La gestió de processos

Teodor Jové Lagunas
Josep Lluís Marzo i Lázaro
Enric Morancho Llena
José Ramón Herrero Zaragoza

PID_00214799

UOC
Universitat Oberta de Catalunya

Índex

Introducció

Aquest mòdul didàctic se centra en l'**estudi de la vida dels processos** des del punt de vista de les crides al sistema que els permeten manipular. Així doncs, en lloc de veure, com ja hem fet en aquesta assignatura, la creació de fitxers executables i els mecanismes que té el sistema operatiu per a comunicar-se amb els programes (les crides al sistema), ara ens centrarem en **l'estudi de les operacions que permeten gestionar els processos**, concretament les que els permeten crear, destruir i modificar. Abans de res, però, haurem d'aprofundir breument en el concepte de procés, ja vist en aquesta assignatura, i en la gestió interna dels processos que fa el sistema operatiu (SO).

Objectius

Els materials didàctics d'aquest mòdul contenen les eines necessàries per a assolir els objectius següents:

1. Entendre el concepte de *procés* i de *flux d'execució* (*thread*) com a objectes gestionats pel SO.

2. Conèixer les diferents parts que componen un procés i veure com es representa a l'interior del sistema operatiu.

3. Entendre els estats en què pot estar un procés i els motius pels quals un procés pot canviar d'estat.

4. Saber què és el cicle de vida dels processos i conèixer les relacions entre processos.

5. Comprendre el concepte d'*herència* i veure quines repercussions té l'herència en la creació de processos.

6. Conèixer les diferents possibilitats que ens ofereix el fet de poder canviar alguns dels elements que componen els processos, sobretot en relació amb els reencaminaments.

7. Conèixer les crides bàsiques de gestió de processos d'UNIX i de Windows.

8. Conèixer el funcionament bàsic dels flux o fils d'execució (*threads*) POSIX (*pthreads*).

1. El procés: un cop d'ull des de l'interior del sistema

Un **procés** és bàsicament un entorn format per tots els recursos necessaris per a poder executar programes. Des del punt de vista del SO un procés és un objecte més que s'ha de gestionar i al qual s'ha de donar servei.

Per gestionar els processos i donar-los serveis, el sistema operatiu ha de proporcionar eines o ha de portar a terme accions que permetin aconseguir els objectius següents:

- Crear i eliminar processos.
- Garantir que els processos disposin dels recursos necessaris per a avançar en l'execució.
- Actuar en casos excepcionals[1] durant l'execució del procés.
- Proporcionar els mecanismes necessaris perquè el processos es comuniquin, ja sigui per a intercanviar informació, ja sigui per a sincronitzar-se durant l'execució.
- Mantenir estadístiques sobre el funcionament dels processos.
- Temporitzar l'execució d'un procés: fer que un procés s'executi cada cert temps.
- Altres serveis miscel·lanis.

(1)Considerem casos excepcionals successos com ara les interrupcions, els errors, etc.

En aquest apartat fem una introducció a la gestió que el SO porta a terme sobre els processos. Veurem les estructures de dades que representen internament un procés i com diversos processos es poden executar concurrentment en un únic processador.

1.1. Els elements d'un procés i la seva representació

Tal com hem vist, el SO construeix els processos d'acord amb un conjunt d'elements que són necessaris per a l'execució d'un programa. Per poder gestionar aquest conjunt de recursos com un tot, el sistema reuneix informació de

(2)El terme *process control block* o PCB és l'equivalent anglès del terme *bloc de control de processos*.

tots en una estructura de dades anomenada **bloc de control de processos**[2] o **PCB**. A cada procés li correspon el seu PCB propi. Els camps més importants que configuren els PCB són els següents:

1) L'identificador del procés (PID[3]). És el codi únic que identifica de manera biunívoca cada un dels diferents processos que hi ha en execució en el sistema i que, per tant, ha de ser distingit de manera individual. Aquest identificador normalment és numèric i és únic durant tota la vida del SO.

(3)*PID* és l'acrònim del terme anglès *process identifier*.

2) L'estat del procés. Aquest camp indica l'estat del procés en el moment actual, dins d'unes possibilitats determinades (*run*, *ready*, *wait –blocked–*, etc.).

Vegeu també

Vegeu els estats d'un procés i el seu significat en el subapartat 1.3 d'aquest mòdul didàctic.

3) El comptador del programa. Aquest camp és fonamental perquè "assenyala" l'ordre o instrucció que estava a punt de ser executada just en el moment que es produeix una interrupció. Quan el procés pot continuar, ho fa exactament en aquest punt.

4) Els registres arquitectònics de la UCP. Són registres utilitzats per tots els processos. Per tant, després d'una interrupció no n'hi ha prou de continuar l'execució del procés en el punt on es va deixar, sinó que també s'ha de trobar un entorn idèntic al que tenia abans de produir-se la interrupció. Per tenir aquesta possibilitat també hem de desar el valor dels registres.

5) L'estat de la memòria. És difícil generalitzar sobre el que necessita cada sistema operatiu per a tenir tota la informació relativa a la memòria de cada procés, però podem donar algunes idees, com ara la quantitat de memòria assignada, el lloc on es troba, el tipus de gestió que se'n fa, les proteccions de cada part, les comparticions, etc.

6) Comptabilitat i estadístiques. Els sistema operatiu obté per a cada procés una sèrie d'informació relativa al comportament de cada usuari. Aquesta informació té un gran valor per als administradors de sistema, però no en té gaire per als usuaris o els programadors.

7) L'estat dels dispositius d'entrada/sortida. Els dispositius d'entrada/sortida assignats, les sol·licituds pendents, els fitxers oberts, etc., també formen part de l'entorn d'execució.

8) El domini de protecció. Aquest camp conté informació dels dominis als quals pertany el procés i dels drets que tenen associats.

Vegeu també

Podeu veure els dominis de protecció en el subapartat 4.1 del mòdul didàctic 5.

9) La planificació de la UCP. En aquest camp s'agrupa informació relativa a la manera com el procés accedeix al processador en concurrència amb els altres processos.

10) Altres informacions. Cada sistema operatiu manté altres informacions particulars en funció de diferents aspectes, com ara el fabricant del sistema operatiu, el tipus d'orientació[4], i també el tipus d'explotació[5].

(4)El treball en temps real, les comunicacions, etc.

(5)Ús privat, ús públic, etc.

D'acord amb els PCB, el sistema gestiona l'execució dels programes continguts a la memòria dels processos. En els subapartats següents analitzem els principals components d'aquesta gestió tenint en compte les necessitats d'un sistema multiprogramat.

1.2. L'execució concurrent

En la figura 1 podem veure l'execució concurrent d'un conjunt de processos (P1, P2 i P3) sobre un sistema monoprocessador multiprogramat. En funció de l'escala de temps amb què examinem l'evolució dels processos en el sistema en podem tenir visions diferents.

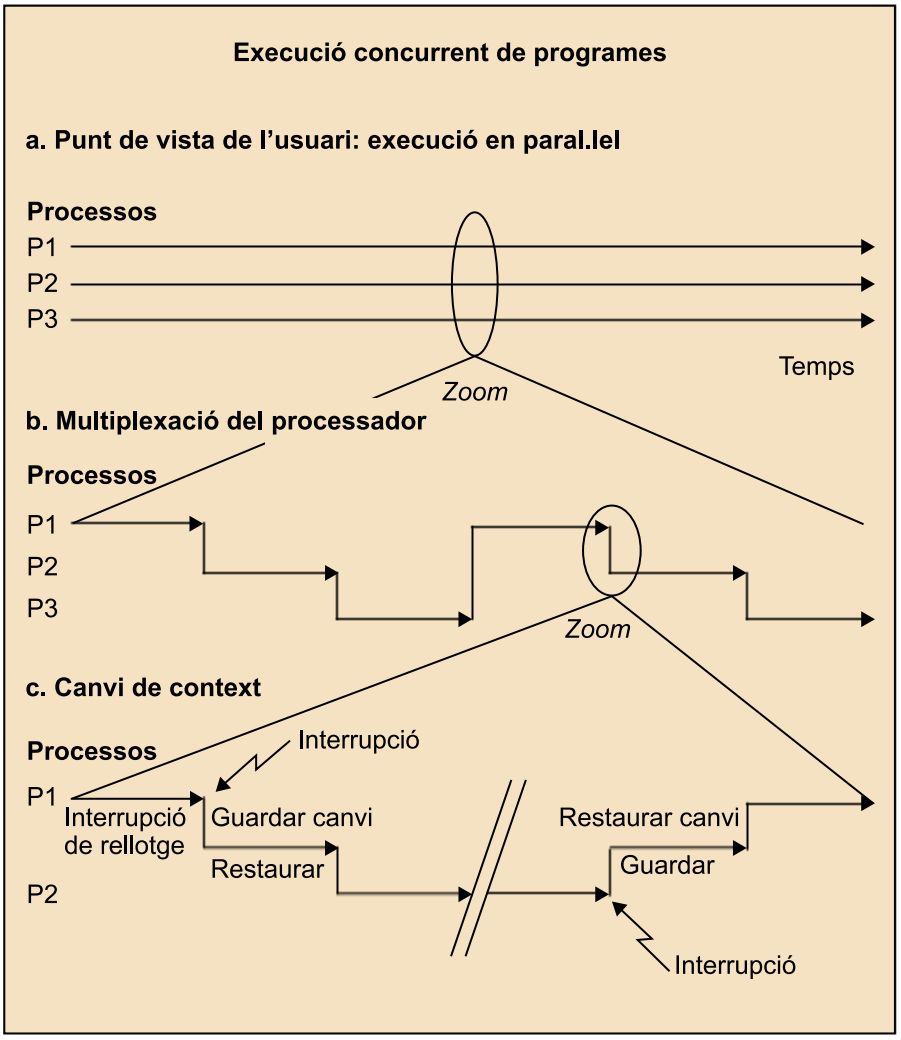


Figura 1

1) La primera escala de temps, que es veu a la figura 1a, correspon al punt de vista de l'usuari. Aquí l'execució dels tres processos sembla que s'efectua en paral·lel, i aparentment cada procés fa servir sempre el processador. Aquesta és la impressió que es vol donar en els sistemes multiusuaris: que cada usuari

té una bona interactivitat amb el sistema i que l'usuari es troba sol treballant davant la màquina. Però en realitat en un sistema que treballa en modalitat de temps compartit això no és cert.

2) Si mirem el comportament dels processos a una escala més petita de temps veiem que hi ha una multiplexació del processador en el temps (vegeu la figura 1b). Ampliant la figura 1a (fent un *zoom*), podem observar que no hi ha una execució real en paral·lel dels processos; només un està en execució real. Per a aconseguir l'efecte d'execució concurrent es fa una commutació entre els processos que es reparteixen el temps del processador. Aquestes commutacions s'anomenen **canvi de context**.

3) Finalment, ampliant més la imatge, és a dir, a una escala de temps encara més petita, podem veure el detall de les transicions entre processos o canvi de context (vegeu la figura 1c). Podem observar que ha d'aparèixer un fragment de codi nou per gestionar la interrupció que provoca el canvi. Per a fer-ho, s'han de dur a terme les operacions següents:

- Desar l'estat del processador tal com el tenia el procés P1 en l'instant que s'ha produït la interrupció sobre el seu PCB.
- Localitzar el PCB del procés P2.
- Restaurar l'estat del processador desat en el PCB del segon procés.

L'estat del processador en un instant concret és format pels valors continguts en cadascun dels registres en llenguatge màquina, pel punter a la pila, pel comptador de programa i, en general, per tota la informació que configura un procés. Els valors concrets d'aquest conjunt de registres en un instant de la vida d'un procés s'anomenen **context del procés**.

1.3. Els estats d'un procés

En un sistema multiprogramat, amb molts processos i un processador, com el que hem descrit en el subapartat anterior, en un moment determinat només hi pot haver un procés en execució; la resta de processos poden estar esperant el seu torn per a accedir al processador o poden estar esperant la finalització d'una operació d'entrada/sortida. Aquesta diversitat de situacions es pot representar amb un diagrama d'estats com el de la figura 2, en què els nodes representen els estats en què poden estar els processos, i els arcs són les accions que fan que un procés canviï d'estat.

Quan el sistema acaba de crear un procés, aquest procés es troba en l'estat inicial (1), l'**estat** *ready*. El sistema pot sortir d'aquest estat per les dues causes següents:

L'estat *ready*

Ready significa 'Ho tinc tot a punt i estic preparat per a rebre la CPU i treballar'.

1) Un esdeveniment extern al procés mateix, que pot ser degut a l'acció d'un altre procés mitjançant un senyal de programari, provoca la finalització del procés (2).

2) El sistema operatiu li assigna la UCP (3) i el procés comença a executar les seves ordres i passa a l'**estat** *run*.

De l'estat *run* es pot sortir pels tres motius següents:

1) El procés executa la darrera línia de codi i acaba (4).

2) El procés ha d'esperar un esdeveniment extern. L'exemple més normal és quan es demana una operació d'entrada/sortida[6] (5) i el procés espera que sigui servida. En aquest cas, el procés passa a l'**estat** *wait* (*blocked*) i s'espera fins que la petició s'hagi servit.

