

Compsys

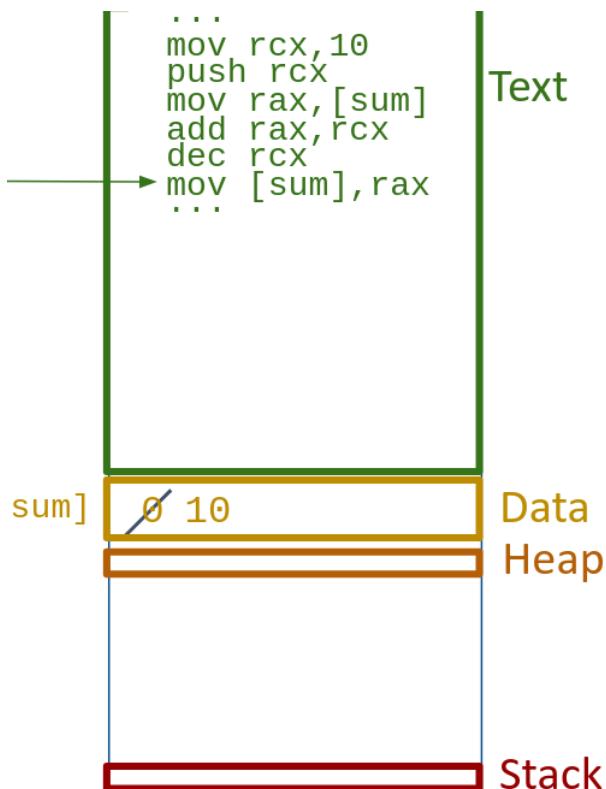
Lundi 17 février

Programme stocké dans le disque, puis quand lancé est chargé dans la mémoire principale. (memory image). Il y a plusieurs segments :

- text (le code)
- data (stocke les variables globales et static)
- heap (stocke les données persistantes du programme)
- stack (stocke les variables locales et les pointeurs de retour, les arguments de fonctions...)

ⓘ variables locales dans le stack ?

On alloue à toutes les variables locales potentiellement créées par la fonction une place dans le stack -- en les laissant vides, non initialisées -- comme ça on sait que si on a plus de place dans les registres on pourra les stocker/récupérer dans le stack.



On les construit comme ça pour qu'ils aient une taille dynamique. Quand on veut remplir le stack on fait `addi sp, sp, -4` (on part des adresses hautes vers les adresses basses).

Dans le CPU, le registre `ip` pointe donc vers le text segment.

Et un process est créé (composé de plusieurs threads, un manager, et des worker threads).

Worker threads : p. exemple dans le cas d'un navigateur, un thread chargé de récupérer une image, une vidéo.. Ils sont créés par le main thread. Ils communiquent entre eux à travers la main memory.

Le CPU est **virtualisé** : chaque thread pense qu'il est le seul à s'exécuter (le CPU stocke différents copies des registres, du stack pointeur, du instruction pointeur, pour chaque thread --> on appelle ça le **contexte**). En comparch c'est le multi-threading (on réutilise la même pipeline mais on exécute différents programmes en même temps --> on fait du context switching à chaque fois qu'on exécute un nouveau thread pour maj les registers). Chaque thread a donc son CPU context.

- exemple d'un process qui a deux threads ? --> ils doivent être indépendants

ⓘ Que partagent les threads d'un même process ?

Chaque thread a son propre stack, son propre CPU context (c'est-à-dire, ses propres registres, son propre stack) mais ils partagent le même text segment, data segment, heap..

Lundi 24 février

Privilege modes, limited direct execution, kernel/loader

Le CPU stocke le **privilege level** (low/high). Certaines instructions ne peuvent être exécutées qu'en high (aussi appelé kernel mode). Les threads sont lancés en low (user mode).

Le CPU est **interrompu** pour donner une chance à tous les threads de s'exécuter (géré par le OS scheduler).

Limited direct execution: limited (en temps et privilèges) mais direct (pas d'intermédiaire entre les instructions et leur exécution).

Qui fait tourner l'OS ? Le même CPU qui fait tourner les threads! d'où l'intérêt du privilege level.

Quand on veut lancer un programme, le **kernel** du système se lance :

- il crée un thread/process
- il initialise l'image mémoire du process (il copie le code du programme dans le text segment, etc.)
- il change le privilege mode du CPU en **low**
- il lance le **loader**

Le **loader** finit de préparer l'image mémoire du programme, il est en low privilege :

- il charge les arguments du programme dans le stack (pourquoi pas dans **data** ? --> les arguments du programme sont des arguments de la fonction main du programme, donc considérés comme des variables à mettre sur le stack)

Les **syscall** sont donc utilisés par les threads en low privilege pour exécuter certaines high-privilege instructions. Dans le kernel, il y a un **syscall handler**. Il y a donc un context switch en cas de syscall pour que le kernel passe en exécution.

États d'un process :

- **Running**: au moins un des threads du process est running
- **Ready**: aucun thread du process est running
- **Blocked**: il ne peut pas run tant qu'un évènement arrive comme un message réseau reçu

quand le kernel tourne pour un syscall exécuté pour un thread, on dit que le thread est running

⚡ un thread ne passe pas en blocked dès qu'un syscall est fait !

si un syscall n'est pas bloquant (accès à du stockage ou au réseau par exemple), alors le thread est toujours running (le kernel tourne pour le thread) ! par contre à partir du moment où le kernel fait un appel bloquant (I/O), alors le thread est blocked.

Limited execution du thread :

- quand il y a une erreur, une exécution illégale (exception/trap) ou un double-clic par exemple, un interrupt est déclenché et est géré par le **interrupt handler** du **kernel**
- si rien ne se passe, il y a toujours le **timer interrupt** qui lance l'OS scheduler toutes les quelques ms (share the CPU fairly)

Quand est-ce que les process sont créés ?

Généralement, on passe par des **wrappers** :

direct syscall:

```
size_t bytes_read = read(fd, buffer, sizeof(buffer));
```

wrapped syscall:

```
size_t bytes_read = fread(buffer, 1, sizeof(buffer) - 1, file);
```

plus performants (car souvent ils sont + bufferisés p. ex un `fread` va lire un peu plus qu'un `read` normal, comme ça si on appelle plusieurs fois `fread` on ouvre pas le fichier à chaque fois).

- `exit` syscall --> ne return jamais, il ferme le process
- `exec` --> mutation, remplace le code du process par le programme qu'on veut exécuter
- `fork` --> clone, créé un jumeau du process en cours (avec le même text segment, etc.). on appelle quand même le process qui l'a créé **parent** et le nouveau **enfant** (même s'ils sont à égalité). `fork` renvoie `0` pour l'enfant et le PID de l'enfant pour le parent.

```
puts("Programme lancé !");
int fs = fork();
if (fs == 0) {
    // code de l'enfant
} else {
    // code du parent
}
```

- `wait` --> attend que le process enfant soit terminé (`wait(NULL)` attend que le premier child meurt)

⚡ Subtilités du fork

Le fork reçoit une **copie** de la mémoire du parent (c'est-à-dire qu'il reprend exactement les mêmes valeurs dans les variables mais s'il modifie les variables ou que le parent modifie les variables, ça ne sera pas visible entre eux)

Le fork démarre exactement de là où il est appelé chez le parent. Dans l'exemple ci-dessus, seul le parent affichera le print, l'enfant non.

Le fork reçoit aussi une copie de la file descriptor table du parent.

le process **init** est exécuté après le démarrage

shell est le process parent des commandes exécutées dans le terminal

GUI manager process pour les commandes lancées depuis une interface

syscalls --> to access resources from processes **and** to spawn/delete, etc. processes from processes

Lundi 3 mars

💡 Méthode : comment représenter l'image mémoire d'un programme ?

On dessine le text segment, le data segment, le heap segment et le stack segment.

On ajoute le code dans le text segment.

On ajoute les variables globales dans le `data` segment (c'est-à-dire toutes les variables au dessus du `int main() {}`).

On ajoute dans le stack :

- les `return pointer` quand on appelle une fonction (à la ligne +1 de l'appel de la fonction)
- les variables locales (en `uninitialized` si elles sont plus loin dans la fonction / pas encore initialisées, ou avec leur valeur si elles sont déjà initialisées)

Convention appeleur/appelé

- le caller doit sauver dans les **saved registers** les valeurs importantes
- le callee doit sauver dans le **stack** les **saved registers** s'il prévoit de les modifier (ou de faire un autre appel de fonction), puis les restaurer dans les **saved registers** avant de `ret`
- le caller doit donner ses arguments via les registres `a0`, `a1`, etc.
- le callee doit renvoyer la valeur de retour

→ le callee ne doit **jamais écrire** dans le stack frame du caller ! c'est ce qu'on appelle le **stack smashing**

Mémoire entre threads/processes

Chaque process dispose de son propre espace d'adresses virtuelles. Si deux processes référencent la même adresse virtuelle, ils tomberont sur deux adresses physiques différentes !

Si deux threads du même process accèdent à la même adresse virtuelle, ils tomberont sur la même adresse physique ! En effet comme ils n'ont pas les mêmes registers (chacun ont leur propre) -- ils communiquent via la mémoire principale.

Ce qui est **sécurisé** et assure que deux process ne peuvent pas accéder à leurs données entre eux et à la fois une **illusion**, car le process pense qu'il a la mémoire principale pour lui. C'est pour ça qu'on parle de **safe illusion**.

Une première méthode, c'est que l'OS stocke, pour chaque process une adresse **base** et une adresse **bound**. Ainsi, quand le MMU (Memory Management Unit) reçoit une adresse virtuelle, il vérifie les bounds (c'est-à-dire que `base + input < bound`), et va ensuite faire `physical = base + input`.

Segmentation et paging : utiliser des chunks continus

On appelle **segmentation** un système de chunks à taille variable, peu utilisé en pratique.

C'est par exemple ce qu'on utilise en base and bound.

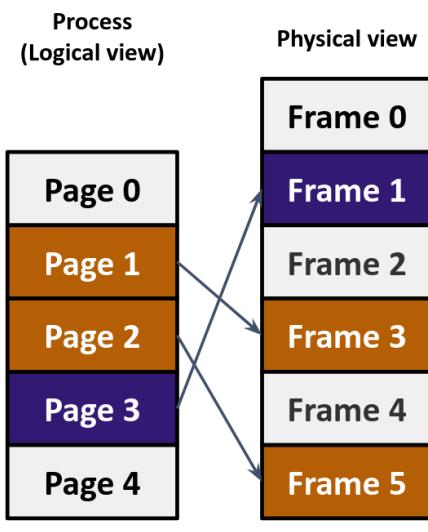
L'intérêt avec cette méthode, c'est que l'image mémoire d'un process est **continue** dans la mémoire ! (elle démarre à `start` et finie à `bound`). Elle nécessite aussi un hardware simple.

Problèmes : **external fragmentation**, des trous de mémoire inutilisables apparaissent entre les segments. Si un segment doit s'agrandir mais qu'un autre le bloque, il faut le déplacer, ce qui est coûteux.

On a vu en comparaison le **paging**, où chaque chunk (chaque **frame**, c'est-à-dire une série d'adresses physiques, est associé à une page d'un programme). Ces frames ont une taille fixe. L'intérêt c'est qu'on a pas besoin d'une range d'adresses continue en mémoire. On peut redimensionner facilement la mémoire en allouant et désallouant des pages/frames. (c'est de la **fragmentation interne** -- il y a des petits trous au sein des frames allouées parce que si on veut stocker 6Kib on est obligés de prendre un segment de 8Kib par exemple si c'est la taille des frames).

ⓘ Où se situe le code de l'OS ?

Le système d'adresses virtuelles nous permettent de toujours mettre le code de l'OS à la même place pour chaque programme (par exemple tout en haut de la range d'adresses virtuelles `0xfffff8000`). Ainsi le programme a juste à `jmp` jusqu'à `0xfffff8000` dans le cas d'un syscall. C'est toujours le même thread qui exécute le code, c'est juste qu'au lieu de lire son code il va lire le code de l'OS.



🔎 Base and bounds OU paging ?

Pros **base and bounds**:

- pas de fragmentation interne

- context switch très rapide (pas de TLB, juste deux registers à changer, base et bound)
- moins de complexité hardware
- demande moins de mémoire (pas de pages à stocker)

Pros paging :

- pas de fragmentation externe
- on peut run des gros programmes car avec le swapping on peut stocker qu'une partie des pages dans la mémoire et le reste sur le disque
- on peut run plein de process en même temps (pas besoin d'allocation contiguës)

Caching

Le cache stocke les valeurs des adresses mémoires accédées récemment ou fréquemment. Le CPU passe toujours par le cache pour accéder à la mémoire.

On a souvent plusieurs niveaux de cache (L1, L2, L3.. du plus rapide/petit au plus lent/gros).

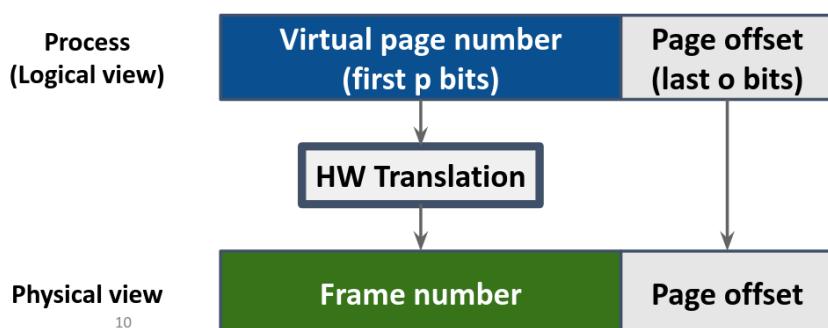
Le cache est séparé entre une partie pour les instructions et une partie pour les données (comme ça on est sûr de garder de la place pour les deux types de cache).

Mercredi 5 mars

Rappel sur les pages

Une page est l'unité minimale d'un espace d'adresse (elle doit être suffisamment petite pour éviter la fragmentation interne). On crée aussi des **super pages**, qui sont des agrégations de plusieurs pages/frames ensemble.

L'OS maintient une **page table** pour garder le mapping entre les pages et les frames. Il y a une table par process.



Si on a une page de taille 2^8 , on a donc un offset de 8 bits.

Il y a un registre spécial qui stocke un pointeur vers la page table.

Lors d'un context switch, le registre qui stocke le pointeur vers la page table est changé aussi.

PTE (Page table entry)

Chaque entrée dans la page table contient un numéro de frame mais aussi... :

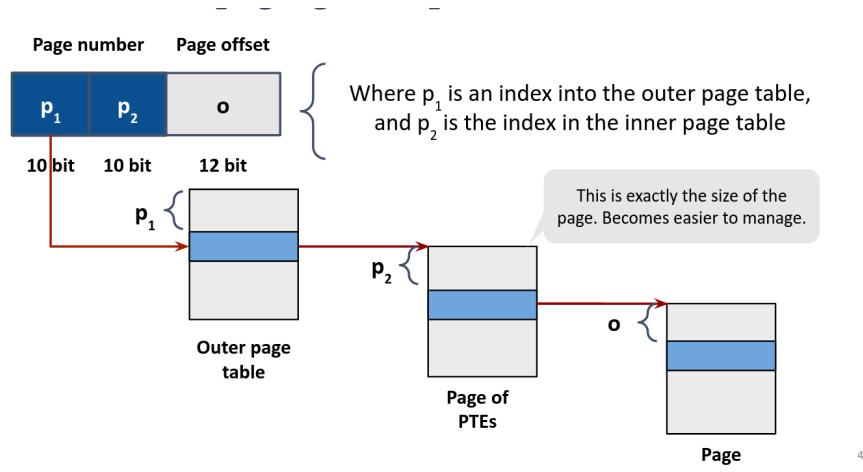
- present bit (*indique si la page est en RAM ou swap, si absent → page fault*)
- protection bit (*définit les permissions : lecture/écriture/exécution*)
- User / Supervisor bit (*contrôle si l'accès est réservé au kernel ou autorisé à l'utilisateur*)
- Dirty (*indique si la page a été modifiée et doit être écrite sur disque avant d'être remplacée*)
- Access/Reference bit (*marque si la page a été récemment utilisée, utile pour l'algorithme de remplacement*)

En comparaison on a vu que les page tables étaient **linéaires**. Si une PTE fait 4B, et qu'on a des adresses virtuelles de 32 bits dont 12 bits pour l'offset => on a 20 bits pour les entries donc 2^{20} PTE donc $2^{20} \cdot 4B$ pour la table.

Si on veut accéder à la PTE de la page 6, on fait `pageTable[6 * 4B]`.

Problème : les **page tables prennent trop de place** ! La solution serait de créer des pages plus grandes ? mais on augmente la fragmentation interne !

Une solution est de créer des **multi-level page table**



Comme ça, pas besoin de créer les tables inutiles !

TLB (translation lookaside buffer) est un cache des adresses virtuelles récemment converties (le CPU va toujours voir dans le TLB. s'il y a un TLB miss, on doit faire tout le travail d'aller dans chaque niveau de page table avec le MMU)

Le TLB n'est pas dans la mémoire, c'est du hardware.

Certaines pages non utilisées sont stockées sur le disque (temporairement), et quand l'OS les charge il y a une page fault, puis il les fait revenir dans la mémoire principale.

Mercredi 12 mars

On a des blocks dans un appareil de stockage (par exemple, un bloc est un stockage de 4Kb de SSD, c'est la plus petite unite sur laquelle on peut écrire). On veut arriver à gérer ces blocks de façon efficace, c'est-à-dire gérer les modifications et lectures concurrentes par exemple (+ stocker des metadata, un nom de fichier, des permissions, un propriétaire, etc).

Les différents layers

Application

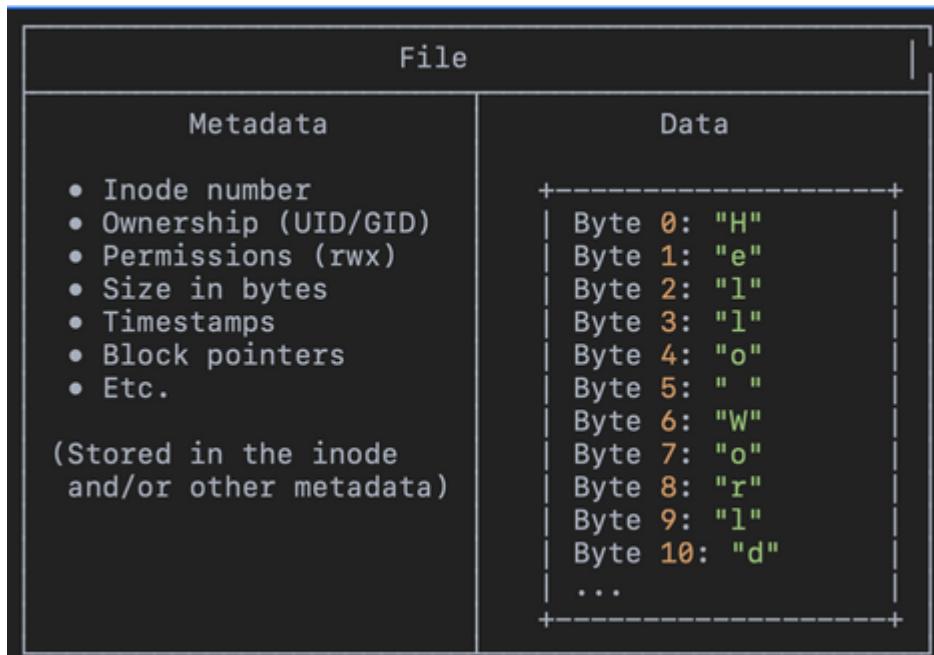
Library (`fopen` , `fread` , `fwrite` , `fseek` , etc. voir `man 3 fread`) qui appellent des commandes Unix/Linux au moyen de **syscalls** (`open` , `write` , etc. voir `man 2 read`)

File System

Physical device

On peut utiliser `strace ./a.out` pour voir tous les calls faits par le programme C a !

Fichiers



- L'humain/utilisateur voit le fichier comme un **chemin**, une chaîne de caractères. Chaque nom local est unique en local, et chaque chemin complet est unique globalement. L'OS voit le fichier et son contenu comme un tableau de bytes (untyped files). Il se fiche et n'a aucune connaissance sur le format du fichier (c-a-d que l'extension dans le nom ne compte pas).
- L'OS voit le fichier comme un inode. Un inode ID est assigné à chaque fichier et est unique **au sein du file system**. Il est recyclé après suppression du fichier. Le inode contient les permissions, la taille, le nombre d'accès, la position des blocks de données, etc. Chaque fichier a exactement un inode.

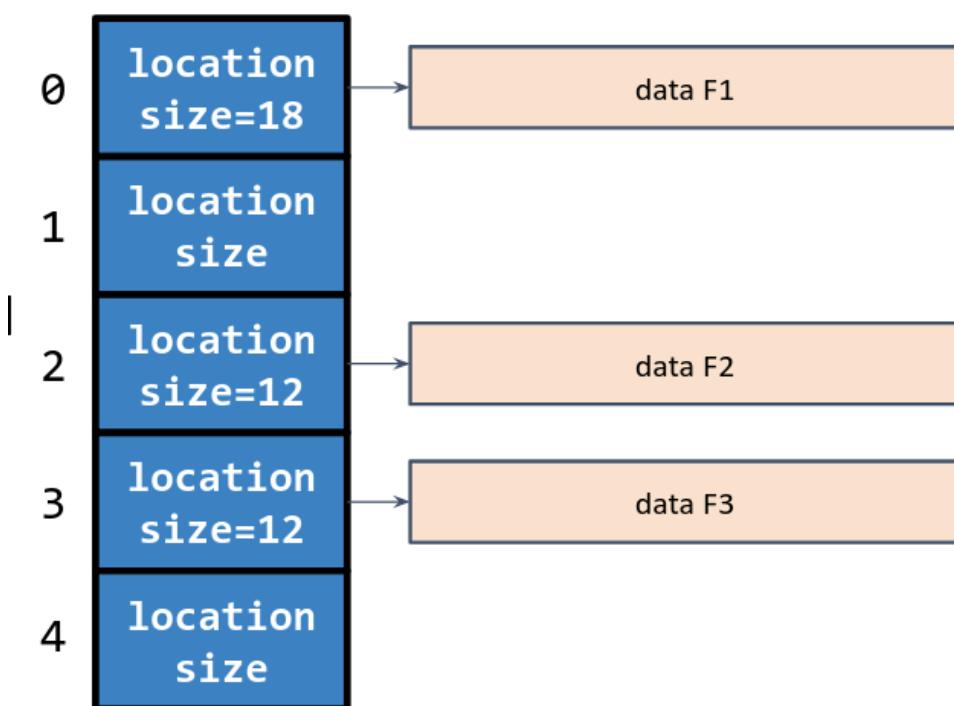
- Le process voit le fichier comme un file descriptor (voir ci-dessous).

ⓘ On peut accéder aux inodes des fichiers/dossiers en utilisant `ls -lih`

Comment le file system gère les inodes ?

Il y a une partie du disque réservée au stockage de la inode table (comme une page table linéaire).

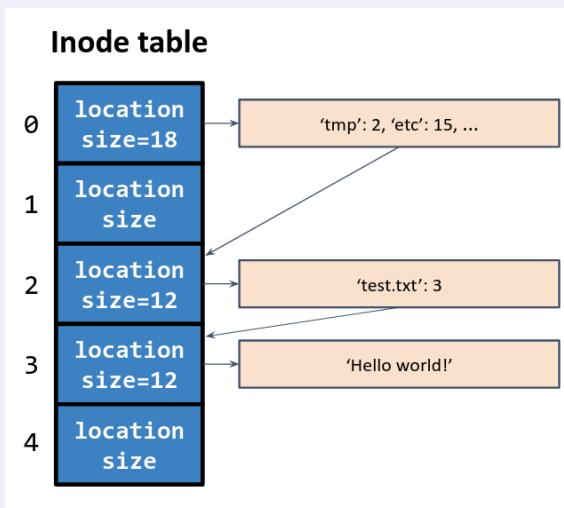
Inode table



Comment passer d'un path à un inode ?

Le **inode** ne stocke **pas** le nom du fichier. En fait c'est le dossier (un fichier spécifique) qui stocke ça. Ils sont marqués avec un flag spécial pour les distinguer des fichiers normaux.

☰ Par exemple, si on veut accéder à `/tmp/test.txt` :



On sait que le inode de `/` est à la position `0` dans la `inode_tables`. On y trouve la référence vers les données de `/` ("location"). Quand on regarde dans les données de `/`, on trouve les références des inodes des dossiers et fichiers stockés dans `/`. On va à la `inodes_table[2]` puis on trouve la référence vers les données de `/tmp` ! et ensuite on charge les données et on trouve `Hello world!` !

