

Un processus peut avoir envie d'exécuter plusieurs choses en même temps. Ces différents flux d'exécution, ou threads partagent le code executable, les segments de données, les fd.

Cependant, ils ont chacun leur propre pile, registres, et program counter.

Il existe :

- Threads user (on lance avec pthread), context switch rapide reposé sur bibliothèques qui peuvent être ≠ d'un programme à l'autre
- Threads kernel (on lance pas), context switch lent reposé sur une abstraction, un scheduling etc qui sont implémentés directement dans le kernel.

pthread-create crée un thread

pthread-join attend la fin de son exécution

pthread-exit termine un thread.

On récapitule :

Processus : programme en cours d'exécution, a son propre AS

thread : flux d'exécution d'un processus, partage la mémoire, fork

task : bout de code chargé en mémoire.

Les processus peuvent communiquer avec

- des signaux
- des pipe
- des sockets

Les signaux:

- certains sont liés à une action par défaut et ne peuvent pas appeler une procédure alternative (SIGKILL, SIGSTOP)
- certains peuvent être récupérés et appeler une procédure à l'aide de signal handlers (SIGCONT, SIGCHLD) par exemple

L'envoi d'un signal peut se faire avec kill (pid, signal)
la fonction pause() permet de faire de l'attente passive de signal.

Les pipe (tube de communication)

- 2 types : sans nom et nommé
pipe() ou mkfifo()

Les sockets

- INET (network) et UNIX (intime)

Synchronisation de processus.

Objet critique: objet partagé par 2 processus qui ne doit pas être accédé simultanément
(ex les variables, les fichiers)

Section critique: section de code qui opère sur un ou plusieurs objets critiques.

Si 2 sections critiques sont exécutées en même temps, on a un undefined behavior.

Pour éviter d'exécuter 2 sections critiques en même temps, on utilise les mutex (Mutual Exclusion)

les sémaphores

les locks

Les locks

Un lock est une sorte de design pattern qui n'autorise qu'un seul thread à la fois à accéder à une section critique.

Test and Set Lock (TSL)

code exécuté

```
{ while(TSL(lock));  
  /* Section critique */  
  lock = 0;  
  unlock }
```

attente active, on attend que le lock soit débloqué par un autre thread.

TSL(lock)

```
res = *lock  
*lock = True  
return res
```

test

Compare and Swap (CAS)

Attention

Deadlock =

2 threads

attendent qu'une
ressource soit

libérée par l'autre

P₁ P₂

lockA lockB

attendB attendA

○ ○

LiveLock =

même chose

mais c'est la

procédure de

débloquage qui

boucle.

```
while (CAS(lock, 0, 1));  
/* Section critique */  
lock = 0;
```

```
CAS(lock, old, new)  
res = *lock  
if (*lock == old)  
*lock = new ← swap  
return res
```

Ici, on utilise des spinlock. (boucle bloquante) pour attendre avant d'exécuter une section critique.
On peut gagner en performance avec SLEEP et WAKEUP pour faire de l'attente passive = ça devient des mutex.
Mais parfois, l'attente active est obligé, si le processus n'est pas schedulable, ou si on sait qu'un context switch est plus long qu'une attente active.

les mutex

Les mutex sont semblables aux lock mais peuvent être partagés par des processus différents, et font de l'attente passive.
On se fait scheduler ≠ un spinlock.

les sémaphores

mutex = synchro entre 2 process.

sémaphore = synchro entre plusieurs process/thread

Si $n = 0$, bloquant si $n > 0$ disponible

P et V (Première et Veconde)

Analogie : n = nombre de voies

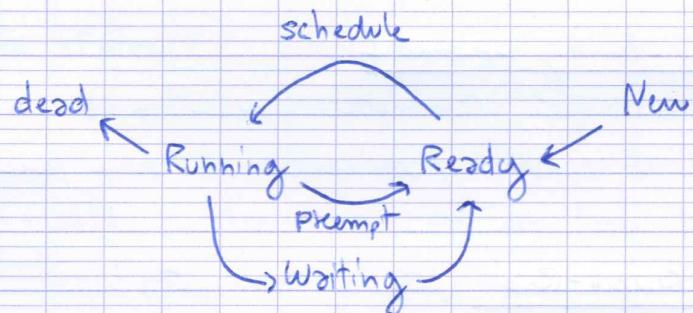
process = train

P = un train s'engage sur une voie

V = un train sort d'une voie

SYS2

Scheduling



Quand le processeur passe de l'exécution d'un programme à un autre, il doit recharger l'état des registres et la pile, cela s'appelle un context switch. Le contexte du process courant est sauvegardé pour plus tard.