[6] Per exemple, una entrada d'informació pel teclat.

3) Quan el sistema operatiu treballa en la modalitat de temps compartit, si el procés supera la quota màxima de temps d'ús d'UCP que té assignada (6), deixa el processador i torna a l'estat *ready*.

Vegeu també

Vegeu l'execució de processos en la modalitat de temps compartit en el subapartat 2.3 del mòdul didàctic "Introducció als sistemes operatius".

Quan un procés surt de l'estat *run* per passar a *ready* o *wait* es produeix un canvi de context tal com hem esmentat en el subapartat anterior.

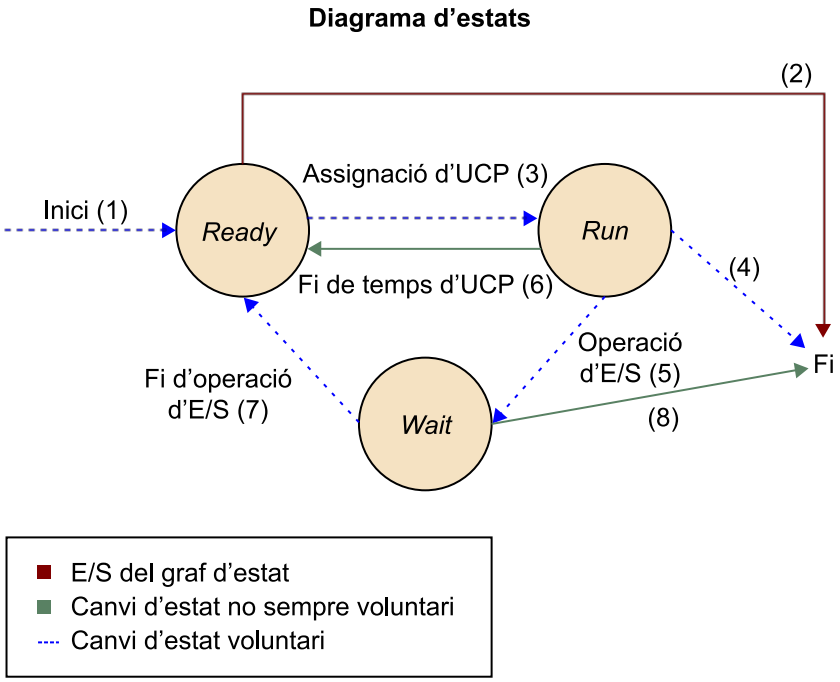


Figura 2

Finalment, els processos tenen dues destinacions possibles quan surten de l'estat *wait*:

1) Una cap a l'estat *ready*, quan finalitza l'operació per la qual esperaven (7). Dit d'una altra manera, la situació en què el procés té prou informació i pot continuar. Per exemple, si el procés estava pendent d'una operació d'entrada/sortida i ja ha arribat la informació esperada perquè l'usuari ha polsat una tecla.

2) Una altra, cap a la finalització del procés (8), deguda a un esdeveniment extern al procés mateix, com succeïa en l'estat *ready*.

Per saber què fa cadascun dels processos i així poder controlar els seus recursos, el sistema operatiu manté unes cues de processos en funció del seu estat. Així, el sistema té una cua de processos preparats (*ready*) i una de processos en estat d'espera (*wait*). No podem parlar d'una cua de processos en execució, ja que en entorns monoprocessador només hi ha un procés en execució. D'aquesta manera el procediment del nucli del SO encarregat de la planificació del processador pot examinar la llista de processos preparats a fi d'assignar el processador al procés que més convingui.

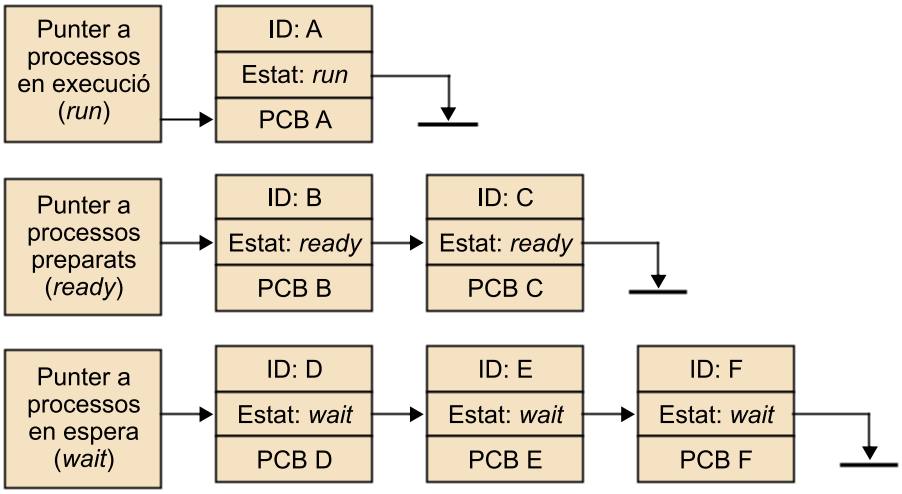


Figura 3

2. El cicle de vida d'un procés

Com hem vist en aquesta assignatura, els processos són un més dels objectes que gestiona el SO. A diferència de molts altres objectes, els processos són dinàmics i solen tenir un temps de vida limitat[7].

[7] Els processos es creen, interactuen amb el sistema i moren.

En aquest apartat analitzarem el cicle de vida dels processos i les operacions que s'hi relacionen.

Les **diferents etapes de la vida d'un procés** són les següents:

1) Creació, **naixement** o **inici**. En aquesta etapa s'assignen i s'inicialitzen els recursos necessaris per a crear un procés nou.

2) Desenvolupament. Un cop creats, els processos evolucionen a partir de l'execució del programa que contenen. Aquest desenvolupament els pot portar a modificar els recursos amb els quals s'han constituït inicialment.

3) Destrucció, **mort** o **finalització**. Un cop acabada la feina que especifica l'aplicació que s'executa en el marc del procés, el SO destrueix el procés i allibera els recursos que se li havien assignat.

En aquest apartat analitzarem el cicle de vida del procés. Per fer-ho estudiarem en primer lloc els processos de creació i destrucció d'un procés, a continuació les relacions que hi ha entre aquestes dues accions i, finalment, veurem algunes de les modificacions que es poden fer sobre l'entorn que constitueix un procés durant la seva existència.

2.1. La creació i la destrucció de processos

Els processos són elements dinàmics que es creen, operen durant un interval de temps i es destrueixen. El sistema operatiu és l'encarregat de proporcionar el conjunt de crides necessàries per a portar a terme totes aquestes accions. En aquest subapartat analitzarem les operacions de creació i destrucció de processos.

2.1.1. Creació de processos

La creació d'un procés nou és el resultat de l'execució d'una crida al sistema del tipus *crear_procés*, que és invocada, com totes les crides, per un procés ja existent.

L'execució de la crida *crear_procés* comporta la **creació d'un entorn d'execució** que conté els elements següents:

- La memòria on residiran el codi del programa, les dades amb què operarà el procés i la pila emprada per a passar paràmetres o desar variables locals a les subrutines.

- El punt d'entrada (adreça inicial) des d'on s'executarà el programa que contingui la memòria.

- L'entorn d'entrada/sortida amb el qual el procés es comunicarà amb l'exterior.

- Els atributs relacionats amb els dominis de protecció amb els quals el sistema operatiu verificarà la legalitat de les operacions que vulgui efectuar.

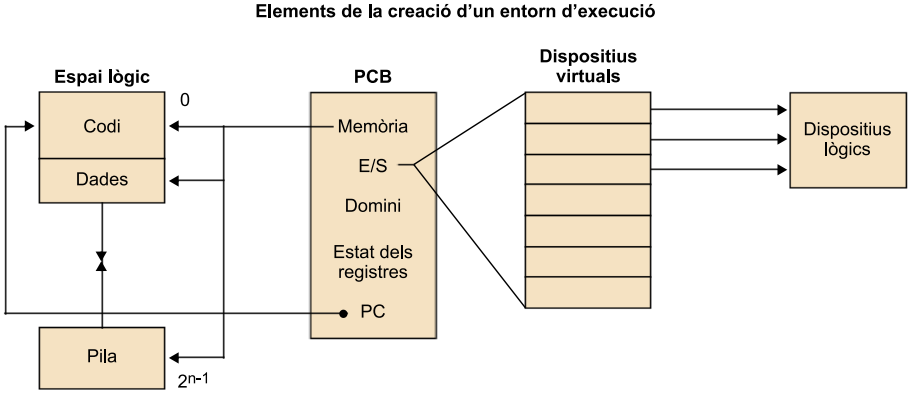


Figura 4

Cadascun d'aquest elements de l'entorn s'haurà d'especificar al sistema perquè aquest pugui crear un procés nou. L'especificació d'aquests elements es pot fer de les dues maneres següents:

a) De manera explícita, amb els paràmetres de la crida al sistema que crea el procés.

b) De manera implícita, fent que el sistema prengui uns valors per defecte.

Normalment, els sistemes combinen les dues alternatives: obliguen a especificar alguns d'aquests elements mitjançant els paràmetres de la crida, i en deixen uns altres amb valors per defecte.

Un altre aspecte que cal tenir en compte és la relació que hi ha entre l'entorn des d'on s'executa la crida al sistema que donarà lloc al nou procés i l'entorn nou que es crearà. Per a veure aquesta relació més clarament partim del fet, ja esmentat, que els processos són creats pel sistema operatiu a petició d'altres processos. Aquesta situació fa que els processos es puguin mirar des del punt de

Vegeu també

Vegeu l'herència entre processos en el subapartat 2.2 i la sincronització entre processos en el subapartat 2.3 d'aquest mòdul didàctic.

vista de la descendència, en què els processos tenen relacions de parentiu, com ara la de pare i fill. En aquest àmbit podem parlar dels conceptes d'**herència entre processos** i de **sincronització entre processos pare i fill**.

Un cop creat el procés, el sistema li dóna un nom (generalment un nombre dins d'un espai lineal) amb el qual podrà ser referenciat en accions de control i manipulació, tant des d'altres processos com directament des del SO. Aquest nom ha de ser únic per a cada procés, no tan sols durant la vida del procés al qual fa referència, sinó durant tota la vida del sistema.

Vegeu també

Vegeu els espais lineals en el subapartat 3.2 del mòdul didàctic 5.

La finalitat del fet de tenir un nom únic per a cada procés durant tota la vida del sistema és evitar confusions i funcionaments incorrectes del SO a causa de la reutilització de noms. Per exemple, imaginem dos processos (procés A i procés B) que col·laboren i se sincronitzen mitjançant senyals gràcies al fet que coneixen els seus identificadors. Si un (el procés B) acaba de manera imprevista i el seu identificador és reutilitzat per a crear un altre procés, aleshores el procés A s'està sincronitzant amb un procés que té un identificador B, que no és el procés amb el qual s'havia previst una comunicació. El resultat pot ser que cap dels dos processos no funcioni correctament o, el que és més problemàtic, que s'hagi obert un forat en la protecció del sistema.

Vegeu també

Vegeu els identificadors dels processos en el subapartat 1.1 del mòdul didàctic 7.

Així doncs, una possible estructura de la crida *crear_procés* podria ser:

Id_procés = crear_procés(*entorn_mem*,*nom_prog*,*punt_execució*, *entorn_E/S*, *entorn_domini*).

Cal comentar què en els sistemes en què el fill que es crea és una còpia del pare*, el valor de retorn de *crear_procés* ha de ser diferent en funció de si el procés és el pare o és el fill. Això permet diferenciar el comportament del pare i del fill, que s'estaran executant just després de retornar de la crida *crear_procés*.

Vegeu també

Vegeu els diferents valors retornats per a l'operació *crear_procés* segons el tipus de procés en el cas d'UNIX en l'apartat 4 d'aquest mòdul didàctic.

Pot resultat útil a un procés trobar informació sobre ell mateix. Per això hi ha una crida que retorna l'identificador del procés que la invoca. Aquesta crida podria ser:

Id_procés = quisoc()

2.1.2. Destrucció de processos

La destrucció d'un procés comporta la destrucció de l'entorn que el constitueix i l'alliberament dels recursos que tenia assignats.

La destrucció d'un procés per part del sistema pot ser conseqüència d'alguna de les situacions següents:

a) L'execució de la crida al sistema *destruir_procés* específica per a aquest motiu. En la majoria de sistemes aquesta crida provoca la destrucció del procés que la invoca, i no pot ser dirigida a d'altres processos.

b) El mal funcionament del procés destruït. Si el sistema operatiu detecta que un procés no funciona correctament i efectua operacions no permeses, el destrueix. Aquestes operacions solen estar associades a les excepcions provocades per accions com accedir a posicions de memòria que no es tenen assignades, executar ordres privilegiades o efectuar operacions aritmètiques incorrectes, com per exemple una divisió per zero, etc.

c) L'efecte lateral de l'execució, per part d'un altre procés[8], d'una crida al sistema diferent de la crida *destruir_procés*, que provoca una excepció sobre el procés que és destruït.

[8] El procés que la invoca ha de tenir dret per a provocar la destrucció de l'altre procés.

