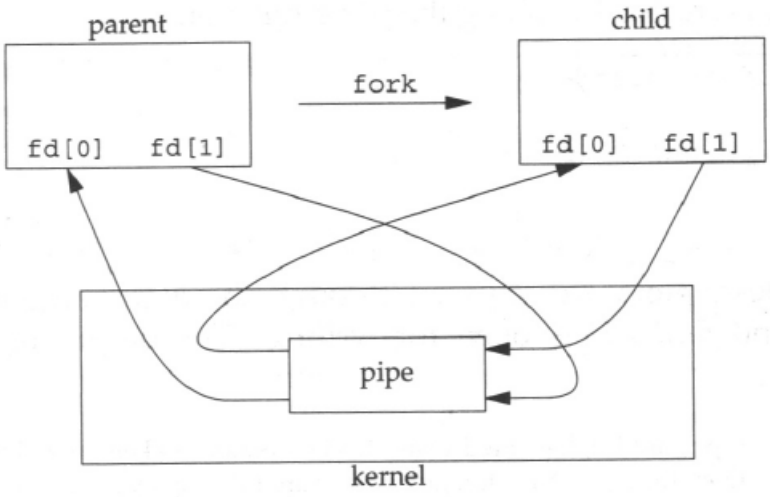

Figura 6: *Imatge d'una Pipe, un pare i un fill creat amb fork*

Per comunicar dos processos mitjantant canonades, aquests **han de tenir una relació pare-fill**.

Si el pare escriu a fd[1], el fill ho podrà llegir d'fd[0], tot i que no té perquè dependre del pare i el fill. fd[1] serverix per escriure a canonada i fd[0] serveix per llegir a canonada.

El buffer és un buffer intern del SO i totes passen per aquest. Té un tamany petit, normalment de 64 KBytes.

Si un procés intenta llegir una canonada que no està escrita, rebrà dades incorrectes, residuals d'altres processos. Si s'ha marcat que un dels processos necessita la informació de l'altre, es bloquejaria el buffer i s'esperaria a que el procés corresponent acabés el que ha de fer.

4 Comunicació Interprocés

En aquesta secció tractarem exclusivament la comunicació entre dos processos, ja que...

Definició 4.1. Una **Comunicació interprocés** (interprocés communication o IPC) és una comuncació entre processos.

i no es refereix a que diferents parts d'un procés es puguin comunicar entre elles.

En aquesta llista (no exhaustiva) anomenen les tècniques de comunicació interprocés: senyals, canonades (pipe), canonades amb nom (named pipe o FIFO), arxius (files), arxius mapats a memòria (memory mapped files) i xarxa (sockets).

Abans de recorre i explicar cada element d'aquesta llista haurem de donar un conceptes prèvis:

Definició 4.2. POSIX **Portable Operating System Interface** és un estàndard Unix per mantenir compatibilitat entre sistemes operatius.

En aquest estàndard, un fitxer pot fer referència a qualsevol sistema de comunicació, tot es tracten igual segons el punt de vista del programador. També tots els fitxers oberts per procés tenen associat un **Descriptor de Fitxer**: un enter no negatiu pels programadors i un índex a un vector del procés amb la informació dels fitxers oberts pel sistema.

Per una altra banda el SO ens ofereix crides a sistema per realitzar les comunicacions interprocés.

- Obrir una comunicació (retornen un descriptor de fitxer): open per a un fitxer a disc, pipe per crear una canonada, mkfito per crear una canonada amb nom, socket per una connexió via xarxa...

- write() i read(): per enviar i rebre informació d'un fitxer. Tenen com a primer argument un descriptor de fitxer.
- close(): per tancar una comunicació. Té només un argument: el descriptor de fitxer a tancar.

4.1 Senyals

Definició 4.3. Els **Senyals** son interrupcions de programari. Proveeixen d'un mecanisme per a gestionar notificacions asíncrones.

El SO pot enviar una notificació asíncrona a un procés (upcall). Llavors els procés s'interromp i es crida la funció que prèviament s'havia indicat al procés.
Un procés pot enviar una notificació asíncrona a un altre procés.
Quan es produeix un senyal cada procés pot indicar al SO què fer:

1. Acció per defecte: Si el procés no indica res.
2. Capturar el senyal: El procés indica al sistema operatiu quina funció cal cridar en produir-se un esdeveniment concret.
3. Ignorar el senyal: el procés pot demanar que s'ignori el senyal. Pot ser perillòs pel procés (per exemple, fer una divisió per 0 i seguir la execució)

La comanda KILL a linux ens permet enviar qualsevol senyal a un procés. En el cas que un procés ens ignorés quan li diem de parar hi ha dos senyals que sempre faran el SO parar el procés: SIGKILL i SIGSTOP respectivament per matar/aturar un procés. SIGCONT permet rependre'l on s'havia deixat.
Els senyals SIGUSR1 i SIGUSR2 són senyals reservats per les aplicacions d'usuari, per a que un procés pugui enviar una notificació a un altre procés. La funció pause() bloqueja el procés fins que en rep un senyal.

4.2 Canonades

També anomenades canonades anònimes, s'utilitzen en processos amb relació pare fill [*3.2.2.3 Canonades*].

