

# Introducció a la concurrència

Sistemes Operatius 2

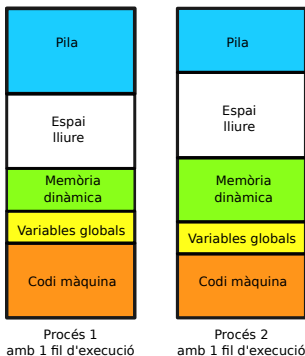
Grau d'Enginyeria Informàtica

## Concurrència

- Al món real es realitzen múltiples activitats de forma concurrent: algunes persones van en bici, d'altres amb el monopatí, d'altres en cotxe, d'altres juguen amb el mòbil, d'altres prenen un cafè...
- En un ordinador també tenen lloc múltiples esdeveniments de forma concurrent: un procés s'executa a una CPU, es llegeixen dades de disc, s'envien dades per la xarxa, ...
- Com a programadors acostumem a pensar de forma seqüencial. Ha arribat el moment de pensar que múltiples tasques es poden realitzar de forma concurrent!

# Introducció: concurrència entre processos

- Cada **procés** té un espai de **memòria independent**. Els processos no poden escriure a l'espai de memòria d'un altre procés. Veure `fork_variables.c`.
- Per **comunicar-se** entre sí cal fer servir els **serveis dels sistema operatiu**. Per exemple, escriure a un fitxer compartit o comunicar-se via xarxa.



# Introducció: concurrència entre processos

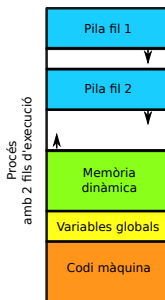
Vegeu exemple `fork-anon-mmap.c`.

- Els processos es comuniquen mitjançant un **fitxer anònim** mapat a memòria. Què estem fent aquí?
- La comunicació es realitza entre dos processos amb una relació pare-fill. Com és que això és factible, essent els dos processos independents?

# Introducció: els fils

El sistema operatiu ens ofereix un altre servei perquè els “processos” es puguin comunicar: **els fils**! Veure codi `fils_vector.c`.

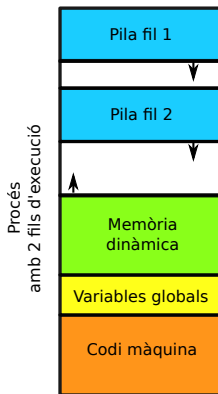
- Els **fils** formen part del mateix procés: **comparteixen l'espai de memòria** del procés!
- Per **comunicar-se** entre sí poden fer servir l'espai de memòria que comparteixen<sup>1</sup>.



<sup>1</sup>Estrictament, la comunicació per fils no entra dintre de l'àmbit de la comunicació inter-procés

# Introducció: els fils

Cada fil té la seva **pròpia pila i registres de la CPU**: cada fil pot executar una part diferent o la mateixa de l'aplicació.



```
int global;  
  
void funcio()  
{  
    int a;  
  
    // Això ho executen els dos fils  
}  
  
void main(...)  
{  
    int fil;  
  
    // Això ho executa un sol fil, una sola pila  
  
    fil = crear_fil_nou(&funcio); // Retorna id fil creat  
  
    // Això ho executa un sol fil, hi ha dues piles  
  
    funcio();  
  
    esperar_que_acabi_fil(fil);  
}
```

# Introducció: processos i fils

Més formalment... un fil (*thread*, en anglès) és la unitat d'execució d'un procés. Un procés pot tenir un o múltiples fils.

- Elements d'un procés
  - **Espai d'adreces** (codi executable, variables globals, ...)
  - Fitxers oberts
  - La **llista de fils del procés**
  - Altres
- Elements propis de cada fil
  - **Comptador de programa**
  - **Registres de la CPU**
  - **Pila**
  - **Estat** (preparat, bloquejat, execució)

# Introducció: interferència a la comunicació

Poden interferir-se entre sí els processos i/o fils en comunicar-se?

Executar el codi `fil_soma.c` amb diferents valors d'iteracions...

- Què surt?
- Quines variables estan compartides?

Executar el codi `fork_anon_mmap_soma.c` amb diferents valors d'iteracions...

- Què surt?
- Per què cal fer servir un fitxer anònim mapat a memòria?



# Objectiu: programació concurrent entre processos i fils

De forma general, l'objectiu en aquest tema és

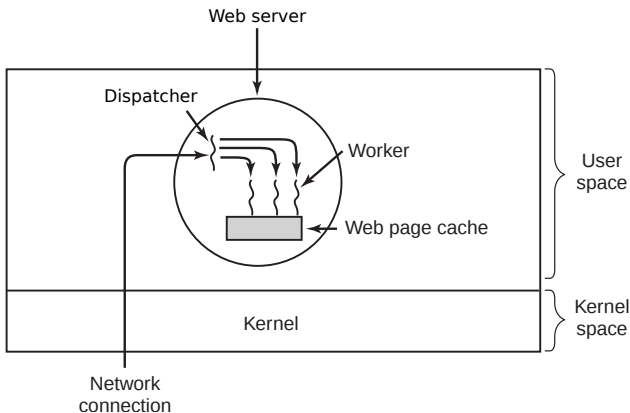
- 1 Aprendre a programar de forma **concurrent** fent servir múltiples processos i/o fils.
- 2 **Sincronitzar** correctament la comunicació entre ells. El sistema operatiu ens ofereix serveis (crides a sistema) per assegurar que diversos processos i/o fils no s'interfereixin entre sí.

Utilitzarem el llenguatge C, tot i que els conceptes i idees que es veuran són vàlids també a altres llenguatges com Java, Python, ... que també suporten concurrència.