Vegeu també

Vegeu la destrucció de processos com a efecte derivat de l'execució d'altres processos o de les crides al sistema demanades per altres processos en el subapartat 3.2.1 del mòdul didàctic 7.

Ara ens centrarem exclusivament en el primer punt: la destrucció d'un procés com a resultat de l'execució de la crida al sistema *destruir_procés*. Les dues últimes situacions les veurem més endavant.

Tot procés, quan finalitza sense incidents l'execució del programa que emmagatzema, ha de ser destruït. Aquesta destrucció només pot ser el resultat de l'execució de la crida *destruir_procés*. Per tal que es porti a terme la destrucció, el compilador insereix automàticament, de manera transparent per al programador, la crida *destruir_procés* al sistema com a última ordre del programa. De manera addicional, el programador pot incloure invocacions a la crida *destruir_procés* per provocar la finalització del procés en situacions controlades pel programa.

La crida per a destruir un procés podria ser, doncs, la següent:

Estat = destruir_procés(*id_procés*).

Si aquesta crida funciona correctament no retornarà mai, ja que provocarà la destrucció del procés mateix que la invoca.

2.2. L'herència entre processos

L'herència entre processos és la relació que hi ha entre els diferents elements que configuren l'entorn del procés pare i els que configuren l'entorn del procés fill.

En concret, tenim els **tres tipus d'herència** següents:

1) Compartició: el procés pare i el procés fill comparteixen un mateix element. Per tant, les manipulacions que es facin d'aquest element afectaran tots dos processos de la mateixa manera.

2) Còpia: el SO crea els elements que configuren l'entorn del procés fill com una còpia dels elements del procés pare en el moment d'invocar l'operació de creació. A partir del moment en què el procés fill ha estat creat, els dos entorns tenen evolucions diferents.

3) Valors nous: en aquest cas el SO crea els elements del procés fill de nou i ho fa sense tenir en compte els del procés pare.

Cada element que configura l'entorn pot tenir una herència diferent, que pot estar predeterminada pel sistema o bé pot ser establerta mitjançant els paràmetres de la crida de creació al sistema. Tot seguit analitzem el que representen aquestes possibilitats pel que fa a cadascun dels elements de l'entorn:

1) La memòria i el seu contingut. La memòria, tal com hem vist anteriorment, es pot organitzar per segments. Per simplificar l'exposició ens centrarem en tres segments: el codi, les dades i la pila. Els atributs d'herència poden ser diferents per a cada segment. El codi i les dades poden tenir qualsevol dels tres atributs d'herència, mentre que la pila només pot tenir herència de còpia o de valors nous, però no compartida, ja que reflecteix l'estat de crides a procediments i a variables locals de cada procés. Això últim és degut al fet que la pila és necessària per a garantir que els dos processos (pare i fill) evolucionen de manera independent. Així doncs, les herències possibles per a la memòria són les següents:

Vegeu també

Vegeu la segmentació de la memòria en el subapartat 3.1 del mòdul didàctic 3.

a) Compartició: els processos fill i pare comparteixen el mateix segment de memòria física. Aquesta és la situació més convenient per a segments de codi i, en general, per a segments que continguin informació que només ha de ser llegida. En cas de compartició d'un segment de dades, qualsevol modificació que faci un dels dos processos alterarà l'estat de la memòria. Aquesta situació permet la col·laboració dels processos mitjançant la compartició d'informació.

Vegeu també

Podeu veure la problemàtica associada a la compartició de la informació en l'apartat 2 del mòdul didàctic 7.

b) Còpia: els segments del procés fill són una còpia exacta dels del pare a l'instant de la creació. Quan es crea un procés per duplicació del procés pare, s'han de copiar, com a mínim, tots els segments que durant l'execució independent del procés pare i del fill poden ser modificats. Aquests són els segments de dades i de pila. Un altre cas és quan es crea un procés per compartició, i pare i fill comparteixen codi i dades. En aquesta situació, el segment de pila no pot ser compartit i ha de ser copiat. El motiu, tal com s'ha exposat anteriorment, és que la pila reflecteix l'estat de les crides a procediments i a variables locals de cada procés que són fruit de l'execució independent de cada procés.

c) Valors nous: en aquest cas es carrega un fitxer executable que definirà de nou el contingut de la memòria.

2) El punt d'execució dintre de la memòria. El punt d'inici d'execució del programa depèn del contingut de la memòria, en concret del contingut del segment de codi. En cas de copiar o compartir el codi amb el procés pare, el punt d'execució pot ser, o bé el mateix que el del procés pare, o bé una adreça que s'introdueix com a paràmetre de l'operació de creació. En cas de carregar a la memòria un programa nou, el punt inicial d'execució ja és especificat en el fitxer executable.

Vegeu també

Vegeu la coincidència dels punts d'execució dels processos pare i fill en l'apartat 4 d'aquest mòdul didàctic.

Així doncs, les herències possibles són les següents:

a) Còpia: el punt d'execució del procés pare es copia en el fill. Aquest cas només és possible si es comparteixen o es copien els segments de codi i de dades. Per a poder distingir el procés pare del procés fill el més habitual és que els dos processos rebin valors de retorn diferents de la crida de creació al sistema.

b) Valors nous: el punt d'execució és definit o bé pels paràmetres de la crida al SO, o bé pel fitxer executable que definirà el contingut de la memòria.

L'entorn de compartició no és possible, ja que els processos pare i fill tenen execucions independents.

3) L'entorn d'entrada/sortida. L'herència de l'entorn d'entrada/sortida és independent de la de memòria. El concepte d'entorn d'entrada/sortida fa referència a les sessions d'accés a dispositius que el procés trobarà obertes de manera implícita.

L'entorn d'entrada/sortida pot tenir les tres modalitats d'herència següents:

a) Compartició: el procés pare comparteix amb el procés fill les sessions de treball amb els dispositius que tingui oberts en el moment de la creació. Les modificacions que faci sobre aquestes sessions de treball un dels dos processos també afectaran l'altre. Per exemple, en cas d'una sessió d'accés seqüencial, els dos processos compartiran un únic punter d'accés. Les sessions d'accés als dispositius que obrin a partir del moment de la creació seran independents en cada procés.

b) Còpia: el procés fill troba obertes les mateixes sessions d'accés als dispositius que tenia obertes el pare, però amb la diferència que les accions que s'efectuïn en aquestes sessions no afectaran les de l'altre.

c) Valors nous: en aquest tercer cas el procés fill troba obertes un conjunt de sessions d'accés als dispositius que s'especifiquen com a paràmetres de la crida al sistema.

4) El domini de protecció. El domini de protecció que hereta un procés és format pels atributs dels dominis als quals pertanyerà i pels seus drets associats. En el cas d'un sistema amb proteccions basades en *capabilities*, l'herència també inclou la llista de *capabilities* inicial del procés nou. Les *capabilities* tenen un comportament anàleg al dels dispositius virtuals.

Vegeu també

Vegeu les *capabilities* en el subapartat 4.4 del mòdul didàctic 5.

Per tant, en la descripció següent només farem referència als atributs de domini. Considerarem que aquests no poden ser copiats, de manera que el domini de protecció pot tenir les dues modalitats d'herència següents:

a) Compartició: el procés fill pertany al mateix domini que el procés pare o, el que generalment és el mateix, pertanyen al mateix usuari. Aquesta és la situació més habitual, i implica que les accions del procés fill són imputables al mateix usuari que les del procés pare.

b) Valors nous: a vegades, però, el procés fill necessita uns privilegis diferents dels que té associats l'usuari al qual pertany el procés pare. En aquest cas, i sota condicions controlades de protecció, es pot especificar un nou domini per al procés fill. El canvi de domini sol estar acompanyat d'un canvi de programa. Per a mantenir el sistema protegit, el programa nou ha d'haver estat escrit per l'usuari que configura el nou domini, o ser de la seva total confiança.

2.3. La sincronització i l'estat de finalització en la destrucció de processos

Fins ara hem analitzat el mecanisme de creació i destrucció de processos bàsicament des de l'òptica del procés creat, i no ens hem fixat en les accions que pot efectuar el procés pare com a conseqüència de la creació d'un fill. Els processos pare poden haver de sincronitzar l'execució amb la finalització dels processos que han creat i, al mateix temps, necessiten conèixer l'estat en què s'ha produït aquesta finalització.

Vegeu també

Vegeu l'estudi en profunditat i des d'un punt de vista més general del concepte de *sincronització* en el mòdul didàctic 7.

2.3.1. La sincronització

Un exemple en què apareix la necessitat de sincronització entre processos pare i fill el trobem quan explorem les possibles modalitats d'execució de les ordres: de primer pla, de fons i diferides. Si ens fixem en les dues primeres modalitats es pot identificar un procés pare, que és el que executa l'intèrpret d'ordres, i uns processos fill, que són els que executen les ordres. Així doncs, podem executar les ordres de les tres maneres següents:

1) En la **modalitat de primer pla** (*foreground*) l'intèrpret d'ordres espera que finalitzi l'ordre abans de demanar-ne una de nova per executar. En aquesta situació, el procés pare ha d'esperar que el procés fill finalitzi i per aconseguir-ho el SO ha de proporcionar eines per congelar l'execució del procés pare fins que el procés fill sigui destruït.

2) En la **modalitat d'execució de fons** (*background*) l'intèrpret d'ordres no espera que finalitzi l'ordre, sinó que immediatament després d'haver creat el procés que executarà l'ordre continua endavant i demana una ordre nova. En aquesta altra situació, el procés pare només necessita que el sistema el retingui mentre crea el nou procés a fi de retornar-li l'identificador del procés si la creació s'ha executat correctament o, en cas contrari, retornar-li un error.

3) Una tercera possibilitat és la **modalitat mixta**, que consisteix en una combinació de les dues modalitats anteriors. Aleshores un procés pare crea un procés fill en modalitat de fons i, a partir d'un cert instant de l'execució, decideix esperar que finalitzi un dels seus processos fill. El SO ha de proporcionar crides específiques de sincronització.

La figura següent mostra els tres models d'execució:

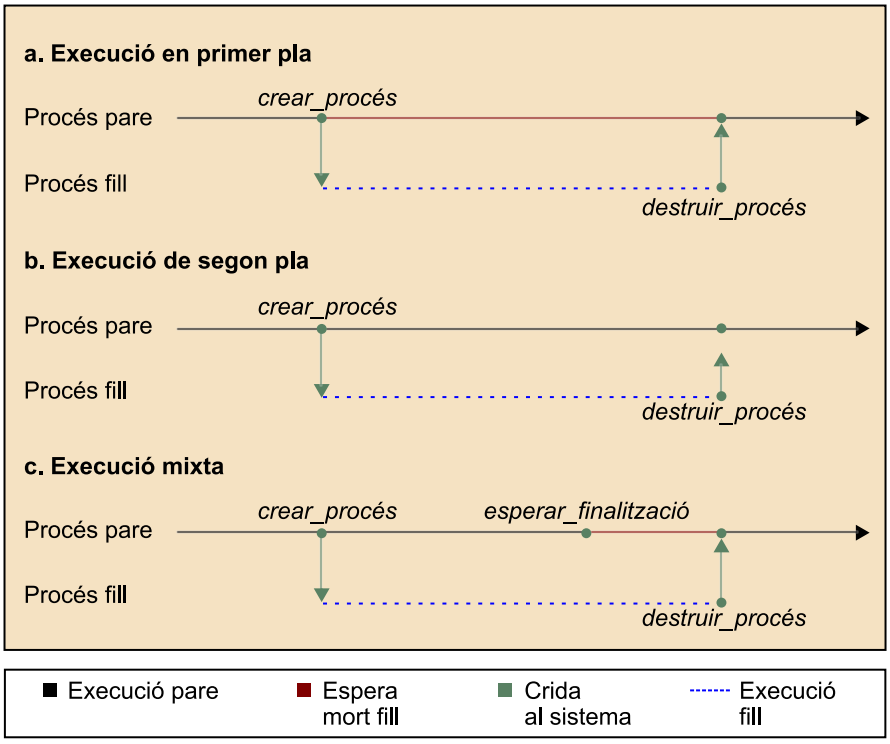


Figura 5

Com a conseqüència de la sincronització entre els processos pare i fill, el sistema pot oferir els models de crides al sistema següents:

a) Introduir un paràmetre, *mode_execució*, que indiqui si el procés pare ha d'esperar la destrucció del fill o no.

- Id_procés = crear_procés(*mode_execució*,*entorn_mem*,*nom_prog*, *punt_execució*,*entorn_E/S*,*entorn_domini*).

- Estat = destruir_procés(*id_procés*).

b) Fer que per a crear no calgui esperar mai la finalització d'un procés i es pugui proporcionar una crida específica que permeti dur a terme aquesta funció.

- Id_procés = crear_procés(*entorn_mem*,*nom_prog*,*punt_execució*, *entorn_E/S*,*entorn_domini*).
- Estat = destruir_procés(*id_procés*).
- Estat = esperar_finalització(*id_procés*).