4.3 Canonades amb nom (FIFO)

Les canonades amb nom combinen les característiques d'un fitxer de disc i una canonada.

Quan s'obre una canonada amb nom, se li associa un nom de fitxer de disc, que funciona com una canonada FIFO.
Qualsevol procés pot obrir aquest fitxer per lectura o escriptura, només fa falta que tinguin el mateix nom. La funció MKFIFO crea la canonada amb nom i OPEN obre la canonada tant per lectura com per escriptura.

4.4 Arxius

Aquí tenim les aventatges i els inconvenients Arxius:

Avantatges:

- Hi poden accedir tots els processos.
- Un procés es pot situar en un fitxer a la posició que vulgui. Això implica que FIFO no és necessari.

Inconvenients:

- Més complicats de manipular que la memòria.
- Ha de tenir en tot moment dades consistents si múltiples processos hi poden accedir (lectura/escriptura).

Tenim dues maneres d'accedir a fitxers per llegir i escriure: **crides a sistema** i la **llibreria estàndard**.

4.4.1 Crides a Sistema

El SO ens ofereix una sèries de crides a sistema per manipular fitxers. Les més importants són:

- OPEN i CREAT: Obrir i crear un fitxer a disc. Retornen el descriptor de fitxer.
- CLOSE: Tanca un fitxer. Té com a paràmetre el descriptor de fitxer.
- READ i WRITE: llegir i escriure dades de qualsevol fitxer. Té com a paràmetre el descriptor de fitxer.
- ISEEK: Establir la posició dins del fitxer a disc (per llegir o escriure-hi). Té com a paràmetre el descriptor de fitxer.

Per optimitzar-ne l'accés a disc usa el buffer intern. Les dades s'emmagatzemen a disc tal com estaven emmagatxemades a memòria (per tant també es llegeixen en aquest ordre) i hi podem escriure (llegir) amb una sola instrucció fent servir vectors de dades. Com més crides a sistema usem, més ineficient és (per això el vector de dades és millor).

Les funcions OPEN, READ, WRITE criden directament al SO. El SO manté un buffer intern per a cada fitxer obert (comú a tots els procesos).
Write escriu les dades al buffer intern i retorna immediatament no escrivint a disc, la escriptura real es farà més tard. Retorna el nombre de bytes que ha pogut escriure (potser la mida màxima del fitxer està limitada)
Read comprova si les dades estan al buffen intern. Si no hi són allà, les llegeix de disc i el procés queda bloquejat. Retorna el nombre de bytes que ha pogut llegir, perque és possible que no hagi pogut llegir tots els que se li han demanat.
Si un procés escriu a un fitxer i un altre hi llegeix després es llegirà la dada correcta tot i no haver llegit a disc.
Qualsevol de les funcions anteriors retornaran un nombre negatiu si hi ha hagut un error, per exemple intentar escriure a un fitxer obert per lectura.

4.4.2 Llibreria estàndard

La llibreria estàndard "stdio"ens ofereix les següents funcions per manipular un fitxer a disc.

- FOPEN: obrir un fitxer
- FCLOSE: tancar un fitxer
- FREAD I FWRITE: llegir i escriure dades. Crida internament a read i a write del SO.
- FSCANF I FPRINTF: llegir i escriur dades en ASCII. Crida internament a read i a write del SO.
- FSEEK: defineix la posició actual dins del fitxer.

Les funcions per gestionar fitxers usen una estructura FILE, que es caracteritza per:

1. Un Int per emmagatzemar el descriptor de fitxer associat.
2. Un buffer propi a la memòria d'usuari per emmagatzemar dades.

Les propies funcions de la llibreria estàndard fan les crides a sistema quan és necessari. La estructura FILE i les funcions que són de "stdio"(les mencionades anteriorment) utilitzen un buffer propi del procés que no és visible als altres processos.
FPRINTF o FWRITE escriuen les dades al buffer de FILE i quan és ple es crida WRITE (crida a sistema) que les escriu al buffer intern.

FSCAN o FREAD llegeixen les dades al buffer de FILE i si no hi són disponibles es fa la crida a sistema per omplir el buffer de FILE. Mentrestant el procés que ha fet la crida es queda bloquejat. Així la llibreria estàndard aconsegueix minimitzar l'ús de write i read, que són crides fora de memòria i per tant molt lentes.

Hi ha casos en els que potser no és adequat usar aquestes funcions per la comunicació interprocés, per exemple, si hi ha múltiples processos cada procés té el seu propi buffer. Una dada que un procés escrigui al seu buffer no es podrà veure immediatament a cada procés.

4.4.3 Crides a Sistema vs. Llibreries

Les funcions open, read, write criden directament al SO.

- Bones per comunicació interprocés.
- Usar-les directament pot reduir l'eficiència ja que una crida a sistema és costosa. (ús del buffer compartit dels fitxers per a poder cridar un vector de dades)

Les funcions fopen, fwrite, fread... són de la llibreria estàndard "stdio"

- Internament criden a sistema (open, close, write...)
- Eficients per l'accés a disc ja que minimitxen les crides a sistema.
- Comunicació interprocés poc òptima, ja que tenen un buffer d'usuari.