# Introducció: processos versus fils

Volem dissenyar una aplicació, per exemple un servidor web o un navegador web...

- Quin disseny és millor: múltiples processos o múltiples fils?



# Introducció: processos versus fils

## Ens els processos

- Cada procés té el seu propi espai d'adreces. La comunicació entre processos es realitza mitjançant serveis que ofereix el sistema operatiu i requereix més “recursos” (és més costós) que els fils.
- Els processos són més “pesats” que els fils: és més costós crear-los així com fer canvis de context entre processos.

## En els fils

- Els fils comparteixen l'espai d'adreces d'un procés. La comunicació es realitza amb les variables que comparteixen i és menys costosa.
- Els fils es poden considerar un procés “lleuger”: no costa tant crear-los i el canvi de context entre fils és menys costós.
- Si un fil fa una operació invàlida, es mata tot el procés.

# Introducció: processos versus fils

Els fils s'acostumen a utilitzar per realitzar les diferents tasques d'un procés

- **Simplificació del codi** per tractar amb events asíncrons (ratolí, teclat, xarxa, ...)
- **Facilitat per comunicar** els fils entre sí (respecte fer-ho amb processos).
- Cada fil té el seu **propri estat** (bloquejat, preparat, execució). Es pot millorar doncs el temps de resposta de l'aplicació.

Atenció!

- No associar programació multifil amb sistemes multiprocessadors.
- Les avantatges de la programació multifil són evidents també amb sistemes uniprocessador.

# Introducció: processos versus fils

Google es va trobar amb aquest dilema en desenvolupar el Chrome... processos o fils?

- Cada tabulador requereix realitzar un munt de tasques: comunicar-se amb el servidor per baixar text, vídeo, àudio, ... i renderitzar-ho tot per mostrar-ho a pantalla.
- Van decidir que cada tabulador fos un procés. Si a un tabulador es produïa un problema no afectaria a la resta.
- A més, la resposta (interactiva) millorava: si un procés (tabulador) és inactiu, el sistema de memòria virtual el pot desar a disc. Això és més difícil amb els fils ja que és més difícil separar els tabuladors actius dels inactius.

# Conceptes bàsics: els fils

Suposem dues funcions, `thread_create` i `thread_wait`, per crear i esperar a un fil<sup>2</sup>.

```
int global;

void funcio(int *a)
{
    // ...
}

void main(...)
{
    int fil;

    // ...

    fil = thread_create(&funcio, NULL);

    // ...

    thread_wait(fil);
}
```

## La funció `thread_create`

- Els paràmetres són la funció d'entrada del nou fil i un paràmetre a passar.
- El nou fil començarà a executar a partir de la seva funció d'entrada com si fos el seu “main”.
- El fil creat mor en acabar l'execució de la seva funció d'entrada.

---

<sup>2</sup>Aquestes no són pas les funcions “reals” en C

# Conceptes bàsics: els fils

Suposem dues funcions, `thread_create` i `thread_wait`, per crear i esperar a un fil<sup>3</sup>. Què fem amb `pthread_create`?

Observacions:

```
int global;

void funcio(int *a)
{
    // ...
}

void main(...)
{
    int fil;

    // ...

    fil = thread_create(&funcio, NULL);

    // ...

    thread_wait(fil);
}
```

- No hi ha relació pare-fill entre els fils creats i no hi ha un fil més “important” que altre per al SO (tot i que sovint es parla de fil principal i secundari).
- El fil que crea el nou fil retorna de la funció amb l'id del fil creat i continua l'execució.
- Quan el fil principal acaba el main, es mor el procés.

---

<sup>3</sup>Aquestes no són pas les funcions “reals” en C

## Què fem amb `pthread_wait`?

```
int global;

void funcio(int *a)
{
    // ...
}

void main(...)
{
    int fil;

    // ...

    fil = thread_create(&funcio, NULL);

    // ...

    thread_wait(fil);
}
```

## La funció `thread_wait`

- Permet que un fil esperi que un altre finalitzi.
- Si el fil al qual es vol esperar ha acabat abans de cridar a `thread_wait`, aquesta funció retorna de seguida.



# Conceptes bàsics: les funcions en POSIX C

Quines són les funcions POSIX C (principals) que hi ha disponibles?

	Funció transparències
Crear fil	<code>fil = thread_create(&amp;funcio, arg);</code>
Esperar fil	<code>thread_wait(fil);</code>

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

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

# Conceptes bàsics: interferència entre processos i fils

Hem vist les funcions bàsiques per manipular fils. En aquests moments sabem

- Crear processos
- Crear fils

Com poden múltiples processos interferir-se entre sí? Com poden múltiples fils interferir-se entre sí?

El sistema operatiu ens ofereix els serveis necessaris per assegurar que múltiples processos i/o fils es puguin comunicar de forma correcta entre sí.

# Conceptes bàsics: interferència entre fils

Cada fil té la seva pròpia pila i, per tant, fils diferents poden executar el mateix codi sense interferir-se entre sí!

```
int imprimir(int a)
{
    int b;
    b = a + 1;
    printf("Valor %d\n", b);
    return b;
}
```

Aquest codi pot ser executat per múltiples fils a la vegada sense interferència

- Les variables a i b es guarden a la pila, i cada fil té una pila diferent.
- El valor de retorn també es guarda a la pila.