El paràmetre de la crida *esperar_finalització* pot ser d'entrada o de sortida. En el primer cas serveix per a indicar el procés concret amb el qual es vol sincronitzar. En el segon cas pot servir per a esperar qualsevol dels fills. La crida esperarà fins que algun dels fills del procés que la invoca hagi acabat[9]. En el moment que un procés fill hagi acabat la crida al sistema retornarà el control al procés pare, que rebrà l'identificador del procés fill que ha acabat i el seu estat de finalització.

[9] En alguns sistemes també s'inclou el cas en què el procés fill, tot i estar encara viu, ha estat aturat per algun motiu.

2.3.2. L'estat de finalització dels processos

Tal com hem avançat, per al procés pare pot ser interessant conèixer el punt en què l'aplicació ha finalitzat o el motiu pel qual ha finalitzat. Amb aquesta finalitat, la crida *destruir_procés* sol tenir un paràmetre que el programador pot utilitzar per a notificar al procés pare el motiu de la finalització o, el que és el mateix, el punt de l'algorisme on s'ha decidit finalitzar el procés.

La crida *destruir_procés* quedaria de la manera següent:

Estat = destruir_procés(*id_procés*,*estat_finalització*).

A més a més, tal com hem vist anteriorment, l'execució d'un procés pot finalitzar com a conseqüència d'accions no previstes pel programador: per errors d'execució o per efectes laterals de l'execució d'altres crides al sistema. En aquests casos el sistema operatiu és l'encarregat de codificar els motius de la finalització a fi de notificar-ho al procés pare.

El sistema operatiu ha de proporcionar una crida que permeti als processos pare recuperar els paràmetres de finalització dels seus processos fill. Aquesta crida sol ser la mateixa que els permet sincronitzar-se amb la finalització del procés fill: *esperar_finalització*.

2.4. Els canvis en l'entorn d'execució

Un cop ha estat creat un procés, el SO inicia l'execució del codi que conté aquest procés. De resultes d'aquesta execució, l'entorn pot evolucionar i canviar, i els canvis poden afectar qualsevol dels elements que configuren el pro-

cés: la memòria, el contingut de la memòria, l'entorn d'entrada/sortida o el domini de protecció. Per a tots, el SO ha de proporcionar crides que els permetin modificar.

En aquesta assignatura ja hem estudiat les crides que fan referència als dispositius, als fitxers i a la protecció. En aquest subapartat ens centrarem especialment en aquelles que modifiquen el contingut i l'estructura de l'espai de memòria en el marc del canvi d'imatge, i aprofundirem en aquelles altres que canvien l'entorn de les entrades/sortides en el marc dels reencaminaments o, en altres paraules, en el marc de l'assignació implícita de dispositius virtuals a lògics.

Vegeu també

Podeu trobar les crides al sistema relacionades amb els dispositius, els fitxers i la protecció en els mòduls 4 i 5.

2.4.1. Canvi d'imatge

El sistemes operatius han de permetre que els processos carreguin nous programes a la memòria a fi de ser executats. Aquesta acció es pot fer de les dues maneres següents:

a) Al mateix temps que es crea un nou procés. Per exemple, en el sistema operatiu Windows es canvia la imatge en el moment en què es crea un nou procés.

b) *A posteriori*, un cop creat el procés, mitjançant la invocació d'una crida específica (*carregar_imatge*). Per exemple, en el sistema operatiu UNIX només es canvia la imatge si s'invoca explícitament una crida al sistema per a carregar una nova imatge. En el moment en què es crea un nou procés no es pot modificar la imatge, sinó que el sistema fa una còpia de la imatge del procés pare.

En funció del SO, s'oferiran combinacions diferents d'aquestes dues possibilitats. Aquí enfocarem l'explicació partint d'un SO que només ofereix la segona possibilitat. Considerarem, doncs, que els processos nous inicialment sempre executen el mateix codi que el procés pare, ja sigui perquè el comparteixen, ja sigui perquè s'ha copiat. *A posteriori*, un cop iniciada l'execució del nou procés, aquest decidirà si ha de carregar un nou codi o no.

Aquest és el cas del funcionament de l'intèrpret d'ordres, que, com veurem més endavant, primerament obtindria per l'entrada estàndard l'ordre que hauria de portar a terme, analitzaria si hi havia algun executable que ho pogués fer i, en aquest cas, crearia un nou procés igual. Aquest nou procés, en executar-se, carregaria l'aplicació que dóna servei a l'ordre rebuda.

Vegeu també

Vegeu el funcionament de l'intèrpret d'ordres en el subapartat 4.2 d'aquest mòdul didàctic.

La càrrega d'un nou executable provoca la reconfiguració total de l'espai lògic del procés sobre el qual s'efectua. Per tant, tots els valors de variables i constants, els procediments i les funcions que es trobaven dintre de l'espai lògic del procés abans de la càrrega, desapareixen. A fi que el codi carregat pugui utilitzar informació elaborada pel codi anterior a la càrrega, ha d'utilitzar mecanismes o dispositius d'emmagatzematge que serveixin de pont entre tots

dos. Una manera senzilla de fer-ho és utilitzar el sistema de fitxers o, en general, un dispositiu d'emmagatzematge. No obstant això, els SO ofereixen la possibilitat de passar informació a través seu en forma de paràmetres de la crida *carregar_imatge*, els quals són rebuts pel nou programa com a paràmetres d'entrada de la funció principal.

El pas de codi a C

Si s'utilitza el llenguatge C, el nou programa rebria la informació del codi anterior a la càrrega com a paràmetres de la funció `main`:

```
main(argc,argv,env)
int argc;
char **argv,**env;
```

Hem de fer notar que la crida *carregar_imatge* només afecta l'estructura i el contingut de la memòria, i també el comptador de programa, que ens indica la propera ordre que cal executar. La resta d'elements que configuren l'entorn del procés no s'han de veure necessàriament afectats, de manera que el nou executable ha de trobar el mateix entorn d'entrada/sortida i el mateix entorn de protecció. Això no obstant, aquesta afirmació pot variar en funció de si el SO assigna altres funcions a la crida *carregar_imatge*.

Vegeu també

Vegeu les funcions de la crida *carregar_imatge* en el cas d'UNIX en el subapartat 4.4 d'aquest mòdul didàctic.

La crida *carregar_imatge* podria tenir la forma següent:

Estat = carregar_imatge(*nom_executable*,*paràmetres*).

Aquesta crida retorna un valor només en cas d'error, ja que si té èxit tot el codi i les dades del programa que la contenien hauran desaparegut de la memòria.

2.4.2. Reencaminaments d'entrada/sortida

La manera d'introduir modificacions en general en l'entorn d'entrada/sortida d'un procés consisteix a obrir i tancar sessions d'accés als dispositius. En aquest subapartat ens fixarem en el cas concret de com es poden fer els reencaminaments de les entrades/sortides. Com hem vist, els reencaminaments de les entrades/sortides es basen en l'assignació de dispositius virtuals estàndard a dispositius lògics. Per a aconseguir la independència i la portabilitat de les aplicacions, aquesta assignació s'ha de fer amb anterioritat a l'execució dels programes que configuren les aplicacions i, per consegüent, de manera transparent per a aquests. Aquest fet és el que hem anomenat **assignació implícita dels dispositius virtuals**.

Vegeu també

Podeu consultar les sessions d'accés a l'entorn d'entrada/sortida en el subapartat 5.2 del mòdul didàctic 4.

El sistema pot fer aquesta assignació bàsicament de les tres maneres següents:

a) El procés fill pot heretar del procés pare els dispositius virtuals que tingui oberts.

b) El procés pare especifica en els paràmetres de la crida *crear_procés* quins dispositius lògics s'han d'assignar als dispositius virtuals del fill.

c) El procés fill modifica el seu entorn d'entrada/sortida un cop creat i abans de carregar un programa nou.

L'última d'aquestes opcions és la que ens interessa en aquest subapartat.

3. Fluxos d'execució

Un **flux** o **fil d'execució** (***thread***) és la mínima unitat de planificació del sistema operatiu.

Un flux forma part d'un procés i gaudeix dels recursos assignats al procés. Un procés té com a mínim un flux, però en els sistemes operatius actuals un procés pot tenir més d'un flux d'execució.

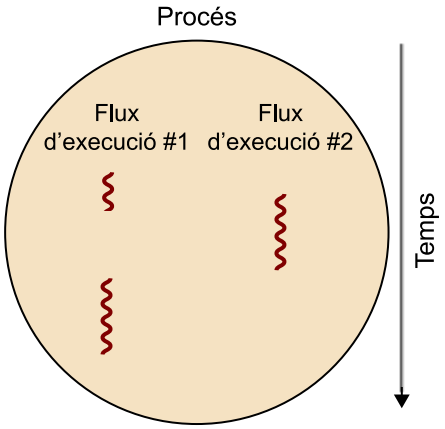


Figura 7

Els diferents fils que formen part d'un mateix procés comparteixen la majoria dels recursos del procés, com l'espai de memòria, els arxius oberts, els permisos, el directori de treball, l'identificador de procés, etc. En canvi, cada fil té la seva pila d'execució pròpia, pot estar executant diferents ordres (cada fil té el seu registre comptador propi de programa o PC), s'executa a una determinada velocitat i té el seu estat d'execució propi. Per tant, tots els fluxos d'un procés disposen del mateix codi i dades, però en un moment determinat poden estar executant parts diferents del codi o treballar amb dades diferents.

Hi pot haver diversos motius pels qual sigui interessant dissenyar aplicacions amb diversos fluxos (*multithreaded*):

- Programació més modular, encapsulant tasques.
- Explotar el paral·lelisme disponible a les màquines amb memòria compartida per més d'un processador (*multicore* o *multiprocessor*).
- Fer entrada/sortida paral·lela, dedicant fluxos a fer l'E/S.
- Fer servidors concurrents, fent que cada flux atengui una petició de servei.

Cal notar que si disposem de diversos processadors, els fluxos es podran executar simultàniament. Si, en canvi, només disposem d'un processador o, en general, tenim menys processadors que fluxos, llavors cal repartir el temps del processador entre els diferents fluxos, multiplexant l'ús del processador en el temps.

Tot i que els fluxos comparteixen tots els recursos del procés, és important que hi hagi alguna informació pròpia de cada flux per a garantir un funcionament correcte. Per exemple:

1) Podem estar interessats a fer una acció determinada sobre un flux concret.

2) Cada flux pot estar executant una part del codi diferent.

3) Encara que cridin una mateixa rutina, cada un necessitarà desar els paràmetres i les variables locals en una zona de memòria diferent que faci les funcions de pila.

4) Quan es produeixin canvis de context entre fluxos cal garantir que es conserva l'estat del flux per a poder continuar amb l'execució posteriorment.

5) Cal diferenciar les condicions d'error produïdes per crides a sistema fetes per diferents fluxos dintre d'un mateix procés.

6) Opcionalment, és interessant controlar la planificació dels fluxos, per exemple prioritzant un flux sobre altres.

Per tot això cada flux té associat:

1) Un identificador.

2) Un punter a l'ordre següent per executar.

3) Un punter al cim de la pila.

4) L'estat dels registres del processador.

5) Hi pot haver informació local a un flux. Això pot ser útil en determinades circumstàncies. Un exemple és la definició de la variable `errno` quan una crida a sistema ha produït un error.

6) Hi pot haver informació de planificació específica per cada flux, usada pel planificador de fluxos.

Els fluxos d'un mateix procés comparteixen la majoria dels recursos. Per això, el canvi de context entre fluxos d'un mateix procés és menys costós que el canvi de context entre fluxos de processos diferents.

Els fils d'execució també són coneguts com a **processos lleugers**, pel fet que consumeixen menys recursos de sistema que els processos.

Possibles utilitats dels fluxos

- Aplicacions gràfiques: un flux s'encarrega de la gestió de la interfície gràfica d'usuari mentre un altre fa les operacions de càlcul.
- Aplicacions client/servidor: el servidor crea múltiples fluxos per donar servei a múltiples clients simultàniament.

4. Exemples de gestió de processos

4.1. La gestió de processos en UNIX

En UNIX un procés és un entorn d'execució identificat per un número anomenat **PID**, que és l'**identificador del procés** i és únic durant tota la vida del SO.

Els **principals elements que constitueixen un procés en UNIX** són:

1) L'**espai de memòria**, format per tres segments lògics: un de codi[10], un de dades i un de pila. El segment de codi pot ser compartit per altres processos. Entre el segment de dades i el de pila hi ha una porció d'espai lògic no assignada al procés. Quan convé, el procés pot augmentar el segment de dades amb la rutina de biblioteca `malloc` (vista al mòdul 3).

(10)El segment de codi també s'anomena *segment de text*.

2) L'**entorn d'entrada/sortida**, que és format per una taula de *file descriptors* (dispositius virtuals), i és local a cada procés. Cada entrada d'aquesta taula apunta a una altra taula, que en aquest cas és global per al sistema, i conté els punters de lectura seqüencial, el mode d'accés i un punter a l'estructura del SO que gestiona el dispositiu lògic associat a un *file descriptor*. Finalment, l'entorn d'entrada/sortida es complementa amb la informació de quin és el directori de treball actual.

Vegeu també

Vegeu els *file descriptors* en el subapartat 7.1 del mòdul didàctic 4.