4.5 Arxius Mapats a Memòria

Es pot mapejar un arxiu de disc a memòria cosa que comporta certs beneficis:

- Accedir el fitxer com si fos memòria (sense READ i WRITE).
- El SO posa a memòria la part necessària del fitxer al llegir-lo i desa la part necessària al fitxer si canviem el vector.
- Molt eficients amb la comunicació pare-fill (no exclusiu, però).
- Super usat a la memòria virtual.

4.6 Xarxes

La comunicació interprocés via xarxa permet comunicar múltiples processos entre el mateix o diferents ordinadors. El SO ofereix 8 funcions bàsiques per manipular sockets: Socket, bind, listen, accept, connect, read, write i close.

Un ordinador s'identifica mitjançant IPs (161.116.83.153). Un procés A pot acoplar-se a un port i un altre procés B pot enviar un missatge al procés A, cosa que obre un canal de comunicació bidireccional amb IP, és a dir, si B escriu un missatge arribarà a A (i a la inversa).

5 Planificació de Processos

En aquest apartat tractarem com el SO organitza els processos. Parlarem del seu estat, la planificació, conceptes darrera d'aquesta i algorismes sobre aquests.

5.1 Estats d'un Procés

Un procés pot tenir tres estats: **preparat**, **executat** i **bloquejat**. Entre els dos primers l'estat hi varia segons la voluntat del SO, però per accedir a bloquejat ha d'estar en execució i després anirà a preparat.

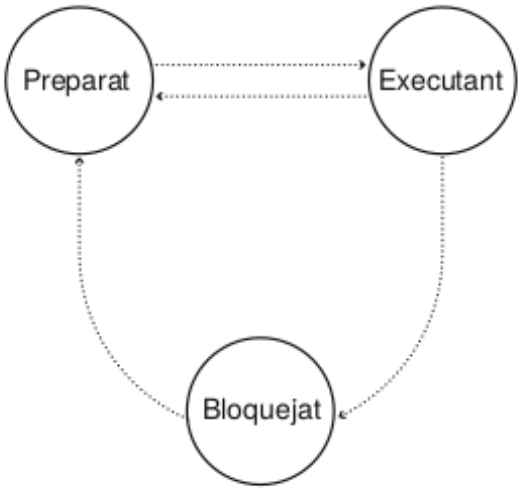

Figura 7: *Graf dels tres possibles estats d'un procés. Atenció al sentit de les fletxes!*

Preparat: un procés està disponible per a ser executat però encara no s'ha realitzat la seva execució. El SO els té guardats com a candidats a execució.
Executant: un procés està executant si està a algun dels processadors disponibles.

Un procés pot alternar entre preparat i executant múltiples vegades. Tant el SO com el procés tenen control sobre l'estat d'un procés, però acostuma a ser una decisió del SO, no del procés.

Quan un procés canvia a preparat o bloquejat es produeix un **canvi de context** (ho veurem més endavant).

Quan es deixa d'executar un procés, el SO emmagatzema tota la informació del procés al BCP (Bloc de Control de Processos) i quan està escollit el següent procés a executar es restaura tots els valors del BCP, es carreguen a la CPU i es tornen a executar on s'havien deixat.

La informació que es guarda el BCP al fer un canvi de context és:

- Identificador de procés (INT)
- Estat del procés (preparat, executant o bloquejat)
- Els registres de CPU si està preparat o bloquejat
- Privilegis o prioritat
- On resideix de la memòria física, fitxer executable associat...

5.2 Canvi de Context

Definició 5.1. Un **Canvi de Context** és el mecanisme que fa servir el SO per canviar l'execució d'un procés a un altre.

Es pot realitzar gràcies al mode dual de funcionament. Per exemple: Si es produeix una interrupció de temporitzador, el maquinari interromp el procés actual, emmagatzema els valors dels registres de la CPU a memòria i crida a una funció. Si el SO decideix canviar de procés emmagatzema la informació de memòria a la BCP i sobreescriu amb els nous valors del procés a executar, es carrega a la CPU els nous valors i ja hem fet un canvi de context.

Els canvis de context es poden produïr quan hi ha:

- Interrupció de temporitzador: El planificador pot decidir deixar d'executar el procés actual i executar un altre.
- Bloqueig de procés: el planificador ha de decidir quin dels processos està preparat per executar-se.

- Altre tipus d'interrupció: arribar dades de xarxa, haver-se llegit de disc... S'interromp la execució del procés. El planificador pot decidir executar el procés que ha demanat les dades o continuar l'execució del procés del moment.

5.3 Planificació

Definició 5.2. El **Planificador** determina l'ordre d'execució dels processos.

El planificador decideix l'ordre mitjançant algorismes. Aquests poden funcionar bé en un entorn de càrrega computacional alta però malament en entorns amb càrrega entrada-sortida alta (o al revés). Així que el nostre objectiu són algorismes que funcionen bé a entorns amb càrrega variada.

Aquest algorisme no és únic, cada política de planifició té avantatges i inconvenients, però abans d'entrar-hi hem de definir tasca:

Definició 5.3. Una **Tasca** és una part concreta d'un procés, i poden tenir característiques diferents entre elles

Per exemple, un procés pot tenir tasques diferents com calcular una area, desar les dades a disc, recuperar valors d'altres execucions... A partir d'ara parlarem sobre tasques, no sobre processos.