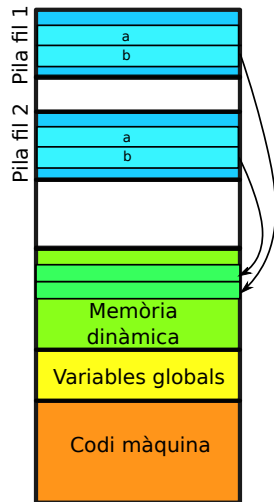
# Conceptes bàsics: interferència entre fils

Cada fil té la seva pròpia pila i, per tant, fils diferents poden executar el mateix codi sense interferir-se entre sí!

```
int *processar(int a)
{
    int i, *b;
    b = malloc(sizeof(int) * a);
    for(i = 0; i < a; i++)
        b[i] = 0;
    return b;
}
```

Aquest codi pot ser executat per múltiples fils a la vegada sense interferència

- En fer malloc es retorna una zona diferent per a cada fil.
- Compte en no escriure fora del rang del vector!



# Conceptes bàsics: interferència entre fils

Quan hi pot haver problemes? En accedir (per escriptura) a **variables globals** o, en general, a variables compartides.

```
int global;

void funcio(int *a)
{
    global = 2;
}

void main(...)
{
    int fil;

    global = 0;

    fil = thread_create(&funcio, NULL);

    if (global == 0)
        global = 1;

    // quin valor tindra global ?

    thread_wait(fil);
}
```

Quin és el valor de `global` després de l'`if`? No se sap! Cal tenir en compte que

- El sistema operatiu és el que planifica els fils. Nosaltres no ho controlem.
- Ens faran falta funcions per sincronitzar els fils (i.e. controlar l'ordre en què s'executen el fils)

# Conceptes bàsics: interferència entre fils

Quan hi pot haver problemes? En accedir (per escriptura) a variables globals o, en general, **variables compartides**.

```
void funcio(int *a)
{
    a[0] = 1;

    // ...
}

void main(...)
{
    int i, fil, a[5];

    for(i = 0; i < 5; i++) a[i] = 0;
    fil = thread_create(&funcio, a);

    if (a[0] == 0)
        a[0] = 2;

    // quin valor tindra a[0] ?

    thread_wait(fil);
}
```

- El segon argument de `thread_create` permet passar una variable al fil que es crea.
- Aquí la variable `a` estarà compartida entre els dos fils. No cal que sigui una variable global. Tindrem el mateix problema que abans.

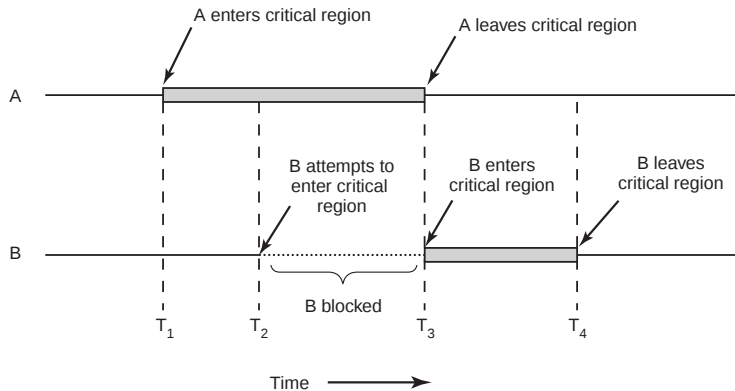
# Conceptes bàsics: seccions crítiques

Tenir en compte que:

- Els fils (o processos) poden realitzar operacions de forma independent si assegurem que les seves variables són independents entre sí.
- Aquelles parts del programa que en què els fils (o processos) **accedeixen** a variables compartides s'anomenen **regions crítiques** o **seccions crítiques**.
- Quan diversos fils (o processos) poden accedir per lectura o escriptura a zones de memòria compartida i no es controla l'ordre d'execució el resultat pot ser impredecible. Aquesta situació se l'anomena **condició de carrera** (*race coditions*), veure codis `fitxer_genera.c` i `fitxer_lectura.c`.

# Conceptes bàsics: exclusió mútua

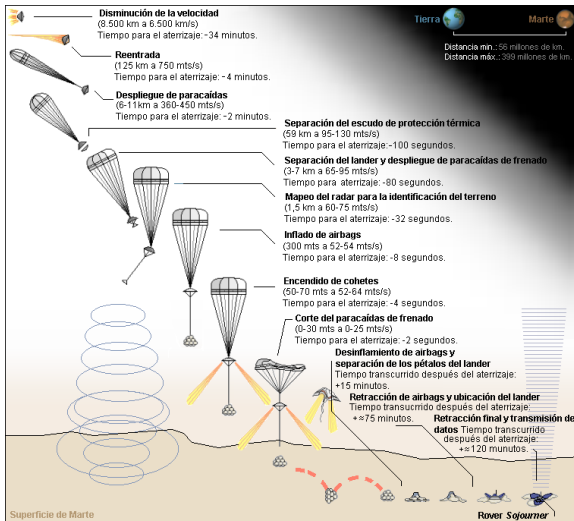
Cal sincronitzar els fils i/o processos entre sí mitjançant l'**exclusió mútua** (*mutual exclusion*): fils i/o processos diferents no han de poder accedir a la mateixa secció crítica al mateix instant de temps.





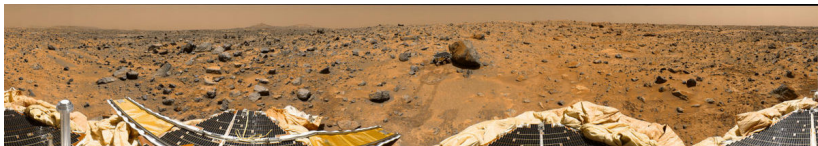
# Sincronitzar múltiples fils no és senzill: Mars Pathfinder

La Mars Pathfinder va aterrar al juliol del 1997 a Mart.



