Sistemes Operatius II

Pau Soler Valadés

Tardor 2019

Índex

S'assumeix que el lector ha cursat i (com a mínim) ha entès els continguts de Sistemes Operatius I. No s'entrarà en detall en parts ja tractades en dita assignatura a menys que ho trobi molt necessari.

1 Màquina Virtual

A la definició de Sistema Operatiu (SO) deiem (entre altres coses) que oferia una **màquina virtual** i una interfície molt més senzilla d'utilitzar. Però tot i oferir-la, podem dir que el SO és una màquina virtual?
No, no ho és estrictament per la seva definició.

Definició 1.1. Una **Màquina Virtual** és una peça de software que emula un SO i involucren diversos components:

- El **Host**: corresponent al maquinari.
- La **Virtual Machine manager** (o **hypervisor**): crea i permet executar diverses màquines virtuals sobre el host, donant-li una interfície a on fer-ho.

A la VMM s'hi poden executar múltiples SOs i per cada un se li poveeix una còpia del host anomenada **Guest**. En sí mateix, guest ja és un SO.

Definició 1.2. La **Virtual Machine Manager** és un SO que permet executar altres SO.

Propietats. Virtual Machine Manager

- Un cop la VMM s'instal·la a l'ordinador proveeix la planificació dels processos i la gestió de memòria
- Permet que amb un sol hardware puguis executar múltiples SOs a la vegada.
- Tenim múltiples VMM:
  - **Type 0 Hypervisors:** Basades en el maquinari que donen suport a la gestió de màquines virtuals via programari. Usades als *mainframes* i als servidors grans
  - **Type 1 Hypervisors:** Basades en programari. Proveeixen tota la virtualització necessària. Exemple: VMWare ESX
  - **SO d'ús general:** Proveeixen addicionalment serveis de VMM. Exemple: mòdul kvm a linux. Anomenats també type 1 hypervisors.

- **Type 2 Hypervisors:** Aplicacions que executen en SO d'ús general i dónen serveis de màquina virtual. Exemple: VMWare Workstation.
- **Virtualització a nivell d'entorn de programació:** Les VMM no virtualitzen el maquinari real sinó un maquinari virtual optimitzat. Exemple: Java.
- **Emuladors:** Permeten que una aplicació programada per un maquinari específic pugui executar-se en un altre maquinari diferent.
- **Contenidors d'aplicacions:** No es pot considerar una virtualització però ofereixen serveis similars i permeten executar una aplicació segregada del SO.

Primera aparició als *mainframes* d'IBM, sistema que ha evolucionat i encara s'usa avui en dia.

S'usen sobretot a servidors o granges de servidors arreu del món.

1.1 Beneficis

- Aïllament: Les màquines virtuals estan (pràcticament) aïllades entre sí, garantint protecció entre elles. Per a fer-ho, es pot oferir una màquina restringida a l'usuari: menys RAM, CPU i recursos dels realment disponibles.
- Flexibilitat i portabilitat: La virtualització ens permet aturar, copiar, transportar a una altra màquina virtual, emmagatzemar-la...
- Desenvolupament i testeig: Podem provar i desenvolupar per més plataformes més fàcilment al mateix temps, ja que podem tenir més d'un SO al mateix hardware.
- Núvol: S'usen molt amb la computació al núvol per oferir computació a internet.

[TODO: fer un petit apartat amb Dockers, que són molt interessants.]

2 Sistemes de Fitxers

Definició 2.1. (Intuïtiva) Els **Sistemes de fitxers** gestionen de forma transparent als usuaris de manera que poguem usar directoris i fitxers tal i com els coneixem.

En aquesta secció tractarem la evolució historica dels sistemes d'arxius, ja que el problema d'emmagatzemar, recuperar i manipular fitxers és molt general i no hi ha una manera correcta d'implementar-ho. Depèn completament del suport d'emmagatzemantge com de les aplicacions que l'usaran. Aquests suports, tot i haver canviat molt durant els anys (des de els diskettes fins a les unitats d'estat sòlid) tenen característiques comunes i molt importants en un sistema d'arxius.

Definició 2.2. Un **Bloc o Sector de Disc** és la mínima unitat que el disc pot llegir o escriure (típicament 512 bytes).

Definició 2.3. Un **Bloc del Sistema d'Arxius** és la mínima unitat que els sistema d'arxius del SO llegeix o esciu.

Propietats. *Bloc del Sistema d'Arxius*

- Tot el que fa el sistema d'arxius està compost d'operacions sobre blocs.
- La mida d'un Bloc del Sistema és major o igual que la de un Sector de disc en múltiples sencers en potència de dos. Els tamanys més comuns son de 1024, 2048, 4096 o 8192 bytes.

[TODO: HDD i SSD]

Per emmagatzemar fitxers en un disc el sistema de fitxers del SO ha d'emmagatzemar les **metadades** associades al fitxer: localització al disc, mida del fitxer, nom del fitxer, drets d'accés, dates de modificació i creació... Aquestes són les operacions que pot realitzar el sistema de fitxers sobre un fitxer:

- Crear Fitxers: Trobar ràpidament espai lliure al disc pel fitxer.
- Escriure a un fitxer: Donat el nom d'un fitxer, trobar-lo a disc i escriure-hi.
- Llegir d'un fitxer: amb el fitxer localitzat, poder llegir les dades que conté.
- Posicionament en un fitxer: posicionar el punt d'escriptura o lectura a qualsevol lloc d'un fitxer.
- Esborrar un fitxer: Eliminar el fitxer de les metadades del directori i l'espai que aquest ocupa a disc.

- Truncar un fitxer: Permet alliberar l'espai que ocupa el fitxer sense esborrar-lo

Un cop tenint fitxers, els sistemes de fitxers habituals han de permetre emmagatzemar el llistat de fitxers i els subdirectoris que contenen. Aquestes són les operacions que pot fer el sistema de fitxers sobre els directoris:

- Buscar un fitxer: Trobar el fitxer al directori per accedir a les metadates.
- Crear un fitxer: permetre afegir un nou fitxer a un directori.
- Esborrar un fitxer: eliminar el fitxer d'un directori amb les dades associades.
- Reanomenar un fitxer: Canviar el nom d'un fitxer del directori.
- Llistar el contingut d'un directori: Llistar els fitxers (i subdirectoris) del directori.

2.1 Nivell Físic

Els arxius i les metadates s'han de guardar en blocs de disc. Aquests no es poden dividir, és a dir, un fitxer i les seves metadades estaran guarades en un nombre enter de blocs del sistema. Dues propietats importants:

- Els bytes d'un bloc utilitzat NO poden ser utilitzats per altres fitxers.
- Per accedir un byte concret d'un arxiu s'ha de carregar el bloc senser a memòria per fer-ho, no s'hi pot accedir directament.
- Ni els arxius ni les metadates tenen perque estar emmagatzemats linealment, encara que a nivell d'usuari es percebi que el bytes son contigus. Mitjançant punters a llocs de la memòria es com s'aconsegueix la il·lusió de la contiguitat.
- Idealment els blocs d'un arxiu han de ser tots continguts a disc, ja que el rendiment del disc dur és major.