Les inodes ont toujours la même taille, que ce soit pour un fichier, pour un dossier, etc.

Bits de permission

Les neufs caractères après `d` ou `.` sont les bits de permission.

- `rwx` pour propriétaire, groupe, tout le monde
 - Owner can read and write; group and others can just read
 - `x` set on a file means that the file is executable
 - `x` set on a directory: user/group/others are allowed to cd to that directory()

Références vers des fichiers (links)

Si on fait un **hard-link**, on va lier le nouveau nom au même inode que le fichier original. Si on supprime le nouveau nom, on ne va pas supprimer le fichier (l'ancien pointera toujours vers le inode).

Si on fait un **soft-link** (lien symbolique), on va mapper logiquement le nouveau nom de fichier au fichier cible. Ce nouveau fichier aura un nouvel inode.

ⓘ Le inode contient une propriété qui est le nombre de références.

Si le nombre de références atteint 0, le inode du fichier est supprimé. Dans le cas d'un hard-link, le nombre de référence est augmenté de 1 quand créé (et diminué de 1 quand supprimé), mais pas dans le cas d'un soft-link.

Vue du process : file descriptors

On peut tout faire avec les chemins des fichiers + un inode/device IDs (pour savoir où trouver la inode table, puis que chercher dans la table) mais c'est très long! on doit faire tout le chemin à chaque fois.

C'est pour ça qu'on stocke le inode final du fichier dans une sorte de cache par process (**file descriptor table**). C'est une table linéaire qui stocke la liste des fichiers ouverts par process. Elle stocke aussi l'offset de lecture dans chaque fichier ("là où on en est").

Les trois premières entrées sont réservées au STDIN, STDOUT et STDERR.

Si on lit 23 bytes de `out.txt`, l'offset du fichier est augmentée.

Le 3 est stocké dans le `fd1`.

Si on réouvre le même fichier, un nouveau file descriptor est créé.

💡 On peut ouvrir les fichiers avec des FLAGS

- `O_RDONLY` : ouvre le fichier en lecture seule
- `O_RDWR` : ouvre le fichier en lecture et écriture
- `O_CREAT` : ouvre le fichier et le crée s'il n'existe pas déjà
- `O_TRUNC` : ouvre le fichier, et s'il existe déjà, le vide (il est donc incompatible avec le flag `O_RDONLY`)

Mount points

Un système est composé de plusieurs file systems. Par exemple, on peut avoir un disque très rapide pour les tâches du quotidien, un disque plus lent pour les sauvegardes, etc.

Tous les systèmes de fichiers ont une racine commune, le `/` ! Par exemple `/home` peut être un système de fichiers différent.

```
mount <device> <directory-to-mount>
```

⚠️ Qu'est-ce qui est stocké `/b/c` est un nouveau file system, `/b` un autre, qu'est-ce qui est stocké dans la inode table de `b` ?

Rien. L'OS va voir qu'il n'existe pas de inode pour `b` donc il va chercher s'il y a un autre file system monté à cet endroit-là.

Comment implémenter un file system?

Un **disque** est divisé en plusieurs partitions. Le secteur 0 du disque stocke le MBR (master boot record), qui contient :

- le bootstrap code (qui est chargé et exécuté par le firmware. le firmware c'est le code qui tourne sur le disque (pour faire tourner la tête de lecture par exemple). C'est une API entre le hardware et l'OS).
- la table de partitions (les adresses de début et de fin de chaque partition)

Le premier bloc de chaque partition a un bloc de boot, qui est chargé.

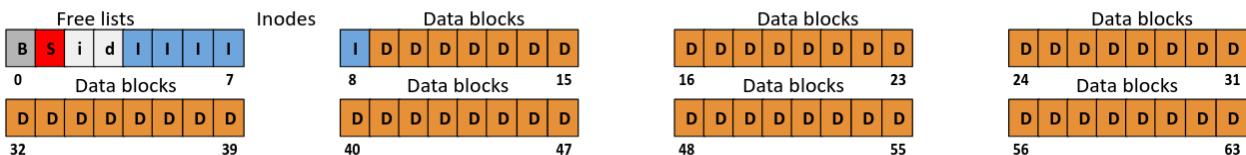
Qu'y a-t-il dans une partition ?

64 blocks, chacun de 4 kb. Certains blocks stockent des données, d'autres des métadonnées.

On peut avoir 5 blocks **i** pour les inodes, un block **d** et **i** pour stocker les blocks libres pour les données et les inodes, et un block **b** pour le boot block et le super block.

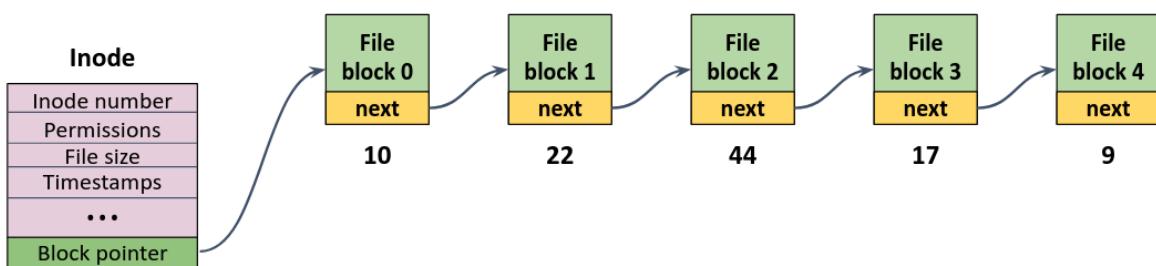
ⓘ C'est quoi un superblock ?

- il stocke le nombre de **inodes**
- le nombre de **data blocks**
- où commence la **inode table**
- il est lu en premier quand on monte le filesystem

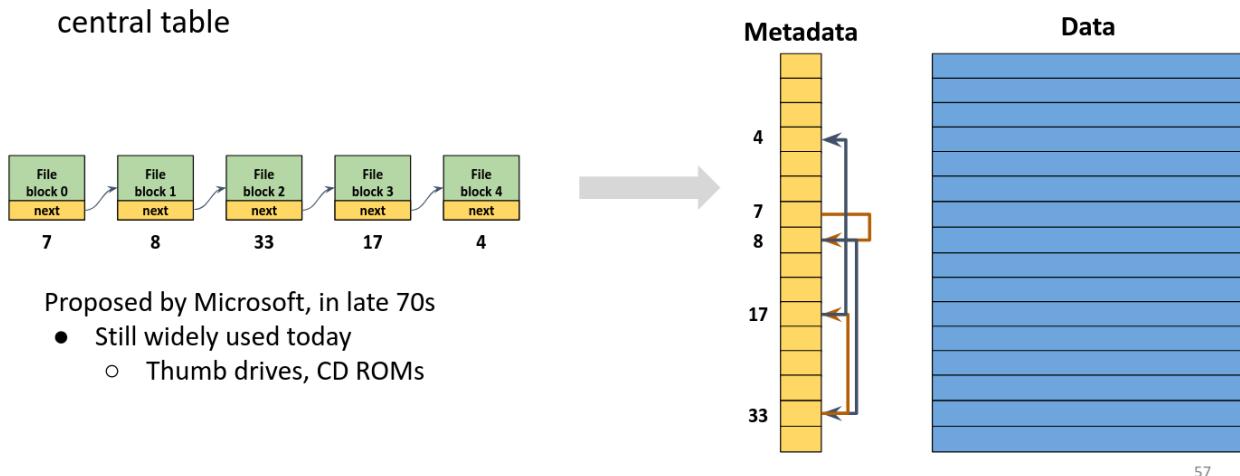


Comment allouer un fichier ?

- **contiguous allocation** : tous les bytes de façon continue
 - mais les fichiers ont pas forcément une taille fixe !
 - et de la fragmentation externe !
 - et si on veut accéder au bloc 2, on doit tout traverser !



- **linked blocks**: on stocke à la fin du bloc le pointeur vers le bloc suivant
 - pas de fragmentation externe
 - ça supporte les fichiers de taille dynamique
 - ça mixe les données et les métadonnées
- **file allocation table (FAT)**
 - on sépare les données et les métadonnées
 - on a pas de fragmentation externe
 - on a besoin que du premier bloc de chaque fichier
 - **mais...** comme les linked list, poor random access (c'est-à-dire accès au milieu de fichier)
 - et puis de la **RAM perdue!** (parce que comme la metadata est stocké dans une giga table et qu'on ne veut pas perdre du temps pour trouver les bouts de fichiers, on doit charger la metadata table dans la RAM, c'est énorme ! pour un disque de 1TB on a 1GB de données à stocker dans la RAM!)



⌚ Question de série

A process can insert entries to a directory inode using a write syscall. (Hint: "Look again at slides 32-35 that discuss the file-system interface. According to those slides, how does a process add a file to a directory?")

Ans : non, le write syscall ne suffit pas, il faut aussi avoir ouvert le fichier avec l'option `O_CREATE` (créé-le s'il n'existe pas).

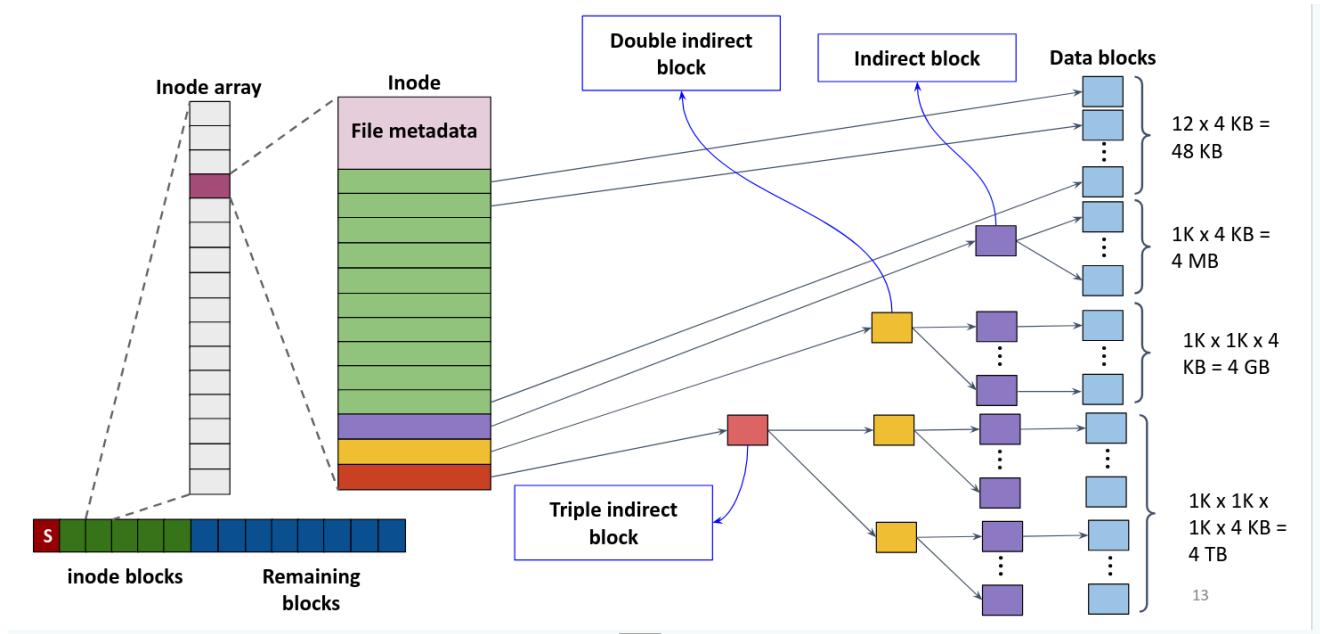
Lundi 17 mars

Multi-level indexing

- Il y a beaucoup de **petits fichiers** et de temps en temps de gros fichiers. Le inode pointe **directement** les pointeurs vers les data blocks pour gagner du temps et de l'espace mémoire. Car :

- moins de redirections
- pas besoin de stocker des tables pour rien !
- **puis**, si les fichiers sont gros, on passe en metadata blocks, un peu comme les multi-level page tables.

Ainsi notre Inode va avoir 12 blocs de direct block, 1 bloc de indirect block, 1 bloc de double indirect block, etc.



Ici nos tables intermédiaires stockent 1000 pointeurs vers des data blocks, donc on a $1,000 * 4\text{Kb} = 4\text{Mb}$!

② Comment allouer un fichier de 5Gb ?

Si on a fichier de 5Gb, on va donc :

- allouer les petits blocs verts du début de 48Kb
- puis ensuite le bloc violet de 4Mb
- puis le bloc jaune de 4Gb
- puis comme ce n'est pas assez, le bloc rouge

Idées clefs :

- structure d'arbre : efficace pour trouver les blocs
- efficace pour lire en séquence (une fois qu'un bloc indirect est lu, on peut lire 1k blocs de plus !)
- structure fixe : simple à implémenter
- asymétrique : supporte des fichiers petits et gros sans coût supplémentaire

Avantages/Inconvénients :

- pas de fragmentation externe
- on a simplement besoin de trouver le premier bloc
- performances sont OK
- mais il y a des reads supplémentaires pour les blocs indirects

Lire depuis un fichier

Question

- on ouvre le fichier en read only
- on trouve l'inode de `root`, on lit, on vérifie les permissions, etc, pour trouver l'inode de `cs202`, etc. jusqu'à l'inode du fichier
- on le lit, on vérifie les permissions, etc, puis on renvoie un fd
- pour chaque `read()` appelé on lit l'inode et on lit le bon data block (en fonction de l'offset) puis on modifie le Last Access Time

Example: Read first 2 data blocks from “/cs202/w07”

- Traverse directories and inodes; Also have to update the last accessed time ...

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs202 data	w07 data[0]	w07 data[1]
<code>open("cs202/w07")</code>			<code>read()</code>			<code>read()</code>			
				<code>read()</code>			<code>read()</code>		
					<code>read()</code>				
						<code>read()</code>			
<code>read()</code>					<code>read()</code>		<code>read()</code>		
						<code>write()</code>			
<code>read()</code>					<code>read()</code>			<code>read()</code>	
						<code>write()</code>			

19

Les `write` sont là pour le Last Access Time.

Écrire dans un fichier

Question

- on va peut-être allouer des nodes en plus ! (l'écriture n'est pas forcément in place). pour ça on doit lire les blocks libres et peut-être modifier l'inode

- si on doit créer le fichier, on doit aussi modifier l'inode du dossier ! et si le dossier est plein, on doit aussi allouer plus de data blocks pour lui...

Example: Create/write first 1 data blocks “/cs202/w07”

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs02 data	w07 data[0]
open("cs202/w07")			read()			read()		
				read()			read()	
		read() write()						write()
					read() write()			
				write()		read()		
write()		read() write()						write()

22

- le file system se rend compte avec le read du `cs202/data` que le fichier n'existe pas !
- il va chercher dans le inode bitmap **qui stocke les inode libres**
- il va allouer un nouveau block et modifier la taille du dossier, etc.
- comme le fichier est vide, on veut allouer un block de données
- le fs trouve un data block vide dans le data bitmap, il **écrit** dans le bitmap pour mettre à jour, il écrit **dans** les données, puis il **écrit** dans le inode

(rappel que le bitmap est stocké dans le superblock)

Comment mesurer la perf ?

- I/O operations per second (IOPS)
 - le débit (throughput)
 - la latence
- on peut améliorer les perfs avec le **caching** (éviter les ops non nécessaires), ou le **batching** (grouper les opérations pour augmenter le débit, sans doute au prix de la latence)

Block cache

L'OS et le FS vont allouer une page pour certaines paires (`inode, offset`) dans la mémoire principale et va lire là-bas après. Donc pas de I/O après la première lecture.

Le FS a un buffer cache qui stocke ce page frame number à partir de l'inode et de l'offset.

Batching operations

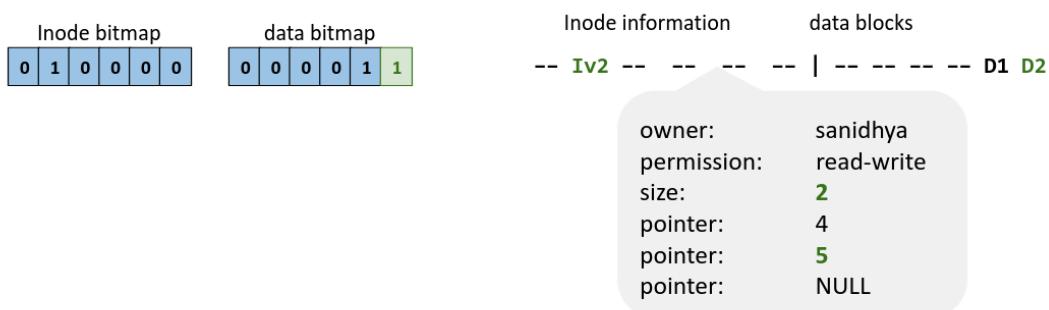
Chaque opération I/O a de la latence et peu de concurrence, donc on préfère quelques gros transferts.

Pour ça, on peut essayer de **retarder les write**. Les faire attendre au plus 30 secondes, les réorganiser, etc. mais... le contenu est perdu si l'OS crash !

Où write-through caches --> écrire de façon synchrone, mais il y a des coûts de perf.

Crash consistency

Example: Updating a block in a file system



- Suppose we append a data block to a file
- Add new data block D2
- Update inode
- Update data bitmap

What if a crash or power outage occurs between writes?

On veut écrire D2 .

- si seul D2 a été écrit c'est **OK**, le write est perdu mais le FS est consistent
- par contre, si le inode est écrit c'est **PAS OK**, car le inode dit que data est utilisé alors que le bitmap dit qu'il est libre !
- si le data bitmap seulement est écrit c'est **PAS OK**, il dit qu'il est pris alors qu'aucun inode ne pointe vers lui

FSCK (file system checker)

Périodiquement, après un crash ou un certains nombres d'opérations, cet outil va vérifier la consistence du système.

☰ Par exemple, le FSCK peut corriger le nombre de liens pointant vers un inode (quand on crée un lien symbolique ou un hard link).

☰ Le FSCK peut déplacer TODO lost+found

☰ Si deux inodes pointent vers le même block de données (ça ne devrait pas arriver), il peut copier le data block vers un nouveau et modifier le lien.

Problèmes :

- c'est lent
- c'est pas toujours correct (il essaye de retrouver un état consistant, mais est-ce que c'est le **bon** état ? consistency ≠ correctness)

Journaling (logs)

Objectifs :

- limiter le travail à faire to recover
- obtenir l'état **correct** et non plus seulement l'état consistant
- être plus rapide (plus besoin de scanner tout le disque)

→ écrire dans le journal **avant** d'écrire dans le disque (**write-ahead logs**) et écrire ce qu'il y avait avant d'overwrite le contenu d'un fichier

Transactions

Propriétés :

- **atomic** : soit tout fonctionne, soit rien
- **consistent** : amène toujours à un état correct
- **isolée** : opérations n'interagissent pas entre elles
- **durable** : une fois qu'elle est complétée, les effets sont persistents

Il peut y avoir deux résultats : un **commit** (la transaction est un succès), un **abort** (elle a échouée, complètement)

Journaling can apply to both data and metadata blocks



TxBeg indique le début de la transaction, **TxEnd** la fin.

Data journaling: an example



- Let's add a new block **D2** to the file
- Three easy steps:
 - Write to the **log**, these 5 blocks: **TxBeg** | **Iv2** | **Bv2** | **D2** | **TxEnd**
 - Write each record to a block for ensuring atomicity
 - Write the blocks **Iv2**, **Bv2**, **D2** to the file system structure place (**checkpoint**)
 - Mark the transaction free in the journal (i.e., remove it)
- If crash happens before the log is updated: **Ignore changes if no commit**
- If crash happens after the log is updated: **Replay changes in log back to the disk**

② mais comment on fait pour le swapping ? on peut pas écrire directement dans les blocks ?

on crée un fichier pagefile.sys ou qqch de similaire.

Mercredi 19 mars

Le FS fait des appels IO au disque, l'OS n'a pas à les faire.

Chaque appareil donne au système une API IO d'entrée et de sortie.

Chaque appareil expose un status (busy/read), CMD (ce qu'on veut faire), DTA (lui envoyer des données..).

- à l'intérieur de l'appareil, il y a un micro-controller (avec un CPU, de la RAM, du stockage, des puces custom...)
- l'OS configure l'appareil et communique avec lui au moyen d'un protocole et utilise un **driver**. Le controller de l'appareil envoie des signaux au CPU soit par du **polling** (keep the CPU busy for nothing) ou des **interrupts**.

CPU communique avec l'appareil

Via **memory-mapped IO** (des instructions load/store). L'OS map l'appareil sur une adresse physique.

avantage :

- supporte tous les appareils

Paramètres pour l'IO

Certains appareils fonctionnent par **byte** (par exemple le clavier, un caractère par caractère) ou par **block** (par exemple un disque).

Certains appareils permettent uniquement l'accès **séquentiel** (tape), tandis que d'autres permettent l'accès **random** (disques..).

Interrupts

- les interrupts sont gérés de façon centrale dans un thread dédié de l'OS
- quand l'interrupt arrive, on réveille un thread kernel qui attendait un interrupt

est-ce que le polling est parfois mieux que l'interrupt ? oui, par exemple si il y a énormément de paquets qui arrivent en même temps, par exemple plus vite que interrupt handling + context switching. **livelock** -> on ne traite plus rien donc les systèmes réels utilisent les deux
on peut aussi utiliser du delay and batch pour éviter beaucoup de contexts switch (mais augmente la latence)

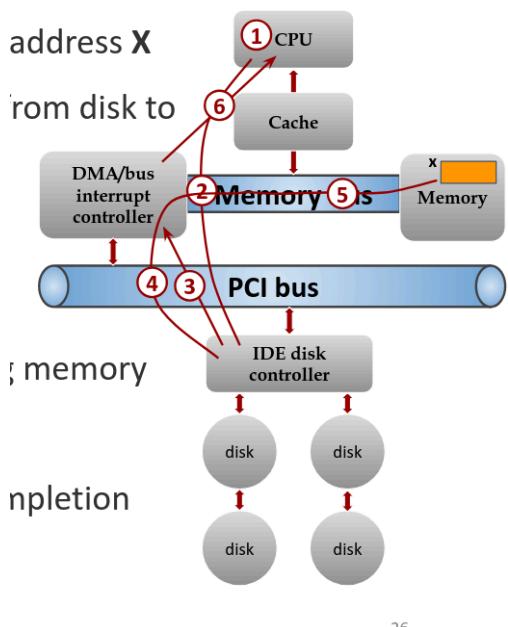
DMA/PIO

Il y a deux moyens d'envoyer des données au contrôleur de l'appareil :

- **PIO (programmed IO)** : le CPU envoie les **données** à l'appareil (prend un temps de CPU proportionnel à la taille des données, efficace pour les petits transferts)
- **DMA (direct memory access)** : le CPU dit à l'appareil **où sont les données** (efficace pour les gros transferts, on doit donner l'accès au bus à l'appareil)

Fonctionnement du DMA

- le driver de l'appareil (du disque) reçoit une requête pour transférer des données à l'adresse **X** (via le memory bus)
- le driver du disque dit au disk controller (interne) de transférer **C** bytes du disque vers le buffer à l'adresse **X**
- le disk controller commence un DMA transfer
- le disk controller envoie chaque byte au DMA
- le DMA controller transfère chaque byte à l'adresse **X** augmente l'adresse mémoire, et diminue **C** jusqu'à ce que **C = 0**
- quand **C = 0**, le DMA **interrupt** le CPU pour dire que le transfert est fini



26

Trop d'appareils...

Challenge : trop d'appareils ont des protocoles différents

Le driver implémente une API. Les appareils similaires (de la même class) utilisent la même API. Le kernel supporte donc quelques APIs, une par classe.