# Sincronitzar múltiples fils no és senzill: Mars Pathfinder

Va ser la primera missió en enviar un vehicle tot terreny a Mart



- Poc després de l'aterratge i que es comencessin a capturar dades, l'estació espacial va experimentar "resets" aleatoris del sistema.
- La premsa deia que l'ordinador estava rebent masses dades...
- Però el problema real era degut a un problema de sincronització de fils!

# Exclusió mútua: motivació

Recordem que els fils (i processos) poden interferir-se entre sí:

```
#define ITERATIONS 1000
```

```
int a;
```

```
void *thread_fn(void *arg)
```

```
{  
    int i;  
    for(i = 0; i < ITERATIONS; i++)  
        a++;  
}
```

```
    return ((void *)0);
```

```
}
```

```
int main(void)
```

```
{  
    pthread_t ntid[2];  
    int i;
```

```
    a = 0;
```

```
    for(i = 0; i < 2; i++)  
        pthread_create(&(ntid[i]), NULL, thread_fn, NULL);
```

```
    for(i = 0; i < 2; i++)  
        pthread_join(ntid[i], NULL);
```

```
    printf("Valor d'a: %d\n", a);
```

```
}
```

Codi `fils_suma.c`

❶ Compilar el codi amb

```
$ gcc fils_suma.c -o  
fils_suma -lpthread
```

❷ Executar amb diversos valors de `ITERATIONS`. Quin valor s'imprimeix per pantalla?

Una **operació atòmica** és aquella que no pot ser interrompuda per un canvi de context.

Suposarem les següents característiques (realistes) de les màquines:

- Els tipus bàsics (int, float, double, ...) s'emmagatzemen a memòria i es poden llegir i escriure amb operacions atòmiques.
- Les variables es manipulen carregant aquestes a un registre (atòmic), operant sobre elles utilitzant els registres (no necessàriament atòmic) i emmagatzemant el resultat un altre cop a memòria (atòmic).
- Cada fil té els seus propis registres i la seva pròpia pila. Aquests es carreguen en memòria a cada canvi de context.

# Operacions atòmiques

Suposem un sistema **uni-processador**. Al codi que hem vist abans:

- Suposem  $a = 0$ .
- A la taula es mostren les operacions atòmiques.
- Nosaltres hem suposat que s'executa d'aquesta forma (que dóna el resultat correcte).

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		

# Operacions atòmiques

Suposem un sistema **uni-processador**. Al codi que hem vist abans:

- Suposem  $a = 0$ .
- Si dos fils fan  $a++$ , es possible que el resultat no sigui 2?

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 $a$	

A l'exemple anterior:

- El resultat de realitzar `a++` per 2 fils pot ser 1 o 2, depenent d'on es faci el canvi de context.
- L'operació `a++` és una **regió crítica**: una operació a què accedeixen (per escriptura) múltiples fils a la vegada.
- En general, l'accés concurrent de múltiples fils a variables compartides (l·listes, ...) pot portar a un resultat no consistent.

## Atenció!

- En **sistemes multiprocessador** no cal que es doni un canvi de context perquè es produeixi aquest problema. Tot depèn del “protocol” que es faci servir per mantenir sincronitzada la memòria cau (cache).

T	fil 1, <b>processador 1</b> : operació a++	fil 2, <b>processador 2</b> : operació a++
1	Carregar en registre la variable a	<b>Veu el bus bloquejat</b>
2	Incrementar la variable a (=1)	Carregar en registre la variable a
3	Desar en memòria la variable a	Incrementar la variable a (=1)
4		Desar en memòria la variable a



# Operacions atòmiques

Suposeu que la vostra màquina una CPU i dos fils que executen aquestes instruccions. Quins valors poden imprimir-se per pantalla?

variables globals:

```
int x = 0; int y = 0; int z = 0;
```

fil1:

```
x = 1; y = 2;
```

fil2:

```
z = x + y;
```

```
printf("El valor de z es %d\n",z);
```

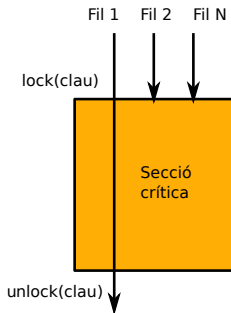
# Funcions lock i unlock

## Objectiu

- Assegurar exclusió mútua a les seccions crítiques entre múltiples fils i/o processos.

## Com ho farem?

- Disposarem de dues funcions, que aquí anomenarem **lock** i **unlock**, utilitzades per entrar i sortir de la regió crítica.

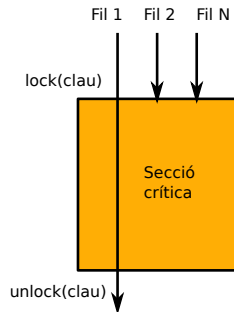


- Les funcions lock i unlock tenen una variable, la **clau**, compartida entre els fils/processos.
- Funció **lock(a)**: agafar la clau "a" (només un fil/procés pot agafar-la).
- Funció **unlock(a)**: alliberar la clau "a" (notificar que se surt de la secció crítica).

# Funcions lock i unlock

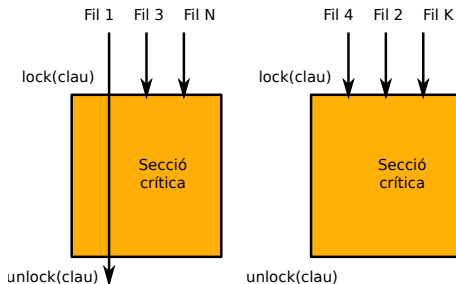
## Idea intuïtiva

- Suposem que el fil 1 crida a la funció lock. Si no hi ha cap fil a la secció crítica, la funció lock retorna i el fil 1 pot entrar-hi.
- A continuació arriba el fil 2 i l'N. Aquests criden també a la funció lock i es bloquegen.
- En sortir el fil 1 de la secció crítica aquest crida a la funció unlock i allibera la clau.
- Els fils 2 i N competeixen per adquirir la clau i entrar a la secció crítica. Només un ho aconseguirà. L'altre es bloqueja un altre cop.