Scheduling non préemptif

c-i-d, une fois l'exécution d'un programme commencé, on l'exécute jusqu'au bout.

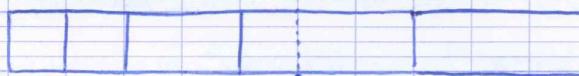
- First come First Serve (FCFS)

= une FIFO, premier arrivé premier exécuté

- Shortest Job First (SJF)

Process le + court en premier

On termine + de tâches + vite



À la fin on a déjà
fini 3/5 tâches

Problème : si on a plein de longs process et que celui qui nous intéresse est en fin de file, on va attendre longtemps. = starvation / famine

Scheduling préemptif

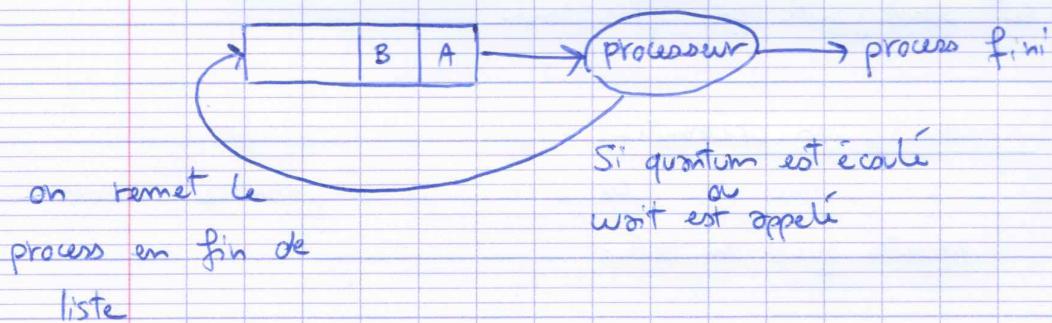
= on peut interrompre l'exécution d'un programme et en continuer / commencer un autre avec un context switch

- Round-Robin (RR ou tourniquet)

on exécute chaque process pendant q un quantum (20-50ms)

Quantum trop long = temps de réponse long

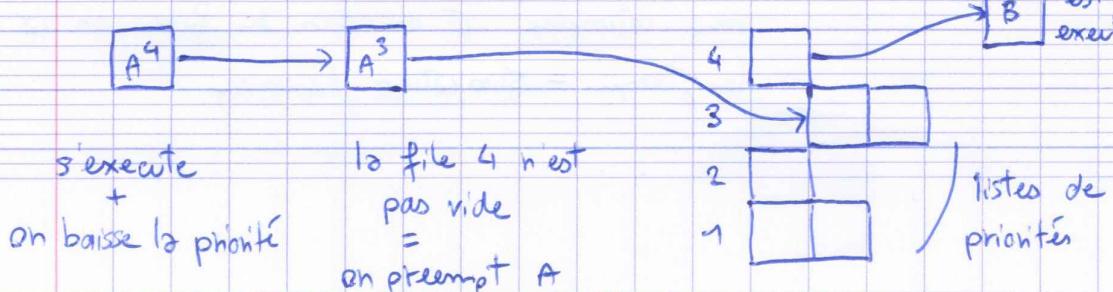
Quantum trop court = trop de context switch



Problème : Unfair, car cet algo ne prend pas en compte la priorité / l'importance des process

- Round-Robin avec priorité ou Priority Queues

- Chaque process a en + une priorité
- On exécute en 1^{er} ceux qui ont une forte priorité
- On baisse la priorité du programme à chaque exécution.
- On a une file par niveau de priorité
- Si le process en cours a une priorité < à celle d'un autre process, on le met en fin de liste et on schedule le process en tête de la liste de priorité supérieure



Avantages :- Fair

- La priorité des processus est remontée en fonction de bcp de paramètres comme les I/O, cela maximise aussi le temps de CPU user (voir bottleneck)

Problème : Il reste des bottlenecks ! 3 types

CPU Bound	Mem bound	I/O Bound
consomme bcp de CPU	bcp d'accès mémoire	bcp de wait

Ces bottlenecks permettent de distinguer les processus interactifs (I/O bound, en comptant les wait) et non interactifs.
Comment minimiser les bottlenecks ?

1) les processus non interactifs ont tendance à ne pas finir leur quantum.

À chaque context switch, on jette les caches. Donc on a envie de rallonger le quantum pour changer de cache moins souvent.

2) les processus interactifs ont tendance à wait avant la fin de leur quantum, on a envie de raccourcir leurs quantum.

