



CentraleSupélec

université
PARIS-SACLAY



ARCHITECTE SOLUTION INFRASTRUCTURE SYSTÈMES D'EXPLOITATION ET VIRTUALISATION

🎓 Mastère Spécialisé Executive - Architecte des Systèmes d'Information
🏛️ CentraleSupélec Executive Education - 2024/2025



Idir AIT SADOUNE
idir.aitsadoune@centralesupelec.fr



IDIR AIT SADOUNE

- **Docteur en Informatique** diplômé par l'[ENSMA](#) en [2010](#).
 - [Thèse](#) sur la modélisation et la vérification des services par une approche basée sur le raffinement et sur la preuve.
- **Enseignant** au sein du département [informatique](#) de [CentraleSupélec - Université Paris-Saclay](#).
- **Chercheur** membre des pôles [Modèles](#) et [Preuve](#) du [LMF - Laboratoire Méthodes Formelles](#).

PLAN

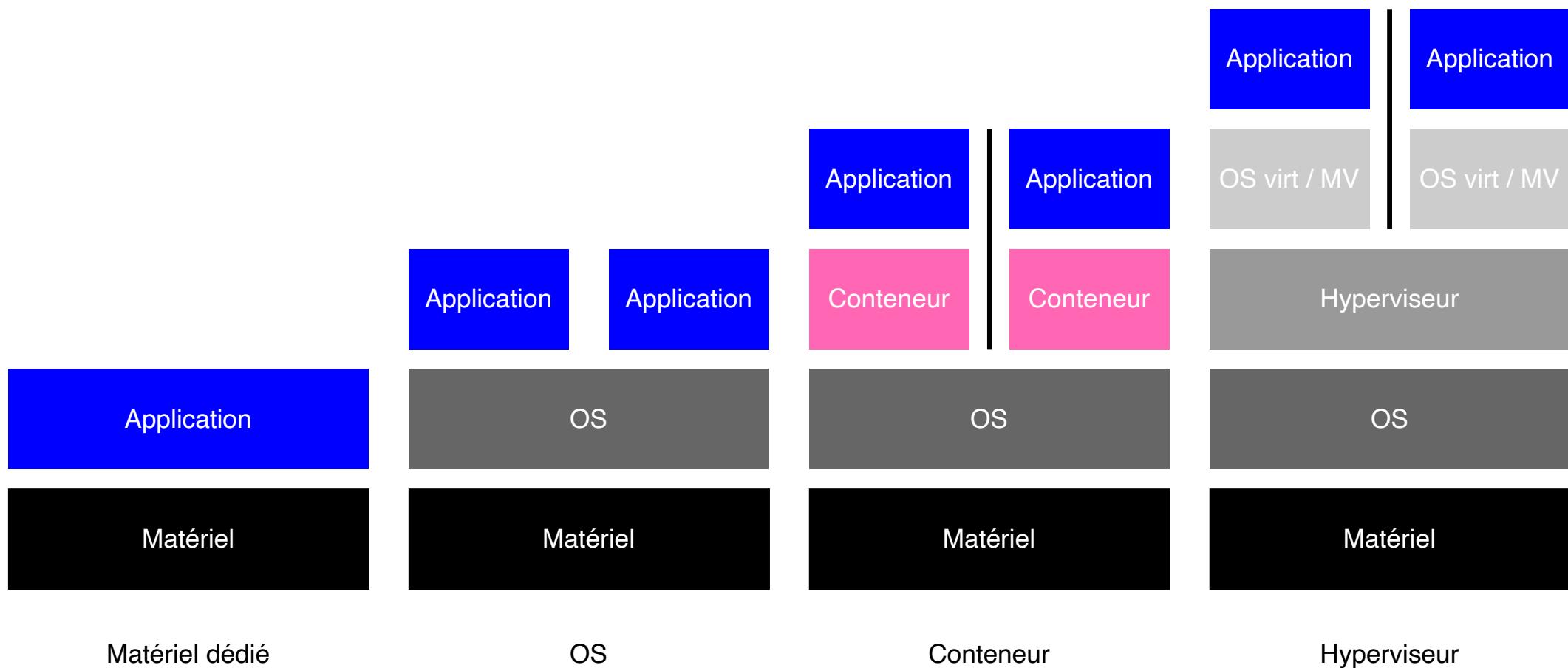
- Architecture des ordinateurs
- Structure et évolution des OS
- Fonctionnalités principales d'un OS
- Virtualisation

[Retour au plan](#) - [Retour à l'accueil](#)

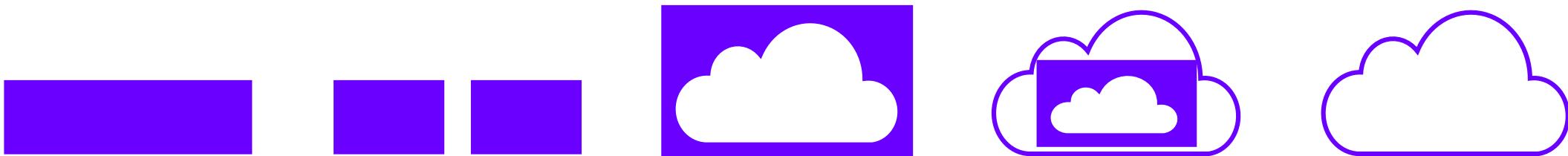
DÉMARCHE POUR ABORDER CE SUJET

- Connaître les **techniques de virtualisation** :
 - proposées par un **OS**
 - proposées par un **hyperviseur**
 - proposées par les **conteneurs**
 - ...
- Comprendre **leurs fonctionnements, leurs limites et leurs performances relatives**.
- Savoir **choisir une technologie de virtualisation en fonction des usages**.

STADES DE VIRTUALISATION



DE LA VIRTUALISATION AU CLOUD



Server Virtualization

Distributed Virtualization

Private Cloud

Hybrid Cloud

Public Cloud

CHEMINEMENT

- Comprendre ce sujet nécessite une compréhension
 - **du matériel** : CPU, mémoire, stockage, périphériques
 - **des OS** : gestion de processus, gestion mémoire, sécurité
 - et **du réseau** : commutation, routage, encapsulation (non abordé)
- Comprendre ces sujets est aussi requis pour appréhender le sujet de **la sécurité des systèmes d'information (SSI)**

SOURCES DE CES TRANSPARENTS

Transparents reposant sur de nombreux auteurs :

*“Idir Ait-Sadoune, Christophe Jacquet, Thibault Le Meur,
Gianluca Quercini, Nicolas Sabouret, Marc-Antoine Weisser, ...”*

PLAN

- Architecture des ordinateurs
- Structure et évolution des OS
- Fonctionnalités principales d'un OS
- Virtualisation

[Retour au plan](#) - [Retour à l'accueil](#)

L'INFORMATIQUE

- L'informatique est la science du **traitement automatique de l'information**.
- Le traitement automatique de l'information s'effectue avec des **programmes informatiques** exécutés par des **machines**
 - **les programmes (software)** décrivent le traitement à réaliser,
 - **les machines (hardware)** exécutent **les programmes**.



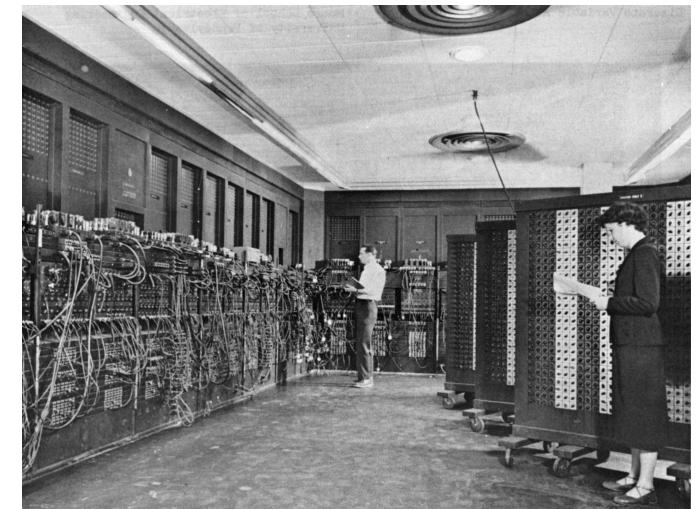
LA NOTION D'ORDINATEUR



- L'ordinateur désigne **un équipement informatique** permettant de traiter des informations en **exécutant des instructions**.
 - ➡ On lui donne **des instructions** (programme/logiciel)
 - ➡ On lui donne **des données** (information)
 - ➡ Il **transforme** les données

ENIAC - 1946

- Construit de 1943 à 1946 par John Mauchley et John Eckert à l'université de Pennsylvanie
- Premier ordinateur entièrement électronique (utilise des tubes à vide).
- Programmé pour résoudre tous les problèmes calculatoires.



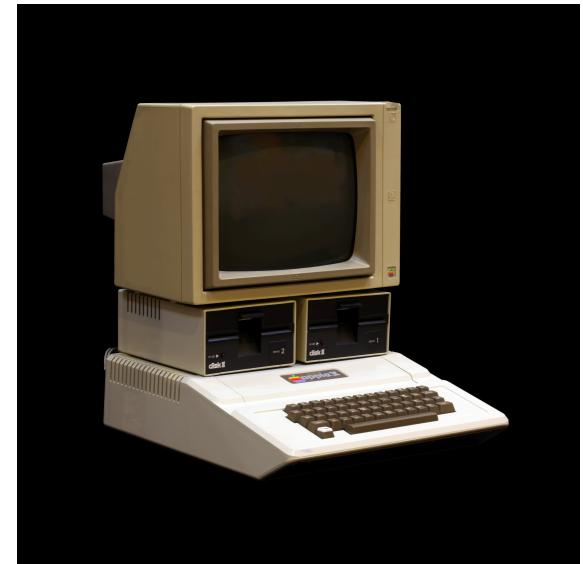
HP 3000 - 1972

- Le **mini-ordinateur** a été une innovation des années **1970**.
- L'intégration de **circuits intégrés à grande échelle** conduit au développement des **micro-processeurs**.



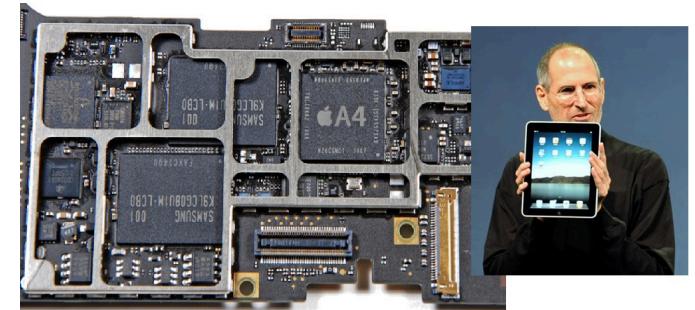
APPLE II - 1977

- Un des premiers **ordinateurs personnels** à **micro-processeur** fabriqué à grande échelle
- Conçu par **Steve Wozniak**, commercialisé le **10 juin 1977** par **Apple**



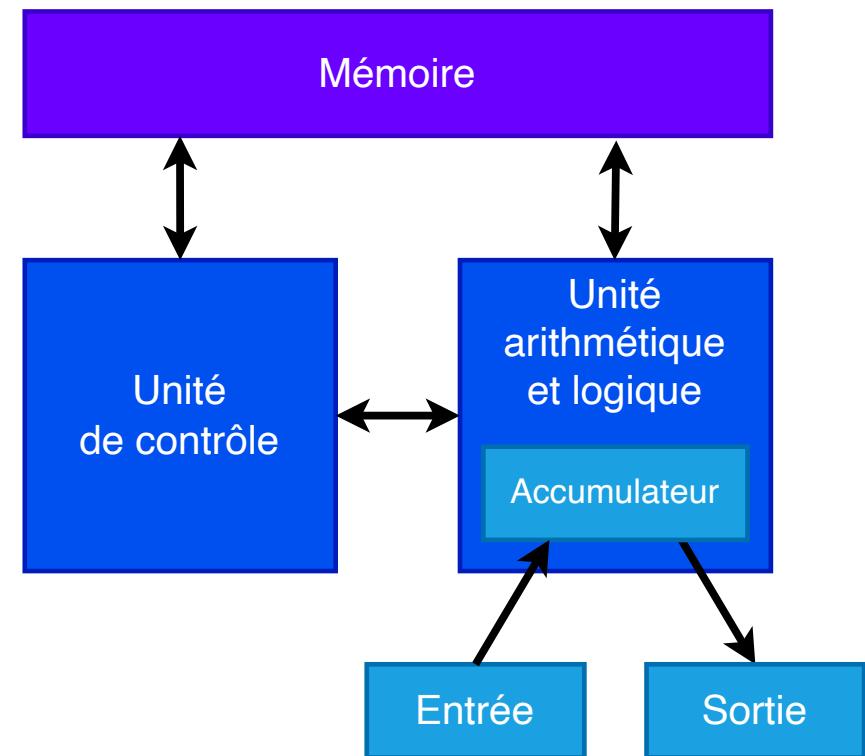
LES ORDINATEURS D'AUJOURD'HUI

- System on a Chip (**SOC**) : un système complet embarqué dans une puce (**circuit intégré**).
- Un **circuit intégré** peut comprendre :
 - un ou plusieurs microprocesseurs
 - de la mémoire
 - des périphériques d'interface
 - ou tout autre composant

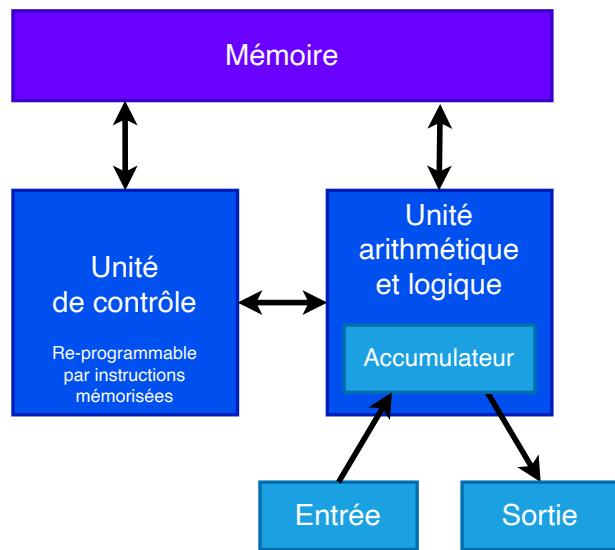


L'ARCHITECTURE DE VON NEUMANN

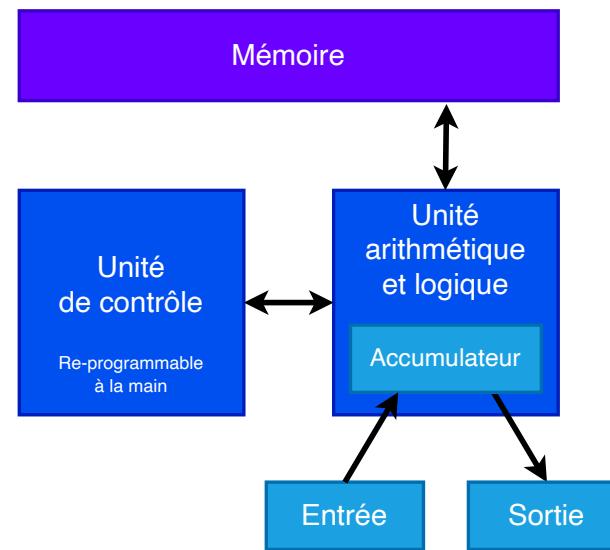
- L'architecture de von Neumann :
un **modèle** pour un ordinateur avec
une **mémoire unique** pour conserver
 - les instructions
 - et les données