3) L'**UID** i el **GID**, que corresponen al **número d'identificador de l'usuari** i al **número d'identificador del grup de l'usuari**, i fan referència a l'usuari i al grup a qui pertanyen els processos dintre del domini de protecció.

UID i GID

Són, respectivament, els acrònims dels termes anglesos *user identifier* i *group identifier*.

4) L'**estat dels registres del processador**, que reflecteix en quin estat es troba l'execució del programa que conté el procés.

5) La **informació estadística**, que recull informacions com ara el temps consumit d'UCP, el volum consumit de memòria o el nombre de processos fill generats.

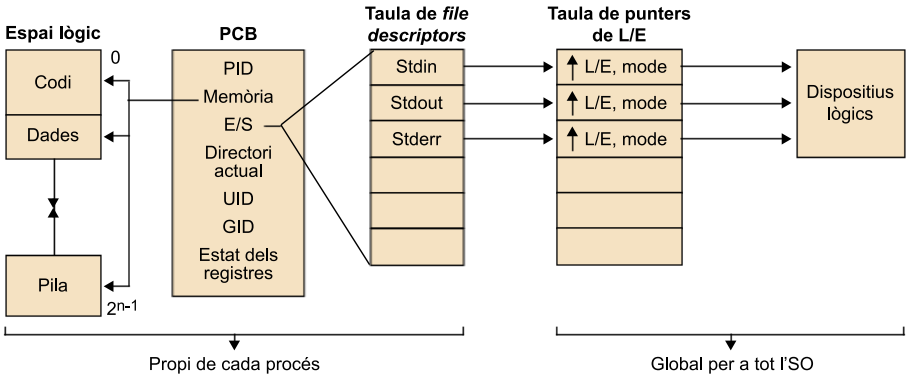


Figura 8

4.1.1. La creació i la destrucció de processos

Les correspondències entre les crides d'UNIX i les vistes en aquest mòdul són:

Operació	Crides d'UNIX
Crear_procés	`fork`
Destruir_procés	`exit`
Esperar_finalització	`wait`
Carregar_imatge	`exec (execl, execlp, execle, execv, execvp, execve)`
Quisoc	`getpid`

UNIX ha optat per tractar el tema de la creació i la destrucció de processos de la manera més senzilla possible. Les crides que ofereix tenen el mínim nombre possible de paràmetres, i creen i destrueixen els processos a partir d'una definició implícita en el sistema. *A posteriori*, i definint-lo per programa, l'usuari pot canviar l'entorn d'execució i fer el procés a la seva mida. A continuació veurem com actua cadascuna de les crides vistes en aquest mòdul en el sistema operatiu UNIX:

Creació i destrucció de processos

En UNIX, la creació i destrucció de processos s'ajusta a la filosofia general del sistema d'oferir eines d'utilització senzilla que permetin desenvolupar qualsevol política que es necessiti.

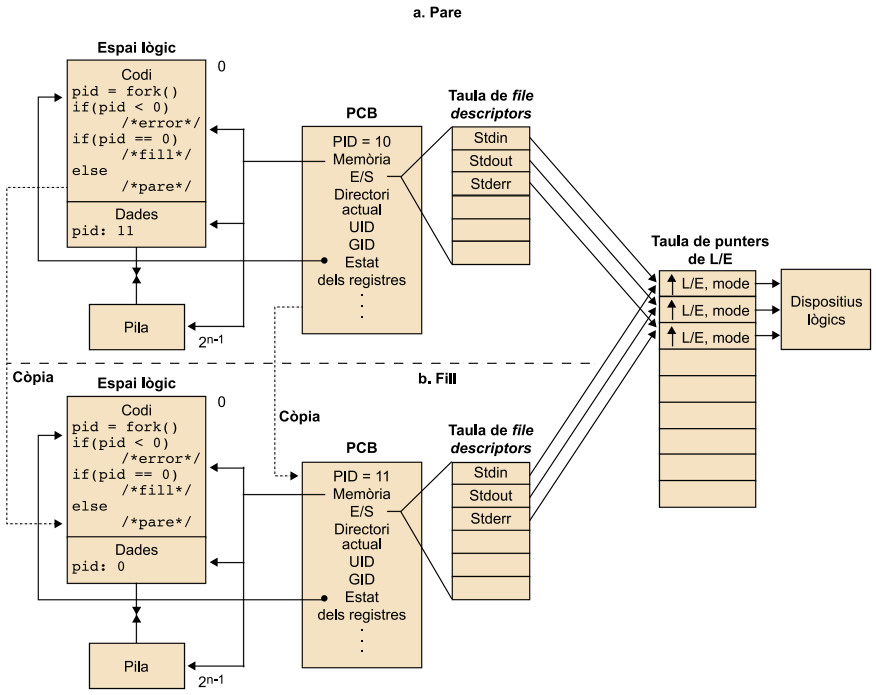


Figura 9

1) La **crida `fork`** no té cap paràmetre i crea un nou procés fill que és un duplicat del procés pare. En general, el PCB del fill és una còpia del PCB del pare; l'espai de memòria del procés fill és una còpia de l'espai del pare i les entrades de la taula de *file descriptors* apunten a les mateixes entrades de la taula de punters de lectura/escriptura. El procés fill pertany al mateix usuari i grup d'usuaris que el pare. Tots dos processos continuen l'execució de manera independent en el punt immediatament posterior a la crida `fork` al sistema.

Normalment després de fer la crida `fork` el procés fill ha d'executar un codi diferent del codi del pare. Per poder distingir-se, la crida `fork` retorna valors diferents en funció de si és el procés pare o el fill: el fill rep un valor 0 i el pare rep l'identificador (PID) del fill.

Durant la creació d'un nou procés, UNIX controla els aspectes importants que afecten el conjunt del sistema, com ara la disponibilitat de recursos[11]. Un exemple concret d'aquest control és el fet que no permet que un usuari normal (no processos de sistema) ocupi l'última entrada de la taula de processos, l'última àrea de memòria lliure, etc. Si això passés, el sistema potser no es podria recuperar de situacions en les quals seria necessari posar en marxa algun procés del sistema de manera urgent. Per exemple, si s'hagués de fer una parada ràpida del sistema, podria ser catastròfic no poder engegar el procés de parada del sistema (*shutdown*).

(11)Els recursos del sistema són la memòria, les entrades en la taula de PCB, etc.

2) La **crida `exit`** destrueix un procés. La destrucció d'un procés mitjançant aquesta crida és idèntica a la que hem explicat en el cas de la creació i la destrucció de processos. El procés que l'executa és destruït pel sistema. A fi de notificar l'estat de finalització es passa al sistema operatiu un paràmetre que el procés pare podrà recollir mitjançant la crida `wait`.

Vegeu també

Vegeu la destrucció de processos en el subapartat 2.1 d'aquest mòdul didàctic.

3) La **crida `wait`** bloqueja[12] el procés que l'ha cridat fins que algun dels seus processos fill hagi estat destruït. Quan això passa, el sistema li retorna el PID del procés destruït i, com a paràmetre de sortida, l'estat en què ha finalitzat. Combinant les tres crides presentades en aquest subapartat, l'intèrpret d'ordres pot executar ordres en les modalitats de primer pla i de fons (vegeu la figura 10).

(12)Hi ha altres variants, com `waitpid` i `waitid`, que permeten un control més fi sobre el procés que es vol esperar o els tipus de canvis d'estat que es volen monitorar.

UNIX desa l'estat de finalització dels processos destruïts en el seu PCB i espera que el seu procés pare el reculli mitjançant la crida `wait`. Per aquest motiu, UNIX no allibera el PCB d'un procés en el moment de la destrucció[13].

(13)Quan es destrueix un procés s'alliberen tots els recursos excepte el PCB: s'allibera memòria, es tanquen els canals d'entrada/sortida, etc.

Els processos esperen en un estat especial anomenat *zombie* fins que s'executa la crida `wait` que els permetrà eliminar definitivament. Per a evitar l'acumulació de processos en estat *zombie* s'han previst els dos mecanismes següents:

1) Cada usuari només pot tenir en un instant determinat un cert nombre de processos actius, inclosos els *zombies*.

2) Els processos que són destruïts més tard que el seus processos pare passen a ser considerats fills del primer procés del sistema[14], que és l'encarregat de recollir el seu estat de finalització.

(14)El primer procés del sistema s'anomena *init* i té un PID igual a 1.

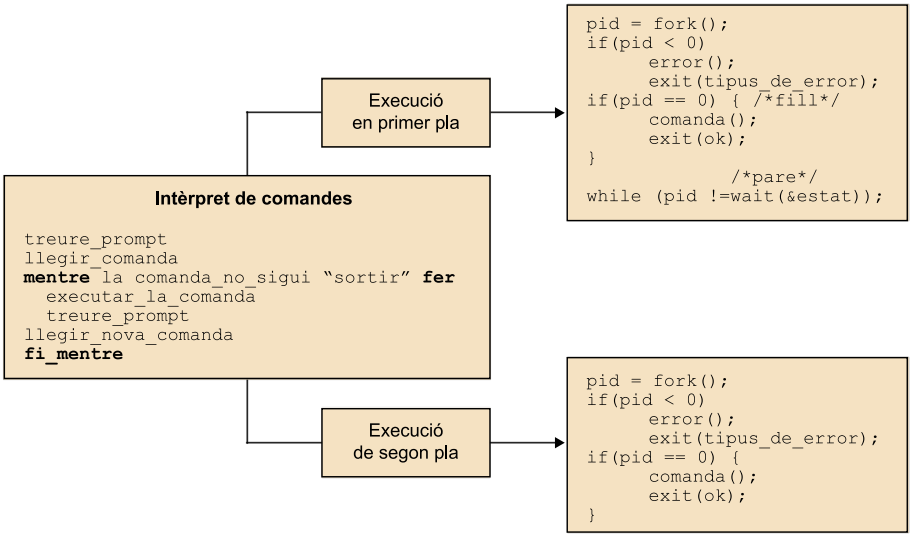


Figura 10

4.1.2. Els canvis de l'entorn que configura un procés

En UNIX hem vist que qualsevol canvi que es vulgui fer en un procés s'ha de portar a terme un cop s'hagi creat. El canvi d'executable i els canvis en les entrades/sortides fets *a posteriori* de la crida `fork` permeten controlar per programa i, de manera flexible, tenir accés a un ventall molt ampli de possibilitats que difícilment es podrien haver previst *a priori* des del SO. Per il·lustrar aquest fet ens fixarem en l'intèrpret d'ordres d'UNIX (*shell*) i en els reencaminaments que permet. Abans, però, analitzarem els puntals d'aquesta flexibilitat, que són els elements següents:

- La **crida `fork`**, que ja hem vist.
- La **crida `exec`** de canvi d'executable.
- La **crida `dup`** d'entrada/sortida, que permet manipular els *file descriptors*.

Tot seguit veurem com els dos últims elements ens permeten modificar l'entorn que configura un procés.

1) El canvi d'executable

La crida `exec` al sistema d'UNIX permet canviar la imatge o l'executable que conté un procés. De fet, hi ha tota una família de funcions que se solen referenciar sota el nom `exec`, que són diferents maneres d'invocar la crida a sistema `execve`. En concret, les funcions són: `execl`, `execlp`, `execle`, `execv`, `execvp`.

El canvi que té lloc mitjançant la crida `exec` no afecta els elements que configuren el procés, com ara l'entorn d'entrada/sortida. En general només afecta la programació dels senyals i el domini de protecció si el fitxer executable que s'ha de carregar té actiu el bit *setuid* o el *setgid*. Els bits *setuid* i *setgid* fan que durant l'execució del programa, el procés que l'ha carregat canviï respectivament al domini de l'usuari propietari, o al del grup de l'usuari propietari del fitxer executable. Aquests drets permeten construir aplicacions que accedeixen de manera controlada a bases de dades o a fitxers en general sobre els quals no es vol donar un dret d'escriptura generalitzat.

Vegeu també

Vegeu la programació dels senyals en el subapartat 3.2 del mòdul didàctic 7 i el canvi de domini d'usuari per part del procés en el subapartat 5.1 del mòdul didàctic 5.

2) La manipulació dels *file descriptors*

La crida `dup` permet duplicar el valor d'una entrada concreta de la taula de *file descriptors* a la primera posició lliure que es trobi dintre de la taula.

Amb aquesta operació es poden modificar fàcilment els valors associats als *file descriptors* estàndards.

Vegem un exemple d'ús de la crida `dup` en combinació amb la crida `exec`:

```
int estat, descFitxer, pid;
...
pid = fork();
switch(pid) {
case 0:     /* Codi del fill. */
   descFitxer = open("fitxer",O_RDONLY);
   if (descFitxer == -1){
   /*tractament de l'error*/
   }
   ...
   close(0);
   dup(descFitxer);
   close(descFitxer);
   ...
   execl("/bin/comanda","comanda",0);
   ...
   /*tractament de l'error*/
   ...
case -1:    /* La crida fork ha fallat. */
   ...
   /*tractament de l'error*/
   ...
default:    /* Codi del pare. */
   while(pid != wait(&estat));
}
...
```

El codi anterior podria ser el que executa l'intèrpret d'ordres donada l'ordre següent: `$ comanda < fitxer`.