5.3.1 Planificació Uniprocessador

Agrupem sota planificació uniprocessador els algorismes bàsics.

5.3.1.1 Round Robin

El Round Robin (o FIFO circular) consisteix en que el planificador disposa d'una llista de tasques i les executa des de al pimera fins la última, de manera circular.
Cadascuna d'aquestes tasques tenen un temps d'execució limitat anomenat **Llesca de temps**. Quan s'acaba la llesca, el planificador canvia de context a una altra tasca. El planificador de temps no és la unitat mínima, una tasca pot executar-se menys estona que una llesca de temps.
Quan es produeix un canvi de context es posa la tasca actual al final de la llista de tasques (estat preparat) i s'agafa la primera de la llista (se l'hi canvia l'estat a executar).

El valor de la llesca de temps és escollible (acostuma a estar entre 10 i 100 milisegons) i important ja que una tasca de temps massa petita sobrecarregarà la planificació (molt

temps perdut als canvis de context) i si la llesca de temps és massa gran les tasques hauran d'esperar molt per obtenir el seu torn i executar (és a dir, el temps de resposta serà baix).

El cost associat al canvi de context és poc, ja que només hem de guardar els registres de dades i carregar la CPU al nou procés.
Si la tasca no té les dades que necessita a la cache haurà d'accedir a memòria RAM per copiar les dades altre cop a la memòria cau, cosa extremadament lenta.

Conclusió:

- Si la llesca és massa gran:
  - Les tasques hauran d'esperar molt per executar-se.
  - Com que hauran estat més estona executant-se, no hi hauran dades del procés que hem d'executar ara a la caché, així que s'haurà d'accedir a memòria per carregar les del nou procés (MOLT LENT).
- Si la llesca és massa petita:
  - Perdrem molt temps als canvis de context (ja que en farem molts)
  - El procés no tindrà molt temps de processament, així que no és massa òptim.

La planificació Round Robin no és adequada en un entorn en càrrega variada:
Suposem tres tasques, dues amb alta càrrega computacional (CPU bound) i una amb càrrega entrada-sortida (IO bound) que només necessien la CPU 1ms i 10ms de disc.

- La tasca IO bound podria tenir ocupat el disc només i acabar la feina ràpid.
- Les tasques CPU bound i una llesca de 100ms fan alentir la tasca IO bound 20 vegades del que tardaria en normal. Per cada cop que la tasca executa, necessita la CPU 1ms, però ha d'esperar-ne 200!

La solució seria prioritzar el procés IO bound perquè no hagi d'esperar tant.

Com hem vist, Round Robin no és bona solució si hi ha tasques CPU bound i IO bound juntes. Les tasques amb càrrega alta en entrada-sortida no necessiten massa temps de CPU per la següent operació entrada-sortida.

Quan Round Robin sí que és un algorisme adequat?

- **Prioritzant la CPU Bound:** Quan totes les tasques són del mateix tipus (sobretot amb CPU bound)
- **Prioritzant velocitat uniforme de dades:** En un servidor de streaming de vídeo/música és important la uniformitat de la freqüència de dades, encara que s'enviin a una velocitat més altra que la del receptor.

5.3.1.2 Max-min Fairness

L'algorisme consisteix a assignar a cada instant de temps la CPU a la tasca preparada que ha rebut menys temps de CPU.

Suposem l'exemple anterior i que cada 0.5ms fem un canvi de context a la tasca que ha rebut menys temps de CPU. El max-min dóna a les tasques petites l'assignació que demanen i després reparteix la resta entre les tasques grans. Els recursos s'assignen en ordre creixent de demanda i cap tasca obté més recursos dels que demana. Si hi ha una barreja de tasques és una assignació més justa. Si no hi ha barreja, acaba sent una Round Robin, ja que cada tasca rep el mateix temps.

El problema és que la seva implementació requereix una cua prioritària, així que implica un gran cost computacional. Potser s'han de pendre centenars o milers de decisions per segon i no convé tenir una llista prioritària ideal, que es tradueix en una simplificació del Max-min Fairness.

5.3.1.3 Multi-level Feedback Queue

El SO com Windows, Mac o Linux usen aquest algorisme per planificar tasques, una simplificació de l'algorisme anterior.

Al MFQ hi trobem diferents cues Round Robin, cada cua amb una prioritat i una llesca de temps diferent.
Les tasques d'una cua al mateix nivell es planifiquen amb Round Robin. Una tasca amb prioritat alta es gestiona abans que alguna amb prioritat baixa. Les tasques es mouen entre les cues de prioritat per obtenir la màxima equitat possible. Quan una tasca exhaureix al seva llesca de temps (sense haver-se bloquejat) es mou a una cua de prioritat més baixa. Si es bloqueja, es mou amunt.

5.3.2 Planificació Multiprocessador

A l'actualitat és ben normal que un ordinador tingui múltiples procesadors. Per trobar les maneres de fer-los servir efectivament per a tasques seqüencials i com modifiquem els algorismes que sabem per a un processador hem de tenir en compte que cada processador té la seva memòria cau.

Com hem explicat a l'apartat de MFQ, els SO actuals tenen aquest algorisme implementat. Per tant, una tasca s'executa a un processador només, ja que cadascun té la seva propia memòria cau i accedir a memòria per copiar-ho a la caché es car. Ara bé, si un processador es queda sense res a executar pot robar la tasca d'un altre processador.