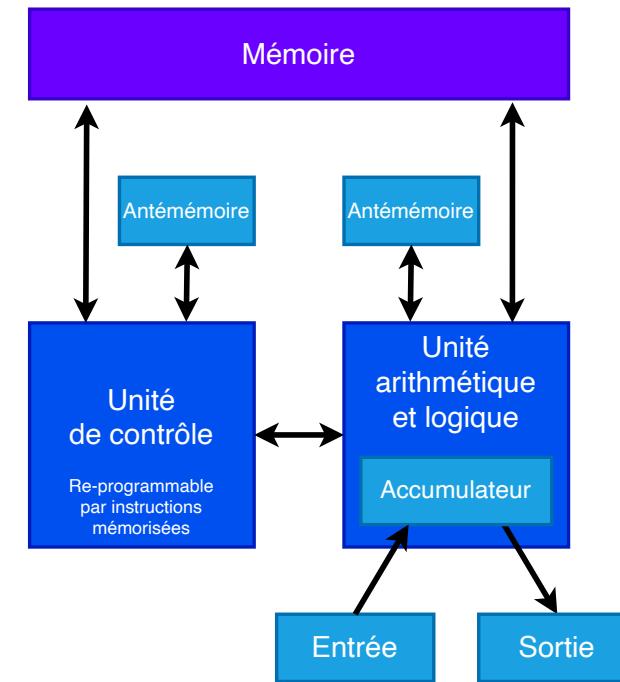
VON NEUMANN / L'ORDINATEUR MODERNE



Von Neumann



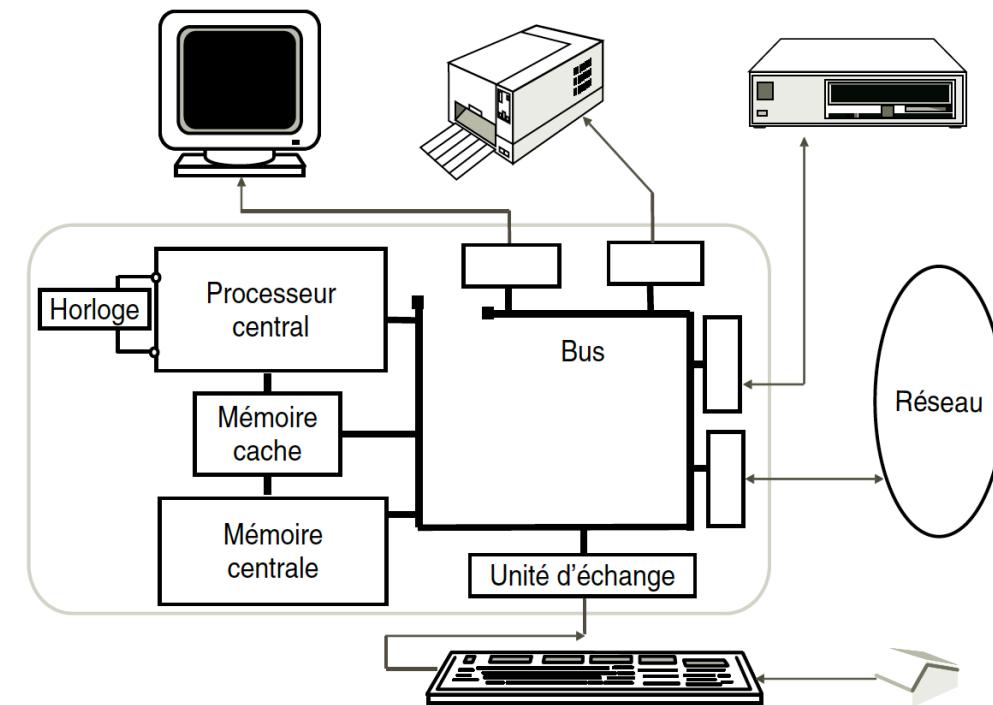
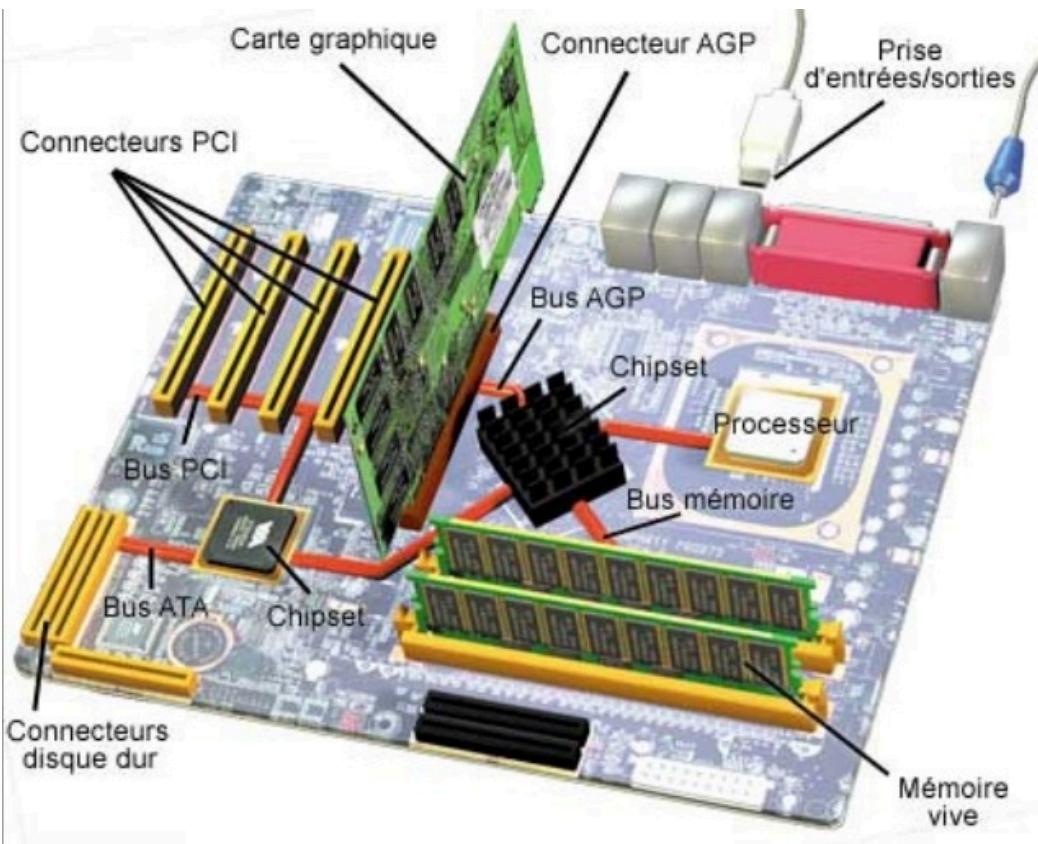
ENIAC



Ordinateur personnel

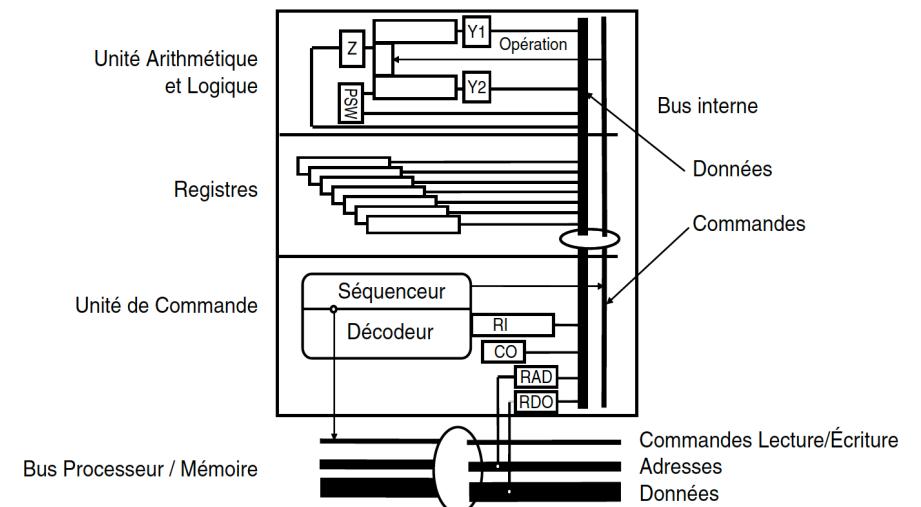
source : [la thèse d'Alexandre Brunet](#)

STRUCTURE GÉNÉRALE D'UN ORDINATEUR



L'ARCHITECTURE D'UN MICROPROCESSEUR

- Le microprocesseur (**CPU**) exécute les instructions machines placées en mémoire centrale.
- Le **CPU** est constitué de quatre parties
 1. l'unité arithmétique et logique (**UAL**),
 2. les registres,
 3. l'unité de commande,
 4. le bus de communication interne.

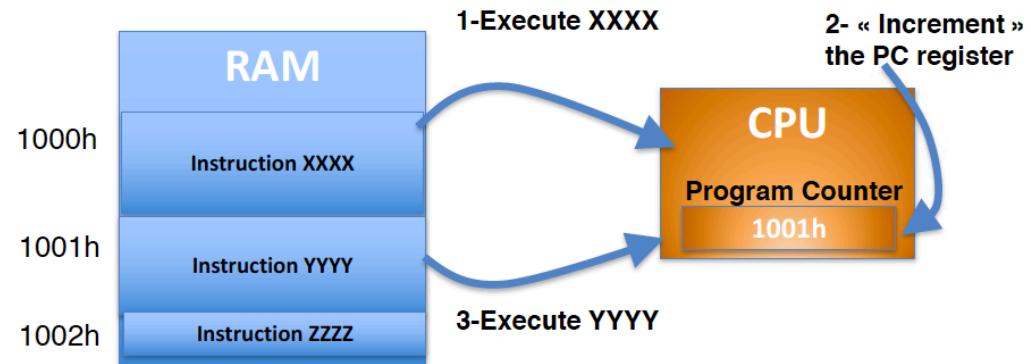


LE MICROPROCESSEUR ET SES REGISTRES

- **Registres génériques**
 - Peuvent contenir des données, adresses mémoire
 - Utilisés comme opérandes de calculs, stockage temporaire, ...
- **Registres spécialisés**, comme par exemple :
 - Program Counter
 - Stack Pointer
 - Interrupt Description Table
 - Page Table Pointer
 - ...

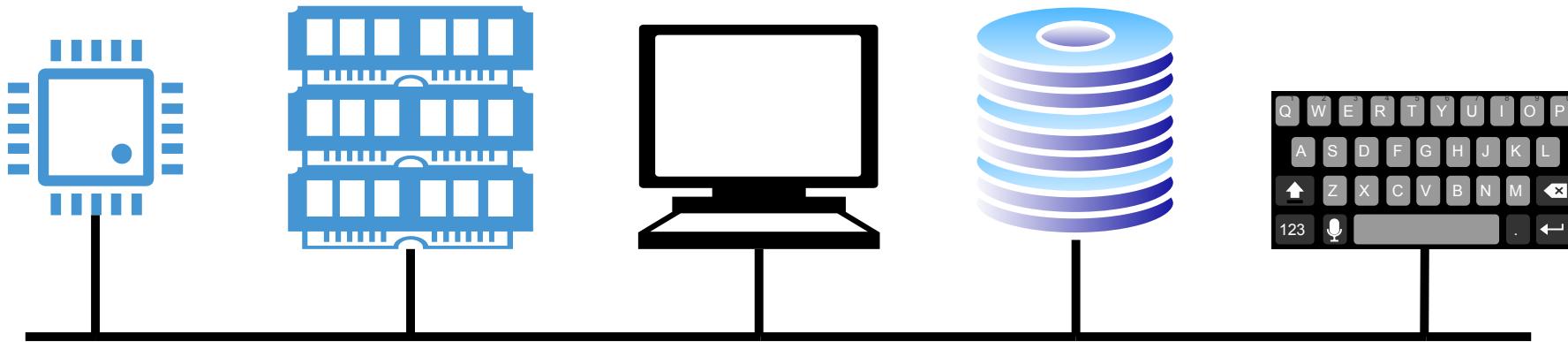
CHARGEMENT D'UN PROGRAMME EN MÉMOIRE

- Un programme est un flux d'instructions.
- Pour être exécutable, un programme doit être présent en RAM.
- Le registre Program Counter (PC) du CPU indique l'adresse de *la prochaine instruction à exécuter*.



LE FONCTIONNEMENT DE L'ORDINATEUR

Comment fonctionne un ordinateur ?



Tout cela n'est que des fils électriques ...
... qu'on allume et qu'on éteint.

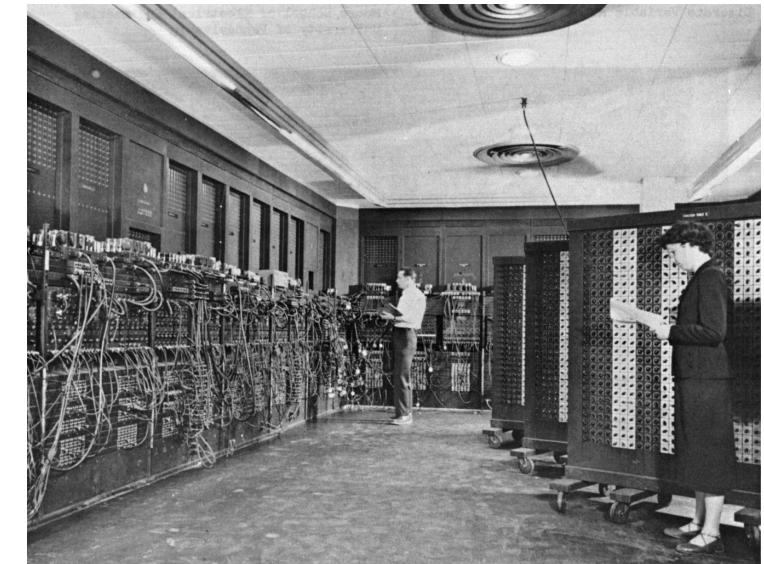
PLAN

- Architecture des ordinateurs
- Structure et évolution des OS
- Fonctionnalités principales d'un OS
- Virtualisation

[Retour au plan](#) - [Retour à l'accueil](#)

AUTREFOIS : ENIAC

- Premier ordinateur **entièlement électronique** :
 - 18 000 tubes à vide
 - 1 500 relais
 - 20 registres de 10 chiffres décimaux
 - programmé à l'aide de 6 000 commutateurs
- **La programmation** se faisait directement **en langage machine**
- **Un seul programme** à la fois pouvait s'exécuté.
- L'absence d'un OS obligeait le programmeur à **charger manuellement le programme**

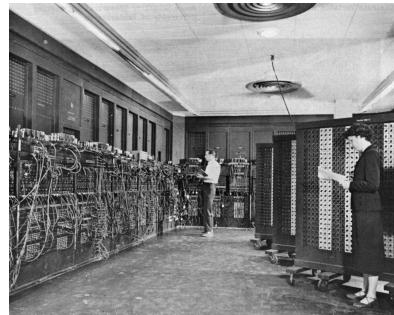


AUTREFOIS : IBM RAMAC 305



- Premier ordinateur à disque dur (l'**IBM 350**) commercialisé en **septembre 1956** par **IBM**.
- Composé des éléments suivants : unité de traitement, imprimante, console, alimentation, disque dur, mémoire 5Mo.
- L'**unité de traitement** est basée sur un tambour magnétique sur lequel est stocké le programme.
- Un opérateur programme à l'aide de **cartes perforées** et inscrit les données sur le tambour.

AUTOMATISER LES TÂCHES



- Comment **automatiser les tâches** des opérateurs et des programmeurs ?
- Écrire un **programme informatique** qui:
 - ➡ décide qui fait quoi et à quel moment
 - ➡ fait le lien entre les applications et le matériel

DÉFINITION

“Un système d'exploitation est un ensemble de programmes réalisant l'interface entre le matériel et les utilisateurs.”

- gère la partie **matérielle**
- sert de socle pour les **applications**

Applications



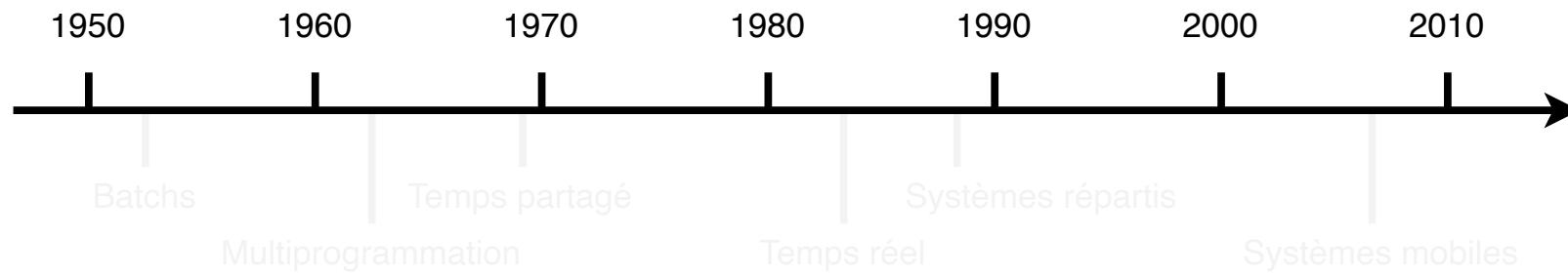
Système d'exploitation



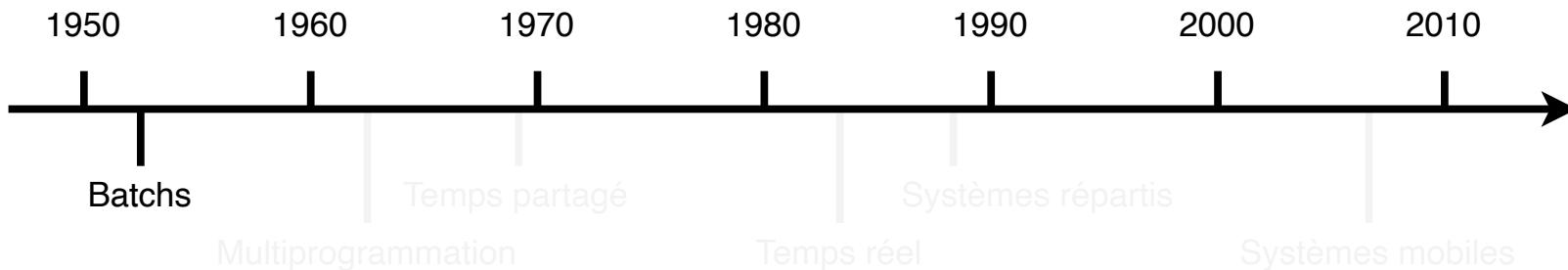
Matériel



HISTORIQUE/TYPES DES OS

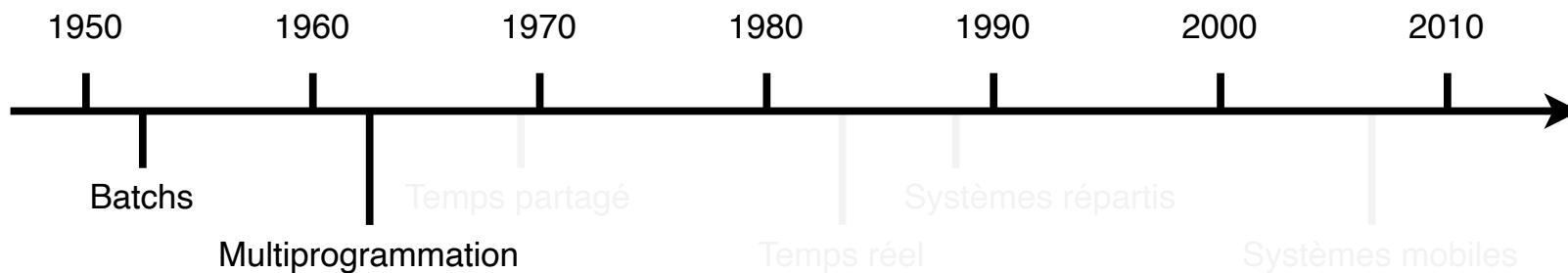


HISTORIQUE/TYPES DES OS



- **Les systèmes batch** sont basés sur deux programmes :
 1. **le chargeur** : charger les programmes dans la mémoire centrale depuis les cartes perforées ou le dérouleur de bandes.
 2. **le moniteur d'enchaînement de traitements** : permettre l'enchaînement des travaux soumis à la place de l'opérateur.
- **Les systèmes batch** automatisent les tâches de préparation des travaux et exploitent efficacement le processeur.

