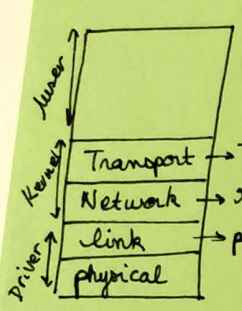
Programme :

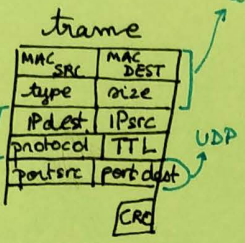
- Process / Thread / Task
- Scheduling
- Mecanisme de synchro
- Reseau
- file system.

Reprendre les cours de Réseaux.

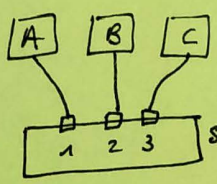
TCP/IP



→ TCP/UDP, i
→ relie les reseaux entre eux (IPv4/IPv6)
→ permet d'envoyer des trames (Ethernet)



hub : passif → broadcast.
capacité du reseau limitée car 1 seul personne peut parler en m temps.
revoir domaine de collision.
Ce dernier doit ê diminué au max.
Nvx soluto



Le switch tient une table d'apprentissage.

MAC	PORT
A	1
B	2

Si A envoie une trame
le switch ~~que~~ voit que A est sur le port 1.
Donc il le note dans la table.
Lorsque B repond le switch note B sur le port 2.

MTU : combien de temps je peux parler. (en nb de bytes).

Network :

Permet d'uniformiser les adresses. = adresses IP : XX.XX.XX.XX.
Routeur fait le lien entre 2 reseaux.
Netmask : met ~~#~~ les bits du prefix à 1 les autres à 0. Cela permet de savoir qui est sur le même reseau.
Revoir les tables de routage. → table d'apprentissage entre routeur
On met la MAC adresse du routeur dans le paquet. Puis le routeur va remplacer se MAC adresse par celle du prochain routeur.
Les packets sont fragmentés afin de s'adapter aux differentes MTU.
TTL : valeur qui est decrementé de 1 à chaque routeur, le packet est perdu lorsque la TTL atteint 0. exprimé en sec (car avant 1 sec = 1 routeur). Mais du coup permet le definir un timeout pour attendre le packet.

ransport :

DP : permet de dialoguer. DNS : on envoie un nom (→ url), ça renvoie une IP.
quette AXFR : permet de demander un nom.

TCP: permet de mettre les paquets ds l'ordre.
Reprendre le cours de Chewie s/ les ACK, HandShake.

Socket: API, implem de Burckley.

une call initial: fd socket (INET, STREAM ou DGRAM)
protocol ↙ type de connexion ↘

initie la connexion — connect (fd, sockaddr) → 3 way handshake.

s'accroche à un port — bind (fd, sockaddr).
listen (fd, ——)

fd = accept (fd, &sockaddr)

send ( ) } qd on sait a qui on
recv ( ) } parle
sinon - sendto et recvfrom.

Process vs thread

Concurence entre les process. plrs file d'execut° ds le m̂ process.
Process = State + file d'execut° (= thread ≠) ils partagent les signaux, file descripteur et l'adress space (bout de memoire alloué)
On peut avoir plrs fois le m̂ programme qui éxecute en // grâce à un fork.
Pour communiquer entre 2 process on utilise un pipe. Mais pas pratique car doit ê prevu à la création. Du coup autre ~~mem~~ moyen: sockets. ≠ types: → INET = ipv4, tcp/ip → UNIX = pr la communicat° locale. On s'y connecte grace à connect (ip + port). Vu qu'on est en local on a un fichier special de type socket. du coup on donne à connect le chemin du fichier. Les sockets UNIX permettent de mettre des permissions s/ le fichier.
Pour ecrire s/ l'ecran, on se connecte s/ le (serveur X. graphq) → Xorg. et on lui demande d'afficher le necessaire. Le serveur graphq est lancé en mode user et le gestionnaire de login permet de recuperer les fd neceassaires (ds lesquels la data à afficher est contenue). Le gestionnaire de login / logind permet de creer les sessions.