5.3.3 Planificació en Temps Real

En alguns sitemes el planificador ha d'assegurar que determinades tasques tenen terminis per executar-se, per exemple: quan un cotxe que controla el bloqueig dels frens ha d'ocorrer a temps per a ser efectiu, o per reproduir un vídeo cada imatge ha de ser visualtzada quan toca per evitar que no es vegi bé.

Tenim dos mètodes per assegurar que els terminis es compleixen:

- Sobre-aprovisionament: les tasques de temps real han d'ocupar només una fracció de la capacitat de processament del sistema, així seran planificades ràpidament sense haver d'esperar altres tasques.
- Prioritització per termini: les tasques en temps real es prioritzen de manera que s'executen primer les que tenen un termini més proper (earliest deadline first). Funciona si les tasques no necessiten d'entrada-sortida, és a dir, amb tasques computacionals.

5.3.4 Planificació en Sistemes Sobrecarregats

Diem que un sistema es sobrecarrega si tenim moltes dades a processar i pocs recursos per fer-ho, per exemple en els sistemes web, on el nombre de conexions pot canviar al llarg del dia.

Per evitar que un sistema se sobrecarregui podem...

- Rebutjar peticions de connexió que d'agafar-les ens portin directament a la sobrecarrega.
- Reduir el temps de resposta en determinades peticions. Amazon si la informació demanada es demora massa, retorna una pàgina estàtica, Youtube redueix la qualitat de video abans de fer-te esperar...
- Realitzar més feina per tasca a mesura que augmenta la carrega, sent aquesta la que fan servir més sistemes.

6 Concurrència

A un ordinador, mòvil, servidor o aparell, moltes activitats tenen lloc al mateix temps, desde més d'un usuari accedint a diverses pàgines web a el mateix navegador carregant-les a la vegada i, filant més prim, la interficie gràfica mateixa ens oculta moltes coses.
Apart, tota aquesta idea es reforça amb el nostre pensament de programadors: tenim

la il·lusió de que el codi és seqüencial, que comença a una funció principal i executa en l'ordre que nosaltres hem estipulat fins que acaba el programa... Però és possible que la aplicació executi diverses parts del codi al mateix temps?

Definició 6.1. Un **Fil (thread)** executa una part del codi d'un procés i canvia les variables associades a aquesta part.

Els fils d'un procés (al contrari que processos independents) comparteixen l'espai de memòria del procés. Cada fil té la seva pròpia pila i registre de la CPU. Per comunicar-se no els hi fa falta fer servir una pipe o una xarxa com als processos ja que comparteixen espai de memòria.

Elements d'un procés:

- Espai d'adreçes
- Fitxers oberts
- Llista de fils d'un procés
- Altres

Elements propis d'un fil:

- Comptador de programa
- Registres de la CPU
- Pila
- Estat (preparat, bloquejat, execució)

Al tema de planificació es parlava de planificar tasques... A l'actualitat es planifiquen fils. Cada fil té un estat diferent i el planificador canvia de context a nivell de fil, no de tasques.
Cada fil pot estar executant una part diferent del codi del procés i el mateix codi pot ser executat per diferents fils a la vegada.

6.1 Avantatges de la Programació Multifil

- Estat Propi: cada fil té el seu propi estat. Això millora el temps de resposta de la aplicació.
- Simplificació del codi: el codi es simplifica, sobretot tractant de nivells asíncrons.
- Facilitat de comunicació interfil: la comunicació entre dos fils és extremadament senzilla.

S'ha de matisar: la associació de programació multifil amb multiprocessadors no és evident ni tampoc necessària. Hi ha sistemes uniprocessador amb multifil que tenen avantatges evidents.

6.2 Processos vs. Fils

Recordem: Als processos la comunicació pot ser entre processos executant al mateix o diferents ordinadors i es realitza mitjançant serveis que ofereix el SO.

Als fils: Tots els fils d'un procés s'executen al mateix ordinador. La comunicació es realitza a través de l'espai de memòria del procés sense necessitat d'usar els serveis del SO. La complexitat es troba en la seva comunicació i sincronització.

Per exemple, si volguessim dissenyar una aplicació web, com decidiem quina implementació és millor? Múltiples processos o múltiples fils? La següent taula ens dóna la resposta més aproximada.

	Processos	**Fils**
Quantitat dades a compartir	Moltes	Poques
Seguretat (Operació invàlida)	Mata només el procés	Mata tot el fil

La conclució és: usarem processos a aplicacions crítiques, fils en menys importants.

7 Memòria Virtual i Traducció d'Adreces

Cada procés creu que té assignat tot l'espai de memòria possible i que és lineal, és més, si cridem a MALLOC amb un nombre de bytes major que la memòria RAM no tindrem error de compilació. Però que un procés tingui l'espai de memòria de tot el PC disponible per a guardar-hi dades ordenades de manera lienal no és cert. Això com s'aconsegueix?

L'entorn de memòria virtual s'aconsegueix amb un sistema que tradueix qualsevol direcció que genera un procés a una direcció física de RAM.

Fites del sistema de traducció d'adreces.