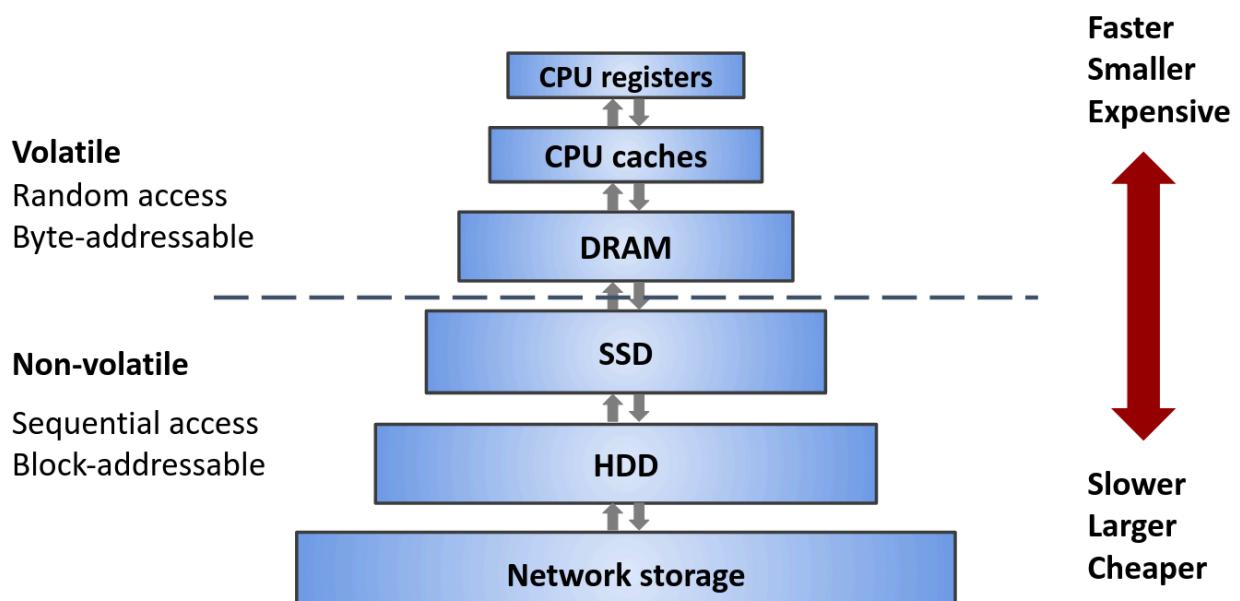
Comment bien designer cette API ?

Abstraction stack

Les systèmes IO sont accédés par une série de layer d'abstraction.

Par exemple si on charge un block de 16Kb, alors que le disque ne supporte que des blocks de 512, il va split les blocks.

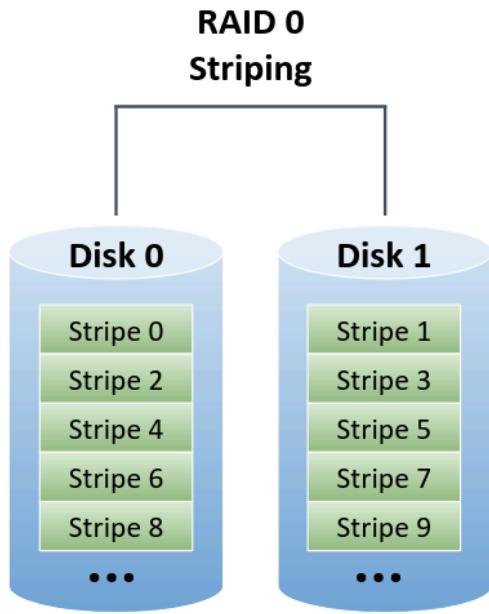
Storage hierarchy



Volatile : disparaît quand shut down

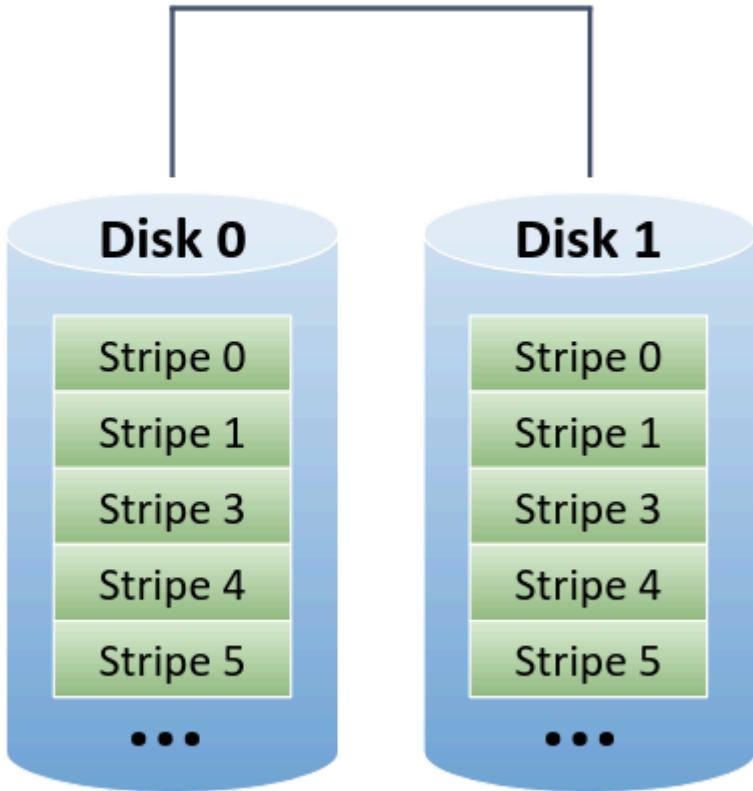
RAID (redundant array of inexpensive disks)

- meilleur débit (on peut écrire à plusieurs endroits en même temps)
- meilleur MTTF (mean time to failure -- temps avant que le système ne fonctionne plus)

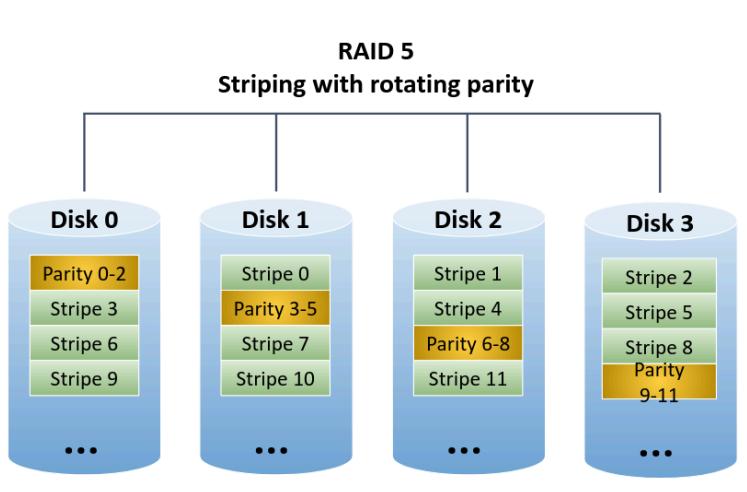


RAID 0 : dans l'exemple ci-dessus, pas de redondance, on fait que du data striping pour un meilleur débit.

RAID 1 Mirroring



RAID 1 : bien quand on perd un disque, mais on a qu'un disque, c'est cher, etc. et ça ne gère pas la corruption



RAID 5 : utilise la parité

$$P_{i-j} = S_i \oplus S_{i+1} \oplus \dots \oplus s_j$$

Si un disque meurt, on peut reconstruire les données en xorant les drives restants ! à partir du 2 et du parity 0/2, on peut retrouver le 0. à partir du 0 et du parity 0/2, on peut retrouver le

2, etc.

problème : les write sont un peu compliqués

Lundi 24 Février

⌚ Rappel polling/interrupts/PIO/DMA

On a vu qu'en faisant du polling, le CPU devait attendre le résultat et qu'on perdait des cycles pour rien. On a ensuite vu les interrupts. Ils ont gérés dans un thread dédié de l'OS.

On a ensuite vu que pour les transferts de données, on pouvait soit utiliser PIO (envoyer les infos au controller), ou alors utiliser DMA (le CPU donne l'endroit où les données sont situées et le transfert se fait ensuite).

CPU Scheduling

Time sharing : chaque tâche tourne toute seule et on change rapidement de l'une à l'autre.

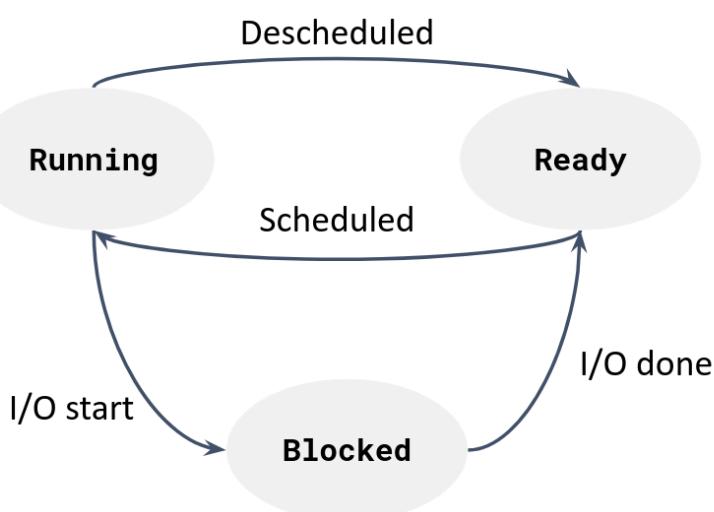
Space sharing : chaque tâche obtient une portion de l'espace disponible.

L'objectif : donner l'illusion à chaque thread qu'il est tout seul dans le CPU.

La réalité c'est que le CPU est partagé entre tous les threads.

OS Scheduler

- il doit choisir le prochain thread à garder
- il maintient la liste des processus et des threads



Jusqu'ici, nous n'avons pas vu la policy appliquée pour choisir le prochain thread.

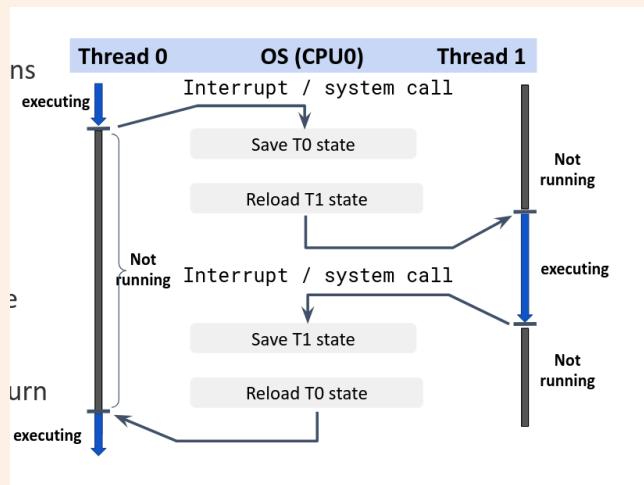
⌚ Pourquoi introduire le thread scheduling ?

- par exemple, si l'OS est en kernel mode et ne peut plus revenir au thread normal (s'il y a eu une erreur p. ex. une opération invalide), ou un system calls
- le thread a déjà utilisé tout son temps (il faut exécuter les autres)

② Comment se passe le switch ?

On fait un context switch (on change les registers, la table de translation des adresses, etc). Plus précisément :

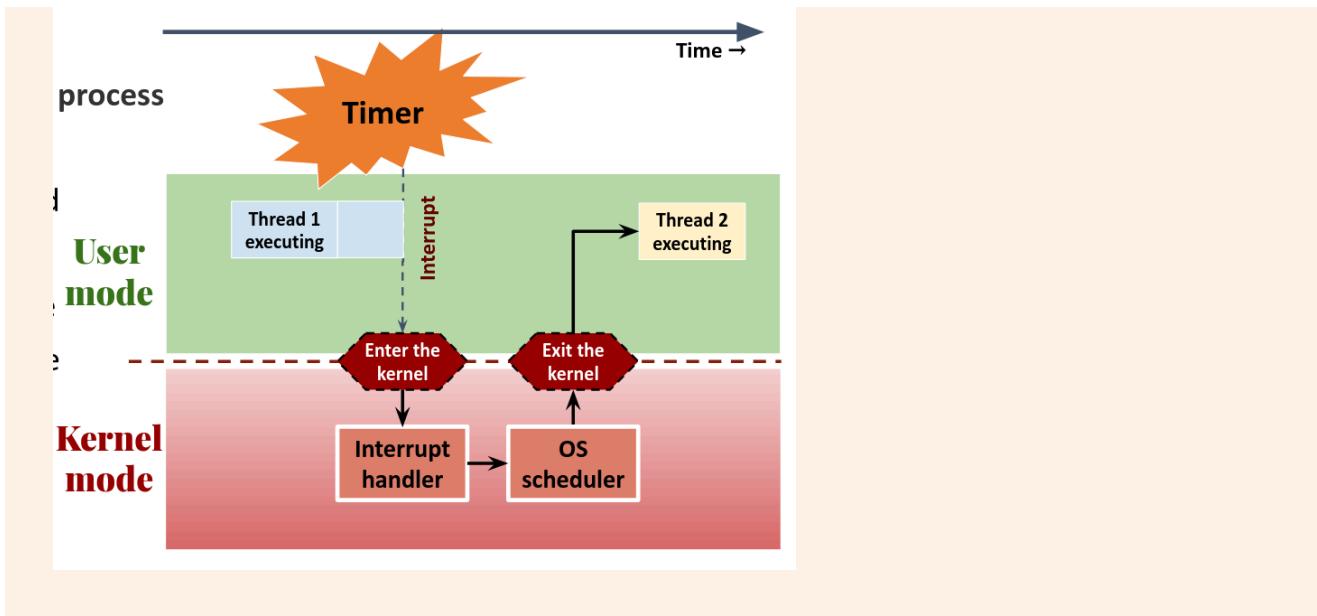
- on sauvegarde le state du thread précédent dans la mémoire
- on choisit le prochain thread
- on restore le state du prochain thread
- on passe le contrôle en faisant un return from trap pour lancer le prochain thread



il y a un coût à ce context switch, un moment où rien ne tourne

③ Comment gérer les threads qui ne se comportent pas bien ?

On utilise des **timer interrupts**. Quand le timer expire, le CPU est interrompu et passe en kernel mode. L'OS Scheduler est invoqué et le scheduler fait un context switch.



Quelle scheduling policy ?

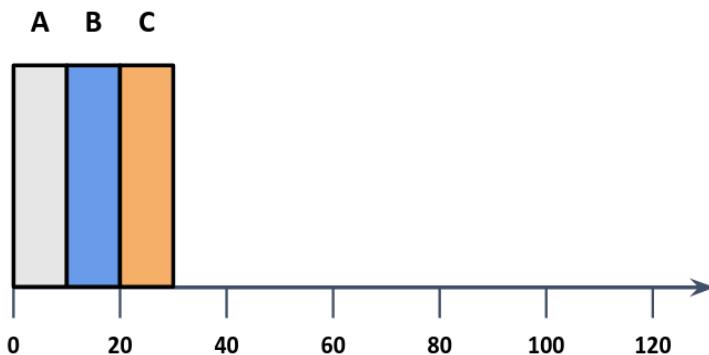
Comment choisir le prochain thread ?

Deux metrics utiles :

- **utilization** : quelle fraction du temps le CPU passe à exécuter un thread. Objectif : maximiser ce temps.
- **turnaround time** : le temps total que les threads mettent à compléter leur tâche. Objectif : minimiser ce temps. $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$.

FIFO First in, first out

On a trois threads A, B, C qui prennent **chacun** 10 secondes. Ils arrivent à peu près en même temps.



ça marche bien quand on sait le temps que va mettre chaque thread, ce qui n'est généralement pas le cas.

$$T_{\text{arrival}} = 0$$

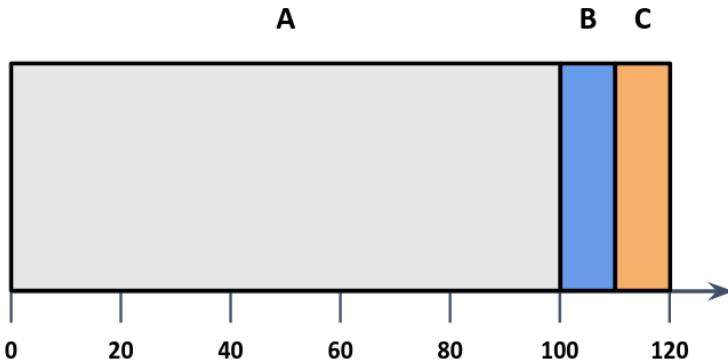
$$T_{\text{completion A}} = 10$$

$$T_{\text{completion B}} = 20$$

$T_{\text{completion C}} = 30$

average turnaround time is 20.

mais que se passe-t-il si A prend 100 secondes ?



average turnaround is 110 (!)

Convoy effect : un certain nombre de clients potentiellement rapides se retrouvent derrière un client très long.

Shortest job first (SJF)

On va choisir le thread le plus rapide à exécuter. Le turnaround baisse beaucoup! (approx. 50)

Mais qu'est-ce qu'il se passe si A arrive à $t=0$ et doit tourner pendant 100 secondes puis B et C arrivent à $t=10$ et tournent pendant 10 secondes ? A est schedulé et on n'a pas prévu de l'arrêter !

Le turnaround est de approx 103.

② polite vs forced scheduling

- FIFO et SJF sont des **non-preemptive**. Ils ne switch que lorsque le thread en cours a fini son exécution.
- **Preemptive** schedulers arrêtent l'exécution du thread en cours et switch à un autre de façon forcée pour éviter que le CPU soit monopolisé.

Shorter time to completion first (STCF)

Il étend le shortest job first.

à chaque fois qu'un thread est créé :

- il détermine lequel des jobs restants (**dont** celui en cours) a le temps restant le plus faible
- il le schedule

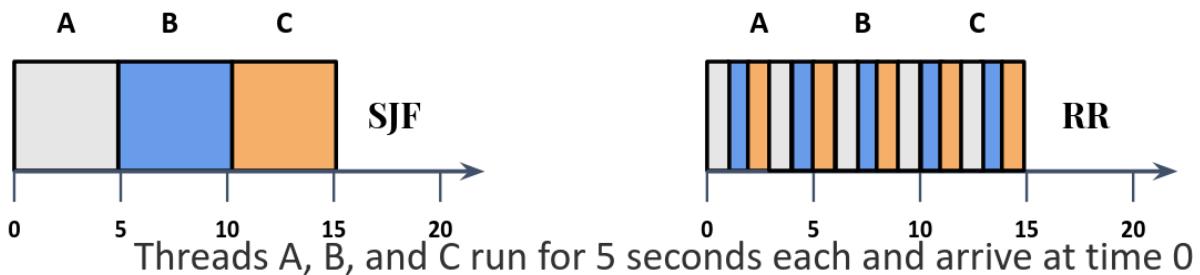
New metric : le temps de réponse

C'est le temps avant que le thread soit scheduled. Les utilisateurs veulent des réponses interactives !

Ce n'est pas du tout pris en compte dans le STCF

Round Robin (RR)

Au lieu de faire tourner les threads jusqu'à ce qu'ils soient complétés, RR schedule un thread pour un intervalle fixe, et switch au prochain thread.



Le temps de turnaround augmente.

Gérer les IO requests ?

```
int traite_arguments(int* nb, char** argv)
{
    int required = 0; // nb d'arguments obligatoires déjà traités
    char const * const pgm_name = argv[0]; // le nom du programme

    ++argv; --(*nb); // passe à l'argument suivant

    while ((*nb) > 0) { // tant qu'il y a des arguments
        if (!strcmp(argv[0], "-P")) { // option -P
            /* par exemple, avec option_P une variable globale,
             * ou mieux : le champ d'une structure passée en paramètre */
            option_P = 1;
        }

        } else if (!strcmp(argv[0], "-i")) {
            // une option avec 1 argument : par exemple -i nom
            option_I = 1;
            ++argv; --(*nb); // passe à l'argument suivant
            if (*nb == 0) { // si l'argument de l'option n'est pas là...
                fprintf(stderr,
                    "ERREUR: pas d'argument pour l'option -i\n");
                return ERREUR_I; // une constante définie globalement
            } else {

```

On veut schedule B pendant que A fait ses requêtes de IO.

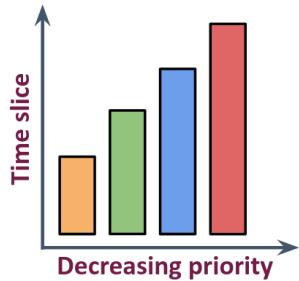
Multi-level feedback queue (MLFQ)

Challenge : le scheduler doit pouvoir supporter de longues tâches dans le background (batch processing) et donner une réponse rapide pour les process interactifs.

Batch-based thread : le temps de réponse n'est pas important (on veut minimiser les context switch)

Interactive thread : le temps de réponse est critique, c'est des bursts courts

Pour cela, MLFQ utilise les past behaviors pour prédire les comportements futurs.



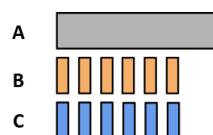
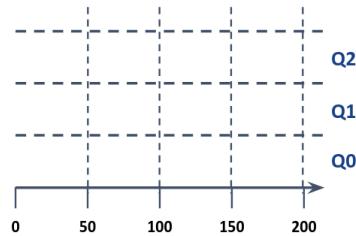
- Les threads de haut niveaux ont un temps de run courts, ceux plus bas un temps plus long.
- Les threads de haut niveaux vont toujours être traités en premier

Règles :

- si priority(A) > priority(B) alors A tourne
- si les deux sont égales, alors on fait du RR (et on change l'intervalle en fonction du niveau)
- les threads commencent tous à une haute priorité (on ne sait pas combien de temps ils vont tourner)
- puis, quand il utilise tout l'intervalle, le scheduler descend sa priorité
- périodiquement, tous les threads sont déplacés dans la topmost queue (priority boosting) pour éviter que les threads interactifs empêchent les threads du bas de ne jamais tourner (**starvation**)

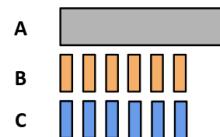
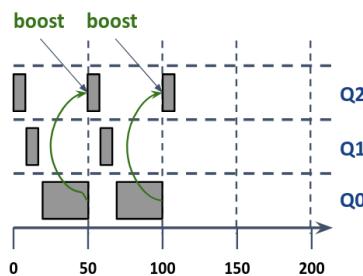
MLFQ example

- 3 priority queues (Q2 highest and Q0 lowest)
- Periodic boosting window: 50 ms (Rule 5)
- Time-slice: 10 ms
- 3 threads
 - A is long running thread
 - B and C are interactive threads issuing IO
- $T_{arrival(A)} = 0$
- $T_{arrival(B)} = T_{arrival(C)} = 100$



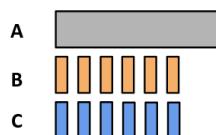
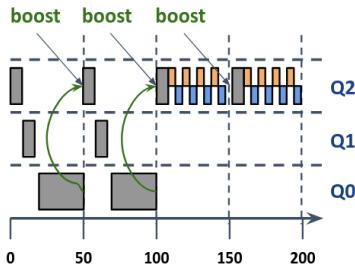
MLFQ example

- A runs for 10 ms in Q2 and gets demoted to Q1
- A then runs for 10 ms in Q1 and demoted to Q0
- A runs for 30 ms and gets **boosted** to Q2 (R 5)
- The same procedure happens until 100 ms



MLFQ example

- A runs for 10 ms in Q2 and gets demoted to Q1
- A then runs for 10 ms in Q1 and demoted to Q0
- A runs for 30 ms and gets **boosted** to Q2 (R 5)
- The same procedure happens until 100 ms
- Process B and C also join Q2
- A scheduled for 10 ms
- B is scheduled and then followed by C
- that are issuing IO requests as well



MLFQ does not starve long running jobs and gives equal time to all jobs

On voit ici que le boost permet à A de tourner de temps en temps au début de chaque boost (et si A s'appelait B alors on aurait ABC, AC, AC, AC, etc. jusqu'au prochain boost)

Mercredi 9 avril

distributed application :

- plusieurs process

- plusieurs ordis possibles
- échanges des messages entre eux

Architecture client/serveur

Le client **fait des appels**, le serveur **y répond**.

HyperText Transfer Protocol (HTTP)

Requêtes courantes :

- GET --> récupère un web object (metadata + contenu)
- HEAD --> le serveur renvoie juste les metadata du web object, mais pas le contenu
- POST --> envoyer des informations au serveur

Web object :

- a une Uniform Resource Locator (URL)
- peut être un fichier image, photo, text, etc.