En aquest cas només es mostra el tall de codi que, un cop llegida l'ordre des del terminal, l'executa. Aquesta execució es fa en la modalitat de primer pla. Ara veurem els diferents passos d'aquesta execució.

En primer lloc l'intèrpret crea el procés fill que haurà d'executar l'ordre. Un cop creat s'ha de fer el reencaminament de l'entrada estàndard a fi que quan s'hagi carregat el programa *ordre* es trobi el reencaminament fet. Per a fer-ho es crea un *file descriptor* vinculat a `fitxer` amb la crida `open (descFitxer)`. Amb `open` s'ocupa la primera entrada lliure de la taula de *file descriptors*.

Les figures següents il·lustren l'evolució de les taules internes del sistema durant aquest procés:

a) Situació després d'haver obert `fitxer`:

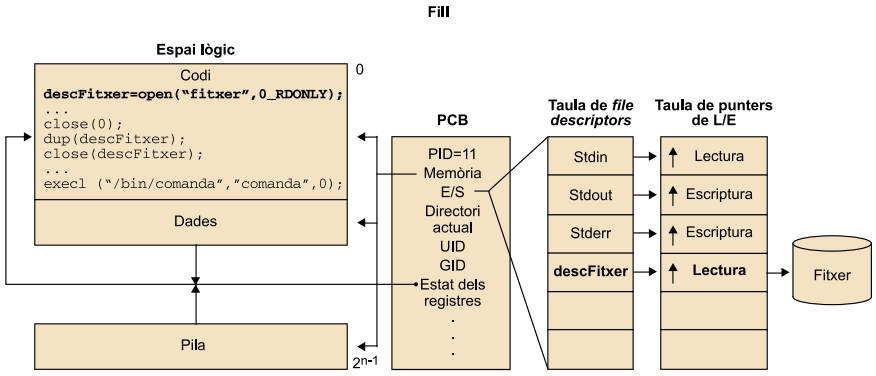


Figura 11

Per a reencaminar l'entrada estàndard des de `fitxer` s'ha de fer que el *file descriptor* 0 estigui associat al fitxer. Per fer-ho, desassignem el dispositiu lògic associat al *file descriptor* 0 i, mitjançant la crida `dup`, el tornem a assignar copiant sobre la seva entrada en la taula de *file descriptors* l'entrada associada a `descFitxer`. Així, els dos *file descriptors* fan referència a la mateixa sessió de treball oberta sobre `fitxer`.

b) Situació després d'haver executat la crida `dup`:

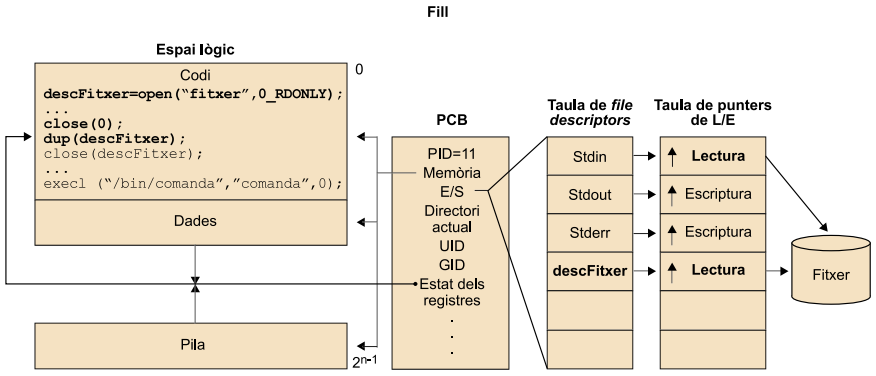


Figura 12

Falta eliminar, mitjançant la crida `close`, el *file descriptor* `descFitxer` per a aconseguir l'entorn d'entrada/sortida que ha de trobar l'ordre que cal executar.

Després, el procés fill pot carregar el nou executable mitjançant la crida `exec`[15].

[15] En l'exemple s'utilitza `execl`, que és una de les diferents formes d'aquesta crida.

c) Situació després d'haver carregat l'executable de l'ordre:

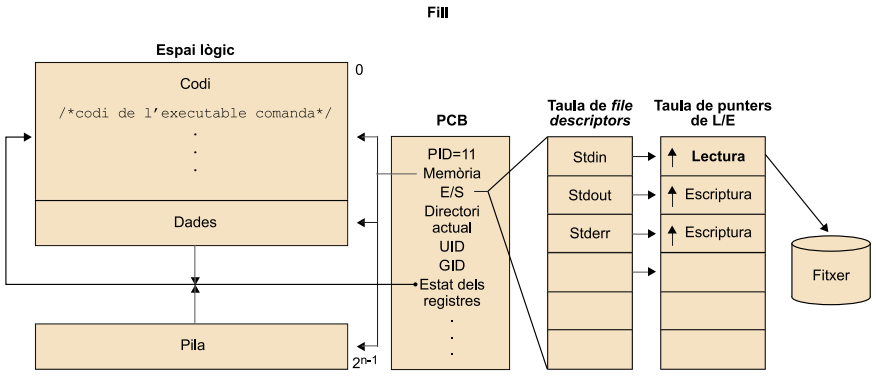


Figura 13

4.1.3. La jerarquia de processos en UNIX

El **procés** *init* amb PID igual a 1 és el primer procés que es crea en un sistema UNIX, i és l'antecessor de tots els processos que es crearan a continuació. *Init* és l'únic procés que no és creat amb la crida `fork` al sistema, ja que el crea directament el nucli durant la inicialització del sistema.

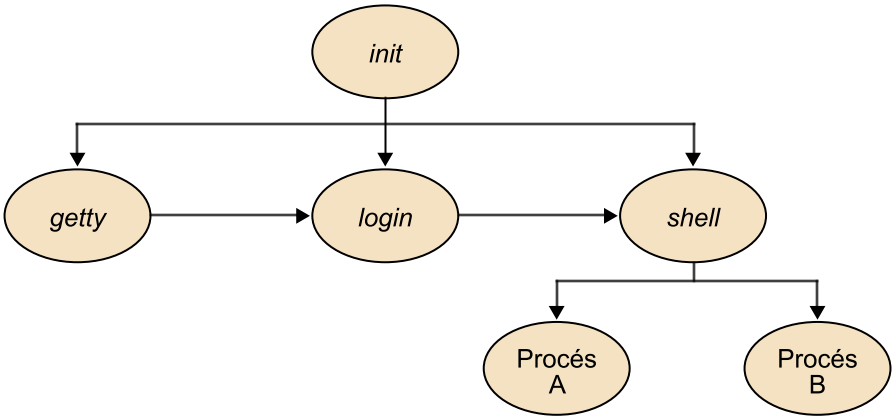


Figura 14

Les principals funcions del procés *init* són les d'inicialitzar tots els processos de sistema, com ara els processos servidors de xarxa, engegar un procés *getty* per a cada terminal del sistema i, finalment, alliberar els processos *zombie* que ja no tenen el procés pare.

Vegeu també

Als recursos de l'Aula disposeu de material addicional amb exemples de programes UNIX que fan servir aquestes crides al sistema.

El **procés** *getty* és l'encarregat d'esperar que s'engeguin els terminals. Quan un terminal és engegat, el procés que executa el programa *getty* carrega el **programa** *login* a fi d'autenticar el nou usuari. Un cop verificada la identitat de l'usuari, el programa *login* carrega l'**intèrpret d'ordres** (*shell*) i inicia una secció de treball amb l'usuari. Quan l'usuari acaba la sessió de treball el procés *shell* es destrueix, i *init* crea un nou procés *getty* que el substitueix.

4.2. La gestió de processos en Windows

Els elements que constitueixen un procés són els que ja hem vist anteriorment i no requereixen noves explicacions. Ens centrarem en aquest subapartat en alguns aspectes particulars de Windows pel que fa a la gestió de processos i el seu entorn.

4.2.1. La creació i la destrucció de processos

El més destacable pel que fa a la gestió de processos en Windows és que aquest sistema segueix un model explícit de pas de paràmetres. És a dir, en el moment en què es fa la crida per a demanar la creació d'un procés cal especificar explícitament els elements que formaran l'entorn d'execució del nou procés. El nou procés fill no serà, per tant, una còpia del pare, com passa en UNIX. Una conseqüència d'això és que la crida de Windows per a invocar la creació d'un procés (`CreateProcess`) té 9 paràmetres. Recordem que la crida equivalent d'UNIX (`fork`) no rep ni un sol paràmetre, ja que en el cas d'UNIX el procés fill és, en el moment de la creació, una còpia idèntica del pare. Algunes de les crides relacionades amb la gestió de processos es mostren en la taula següent:

Operació	Crides Windows
Crear_procés + Carregar_imatge	`CreateProcess`
Destruir_procés	`ExitProcess`
Esperar_finalització	`WaitForSingleObject` o `WaitForMultipleObjects`
Quisoc	`GetCurrentProcessId`

En aquest apartat mostrem alguns exemples o parts d'exemples extrets de la web MSDN que il·lustren la creació de processos, la sincronització i el readreçament de l'E/S. En els exemples observem com es poden manegar objectes per mitjà dels seus manejadors (*handles*).

Web recomanat

Podeu trobar més informació a la web de suport als desenvolupadors de Microsoft (MSDN), a la part de processos i fluxos. Actualment l'URL és http://msdn.microsoft.com/en-us/library/ms684841.

El primer exemple mostra com es pot crear un procés en Windows i com es pot sincronitzar amb la finalització del fill. Es pot observar com s'inicialitzen les estructures de dades de tipus `STARTUPINFO` i `PROCESS_INFORMATION` i com es fan les crides `CreateProcess` i `WaitForSingleObject`. El procés fill executarà el programa que s'indiqui com a paràmetre en executar el procés pare.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( int argc, TCHAR *argv[] )
{
   STARTUPINFO si;
```

```
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
       printf("Usage: %s [cmdline]\n", argv[0]);
       return;
    }

    // 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
    )
    {
       printf( "CreateProcess failed (%d).\n", GetLastError() );
       return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

4.2.2. Els canvis de l'entorn que configura un procés

L'exemple següent mostra com es pot crear un procés en Windows i com es pot fer el readreçament de les entrades i les sortides per mitjà dels manejadors (*handles*). Mostrem només alguns trossos que il·lustren com es pot fer el readreçament. L'exemple usa *pipes*. Tot i que estudiarem amb detall les *pipes* al darrer mòdul del curs, hem vist ja el seu funcionament bàsic en parlar de l'intèrpret d'ordres i la creació de filtres que connecten la sortida d'unes ordres

amb l'entrada d'altres. L'exemple complet es pot trobar a la web de suport als desenvolupadors de Microsoft sota el concepte "Creating a Child Process with Redirected Input and Output".

El codi del pare conté:

```
...
HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;
...
int _tmain(int argc, TCHAR *argv[])
{

   SECURITY_ATTRIBUTES saAttr;

   printf("\n->Start of parent execution.\n");

   // Set the bInheritHandle flag so pipe handles are inherited.
   saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
   saAttr.bInheritHandle = TRUE;
   saAttr.lpSecurityDescriptor = NULL;

   // Create a pipe for the child process's STDOUT.
   if ( ! CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0) )
      ErrorExit(TEXT("StdoutRd CreatePipe"));

   // Ensure the read handle to the pipe for STDOUT is not inherited.
   if ( ! SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0) )
      ErrorExit(TEXT("Stdout SetHandleInformation"));

   // Create a pipe for the child process's STDIN.
   if (! CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
      ErrorExit(TEXT("Stdin CreatePipe"));

   // Ensure the write handle to the pipe for STDIN is not inherited.
   if ( ! SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0) )
      ErrorExit(TEXT("Stdin SetHandleInformation"));

   // Create the child process.
   CreateChildProcess();

...
}

void CreateChildProcess()
```

```
// Create a child process that uses the previously created pipes for STDIN and STDOUT.
{
   TCHAR szCmdline[]=TEXT("child");
   PROCESS_INFORMATION piProcInfo;
   STARTUPINFO siStartInfo;
   BOOL bSuccess = FALSE;

   // Set up members of the PROCESS_INFORMATION structure.
   ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

   // Set up members of the STARTUPINFO structure.
   // This structure specifies the STDIN and STDOUT handles for redirection.
   ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
   siStartInfo.cb = sizeof(STARTUPINFO);
   siStartInfo.hStdError = g_hChildStd_OUT_Wr;
   siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
   siStartInfo.hStdInput = g_hChildStd_IN_Rd;
   siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

   // Create the child process.
   bSuccess = CreateProcess(NULL,
    szCmdline,   // command line
    NULL,       // process security attributes
    NULL,       // primary thread security attributes
    TRUE,       // handles are inherited
    0,       // creation flags
    NULL,       // use parent's environment
    NULL,       // use parent's current directory
    &siStartInfo, // STARTUPINFO pointer
    &piProcInfo);  // receives PROCESS_INFORMATION

   // If an error occurs, exit the application.
   if ( ! bSuccess )
      ErrorExit(TEXT("CreateProcess"));
   else
   {

   // Close handles to the child process and its primary thread.
   CloseHandle(piProcInfo.hProcess);
   CloseHandle(piProcInfo.hThread);
 }
}
```

El codi del fill és:

```
#include <windows.h>
#include <stdio.h>
```

```
#define BUFSIZE 4096

int main(void)
{
  CHAR chBuf[BUFSIZE];
  DWORD dwRead, dwWritten;
  HANDLE hStdin, hStdout;
  BOOL bSuccess;

  hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
  hStdin = GetStdHandle(STD_INPUT_HANDLE);
  if (
     (hStdout == INVALID_HANDLE_VALUE) ||
     (hStdin == INVALID_HANDLE_VALUE)
     )
     ExitProcess(1);

  // Send something to this process's stdout using printf.
  printf("\n ** This is a message from the child process. ** \n");

  // This simple algorithm uses the existence of the pipes to control execution.
  // It relies on the pipe buffers to ensure that no data is lost.
  // Larger applications would use more advanced process control.

  for (;;)
  {
    // Read from standard input and stop on error or no data.
    bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);
    if (! bSuccess || dwRead == 0)
       break;

    // Write to standard output and stop on error.
    bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);

    if (! bSuccess)
       break;
  }
  return 0;
}
```

4.2.3. Jerarquia de processos en Windows

En Windows els processos no tenen una relació jeràrquica, sinó que es tracten tots els processos com si pertanyessin a una mateixa generació. Per a sincronitzar-se amb la finalització d'un procés només cal tenir el manejador i l'identificador del procés. Com el procés pare té aquestes informacions, es pot mantenir o simular una relació jeràrquica si vol.