Però com sap el SO que un fitxer té tants blocs, on son i com accedir-hi?

2.2 Fonaments de la implementació

Intuïtivament, cada directori s'emmagatzema com un fitxer amb una estructura similar a una taula que conté:

- El llistat de fitxers, amb la mida, els permisos amb un apuntador cap als blocs que formen el fitxer.

- Per cada directori que tingui un subdirectori hi haurà l'apuntador cap el fitxer que conté els fitxers del subdirectori.

Aquesta taula acaba sent massiva i molt difícil de gestionar de manera eificient. El sistema de fitxers ha de saber quins blocs estan lliures i si són contigus millor. Si podem evitar la fragmentació d'un fitxer, ho farem. Generelitzant les característiques anteriors, un sistema de fitxers serà bo si compleix les següents característiques:

- Les dades dels fitxers es tendeixes a guardar-se a disc de manera contínua.
- Tenir un lloc propi per les metadates dels fitxers.
- Accés eficient a un bloc d'un fitxer (minimitzar el moviment del braç del disc en un HDD)
- Eficient tan per fitxers petits com grans.

Durant la historia dels computadors s'han donat diferents respostes a com fer sistemes de fitxers,i en veurem la evolució històrica de 4: el FAT (File Allocation Table), FFS (Unix Fast File System), NTFS (Microsoft New Technology File System) i COW (ZFS, btrfs).

2.3 FAT: File Allocation Table

La primera versió va ser desenvolupada per Microsoft. Tot i que nosaltres ens centrarem en la FAT-32, capaç de gestionar discs amb 2^{28} blocs i fitxers de fins a $2^{32} - 1$ bytes.

L'estructura dels FAT és la següent:

- **Taula Fat:** conté informació sobre els blocs que formen part de cada fitxer.
- **Directori arrel:** Conté la informació sobre els subdirectoris i els fitxers que formen (hi ha a?) el directori arrel.
- **Fitxers i directoris:** s'emmagatzemen els fitxers i els directoris de disc. S'omple a mesura que s'hi afegeixen dades.

[TODO: imatge de com esta repartit en una linia (diapo 26)]

La FAT és **una llista enllaçada de sensers** que emmagatzema els blocs de disc del qual es composa cada fitxer.

Els fitxers que explicíten un directori son senzills: una taula (no ordenada) dels fitxers que componen el directori. Per cada fitxer es guarden les seves **metadades** i el primer

índex de la taula FAT que ocupen, que és el primer bloc de disc que ocupa el fitxer en qüestió.

Si volem entrar un nou fitxer a la taula només hem de recorrer la taula FAT buscant si hi ha elements no ocupats, que son els marcats amb un 0 a la taula. Un cop se'n troba un, un algorisme senzill busca el següent element de la taula lliure. Això implica que els fitxers de la FAT tendeixen a estar molt fragmentats. Per a solucionar això, la FAT té un desfragmentador que intercanvia blocs de disc de manera que estiguin més contiguus.

[TODO contestar les preguntes a caire d'exemple o no, depen de si em fa molt pal]

2.3.1 Limitacions del Sistema FAT

FAT és molt usat per la seva simplicitat, però té moltes limitacions:

- **Fitxers fragmentats:** tal com hem explicat abans (i tot i tenir l'intercanviador) els fitxers queden molt fragmentats, cosa que repercuteix directament en la veloctiat a la localització i l'accés a les dades.
- **Accés aleatòri pobre:** Per poder accedir a un bloc del fitxer, hem de recórrer forçosament la taula FAT.
- **Taula de fitxers d'un directori:** No els tenim ordenats, cosa que dificulta molt la tasca.
- **Control d'accés pobre**: No té permisos d'accés pels diferents usuaris.
- **Limitació en la mida de disc i fitxer:** En un disc amb mida de 4KB, la mida màxima del disc és de 1TB, i la del fitxer de 4GB.
- **Seguretat:** No té control sobre recuperació ni de dades ni de metadades en cas de fallada del sistema.

En els sistemes que veurem a continuació veurem com a mesura que avançava el temps s'anaven perfeccionant i solucionant els errors que acabem d'enumerar.

2.4 FFS: Unix Fast File System

A diferència dels FAT, els FFS aporten noves idees com:

1. Controlar quins usuaris i què poden fer (lectura, escriptura o execució) a cadascun dels fitxers.

2. Els blocs que formen part d'un fitxer s'indexen de manera que sigui eficient treballar amb fitxers petits i grans.

3. Els fitxers estan en blocs contiguus a disc mitjançant heurístiques.

L'estructura del sistema FFS és la següent:

- **i-nodes:** Emmagatzemen els blocs que componen un fitxer.
- **Directori Arrel:** El fitxer que conté la informació sobre els subdirectoris i els fitxers que formen el directori arrel.
- **Free Space Management:** S'emmagatzema la informació dels blocs lliures a disc.
- **Fitxers i Directoris:** s'emmagatzemen els fitxers i directoris del disc. S'omple a mesura que s'hi afegeixien dades.

[TODO: Imatge de la estructura.]

Els **i-nodes** tenen una mida fixa i n'hi ha un nombre determinat a memòria. Al ser fix aquest nombre, podem implementar els i-nodes com una taula indexada on cada node té un índex a la taula. Llavors, cada i-node emmagatzema els atributs i els blocs de fitxers que formen part d'un fitxer. D'això n'extraiem dues conclusions:

- Un directori no és més que un llistat amb noms de fitxers i els i-nodes que l'hi corresponen.
- Cada i-node es pot veure com un arbre, on la root és l'i-node i les fulles són els blocs de fitxers.

En resum:

- Com que hi ha un i-node per a cada fitxer (amb els seus atributs) i aquests tenen una mida fixa son extremadament ràpids de localitzar.
- Al fer servir una estructura d'arbre als i-nodes fa que es pugui accedir a qualsevol bloc de manera eficient.

A part d'això, els i-nodes contenen una array de punters cap als blocs que formen els fitxer. Poden haver-hi diferents tipus de punters, cosa que ara explicarem.

2.4.1 Puntes Directes i Indirectes

Quan obrim un fitxer es llegeix l'i-node, s'emmagatzema a memòria i se saben els blocs que formen el fitxer. Quant ocupa això?

En un sistema amb una mida de bloc de 4096 bytes si fem servir 12 punters directes tenim màxim (12×4096) 48 KB de mida de fitxer. Això ens porta a dues preguntes: és suficient per la majoria de fitxers i què passa si el fitxer és major que aquest tamany.

La resposta a la primera pregunta és que sí, 48 KB és suficient per la grandíssima majoria de fitxers (si voleu, comproveu-ho vosaltres mateixos amb un l·ls -l"). Si el fitxer que hem de guardar és més gran hem d'usar punters indirectes. Continuant amb el sistema de 4 KB, teniem 48 KB guardats de punters directes. Si afegim punters indirectes de 4 Bytes a la equació ens queda $48KB + 4096 \times 4 = 48 + 1024 \approx 4MB$. Aquest procés es pot prolongar amb punters doble i triplement indirectes. Aquí deixo un taula que mostra aproximadament quant es pot emmagatzemar.