Web page (type particulier de web object)

- consiste d'un fichier de base (html)
- des objets référencés (images, scripts, etc)

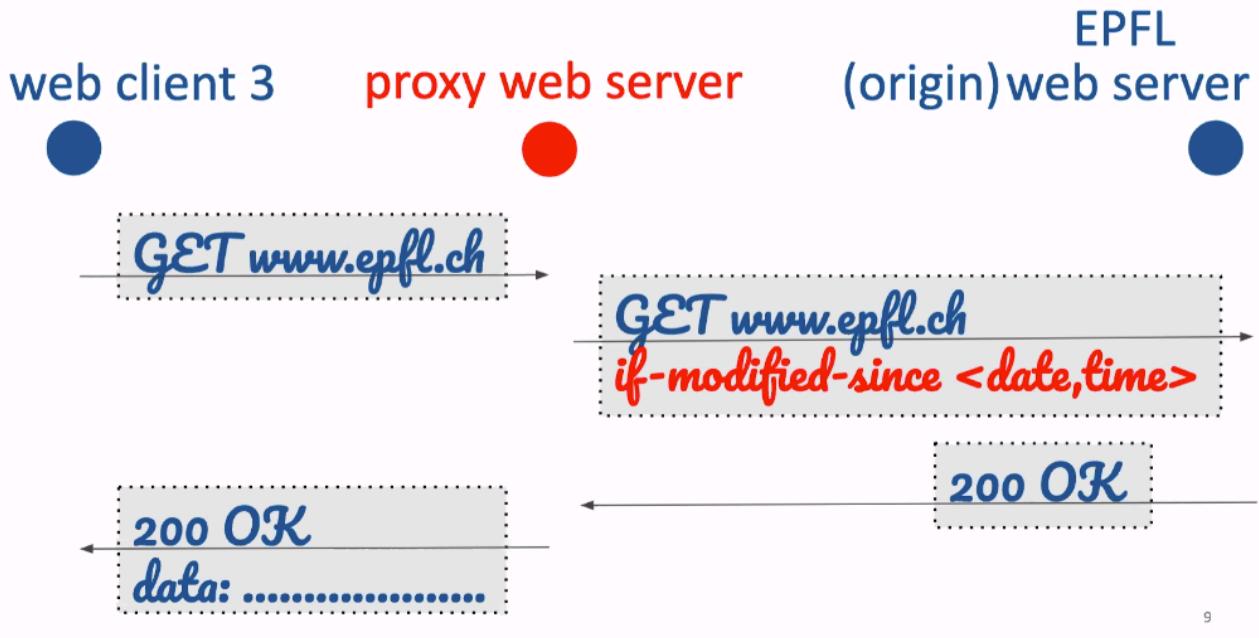
Stateless protocol : chaque requête est indépendante mais.. cookies ! qu'on ajoute à chaque requête

Cookies : state créé par le web server, mais stocké sur le web client. Ils sont envoyés à chaque fois par le client au serveur. facile pour le serveur.

Third-Party cookies : cookies créé par un web serveur, utilisé par un autre serveur

Lundi 14 avril

On a des serveurs proxy qui permettent de stocker les données entre les deux. EPFL origin ne va envoyer que les metadata (taille du fichier, etc) **ou/et** les données si elles ont changées.



9

if-modified-since <date, time>

Une interface réseau est une partie de hardware utilisée pour envoyer et recevoir des informations sur un réseau.

Une adresse IP identifie une interface réseau.

Chaque process a un port unique au sein de l'ordinateur.

Un serveur web peut seulement écouter sur 80, 8080 (http), 443, 8443 (https).

DNS name = host name

② **www?**

C'était pour indiquer qu'on voulait accéder au serveur web!

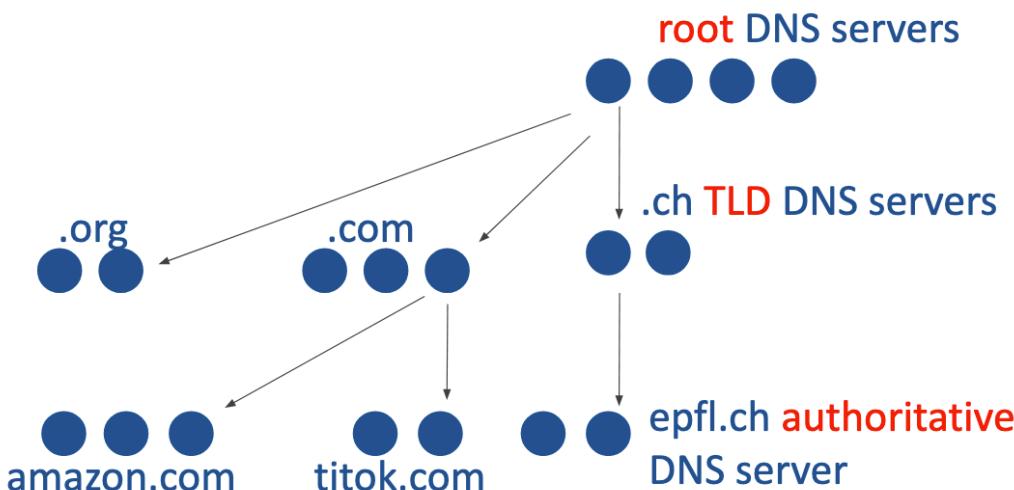
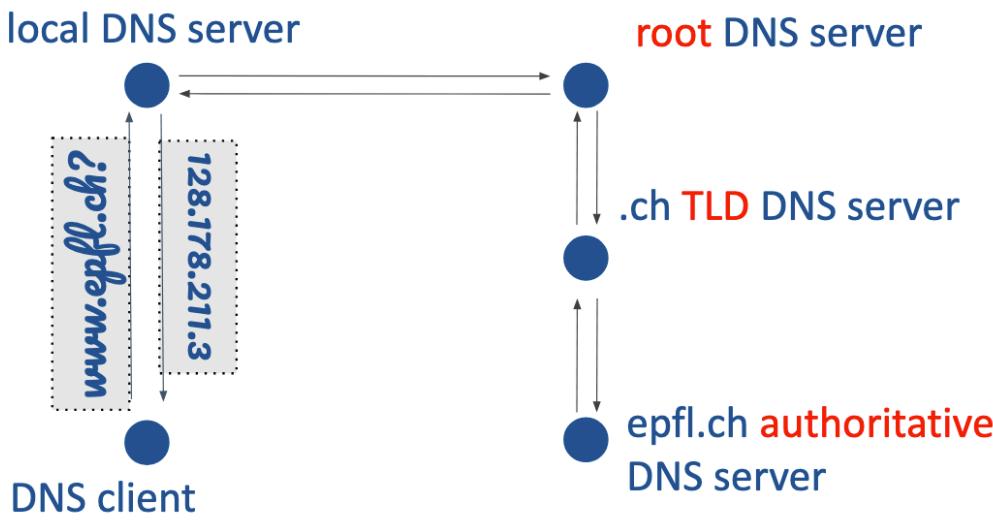
DNS Domain Name System

Il traduit les DNS names en adresses IPs

DNS server port : 53

Il y a des serveurs DNS locaux hébergés à l'EPFL pour que ça aille vite.

Chaque machine a besoin d'au moins une IP de DNS serveur pour que ça marche.



EPFL CS202 Computer Systems

Les authoritative DNS servers gèrent les domaines de second niveau (comme `epfl.ch`). Leur travail est de donner les adresses IPs des DNS servers des domaines qui finissent par `epfl.ch`.

TLDs: top level domains ont des DNS servers dont le travail est de connaître les adresses IPs des DNS servers des domaines qui finissent par ch.

② Qu'est-ce qui se passe si on demande `epfl.abc` ?

Le root DNS dirait : "je ne connais aucun TLD de cette forme, je discard la requête". En vrai les DNS locaux filtrent les requêtes qui n'ont pas de sens.

② Comment créer un nouveau site .ch ?

Périodiquement, les DNS échangent des informations entre eux.

Approche récursive :

- le DNS local demande au root,
- le root demande au .ch ,
- le epfl.ch renvoie l'IP

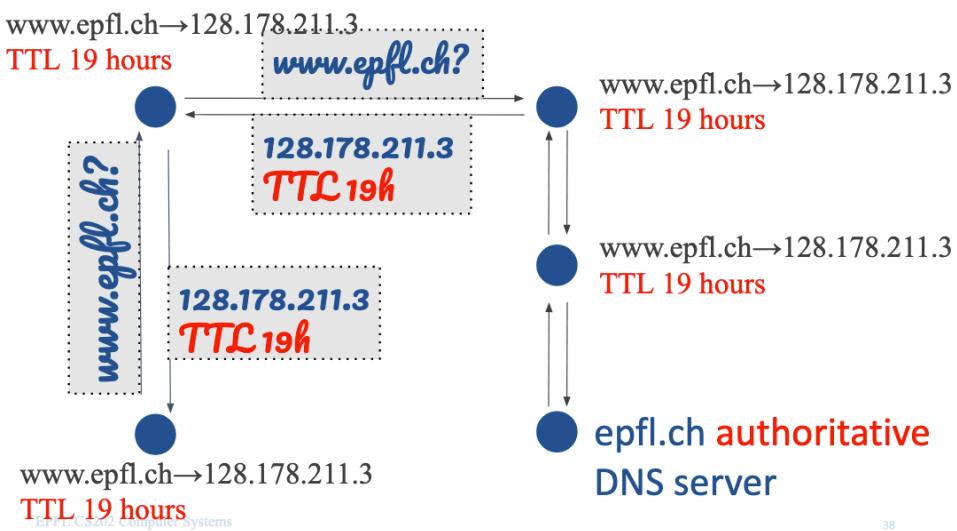
Approche itérative :

- le DNS local demande au root,
- le root renvoie l'adresse du DNS TLD .ch
- le DNS local demande au DNS TLD .ch
- le DNS TLD renvoie l'adresse du DNS authoritative epfl.ch
- le DNS local demande finalement l'IP au epfl.ch

⌚ Comment résoudre moodle.epfl.ch ?

Le second-level domain est epfl.ch . C'est lui (le authoritative DNS server) qui saura quelle est l'IP derrière. L'adresse de l'autoritative server epfl.ch n'est pas forcément epfl.ch ! Le TLD est .ch . Le root est comme d'habitude les 13 serveurs principaux.

TTL : time-to-live



(c'est le serveur authoritative qui décide le TTL et renvoie l'information)

⌚ Quel lien avec les vendeurs de noms de domaine ?

Quand on achète un nom de domaine par exemple `simon.ch` sur Infomaniak, Infomaniak va informer le DNS TLD qu'un nouveau domaine `simon.ch` va maintenant pointer vers le DNS server de Infomaniak. Le authoritative server de `simon.ch` est donc maintenant celui d'Infomaniak. C'est pour ça que dans le dashboard de Infomaniak on peut changer le TTL et ce genre de paramètres.

Mercredi 16 avril

Le serveur utilise le syscall `socket` pour ouvrir une connexion, `const int s = socket(...)`

Il utilise le syscall `bind` pour dire à l'OS que son IP est 8.8.8.8 et son port, `bind(s, [8.8.8.8, 53])`

Le client ouvre aussi un `socket` avec le même syscall.

Il peut envoyer un message `sendto(s, [request], ..., [8.8.8.8, 53])`.

Le client n'a pas besoin de faire un `bind` car l'OS assigne un port aléatoire au client qui fait la requête. En effet le port n'a pas besoin d'être fixe pour le client.

Le serveur fait un `recvfrom(s, ...)` puis un `sendto(s, [response], [18.6.2.5:57050])`.

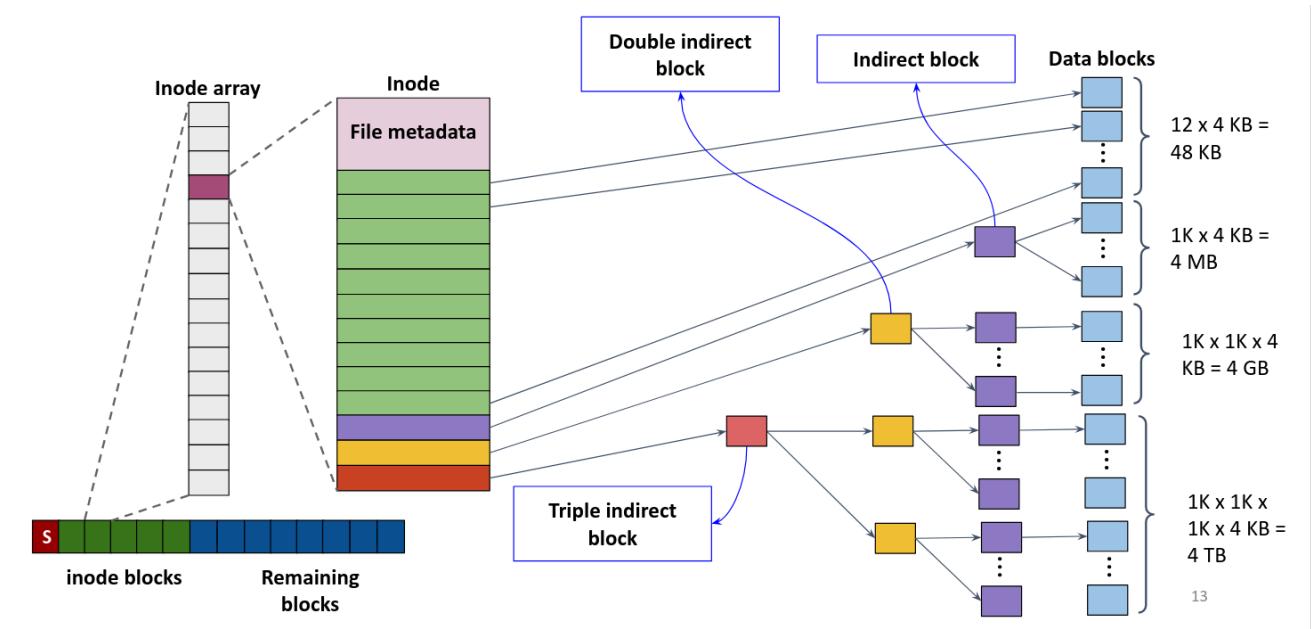
end-system : un appareil qui utilise un réseau

Notre maison est connecté à un central office, à travers des lignes téléphoniques.

La vitesse de cette "phone lines" est entre 10 Mbps et 10 Gbps DSL, FTTH, FTT.

Tout le monde se connecte à ce central office.

(ou on se connecte à un cable model à travers une ligne cable TV, cable head end, 100+ Mbps, DOCSIS. Sauf que cette infrastructure est **partagée**, contrairement aux lignes téléphoniques ! donc on ne peut pas comparer uniquement la vitesse, on doit aussi regarder si l'entreprise partage ou pas notre câble. Et on fait aussi du **broadcast**, parce qu'avant ces câbles étaient utilisés pour la TV! donc quand une réponse arrive, on broadcast et on envoie la réponse à tout le monde ! c'est très adapté.)



Example: Read first 2 data blocks from “/cs202/w07”

- Traverse directories and inodes; Also have to update the last accessed time ...

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs202 data	w07 data[0]	w07 data[1]
open("cs202/w07")			read()			read()			
				read()			read()		
					read()				
						read()			
read()				read()			read()		
					write()				
read()					read()				
						read()			
					write()				

19

swisscom est un tier-1 ISP, il a définitivement des connexions peering entre d'autres majors ISPs.

mais c'est aussi un access ISP !

Le **peering** est un **accord entre deux réseaux** (souvent des fournisseurs d'accès à Internet ou des opérateurs de backbone) pour **échanger directement du trafic Internet**, sans passer par un fournisseur tiers.

- Il peut être :
 - Public** (via un point d'échange Internet, ou IXP)
 - Privé** (liaison directe entre deux réseaux)
- Ce n'est pas une **relation client-fournisseur** : chacun gère ses coûts.

Example: Create/write first 1 data blocks “/cs202/w07”

	data bitmap	inode bitmap	root inode	cs202 inode	w07 inode	root data	cs02 data	w07 data[0]
open("cs202/w07")			read()			read()		
		read() write()		read()			read()	
					read() write()			write()
				write()		read()		
write()		read() write()						write()
						write()		

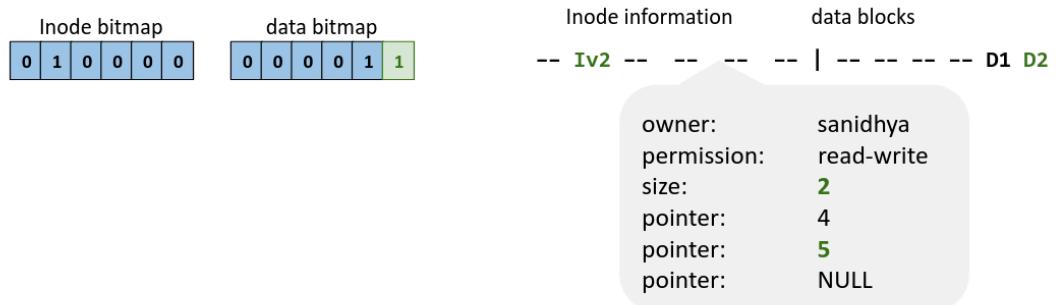
22

On construit un IXP (Internet Exchange Point) pour que si un access ISP veut changer de regional ISP, il puisse changer facilement ! ("please disconnect me from Swisscom and connect me to Sunrise")

Les **off-net caches** ou **edge caches** sont des serveurs de cache placés au plus près des utilisateurs, souvent dans les réseaux des fournisseurs d'accès à Internet (ISP). Leur but est de stocker localement des contenus populaires (comme des vidéos YouTube, des séries Netflix, etc.) pour accélérer l'accès et réduire la bande passante utilisée entre les serveurs d'origine et les utilisateurs finaux.

Dans certaines parties du monde, certaines entreprises riches viennent et créent un ISP gratuit, bien mieux que le ISP original. Mais ils ajoutent des conditions... : vous ne pouvez qu'utiliser Facebook. pas de concurrence.

Example: Updating a block in a file system



- Suppose we append a data block to a file
- Add new data block **D2**
- Update inode
- Update data bitmap

What if a crash or power outage occurs between writes?

Chaque couche, layer, ajoute un header aux données (encapsulation) et enlève les couches au retour (decapsulation).

AF_INET --> on verra plus tard ce que c'est

SOCK_DGRAM --> spécifier UCP/UDP

```
→ ~ dig @dns19.ovh.net androz2091.fr A

; <>> DiG 9.20.4-3ubuntu1-Ubuntu <>> @dns19.ovh.net androz2091.fr A
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34839
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: c589a8fe24ffb5c0010000068055a4837c9ba565179a0f3 (good)
;; QUESTION SECTION:
;androz2091.fr.           IN  A

;; ANSWER SECTION:
androz2091.fr.      3600    IN  A   54.39.102.76
```

```

;; Query time: 20 msec
;; SERVER: 2001:41d0:1:4a8b::1#53(dns19.ovh.net) (UDP)
;; WHEN: Sun Apr 20 22:34:16 CEST 2025
;; MSG SIZE  rcvd: 86

→ ~ dig androz2091.fr A

; <>> DiG 9.20.4-3ubuntu1-Ubuntu <>> androz2091.fr A
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51591
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;androz2091.fr.      IN  A

;; ANSWER SECTION:
androz2091.fr.    3491    IN  A   54.39.102.76

;; Query time: 0 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Sun Apr 20 22:34:22 CEST 2025
;; MSG SIZE  rcvd: 58

→ ~

```

on peut voir qu'il y a le flag `aa` (authoritative answer) quand on contacte directement les serveurs d'OVH !

Lundi 28 Avril

On appelle le total (toutes les couches) un **paquet**. Un packet switch peut implémenter un network layer, et toujours un link et physical layer. Le packet switch peut donc modifier les headers de ces trois couches.

Propriétés d'un network link

- transmission rate (bits per sec)
- length (en mètres)
- propagation speed (meters per sec)

Chaque network link est bidirectionnal.

Il y a des queues à l'intérieur des paquet switches. Elles stockent les paquets (typiquement une par network link).

Et une forwarding table. Elle stocke des metadata et permet de décider où envoyer le paquet.

- le switch extrait les headers du paquet, et consulte la forwarding table.
- il émet une forwarding decision
- il le met dans la queue du link choisi

Chaque queue se vide en fonction du transmission rate.

Store-and-forward : la switch process un paquet et émet une décision après avoir reçu tous les bits du paquet. C'est très long !

⌚ Network congestion

Il est possible qu'une queue se remplisse plus rapidement qu'elle ne se vide ! On peut donc perdre des paquets (plus de place dans la queue), ou avoir un delay.

Caractéristiques importantes pour évaluer les performances d'un réseau :

- **packet loss**, la fraction de paquets qui sont perdus (p. ex. 1%)
- **packet delay**, le temps qu'on met d'aller à la source à la destination (p. ex. 10ms)
- **average throughput** (différence du transmission rate ! le dernier peut être supérieur au avg throughput) : la rate à laquelle la destination reçoit les données (bits per sec)

Journaling can apply to both data and metadata blocks



C'est intéressant d'avoir des chemins parallèles (how long somebody has to wait in order to get on the path vs how long somebody takes to cross).

delay (ne change pas) vs throughput (change avec les chemins parallèles)\

delay : pour les messages rapides

throughput : pour les transferts bulk

Data journaling: an example



- Let's add a new block **D2** to the file
- Three easy steps:
 - Write to the **log**, these 5 blocks: **TxBeg | Iv2 | Bv2 | D2 | TxEnd**
 - Write each record to a block for ensuring atomicity
 - Write the blocks **Iv2, Bv2, D2** to the file system structure place (**checkpoint**)
 - Mark the transaction free in the journal (i.e., remove it)
- If crash happens before the log is updated: **Ignore changes if no commit**
- If crash happens after the log is updated: **Replay changes in log back to the disk**

$$\text{transmission delay} : \frac{\text{packet size}}{\text{link transmission rate}} = \frac{3 \text{ bits}}{1 \text{ Gpbs}} = 3 \text{ ns}$$

- la source doit push chaque bit sur le link (transmission delay du premier link)
- on doit attendre que le dernier bit atteigne la switch (propagation delay du premier link)
- processing delay (delay de la switch)
- on doit push chaque bit sur le link (transmission delay du 2nd link)
- on doit attendre que le dernier bit atteigne la destination (propagation delay du 2nd link)

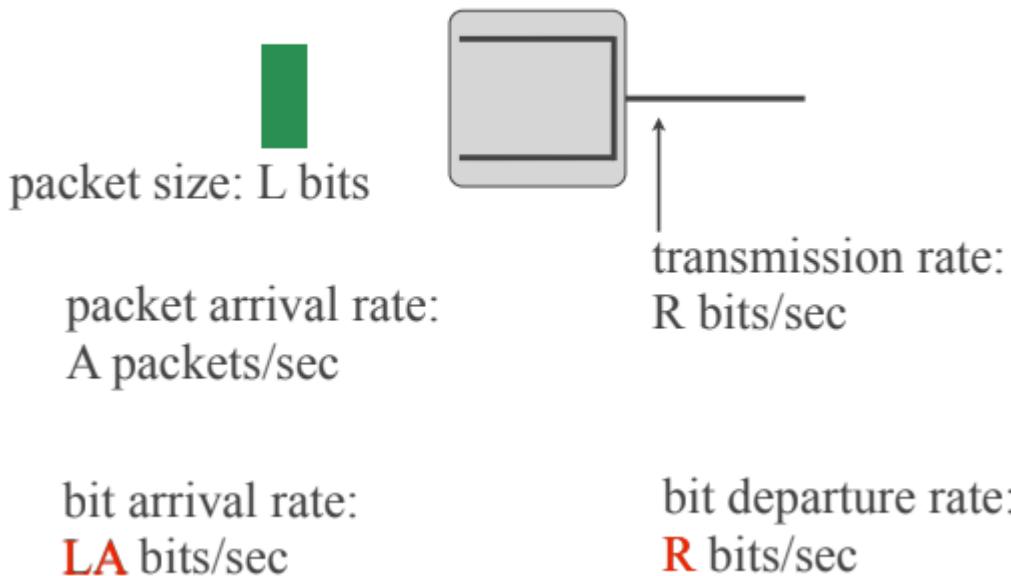
$$\text{propagation delay} : \frac{\text{link length}}{\text{link propagation speed}} = \frac{1 \text{ meter}}{3 \cdot 10^8 \text{ meters per sec}} = 3.34 \text{ ns}$$

le temps qu'on mettrait à aller de la source à la destination

$$\begin{aligned} \text{total packet delay} : & \text{ transmission delay 1} + \text{ propagation delay 1} \\ & + \text{ processing delay} + \text{ transmission delay 2} + \text{ propagation delay 2} \end{aligned}$$

② Queueing delays?

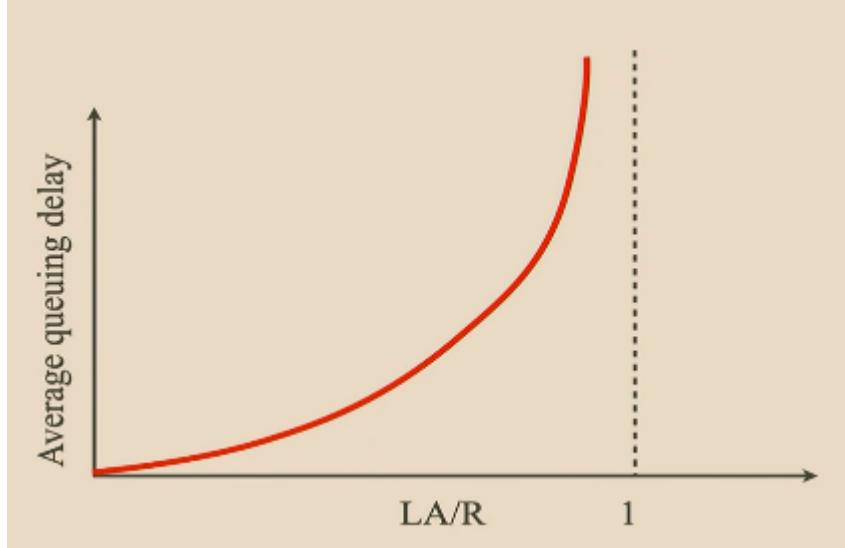
il dépend du traffic (arrival rate mais aussi s'il est en pic ou non), caractérisé par des mesures statistiques (average queuing delay, variance du queuing delay, la probabilité qu'il dépasse une certaine valeur, etc).