Une autre methode:
Memoire (SHM) permet de partager en mémoire via un fichier.
mkfifo.
Pour executer 2 ≠ thread en m̂ temps il faut 2 stacks ≠ et un set de registre ≠, on partage les variables globales et une zone de memoire mappable.
Du coup pour lancer un thread on alloue une stack et un ~~nouveau set de registre~~.
Les threads partagent les registres dc il faut enregistrer les valeurs des registres à un autre.

2019

EAGAIN → bloque si jamais le fichier n'est pas prêt à être lu.

fork est remplacé par clone qui permet de preciser les paramètres de la duplicat°.

task → permet une meilleure relat° avec le kernel mais passage de l'un à l'autre + lent.

pthread → implementé ds la lib C. permet de faciliter l'appel à clone + d'autre details

Scheduling

Rappel : time sharing = executer plus tâches en m tmps.

Tache = temps d'execut° + deadline + dependance.
(pas forcement lié au système).

batch : 1 programme à la fois qui tourne jusqu'à la fin.

→ Turnaround time = temps entre la (début? fin de l'execut° et l'obtent° du resultat.

→ Throughput (débit) = nb de tâche à la (heure/ journée, ...).

Le but est de minimiser le turnaround et de maximiser le débit.

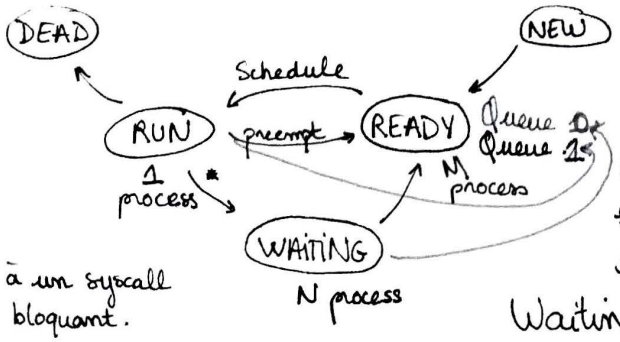
Pour ~~maximiser~~ minimiser le turnaround on fait du FCFS (First Come, First Served)

Pour maximiser le debit on fait du SJF (Shortest Job first).

Time Sharing

Il n'y a plus de temps d'execution, ni de dead line, ni de dependances.

d'algo qui determine qui passe après (scheduler) doivent s'executer en temps constant sinon il prennent du temps sur celui d'execut°. Starvation = tt les process doivent tourner. Fair = pas de favoritisme entre les process (donner le m temps à chacun)



* appel à un syscall bloquant.

FIFO : Choisis le 1er de la queue

On veut mettre les news en début de queue afin d'avoir un système + réactif. = Response Time (temps entre NEW et RUNNING).

Waiting time = temps passé en READY.

Round Robin : ajoute des interruptions lié à un timer, qui peut envoyer un process de RUN à READY. On choisis un quantum de temps maximum pr le RUN. pb : si le process d'avant est passé en waiting on tourne sur la fin de sa clock. e (la clock est longue à reset).

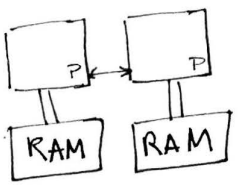
Système en temps réel : connait le plan d'execut° à l'avance. ex s/ un avion.

2 types de programmes :
→ Interactif : utilise des syscalls ex: shell, http, init
→ Non Interactif : ex: compilateur, program qui calcule.

Bottleneck (goulot d'étranglement) = ralentit le flux.

↳ CPU-Bound, Mem-Bound, IO-Bound (ex: téléchargemt) → SVt pr les interactif.
(cpu lent). (ex: accès trop fréquent à la mémoire) tendance à ne pas finir le quantum.
↳ SVt pr les non-interactif, tendance à ~~ne pas~~ finir le quantum.