HISTORIQUE/TYPES DES OS

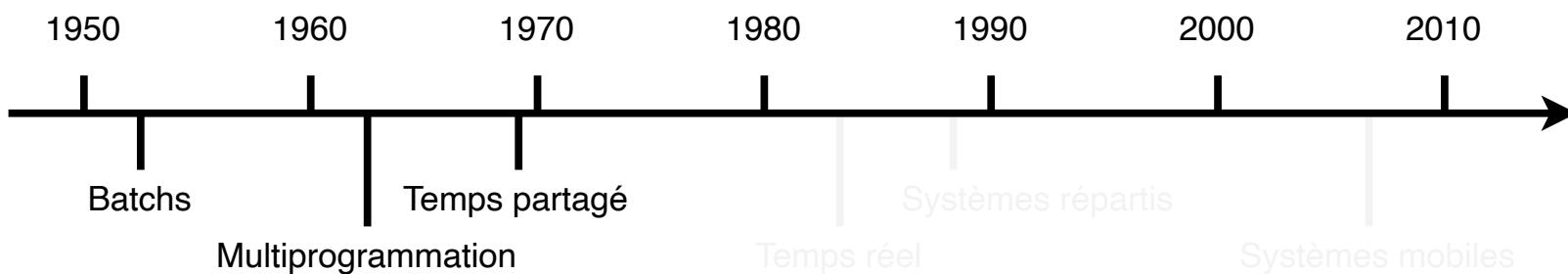


Utiliser plusieurs composants en parallèle, ce qui nécessite:

- **Gestion de la priorité** (*quel processus peut accéder à la ressource*)
➡ ordonnancement
- **Mémoire partagée** (*gérer des informations de plusieurs processus*)
➡ adressage et mémoire

Exemple : MULTICS

HISTORIQUE/TYPES DES OS

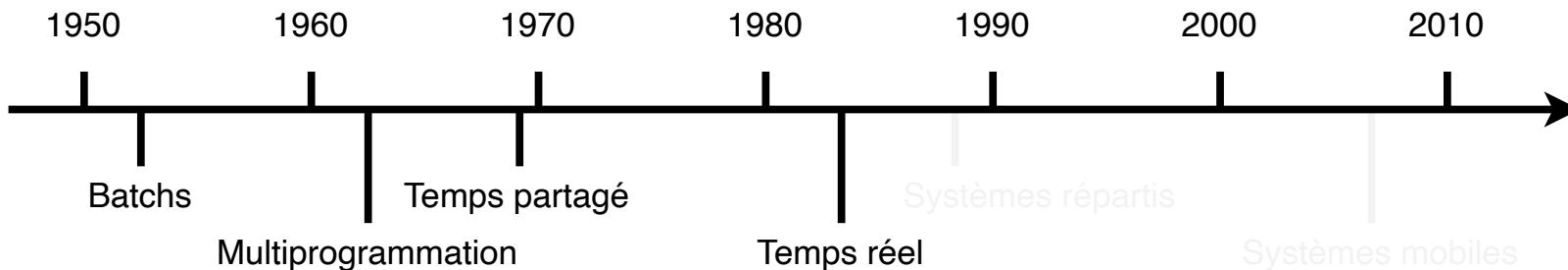


Plusieurs processus actifs alternant sur le processeur

- Gestion des interruptions
- Cycle de vie du processus
- Synchronisation de processus et programmation concurrente

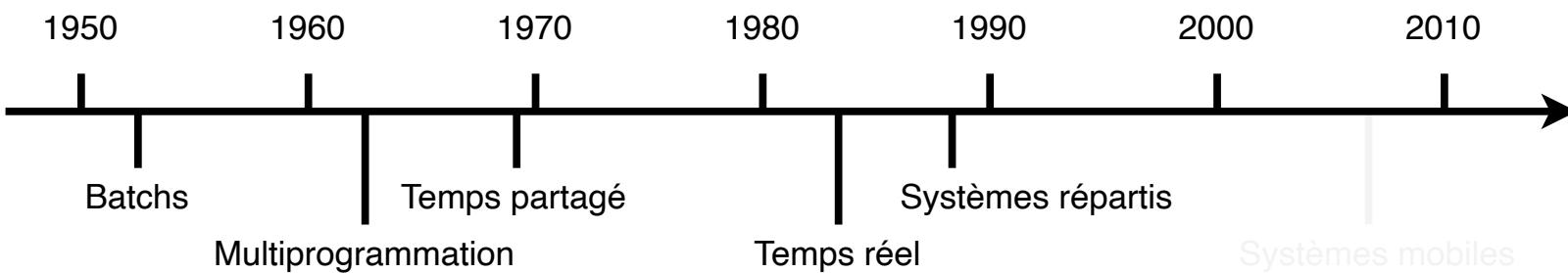
Exemple : UNICS ou UNIX

HISTORIQUE/TYPES DES OS



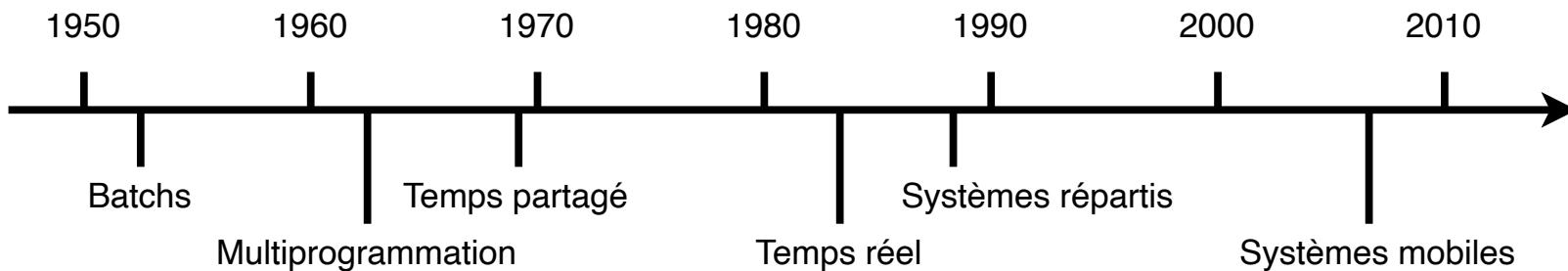
- **Gestion des délais:** contrainte de **temps de réponse**
➡ les processus doivent répondre vite
- Apparition des **micro-ordinateurs**
 - CP/M → IBM PC (MSDOS)
- Apparition des **interfaces graphiques**
 - Xerox → Apple Macintosh 1984, Windows 95, Linux 1991

HISTORIQUE/TYPES DES OS



- **Les ordinateurs communiquent** pour échanger des données !
 - Arpanet (1967) conçu par la DARPA
 - E-mail (1972) avec Ray Tomlinson
 - TCP/IP(1972)
 - Clients-Serveur années 80 → NFS - Network File System (Sun, 1984)
 - Arpanet ouvert fin 80 → Web début 90 (CERN , Tim Berners-Lee)

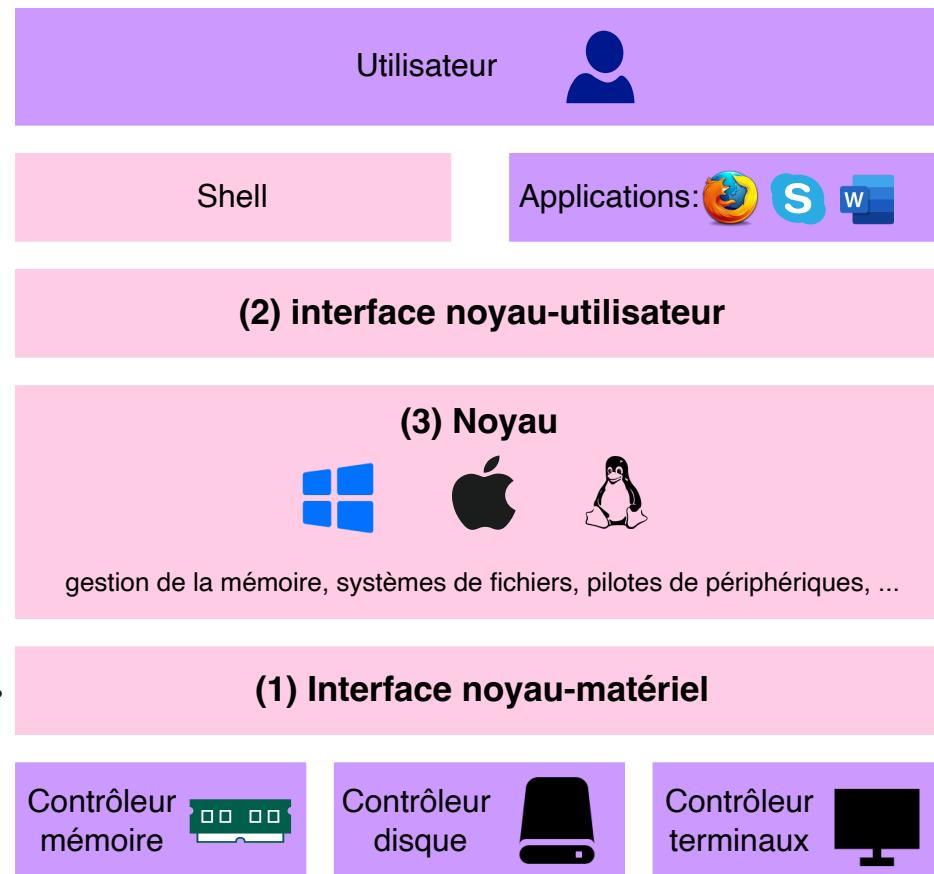
HISTORIQUE/TYPES DES OS



- **Les ordinateurs de poche** existent depuis les années 80
 - 1986 : sortie des PDA → PalmOS
 - 2007 : sortie des smartphones → android OS
 - 2007 : sortie de l'iPhone → iOS

RÔLES DU SYSTÈME D'EXPLOITATION

1. **L'interface noyau-matériel** prend en charge la gestion et le partage des ressources de la machine.
2. **L'interface noyau-utilisateur** construit une machine virtuelle plus facile d'emploi et plus conviviale.
3. **Le noyau** assure plusieurs grandes fonctionnalités.



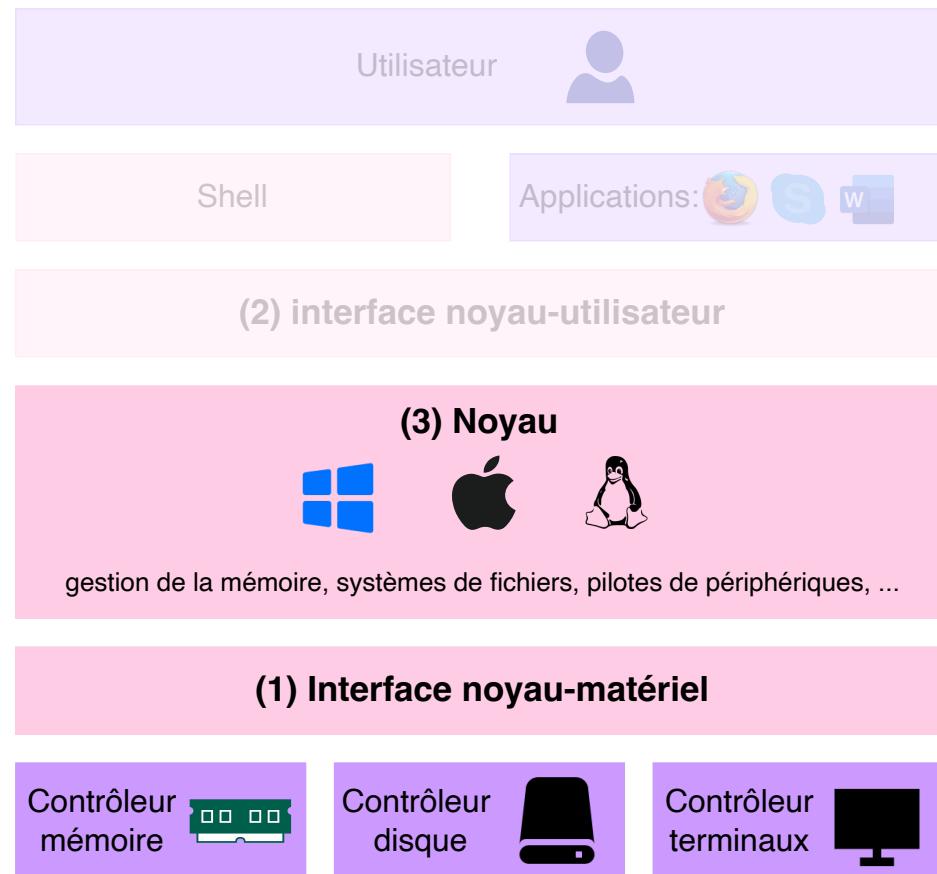
INTERFACE NOYAU-MATÉRIEL

- Gérer l'accès et le partage des ressources matérielles (arbitrage).

- processeur
- mémoire centrale
- périphériques
- ...

- Cet arbitrage doit assurer:

- l'équité d'accès aux ressources
- la protection de l'accès aux ressources
- la cohérence des états des ressources

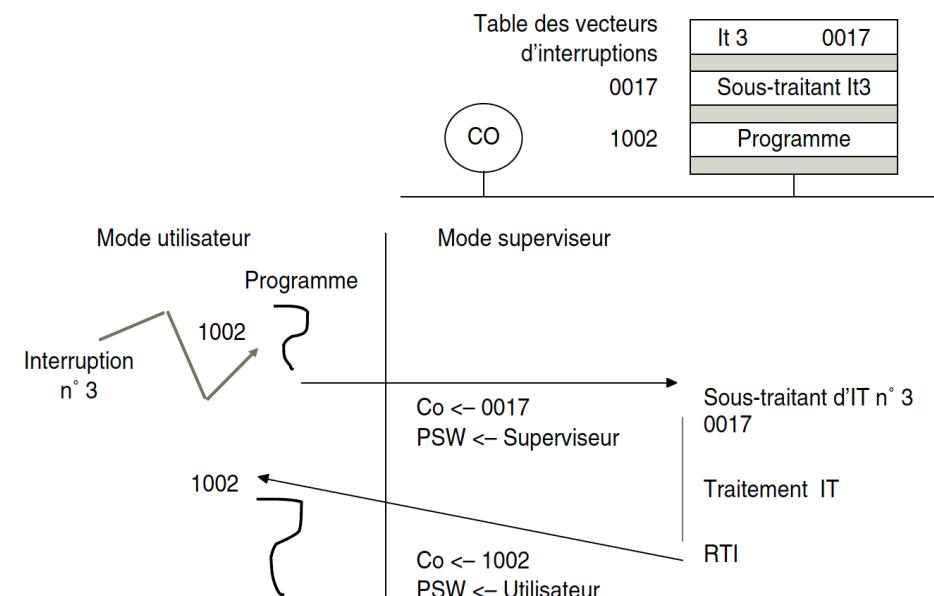


INTERRUPTION - IRQ

- L'OS s'interface avec la couche matérielle, par le biais du mécanisme des **interruptions (Interrupt ReQuest ou IRQ)**.
 - ➡ prendre connaissance des **événements survenant sur le matérielle**
- L'**IRQ** est **un signal (code)** permettant à un dispositif externe d'interrompre le processeur pour lancer un traitement particulier.
 - à chaque code correspond **une routine de traitement** de l'OS.
 - les adresses des routines sont dans **une table** placée en mémoire (**la table des vecteurs d'interruptions**).
 - **les routines d'interruptions** sont chargées en mémoire au moment du chargement de l'OS et exécutées en **mode superviseur**.

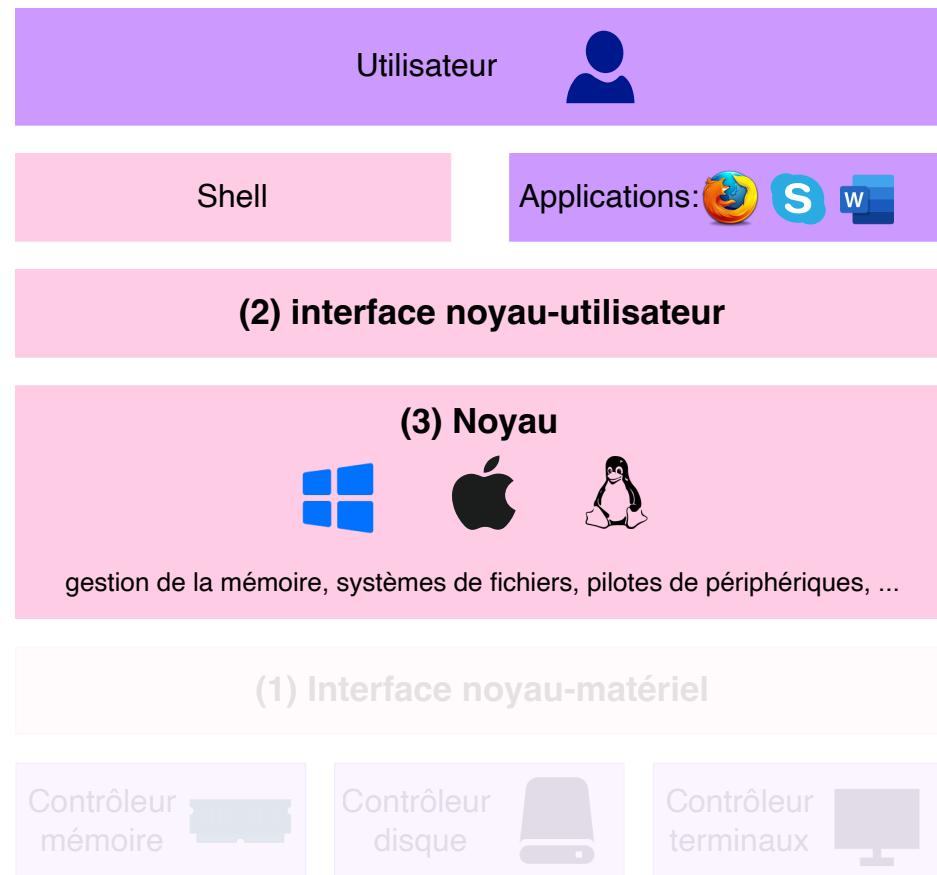
PRISE EN COMPTE D'UNE IRQ

- Enregistrer → **pile de l'OS**
 - l'adresse d'**instruction interrompue**
 - l'**état du processeur** (registres)
- Passer en **mode superviseur**
- Charger la **routine correspondant à l'interruption**



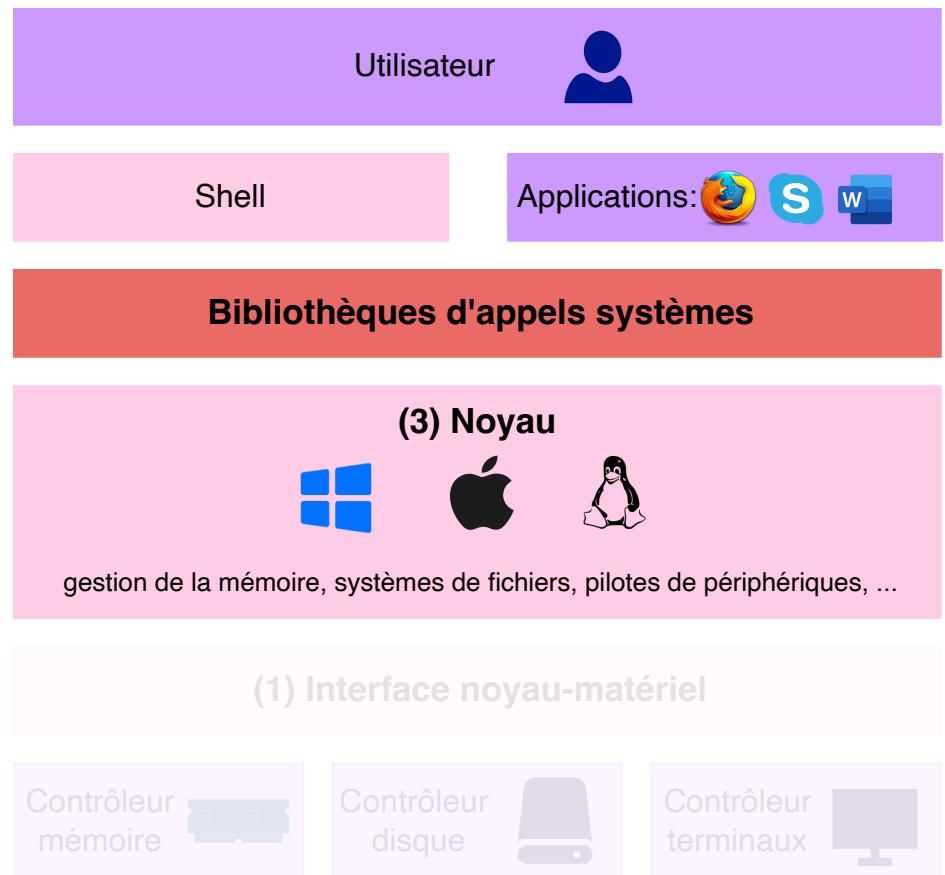
INTERFACE NOYAU-UTILISATEUR

- Présenter une **interface** entre le hardware et les applications.
➡ une **interface simplifiée et unifiée**.
- Présenter au-dessus de la machine physique, **une machine virtuelle** plus simple et plus conviviale.
- Créer **l'illusion de vrais ressources physiques** (processeur, mémoire, périphérique ...).