On suppose une queue infinie.

Si $LA > R$, on a un delay qui va vers l'infini.

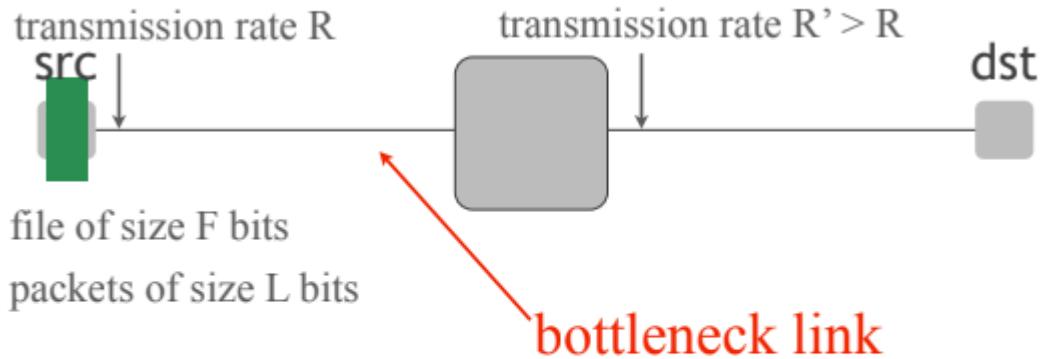
Si $LA \leq R$, on peut quand même avoir un delay ! Si par exemple 4 paquets arrivent en même temps. ça dépend du burst size. et comme on est dans le cas d'un modèle aléatoire, même si on a une moyenne LA, rien ne dit qu'il ne peut pas y avoir un petit pic à un moment, comme chaque paquet est indépendant de l'autre.



donc c'est pas parce que $LA \leq R$ qu'on a un delay nul.

upper bound delay:

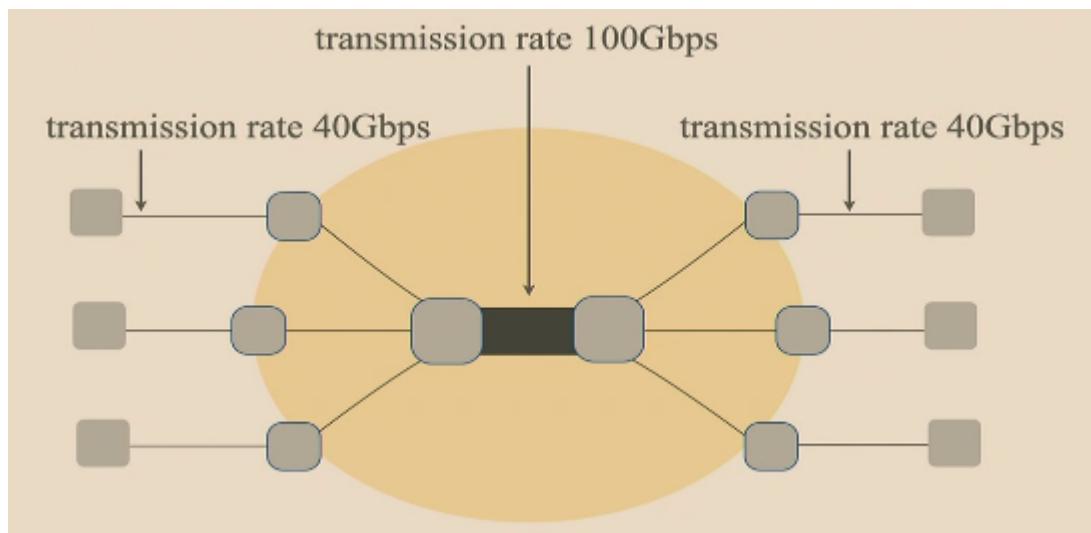
N (taille de la queue) $\cdot \frac{L}{R}$ (transmission delay pour push chaque paquet sur le link)



Transfer time = $F/R + \text{propagation delay 1st link}$
 $+ L/R' + \text{propagation delay 2nd link}$

Average throughput $\approx \min \{ R, R' \} = R$

Si le premier link est plus rapide que le deuxième, il y a un goulot d'étranglement.



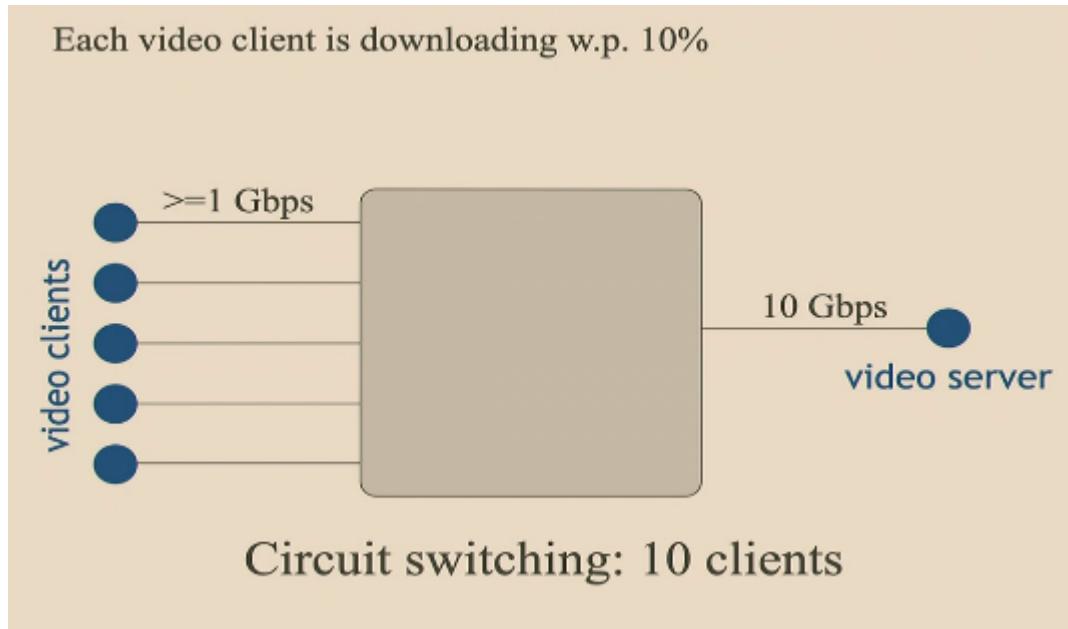
① Packet switching

Le réseau découpe les données en **petits paquets** → chaque paquet est envoyé **séparément**, en passant par **n'importe quel chemin** libre. Gestion efficace des ressources (quand une personne n'utilise pas la ligne, quelqu'un d'autre peut l'utiliser), mais les performances ne sont pas garanties (par exemple, la voix peut couper en téléphonie IP si le réseau est chargé). Demande une gestion de la congestion.

ⓘ Circuit switching

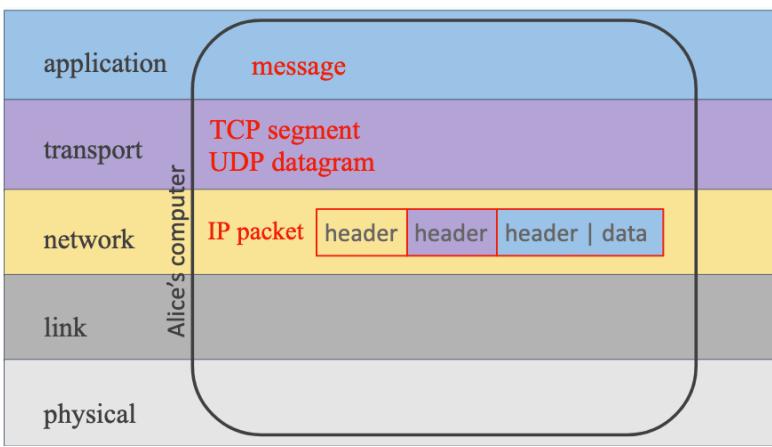
Le réseau réserve **un chemin entier** pour **toute la durée de la communication**. **Personne d'autre ne peut utiliser ce chemin**, même si tu restes silencieux. **Très inefficace** : si tu ne parles pas (silence au téléphone), **la ligne est réservée pour rien**.

On peut réserver une ligne physique, ou réserver une partie d'une ligne physique (virtualisation). On peut même réserver du temps sur une ligne, ou une fréquence. Tout ça dans l'objectif que le client pense être le "seul" sur une ligne sans délai imprévisible.



Lundi 5 mai

Deux protocoles de communication, TCP et UDP.

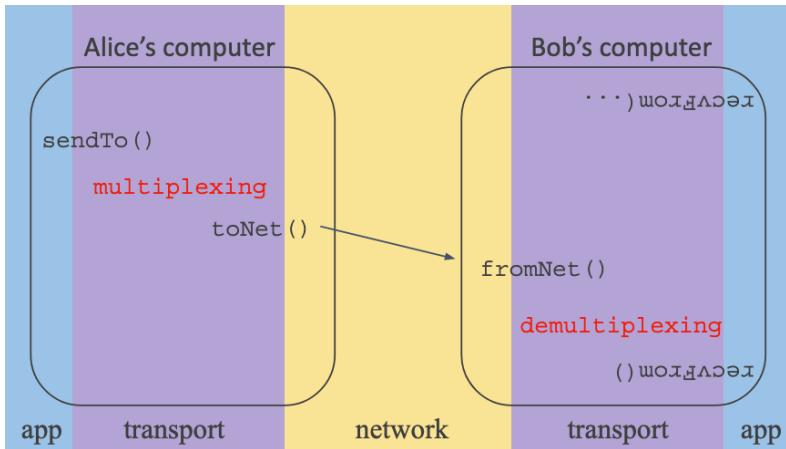


UDP

D'abord, Bob et Alice ouvre une socket. Ensuite, on fait un bind et un recvfrom.

Dans cet exemple, on utilise des **blocking** sockets (la fonction ne return que quand des données sont reçues).

Alice envoie sendTo, puis son OS va appeler toNet. Bob OS va utiliser fromNet, et recvFrom va return.



La même socket UDP peut être utilisée pour communiquer avec plusieurs processus distants.

- il n'y a pas de lien établi entre la socket locale et une seule destination.
- À chaque envoi (`sendto()`), on précise l'adresse IP + port du destinataire.
- À chaque réception (`recvfrom()`), on peut savoir qui a envoyé le message (car le système fournit l'adresse source).

multiplexing : combiner les flux des différentes applications pour les faire passer à travers une seule "porte de sortie" vers le réseau.

demultiplexing : c'est-à-dire regarder le port de destination pour rediriger le bon paquet à la bonne application.

UDP permet de détecter la corruption en ajoutant un checksum dans le header de transport. UDP ne corrige pas cette corruption, ne redemande pas les paquets perdus ou corrompus.

DNS utilise UDP (on veut une latence très faible, si on reçoit pas on renvoie la requête). Fortnite, etc. pareil si la position d'un joueur n'arrive pas, la suivante arrive très vite.

TCP

Établissement de la connexion (Three-way handshake)

1. SYN

1. Alice envoie un segment **SYN** (Synchronization), avec un numéro de séquence initial **ISN_A** (p. ex. 0).
2. Ce SYN « consomme » 1 dans la numérotation TCP.

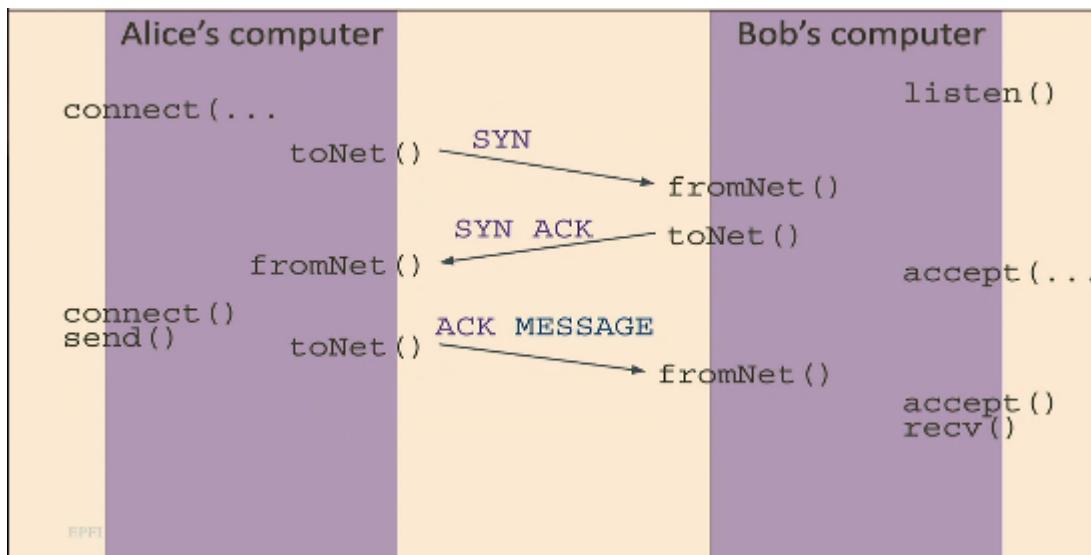
2. SYN + ACK

1. Bob répond par **SYN + ACK**, avec son propre **ISN_B** et $ACK = ISN_A + 1$, $SEQ = ISN_B$.
2. Ce SYN consomme aussi 1 dans la numérotation.

3. ACK (et données éventuelles)

1. Alice renvoie **ACK**.
2. $ACK = ISN_B + 1$
3. $SEQ = ISN_A + 1$

Bob crée **une nouvelle socket**, dédiée aux communications avec Alice. La socket utilisée pour TCP n'attend donc pas les données, uniquement des **demandes** de connexion.



On a un **Maximum Segment Size**, la taille maximale d'un paquet TCP. On crée plusieurs paquets pour envoyer un message.

TCP multiplexing et demultiplexing

- Un serveur crée une **listening socket** (socket, bind, listen)
- Le client crée une **connection socket** (socket, connect)
- Le serveur crée une **connection socket** (accept)

- Le client et le serveur échangent des messages via des **connection sockets** (recv et send)

On va donc avoir plein de sockets, une par client et **au moins une listening socket**.

C'est donc un handshake client-server-client.

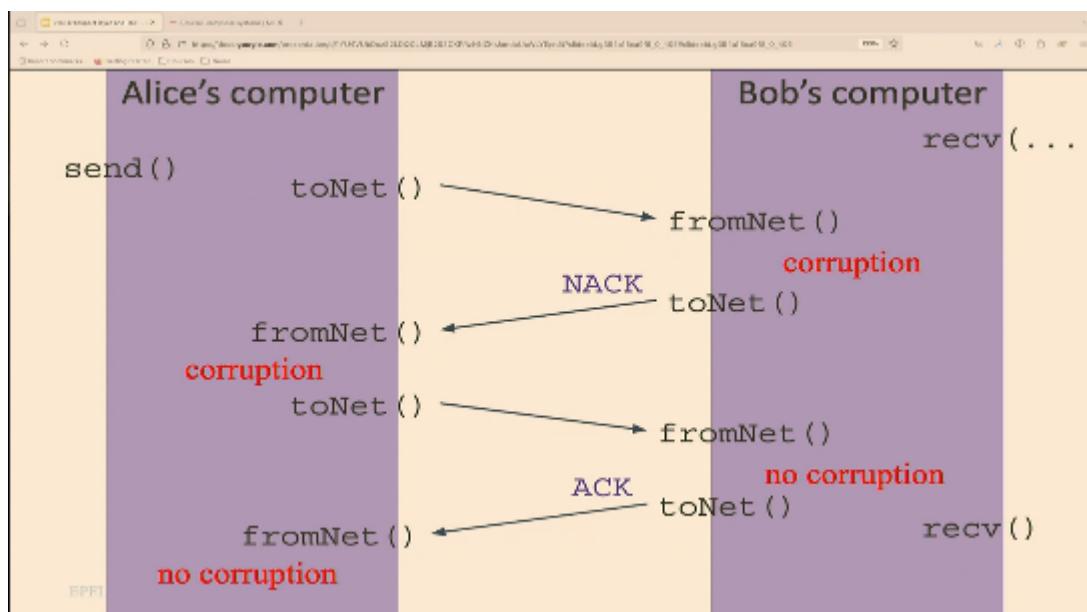
Les deux premiers segments TCP contiennent un SYN flag dans les headers.

Le troisième segment peut contenir des données.

Fiabilité, gestion de la corruption et des pertes

- Chaque segment TCP porte un **checksum**.
- Quand un paquet est reçu via TCP et qu'il est corrompu, Bob va renvoyer un NACK.
- Alice va renvoyer le paquet jusqu'à recevoir un ACK.

L'application n'a jamais connaissance de tout ça, c'est TCP qui gère le reste!



⚡ Que se passe-t-il si l'acknowledgment envoyé par BOB est lui-même corrompu?

Que doit faire Alice ? Alice doit donc renvoyer le paquet. Bob renvoie ACK. Mais là on a envoyé deux fois le même paquet !

Si l'application est Telegram, ça ne doit pas envoyer deux messages !

Solution : ajouter un SEQ (un ID) dans le header. on appelle ça un sequence number.
ACK cumulatifs : l'ACK numéro n signifie que tous les segments jusqu'à $n - 1$ ont été reçus correctement.

Gestion du timeout

Quand l'ACK n'arrive pas à temps, un segment a été perdu ou retardé.
L'envoyeur l'utilise pour **retransmettre** le message.

⌚ Comment calculer le timeout ?

RTT : round-trip time.

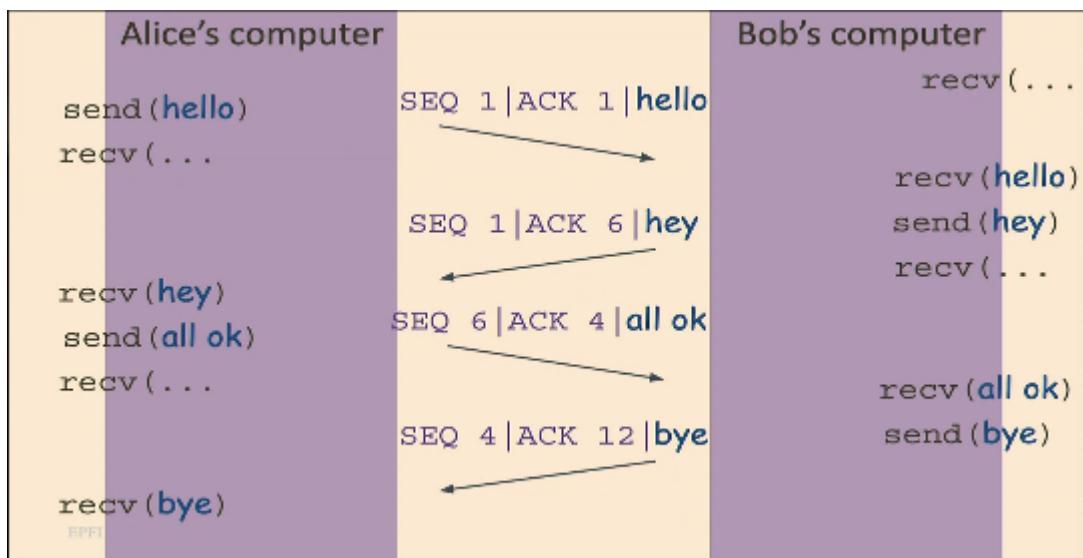
$$\text{Estimated RTT} = 0.875 \text{ estimated RTT} + 0.125 \text{ sample RTT}$$

Dev RTT = function(RTT variance)

$$\text{timeout} = \text{estimated RTT} + 4 \text{ Dev RTT}$$

On utilise la variance pour déterminer la marge d'erreur qu'on a (si un paquet a toujours pris 1h00 pour être reçu alors le timeout doit être presque près d'une 1h).

Sample RTT	Le temps mesuré réellement pour un aller-retour d'un segment donné. C'est une valeur instantanée .
Estimated RTT	Une moyenne mobile qui représente une estimation "lissée" de la RTT, basée sur plusieurs mesures précédentes. C'est une valeur stable et prédictive .



ACK 6 parce que hello a été envoyé (5 bytes).

Contrôle de flux (Flow Control)

- Le récepteur annonce dans chaque ACK un **rwnd** (receiver window), la taille de sa fenêtre de réception libre pour éviter d'être submergé.
- L'émetteur ne peut pas envoyer plus de $\min(\text{cwnd}, \text{rwnd})$ octets non acquittés.

Contrôle de congestion (Congestion Control)

- cwnd** = congestion window (contrôle du réseau, non de l'application).

- On compare `cwnd` à un **seuil ssthresh** pour passer du state **slow start** à **congestion avoidance**.

Objectif : ne pas submerger le réseau.

- le sender l'estime lui-même (**self-clocking**)
 - new ACK : pas de perte, on peut envoyer plus vite !
 - no new ACK (ou ack dupliqué) : perte, envoyer plus lentement

Tahoe vs Reno (timeout)

Événement	Tahoe	Reno
Timeout	$cwnd \rightarrow 1$ MSS $ssthresh \rightarrow cwnd/2$ state → slow start	même que Tahoe
3 duplicate ACKs	$cwnd \rightarrow 1$ MSS $ssthresh \rightarrow cwnd/2$ state → slow start	dès que les 3 acks dupliqués sont reçus: - $ssthresh = cwnd/2$ - $cwnd = ssthresh$ (+ 3 MSS) - on retransmet le paquet manqué et on passe en fast recovery - on reste en fast recovery jusqu'à ce qu'on reçoive un nouvel ACK - pour tout ACK dupliqués reçus entre temps (en fast recovery), on augmente la window de 1 MSS - dès qu'on reçoit un nouvel ACK on reset la window à $ssthresh$

`ssthresh` : slow start threshold

Quand on a un timeout, on fait donc:

- $ssthresh = \frac{cwnd}{2}$ puis $cwnd = 1$
- on reste en slow start
- quand on atteindra `ssthresh`, on passera automatiquement en congestion avoidance

States

- slow start** : augmenter la window de manière agressive
 - aumente de 1 MSS par ACK. à chaque ACK :

$$cwnd = cwnd_{t-1} + \text{MSS}$$

- congestion avoidance** : augmenter la `cwnd` précautionneusement

- augmente de 1 MSS par RTT. à chaque ACK :

$$\text{cwnd} = \frac{\text{MSS}^2}{\text{cwnd}_{t-1}}$$

- parce que :

- on envoie cwnd bytes par RTT, soit $\frac{\text{cwnd}}{\text{MSS}}$ segments par RTT
- on va donc augmenter, par RTT

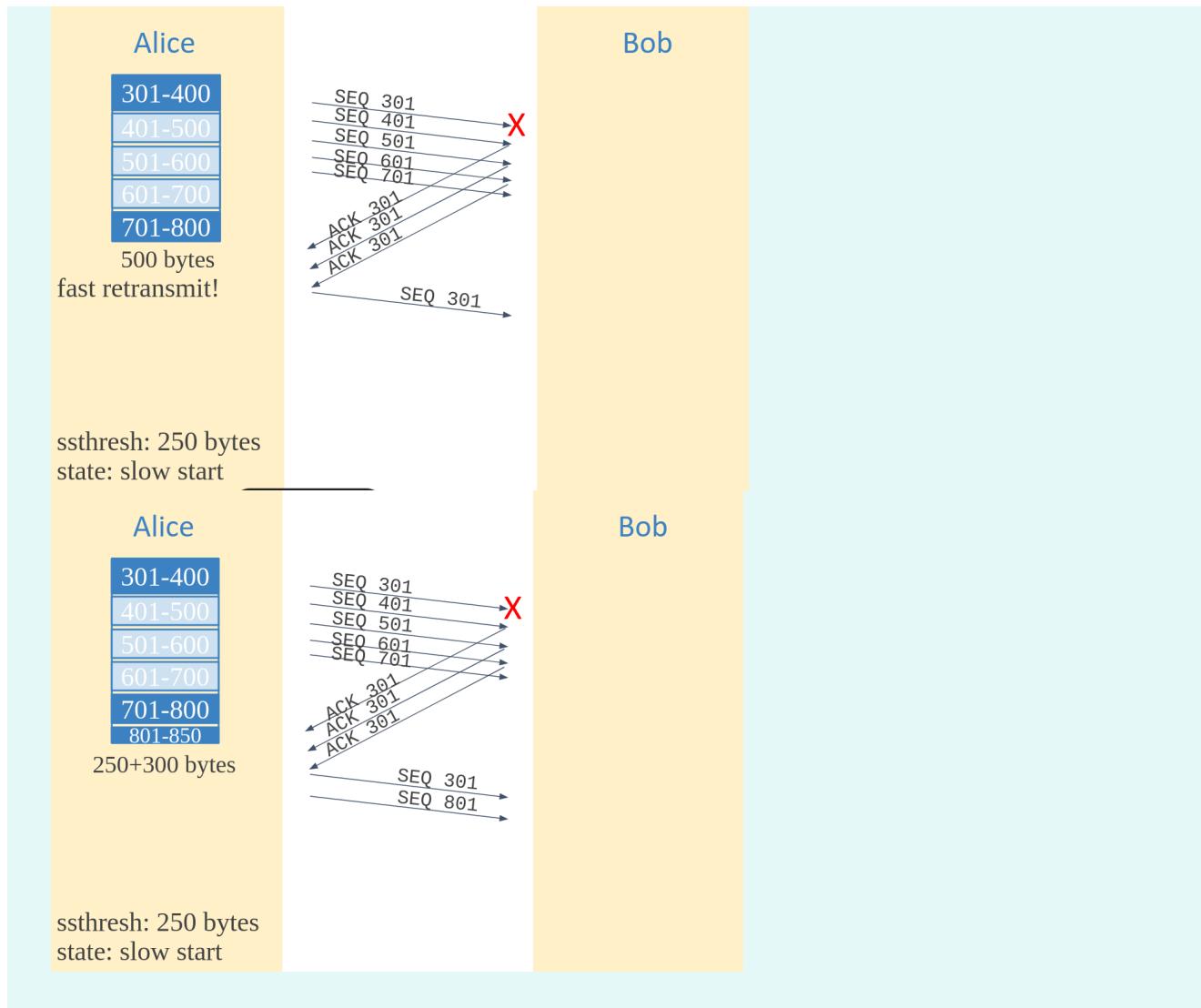
$$\frac{\text{cwnd}}{\text{MSS}} \cdot \frac{\text{MSS}^2}{\text{cwnd}_{t-1}} \approx \text{MSS}$$

- on fait l'approximation $\text{cwnd} \approx \text{cwnd}_{t-1}$.