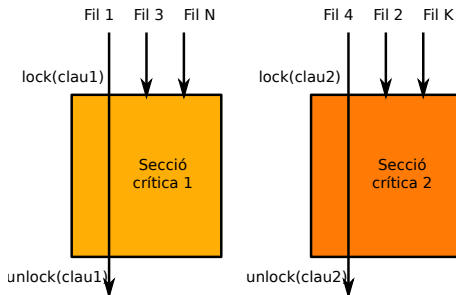
# Protecció de seccions crítiques

Parts diferents del codi es poden protegir amb una mateixa clau (a la secció crítica s'accedeixen a les mateixes variables compartides).



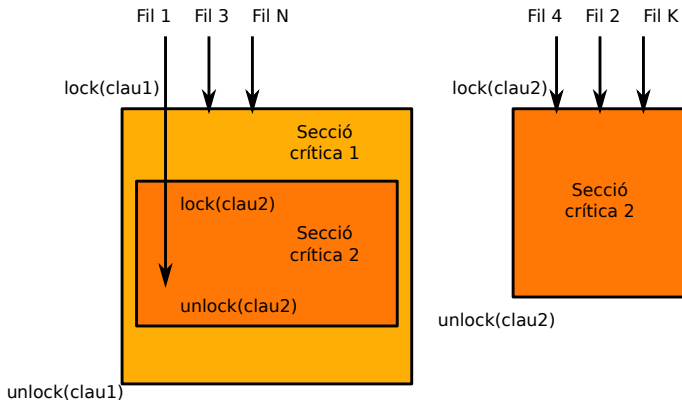
# Protecció de seccions crítiques

Fent servir múltiples variables (clau1, clau2, ...) podem tenir seccions crítiques independents entre sí.



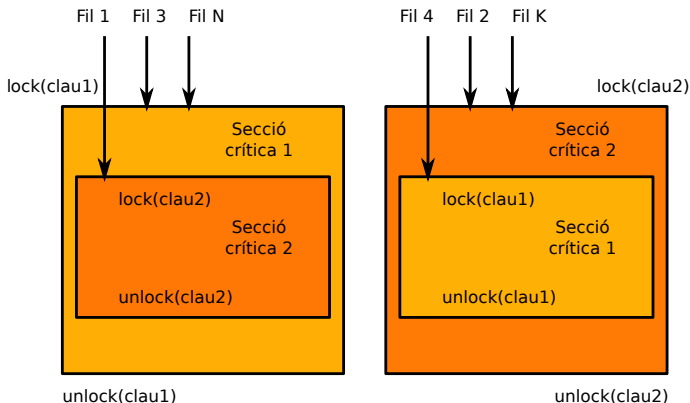
# Protecció de seccions crítiques

Les seccions crítiques poden estar niades entre sí.



# Protecció de seccions crítiques

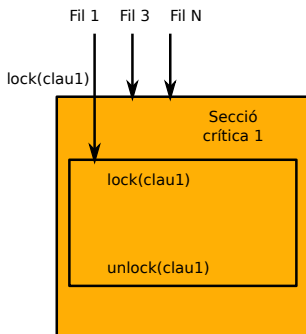
Niar les seccions crítiques pot portar a un **bloqueig total** del procés  
– **deadlock** – si no es va amb compte.



# Protecció de seccions crítiques

Atenció!

- **Per defecte** un fil no pot agafar dues vegades la mateixa clau: la segona vegada que intenti agafar-la es bloquejarà, veure codi `deadlock.c`.
- Una funció de lock es **re-entrant** si permet que un mateix fil agafi diverses vegades una mateixa clau. Consulteu la documentació!





# Exclusió mútua: discussió

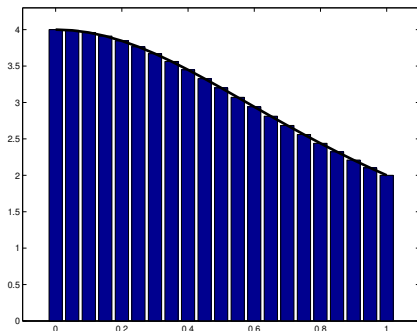
Tingueu en compte que

- Amb lock i unlock **els fils executen** la secció crítica de **forma seqüencial**, un al darrera l'altre, encara que hi hagi múltiples processadors. Intuïvament es pot interpretar com si la secció crítica fos **una instrucció atòmica**: els fils veuran entrar i sortir als altres fils de la secció crítica, però no “veuran” cadascun dels passos intermitjos que els fils realitzen a dins.
- És convenient **reduir** al màxim l'ús de lock i unlock, ja que són **crides a sistema** al sistema operatiu. Feu servir variables independents al fil sempre que sigui possible.
- Per fer un codi eficient no hi ha cap fórmula màgica: es tracta de provar quina solució pot ser la més adequada.

# Exclusió mútua: exemple

Veiem un exemple més el·laborat: càlcul del valor de  $\pi$

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



```
#define NUM_RECTS 100000000
int main()
{
    int i;
    double mid, height, width, sum = 0.0;
    double area;

    width = 1.0 / (double) NUM_RECTS;
    for(i = 0; i < NUM_RECTS; i++) {
        mid = (i + 0.5) * width;
        height = 4.0 / (1.0 + mid * mid);
        sum += height;
    }
    area = width * sum;
    printf("pi = %e\n", area);

    return 0;
}
```

El càlcul de l'àrea de cada rectangle és independent de la resta.