Punters	**Equivalent**	$\approx$
12 directes	-	48KB
1 indirecte	1024	4MB
1 doble ind	1024^2	4GB
1 triple ind	1024^3	4TB

Resumint: aquest tipus d'emmagatzament expressa un arbre fixe multinivell asimètric capaç de donar suport tant a fitxers petits com a fitxers grans de manera eficient. Cal mencionar que, sempre s'omplen els punters menors i si en fan falta s'omplen els majors. És a dir, per guardar un fitxer de 4694016 bytes (4MB i 48KB) primer es guardaran en punters directes els 48 KB i quedaran $4MB + 48KB - 48KB = 4MB$ que es guardaran en punters indirectes.

[TODO: potser fer esquema de com s'ha de buscar un fitxer, però és prou elemental i al power s'enten bé.]

2.4.2 Espai Lliure

Per gestionar l'espai lliure, FFS usa un bitmat emmagatzemat a disc en què cada bloc es representa amb un bit. El bitmap és en una posició coneguda pel sistema de fitxers. Com cerquem l'espai lliure eficienment? El disc es divideix en grups que contenen les metadates associades (els i-nodes, directoris, bitmap d'espai buit...) Cada directori està associat a un determinat grup, i FFS intenta guardar els fitxers del directori al grup que l'hi correspon, així ens assegurem dues coses:

- Tenir controlat l'espai lliure, ja que va associat dins de cada directori.
- Guardar els fitxers lo més continguus possible, ja que s'intenta associar amb cada directori.

[TODO: Comparació FAT vs FFS]

2.5 NTFS: Microsoft New Technology System

Els sistemes NTFS flexibilitzen els i-nodes dels FFS. En comptes de referenciar blocs individuals de 4KB, es referencien els extents: blocs de mida variable i tant els directoris com l'espai disponible a disc es guarden com a arbres balançejats. Donat el nom d'un fitxer, es calcula un valor de hash associat que els permet localitzar rapidament.

NTFS va introduïr un sistema per mantenir la consistència de les metadades en cas de fallada del sistema, anomenat **Journalling** o registre per diari.

Definició 2.4. Journalling és una tècnica usada als sistemes de fitxers per assegurar que les estructures es mantenen consistents independentment de les fallades del sistema que hi pugui haver.

Propietats. *Journalling*

- Garanteixen la consistència de les metadades dels fitxers, però no les dades de cada fitxer.
- La recuperació de les fallades de sistema es redueix de minuts a segons (comparant un sistema que no té *journaling* contra un que sí.)

Com funciona?

S'implementa un **registre (log) circular** (diguem-ne un fitxer "especial") en que s'emmagatzemen les operacions previstes a realitzar a disc. Abans de fer una modificació als i-nodes s'indica al registre circular quina operació es realitzarà. Un cop feta, es marca com a tal al registre. Si l'ordinador fallés en algun moment mentres es realitza una operació, es podrà comprovar al engegar el SO, si totes les operacions al registre circular han estat realitzades. En cas negatiu, es procedeix a çorregir"els i-nodes afectats.

2.6 COW: Copy On Write

Ara per ara ja tenim sistemes que:

- Localitzen ràpid les parts del fitxer

- Localitzen ràpid la memòria lliure i els directoris
- I que tenen recuperació de les metadades al fallar.

Només ens queda implementar un sistema de fallades sobre les dades del propi fitxer.

Els sistemes Copy on Write ho solucionen de la següent manera: al actualitzar les dades d'un fitxer, no s'actualitzen ni les metadades ni les dades que hi ha escrites, sinó que escriuen noves versions a noves posicions amb una còpia del fitxer.

En tots els sistemes que hem vist fins ara, hi ha un únic vector d'i-nodes en un posició coneguda però a COW, el vector d'i-nodes s'emmagatzema a un fitxer (un bloc del disc) i hi ha un "meta-vector"en una posició fixa que apunta a la posició real dels i-nodes, així continuem sabent la posició de tots. Aquest sistema es pot veure com múltiples sistemes de fitxers.

Quan s'actualitza un fitxer de dades s'hi associa un nou i-node a la següent posició del meta-vector. Al nou i-node s'hi emmagatzemen les dades actulitzades del fitxer. Cada fitxer de dades té múltiples i-nodes i correspon a la nova versió de l'anterior. Així podem accedir a la versió anterior del i-node si en qualsevol moment hi ha un error en emmagatzemar el següent i-node.

[TODO: estructura de metafitxers (diapo 61)]

El meta-vector circular és el que ens permet emmagatzemar els següents i-nodes mitjançant rotacions, que es poden produir cada setmana, mes o quan el SO ho cregui oportú. Un cas concret d'aquest vector és el ZFS, que explicarem a continuació per deixar clar un cas real.

L'uberbock té 256 posicions. En engear l'ordinador ZFS escaneja l'array i comprova l'últim i-node associat i el checksum vàlid. Llavors rota periòdicament per tenir les versions anteriors del sistema del fitxers. També es gestiona que en comptes d'accessos aleatoris a disc, s'escriguin les noves versions en una sola escriptura contigua a disc, que és molt més òptim.

El **desaventatge** evident d'aquest sistema és el tamany extra que necessitem per mantenir diferents versions d'un fitxer.

3 Concurrència

Els programadors tendim a pensar de manera seqüencial, ja que és la manera més natural per nosaltres i com se'ns ha ensenyat sempre. Però en un ordinador hi succeeixen diversos esdeveniments de forma concurrent, a la vegada. Com a programadors podem aprofitar-nos d'aquestes característiques i manipular-les a la nostra voluntat.

Per executar processos concurrents necessitem que aquests es puguin comunicar entre ells. Tot i que cada procés té un espai de memòria independent, es poden comunicar mitjançant serveis del SO (canonades, mmaps, buffers, via xarxa...) que hem explicat a *Sistemes Operatius I*. En aquesta segona part però ens centrarem sobretot amb els **fils**:

3.1 Processos i Fils

A diferència de dos processos amb memòria independent, tots els fils d'un procés formen part del mateix procés. Això implica que **comparteixen l'espai de memòria:** per comunicar dos fils **no fan falta mètodes de comunicació interprocés**. Igual que els processos, cada fil té la seva pròpia pila i registres de la CPU corresponents.

Un procés pot tenir **un o múltiples fils**. A continuació els elements de cadascun:

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)

Processos o fils, la millor elecció depen del context i de la funció del programa. Per norma general els processos són més costosos, ja que la comunicació entre ells requereix de mecanismes específics i els canvis de context costen més al SO. Per altra banda, els fils s'acostumen a fer servir per diferents tasques d'un procés, això ens beneficia en:

- **Simplificació de Codi** pel tractament d'esdeveniments asíncrons.
- **Facilitat de Comunicació** entre els propis fils comparat amb els processos.

- **Estat propi per fil:** Cada fil té el seu estat (bloquejat, preparat o execució) cosa que millora el temps de resposta de la aplicació.

Les aventatges dels fils no són només en sistemes multiprocessador, en uniprocessador representen avantatges evidents a l'hora d'organització i seguretat dins les aplicacions.