LES APPELS SYSTÈMES

- Fournir une **interface d'accès** aux ressources matérielles.
 - ➡ par le biais de **fonctions prédéfinies (appels/routines systèmes)**.
 - ➡ les points d'entrées aux **fonctionnalités de l'OS**.



EXEMPLES D'APPELS SYSTÈMES

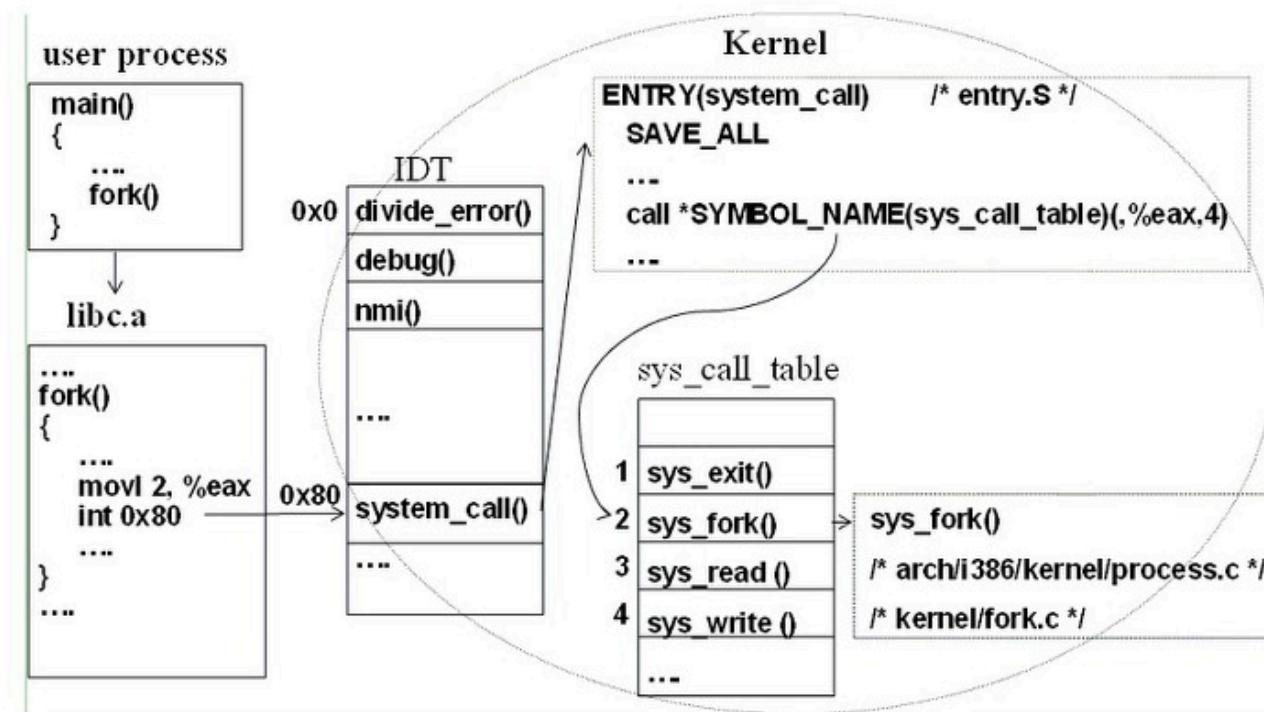
- **Contrôle de processus**

- `sys_fork` : créer un processus
- `sys_wait` : attendre la terminaison d'un processus
- `sys_exit` : terminer l'exécution d'un processus
- `sys_kill` : Envoyer un signal à un processus

- **Gestion des fichiers**

- `sys_open/sys_close` : ouvrir/fermer un fichier
- `sys_read/sys_write` : lire/écrire des données dans un fichier
- `sys_mkdir/sys_rmdir` : créer/supprimer un répertoire

L'APPEL SYSTÈME `fork()`

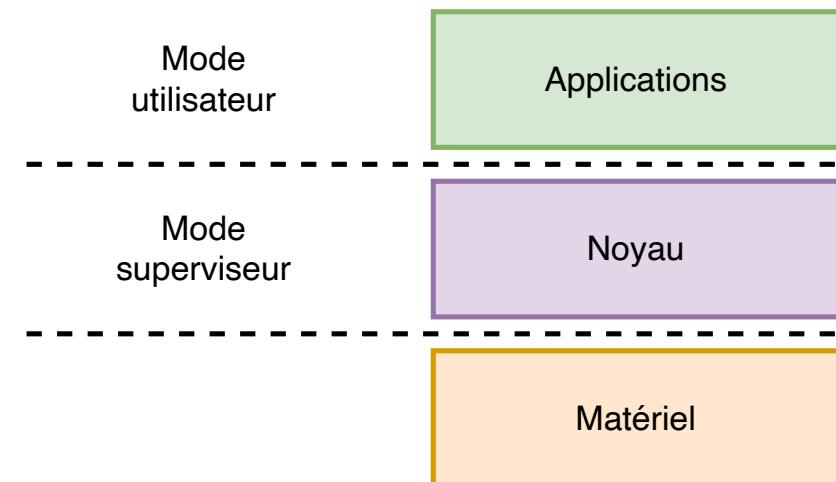


EXAMPLE SOUS UNIX

- L'instruction `os.chdir(path)` permet de changer le répertoire courant d'un programme **Python** en cours d'exécution.
- La commande `cd path` permet de changer le répertoire courant depuis l'**interpréteur de commandes (Shell)**.
- Les deux exécutent la routine système `sys_chdir`.

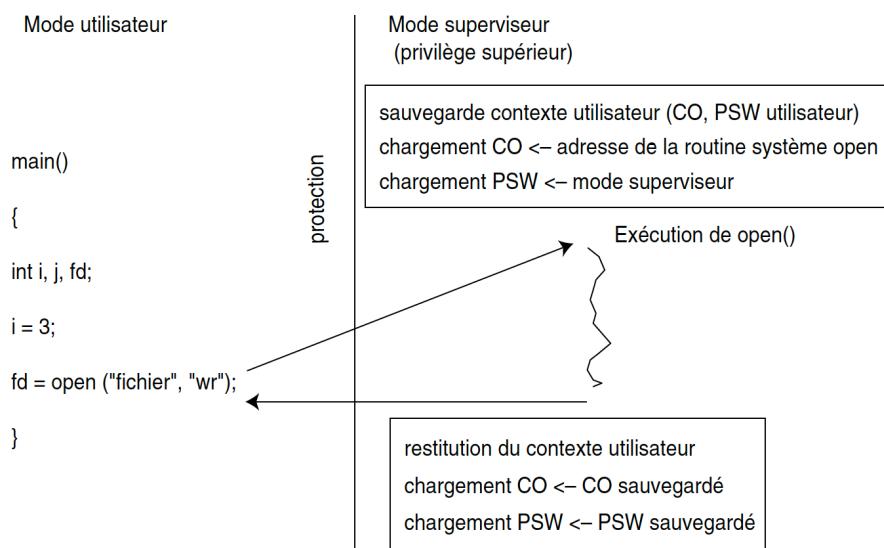
MODES D'EXÉCUTIONS

- Un programme utilisateur s'exécute dans un mode utilisateur :
 - un jeu d'instructions restreint pour protéger la machine.
 - ex. manipulation des IRQs interdite.
- L'OS s'exécute dans un mode privilégié (mode superviseur):
 - aucune restriction de droits n'existe.



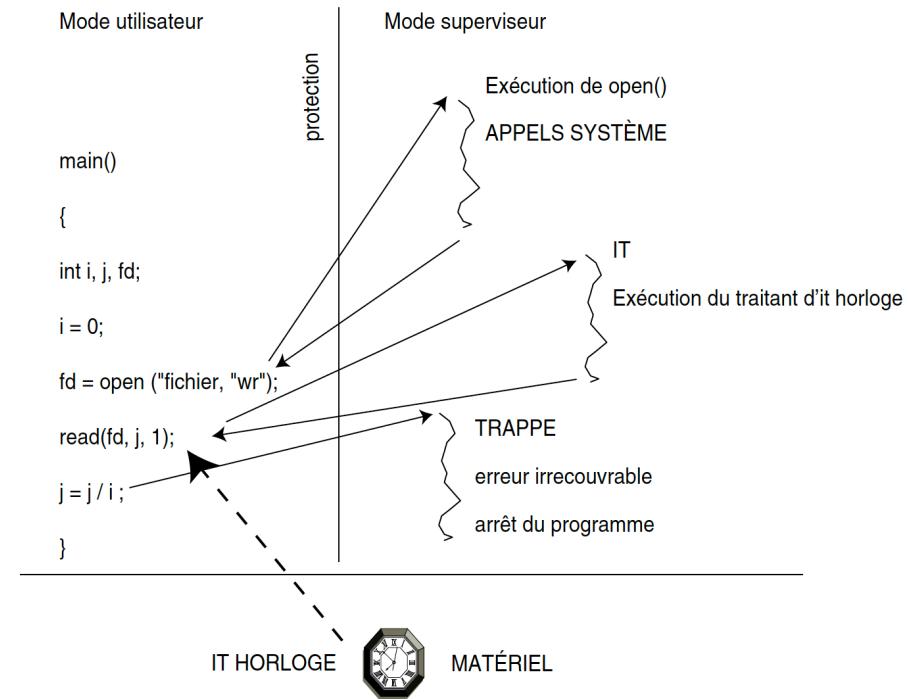
COMMUTATIONS DE CONTEXTE

- A l'appel d'une fonction du noyau, il y a passage au **mode superviseur** (**commutation de contexte**).
- A la fin de l'exécution de la fonction du noyau, le programme repasse au **mode utilisateur**.
 - commutation de contexte avec **restauration du contexte utilisateur**.
 - reprise de l'exécution du programme utilisateur



LES CAUSES DE COMMUTATIONS DE CONTEXTE

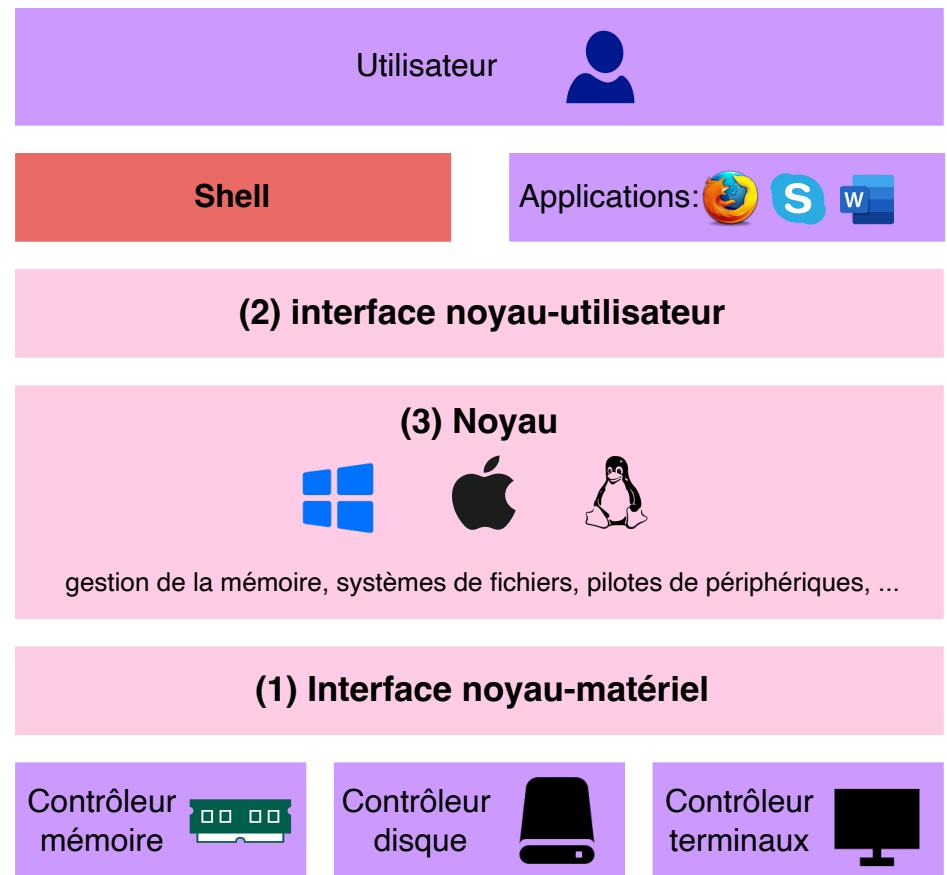
1. appelle d'une **fonction système**.
2. exécute une opération illicite (**trappe** ou **exception**).
3. prise en compte d'une **interruption matérielle**.



Trappes et appels systèmes sont parfois qualifiés d'interruptions logicielles par opposition aux interruptions matérielles.

INTERPRÉTEUR DE COMMANDE (SHELL)

- **Langage de commandes :**
l'interface de niveau utilisateur avec le système d'exploitation.
- **Interpréteur de commandes :**
exécuter des commandes de l'utilisateur en appellant la routine système appropriée.



INTERPRÉTEUR DE COMMANDE (SHELL)

- Chaque système d'exploitation a son propre **langage de commandes**:
 - **MSDOS/Unix** : console + clavier
 - **Mac/Windows** : souris + clavier
 - **iOS/Android** : boutons + écran tactile

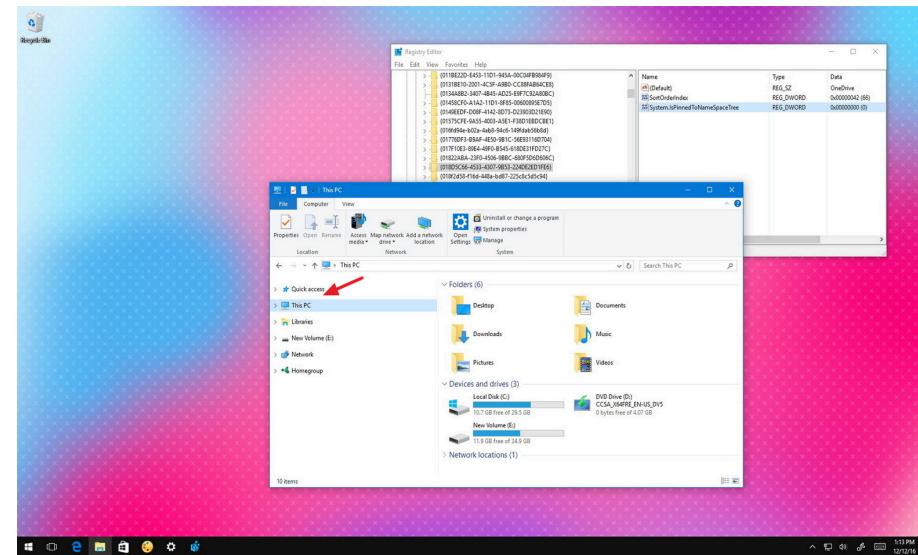
```
C:\>unformat /?
Récupère un disque détruit par la commande FORMAT
ou par la commande RECOVER.

UNFORMAT lecteur: [/J]
UNFORMAT lecteur: [/U] [/L] [/TEST] [/P]
UNFORMAT /PARTN [/L]

lecteur: Lecteur à récupérer.
/J      Vérifie que les fichiers MIRROR correspondent à l'information
       système sur le disque.
/U      Restaure sans utiliser les fichiers MIRROR.
/L      Affiche les noms de tous les fichiers et répertoires trouvés,
       ou, en conjonction avec /PARTN, affiche la table des partitions.
/TEST   Affiche les infos mais n'écrira pas les modifications sur disque.
/P      Envoie les messages sur l'imprimante connectée au port LPT1.
/PARTN Restaure la table des partitions du disque.

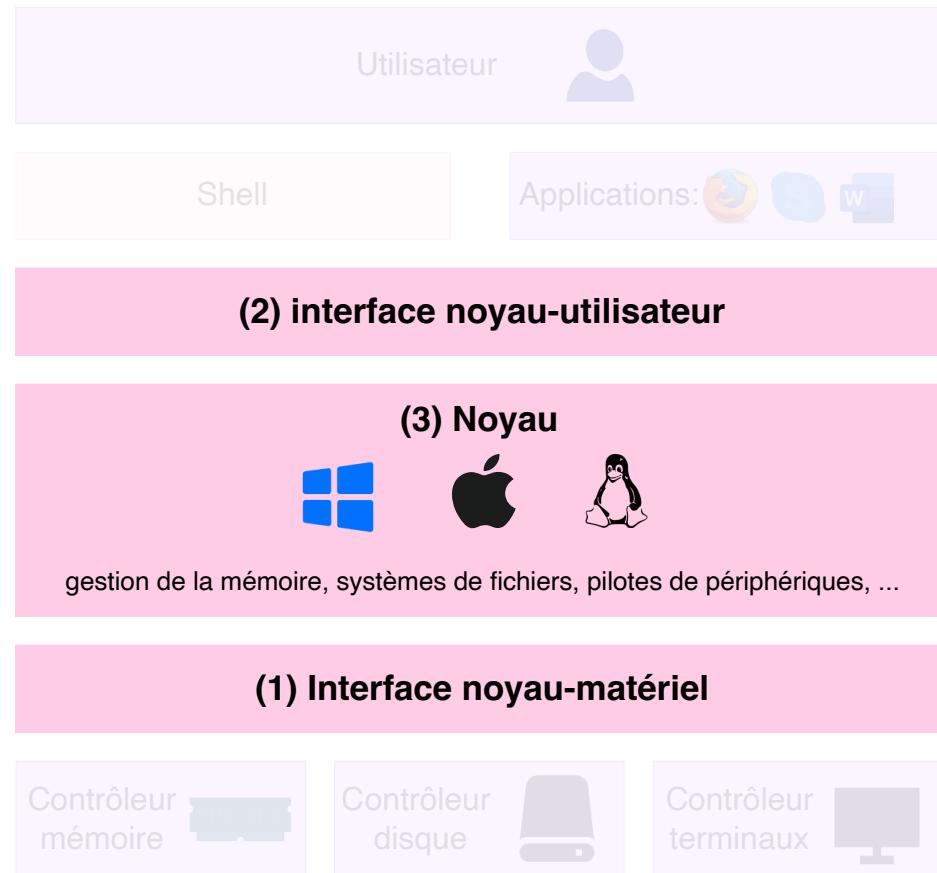
MIRROR, UNDELETE et UNFORMAT Copyright (C) 1987-1993 Central Point Software,
Inc.

C:\>_
```