# Exclusió mútua: exemples amb fils

Exemple: codi `calcul_pi_fils.c`

- Aquest codi no requereix l'ús de seccions crítiques. Per què?

Exemple: codi `calcul_pi_fils_semafors.c`

- El codi requereix utilitzar semàfors, un servei del sistema operatiu, per accedir a la secció crítica. Quina és la secció crítica? Per què?
- Es redueix molt l'eficiència de l'algorisme. Per què?

# Exclusió mútua: exemples amb fils

Exemple: codi `calcul_pi_fils_mutex.c`

- Aquest codi és “equivalent” al de semàfors d’abans. En comptes de fer servir semàfors fa servir monitors.
- Ja veurem quines “avantatges” ens donen els monitors respecte els semàfors...

Exemple: codi `calcul_pi_fils_mutex_millorat.c`

- S’ha modificat el codi perquè s’executi de forma molt més eficient. Com s’ha aconseguit? Per què s’executa de forma més eficient?

# Exclusió mútua: exemples amb processos

Exemple: codi `calcul_pi_2processos_mmap.c`

- El codi utilitza un fitxer anònim mapat a memòria per realitzar la suma. No calen seccions crítiques. Per què?

Exemple: codi `calcul_pi_2processos_mmap_semafors.c`

- El codi requereix utilitzar els serveis del sistema operatiu per accedir a la secció crítica. Quina és? Per què?
- El semàfor està compartit entre els dos processos. Com s'aconsegueix que el semàfor estigui compartit?

# Exclusió mútua: implementació

Volem disposar de funcions que es puguin utilitzar per entrar i sortir d'una secció crítica. Aquestes funcions reben el nom de **funcions de bloqueig** (*locks*).

Tipus de bloquejos que existeixen

- Per espera activa
- Per espera passiva

## Per espera activa

- Els fils que esperen per entrar a la secció crítica ho fan de forma activa: contínuament comproven (consumint CPU) si la clau s'allibera.
- En alliberar la clau els fils que volen entrar competeixen per adquirir la clau. Només un fil ho aconseguirà.
- No cal realitzar una crida al sistema operatiu. Es pot implementar amb la CPU en mode usuari.

## Per espera passiva

- Els fils que no poden entrar a la secció crítica s'adormen en una cua associada a la clau (ho fa el sistema operatiu).
- En alliberar la clau es desperten tots (o un) els fils de la cua. Aquests competeixen per adquirir la clau. Només un ho aconseguirà.
- Implica una crida al sistema operatiu.



Quin tipus de clau és millor?

- Per espera activa: aplicacions amb pocs fils i si sabem que els fils esperaran poc (és a dir, que no val la pena adormir-los). S'utilitza en computació científica.
- Per espera passiva: aplicacions amb molts fils o bé si no sabem quant de temps pot trigar un fil a entrar a una secció crítica. Són el que s'utilitzen habitualment.

En alguns casos les funcions lock i unlock s'implementen amb una combinació dels dos:

- En cridar a lock, es realitzen algunes iteracions de forma activa per comprovar si la clau s'allibera.
- En cas que no s'alliberi, s'adorm el fil.

# Tipus de bloquejos

Què veurem als següents temes?

- Tècniques d'espera activa
- Semàfors (espera passiva)
- Monitors (espera passiva)

Quins tipus d'algorismes podem implementar mitjançant aquests bloquejos?

# Paradigmes o models de programació concurrent

## Paradigmes de programació concurrent<sup>4</sup>

- 1 Paral·lisme iteratiu
- 2 Paral·lisme recursiu
- 3 Productors i consumidors
- 4 Lectors i escriptors
- 5 Clients i servidors
- 6 Parells (Peers)

A l'hora de dissenyar la nostra aplicació, cal identificar quin tipus de paradigma (o model) hem de fer servir.

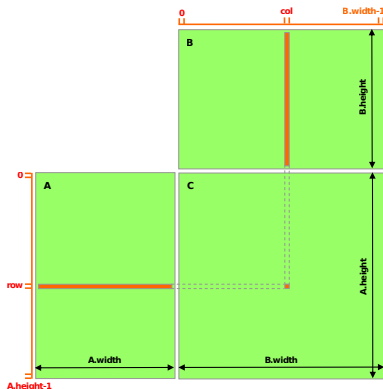
---

<sup>4</sup>Els dos darrers paradigmes es veuran a Software Distribuït.

# Paradigma: paral·lelisme iteratiu

- Un programa paral·lel iteratiu conté dos o més fils que realitza operacions iteratives amb **bucles for i while**: cada fil calcula el resultat per a un subconjunt de les dades i després es combinen.
- Exemple: producte de matrius

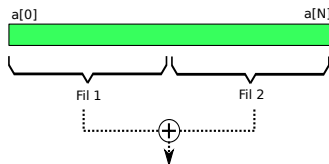
```
double A[N][N], B[N][N], C[N][N];  
  
for(i = 0; i < N; i++) {  
    for(j = 0; j < N; j++) {  
        C[i][j] = 0.0;  
        for(k = 0; k < N; k++)  
            C[i][j] += C[i][j] +  
                A[i][k] * B[k][j];  
    }  
}
```



# Paradigma: paral·lelisme iteratiu

- Un programa paral·lel iteratiu conté dos o més fils que realitza operacions iteratives amb **bucles for i while**: cada fil calcula el resultat per a un subconjunt de les dades i després es combinen.
- Exemple: suma dels valors d'un vector

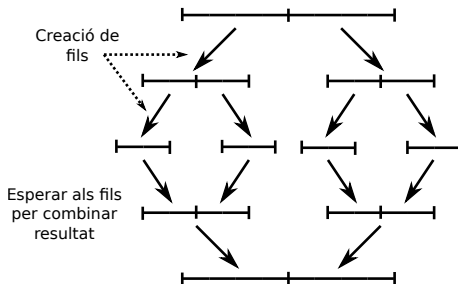
```
double A[N];  
double sum;  
  
sum = 0.0;  
for(i = 0; i < N; i++)  
    sum += A[i];
```



- Cada fil realitza la suma parcial per la seva banda (amb una variable local, per exemple). En acabar els dos fils es pot obtenir el resultat final.