- Protecció dels processos per aïllament entre ells. Permet construir sandboxes per executar aplicacions externes.
- Auto-aïllament d'un procés: Cada part del codi està protegida entre sí.
- Execució d'un procés: Pot executar-se sense tenir tot el codi a memòria RAM.
- Comunicació interprocés: Perment tenir una zona de memòria compartida entre processos per a compartir dades.
- Codi executable compartit: Diferents processos poden compartir codi; per exemple llibreries comunes.

- Gestió de memòria de la pila o memòria dinàmica: El SO ubica memòria per aquestes zones a mesura que creixen.
- Fitxers mapejats a memòria: Es pot accedir a un fitxer accedint a posicións de memòria com si fos un vector.

7.1 Traducció d'adreces

La traducció d'adreça la duu a terme maquinari especialitzat gestionat pel SO. Totes les direccions de memòria generades per processos són traduïdes a una adreça de memoria física.

La traducció de les direccions de memòria virtual a física es fan a nivell de maquinari, amb la Memory Mapping Unit (MMU). El SO s'encarrega de gestionar-la i configurar-la per que es tradueixi de forma correcta. Un cop traduïda es comprova si l'adreça virtual correspon a una física; si no ho està, es produeix una excepció.
Quan aquesta excepció es produeix el SO comprova si l'adreça virtual pertany al procés.

- Si no hi pertany es mata el procés (amb el senyal SIGSEGV)
- Si pertany al procés, el SO gestionarà que l'adreça virtual es mapegi a una adreça física perquè el procés pugui accedir a la dada demanada

Veurem dues maneres d'implementar aquesta traducció: base i límit amb memòria segmentada (sistema antic) i memòria paginada (sistema actual).

7.2 Base i Límit

Uns dels esquemes bàsics de traducció d'adreces fa servir dos registres: base i limit. Cada procés té el seus propis registres base i limit i només poden ser modificats per instruccions priviliegiades.

La traducció realitzada amb base i limit consisteix a, cada cop que s'accedeix a memòria, se suma la base a l'adreça. Si aquest nombre és major que límit es produeix una excepció.

Tenint només la base i el límit s'hi troben importants defectes:

- Pila i memòria dinàmica predeterminada: Amb només dos registres no podem expandir la memòria. S'ha de preveure la màxima memòria que ocuparà el procés quan el carreguem a disc, previsió que no és fàcil fer.
- Linealitat de memòria: tot el procés ocupa un espai seguit, així que no es pot fragmentar.

- No auto-aïllament: Un procés pot escriure sobre la seva zona de memòria executable, per exemple.
- Memòria individual: no podem aconseguir que els processos comparteixin zona de memòria.
- Comunicació interprocés nul·la: Els processos no es poden comunicar entre sí.

7.3 Memòria Segmentada

Per arreglar els problemes de la base i límit que mencionem més amunt definim la segmentació:

Definició 7.1. La **Segmentació** és la funció de que cada procés pugui donar múltiples registres base i límit per a ell mateix.

Cada parell de registres té asociada una porció de l'espai d'adreces anomenada segment. Cada un d'aquests s'emmagatzema de forma contingua a memòria i pot tenir una mida variable.

Cada adreça virtual té dos components:

- El número de segment: la base i el límit asociats
- L'offset dins del segment

En aquest sistema de memòria segmentada els bits alts del segment s'associen al número de segment i els baixos a l'offset. El nombre màxim de segments depèn totalment dels nombre de bits asociats a representar el segment.
El SO gestiona que el maquinari pugui assignar permissos (lectura, escriptura, execució) a cada segment.
Si un procés produeix una adreça de memòria virtual invàlida (que no és del propi procés), es produeix una excepció i el SO genera un senyal SIGSEGV que envia al procés.

Avantatges de la memòria segmentada contra Base i Límit:

- Auto-aïllament: Permet que un procés es protegeixi a sí mateix diverses parts del codi.
- Fragmentació de memòria: l'espai de memòria d'un procés pot estar fragmentat en diversos trossos.
- Compartició de segments: diversos processos poden compartir codi (llibreries) compartint els registres base i limit d'un segment.

- Comunicació interprocés: Diversos processos poden comunicar-se entre sí compartint una zona de memòria.

Tot i que aquest sistema també te desaventatges marcades:

- La memòria acaba dividida en trossos ocupats i lliures a mesura que passa el temps.
- Quan creem un nom procés hem de crear nous segments. Potser no hi ha cap zona lliure prou gran per a un segment, però si suméssim totes les que estan lliures potser si que hi hauria espai. Quan això passa, el SO pot compactar les regions per a unificar les regions lliures. Cal canviar la base i el límit per a cada segment i moure la memòria física. Aquesta és una operació increïblement costosa.

7.4 Memòria Paginada

La memòria sencera es divideix en blocs de mida fixa anomenats marcs de pàgina.

Definició 7.2. Un **Marc de Pàgina** és la unitat mínima accedible i ocupable que hi ha a memòria.

La traducció es realitza amb una taula, única per procés i gestionada pel SO.

Cada marc de pàgina té mida de 2^n on $n \in \mathbb{N}$, osigui una potència de 2. Els bits alts es fan servir com a índex de la taula i l'offset la posició del marc.
Per traduir només cal substituir els bits alts de la adreça virtual pels indicats a la taula.