3.2 Interferències en Processos i Fils

En l'standard *POSIX C* tenim diverses funcions principals pels fils

	Funció transparències
Crear fil	fil = thread_create(&funcio, arg);
Esperar fil	thread_wait(fil);

	Funció en llenguatge C (POSIX)
Crear fil	err = pthread_create(&fil, NULL, &funcio, arg);
Esperar fil	err = pthread_join(fil, NULL);

Altres funcions útils... i interessants!	
Finalitzar l'execució d'un fil	pthread_exit(void *retval)
Deixar d'executar un fil (forçar canvi de context)	pthread_yield()

Figura 1: *Funcions per fils de Posix*

Les funcions `pthread_create` i `pthread_join` creen un fil i fan que el fil principal esperi als altres fils. Llavors, poden múltples fils interferir-se entre sí afectant el correcte funcionament del programa?

A priori, com que cada fil té la seva pròpia pila, no s'interfereixen en la gran majoria dels casos. Hi ha problemes de sincronització quan intentem accedir (per escrpitura) a **variables globals** i a **variables compartides**.

Al estar gestionats pel SO, no podem controlar l'ordre en que un fil executa les intruccions que ha d'executar. Llavors, definim...

Definició 3.1. Una **Secció o Regió Crítica** són les parts del programa en què fils (o processos) accedeixen a variables compartides.

Si no gestionem bé els fils a les seccions crítiques en trobem en que el fil que arriba abans

és el que executa primer el que ha de fer, i com que ho gestiona el SO, pot ser que dos fils executin canvis a la vegada. El resultats són imprevisibles:

Definició 3.2. Les **Condicions de Carrera** o **Race Conditions** succeeixen quan no es controla l'ordre d'execució a una secció crítica.

A la sincronització correcta de les seccions cítiques se n'anomenena **Exclusió Mútua** o **Mútual Exclusion**

Per a donar un exemple més de problemes de sincronització, necessitem definir operació atòmica:

Definició 3.3. Una **Operació Atòmica** és aquella que no pot ser interrumpuda per un canvi de context.

Suposem les següents característiques d'una màquina uni-processador:

- Els tipus bàsics s'emmagatzemen a memòria i es llegeixen/escriuen amb operacions atòmiques. Aquestes es manipulen sent carregades a un registre atòmic, s'operen amb els registres de la CPU (no atòmics) i emmagatzemes el resultat a memòria (operació atòmica).
- Els registres i piles de cada fil es carreguen a memòria en cada canvi de context.

En aquestes dues situacions, quin valor té la variable `a`?

T	fil 1: operació a++	fil 2: operació a++
1	Carregar en registre la variable a	
2	Incrementar la variable a (=1)	
3	Desar en memòria la variable a	
4	Canvi de context	
5		Carregar en registre la variable a
6		Incrementar la variable a (=2)
7		Desar en memòria la variable a
8		

T	fil 1: operació a++	fil 2: operació a++
1	Carregar en registre la variable a	
2	Canvi de context	
3		Carregar en registre la variable a
4		Incrementar la variable a (=1)
5		Desar en memòria la variable a
6		Canvi de context
7	Incrementar la variable a (=1)	
8	Desar en memòria la variable a	

Doncs en el cas 1 valdrà 2, i en el 2 valdrà 1. La operació a++ és una secció crítica, així que s'ha de sincronitzar perque no hi hagin canvis de context quan no siguin possibles. Tot accés de fils a variables compartides porta a resultats no consitents si no estan ben tractats.

En un sistema amb les mateixes característiques que hem assumit però a més a més multiprocessador, no només els canvis de context donaran aquests problemes, ja que depen de com se sincronitzi la caché quan dos processadors n'estan fent canvis a la vegada.

3.3 Funcions Lock/Unlock

Volem controlar les seccions crítiques dels programes. Per fer-ho, linux ens deixa a la nostra disposició les funcions Lock i Unlock.

Usarem Lock i Unlock per entrar i sortir d'una secció crítica respectivament. Funcionen mijtançant **una clau**. Aquesta és la que permet bloquejar o desbloquejar la entrada de fils o processos a la secció crítica. Quan un fill entra amb la clau, la comanda bloquejarà a tots els fils exepte el que l'hi ha donat la clau, i quan el que era dins surt de la secció crítica amb la clau, la desbloqueja la pels altres. Llavors els fils que no havien pogut entrar fins ara competiran per qui entra primer.

Propietats. *Seccions Crítiques*

1. Les claus són úniques per secció crítica. Això vol dir que si tens dues seccions amb la mateixa clau, no són dues seccions crítiques diferents, són la mateixa. Amb n fils i la secció critica A a dos llocs del codi, si el fill k amb $1 \leq k \leq n$ bloqueja A, cap altre fil podrà entrar ni a la part que ha bloquejat aquest ni a l'altra, que no tenia cap fil dins.

2. Amb múltiples claus les seccions crítiques es tornen independents.

3. Les seccions es poden niuar sempre que siguin diferents. Seguiran complint les condicions 1. i 2.

4. Les seccions crítiques s'executen seqüencialment encara que disposem de mútliples nuclis.

5. Són crides a sistema, cal reduir-ne l'ús al necessàri.

3.4 Tipus de Bloquejos

A les funcions que ens permeten entrar i sortir d'una zona crítica en direm funcions de bloqueig. Segons els tipus de bloqueig podem classificar-los en dos tipus: espera activa i espera passiva.

3.4.1 Espera Activa

Els fils que han d'entrar a la secció crítica comproven contínuament (mitjançant CPU) si la clau s'allibera. Quan la zona s'allibera els fils competeixen per qui s'endú la clau. No són crides al SO, així que son molt menys costoses.

Aplicacions

Millor per les aplicacions amb pocs fills i amb poca espera (és a dir, que pel poc temps que és és més costós adormir-los). Molt usats en computació científica.

3.4.2 Espera Passiva

Els fils que no poden entrar a la secció crítica s'adormen a una cua associada a la clau, gestionat pel SO. Quan la clau s'allibera es desperten tots (o un) i competeixen per qui entrara a la secció. Implica una crida a SO.

Aplicacions

Millor per les aplicacions amb molts fills o si no sabem quanta estona pot un fill trigar a entrar a la secció crítica. S'usen molt més habitualment.

Tipus Espera	Molts Fils	Pocs Fils
Espera Incerta	Passiva	Provar
Poca Espera	Provar	Activa

3.5 Paradigmes de Programació Concurrent

En aquesta assignatura veurem els següents models de programació concurrent. Quan volguem dissenyar la nostra aplicació haurem de consultar quin paradigma s'ajusta millor a les nostres necessitats i aplicar-lo per solucionar problemes.

3.5.1 Paral·lelisme Iteratiu

El paral·lelisme iteratiu s'ha d'aplicar en programes que realitzen operacions iteratives amb *for* i *while*. Cada fil s'ocupa d'una part del bucle i després es combina el resultat al fil principal per obtenir la solució.