Pel que fa als processos de sistema, hi ha tot un conjunt de processos que implementen diferents serveis. Cal destacar com a processos de rellevància especial i que no poden ser aturats: el procés nul, anomenat *System Idle Process*, que té el PID 0 i s'executa quan no hi ha res per fer, i un procés anomenat *System* **amb PID 4, que és el nucli del sistema operatiu.**

5. *Pthreads*

Avui dia els sistemes operatius solen proporcionar directament implementacions de fluxos. Diversos llenguatges de programació també en faciliten la programació si els tenen integrats a la seva sintaxi. No obstant això, aquí presentem els fluxos definits per l'estàndard POSIX, anomenats *pthreads*.

L'avantatge de fer implementacions amb *pthreads* és la gran portabilitat que es pot obtenir, atesa la gran implantació d'aquests fluxos en la pràctica totalitat dels sistemes.

Hi ha moltes primitives relacionades amb la gestió dels *pthreads*, però aquí ens centrarem en les més bàsiques. El primer que hem de conèixer són les primitives que permeten crear (`pthread_create`), finalitzar (`pthread_exit`) i sincronitzar (`pthread_join`) *pthreads*. Una crida a `pthread_join` bloquejarà el *pthread* que la faci fins que el *pthread* amb el qual es demana sincronitzar no hagi acabat. A més, la crida li retornarà informació sobre l'estat de finalització que el *pthread* hagi indicat en fer la crida `pthread_exit`. Si no es vol esperar un *pthread* es pot indicar per mitjà de la primitiva `pthread_detach`.

A més, es pot aconseguir informació sobre l'identificador del *pthread* emprant la primitiva `pthread_self`.

Primitiva	Descripció
`pthread_create`	Creació d'un *pthread*
`pthread_exit`	Terminació d'un *pthread*
`pthread_join`	Espera la finalització d'un *pthread*
`pthread_detach`	Indica que no es voldrà esperar al *pthread*
`pthread_self`	Retornar l'identificador del *pthread*

Cal desar el valor de retorn indicat en fer un `pthread_exit` fins que es faci un `pthread_join` o un `pthread_detach`. Per això, l'estructura de dades que representa el *pthread* no serà alliberada fins després d'executar la crida que es faci més tard de la combinació `pthread_exit` i `pthread_join`, en el cas que vulguem una sincronització, o `pthread_exit` i `pthread_detach`, en el cas que no vulguem una sincronització. Hem de ser conscients que, com amb qualsevol altre recurs, cal que s'alliberi l'estructura de dades del *pthread* quan es deixi de necessitar. Per això, com a dissenyadors de programes que usen *pthreads*, hem de tenir en compte aquests aspectes.

Com que els *pthreads* d'un procés comparteixen memòria, es poden comunicar ràpidament per mitjà d'aquesta. Però això porta a la necessitat d'emprar mecanismes de sincronització i exclusió mútua. És convenient conèixer com a mínim algunes primitives que permeten protegir la modificació d'estructures de dades de manera concurrent i la implementació de regions crítiques. Això es pot fer forçant l'accés a les regions crítiques en exclusió mútua (*mutual exclusion* o *mutex*). Per això hi ha unes variables de tipus `pthread_mutex_t` sobre les quals es pot demanar l'accés de manera exclusiva en un moment determinat emprant la primitiva `pthread_mutex_lock` (adquirir el *lock* sobre el *mutex*). Un cop la crida retorna, tenim la garantia que el *pthread* que ha fet la crida és l'únic que té dret a fer les accions que volem protegir amb aquesta variable de *mutex* i pot entrar a la regió crítica i executar les operacions que conté. En acabar aquestes operacions, cal alliberar la variable de *mutex* emprant `pthread_mutex_unlock` per tal de permetre l'accés a la regió crítica a altres *pthreads* que estiguin esperant.

Vegeu també

Al mòdul 7, tractarem a fons el problema de la sincronització i de l'exclusió mútua.

Primitiva	Descripció
`pthread_mutex_lock`	Demanar accés en exclusivitat al recurs *mutex*
`pthread_mutex_unlock`	Alliberar recurs *mutex*

A continuació mostrem un exemple senzill disponible en múltiples llocs d'Internet que il·lustra l'ús de les crides bàsiques per a aconseguir crear 10 *pthreads* que treballin concurrentment, però modifiquin una variable compartida en exclusió mútua, i garanteixin que el resultat final d'un comptador sigui equivalent a una execució en sèrie.

```
#include <stdio.h>
#include <pthread.h>

#define NTHREADS 10

void *thread_function();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;
   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, &thread_function, NULL );
   }
   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id[j], NULL);
```

```
    }
    /* Now that all threads are complete I can print the final result. */
    /* Without the join I could be printing a value before all the threads */
    /* have been completed. */
    printf("Final counter value: %d\n", counter);
}

void *thread_function()
{
    printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}
```

Resum

Vegeu també

Vegeu la definició de *procés* en el subapartat 1.2. del mòdul didàctic 2.

En aquest mòdul didàctic hem analitzat la **gestió de processos** que fa el SO partint de la definició de procés que ja havíem vist en aquesta assignatura. Ho hem fet seguint els passos següents:

a) En primer lloc, per entendre millor el concepte de procés, hem analitzat de manera superficial la **representació interna del procés en el SO** i la **gestió dels processos en un sistema multiprogramat de temps compartit**. Els principals conceptes que hem presentat són els següents:

- El **bloc de control de processos**, com a estructura bàsica que configura un procés.
- L'**execució concurrent**, mitjançant la multiplexació del processador en temps.
- L'**estat dels processos** (*run*, *ready* i *wait*).
- El **canvi de context** com a mecanisme per a passar un procés de l'estat *run* a *ready* o *wait*, o al revés.

b) En segon lloc, hem analitzat des de l'òptica de l'usuari les **crides que permeten manipular els processos**, és a dir, que els permeten crear, modificar i destruir. Com a conceptes principals relacionats amb aquestes crides podem destacar els següents:

- L'**herència entre els processos pare i fill** en el moment de la creació d'un nou procés.
- La **sincronització**, per a veure com el procés pare sincronitza la creació i la destrucció d'un procés fill.
- Els **canvis d'executable** i els **reencaminaments**, com a principals canvis de l'entorn que configura un procés.

A continuació hem descrit els fluxos d'execució (*threads*) que permeten l'execució concurrent del codi d'un procés. Hem vist les característiques dels fluxos que comparteixen la majoria dels recursos del procés al qual pertanyen, tot i requerir alguns recursos propis de cada flux per a garantir un funcionament correcte. Hem vist que el canvi de context entre fluxos d'un mateix procés és menys costós que el canvi entre fluxos de diferents processos. Emprant fluxos es pot facilitar la programació i millorar l'eficiència de molts programes.

La figura següent mostra un mapa conceptual amb els principals conceptes explicats en aquest mòdul didàctic i les seves relacions:

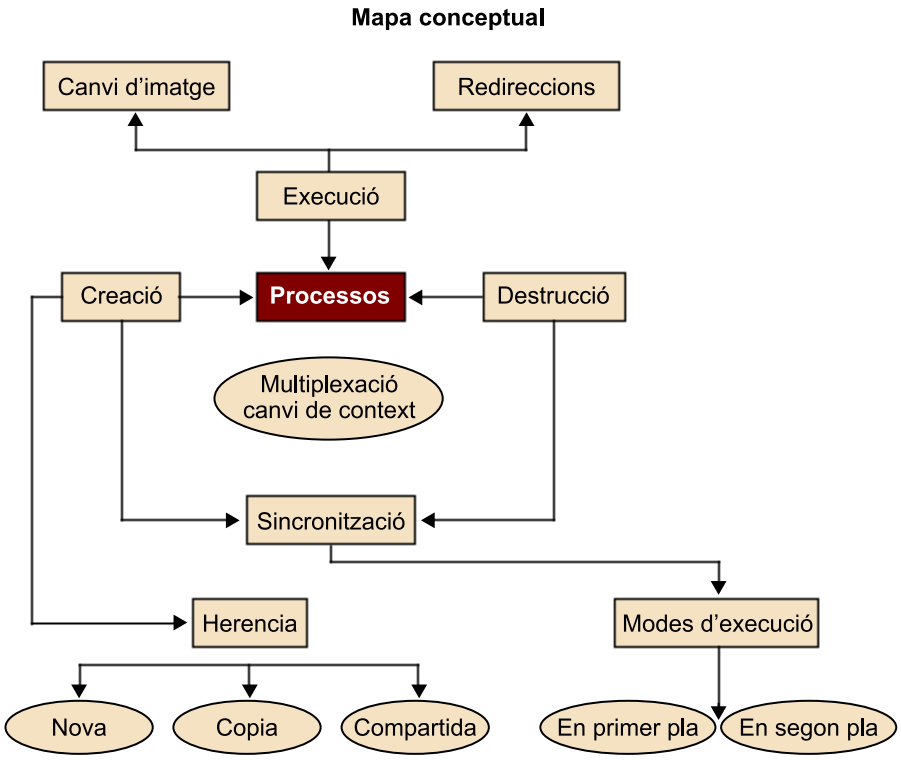


Figura 15

Després de veure els SO en general, hem fet una confrontació dels conceptes anteriors amb el sistema UNIX. Les crides al sistema de gestió de processos segueixen una filosofia de disseny basada en la senzillesa i la flexibilitat que permet crear i modificar els processos amb tota llibertat des del programa. Una conseqüència d'aquest fet és la gran quantitat de possibilitats que pot oferir l'intèrpret d'ordres (*shell*) pel que fa a modes d'execució i de reencaminament de les entrades i sortides.

A continuació hem destacat les diferències que trobem en els sistema Windows relatives a la gestió de processos, l'herència de recursos i la jerarquia de processos.

Finalment, hem introduït la funcionalitat bàsica del paquet de fluxos de l'estàndard POSIX (*pthreads*).

Activitats

1. Estudieu l'ordre `ps` d'UNIX. Vegeu quins camps presenta i quin significat tenen. Analitzeu quins processos teniu en marxa en el sistema i en quin estat es troben.

2. Linux permet veure els recursos assignats als processos com a fitxers que pengen del directori `/proc`. Mireu en el manual d'UNIX l'entrada associada a aquest directori, i analitzeu què s'hi pot trobar.

3. Analitzeu les diferents possibilitats d'execució d'ordres i de reencaminaments que ens ofereix l'intèrpret d'ordres (*shell*) d'UNIX i penseu com es podrien fer des de les crides al sistema.

4. Estudieu la manera de crear processos en Windows i la manera de reencaminar les entrades i les sortides estàndard.

5. Creeu un programa que creï diversos `pthreads` que executin una rutina, investigant com es pot passar paràmetres al *pthread* en el moment de fer el `pthread_create`. Us heu d'assegurar que la informació passada per paràmetre seguirà estant disponible en el moment en què el *pthread* hi intenti accedir, sense suposar res sobre la velocitat ni l'ordre en el qual s'executen els *pthreads*.

Exercicis d'autoavaluació

1. Dintre del diagrama d'estats dels processos que es mostra en el subapartat d'estats d'un procés, on col·locaríeu l'estat *zombie* d'UNIX? Justifiqueu la resposta.

2. Com es podrien reencaminar les entrades/sortides dels processos fill en un sistema en què la crida *crear_procés* força el canvi d'imatge? Justifiqueu la resposta.