On va prioriser les process interactifs, pour les repérer on compte le nb de fois qu'il passe en waiting.
Pour les non-interactif il est + rapide de faire tourner le process sur une durée de 2 n que sur 2 durées de n car on a pas a changer les caches.
≠ types de caches = L_1, L_2, TLB, ... L_2 = accès mémoire L_1 = Core Cash.
le cash est utilisé lorsqu'on charge + de mémoire que demandé. ex si on travaille s/ un tableau il est plus rapide de charger tt le tableau lorsqu'on traite la 1ère case. que si on va chercher cases par case en mémoire.
Dans la pipeline d'exécut° on a un cache L_1 au niveau du fetch pr charger les instruct° et un au niveau de memory où sera chargé la data. C'est un L_2 qui coordonne les deux L_1. Le TLB permet de sauvegarder les permissions et accès à la page. Il est vidé lors d'un changement de process.
On répare la queue en 2, une pour les CPU-Bound et une pour IO-Bound
On prend n queues allant de 0 à n-1, la queue 0 est la + prioritaire.
Du coup les programmes vont aller ds la queue correspondant à leur priorité.
RUN → READY = ↘ Priorité ; RUN → WAITING = ↗ Priorité ; READY → RUNNING = ↗ Priorité.
Plus la priorité est basse, + on alloue un quantum long.
Ticket Scheduling : les programmes ont des tickets et ils utilisent 1 ticket par round on met une priorité à ceux qui ont le - de ticket
CFS (= Completely Fair Scheduling) = on unifie le temps d'execut° des programmes.
NUMA : Système à deux processeurs.



↔ = canal de dialogue
Si on veut accéder à qqchose qui est sur la rame voisine on met un peu + de temps.

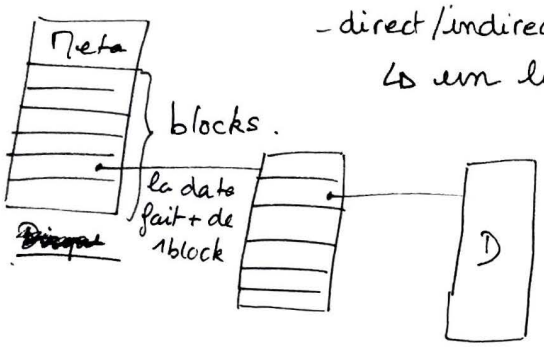
File System.

Abstaction Storage
→ files
→ hiérarchie de nom

But : partager et nommer des ℓ informations.
Block Device (Disk)
→ lire et écrire des blocks. (= suite de data continue) s'apparente à un buffer de taille fixe.

Sizes : - 2048 (cd-rom)
- 4K (disque moderne).

On utilise les BTree pour organiser les blocks ds le disque
inode : - un en mémoire (utilisé par l'OS)
- un s/ le disque
- il contient de la data et metadata, il est referencé par un N°.

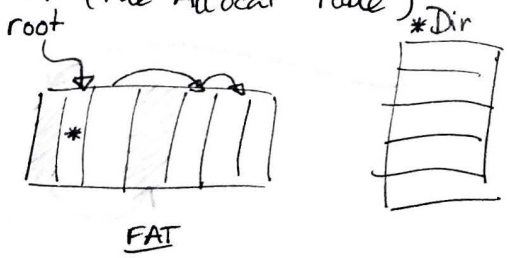


- direct / indirect block / double / triple indirect.
↳ un lien de chaque qui correspondent aux tailles de blocks = modèles classique.

Directory : fichier ĉ un autre sauf qu'on change le type et que ds la data on met un tableau d'entry (= nom + n°-inode) + length du nom.

→ Superblock placé en début de fileSystem
on en met plus car si il est effacé c'est le merde. Les superblocks secondaires ne contiennent que les stats ▷

FAT (=File Allocat° Table)



pb : on écrit tjrs au même endroit.
pratique car l'implem prend peut de place c'est simple.

(regarder direntry) = roots ?

Dans une carte SD on a une mini RAM ds laquelle se trouve la FAT. (pr aller + vite).
Du coup elle est commit avant qu'on retire la carte de l'ordi.

- zfs
- btrfs.
- Merkle Tree.
arbre dt chaque noeud est hashé et contient le noeud de son parent
ex : git.