Exemples:

En un **producte de matrius** tindriem cada fil calculant una submatriu. Per exemple, amb dos fils podriem tenir el primer calculant la diagonal superior i l'altre la diagonal inferior. Llavors combinem les dues matrius al fil principal pel resultat.

En la **Suma dels valors d'un vector** la implementació seria fent que cada fil s'ocupi d'una part del vector i sumant el resultat de cada fil.

3.5.2 Paral·lelisme Recursiu

El paral·lelisme recursiu s'aplica quan un programa té una o més funcions **recursives** i treballen amb parts **independents** de les dades.

Tot i que intuitivament pugui semblar correcte, no hem de crear un fil a cada recursió ja que té un cost considerable (recordem que crerar un fil implica una crida a sistema). La solució a això és crear n fils i usar una **cua de tasques**. Cada fil agafarà les tasques a mesura que es vagi acabant les ja assignades, traient-les de la cua de fils.

Exemples:

L'algorisme quicksort. Cada nova partició de la llista serà processada per un fil. Quan hi hagi més particions que fils creats doncs s'haurà d'usar la cua de tasques per acabar-ho de computar.

3.5.3 Productos i Consumidors

Aquest paradigma s'ha d'usar si hi ha dues parts molt definides a l'algorisme. Una que genera dades i una altra que diu què s'han de fer amb aquestes dades.

Aquests algorismes s'implementen mitjançant buffers circulars. El productor hi escriu i el consumidor hi llegeix. Al ser circular ens hem d'assegurar que el punter dels escriptors mai superi el dels lectors, tot i que els escriptors generin les dades més ràpidament de la que el lector les llegeix i viceversa. S'hauran d'esperar mútuament.

Exemples:

La **gestió d'un servidor web** amb dispatcher (productor) i workers (consumidors).

La lectura d'un llistat de fitxers a processar (productor) i el seu processament (consumidor).Tame hem usat aquest paradigma a la pràctica 3 de *Sistemes Operatius I*.

3.5.4 Lectors i Escriptors

Aquest paradigma s'acostuma a aplicar en bases de dades, on els **lectors** volen accedir-hi per lectura i els **escriptors** per escriptura. El problema està en la gestió simulània dels dos.

Per evitar les interferències hem de garantir que:

1. Cada escriptor necessita accés **exclusiu** a la base de dades per escriure-hi. Si n'hi ha més d'un, el resultat sobre les dades serà impredictible.

2. Mentres no hi hagi cap escriptor, múltiples lectors poden accedir a la base de dades **simultàniament**. En cas de que hi hagi un escriptor, ningú hi pot estar exepte ell.

Exemples:

A la **programació orientada a objectes** es dóna sovint aquest paradigma amb les funcions get (lectors) i set (escriptors).

4 Gestió de l'Espera Activa

Anem a veure i discutir diferents implementacions d'algosrismes per gestionar una espera activa. Abans però, una sèrie d'axiomes:

- La variable **clau** és compartida entre tots els fils.
- Per entrar a la secció crítica s'ha de cridar la funció Lock
- Per sortir de la secció crítica s'ha de cridar la funció Unlock
- Apart de la secció crítica, cada fil té codi en el que no es produiran **Race Conditions**.

4.1 Algorisme 1

```
variables globals:
    boolean flag[2] = {false, false};

lock fil 0:
    while (flag[1]) {};
    flag[0] = true;
    return;

unlock fil 0:
    flag[0] = false;
    return;

lock fil 1:
    while (flag[0]) {};
    flag[1] = true;
    return;

unlock fil 1:
    flag[1] = false;
    return;
```

Figura 2: *Primera proposta amb Espera Activa*

No proveeix exclusió mútua. Suposem que el fil 1 no és a la secció crítica i el fil 0 hi vol entrar.

1. El fil 0 crida a *lock*, entra al while i n'executa la condició.
2. Com que el fill 1 no és a la secció crítica, se surt de seguida del while.
3. CANVI DE CONTEXT cap al fil 1 entre el while i el flag[0] = true.
4. El fil 1 vol entrar a la secció crítica: crida a lock, veurà que flag[0] = false (perque el fil 0 no ha pogut actualitzar el valor.) i hi entrarà.

5. CANVI DE CONTEXT cap al fil 0 amb el fil 1 dins la secció crítica. Això fa que el fil 0 entri a la secció crítica, ja que flag[1] = false. Ja tenim els dos fils dins la secció crítica.

4.2 Algorisme 2

El canvi principal és la inversió de les instruccions de la funció lock. En aquest cas hem arreglat la exclusió mútua, però ara podem tenir un deadlock.

```
variables globals:
    boolean flag[2] = {false, false};

lock fil 0:
    flag[0] = true;
    while (flag[1]) {};
    return;

unlock fil 0:
    flag[0] = false;
    return;

lock fil 1:
    flag[1] = true;
    while (flag[0]) {};
    return;

unlock fil 1:
    flag[1] = false;
    return;
```

Figura 3: *Segona proposta amb Espera Activa*

Definició 4.1. Un **Deadlock** és quan els dos fils es queden comprovant la condició d'entrada indefinidament.

Suposem que el fil 1 no és a la secció crítica i que el fil 0 hi vol entrar.

1. El fil 0 posa la seva variable a true.
2. CANVI DE CONTEXT al fil 1 entre la variable i el while.
3. El fil 1 vol entrar a la secció crítica. Posa la seva variable a true i es queda esperant a que el fil 0 surti de la secció crítica.
4. CANVI DE CONTEXT al fil 0 mentres el fil 1 espera.
5. El fil 0 entra al while i s'espera a que el fil 1 surti de la secció crítica. Ja s'ha produit un *deadlock*.

4.3 Algorisme de Peterson

L'algorisme de Peterson evita la excusió mútua i els deadlocks mitjançant la variable `victima` compartida entre els fils. Com es veu al codi, si es produeix un canvi de context entre victima i el while, l'altre fil canviarà altre cop el valor de víctima, així que quan es produexi un altre canvi de context, el fil original no podrà entrar, ja que el valor de victima no complirà la condició del while.

```
variables globals:
    int victima;
    boolean flag[2] = {false, false};

lock fil 0:
    flag[0] = true;
    victima = 0;
    while (flag[1] &&
      victima == 0) {};
    return;

unlock fil 0:
    flag[0] = false;
    return;

lock fil 1:
    flag[1] = true;
    victima = 1;
    while (flag[0] &&
      victima == 1) {};
    return;

unlock fil 1:
    flag[1] = false;
    return;
```

Figura 4: *Algorisme de Peterson*

L'únic problema amb Peterson és que no funciona als ordinadors acutals. El compliadors poden canviar l'ordre de les intruccions per ser més eficients, les CPU poden també fer-ho i depenent del protocol de sincronització de les cau pot fallar i per no mencionar que amb múltiples CPU també falla. Com ho podem arreglar doncs?

4.4 Algorisme Test-and-Set

Per solucionar els pproblemes de l'algorisme de Peterson hem de recórrer a la funció *Get-and-Set*: Aquesta funció té per argument una adreça de memòria. Emmagatzema el valor d'aquesta posició en un variable, es modifica el seu valor a true i es retorna el valor inicial. Tot això ho exectua de **forma atòmica** (bloquejant el bus), així que no pot ser destorbada per un canvi de context.