# Paradigma: paral·lelisme recursiu

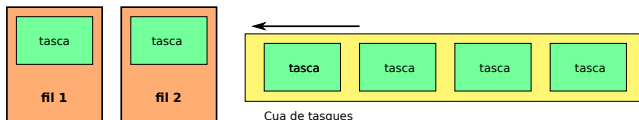
- El paral·lelisme recursiu s'utilitza quan un programa té una o més funcions **recursives** i aquestes treballen amb **parts independents** de dades.
- Típicament associat a algorismes de tipus divisió-i-vèncer.
- Exemple: algorisme quicksort



# Paradigma: paral·lelisme recursiu

Respecte el paral·lelisme recursiu:

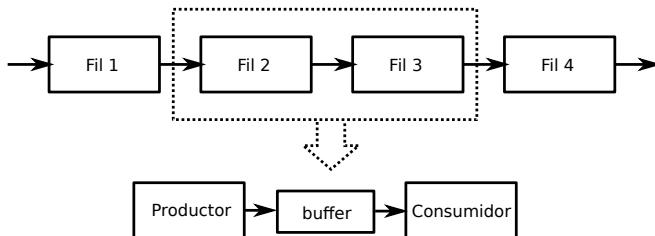
- Crear fils té un cost considerable. No és bona idea crear fils a cada recursió!
- Una solució és utilitzar una cua de tasques:
  - Es crea un determinat nombre de fils a l'inici i les tasques a realitzar s'insereixen en una cua.
  - Els fils agafen les tasques, les executen i poden crear noves tasques que s'insereixen a la cua.



- A Programació Paral·lela aprendreu a programar-la (amb una llibreria especialitzada).

# Paradigma: productors i consumidors

- Un productor processa i **produeix una sortida** de dades. Un consumidor **processa l'entrada** de dades.
- Moltes aplicacions són de tipus productor-consumidor, en què els productors i consumidors estan organitzats al llarg d'una **cadena de processament**.

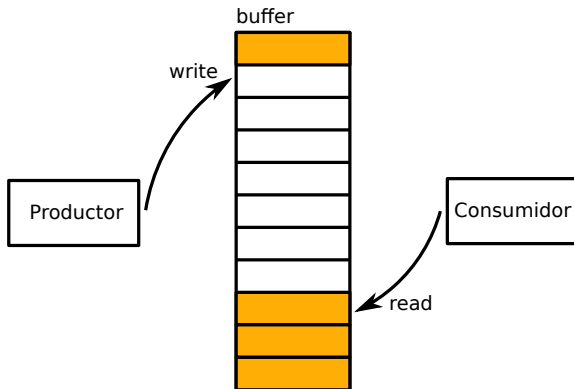


- El productor pot produir les dades a ràfegues. El *buffer* ha d'assegurar que el productor hi pugui escriure tot i que el consumidor no les pugui processar tan ràpid.



# Paradigma: productors i consumidors

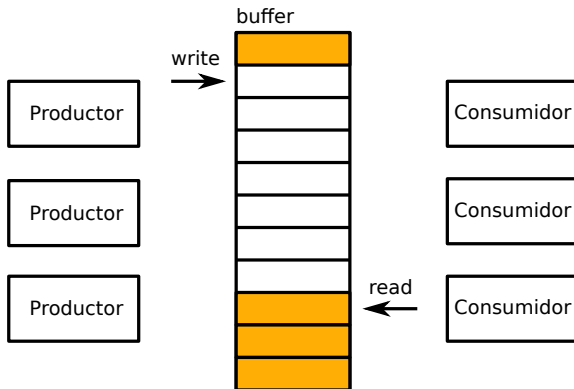
- Moltes vegades s'utilitza un *buffer* circular.
- Ens hem d'assegurar que el punter *write* mai superi a *read* (el productor ha d'esperar si el consumidor és lent), i que *read* mai superi a *write* (el consumidor haurà d'esperar si el productor és lent).



# Paradigma: productors i consumidors

Hi pot haver múltiples productors i múltiples consumidors.

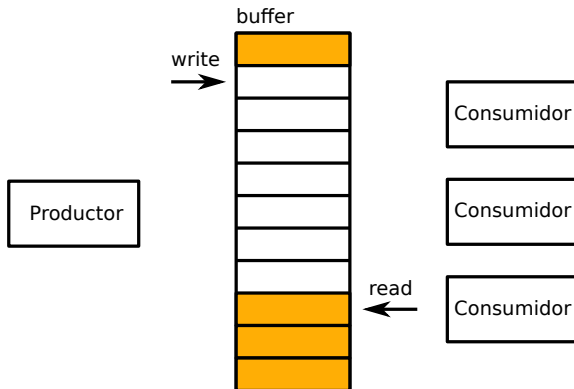
- Cada productor i cada consumidor és un fil.
- Els productors comparteixen l'apuntador a `write`. Els consumidors l'apuntador a `read`.



# Paradigma: productors i consumidors

Típicament: un productor i múltiples consumidors

- Servidor web: dispatcher (productor) i workers (consumidors)
- Lectura d'un llistat de fitxers a processar (productor) i processament dels fitxers (consumidors)



# Paradigma: lectors i escriptors

En una base de dades

- Els **lectors** accedeixen a la base de dades i la llegeixen.
- Els **escriptors** hi accedeixen però per escriptura.