3. Després d'executar *primer,* quines sortides obtindrem i per quin canal es faran? Justifiqueu la resposta.

Primer

```
main() {
        char a;

        a = 'A';
        if (write(1,&a,1) == -1) {
                /*error*/
                write(2,"error",5);
        }
        close(1);
        execl("segon","segon",0);
        if (write(2,&a,1) == -1) {
                /*error*/
                exit(1);
        }
}
```

Segon

```
main() {
         char a;

         if (write(1,&a,1) == -1) {
                  /*error*/
                  write(2,"error",5);
         }
         if (write(2,&a,1) == -1) {
                  /*error*/
                  exit(1);
         }
        a = 'B';
        if (write(2,&a,1) == -1) {
                /*error*/
                exit(1);
        }
}
```

De selecció

4. Quan es porta a terme un canvi de context entre dos processos, el sistema ha de desar els valors que configuren l'estat del procés que deixa el processador a fi de poder reprendre'n l'execució més endavant. Digueu quins dels elements següents que configuren un procés han de ser desats explícitament en el moment del canvi:

a) El comptador de programa.

b) Els segments de memòria.

c) Els dispositius virtuals.

d) L'identificador del procés.

e) Els registres del processador.

Justifiqueu la resposta.

5. Quin dels resultats següents creieu que produirà l'execució del programa següent:

```
main() {
   int fd, pid;
   char buff[2];

   fd = open("fitxer",O_RDONLY);
   if (read(fd,buff,2) == -1){
      /*error*/
      exit(1);
   }
   write(1,buff,2);
   pid = fork();
   switch(pid){
      case 0: if (read(fd,buff, 1) == -1){
                /*error o fi de fitxer*/
                exit(1);
              }
              write(1,buff,1);
              exit(0);

      default: if (read(fd,buff,1) == -1){
                /*error o fi de fitxer*/
                exit(1);
              }
              write(1,buff,1);
              exit(0);
   }
}
```

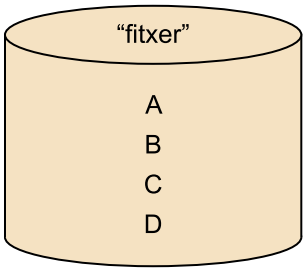


Figura 17

a) ABCD.

b) AABBCCDD.

c) ABCCDD.

d) ABDC.

e) Les opcions ***a*** o ***d*** indistintament.

Justifiqueu la resposta.

6. Digueu quin dels resultats següents creieu que produirà l'execució del programa que presentem a continuació:

```
main() {
        int num, pid;

        num = 3;
        pid = fork();
        switch(pid) {
                    case 0:
                            num = num + 1;
                            printf("fill: num = %d\n",num);
                    default:
                            num = num + 2;
                            printf("pare: num = %d\n",num);
        }
}
```

a) "fill: num = 4", "pare: num = 6".

b) "fill: num = 4", "pare: num = 5".

c) "fill: num = 1", "pare: num = 2".

d) "fill: num = 4", "pare: num = 5".

e) El valor de *num* no està definit en el fill i el pare efectua la sortida "pare: num = 5".

Justifiqueu la resposta.

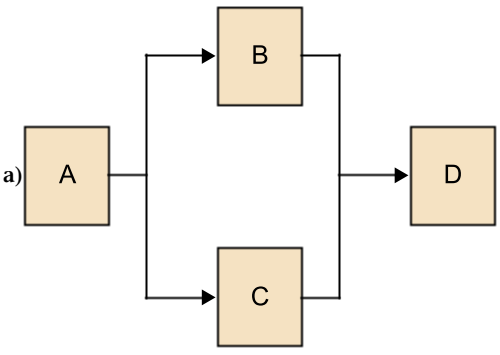
7. Quin dels diagrames de temps és el que representa l'execució del codi següent?

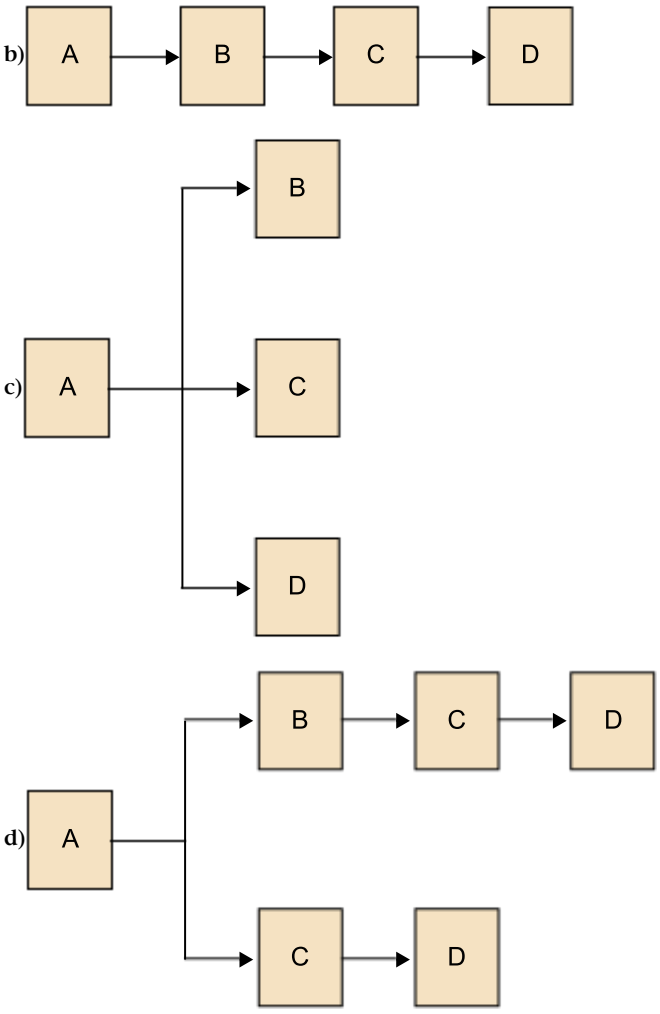
Justifiqueu la resposta.

```
main() {
        int pid1, pid2, estat;

        A
        pid1 = fork();
        switch(pid1) {
                     case 0:
                             B
                     default:
                             pid2 = fork();
                             switch(pid2) {
                                          case 0:
                                                  C
                                          default:
                                                  while (pid1 != wait(&estat));
                                                  while (pid2 != wait(&estat));
                                                  D
                             }
        }
}
```

a)




b)
A
B
C
D
c)
A
B
C
D
d)
A
B
C
D
C
D

Solucionari

Exercicis d'autoavaluació

1. L'estat *zombie* és previ a la destrucció total del procés mentre aquest espera que el procés pare, o si no el procés *init*, reculli el paràmetre de finalització. Un cop recollit s'acaben d'alliberar els recursos del procés.

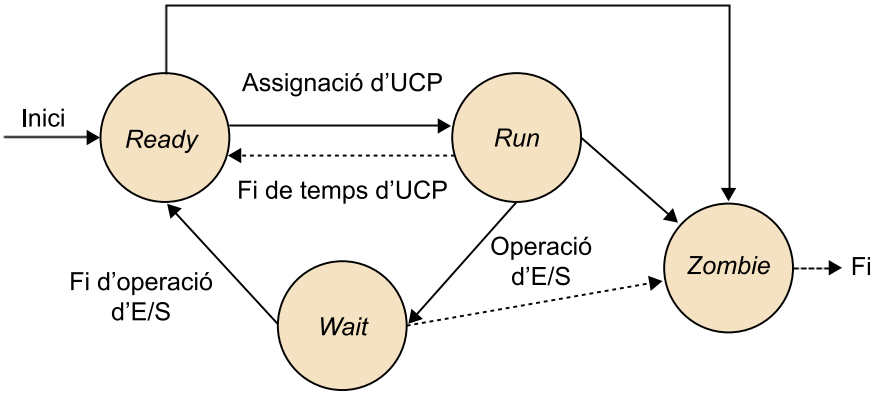


Figura 16

2. Com que la crida *crear_procés* obliga a carregar un nou executable, és impossible que el fill hereti el codi del procés pare, per la qual cosa no es pot especificar mitjançant el programa com s'ha de modificar l'entorn d'entrada/sortida del procés fill. En aquestes condicions, amb els paràmetres de la crida *crear_procés* s'han de poder especificar els dispositius lògics que es volen associar als dispositius virtuals estàndard del procés fill. Per exemple:

crear_procés(*nom_executable*,*fitxer1*,*fitxer2*,*fitxer3*,*altres paràmetres*...)

en què *fitxer1* correspon a l'entrada estàndard, *fitxer2* a la sortida estàndard i *fitxer3* a la sortida d'error. Si es volgués fer més flexible permetent reencaminar qualsevol dispositiu amb qualsevol mode d'accés, caldria incloure-hi més paràmetres.

3. La sortida s'efectuarà de la manera següent:

a) L'executable *primer* escriu "A" per a la sortida estàndard.

b) Es tanca la sortida estàndard.

c) Es canvia d'executable. Es carrega *segon* i tot el codi que l'executable *primer* té per sota de la crida `exec` desapareix, i no s'executarà mai.

d) L'executable *segon* escriu "error" per a la sortida d'error estàndard, ja que està tancada, tal com hem comentat en el punt ***b***.

e) L'executable *segon* escriu un caràcter indeterminat per a la sortida d'error estàndard, ja que la variable a de l'executable *segon* s'ha creat en el moment de la càrrega, i no ha estat inicialitzada.

f) L'executable *segon* escriu "B" per a la sortida d'error estàndard.

4. S'han de desar el comptador de programa (**a**), que indica la propera ordre que s'ha d'executar, i el valor dels registres del processador (**e**), que contenen, bàsicament, valors parcials dels càlculs que s'estan duent a terme i el punter a l'última posició de la pila. S'han de desar tots aquells elements dinàmics que configuren el punt d'execució en què es troba el programa que conté el procés. La resta d'elements, tot i que configuren l'entorn d'execució, ja estan desats en el PCB, de manera que no cal tornar-los a desar.

5. La resposta correcta és la ***e***. Els processos pare i fill comparteixen els punters de lectura/escriptura dels fitxers que el pare té oberts en el moment de la creació del fill. Així, les operacions de lectura que faci qualsevol dels dos modifiquen el mateix punter. D'altra banda, els dos processos s'executen concurrentment i, per tant, no podem saber quin dels dos efectuarà primer l'operació de lectura. Per aquest motiu es poden donar totes dues sortides.

6. La solució correcta és la ***b***. L'herència de la crida `fork` és de còpia. Per tant, el procés fill tindrà, un cop creat, una còpia del mateix codi i les mateixes dades que el pare. A partir del moment de la creació, cadascun evolucionarà independentment i amb les seves variables.

7. La resposta correcta és la ***a***. El procés pare executa el codi A i després crea un procés fill que executa el codi B. Tot seguit, el pare crea un segon procés fill que executa concurrentment el codi C. Un cop creats tots dos fills, el procés pare espera que els seus dos processos fill hagin acabat, i executa D.

Glossari

bloc de control de processos *m* Estructura de dades que conté la informació de l'entorn de cada procés necessària perquè el sistema operatiu pugui gestionar l'execució concurrent d'un conjunt de processos.
sigla **PCB**

canvi de context *m* Tècnica que, mitjançant la multiplexació del temps de processador, aconsegueix l'execució concurrent de tots els processos preparats per a utilitzar-lo. Requereix desar l'estat dels processos o fluxos per a restaurar-lo quan es vulguin continuar executant.

estat d'un procés *m* Estat que s'assigna a cada procés per a controlar els seus canvis de mode d'execució al llarg de la seva existència en el sistema. Només hi ha un conjunt limitat d'estats possibles, i les accions que poden donar lloc a transicions entre els estats també estan definides.

flux d'execució *m* Unitat mínima de planificació del sistema operatiu. Un flux forma part d'un procés i gaudeix dels recursos assignats al procés. Un procés té com a mínim un flux, però en els sistemes operatius actuals en pot tenir més d'un. Els diferents fils que formen part d'un mateix procés comparteixen certs recursos, com l'espai de memòria, els arxius oberts, els permisos, el directori de treball, l'identificador de procés, etc. En canvi, cada fil té la seva pròpia d'execució pila, pot estar executant diferents ordres (cada fil té el seu registre comptador propi de programa o PC), s'executen a diferents velocitats i tenen el seu estat d'execució propi. Els fils d'execució també són coneguts com a **processos lleugers** pel fet que els fils d'execució consumeixen menys recursos de sistema que els processos.
sin fils d'execució
en thread

fil d'execució *m* Vegeu **flux d'execució**.

PCB *m* Vegeu **bloc de control de processos**.

quota (*quantum*) *f* Temps màxim d'UCP que pot utilitzar un procés de manera contínua. Els sistemes operatius que treballen en la modalitat de temps compartit fan ús de la quota per a fer canvi de context i donar el control del processador a un nou procés.
en quantum

quantum *m* Vegeu **quota**.

thread *m* Vegeu **flux d'execució**.

Bibliografia

Bibliografia bàsica

Silberschatz, A.; Galvin, P.; Gagne G. (2008). *Operating Systems Concepts* (8a. ed.). John Wiley & Sons.

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice Hall.

Bibliografia complementària

Kernighan, B.; Pike, R. (1987). *El entorno de programación UNIX*. Mèxic: Prentice Hall Hispanoamericana.

Robbins, K.; Robbins, S. (2000). *UNIX: programación práctica*. Prentice Hall.

Documentació disponible sobre crides al sistema UNIX en els recursos de l'aula.