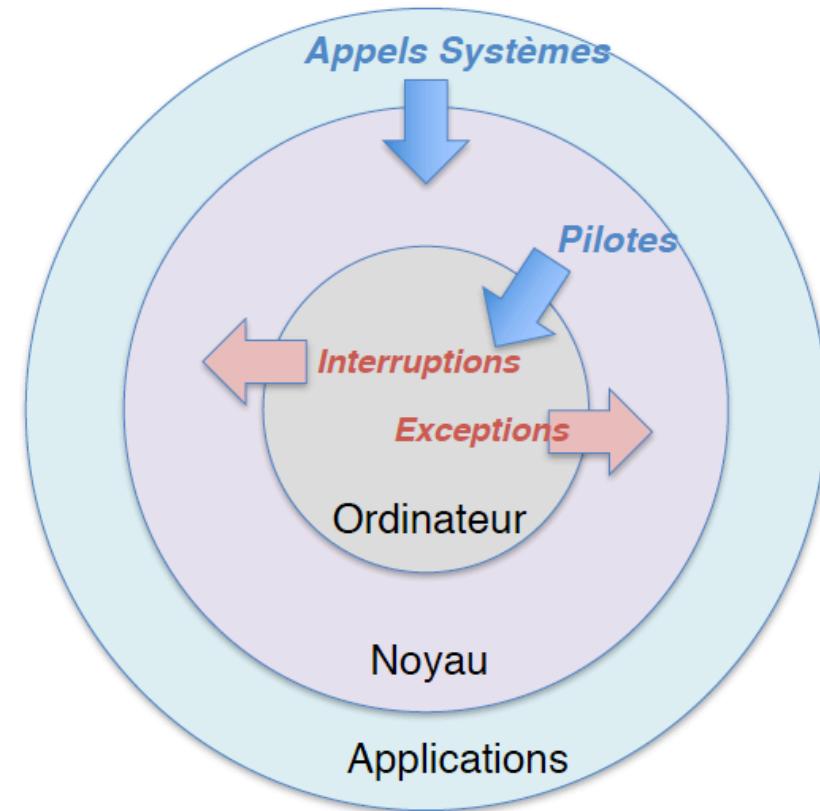
NOYAU D'UN SYSTÈME D'EXPLOITATION

- **Gestion des entrées/sorties (I/O)**
 - contrôleurs, pilotes, ...
- **Gestion des processus**
 - ordonnancement, synchronisation, ...
- **Gestion mémoire**
 - allocation, gestion des espaces, ...
- **Gestion du stockage secondaire**
 - système de fichiers, ...
- **Gestion de la sécurité**



ORGANISATION GÉNÉRALE DE L'OS

- **Interruptions** : évènements produits par le matériel.
- **Exceptions** : événements générés par le processeur.
- **Pilotes (drivers)** : applications contrôlant les périphériques.
- **Noyau (kernel)** : application rendant des services généraux.
- **Appels Systèmes** : demandes de services.

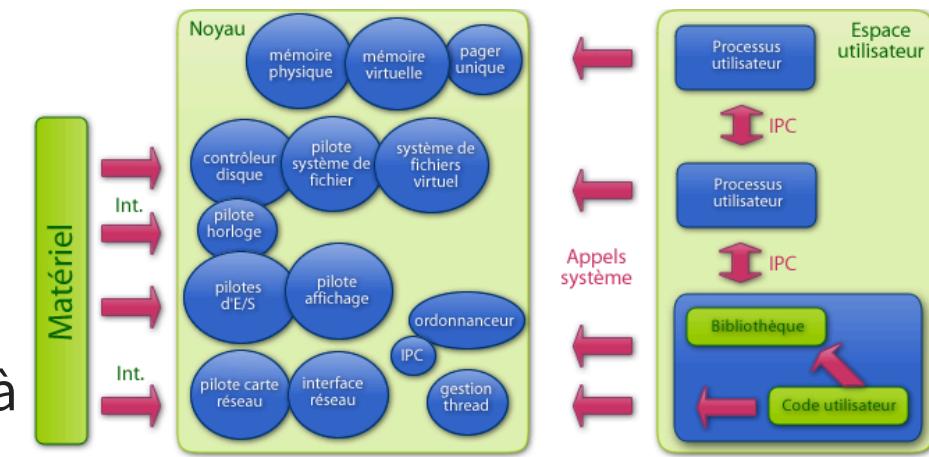


STRUCTURE DES OS

- Comment organiser les différentes fonctions d'un **OS** ?
 - ➡ Qu'est-ce qui est dans le **noyau** (en **mode Superviseur**) ?
 - ➡ Comment interagissent les différents composants ?

NOYAUX MONOLITHIQUES

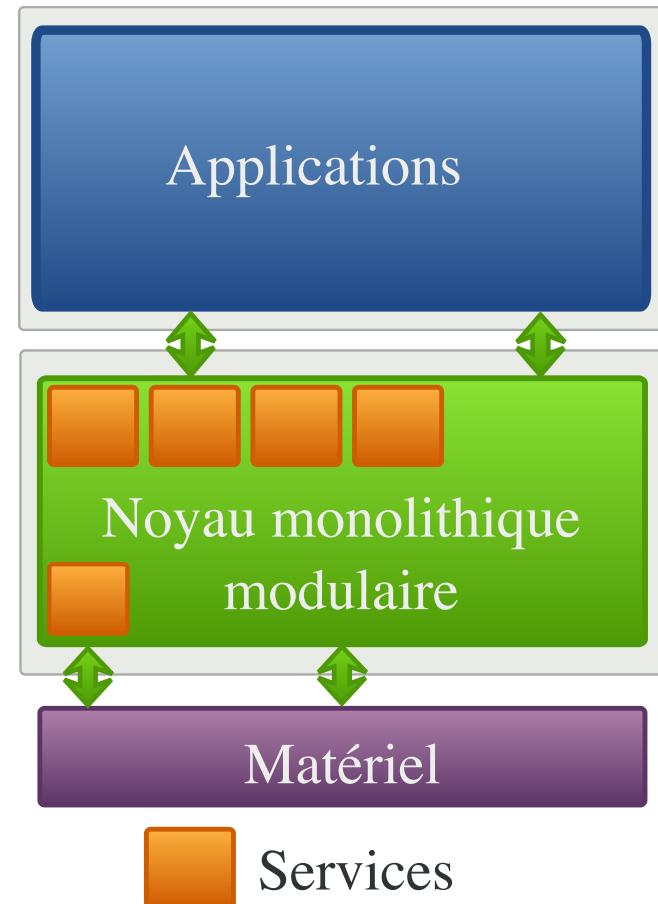
- L'ensemble des fonctions/pilotes sont regroupés dans **un seul bloc**.
- **Ex.** anciennes versions de **Linux** ou certains **vieux Unix**.
- Les OS **monolithiques** sont rapides mais délicats à maintenir.



source : <https://fr.wikipedia.org>

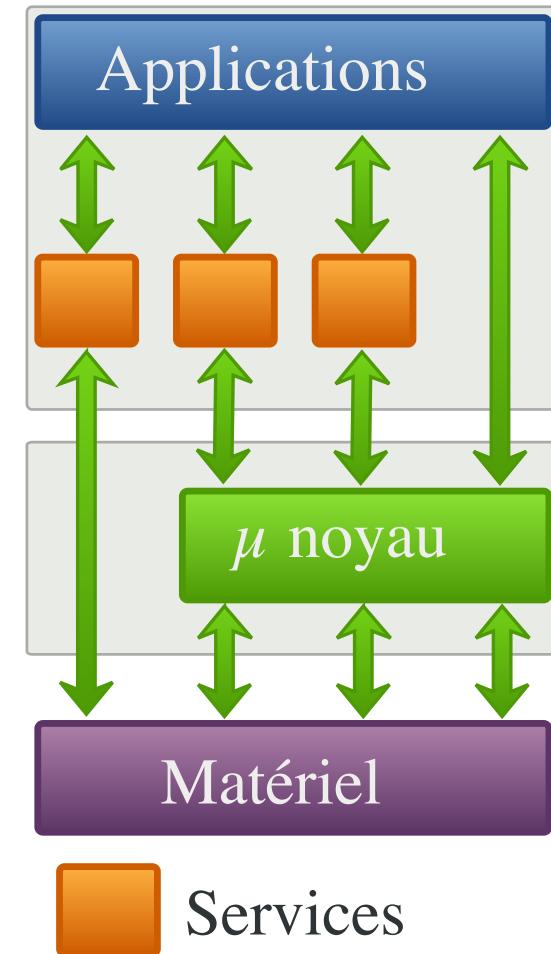
NOYAUX MONOLITHIQUES MODULAIRES

- Seules **les parties fondamentales** de l'OS sont regroupées dans **un bloc unique**.
- Les autres fonctions (**ex.** les pilotes) sont regroupées dans des **modules séparés**.
- **Ex.** Linux ou **Solaris**.
- Les OS **monolithiques modulaires** ne sont pas faciles à concevoir (dépendances multiples).



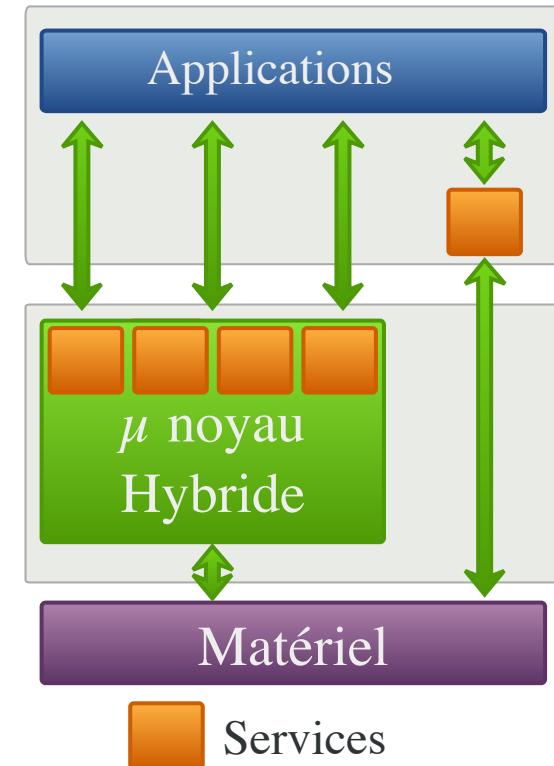
SYSTÈMES À MICRO-NOYAUX

- **Minimiser les fonctionnalités dépendantes** du noyau en plaçant des services l'extérieur.
- **Éloigner les services « à risque »** des parties critiques de l'OS regroupées dans le noyau.
- **Ex. Mach de Mac OS X.**
- Les OS à **micro-noyaux** pur sont trop lents.



SYSTÈMES À NOYAUX HYBRIDES

- Combiner des noyaux **monolithiques** et des **micro-noyaux** pour avoir les avantages des deux.
- Ex. XNU de Mac OS X.



source : <https://fr.wikipedia.org>

CHARGEMENT D'UN OS

- L'**OS** est le **premier programme exécuté** lors de la mise en marche de l'ordinateur, après l'amorçage (**boot**).
- Le **boot** (**bootstrap**) désigne les **étapes successives du démarrage**.

LES ÉTAPES DU BOOT

1. le **POST** test - **Power On Self Test**

- après un start ou un reset, le processeur charge **les premières instructions** à partir de la **ROM du BIOS** situées à l'adresse **FFFF0**.
- des instructions de **branchement vers un programme du BIOS** qui **initialise et teste les fonctions vitales du hardware**

2. le chargement du **MBR** - **Master Boot Record**

- si le **POST** réussit, il consultera la **RAM CMOS** pour identifier le **disque système** dont le premier secteur est appelé **MBR**.
- le code du **MBR** teste la table de partition pour charger la partition contenant le secteur d'amorçage avec l'**IPL - Initial Program Load**.
 - ➡ l'**IPL** charge l'**OS** ou le **bootmanager** en RAM.
 - ➡ l'**OS** est lancé

PLAN

- Architecture des ordinateurs
- Structure et évolution des OS
- Fonctionnalités principales d'un OS
- Virtualisation

[Retour au plan](#) - [Retour à l'accueil](#)

FONCTIONNALITÉS PRINCIPALES D'UN OS

1. Gestion des processus
2. Gestion de la mémoire
3. Mémoire virtuelle
4. Système de fichiers
5. Sécurité

GESTION DES PROCESSUS

⌚ Fonctionnalités principales d'un OS

DÉFINITIONS

- Un **programme informatique** : une suite statique d'instructions.
- Un **processeur** : un **automate** (électronique) de **traitement**.
 - Il peut **exécuter** un programme
 - Il **modifie son état** en fonction des instructions
- Un **processus** : un programme exécuté par un processeur.
 - capte le **caractère dynamique** d'un programme.

PROGRAMME vs PROCESSUS

- Un **processus** : un programme exécuté par un processeur.
 - capte le **caractère dynamique** d'un programme.
- **Un programme** peut donner **naissance à plusieurs processus**.
- **Un processus** est forcément **créé par un autre processus**
(le système d'exploitation par exemple)

Exemples de processus

- Utilisation d'un logiciel de traitement de texte
- Compilation de code source
- Tâches système (envoi de données vers l'imprimante)

PROCESSUS

- Dans un OS moderne, plusieurs processus s'exécutent en parallèle :
 - Les processus de l'OS (gestion du réseau, gestion des utilisateurs, ...)
 - Le shell (toute l'interface graphique → plusieurs processus).
 - L'IDE VSCode avec lequel je tape ce cours.
 - Le navigateur Chrome qui me permet de visualiser ce cours.

```
$ top -stats command,pid,ppid,cpu,pstate
```

COMMAND	PID	PPID	%CPU	STATE
com.docker.hyper	674	667	34.9	sleeping
launchd	1	0	8.5	sleeping
top	74562	34386	3.7	running
Terminal	34311	1	3.0	sleeping
WindowServer	169	1	2.5	sleeping
...				

RÔLE DE L'OS

- **Création et suppression de processus**
 - Programme → processus
 - **Munir** le programme des **informations** nécessaires pour son **exécution**
- **Suspension et reprise**
 - Multiprogrammation → **interrompre et reprendre** les processus
 - **Gestion de la mémoire** où sont stockées les processus interrompus
- **Communication et synchronisation**
 - Partage de données entre plusieurs processus
 - **Consistance** de l'état de la mémoire

RÔLE DE L'OS

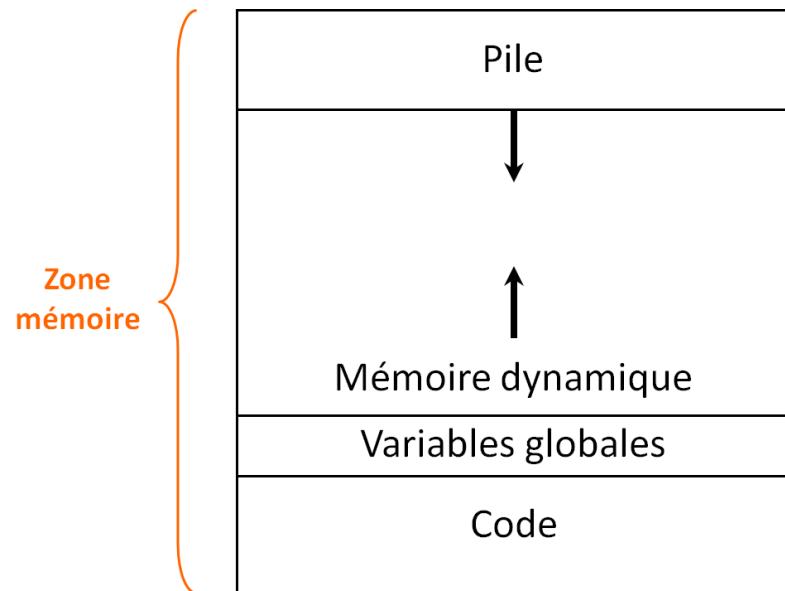
- **Création et suppression de processus**
 - Programme → processus
 - Munir le programme des **informations** nécessaires pour son exécution
- **Suspension et reprise**
 - Multiprogrammation → **interrompre et reprendre** les processus
 - **Gestion de la mémoire** où sont stockées les processus interrompus
- **Communication et synchronisation**
 - Partage de données entre plusieurs processus
 - **Consistance** de l'état de la mémoire

CRÉATION DE PROCESSUS

Rappel : un processus est forcément créé par un autre processus

- **Sous UNIX** → 2 appels système
 - **fork** pour créer un processus à partir du processus courant
 - ➡ le processus courant est **duplicé**
 - **exec** pour **remplacer** le processus courant par un autre processus
- **Sous WINDOWS**
 - **createprocess** pour créer un processus (**cf. exec Unix**)
 - ➡ le processus courant est **conservé**

L'ESPACE MÉMOIRE D'UN PROCESSUS



- **Code exécutable** en lecture seule
(taille connue)
- **Variables/Constantes globales**
(taille connue)
- **Pile** pour gérer les contextes et les variables temporaires (taille inconnue)
- **Le TAS ou la Zone d'allocation dynamique de mémoire** (taille inconnue)

BLOC DE CONTRÔLE DE PROCESSUS

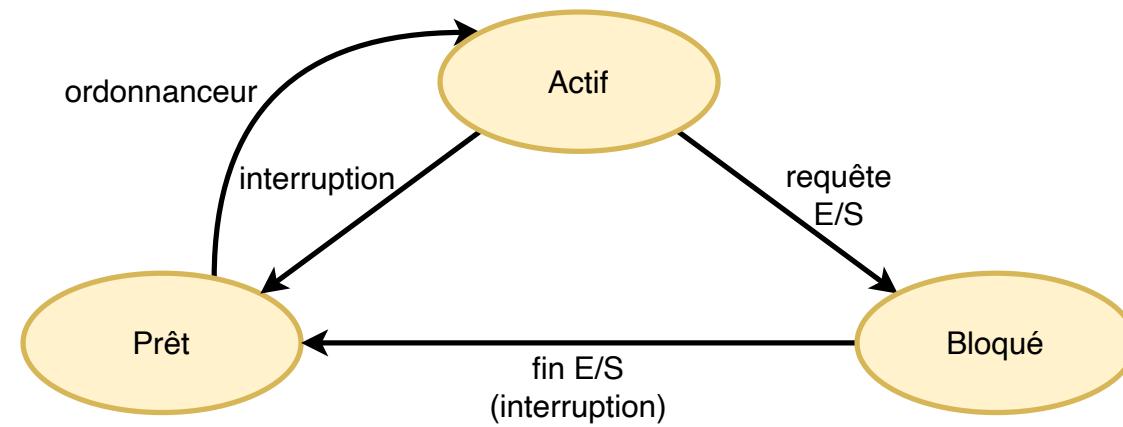
Process Control Block - PCB

- **Structure de données** contenant les informations relatives à un processus utilisée par l'OS pour la **gestion des processus**.