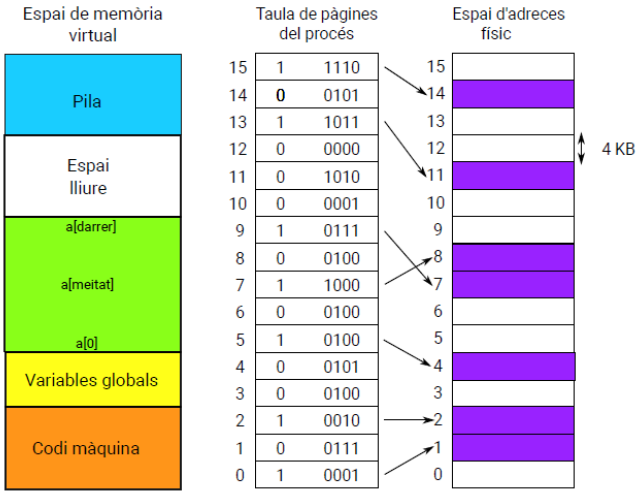

Figura 8: *Representació del paginat i la memòria física*

Per exemple, el fa figura tal és una estructura de 16 bits amb marcs de pàgina de 4KBytes. Si dividim $2^{16}/2^{12} = 2^4$ marcs de pàgina. Es faran servir doncs els 4 bits superiors de

l'adreça per referenciar un marc.
La adreça virtual 0000 1111 0000 1111 es mapa a 0001 1111 0000 1111 (seguint la fletxa es veu, ja que 0000 = 0 i anem a la primera posició de la taula de marcs de pàgina) però la 0001 1111 0000 1111 produeix una excepció (ja que 0001 = 1 i allà no hi ha cap fletxa cap a memòria física). Això el SO ho sap perque el primer bit (el que no està agrupat en 4) és 1 si el marc virtual al que fem referencia està a memòria. Si és 0, vol dir que no hi és i que allà no se sap que hi ha.

Cada entrada a la taula conté:

- Traducció al marc físic: on són realment les dades a memòria.
- Propietats asociades: Lectura, escriptura, execució, disponibilitat...

Si un procés fa una operació invàlida es produeix una excepció.
Les taules es guarden a memòria RAM.

Amb dos processos es permet gestionar la protecció de memòria entre ells, ja que cadascun guarda als marcs que pot intercalant-se entre ells tal i com es veu a la figura seguent.

7.4.1 Copy-On-Write

Quan tractem amb la comanda FORK es fa servir una tècnica anomenada Copy-On-Write per optimitzar el procés.

Definició 7.3. El **Copy-On-Write** es realitza quan es fa un FORK. Consisteix en marcar com a lectura la taula del procés fill i es copien a una memòria física diferent quan el fill demana escriptura. Tots els altres marcs de pagina apunten a la mateixa adreça de memòria física.

Suposem un exemple on el pare assigna una variable a=2 i el fill la mateixa variable a a=1.
Abans de fer el fork tenim el pare omplint certes parts de l'espai d'adreces físic. Quan es fa el FORK el SO **duplica la taula de pàgines del pare però no la memòria física** i es marquen totes només de lectura pel fill.
Quan el fill intenta reassignar a=1 es produeix una excepció (fallada de pàgina) i quan el SO la captura és quan duplica les pàgines físiques. Es surt de la fallada i la execució continua amb normalitat.

Com es pot veure en l'exemple anterior, el codi màquina es comparteix entre pare i fill i compartir zones de memòria (llibreries, codi, variables...) és molt senzill amb la memòria paginada.

7.4.2 Inicialització

L'assignació de marcs de pàgina físics només es fa a mesura que hi accedim realment, és a dir: inicialitzar el mapa de memòria virtual no assigna parts de les adreces de memòria física.

Suposem que cridem MALLOC de manera que només inicialitzi el mapa de memòria virtual.
Quan es crida MALLOC es reserva l'espai virtual només.
Després dels accessos si no existeix l'adreça a l'espai físic tenim una excepció (fallada de pàgina). El SO cerca una pàgina disponible a l'espai físic, es retorna de la fallada i ja es realitza l'accés.

7.4.3 Aventatges i Inconvenients

La memòria paginada resol els problemes de memòria lliures, ja que tots els blocs tenen el mateix tamany i estan repartits des del principi, el SO no s'ha de preocupar d'ajuntar memòria lliure solta (problemes memòria segmentada) i només ha de mantenir la taula a la memòria RAM.

Però per molt ben segmentada que estigui la memòria física és limitada: tots els marcs de pàgina poden estar ocupats. Si un procés necessita nous marcs s'ha de decidir quins es descarten de la memòria física. Això es fa amb algorismes especialitzats per aquesta tasca.

Un altre problema és la mida de les taules. Suposem un sistema x86 de 32 bits on els marcs mesuren 2^{12} bytes. Això implica que cada taula té 2^{20} entrades, i si cada entrada ocupa 4 bytes tenim que una taula ocupa 4Mbytes i n'hi ha una per a cada procés. En sistemes de 64 bits les taules ocuparien gigues! Com arreglem aquest problema de memòria?