Per evitar interferències entre fils

- Un escriptor necessita accés exclusiu a la base de dades (o el registre de la BBDD) en escriure-hi.
- Si no hi ha cap escriptor a la base de dades (o el registre de la BBDD), múltiples lectors poden accedir a la vegada per realitzar transaccions.

# Paradigma: lectors i escriptors

En **programació orientada a objectes** es dona sovint aquest paradigma:

- Les funcions tipus “get” no modifiquen l'objecte (lector).
- Les funcions tipus “set” modifiquen l'objecte (escriptor).

Recordeu

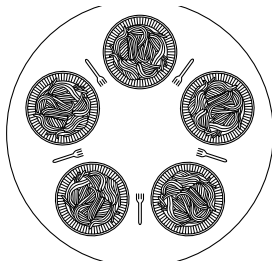
- Hi pot haver múltiples lectors accedint de forma concurrent a un objecte.
- Quan hi accedeixi un escriptor hem d'assegurar que cap lector i ni escriptor hi accedeixi al mateix temps.

A l'hora de programar amb múltiples fils i/o processos

- El sistema operatiu ens ofereix dels serveis necessaris (crides a sistema) per evitar interferències.
- No es pot preveure quan els fils/processos canviaran de context. Això fa difícil la programació multifil... al principi!
- Identifiqueu quin(s) paradigma és l'adequat per implementar la vostra aplicació i utilitzeu-lo. No cal inventar res nou!
- En utilitzar una llibreria externa, assegureu-vos que aquesta és *thread-safe*, és a dir, permet cridar-la des de múltiples fils de forma concurrent.
  - La majoria de les funcions de la llibreria estàndard (malloc, printf, ...) són *thread-safe*. Internament tenen els mecanismes de sincronització necessaris.

## Annex: el problema dels filòsofs

Es habitual utilitzar l'anomenat “**problema dels filòsofs**” per exemplificar els problemes associats a la sincronització de fils. Observeu la figura:

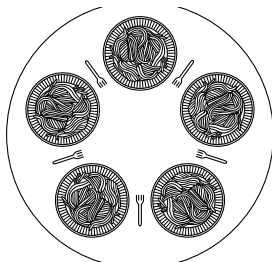


- Hi ha 5 filòsofs (un per plat de pasta) que estan pensant (i no parlen entre ells).
- Quan tenen gana, han d'agafar les dues forquilles del seu costat i menjar. En acabar de menjar, deixen les forquilles a taula.

## Annex: el problema dels filòsofs

Les forquilles representen el “recurs” compartit.

- Els filòsofs no poden menjar si no disposen de les dues forquilles.
- Per menjar, primer han d'agafar una forquilla (pex la de l'esquerra) i després l'altra (pex la de la dreta).
- En acabar de menjar, deixen primer una forquilla (pex la de l'esquerra) i després l'altra (pex la de la dreta).

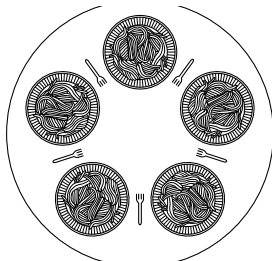




## Annex: el problema dels filòsofs

Es poden produir situacions “curioses”?

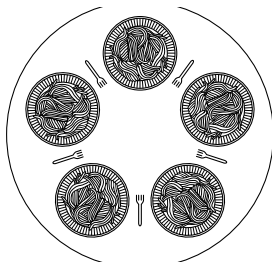
- Suposem que tot els filòsofs tenen gana a la vegada.
- Tots agafen a la vegada la forquilla de l'esquerra.
- Els filòsofs voldran agafar la forquilla de la dreta però està ocupada de forma indefinida pel filòsof del costat que també vol agafar la seva forquilla de la dreta. Es produeix un **deadlock**, veure codi `filosofs.c`.



## Annex: el problema dels filòsofs

Proposem una solució al problema anterior

- Un filòsof, quan té gana, comença per agafar la forquilla de l'esquerra.
- Abans d'intentar agafar la forquilla de la dreta mira si està lliure<sup>5</sup>. Si està lliure, agafa la forquilla i menja.
- Si no ho està, deixen la forquilla de l'esquerra, espera un temps aleatori i torna a intentar menjar.



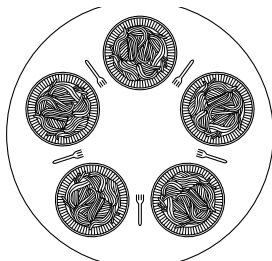
---

<sup>5</sup>Hi ha funcions C/Java que permeten saber si la clau ha estat agafada.

## Annex: el problema dels filòsofs

Què pot passar?

- Existeix la possibilitat que un filòsof arribi a esperar molt de temps per menjar si els filòsofs del seu costat no paren de menjar.
- Tècnicament es diu que es produeix inanició (**starvation**) i no es una solució acceptable.



## Annex: el problema dels filòsofs

Quines solucions s'han proposat?

- L'original, per Dijkstra. Es basa en numerar les forquilles del 1 al 5. Quan un filòsof vulgui menjar agafa primer la forquilla amb índex més baix. És a dir, 4 filòsofs comencen per agafar la forquilla de l'esquerra i un filòsof comença per la forquilla de la dreta. No hi ha deadlock, ni starvation!
- Utilitzar un àrbitre. Quan un filòsof vulgui menjar ha de demanar permís a l'àrbitre per fer-ho. L'àrbitre només pot donar permís, en un moment determinat, a un filòsof per menjar. Amb aquesta solució es redueix el paral·lelisme.