RÔLE DE L'OS

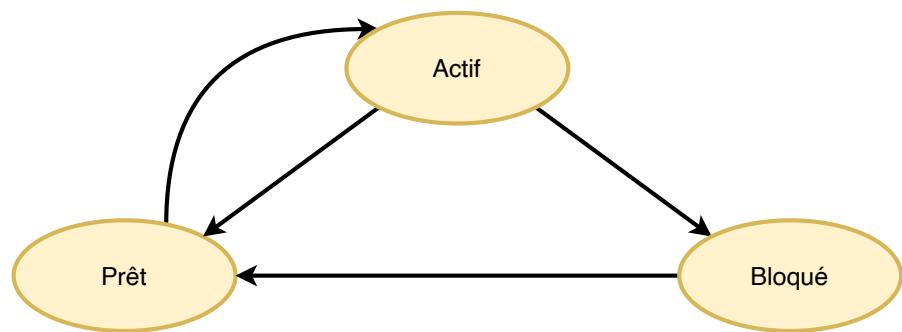
- **Création et suppression de processus**
 - Programme → processus
 - **Munir** le programme des **informations** nécessaires pour son **exécution**
- **Suspension et reprise**
 - Multiprogrammation → **interrompre et reprendre** les processus
 - **Gestion de la mémoire** où sont stockées les processus interrompus
- **Communication et synchronisation**
 - Partage de données entre plusieurs processus
 - **Consistance** de l'état de la mémoire

CYCLE DE VIE DU PROCESSUS



[0-1] processus en exécution, [0-n] processus prêts, [0-n] processus en attente

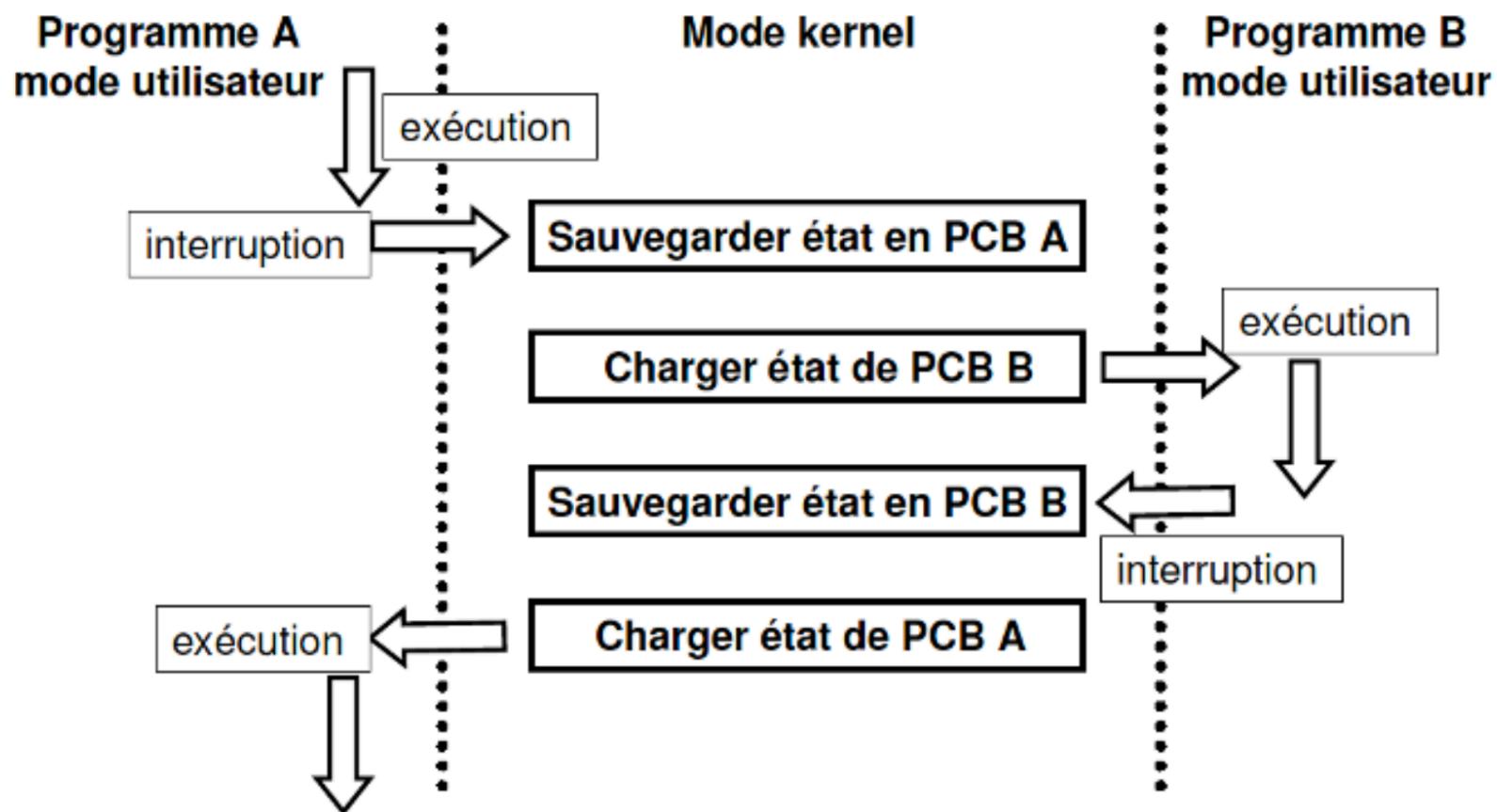
SUSPENSION DE L'EXÉCUTION



- Le processus en exécution laisse la main si:
 - ⇒ son quantum a expiré → **Prêt**
 - ⇒ crée un processus fils → **Prêt**
 - ⇒ fait une demande d'E/S → **Bloqué**
 - ⇒ exécute `wait` → **Bloqué**

COMMUTATION DE PROCESSUS

Changements de contexte



RÔLE DE L'OS

- **Création et suppression de processus**
 - Programme → processus
 - **Munir** le programme des **informations** nécessaires pour son **exécution**
- **Suspension et reprise**
 - Multiprogrammation → **interrompre et reprendre** les processus
 - **Gestion de la mémoire** où sont stockées les processus interrompus
- **Communication et synchronisation**
 - Partage de données entre plusieurs processus
 - **Consistance** de l'état de la mémoire

ACTIONS DE L'OS

- **Mémoire**: chaque processus a son propre espace mémoire
 - ➡ pas de problème de consistance mémoire/processeur
- **Verrous**: un processus peut verrouiller l'accès à une ressource
 - ➡ file d'attente pour l'accès à la ressource
- Outils et Algorithmes de **synchronisation**
 - ➡ voir partie Synchronisation des processus

NOTION DE THREAD

- Un **thread** est l'**unité d'exécution de base d'un processus**, décrite par son **point d'exécution** et son **état interne** (registres, pile, ...).
 - les threads partagent le **même code et les mêmes données**
 - chaque thread a **sa propre pile**
 - **un processus** peut avoir **plusieurs threads**
- Un **thread** partage l'espace mémoire du processus qui l'a créé.
- Un **thread** est également appelé **processus léger**.

NOTION DE THREAD

UTILISATION DES THREADS

- Le **Thread** permet la gestion de **plusieurs traitements en parallèle** dans le même processus.
 - ➡ passage de ressources entre threads facilité.
 - ➡ les variables sont dans le contexte du même processus.
- **Performances améliorées** par rapport aux processus :
 - création plus rapide;
 - changement de contexte plus rapide;
 - partage du code → gain de place en mémoire;
 - réactivité → le processus s'exécute pendant qu'un thread est en attente.

MULTIPROGRAMMATION ET TEMPS PARTAGÉ

- Les processus sont répartis sur les ressources :
 - Plusieurs processus peuvent vouloir la même ressource en même temps
 - File d'attente de PCB
 - Choisir un processus parmi tous les processus dans la file d'attente
- Exemple
 - le processeur est une ressource hautement critique.
 - l'OS est en charge de sa répartition entre les processus
 - ➡ l'ordonnancement (scheduling)

ORDONNANCEMENT

- On ne s'intéresse pas à la durée totale du processus ...
mais au **temps** pendant lequel il va **garder le processeur** :
 - ➡ jusqu'à ce qu'il **termine**
 - ➡ jusqu'à ce qu'il **fasse une E/S**
 - ➡ jusqu'à ce que l'**OS** décide que **ce n'est plus son tour**
- **Le remplacement d'un processus** en exécution a un coût
(commutation de contexte)
 - exécution de la **routine d'ordonnancement**
 - sauvegarde du **contexte** (registres + **PC**)
 - chargement d'un nouveau **contexte**

OBJECTIFS POSSIBLES DE L'ORDONNANCEMENT

- être **équitable (fairness)** vis-à-vis des processus ;
- maximiser l'utilisation globale du processeur (**efficace**) ;
- avoir un comportement le plus **prévisible** possible ;
- permettre un maximum d'utilisateurs interactifs (**réactif**) ;
- **minimiser le surcoût (overhead)** lier à la parallélisation ;
- assurer une **utilisation maximale** des ressources ;
- gérer convenablement les **priorités**.

Objectif → choisir un **algorithme d'ordonnancement**
qui **minimise** ou **maximise un critère**.

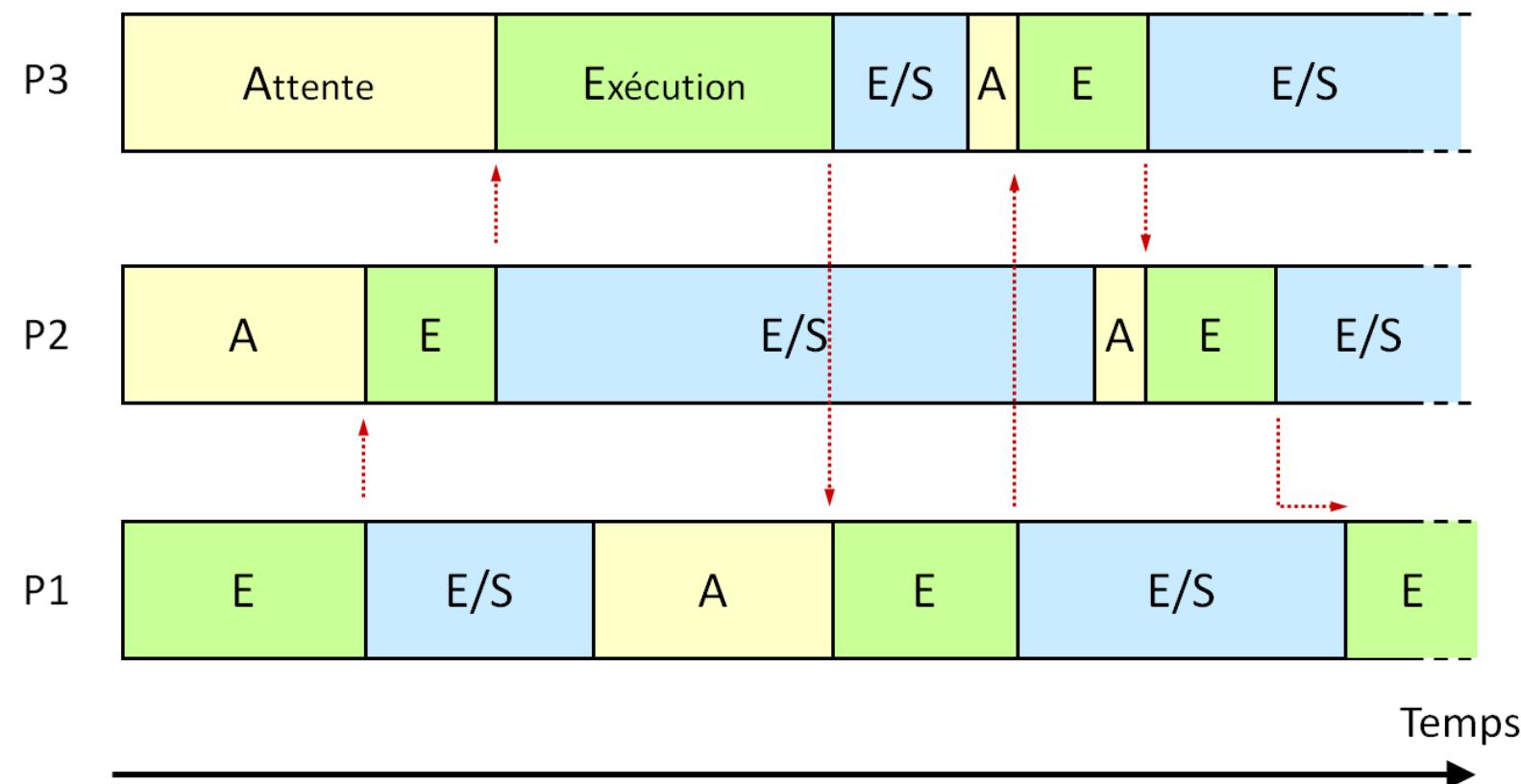
ORDONNANCEMENT

PRÉEMPTIF vs NON PRÉEMPTIF

- Ordonnancement **non préemptif** → après avoir donné le contrôle à un processus, l'OS **ne peut pas l'interrompre**
 - sauf si en attente d'une ressource
- Ordonnancement **préemptif (avec réquisition)** → l'**OS** peut interrompre un processus si :
 - ➡ le **quantum** (le temps d'utilisation maximum consécutif) est atteint
 - ➡ **un processus plus prioritaire** demande d'utiliser le processeur
- **L'ordonnancement préemptif** est indispensable pour gérer des **systèmes temps réel** ou des **systèmes interactifs**.

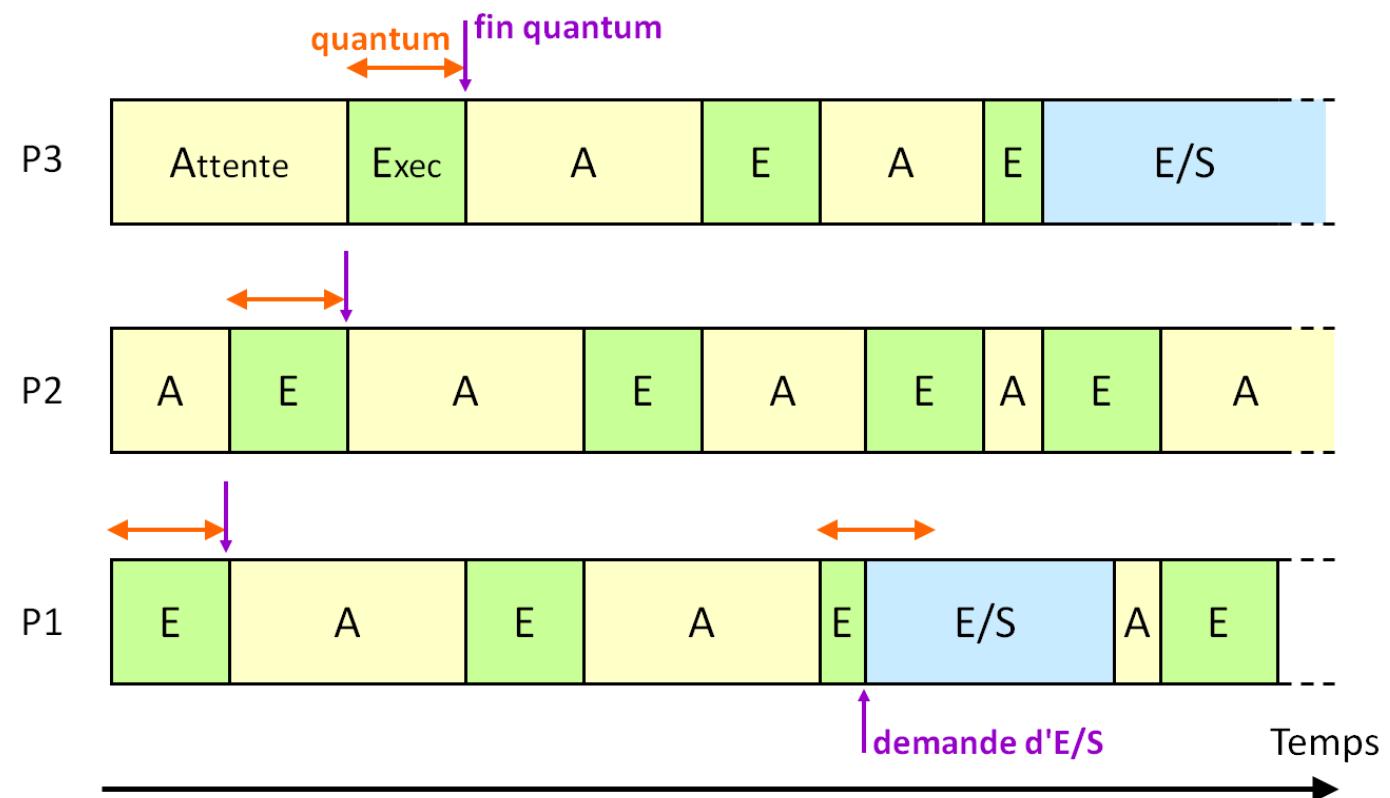
ORDONNANCEMENT

NON PRÉEMPTIF



ORDONNANCEMENT

PRÉEMPTIF



DE NOMBREUSES STRATÉGIES D'ORDONNANCEMENT PRÉEMPTIF

- FIFO sans préemption
- Plus court d'abord
- Round-Robin (FIFO préemptif)
- Files de priorités
- Files de priorités dynamiques

EXAMPLE

SOLARIS

OS DES MACHINES SUN ENTRE 1993 ET 2000

Principe

- Quantum de temps selon priorité (0 = priorité max)
- Priorité modifiée à la fin du quantum ou après une E/S
 - ➡ prioritaire → grand quantum
 - ➡ quantum consommé → priorité augmentée
 - ➡ E/S → priorité diminuée

EXAMPLE

WINDOWS XP ET APRÈS

Principe

- Priorité + Round Robin
 - ➡ gérée au niveau des threads uniquement
 - ➡ 32 niveaux de priorité
 - ➡ ordonnancement préemptif par niveau de priorité
- Priorité dynamique
 - ➡ baissée à la fin du quantum
 - ➡ remontée après chaque E/S → interface graphique plus réactive!