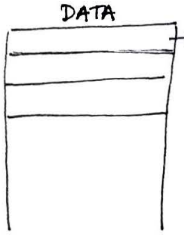
---

1 seul FS visible s/ Unix.
mount() → prend un disque et l'expose ds un dossier.
VFS → Virtual File System
vtable contient les pointers vers tt les fonctions d'un objet, elle est placée debut de la structure de l'objet.
même principle est utilisé dans les fichiers et dossiers ? ils possèdent des pointers sur r tt les f° de manipulat° de fichier (read, write, open, ...)
On a un compteur de read, ainsi si deux pointeurs pointent s/ le m̂ fichier un des deux close, le ficheer n'est pas detruit. ? principe que les shared ptr.

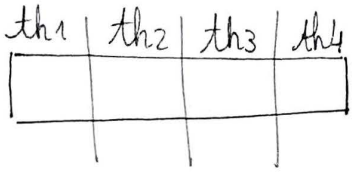
Lorsque fread est appelé on appelle vfs_read qui lui même appelle le read du fichier.
↳ gère les trucs liés au kernel
Le numero major indique la bonne struct file_operation
Le numero minor indique quel object il faut traiter.
voir ioctl → syscall special pour gerer les IO s/ devices exterieurs.

Multithread.

SiMD → Single Instruction Multiple Data.
example avec addition sur un vecteur :
il existe des registres qui prennent 4 float à la fois pour faire des operat° 4x plus vite.

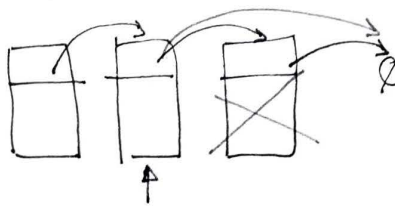


→ cashline (≈128) Lorsqu'on charge de la data on charge une cashline entière.



Pour aller + vite on peut diviser le calcul s/ le vecteur en le confiant à ≠ thread s'executant en m̂ temps. Un thread s'occupe des cases de 0 à N et non pas % nb_thread = th_id car l'ecriture et lecture des la data se fait par cashline. Si 2 ~~cas~~ thread ecrivent s/ la même data, (chacun dans leur cash) les cash se synchronisent ce que ↗ le temps d'execution. = False sharing.

exemple sur une liste chainée


Imaginons qu'un thread A s'apprête à lire un pointeur next et qu'un thread B supprime l'elt pointé au m̂ moment. A va lire de la mémoire qui n'est plus dispo.

On a les sect° critiques st des endroits dangereux (ici ajout, suppression, ...) le thread va donc devoir ê le seul. On lock donc la partie critique qui va devoir attendre que le reste soit finis

Atomic Primitives
→ test and ~~test~~ set (TAS)
→ compare and swap (CAS)

Volatile: dis au compilo de fetch la variable

```
lock
for(,,)
{
  if (v==0)        } CAS
    {v=1           }
    break;}*       }
}
```
↳ spin lock : le CPU attend activemt ⇒ mieux si
→ j'ai ne peux pas me faire scheduler, ex: controleur d'interrupt°
→ j'ai besoin d'aller vite

```
* else
    sleep(0)
```
↳ Mutex : dors pendant un temps ⇒ ?? les autre cas.

~~unlock~~.

pb producteur / consomateur : un thread produit et l'autre consomme.
prod → Queue ↗ conso.

La queue est implementé s/s la forme d'une ring buffer

Le pb c'est si + thread et consumer ou producteur.
On peut mettre des mutex quand on modifie la queue mais

* Semaphore (demander à Jeanne)
→ Queue de process en attente et count.
→ Quand on a besoin de compter.

Deadlock: lock indeblocable :

T1	T2
lock(A)	lock(B)
lock(B)	lock(A)
unlock(A)	

→ ici T2 attend que T1 unlock A mais T1 attend que T2 unlock B pour unlock A.

Live lock: Deux personnes qui se gène dans un couloir

T1 ——— lock(A)
T2 ——— lock(A) x unlock(A)

* ici T1 + prioritaire que T2 donc il est scedulé avant T2 qui a déjà lock A. Du coup on fait une inversion de priorité pr que T2 soit executé avant pr pouvoir libérer A.