Implementant la funció *lock* amb només `while Get-and-Set(flag)` tenim un algorisme que satisfà l'exclusió mútua, no provoca deadlock i funciona amb mútliples fills. L'únic inconvenient d'aquest és que tots els fils competeixen per entrar a la secció crítica

en comptes de tenir un ordre o prioritat. Per solucionar-ho podem implementar l'algorisme següent, on s'assigna un nombre a cada fil únic per a cadaun. Llavors quan vulguin entrar a la secció crítica només ho podran fer si el torn_actual és el nombre assignat a l'algorisme.

```
variables globals:
    boolean flag[N] = {false, false, ..., false}; flag[0] = true;
    int torn_actual; /* el número que surt a pantalla */
    int torn = 0; /* el següent número al ticket */

lock:
    int torn_meu;
    <torn_meu = torn; torn++;>
    torn_meu = torn_meu % N;
    while (!flag[torn_meu]) {};
    torn_actual = torn_meu;
    return;

unlock:
    flag[torn_actual] = false;
    flag[(torn_actual + 1) % N] = true;
    return;
```

Figura 5: *Algorisme de Peterson*

El defecte més important d'aquest algorisme és que ha de tenir un nombre concret de fils definits al principi i immutable durant aquest.

5 Semàfors

El **Semàfors** són eines habitualment usades pel SO per sincronitzar l'accés a recursos compartits. També els podem usar en programes per gestionar la programació concurrent, facilitant el disseny de protocols de sincronització com productors-consumidors, lectors-escriptors...
Poden ser implementats tan amb espera activa com amb passiva.

En C, declarem un semàfor com `sem_t s` i disposem de les següents funcions.

1. **Inicialitzar:** `sem_open(...)` per processos i `sem_init(...)` per fils.
2. **Sincronitzar:** Tant processos com fils
   (a) `sem_wait(s)`: Per entrar a la secció crítica.
   (b) `sem_post(s)`: Per sortir de la secció crítica.
3. **Alliberar:** `sem_close(...)` per processos i `sem_destroy(...)` per fils.

Un semàfor es pot interpretar com un nombre natural, el qual s'augmentarà o disminuirà segons les funcions *sem_wait* i *sem_post*, que són les dues úniques funcions que existeixen per manipular semàfors. Totes les operacions que es realitzen dins d'aquestes funcions són operacions atòmiques (representades amb <>), per assegurar-ne els valors en sincronització.

1. `sem_post(s)` incrementa el valor de s
2. `sem_wait(s)` decrementa el valor de s si i només si és positiu. Si $s = 0$ el fil que ha cridat a la funció entra en espera fins que s tingui un valor positiu.

```
sem_wait(s):
    while (true) {
        <if (s > 0)
            s = s - 1;
            return;>
    }
```

```
sem_post(s):
    <s = s+1;>
    return;
```

Figura 6: *Definicions de Wait i Post*

Depenent dels valor que pugui pendre s distingirem entre dos tipus de semàfors: **Binaris** si el valor pot ser 0 o 1 o **Generals** si s pot tenir qualsevol valor natural. Aquests últims s'usen per limitar l'accés a determinats recursos.

5.1 Exclusió Mútua amb Semàfors

Per gestionar fils amb semàfors es fa amb un codi amb la següent estructura, consistint a envoltar la secció crítica per *sem_wait* i *sem_post* tal com es veu a la figura:

```
variables globals:
    sem_t s;

funcio_fil:
    codi independent per a cada fil
    sem_wait(&s);
    secció crítica
    sem_post(&s);
    codi independent per a cada fil

inicialització semàfor:
    sem_init(&s, 0, M); // inicialitzacio s = M
```

Figura 7: *Proposta d'ús de semàfors per exclusió mútua.*

Quan els fils no poden entrar a la secció crítica s'esperen (activa o passivament) a una cua associada. Quan surten de la secció crítica es desperten els que estaven esperant a la cua del semàfor *s*.

Si M és 1 vol dir que estem fent servir semàfors binaris, així que només un fil a la vegada pot entrar a la secció crítica. Si M és un altre nombre indica que M fils poden entrar a la secció crítica a la vegada.

5.2 Productors i Consumidors

Es tractaran algorismes de comunicació entre productors i consumidors mitjançant un buffer, on el productor entrega les dades al consumidor.

5.2.1 Buffer de Mida 1

Aquest codi l'implementarem amb un semàfor binari per indicar si el *buffer* està ple o buit respectivament semàfor ocupat o buit.

El **productor** comproven si el *buffer* és buit, entra a la secció crítica, copia la dada al *buffer* i avisa als consumidors que és ple.

El **consumidor** comproven si el *buffer* és ple, entra a la secció crítica, copia el *buffer* a una variable local i avisa als productors que és buit.

```
variables globals:
    typeT buffer;
    sem_t buit (=1), ocupat (=0);

productors:                          consumidors:
    typeT data;                          typeT data;
    while (true) {                       while (true) {
        produeix "data"                      sem_wait(&ocupat);
        sem_wait(&buit);                     copia "buffer" a "data"
        copia "data" a "buffer"              sem_post(&buit);
        sem_post(&ocupat);                   processa "data"
    }                                    }
```

Figura 8: *Productor i Consumidor Buffer mida 1*

5.2.2 Buffer de Mida M

Si canviem la mida del *buffer* per un de mida M, necessitarem implementar-ho amb dos semàfors generals: buit inicialitzat a M i ocupat inicialitzat a 0.

```
variables globals:
    typeT buffer[M];  int w = 0, r = 0;
    sem_t buit (=M), ocupat (=0);

productor:                           consumidor:
    typeT data;                          typeT data;
    while (true) {                       while (true) {
        produeix "data"                      sem_wait(&ocupat);
        sem_wait(&buit);                     copia "buffer[r]" a "data"
        copia "data" a "buffer[w]"           r = (r + 1) % M;
        w = (w + 1) % M;                     sem_post(&buit);
        sem_post(&ocupat);                   processa "data"
    }                                    }
```

Figura 9: *Productor i consumidor Buffer mida M*

Mitjançant els semàfors generals es controla quantes dades hi ha al *buffer*. Quan el productor pasa per *sem_wait(buit)* resta un de buit (M-1), i suma un a *sem_post(ocupat)*. Així el consumidor sabrà que té 1 dada a llegir quan entri al *sem_wait(ocupat)* i la sumarà a *sem_post(buit)*. Evidentment el productor no podrà entrar a la secció crítica si el buffer és ple i el consumidor si el buffer és buit.