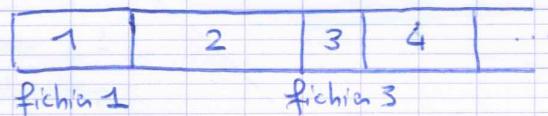
⇒ Pour chaque niveau de priorité, on va diviser la queue en 2 (pour interactif et non interactif)

et on définit 2 types de quantum : court pour interactif long pour non interactif

court = priorité haute, long = basse.

3 systèmes d'alloc de fichiers

1 - Allocation contigüe (pas utilisé)



Problème :

- un fichier ne peut pas être redimensionné
- ça fait des trous dans la mémoire
(rappel : la mémoire est découpée en blocs, si un fichier est + petit qu'un bloc, on lui alloue quand même le bloc entier, donc plein de petits fichiers = gachis)

2 - Liste chaînée (utilisé)

on découpe le fichier en blocs et chaque fin de bloc pointe sur le bloc suivant (512, 1024 2048 4096 souvent)

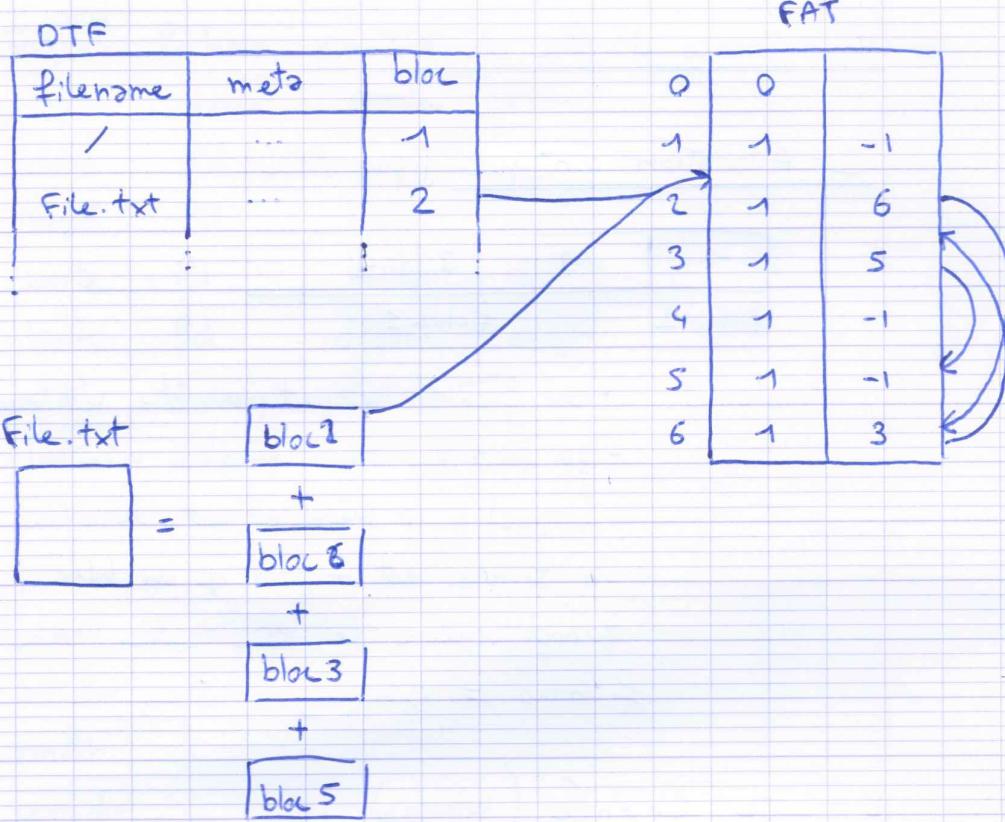


fichier de 3 blocs

Problème : on doit se taper tous les blocs si on veut accéder à un endroit précis du fichier

L'exemple de Liste chaînée : FAT.