7.4.4 Esquemes de Traducció Multinivell

La solució radica en no fer servir una taula, sinó una estructura multinivell.
Tots els sistemes que mencionarem usen el sistema de paginació, el quid de la qüestió es troba a com arribem de ràpid a accedir a la pàgina que volem.

7.4.4.1 Paginació Segmentada

L'arbre té dos nivells:

- 1r nivell: apunta a una taula. Anomenat **Segment**.

- 2n nivell: apunta als marcs de pàgina físics.

Les propietats d'accés es poden accedir a nivell segment.

7.4.4.2 Paginació Multinivell Segmentada

És com la paginació segmentada però l'arbre té múltiples nivells. Cada segment conté una taula multinivell, fent-lo l'emprat pels sistemes x86. El número de segment es guarda a un registre de la CPU separat per qüestions històriques. Els sistemes de 32 bits tenen un sistema multinivell de dos nivells: 10 bits al primer, 10 al segon i 12 per l'offset.
Els sistmes de 64 bits només s'utilitzen 48 bits de l'espai d'adreces virtuals. S'usen 4 nivells de l'arbre (48=9+9+9+9+12) i només s'ubiquen les que son realment utilitzades pel procés.

Per arreglar la ineficiència d'accedir a elements d'un vector dins dels elements d'un vector múltiples vegades? Col·loquem memòries caché específiques que es dediquen a controlar les entrades perquè sigui més ràpid.

7.5 Fitxers Mapats a Memòria

El SO permet manipular fitxers mapant-los directament a memòria.

Característiques bàsiques: - Qualsevol fitxer de disc pot ser manipulat fet servir l'espai d'adreces virtuals.
- Els blocs es traslladen a memòria si són referenciats i es guarden a disc automàticament si són modificats per la aplicació o quan la aplicació es tanca.

Això ens duu a la paginació sota demanda.

7.5.1 Paginació Sota Demanda

La paginació sota demanda permet accedir a més memòria de la que es pot físicament. Si una aplicació accedeix a una pàgina no disponible a memòria RAM el SO la porta de disc a memòria.

Funcionament:

1. Quan intentem accedir a una posició vàlida que no està carregada a memòria, saltarà una excepció de pàgina.

2. El SO captura l'excepció de pàgina i detecta que l'adreça és vàlida. Llavors converteix l'adreça virtual a una posició de bloc a disc.

3. Es llegeix el bloc a disc. Nota: la lectura bloqueja el procés, així que el SO ha de fer altres coses, com executar un altre procés.

4. Quan s'acaba la lectura es produeix una interrupció al processador, es crida el SO i aquest actualitza la taula multinivell amb la nova pàgina carregada. Si no hi havia lloc per la pàgina, el SO n'haurà expulsat una perque aquesta hi càpiga.

5. Es recupera l'execució del procés on s'havia produït la excepció de pàgina.

Quan s'expulsa una pàgina de memòria el SO pot haver-la de guardar a disc. Ho sap perque el maquinari guarda un dirty bit a la taula que indica si la pàgina ha estat modificada.

Per decidir la expulsió d'una pàgina s'usa el *use bit*, un bit de la taula de pàgines on el maquinari indica si s'ha accedit a la pàgina. Per saber quines pàgines tenen el bit, el SO recorre periòdicament les taules per saber quines pàgines han estat accedides i fer-ne el reset de taula. Pot ser descartada una pàgina que no pertanyi al procés en aquest moment.

7.5.2 Memòria Virtual

La memòria virtual és una generalització dels fitxers mapats a memòria. Tots els segments o regions estàn mapades a disc.

Quan s'executa un procés es mapen a disc l'executable, les llibreries, les variables globals, la pila... El SO carrega a memòria les pàgines a mesura que es va executant el programa. Com que la memòria no és il·limitada, al sortir d'un procés no s'emmagatzemen les dades a disc com es fa amb els fitxers mapats a memòria.

Gràcies a la memòria virtual podem tenir executant més processos del que realment caben a memòria, ella mateixa té un sistema que gestiona les pàgines encarregades a cada procés.

El balanceig de quantes pàgines té cada procés a memòria és delicat, ja que un ordinador modern pot executar només 100 fallades de pàgina per segon (en el context de que el processador executa milers de milions d'instruccions, 100 és un nombre petit).
Quan es produeix una d'aquestes fallades és possible que es descarti una pàgina: ho fa basant-se en el **use bit** amb el criteri de les pàgines no accedides recentment seran les descartades.

L'executable o les llibreries dinàmiques es mapen amb el fitxer original amb només lectura. Les zones de la pila o memòria dinàmica es mapen a swap perque s'hi puguin guardar les pàgines que no hi caben a memòria. S'hi guarden les modificacions de les pàgines però al sortir del procés aquestes dades es perden.

7.6 Conclusions del Sistema de Memòria

El sistema de memòria permet:

- Traduir una direcció virtual a una física mitjançant una taula.
  - Hi ha una taula per a cada procés.
  - Cada taula la gestiona el SO, no el procés.
- La taula permet:
  - Protegir els processos entre sí (no deixant que un procés escrigui sobre un altre procés)
  - Que diferents processos comparteixn codi: des de llibreries fins a variables.
- El sistema de Memòria Virtual gestiona quines pàgines es carreguen a memòria física i allibera les adients per a encabir-ne més.