5.2.3 Múltiples Productors i Consumidors

A diferència de l'anterior, aquí necessitem controlar que només un productor i consumidor a la vegada puguin escriure/llegir a buffer: això ho farem amb dos semàfors nous, quedant-

nos amb un total de 4 semàfors!

```
variables globals:
    typeT buffer[M];  int w = 0, r = 0;
    sem_t buit (=M), ocupat (=0);
    sem_t clauProd (=1), clauCons (=1);

productor:
    typeT data;
    while (true) {
        produeix "data"
        sem_wait(&buit);
        sem_wait(&clauProd);
        copia "data" a "buffer[w]"
        w = (w + 1) % M;
        sem_post(&clauProd);
        sem_post(&ocupat);
    }

consumidor:
    typeT data;
    while (true) {
        sem_wait(&ocupat);
        sem_wait(&clauCons);
        copia "buffer[r]" a "data"
        r = (r + 1) % M;
        sem_post(&clauCons);
        sem_post(&buit);
        processa "data"
    }
```

Figura 10: *Productors i Consumidors amb Buffer mida M*

Els semàfors *clauProd* i *clauCons* s'usen per assegurar la exclusió mútua. És a dir, que més d'un productor no pugui escriure a la vegada que un altre, provocant race conditions (anàlogament amb els consumidors). Els altres dos semàfors s'usen per saber quantes dades hi ha al buffer exactament igual que a la implementació anterior.

5.3 Múltiples Lectors i Escriptors

Per donar una solució a aquest problema, hem de tenir en compte que cada escriptor necessita accés exclusiu a la base de dades a escriure-hi i, si no hi ha cap escriptor, múltiples lectors hi poden accedir a la vegada. De moment donem una primera implementació:

```
variables globals:
    sem_t rw (=1);

lectors:
    while (true) {
        sem_wait(&rw);
        llegeix dades
        sem_post(&rw);
        processa dades
    }

escriptors:
    while (true) {
        processa dades
        sem_wait(&rw);
        escriu dades
        sem_post(&rw);
    }
```

Figura 11: *Algorisme injust lectors-escriptors*

Només tenim un escriptor modificant les dades tal i com hem mencionat, però no tenim

múltiples lectors accedint a les dades. Per aconseguir que múltiples puguin entrar a la vegada hem d'afegir un altre semàfor *clauLec* de la següent manera:

```
variables globals:
    int nr = 0;
    sem_t rw (=1), clauLec (=1);

lectors:
    while (true) {
        sem_wait(&clauLec);
        nr = nr + 1;
        if (nr == 1) sem_wait(&rw);
        sem_post(&clauLec);
        llegeix les dades
        sem_wait(&clauLec);
        nr = nr - 1;
        if (nr == 0) sem_post(&rw);
        sem_post(&clauLec);
        processa dades
    }

escriptors:
    while (true) {
        processa dades
        sem_wait(&rw);
        escriu dades
        sem_post(&rw);
    }
```

Figura 12: *Algorisme just lectors-escriptors*

En aquest cas tots els lectors llegeixen la mateixa dada. L'algorisme funciona de la següent manera (assumim que ja tenim dades escrites, osigui $rw = 1$)

1. El primer lector decrementra *clauLec* a 0, evitant que ningú més entri a la primera zona crítica, augmentant nr en 1 (nr = nombre readers).

2. Si només hi ha un lector, comprova si hi ha dades a llegir (*sem_wait(rw)*). Si no hi ha dades, espera fins que n'hi hagi.

3. Si n'hi ha ($rw = 1$), decrementa la clau, cosa que preven un escriptor escriure i desbloqueja el *clauLec*, fent que tots els altres lectors vagin entrant.

4. Quan acaba de processar les dades, decrementa el semàfor (*sem_post(clauLec)*) i baixa el nombre de lectors, ja que un lector ha acabat. Llavors es torna a incrementar el semàfor.

5. l'últim fil que passi per la segona zona crítica és l'encarregat de augmentar rw a 1 altre cop, dient al escriptor que pot tornar a escriure dades.

Tot i que múltiples escriptors hi puguin escriure, hi ha un biaix cap els lectors. Aquests podran accedir sempre abans que els escriptors. Per proposar una solució a aquest problema amb semàfors seria molt complex, però la donarem amb **Monitors** manera amb la qual és molt senzilla d'implementar.

6 Monitors

Tal com els semàfors, els Monitors serveixen per sincronitzar fils i processos, però amb molta més llibertat. Això es per la existència de moltes més funcions per controlar la entrada i la sortida adormint o despertant fils amb molta més llibertat que amb els semàfors.

En C, un semàfor es declara com `pthread_mutex_t mutex` i s'inicialitza per defecte amb `pthread_mutex_init`, on mutex és la clau (mutex = mutual exclusion). La variable es fa servir per entrar o sortir de la secció crítica i inclou una cua sobre la qual els fils s'adormen.

Per entrar i sortir d'una secció crítica:

- `pthread_mutex_lock(mutex)`: Agafa el bloqueig de la clau i s'usa perque un fil entri a secció crítica. Només hi pot entrar un fil; la resta esperen adormits a la cua.
- `pthread_mutex_unlock(mutex)`: Allibera el bloqueig de la clau i s'usa per sortir de la zona crítica.

Els mutexs (al contrari dels semàfors) només poden ser **binaris**.

També tenim funcions adicionals per adormir o despertar els fils d'una cua quan vulguem. PEr accedir a la cua d'un mutex ho farem amb `pthread_cond_t cond` com a variable condicional (si es compleix x condició y fils es desperten).

Tenim les següents funcions per operar sobre la llista. És imprescindible tenir agafada la clau mutex per cridar-les:

- `pthread_cond_wait(cond, mutex)`: el fil que ho executa s'adorimrà al final de la cua i alliberarà la clau.
- `pthread_cond_signal(cond)`: es despertarà el primer fil de la cua cond. Haurà d'agafar la clau *mutex* per continuar executant. Si aquesta ja està agafada, haurà d'esperar que s'alliberi.
- `pthread_cond_broadcast(cond)`: desperta a tots els fils de la cua i es posaran a competir per agafar la clau mutex. Només un ho aconseguirà. La resta es posarà a dormir al mutex, no a la cua cond.

Aquestes funcions només s'han de cridar dins de la secció crítica. No fer-ho pot portar a comportaments imprebisibles.

Ara veurem diferents algorismes per fer amb monitors.

6.1 Semàfor amb Monitors

Per implementar la funcionalitat d'un semàfor amb protocols d'entrada/sortida amb mutex tenim el codi següent:

```
variables globals:
    pthread_mutex_t mutex;  pthread_cond_t cond;
    int s = M; // semàfor general

sem_wait:
    pthread_mutex_lock(&mutex);
    while (s == 0) {
      pthread_cond_wait(&cond, &mutex);
    }
    s = s - 1;
    pthread_mutex_unlock(&mutex);

sem_post:
    pthread_mutex_lock(&mutex);
    s = s + 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
```

Figura 13: *Funcionalitat de semàfor implementada amb monitors*

Tal i com amb els semàfors, podem limitar l'accés dels fils a la secció crítica, aquí hi pot haver un màxim d'M fils a dins la de pròpia secció.

La Importància del While

Notis que al codi les condicions es comproven amb **while** no amb un if. Això és perque cal continuar comprovant la condició $s = 0$ per el fil que tenim. Si ho féssim amb un if, un altre fil prodria colar-se i propiciar un funcionament no desitjat.