🔥 Fast retransmit (TCP Reno)

Si un paquet est perdu, normalement on attend le timeout. Or ici, avec fast retransmit, si on reçoit 3 fois un ACK dupliqué, on se doute que le paquet est manquant et on le revoit directement.

Ensuite, on set ssthresh (le threshold avant de passer en congestion avoidance, donc avant de passer en augmentation précautionneuse a window/2).



Mercredi 7 mai

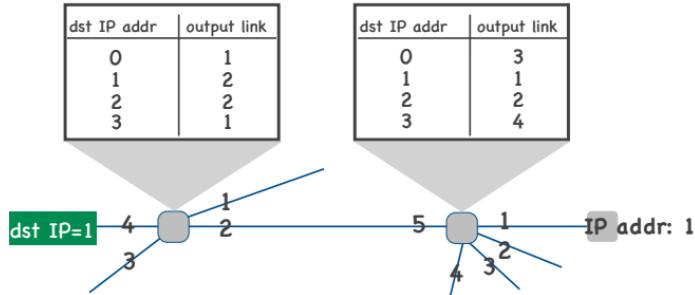
ⓘ Rappel sur les headers

TCP: SYN flag, checksum, SEQ, ACK, receiver window, src port, dst port.

IP: src IP address, dst IP address

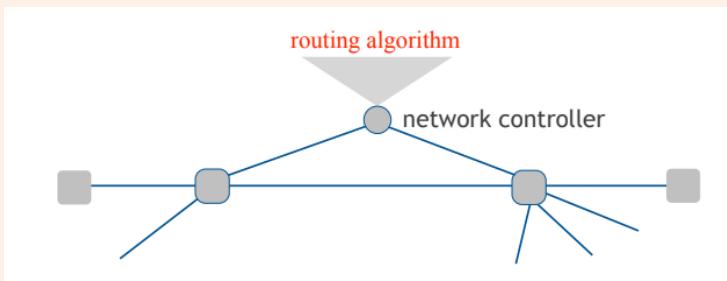
🔮 Routeurs, des network-layer packet switches spécifiques

Ils font d'abord du **forwarding** : regarder le paquet qui arrive et décider où il va (dans quel link). Dans chaque routeur, il y a une forwarding table. Elle est utilisée par le routeur pour faire une décision de routage. Un routeur assigne un numéro à chacun de ses links. Pour chaque dst IP address, le routeur a dans sa table vers quel link l'envoyer.



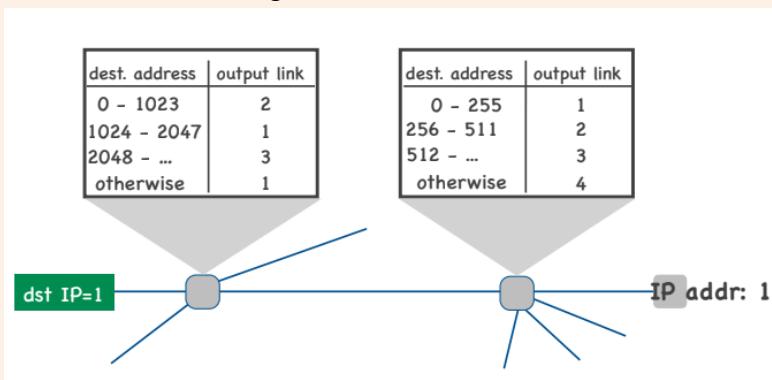
② Comment gérer ces règles de forwarding ?

On pourrait connecter tous les routeurs à un unique network controller, qui a connaissance toute la forme du réseau complet et décide où envoyer quoi.
C'est principalement utilisé en datacenter.



Sinon, on peut imaginer la même chose mais en décentralisé (où l'algorithme tourne en local sur les routeurs). Comment faire ça ?

→ on crée des ranges d'IP adresses.



le routeur utilise des étoiles pour représenter les ranges.

dest. address range		output link
0 - 1	0000 - 0001	000*
2	0010	0010
3	0011	0011
4, 6, 7	0100, 0110, 0111	01**
5	0101	0101
8 - 15	1000 - 1111	1***
10	1010	1010

On fait du **longest prefix matching** : 1000 va être choisi en priorité sur 1***.

⌚ Les IP addresses sont location-dependent

Si quelqu'un pouvait avoir une adresse IP de l'EPFL aux Etats-Unis, il faudrait une règle dans tous les routeurs "si l'IP commence par EPFL.X.X.X --> Suisse" sinon si "EPFL.1.2.3 --> Etats-Unis" c'est très lent : donc address proximity implies location proximity.

(?) Range d'adresses IPs

IP prefix = range d'adresses IPs

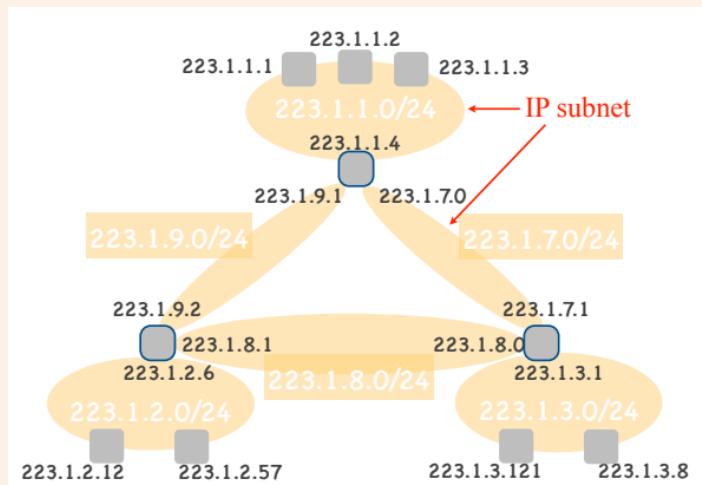
mask -> le nombre de significant bits qui comptent

par exemple 223.1.1.74/24, identique à 223.1.1.*

ou par exemple 223.1.1.9/8, identique à 223.*

ou 223.1.1.0/12, plus difficile: de 223.0.0.0 à 223.15.255.255

On appelle ça un subnet.



Le routeur a l'IP 223.1.1.4, 223.1.9.1, 223.1.7.0, une IP par subnet. Un routeur a donc autant de network interfaces que de subnet auxquels il se connecte.

Un subnet est donc une zone de réseau continue **qui ne contient pas de routeur!**

Tous les end-systems et routeurs ont une IP avec le même préfixe. Un routeur a une IP par subnet qu'il touche.

On verra plus tard qu'on regroupe des subnets ensemble pour former un autonomous system.

⌚ Comment obtenir l'IP du routeur?

```
ip route | grep default
```

② Broadcast IP address?

La plus grande IP address dans un IP subnet. Quand un paquet arrive avec cette IP adresse, il est envoyé à tous les end-systems et routeurs du IP subnet.

Par exemple si un IP subnet a pour IP prefix 223.1.1.0/24, l'IP broadcast est 223.1.1.255.

```
2: wlp0s20f3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default qlen 1000
    link/ether 4c:79:6e:d6:48:ef brd ff:ff:ff:ff:ff:ff
    altname wlx4c796ed648ef
    inet 128.179.153.159/21 brd 128.179.159.255 scope global dynamic noprefixroute wlp0s20f3
        valid_lft 778sec preferred_lft 778sec
        inet6 2001:620:618:598:2:80b3:0:3ec/128 scope global dynamic noprefixroute
            valid_lft 2006sec preferred_lft 993sec
        inet6 fe80::b78a:b721:8132:9d85/64 scope link noprefixroute
            valid_lft forever preferred_lft forever
3: br-0313dfe62601: <NOFORWARD,BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue sta
```

③ Network address?

La plus petite IP de chaque IP subnet est réservée pour nommer le réseau.

Imagine que tu donnes le nom "Groupe A" à un élève spécifique. Quand on dit "envoyez un message au Groupe A", on **ne sait pas si on parle de la personne ou du groupe entier**. C'est ambigu, donc on interdit ce genre de confusion.

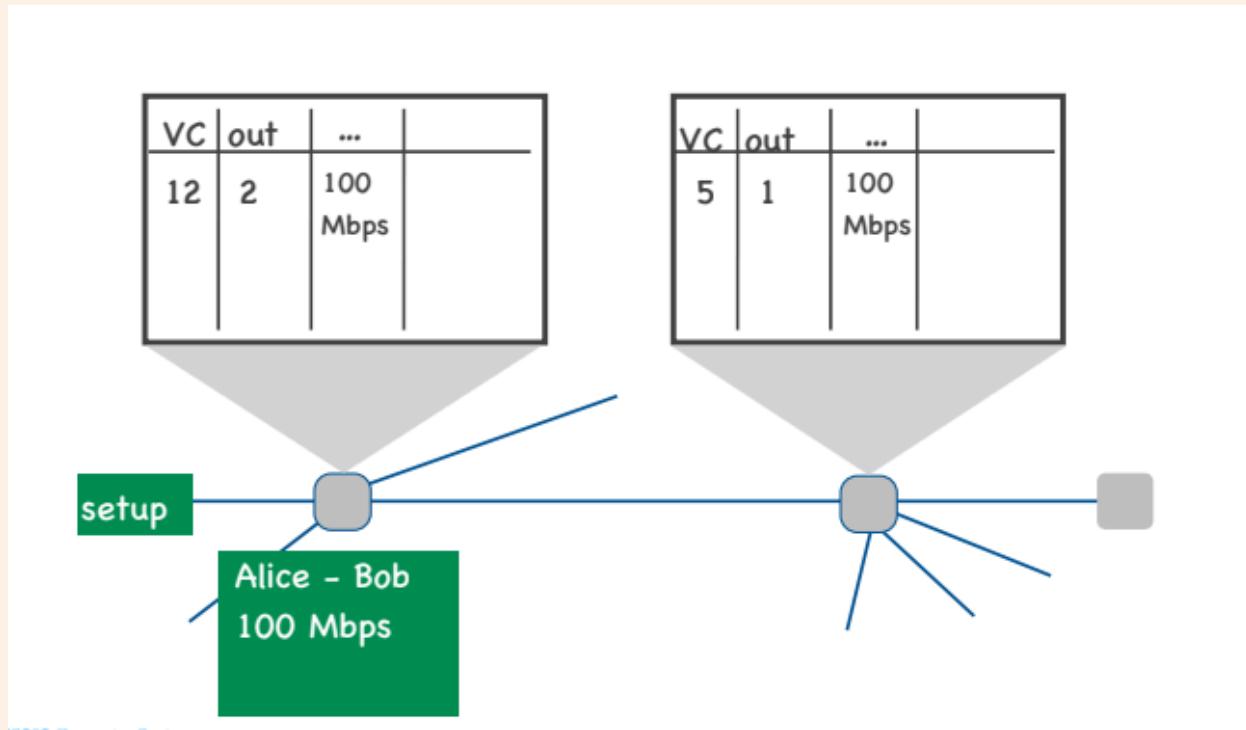
⌚ Chaque organisation obtient un ensemble de IP prefixes, from ISP ou from regular body

Les opérateurs réseau assignent une IP adresse:

- aux routeurs : manuellement
- aux end-systems : manuellement ou through DHCP

Routing (remplir la table de forwarding) \neq fowarding (détermine quel lien choisir)

④ Virtual circuits ?



Si on voulait créer un virtual circuit entre Alice et Bob, on devrait passer par tous les routeurs, et leur faire écrire dans leur forwarding table qu'il existe un virtual circuit entre les deux.

C'est approprié quand on a besoin d'une **garantie de performance** (et pas un best-effort comme le fait TCP de toute façon). Mais on a besoin d'un state, et on ne peut pas vérifier la légitimité d'un client.

Il y a un state pour chaque connection. Le garder au transport layer (= at end-systems), c'est OK mais au niveau du réseau (= at all the routers), c'est PAS OK.

IP packet-switched network : no network-layer connections.

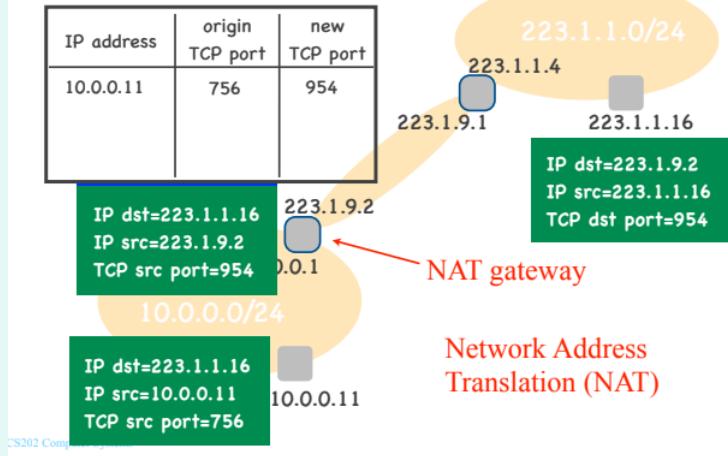
⌚ Global connectivity

Chaque end-system doit être atteignable depuis chaque autre end-system.

Ça implique une IP globale pour chaque end-system. Ça fait beaucoup d'IPs.

On crée des **adresses privées, valides uniquement dans le subnet**. Par exemple `10.0.0.0/24`. On appelle ça un **private address space**.

En fait seul le routeur a une IP publique. Mais comment le routeur sait à quel end-system le packet est destiné ?



Il ajoute un port pour savoir vers quel end-system rediriger le traffic. On ajoute donc du state ! mais ça va parce que c'est dans les routeurs de bordure (ceux qui sont situés juste devant l'entrée d'un subnet).

Cela signifie donc que les end-systems ne sont pas atteignables par l'extérieur. Ils peuvent faire une requête **vers** l'extérieur mais personne ne peut contacter un end-system directement de l'extérieur (à moins que le routeur ne soit configuré pour).

Lundi 12 mai

⚡ forwarding ≠ routing

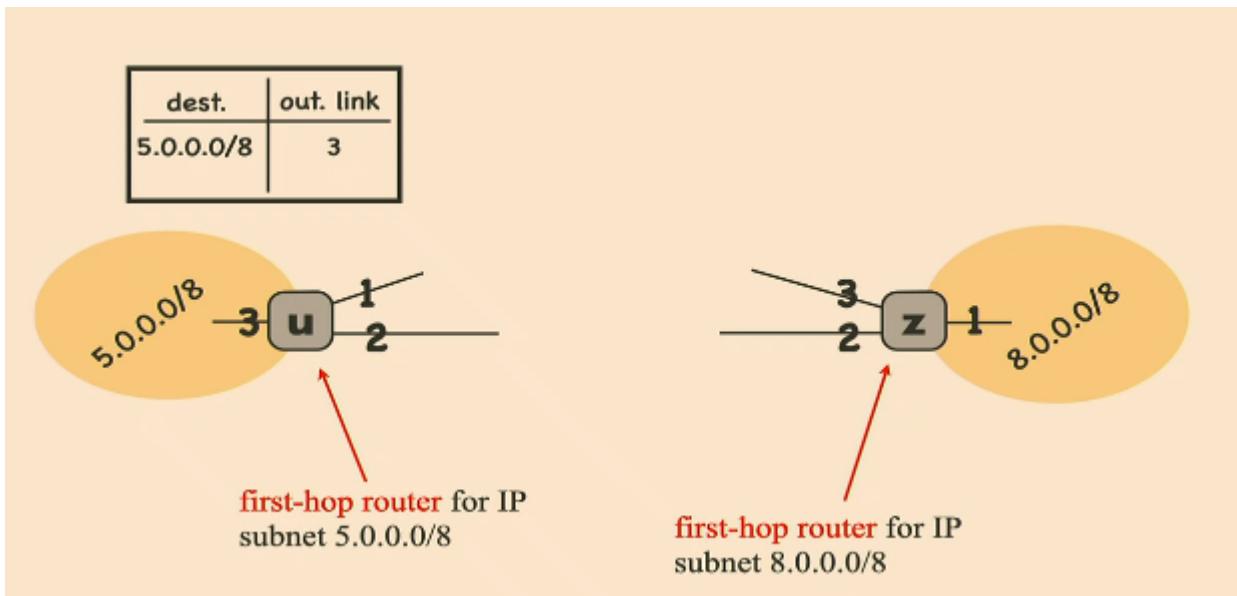
forwarding : opération locale qui se passe quand un packet arrive pour savoir sur quel link il doit aller. Sur Internet, il n'y a pas de circuit virtuel/network-layer connection, c'est que du packet switching, approprié pour un "best-effort service" -- c'est pour cela qu'on a créé TCP, un protocole qui donne l'illusion qu'Internet est parfait (retry policies, etc).

routing : network-wide operation qui remplit les forwarding tables

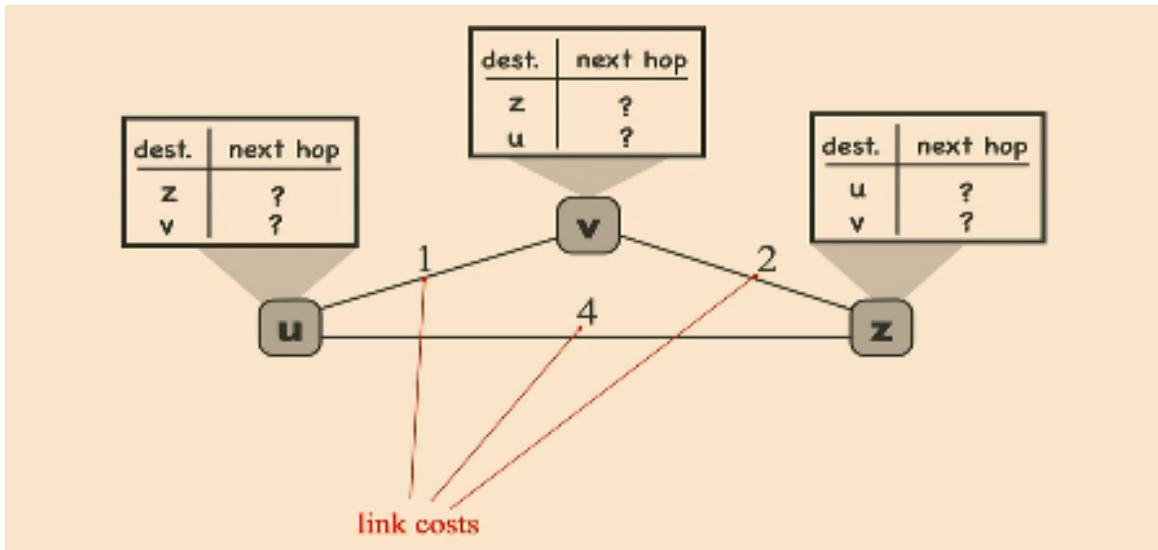
- on peut imaginer un protocole centralisé où un network controller remplit les tables
- on peut imaginer un protocole de routing distribué, qui tourne sur les routeurs

Toutes les adresses IPs doivent être dans la forwarding table (soit hardcodée soit avec des préfixes), chaque routeur doit savoir router toutes les IPs.

first-hop routeur : le premier routeur qui observe les paquets avant qu'ils partent du subnet vers Internet.



Le fait que les IPs locales arrivent vers le link 3 (of course), c'est setup manuellement par l'administrateur réseau.



le coût d'un link est par exemple la performance du link (propagation delay, etc), ou le prix que doit payer l'ISP pour l'utiliser.

⌚ Coûts dynamiques en fonction de l'état du réseau (congested)

On veut aussi faire en sorte que si un link est congested, il ait un coût plus haut. Mais les ISPs, pour éviter les coûts dynamiques qui peuvent créer des oscillations, préfèrent hardcoder ces coûts puis s'ils remarquent qu'un link est surchargé, ils l'upgradent.

Least-cost path routing : trouver le chemin le moins cher d'un routeur à l'autre.

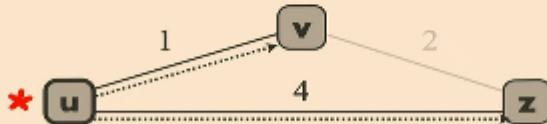
⌚ Link-state routing algorithm

Entrée : le graphe et le coût des links

Sortie : le chemin le moins coûteux du routeur u vers le reste du réseau

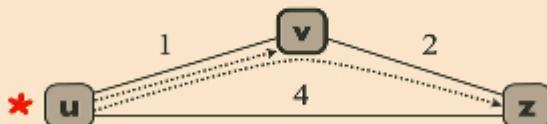
Étape 1 : le routeur considère uniquement ses voisins directs, et note le coût.

dest.	next hop	cost
z	z	4
v	v	1



Étape 2 : le routeur considère les voisins directs de ses voisins.

dest.	next hop	cost
z	v	3
v	v	1



à la prochaine étape, le routeur va considérer les voisins des voisins de ses voisins.

Cet algorithme run périodiquement. Si on ajoute un routeur, il sera identifié.