EXAMPLE

LINUX, MACOS X

Principe

- 2 algorithmes:
 1. Tâches **temps réel** → préemptif selon priorité, **FIFO** ou **RR** par priorité
 2. Autres tâches → temps partagé équitable
- Système de crédits
 - ⇒ chaque processus dispose d'un **crédit** = sa priorité
 - ⇒ le processus le plus riche l'emporte (préemptif)
 - ⇒ perte de 1 crédit à la fin du quantum
 - ⇒ si aucun processus **prêt** n'a de crédit, **tous** les processus sont re-crédités
→ $credit' = credit/2 + credit_{init}$

INTER PROCESS COMMUNICATION - IPC

- L'**OS** garantie l'**indépendance des processus**
 - par l'**ordonnanceur CPU**
 - par la **gestion mémoire** que l'on verra plus tard
- Un processus peut **communiquer** avec un autre processus ou avec des périphériques (fichiers, imprimantes, réseaux, ...).
- Il est alors nécessaire de mettre en oeuvre un mécanisme de **communication inter-processus**.

PRINCIPALES MÉTHODES DE COMMUNICATIONS

Méthode	Description
Signal	Un message système est envoyé d'un processus à un autre.
Pipe	Un canal unidirectionnel ; les données émises sont accumulées dans une mémoire tampon (FIFO).
File	Lecture/Écriture dans un fichier.
Socket	Un flux de données envoyé à travers une interface réseau à un autre processus.
Mémoire Partagée	Espace de mémoire alloué à plusieurs processus.
Moniteur/Sémaphore	Une structure de synchronisation pour les processus travaillant sur des ressources partagées.

PROBLÈME DE LA CONCURRENCE

EXEMPLE

Soit la gestion d'un compte bancaire

- Une variable partagée `balance`
- Une fonction `add(1)` (`balance = balance + 1`)
- Une fonction `sub(1)` (`balance = balance - 1`)
- Le montant initial du compte est de `9€`

```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

PROBLÈME DE LA CONCURRENCE

EXEMPLE

```
1 ;add(1) or balance = balance + 1
2 mov eax, balance
3 add eax, 1
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1
2 mov eax, balance
3 sub eax, 1
4 mov balance, eax
```

On lance 2 threads en parallèle

- La première thread exécute 10 000 000 fois `add(1)`
 - La deuxième thread soustrait 10 000 000 fois `sub(1)`
- ➡ **Résultat attendu** → `balance = 9€`
- ✗ **Résultat obtenu** → `balance = -98599€`

PROBLÈME DE LA CONCURRENCE

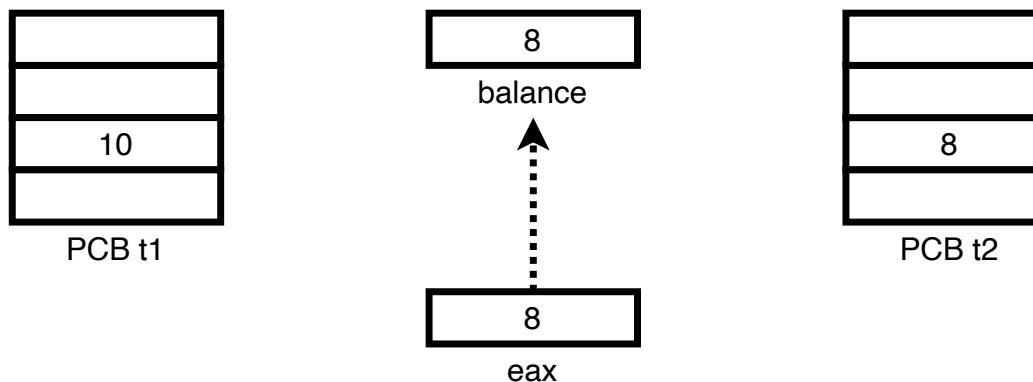
```
1 ;add(1) or balance = balance + 1  
2 mov eax, balance  
3 add eax, 1  
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1  
2 mov eax, balance  
3 sub eax, 1  
4 mov balance, eax
```



Comment expliquer ces erreurs de calcul ?

- ⇒ les entrelacements se font au niveau du code binaire
- ⇒ couper **entre chaque instruction assembleur**



PROBLÈME DE LA CONCURRENCE

CONCLUSION

```
1 ;add(1) or balance = balance + 1  
2 mov eax, balance  
3 add eax, 1  
4 mov balance, eax
```

```
1 ;sub(1) or balance = balance - 1  
2 mov eax, balance  
3 sub eax, 1  
4 mov balance, eax
```

Après une itération → **balance = 8**
alors qu'on s'attend à **balance = 9**

PROBLÈME DE LA CONCURRENCE

CONCLUSION

Situation de compétition

- ➡ **erreur** dépendant de l'enchaînement temporel d'événements impliquant une ressource partagée
- ➡ non déterministe
- ✗ difficile à détecter (**tests**)
- ✗ difficile à corriger (**debug**)

SECTION CRITIQUE

Lorsqu'il y a des **variables partagées**, il existe des **portions de code** qu'on ne veut pas pouvoir interrompre.

- ➡ des **zones du code** qui manipulent des **ressources partagées**
- ➡ ces zones sont appelées **sections critiques**

EXCLUSION MUTUELLE

- Lorsqu'on déclare une **section critique**, on doit garantir qu'**au plus un seul** processus/thread est dans la section critique.
 - ➡ besoin de gérer l'**exclusion mutuelle** des **sections critiques**.
 - ➡ **une seule section critique** peut être exécutée à la fois.
- Pour résoudre ce problème il faut un **système de verrou**.

LES PROPRIÉTÉS À RESPECTER

1. **Exclusion mutuelle** → si une thread effectue sa section critique, alors **aucune autre thread** ne peut entrer en section critique.
2. **Déroulement** → une thread, qui souhaite entrer en section critique, **ne peut pas décider** qui doit rentrer en section critique.
3. **Vivacité** → une thread, qui souhaite entrer en section critique, **y rentre en temps borné**.

Il faut définir des **mécanismes** qui garantissent ces trois propriétés

SYSTÈMES DE VERROU

1. Processeur non-parallélisé (mono-Processeur, mono-Thread)

- masquer les interruptions pendant la section critique
- empêche la préemption

2. Processeur parallélisé

- coopération Hardware/Logiciel (quelques instructions atomiques)
- des mécanismes de haut niveau de l'OS (moniteurs, sémaphores, ...)

MONITEUR

Un moniteur est un module constitué de:

- objets inaccessibles de l'extérieur
- fonctions manipulant l'objet en exclusion mutuelle

MONITEURS EN JAVA

- Dans la **JVM** de **Java**, on peut déclarer que des **blocs** de code sont en **exclusion mutuelle**.
 - utilisation du mot clé **synchronized** → **un verrou (lock)** sur un objet
 - une seule thread dans les blocs **synchronized** sur le même objet
- Une thread, qui possède un verrou, peut exécuter n'importe quelle méthode, verrouillée ou non (**verrou récursif**).
- Une thread peut **verrouiller plusieurs objets**, ceci peut provoquer une situation **d'interblocage**.
- Tout bloc **non synchronized** (**non verrouillé**) peut être appelé par n'importe qui n'importe quand.

MONITEURS EN JAVA

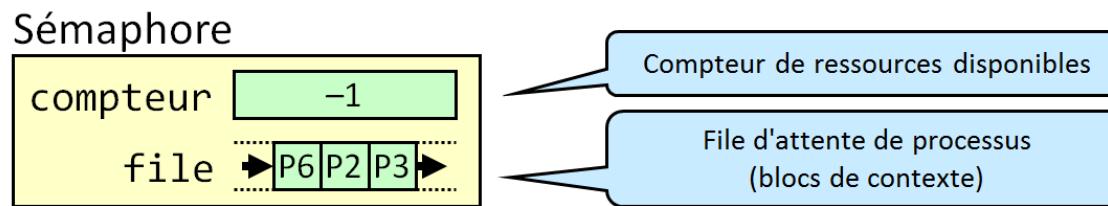
EXEMPLE

```
1 public class Account {  
2     private int value;  
3  
4     public Account(int i) {  
5         this.value = i;  
6     }  
7  
8     synchronized public void add(int v){  
9         this.value = this.value + v;  
10    }  
11  
12    synchronized public void sub(int v){  
13        this.value = this.value - v;  
14    }  
15 }
```

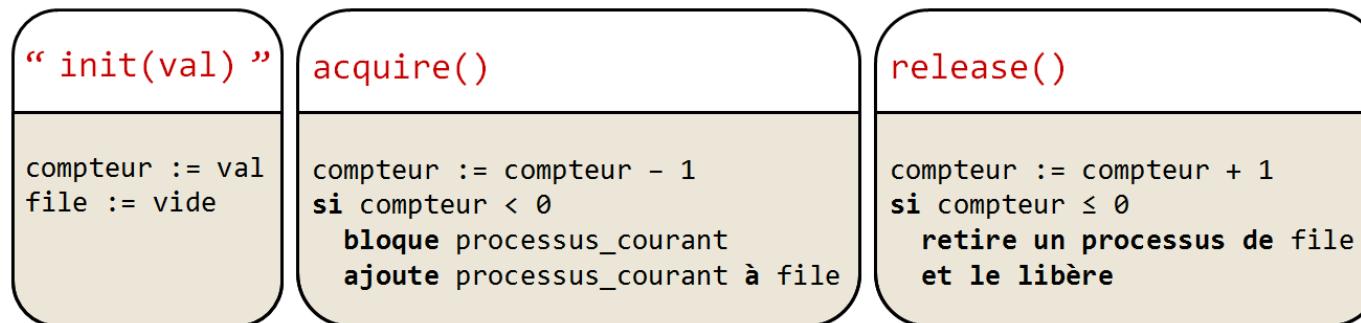
SÉMAPHORE - DIJKSTRA - 1962

- Un sémaphore définit un objet partagé
 - qu'on peut acquérir;
 - qui met en attente ceux qui le demandent;
 - qui donne la main dans l'ordre des demandes.
- Toutes les threads en concurrence sur une ressource partagent un même sémaphore
 - on acquiert le sémaphore avant d'entrer en SC;
 - on relâche le sémaphore en sortant de la SC.

IMPLÉMENTATION DU SÉMAPHORE



3 primitives (opérations **non interruptibles** = atomiques) :



initialisation

prend une ressource, attente si nécessaire

libère/produit une ressource

UTILISATION DU SÉMAPHORE

```
1 Semaphore s = new Semaphore();  
2 // ... code non-critique ...  
3 s.acquire();  
4 // ... code critique ...  
5 s.release();  
6 // ... code non-critique ...
```

Les propriétés d'un sémaphore

- **Vivacité** : on veut passer la main dans le bon ordre
 - ➡ utilisation d'une **file d'attente** des **PCB**
- Les fonctions **acquire** et **release** doivent être **atomiques** !
 - ➡ elles sont elles-même des **SC** pour le sémaphore ...

LES SÉMAPHORES

EXEMPLE

```
1 public class Account {  
2     private int balance;  
3     private Semaphore mutex = new Semaphore(1);  
4  
5     public Account(int balance) {  
6         this.balance = balance;  
7     }  
8  
9     public void add(int v){  
10        this.mutex.acquire();  
11        this.balance = this.balance + v;  
12        this.mutex.release();  
13    }  
14  
15    public void sub(int v){  
16        this.mutex.acquire();  
17        this.balance = this.balance - v;  
18        this.mutex.release();  
19    }  
20 }
```

GESTION DE LA MÉMOIRE

⌚ Fonctionnalités principales d'un OS

LA MÉMOIRE POUR QUI ET POUR QUOI ?

Pour le système d'exploitation

- Au lancement d'une machine, l'**OS** est le **premier programme** chargé en mémoire
- L'**OS** aussi a besoin d'espace mémoire
 - ➡ le **code** de son **Noyau**
 - ➡ la table des **interruptions**
 - ➡ la table des **processus**
 - ➡ des structures de données (**PCBs** et autres)
 - ➡ ...

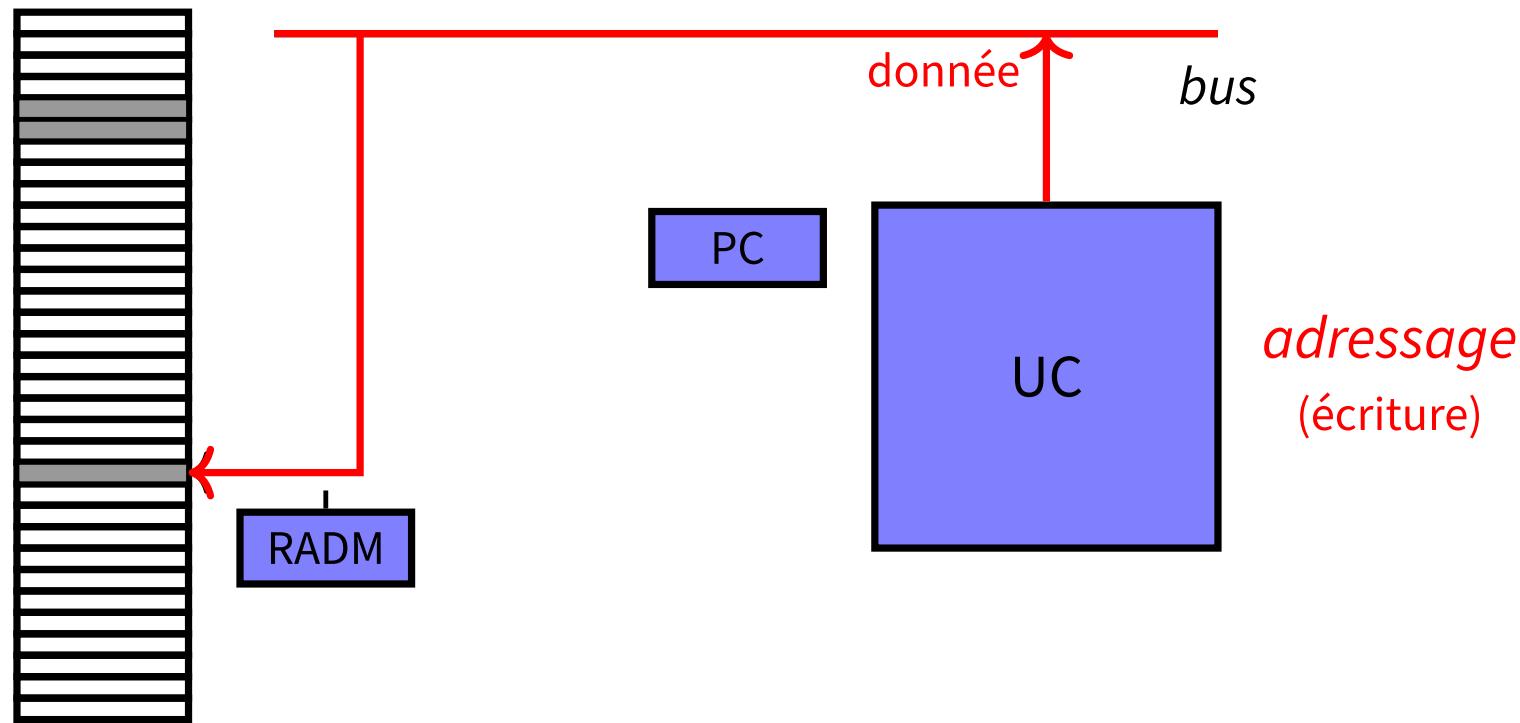
LA MÉMOIRE POUR QUI ET POUR QUOI ?

Pour les processus

- A la création d'un processus, l'OS crée un PCB et alloue de la mémoire pour le processus.
- Pour des raisons de sécurité, chaque processus doit utiliser une zone mémoire distincte (un espace d'adresses).
 - ➡ quel mécanisme d'allocation de cet espace ?
 - ➡ comment assurer la protection de cette zone ?
 - ➡ comment assurer la transparence de cet espace ?

ESPACE DE STOCKAGE

- Ensemble ordonné de **cases** indexée par leur **adresse** (**numéro de la case**) et contenant:
 - des **instructions** → registre **PC** dans le processeur
 - des **données** → registre **RADM** dans le processeur



D'OU VIENNENT LES PROGRAMMES?

Le programme (code + données) est chargé depuis le disque vers la mémoire ... il est placé à un endroit donné dans la mémoire

Question ?

quelles sont les adresses des variables en mémoire ?

PROGRAMME vs PROCESSUS

Adresses **symboliques** vs Adresses **mémoires**

```
1 int a = 3;  
2 a = a + 2;
```

```
1 @a: memval 3  
2     mov eax, a  
3     mov ebx, 2  
4     add ecx, eax, ebx  
5     mov a, ecx
```