És a dir, que per comprobar una condició que típicament fariem amb un if, per exemple:

```
if (condicio){wait(cond, mutex)}
```

generalment no funcionarà. Quan un fil es desperta d'un wait, s'ha de tornar a comprovar la condició que el va fer adormir:

```
while (condicio){wait(cond, mutex)}
```

Practicament tots els algorismes que veurem amb un mutex fan servir whiles, per garantir que les condicions es compleixen.

6.2 Barrera

Definició 6.1. Una **Barrera** és un mecanisme de sincronització de fils. Permet que fils esperin a que altres fils acabin de fer els seus processos.

El concepte sobre barrera és que el primer fil acaba la seva part del processament i crida a la funció barrera i s'adorm a la cua *cond* esperant a que la funció acabi. El darrer fil que crida a la barrera ha de despertar a tots els altres per poder continuar el programa.

```
variables globals:
    pthread_mutex_t mutex;  pthread_cond_t cond;
    int comptador = N; // nombre de fils

barrera:
    pthread_mutex_lock(&mutex);
    comptador = comptador - 1;
    if (comptador != 0) {
        pthread_cond_wait(&cond, &mutex);
    } else {
        comptador = N;
        pthread_cond_broadcast(&cond);
    }
    pthread_mutex_unlock(&mutex);
```

Figura 14: *Implementació barrera amb mutex*

La variable comptador és el nombre de fils que la barrera haurà d'esperar. Cada vegada que un fil arriba a la barrera, el codi resta a *comptador* 1. Quan *comptador* = 0, es fa un broadcast dels fils que desbloquejarà tots els que estiguessin a la cua. Com que el fill que fa la crida a broadcast manté la clau, serà el primer que la alliberarà, deixant que tots els altres la vagin agafant a mesura que aquesta quedi lliure.

6.3 Productors i Consumidors

Recorem que amb la implementació dels semàfors teniem 4 semàfors: clauProd i clauCons amb els que controlavem que només hi hagués un fil a la secció crítica i buit i ocupat que controlen que el nombre de dades que hi ha dins del buffer.

A diferència dels semàfors només ens fa falta un sol mutex i la gestió de dues cues dels fils, les dels productors i els consumidors. Assumim que les dades ja estan totes generades.

A mesura que el productor va generant la data, aquesta va guardant-se al buffer. Quan hi ha una dada dins del buffer, s'envia un senyal a consumidor per a que copii a memòria

el contingut del buffer. Com sap quantes dades pot consumir? Gràcies a comptador, que el productor augmenta quan hi ha dades noves i el conumidor el decrementa quan les ha tret. És a dir, comptador respresenta quantes dades hi ha escrites al buffer (el que es feia amb semàfors generals a la implementació en semàfors). Els dos whiles que hi ha (un per productors i l'altre per consumidors) controlen que s'hi pugui escriure si el buffer no està ple o no està buit respectivament.

```
variables globals:
    typeT buffer[M];  int w = 0, r = 0;
    int comptador = 0; // nombre d'elements ocupats
    pthread_mutex_t mutex;   pthread_cond_t condP, condC;
productor:                          consumidor:
    while (true) {                      while (true) {
      generar "data"                      lock(&mutex);
      lock(&mutex);                       while (comptador == 0) {
      while (comptador == M) {              wait(&condC, &mutex);
          wait(&condP, &mutex);           }
      }                                   copia "buffer[r]" a "data"
      copia "data" a "buffer[w]"          r = (r + 1) % M;
      w = (w + 1) % M;                    comptador--;
      comptador++;                        signal(&condP);
      signal(&condC);                     unlock(&mutex);
      unlock(&mutex);                     consumir "data"
    }                                   }
```

Figura 15: *Productors i Consumidors Mida M amb Mutex*

6.4 Lector i Escriptors amb Preferència

En el següent algorisme el lectors tenen preferència sobre els escriptors. Quan un escriptor vulgui escriure, haurà d'esperar-se a que tots els lectors acabin de llegir. Al contrari que amb semàfors, tornem a necessitar només un mutex i la cua en comtpes de dos semàfors.

```
variables globals:
    int nr = 0;   boolean w = false;
    pthread_mutex_t mutex;   pthread_cond_t cond;
read_lock:                          write_lock:
    lock(&mutex);                       lock(&mutex);
    while (w) {                         while (nr > 0 || w) {
        wait(&cond, &mutex);                wait(&cond, &mutex);
    }                                   }
    nr++;                               w = true;
    unlock(&mutex);                     unlock(&mutex);
read_unlock:                        write_unlock:
    lock(&mutex);                       lock(&mutex);
    nr--;                               w = false;
    if (nr == 0) broadcast(&cond);      broadcast(&cond);
    unlock(&mutex);                     unlock(&mutex);
```

Figura 16: *Lectors-escriptors injust amb mutex*

La variable *nr* és el nombre de lectors, que s'incrementa o decrementa a mesura si n'hi ha més o menys llegint. La variable booleana *w* és qui controla si un lector està escrivint o no. Si és falsa vol dir que hi ha escriptors escrivint, per tant els lectors hauran d'esperar.

El funcionament és senzill. Quan un fil escriu ($w = true$) els lectors entren al *while* i s'esperen a *wait*. Quan acaba retorna $w = false$.
En canvi, si tenim lectors escrivint ($nr>0$), el *while(nr>0 // w)* farà que els escriptors parin al *wait* fins que $nr=0$, per això diem que dona prioritat als lectors.

Quan al fer un *broadcast* farem que els fils que hi hagi a la cua quedin sobre el semàfor esperant a que la clau es lliberi. El cas és que en el read només hi poden haver escriptors, mentres que en el write hi poden haver tan lectors com escriptors.

6.5 Lectors i Escriptors Justos

Aquest algorisme no n'hem donat una implementació amb semàfors per la seva dificultat. En canvi amb mutexs es pot fer senzillament de la següent manera:

```
variables globals:
    int nr = 0;   boolean w = false;
    pthread_mutex_t mutex;   pthread_cond_t cond;

read_lock:
    lock(&mutex);
    while (w) {
        wait(&cond, &mutex);
    }
    nr++;
    unlock(&mutex);

read_unlock:
    lock(&mutex);
    nr--;
    if (nr == 0) broadcast(&cond);
    unlock(&mutex);

write_lock:
    lock(&mutex);
    while (w) {
        wait(&cond, &mutex);
    }
    w = true;
    while (nr != 0) {
    }   wait(&cond, &mutex);
    unlock(&mutex);

write_unlock:
    igual que abans
```

Figura 17: *Algorisme just lectors-escriptors amb mutex*

El canvi que fa que sigui just és el separar les condicions del while. En aquesta implementació si un fil està esperant perque ja hi ha un fil escrivint (*while(w)*) o perque hi ha lectors llegint *while(nr != 0)* és diferent. La variable w s'usa per saber si **hi ha un escriptor que vol escriure**, no si un fil està escrivint.

A l'engròs, si un lector vol llegir, ha de comprovar si un escriptor vol escriure abans. Si w = true, s'adorm al while de write_lock(). De la mateixa manera, si hi ha un lector llegint, l'escriptor s'adorm fins que acaba (al segon *while* de while_lock).