Dijkstra's algorithm

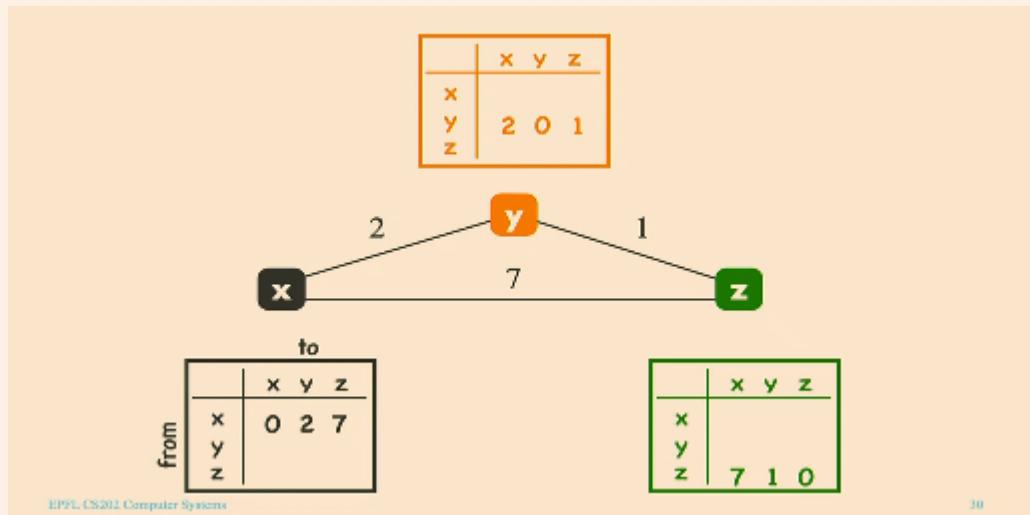
- At each step, consider a new router
 - starting from “closest” neighbor
- Check whether current paths can be improved
 - by using that router as an intermediate point
- End when no improvement is possible

② Distance-vector routing algorithm

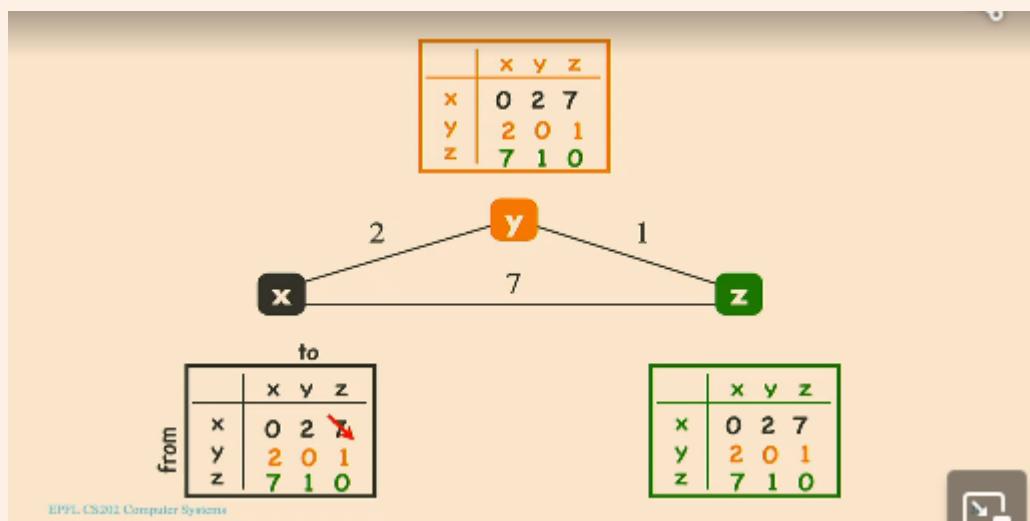
Entrée : les coûts locaux et les messages échangés par les routeurs.

Sortie : le chemin le moins coûteux

À chaque étape, les routeurs se partagent leurs tables.



Au bout d'un moment, ils convergent vers la meilleure solution.



Purely distributed.

🔗 Link state vs Distance vector

Link state : chaque entité obtient d'abord une vue complète du réseau puis compute le least-cost path

Distance vector : chaque entité obtient de façon incrémentale des nouvelles informations à propos du réseau à chaque round

Souvent, link state converge plus tôt, parce qu'une fois qu'on a toute l'information en local, c'est facile de trouver le chemin le plus court. Cependant, pour les grands réseaux, on a pas forcément de ressources. Chaque routeur n'a pas la possibilité de stocker les informations à propos de tous les autres réseaux. On a aussi pas la bandwidth pour toutes ces communications entre chaque routeur.

Chaque routeur partage aussi à tous l'IP prefix qu'il own.

⌚ Security issues?

<https://www.cloudflare.com/learning/security/glossary/bgp-hijacking/>

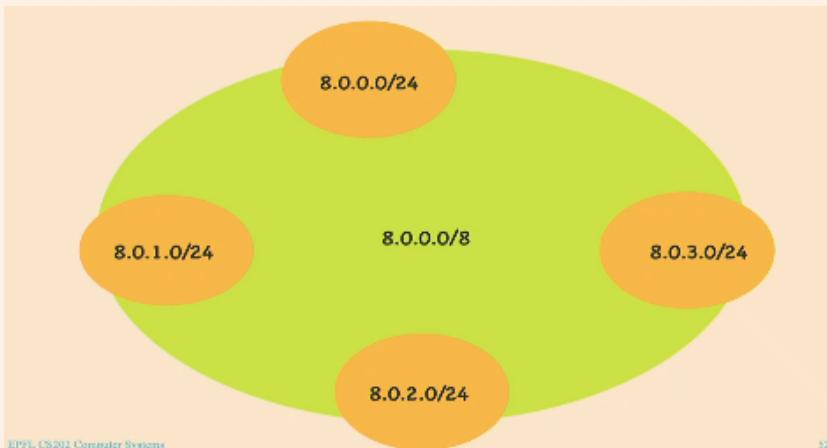
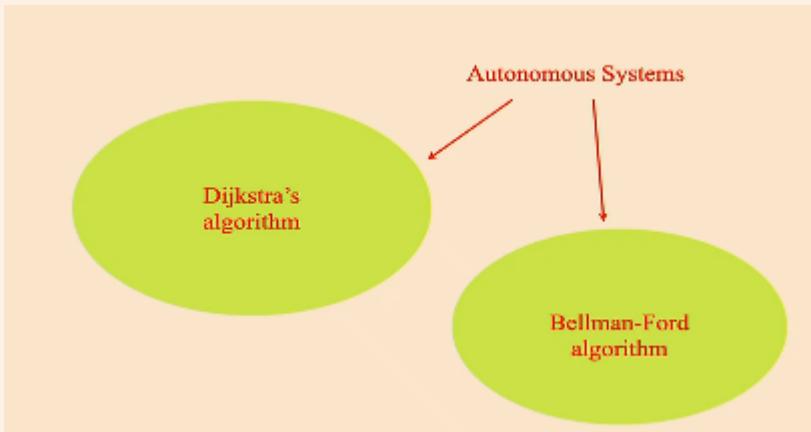
BGP signifie **Border Gateway Protocol**. C'est le **protocole de routage principal** utilisé pour **échanger des informations de routage entre les différents systèmes autonomes (AS)** sur Internet.

Because BGP is built on the assumption that interconnected networks are telling the truth about which IP addresses they own, BGP hijacking is nearly impossible to stop – imagine if no one was watching the freeway signs, and the only way to tell if they had been maliciously changed was by observing that a lot of automobiles were ending up in the wrong neighborhoods.

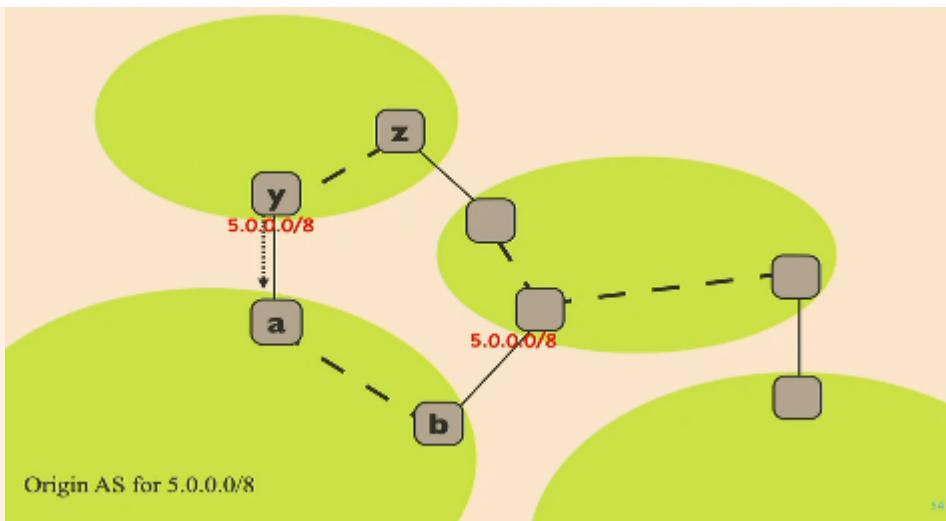
⌚ Autonomous systems ? (ASs)

Un système autonome est typiquement composé de plusieurs subnets.

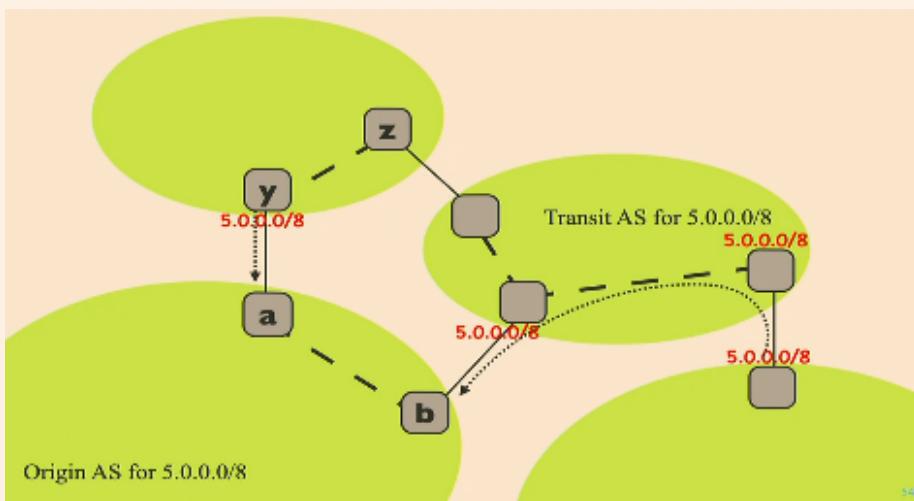
Intra-domain routing protocols.



Les routeurs n'ont pas besoin d'une entrée par sous-sousnet dans chaque AS! il peut simplement stocker le plus gros préfixe du AS!



L'origin AS 5.0.0.0/8 advertises, et dit à ses voisins directs que s'ils voient un paquet arriver pour lui il peut lui envoyer.



Certains AS agissent comme des AS de transit, c-a-d qu'ils advertised à leur voisin pour un autre AS.

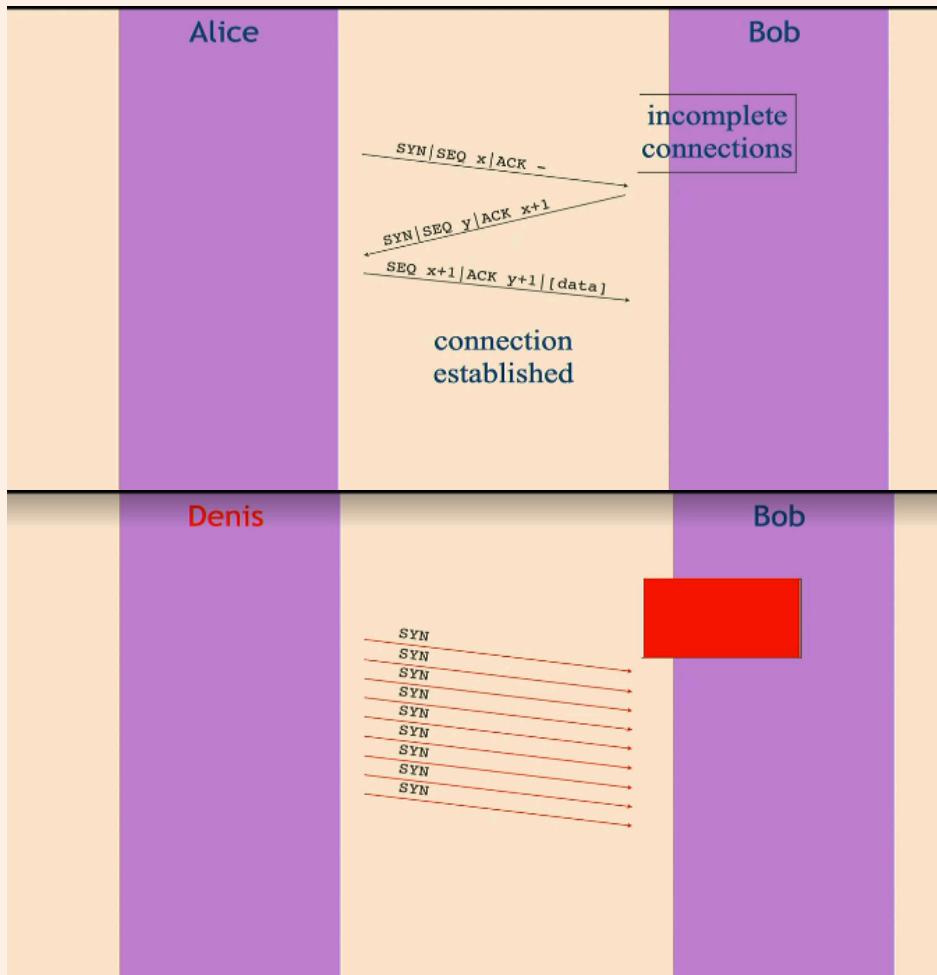
② Border routers

Un **border router** (ou **routeur frontière**) est un routeur situé à la **limite d'un réseau** — typiquement entre un réseau interne (comme celui d'une entreprise, d'un campus ou d'un fournisseur d'accès) et un **réseau externe**, comme Internet ou un autre réseau autonome.

Ils utilisent BGP, Border Gateway Protocol (qui utilise Bellman Ford).

- **L'échange de routes** via des protocoles comme **BGP** avec d'autres AS (inter-AS routing).
- **Le transfert de paquets** vers des destinations situées **hors de l'AS**.
- **Le filtrage** du trafic entrant et sortant pour des raisons de sécurité ou de politique réseau.

② DDOS, SYN flooding



Une attaque était d'envoyer plein de SYN à Bob. Et quand Alice envoie SEQ $x + 1$, la requête n'a pas de succès. Ainsi, c'est vraiment facile de DDOS Bob.

On pourrait mettre une ratelimit. Mais si l'attaque est distribuée.. :(
IP spoofing.

Ainsi, ce qu'on a fait c'est qu'au lieu que Bob garde le state des incomplete connections, il le donne à Alice. Mais où garder ce state ? --> dans le sequence number!

Le nombre y est un hash de l'IP adresse d'Alice à partir d'un secret que Bob connaît uniquement.

Mercredi 14 mai

③ link vs network layer

Link layer (couche 2) :

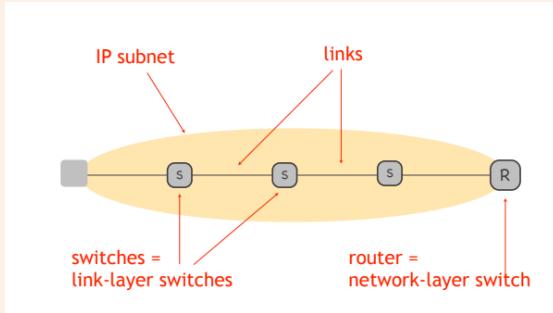
- Prend chaque frame (en fait c'est une unité pour la couche L2 qui contient des en-têtes en + comparé à un packet) et l'envoie d'un lien physique à un autre, à

I l'intérieur d'un même sous-réseau IP.

- Assure la **détection d'erreurs** (checksum), la **fiabilité** (ACKs, retransmissions...).

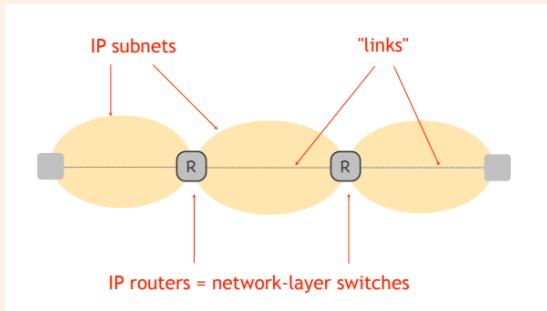
② Pourquoi le faire dans cette couche si le transport le fait aussi ?

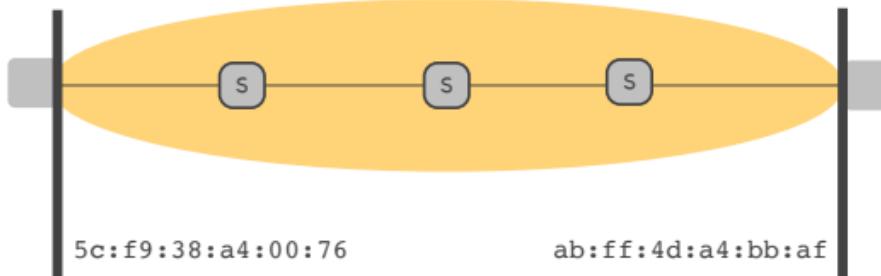
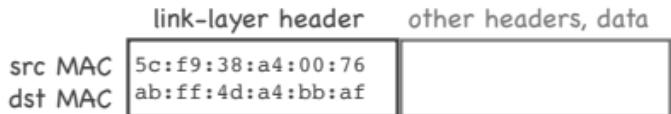
simplement parce que si un petit link est très peu reliable (typiquement wireless), à chaque fois on va perdre le packet et Alice va devoir renvoyer peut-être 3-4x ses packets.



Network (couche 3) :

- Prend chaque paquet et l'achemine **d'un sous-réseau IP vers un autre**, sur l'ensemble de l'Internet.
- Utilise des adresses IP et le protocole de routage BGP pour construire des tables de forwarding globales.





destination MAC address : l'adresse locale de l'appareil dans le subnet

💡 MAC address

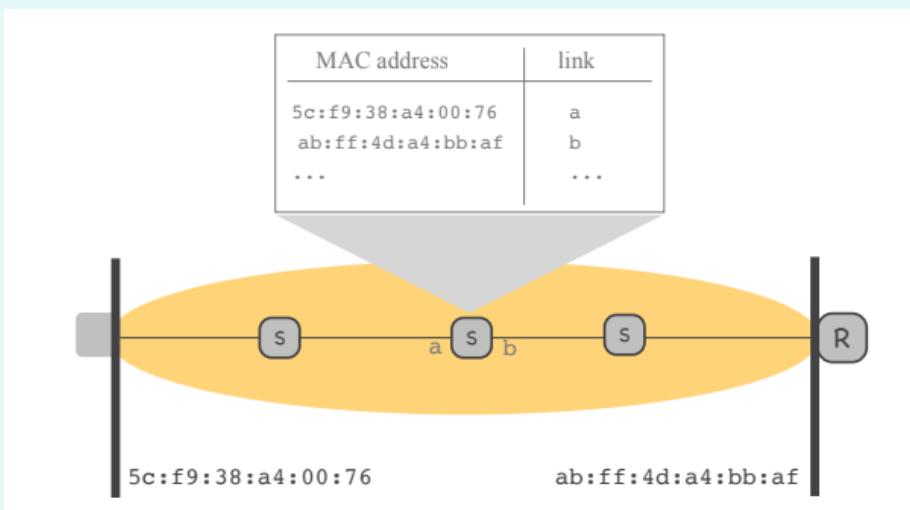
48-bit number

Flat : elles n'ont pas de hiérarchie comme des adresses IP, ne sont pas location-dependent.

Chaque interface réseau possède une MAC unique.

Forwarding et remplissage des tables

💡 L2 Forwarding



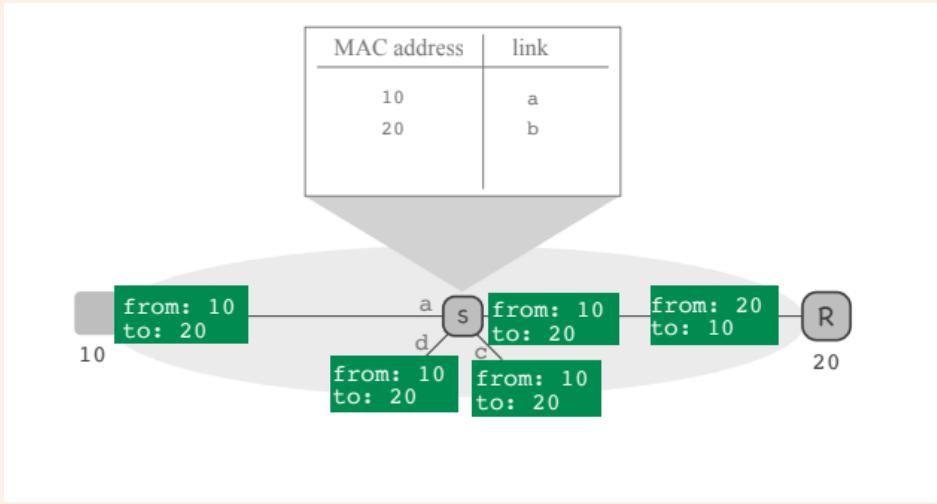
La switch détermine l'output link de chaque packet. Il se base sur les MAC addresses. Il y a donc des grandes tables.

⌚ Qui remplit les forwarding tables du L2 ?

Pour le L3, c'était le routing protocol.

Pour le L2 :

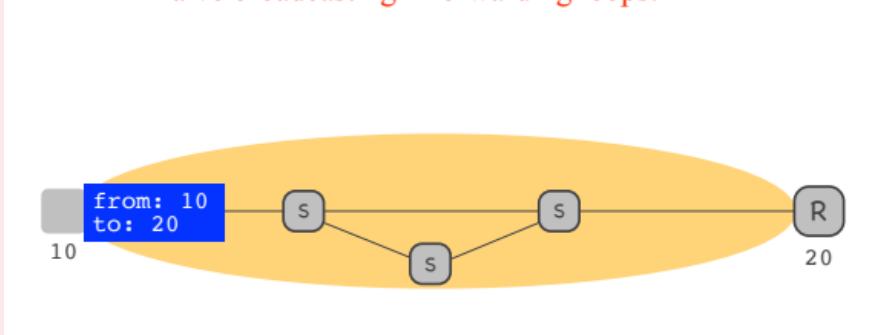
- chaque switch apprend dynamiquement les MAC : quand il reçoit une frame, il enregistre l'adresse source et le link d'arrivée.
- Pour acheminer, il consulte sa table MAC → link de sortie.
- Si destination inconnue → inondation (broadcast) sur tous les links.



⚡ loops

Les boucles Ethernet (cycles de liens) entraînent des inondations infinies et des problèmes de congestion.

naïve broadcasting = forwarding loops!



Comment gérer ça ?

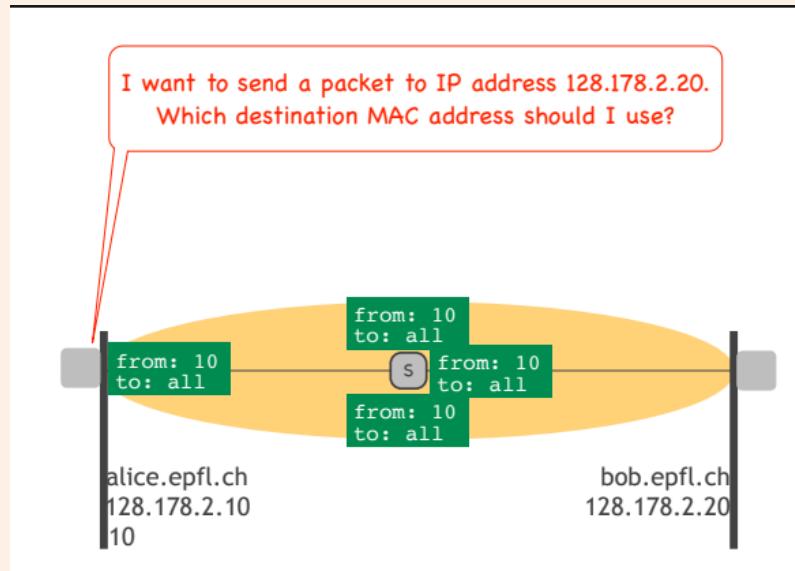
- avec du state ? on pourrait noter qu'on a déjà vu le packet (problème : l'espace n'est pas infini et c'est un vecteur d'attaque)
- on crée un minimum spanning tree pour éviter les cycles

⌚ Address resolution

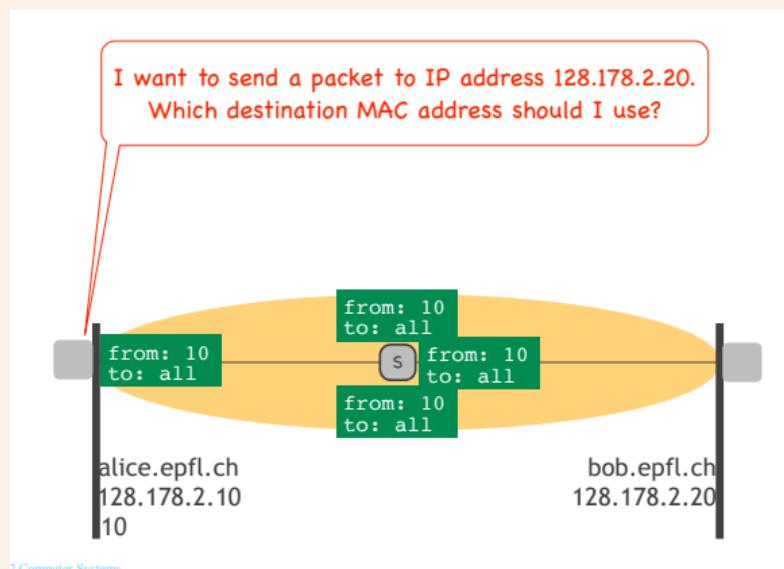
Il y a deux cas.