- FAT (File allocation table) est une table en début de disque qui indique pour chaque bloc du disque
 - s'il est libre ou non
 - le bloc qui le suit
- DTF (Directory table format) est une table qui lie les noms de fichiers/dossiers à des blocs.
structure



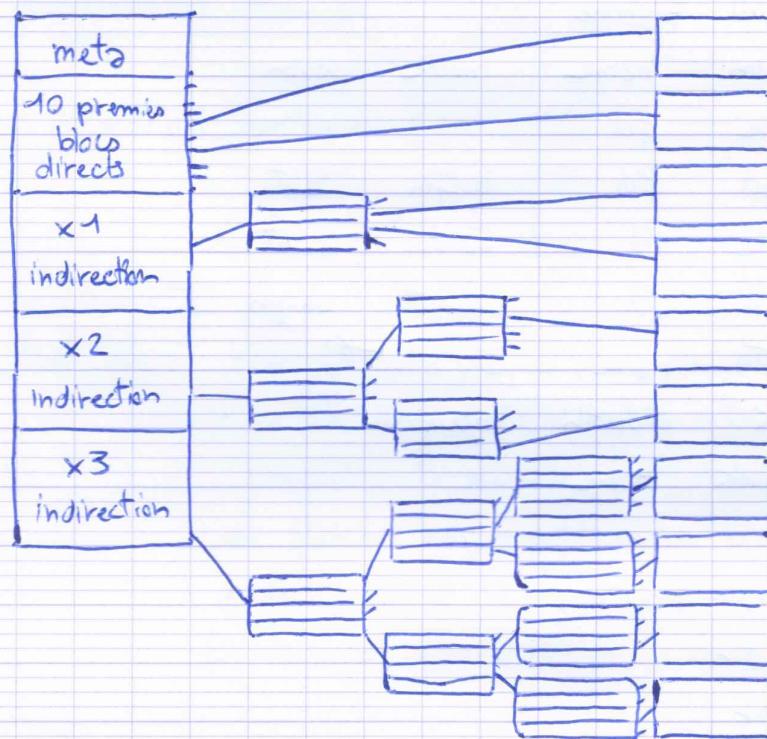
Décomposition d'un fichier en FAT (winodis)

3- Les 1-inodes (Unix)

- Les dossiers sont des direct inodes dont la data est une liste de direct.
- Les direct contiennent le nom et le type et le numéro d'inode du fichier correspondant
- Les inodes contiennent les métadonnées (taille, type, dernière modif, droits, accès) du fichier
 - Des liens directs vers les 10 premiers blocs
 - Des liens indirects pour les autres blocs

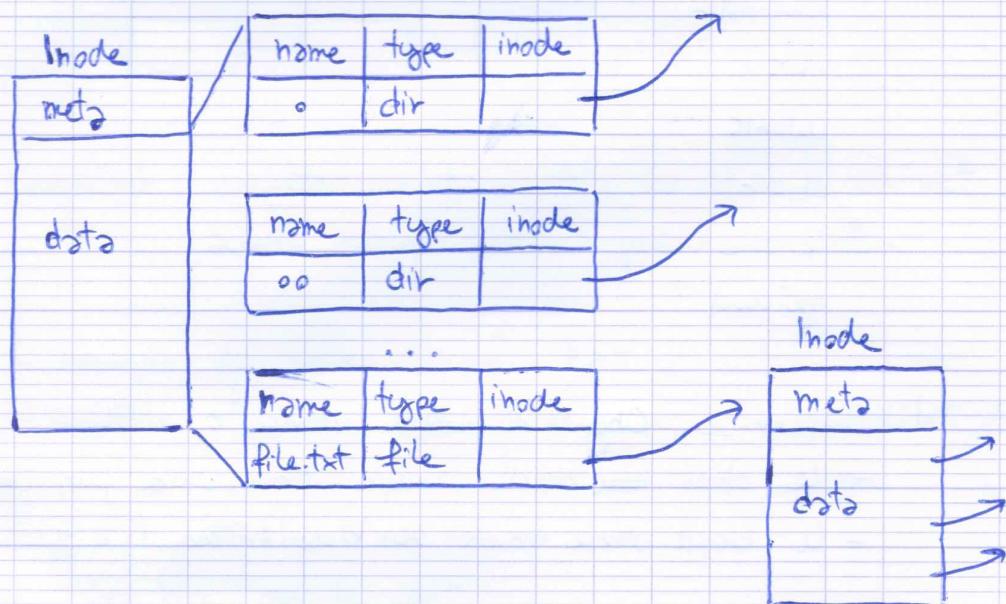
inode

blocs



Structure d'un inode

Direct list



inode d'un dossier

(
/ .
/ ..
/ file.txt)

Syscall fichiers

open	name	mode	perm
close	fd		
read	fd	buffer	count
write	fd	buffer	count
lseek	fd	offset	origin
stat	name	statstruct*	
link	oldpath		newpath
unlink	name		

Syscall directory

mkdir
rmdir
opendir
closedir
readdir
rewinddir
link
unlink
chdir
rename
getwd

Difference chan vs block device :

- Le chan ~~block~~ device envoie caractère par chan
- Le block device envoie des blocs entiers