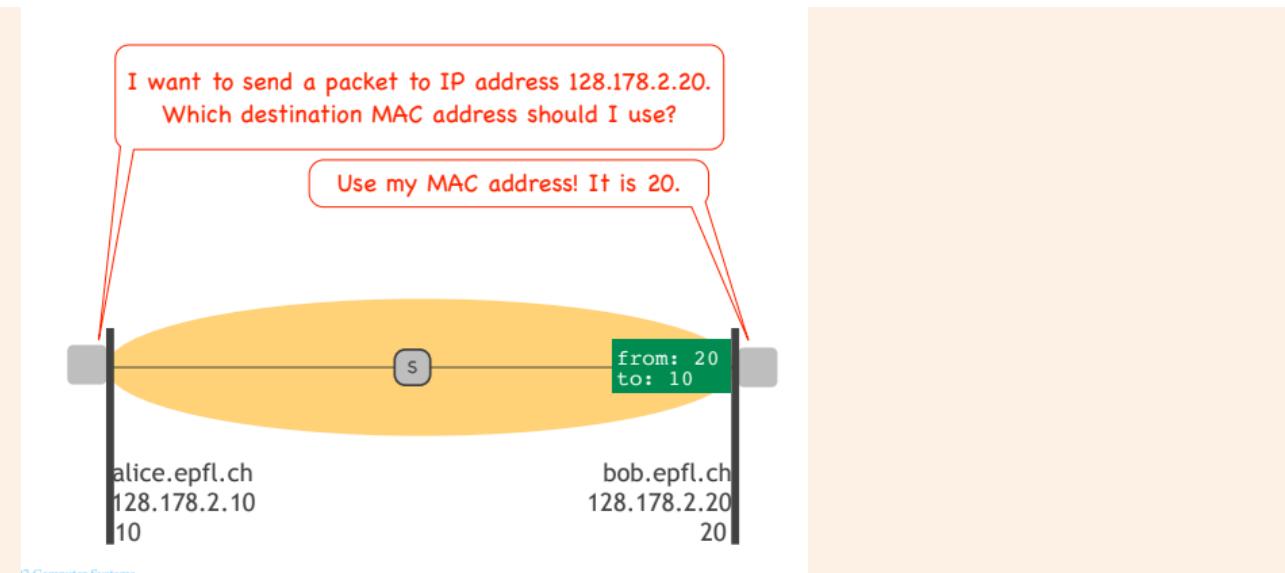
CAS 1 : IP cible dans le même subnet

- Le client sait, d'après son masque, que l'adresse est "locale".
- Il émet une frame Ethernet **ARP Request** ("Qui a l'IP X.X.X.X ?") en broadcast.
- L'hôte correspondant répond par un **ARP Reply** contenant sa MAC.
- Le client peut alors envoyer la frame Ethernet directement à la MAC cible.



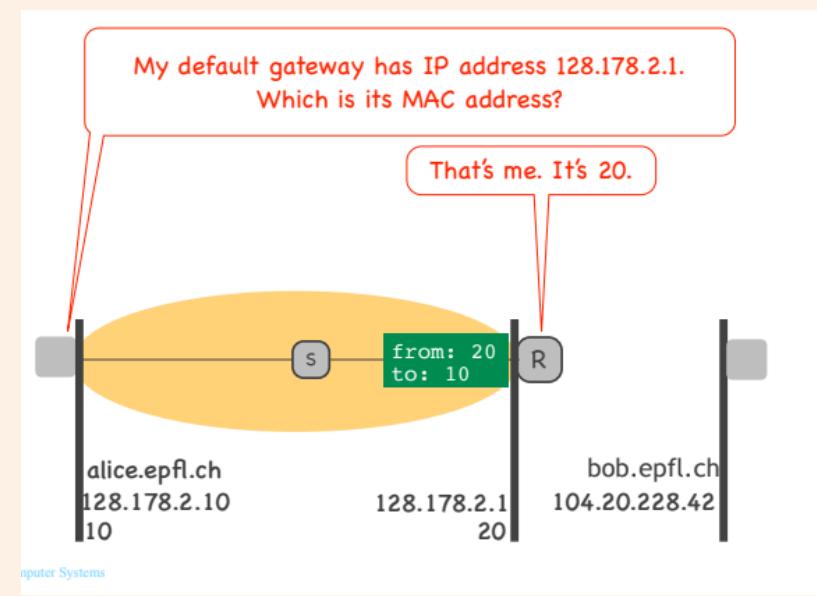
Solution : on broascast!





CAS 2 : IP cible en dehors du sous-réseau

- Le client constate que l'adresse n'est **pas** dans son subnet.
- Il encapsule son paquet IP dans une frame Ethernet destinée à la **MAC de la passerelle** (default gateway).
- Pour cela, il émet une **ARP Request** pour l'IP de la gateway, reçoit la MAC, et envoie la trame à cette MAC.
- C'est ensuite le routeur (la gateway) qui, lui, a une table de routage pour transférer le paquet vers le réseau de destination.



⌚ Adress Resolution Protocol (ARP)

L'objectif est de mapper une adresse IP à une adresse MAC.

Comment ? broadcasting + caching

ARP vs. DNS

- ARP: relies on broadcasting
 - *no logically centralized map*
 - *each entity knows its own MAC address and knows which requests to respond to*
- DNS: relies on DNS infrastructure
 - *logically centralized map*
 - *stored in DNS servers*

⚡ Get rid of IP addresses and IP forwarding?

Problème de scalabilité : location-dependent c'est utile. le fait qu'on puisse avoir des petites forwarding tables. + le broadcasting ne scale pas.

⚡ Get rid of MAC addresses and L2 forwarding?

Le lien physique (L2) ne comprend que des **MAC addresses**. Les L2 sont conçus pour relier efficacement des hôtes proches (même sous-réseau), tandis que la couche 3 organise la communication globale via des adresses hiérarchiques.

Il faut vraiment voir l'IP comme un moyen global de connecter les end-systems entre eux mais à l'intérieur d'un subnet on peut se connecter comme on veut, avec le protocole qu'on veut (comme Ethernet avec les adresses MAC).

Exemple concret :

nginx

 Copy  Edit

```
Alice (192.168.1.2 /24, MAC: AA)
Routeur (192.168.1.1 /24, MAC: RR)
But : envoyer un paquet à 8.8.8.8
```

Étapes :

1. Alice veut envoyer un paquet IP à 8.8.8.8.
2. Elle voit que 8.8.8.8 n'est pas dans son subnet (192.168.1.0/24).
3. Donc elle envoie le paquet à son routeur par défaut, à l'adresse IP 192.168.1.1.
4. Pour cela, elle doit connaître la MAC de 192.168.1.1 → elle fait un ARP request pour 192.168.1.1.
5. Elle reçoit la MAC RR.
6. Elle envoie maintenant :
 - Un paquet IP pour 8.8.8.8
 - Encapsulé dans une trame Ethernet à destination de RR.



Le routeur reçoit, puis :

- Il ouvre le paquet IP, voit la vraie destination : 8.8.8.8.
- Il consulte sa table de routage.
- Et transmet le paquet à la prochaine étape :
 - Soit il a une connexion directe vers 8.8.8.8 → il fait un ARP pour obtenir la MAC.
 - Soit il l'envoie à un autre routeur (son propre next hop), en obtenant là aussi la MAC correspondante via ARP.

la MAC correspondante c'est celle correspondante au prochain routeur

Si un end-system dans un subnet veut faire une requête à un end-system dans un autre subnet, il ne va pas faire de requête ARP pour connaître sa MAC address (mais uniquement celle de sa gateway)

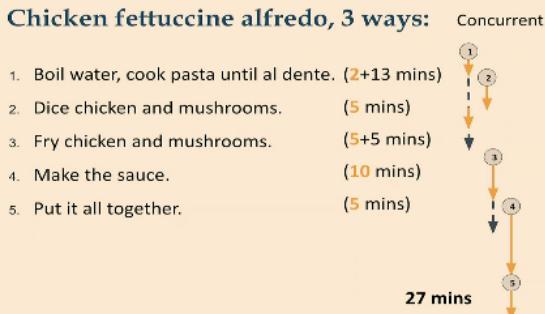
 les switches ne répondent PAS au requête ARP ! si switch 1 connaît la MAC de B demandé par A, il va juste broadcast la requête ARP

Lundi 19 Mai

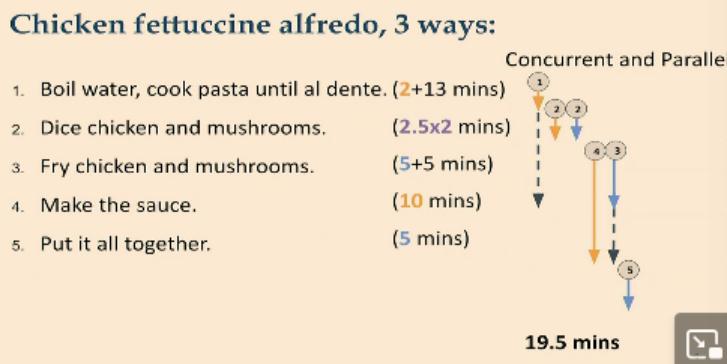
Sequentiel : 35 min, les tâches les unes après les autres.

 **Concurrency vs Parallelism**

Concurrent : lorsqu'une tâche attend une opération d'I/O (par exemple, attendre que l'eau des pâtes se mette à bouillir), le processeur peut s'occuper d'une autre tâche en attendant. Cela donne l'**illusion** que plusieurs tâches s'exécutent en même temps.



Parallèle : Si l'on dispose de plusieurs unités de calcul (plusieurs "chefs"), on peut **réellement** exécuter plusieurs tâches simultanément. Les tâches sont alors parallélisées.



② Pourquoi parallélisme et concurrency ?

D'un point de vue utilisateur :

- **Augmenter le débit (throughput)** : traiter plus de tâches en un temps donné.
- **Réduire la latence (latency)** : obtenir une réponse plus rapidement.

D'un point de vue système :

- **Utiliser efficacement tous les coeurs** du processeur.
- **Masquer la latence** des opérations d'I/O (disque, réseau, etc.).

:≡ Bash

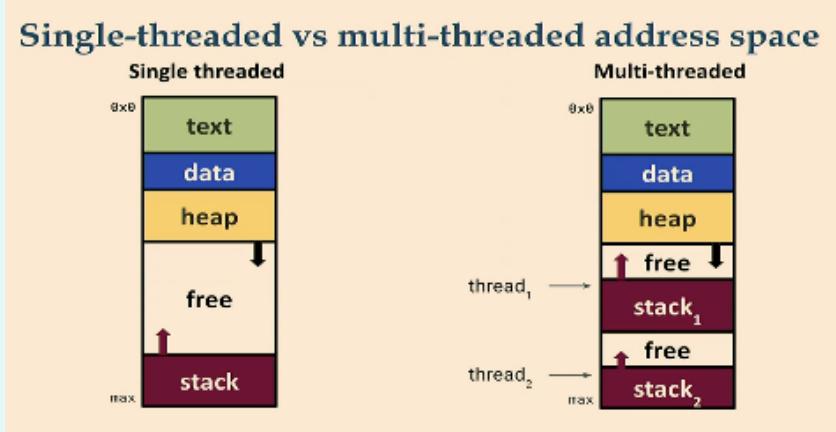
& à la fin de la commande, permet de lancer un process avec un fork et de lancer une autre commande pendant ce temps.

Problème : on copie aussi le programme ! Les process ne partagent pas la mémoire!

Solution : les threads !

Rappel sur les threads

Ils partagent l'address space (heap, data, text), les file descriptors.
 mais... chaque thread a son propre stack, est scheduled indépendamment par son OS.
 Ils ne partagent pas les system calls.



Gérer les threads:

`pthread_create`, `pthread_join` : wrappers autour de syscalls. ça évite de faire tous les if == 0, gérer les return values, etc.

Managing threads - the API

```

pthread_create(&t,&attr,(void*) (*start_routine)(void *),args)
• Create a new thread
• The pthread library calls *start_routine(args)

pthread_join(&t,void **retval)
• Wait for the completion of a thread
  (return from *start_routine)

```

`objdump -S race` permet de voir les instructions assembly d'un programme

```
#include <stdio.h>
#include <pthread.h>
#define NUM_ITER 100000
int counter = 0;

void *incr(void *arg) {
    printf("%s starts\n", (char *)arg);
    for (int i=0; i < NUM_ITER; i++)
        counter = counter + 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t t1, t2;
    // Create two threads T1 and T2
    pthread_create(&t1, NULL, incr, "T1");
    pthread_create(&t2, NULL, incr, "T2");
    pthread_join(t1, NULL); // Wait for T1 to finish
    pthread_join(t2, NULL); // Wait for T2 to finish
    printf("Counter: %d (expected: %d)\n", counter,
NUM_ITER*2);
    return 0;
}
```

race condition (et plus précisément data race), quand deux threads veulent accéder à une variable mutable partagée
(comme les threads s'exécutent en parallèle)

⌚ Atomicity et locks

L'incrémantation d'une variable (`count++`) n'est pas une opération unique pour le processeur. Elle se décompose en trois instructions : chargement, incrémantation, et écriture (load, increment, store).

```
mov    0x2e95(%rip),%eax      # 4034 <counter>
add    $0x1,%eax
mov    %eax,0x2e8c(%rip)      # 4034 <counter>
```

On veut que ces trois instructions s'exécutent de façon atomique, que rien ne s'exécute entre eux.

Critical section : quand deux threads essayent d'accéder à une zone mémoire partagée.

Mutual exclusion : un seul thread peut exécuter une section critique à la fois.

```
lock_t mutex;
...
lock(&mutex);
counter = counter + 1
unlock(&mutex)
```

A lock is a declared variable

Le lock protège cette zone critique.

Un seul thread peut obtenir un lock à la fois : lock holder

Les autres threads doivent attendre que le lock soit released : lock waiter

⌚ Interrupt lock

```
void lock(lock_t &l) {
    disable_interrupts();
}

void unlock(lock_t &l) {
    enable_interrupts();
}
```

le pb c'est qu'on avait un context switch au milieu de l'instruction --> maintenant le CPU ne va plus interrompre le thread.

mais... dangereux de laisser un thread désactiver tous les interrupts.

Et si par exemple le programme attendait un network interrupt depuis 15min, il peut le rater.

⌚ Software lock ?

Utiliser une variable partagée pour synchroniser l'accès à une section critique :

```
bool lock1 = false;

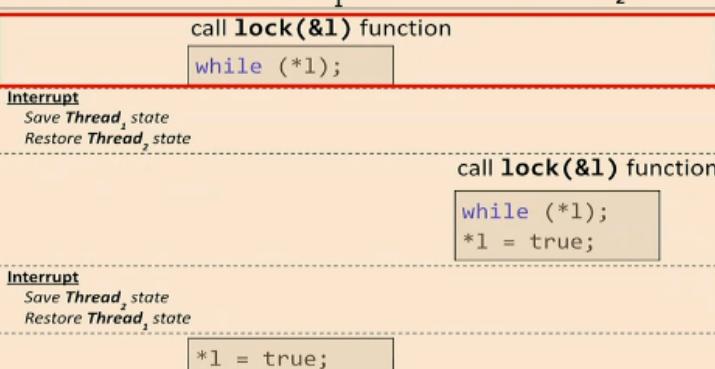
void lock(bool *l) {
    while (*l); /* spin until we grab the lock */
    *l = true;
}
void unlock(bool *l) {
    *l = false
}
```

Problème : c'est pas atomique!

Issue with this lock

```
void lock(bool *l) {
    while (*l); /* spin until we grab the lock */
    *l = true;
}
```

OS

Thread₁Thread₂

Thread₁ first calls the `lock` function and there is already some other thread in the critical section

At this point Thread₁ keeps spinning and it is about to acquire the lock as the other thread has left the critical section

60

Il check le lock, le second prend le lock et quand le premier revient il prend aussi le lock.

⌚ test-and-set

En une instruction, on set le lock et on le get en même temps (son ancienne valeur).

```
bool lock1 = false;

void lock(bool *l) {
    while (*l); /* spin until we grab the lock */
    *l = true;
}

void unlock(bool *l) {
    *l = false
}
```

```
bool lock1 = false;

void lock(bool *l) {
    while (test_and_set(l, true));
    // spin and wait (do nothing)
}

void unlock(bool *l) {
    *l = false
}
```

hardware lock, en une seule instruction --> atomicité !

pb : pendant le while le CPU n'exécute aucune instruction ! c'est comme regarder les pâtes cuire

⌚ avoid wasting cycles? --> mutexes

lock waiters: busy waiting.

involve the OS!

avec yield, le waiter dit au CPU qu'il peut run autre chose a la place

mutex : waiters go to sleep et le lock holder les réveille quand le lock est released.

pthread_mutex()

```

1 #include<stdio.h>
2 #include<pthread.h>
3 pthread_mutex_t m =
        PTHREAD_MUTEX_INITIALIZER;
4
5
6
7 void critical_section(int *counter) {
8     pthread_mutex_lock(&m);
9     *counter += 1;
10    pthread_mutex_unlock(&m);
11    return;
12 }
13
14 }
```

Standard POSIX-compliant API

- Semantic of a mutex:
 - Thread **blocks** when lock is held
- Integration with other synchronisation primitives (e.g., condition variables)
- Implementation within the OS

☰ Dining philosophers problem -- deadlocks!

Philosophers think, eat, think, eat

Need both forks to eat (a gauche et a droite)

ils ne peuvent pas la grab en même temps



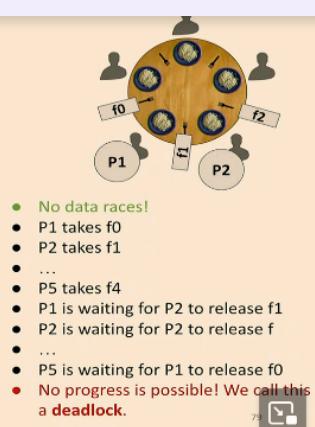
Candidate solution 2

```

pthread_mutex_t fork_locks[5] = \
        {PTHREAD_MUTEX_INITIALIZER};

void take_forks(i) {
    // take fork to the left
    pthread_mutex_lock(&fork_locks[i]);
    // take fork to the right
    pthread_mutex_lock(&fork_locks[(i + 1) % 5]);
}

void put_forks(i) {
    ...
}
```



deadlock! Ils prennent tous la fourchette à sa gauche et ils attendent tous que quelqu'un release l'autre fourchette (mais personne ne va le faire !)

Mercredi 21 mai

② Multiplexing

On a 1 ressource physique et on l'expose à travers différentes entités virtuelles.

On fait ça avec les CPUs et la mémoire.

(L'OS est responsable de l'isolation -- on ne veut pas que les pages se chevauchent).

② Aggregation

On fait apparaître plusieurs ressources comme une seule ressource virtuelle.

On fait ça avec les disques. En DC, on a peut-être 64 disques connectés à un serveur.

On peut créer 64 file systems, mais certains seront remplis, certains vides, etc. --> difficile à gérer pour le serveur. solution: DIMM, RAID

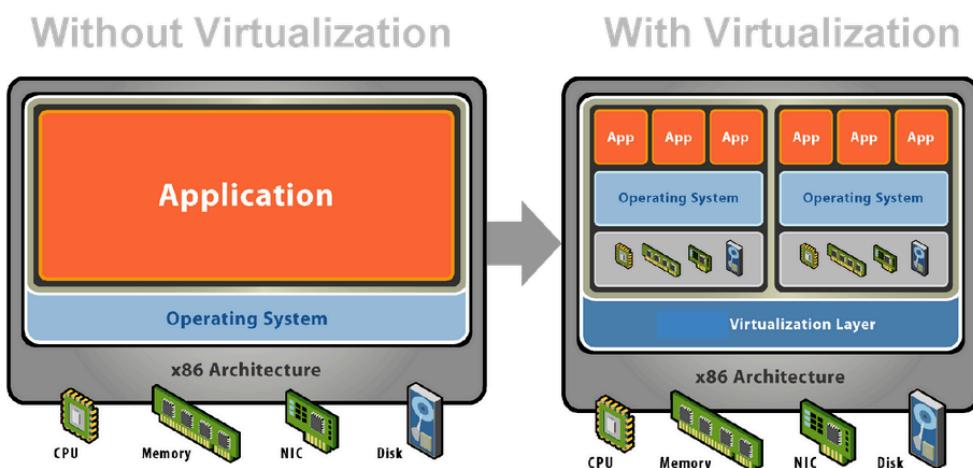
③ Emulation

On fait apparaître une ressource comme une autre ressource physique.

Par exemple faire apparaître une clef à travers SSH.

ou par exemple faire apparaître un CPU comme ARM alors qu'on a un x86...

Virtual Machines



Apply virtualization principle to **an entire computer system**

- P = computer system
- V = computer system (== **virtual machine**)

④ pourquoi utiliser des VMs ?

- il y a longtemps, hardware expensive --> on peut utiliser le même pour run différents OS
- multi-tenancy (propriétaires différents)

Terminologie

Hypervisor, virtual machine monitor VMM : a un accès complet au système.

Guest operating system : run dans la VM

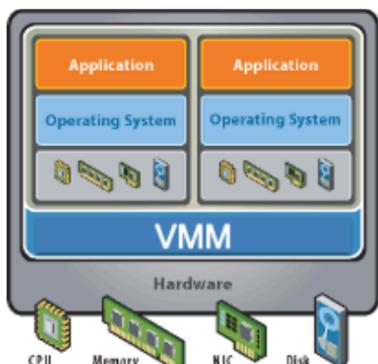
Host operating system : run sur le hardware directement - combiné avec le VMM (par exemple KVM)

Virtual memory : comme d'habitude

Guest physical memory : ce que l'OS guest pense avoir comme mémoire virtuelle

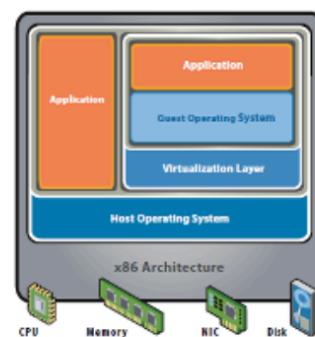
Host physical memory : la vraie mémoire physique complète

Type 1 vs. Type 2



Type 1

- VMM is also the host OS
- E.g., Xen, vSphere



Type 2

- VMM optional extension to the host OS
- E.g. KVM and Linux

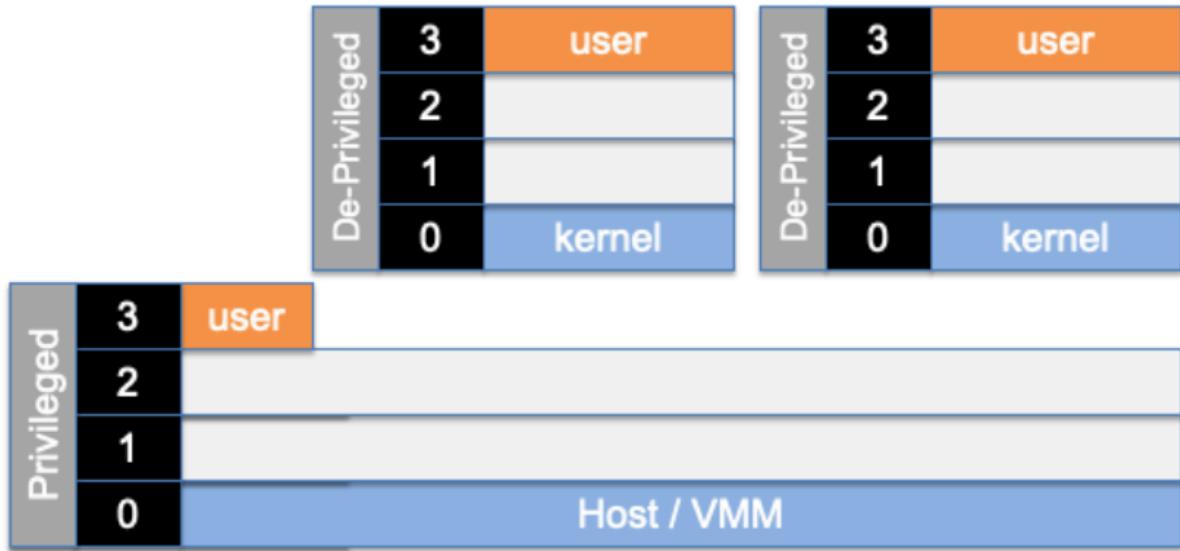
27

Une VM utilise le multiplexing et l'émulation

Chaque VM a son propre CPU virtuel (virtual core).

Chaque VM a son propre guest physical memory

On émule les I/O devices (disque virtuel, clavier virtuel, écran virtuel, NIC virtuel, Ethernet switch virtuelle, etc.).



on a quatre niveaux de protection user, kernel (et on a mis deux niveaux entre les deux, protected librairies par exemple, mais pas vraiment utilisé)