```
1 2B50: mov eax, 2B1E  
2 2B52: mov ebx, #0002  
3 2B54: add ecx, eax, ebx  
4 2B55: mov 2B1E, ecx  
5 ...  
6 2B1E: 0003
```

Édition de liens

Lors de la création de processus, l'**OS** instancie le programme.

➡ transformer **les noms** des variables en **adresses**.

MÉTHODES DE LIAISON D'ADRESSES

- **À la compilation** → on connaît les adresses des instructions et de toutes les données
 - ⇒ adresses utilisées à l'intérieur d'un programme
 - ⇒ adresses relatives au début du programme
- **Au chargement** → la taille du processus est fixée
 - ⇒ **adresses virtuelles (logiques)**: adresses utilisées dans le programme
 - ⇒ **adresses physiques**: adresses utilisées dans la **RAM**
 - ⇒ nécessite un **composant de translation** dans l'**UC**
(Memory Management Unit - MMU)
- **À l'exécution** → le processus a besoin de plus de place
 - ⇒ déplacer le processus + réédition de lien

RÉSOLUTION D'ADRESSE

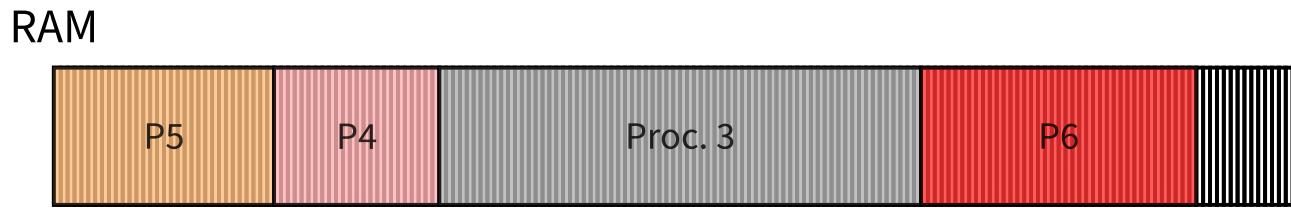
MEMORY MANAGEMENT UNIT - MMU

STRATÉGIES D'ALLOCATION DE MÉMOIRE

- Il faut choisir une stratégie pour **allouer et libérer la mémoire** en fonction des besoins des processus.
- Deux stratégies possibles:
 1. **Allocation contiguë de cases mémoire (par partition)**
 2. **Allocation non contiguë (par pagination)**

ALLOCATION PAR PARTITION

Les processus constituent **un seul bloc** non décomposable



- ✗ des trous apparaissent → **fragmentation**
- ✗ les gros processus ne peuvent pas rentrer → **défragmenter**
- ➡ réduire la **fragmentation**
- ➡ limiter les opérations de **défragmentation**

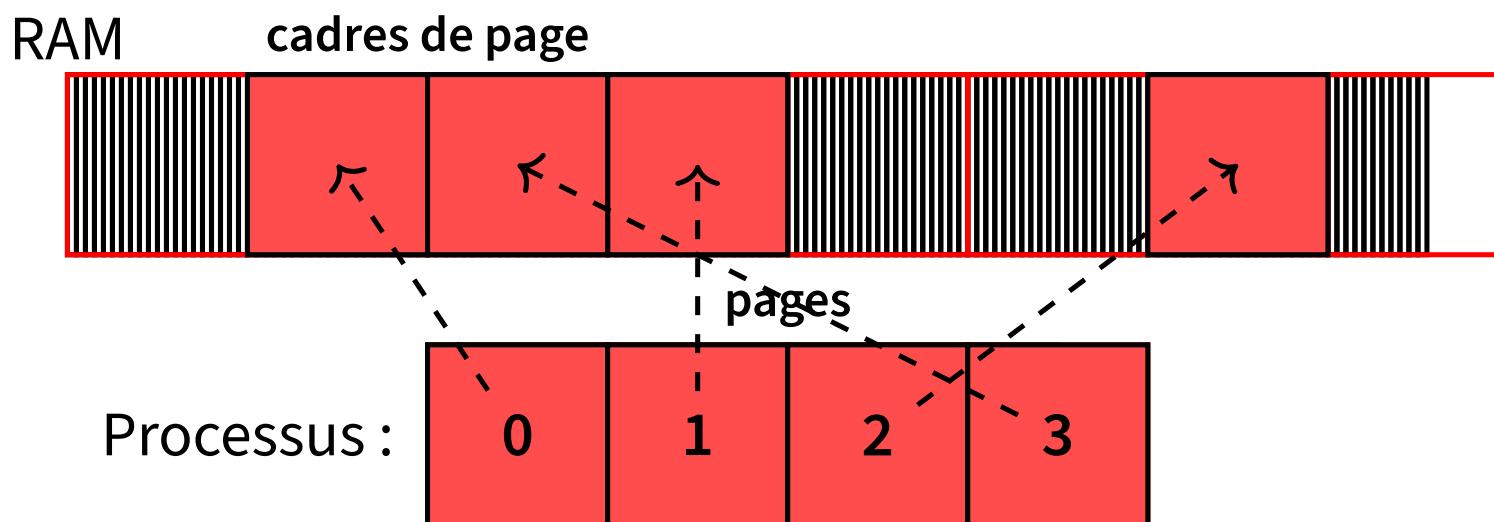
ALLOCATION PAR PARTITION

Stratégies d'allocation → choisir dans quelle **zone libre** placer un processus

- **First Fit**: premier bloc libre
- **Best Fit**: plus petit bloc libre
- **Worst Fit**: plus grand bloc libre

ALLOCATION PAR PAGINATION

- Découper la **mémoire physique** en **blocs** de taille T_c constante, appelés **cadres de pages**
- Découper l'**espace mémoire** utilisé par un processus (**espace logique**) en paquets de x **pages** de taille T_c
 - chaque **page** a la **même taille** qu'un **bloc**
- Placer les pages dans les cadres



ALLOCATION PAR PAGINATION

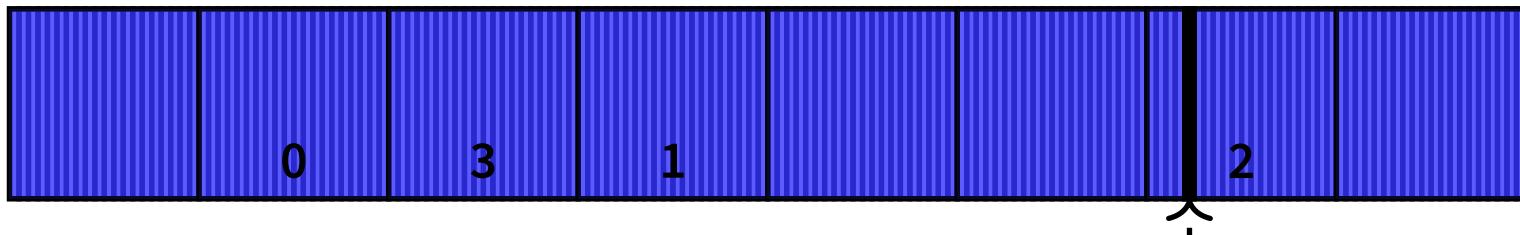
- **Allocation mémoire**
 - ⇒ un processus est dans des zones **disjointes**
 - ⇒ pas besoin de défragmenter
- **Adaptation:**
 - ⇒ besoin de plus de mémoire → **rajouter des pages**
 - ⇒ pas besoin de le ré-allouer entièrement
- **Mémoire virtuelle:** charger uniquement les pages dont le processus a besoin.

ALLOCATION PAR PAGINATION

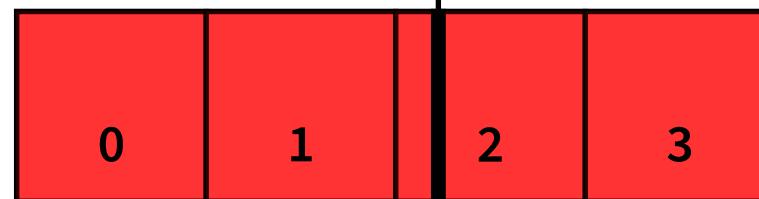
ADRESSAGE

Déterminer l'adresse **physique** à partir de l'adresse **logique**

RAM



Processus :



ALLOCATION PAR PAGINATION

ADRESSAGE

Déterminer l'adresse **physique** à partir de l'adresse **logique**

- **Adresse logique**
 - Numéro de page (n bits) + décalage (m bits)
- Chaque processus maintient une liste:
 - numéro de page → numéro de cadre
 - c'est la **table des pages**

ALLOCATION PAR PAGINATION

RÉSOLUTION D'ADRESSE

ALLOCATION PAR PAGINATION

RÉSOLUTION D'ADRESSE

- En pratique, géré au niveau matériel → la **MMU**
 - L'**OS** charge la table des pages du processus dans la **MMU**
 - Pas de calcul d'adresse au niveau de l'**OS**
- **Allocation des cadres:** ne pas allouer le même cadre à deux processus différents
 - ➡ L'**OS** doit savoir quel processus utilise quel cadre
 - ➡ **Table des cadres de page libres**

num. cadre	num. proc.	libre
0	42	1
1	37	1
2	-	0
...

GESTION DE L'ESPACE D'ADRESSAGE

Problème → Grand espace d'adressage (ex: 32 bits)

- **Trop de pages** (ex: $n = 20, m = 12$)
 - Grande table des pages → place mémoire perdue
 - 2^{20} lignes de 20 bits ≈ 2.5 Mo par processus
 - Allocation et commutation plus coûteuse en temps
- **Pages trop grosses** (ex: $n = 10, m = 22$)
 - Fragmentation = 2^{m-1} → place mémoire perdue
 - 2^{21} octets ≈ 2 Mo par processus
 - Pas gérable au niveau du MMU

PAGINATION HIÉRARCHIQUE

2 NIVEAUX (OU PLUS)

Solution → paginer la table des pages

- ➡ ne charger que les tables utiles
- ➡ réduire l'espace mémoire utilisé par le système d'adressage
- ➡ réduire la fragmentation due aux pages

PAGINATION À DEUX NIVEAUX

PAGINATION À DEUX NIVEAUX

EXEMPLE

- Adresses sur 32 bits et cadres de 4 Ko $\rightarrow m = 12$
- Pagination à 1 niveau ($n = 20, m = 12$)
 - Table des pages = 2^n lignes de n bits
 - Total = $2^{20} \times 20 \approx 2.5$ Mo par processus
- Pagination à 2 niveaux ($n_1 = 10, n_2 = 10, m = 12$)
 - Répertoire = 2^{10} lignes de 32 bits ≈ 4 Ko
 - 1 table de pages = 2^{10} lignes de 20 bits ≈ 2.5 Ko
 - Total = entre 6.5 Ko et 2.5 Mo par processus

MÉMOIRE VIRTUELLE

⌚ Fonctionnalités principales d'un OS

ÉTAT DE LA MÉMOIRE

- Nombre et taille des processus:
 - Entre 200 et 500 processus en parallèle sur un **PC**
 - Gros processus: Eclipse = 250 Mo, données d'un jeu ≥ 1 Go
 - ✖ Somme des tailles des processus \geq Capacité **RAM**
 - Portions de code inutilisées
 - Traitement d'erreur → rarement utilisé
 - Données (tableau, jeu, ...) → pas tout en même temps
 - Bibliothèque → très variable!
-
- ➡ ne charger que les pages utiles!
 - ➡ laisser le reste sur le disque

MÉMOIRE VIRTUELLE

- Extension des mécanismes de **pagination**
 - la mémoire paginée peut représenter des **espaces non présents en RAM** (ex. disque) → **mémoire virtuelle**
 - les pages sont soit en **RAM**, soit en **mémoire auxiliaire (swap)** → la **RAM** est un cache
- Chaque ligne de **la table des pages** contient:
 - un **bit de la validité** qui indique si la page est en **RAM**
 - l'adresse correspondante en **RAM** (**numéro du bloc**)
 - sinon une information pour la trouver sur **le disque**

SCHÉMA DE PRINCIPE

Chaque processus peut **adresser** plus d'espace
qu'il n'a effectivement en **mémoire physique**

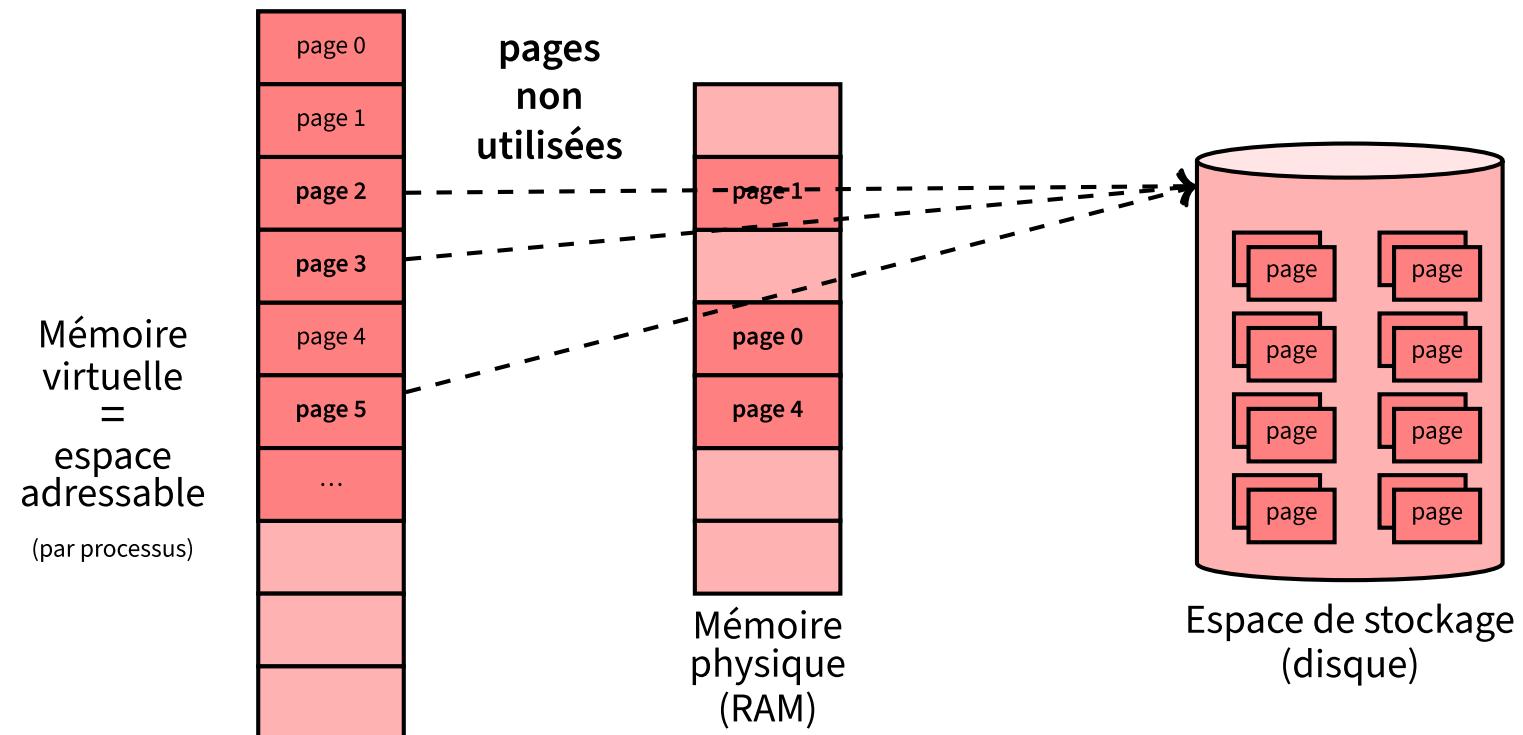


SCHÉMA DE PRINCIPE

Chaque processus peut **adresser** plus d'espace
qu'il n'a effectivement en **mémoire physique**

- Accès à une page non présente en **RAM** :
 - ➡ levée d'une **exception CPU Page-Fault**
 - ➡ processus courant bloqué et **chargement** de la page en **RAM**
- **Avantages**
 - ✓ masquer la taille de la RAM
 - ✓ possibilité de mettre plus de processus en parallèles
 - ✓ affecter plusieurs adresses virtuelles à une adresse physique
 - ✓ une pagination à la demande

PAGINATION À LA DEMANDE

REEMPLACEMENT DES PAGES

- Chaque processus dispose d'un **nombre de cadres limité**
 - ➡ selon la politique d'allocation
 - ➡ libérer un cadre lorsqu'on a besoin d'une nouvelle page

Problème

- **Temps d'exécution** proportionnel au **nombre de défaut de page**
 - ➡ réduire le nombre de défauts de page
- Quelle **stratégie** pour choisir les pages à supprimer de la RAM ?

REEMPLACEMENT DES PAGES

Algorithme de remplacement

Définir une **fonction** qui, étant donné l'état actuel (occupation des cadres par des pages), décide **quel cadre doit être libéré**.

- Algorithme **First In, First Out (FIFO)**
 - retirer les pages les plus anciennes
 - seconde chance
- Algorithmes **basés sur l'utilisation des pages**
 - la page la moins utilisée (**Least Frequently Used**)
 - une page pas récemment utilisée (**Not Recently Used**)
 - la page la moins récemment utilisée (**Least Recently Used**)

LIMITES DE LA MÉMOIRE PAGINÉE

- **Découpage des processus**
 - ✗ Taille arbitraire ($2^{nb_bits_cadre}$)
(peut couper une portion de code, un bloc de données ...)
 - ✗ Fragmentation résiduelle
- **Chargement d'une page**
 - ✗ Plein de données inutiles
 - ✗ Pas forcément tout ce dont on a besoin
- **Idée → mémoire segmentée** (non traitée dans ce cours)
 - Découper en tenant compte de la structure du processus
(code + données)

SYSTÈME DE FICHIERS

⌚ Fonctionnalités principales d'un OS

LE STOCKAGE SECONDAIRE

- Le stockage secondaire conserve des programmes et des données.
- L'OS masque la complexité et la diversité des unités de stockage (matériel, système de fichiers, ...) grâce à:
 - Une vue logique des données :
 - ⇒ fichiers: unité de stockage logique
 - ⇒ répertoires: classement arborescent
 - ⇒ volumes montés: vue globale des systèmes de fichiers
 - Une organisation physique des espaces de stockage :
 - ⇒ découpage en blocs
 - ⇒ affectation et libération de blocs
 - Un système d'entrées/sorties (non traité dans ce cours) :
 - ⇒ gestion du caches, algorithmes d'optimisation des accès,
 - ⇒ pilotes gérant les communications avec les périphériques

LA NOTION DE FICHIER

- **Fichier**
 - une collection nommée d'information accessibles via un périphérique.
- **Unité logique**
 - indépendante du support physique (périphérique)
 - abstraction des propriétés physiques
- **Type de fichier**
 - code source, données, bibliothèque, exécutable, ...
 - généralement indiqué par son **extension**
 - un type de fichier → une **structure spécifique**

FILE CONTROL BLOCK - FCB

Structure de données de l'**OS** pour stocker les informations nécessaires
à la gestion des fichiers

Nom	indépendant de l'OS, lisible
Identifiant	numérique, unique, pour l'OS
Type	extension ou en-tête de fichier
Emplacement	pointeur sur un périphérique
Taille	en octets ou en blocs
Protection	lecture, écriture, exécution ...
Date(s)	création, modification, accès ...
Utilisateur	propriétaire du fichier
...	

PROTECTION

- **Partage de fichiers**
 - rendre **accessible** à un **utilisateur B** un fichier de l'**utilisateur A**.
- **Politique de protection**
 - définir qui peut accéder à quel(s) fichier(s)
 - ➡ identifiant utilisateur → identifiant processus
 - ➡ contrôle d'accès dans le **FCB**
 - spécifier pour chaque fichier la liste des **sujets** autorisés à effectuer un **type d'accès**

TECHNIQUES DE PROTECTION

- **Liste de contrôle d'accès (ACL)**
 - utilisateur → droits
 - l'ensemble des utilisateurs doit être connu a priori
 - ✗ taille du **FCB** grossit avec le nombre d'utilisateurs
- **Mot de passe**
 - 1 mot de passe par fichier × type d'accès (**r**, **w**, ...)
 - ✗ pas très pratique → peu utilisé
- **Classes d'utilisateurs**
 - Exemple: Propriétaires **vs** Autres

TECHNIQUES DE PROTECTION

- **Notion de groupe**
 - Ensemble de groupes définis a priori
 - Ex: admin, dev-disque, user-disque, dev-ram, user-ram
 - **FCB**: 1 utilisateur + 1 groupe (propriétaires)
 - Ex: toto.c, u=batman, g=dev-disque
 - Utilisateur → liste de groupes
 - Ex: robin, g=[user-disque, dev-ram]
 - ➡ robin n'a pas accès à toto.c
- Dans les systèmes **POSIX** on distingue :
 - **3 modes** (lecture, écriture, exécution)
 - **3 catégories de sujets** (le propriétaire, son groupe et le reste des sujets)

TECHNIQUES DE PROTECTION

- Dans les systèmes **POSIX** on distingue :
 - **3 modes** (lecture, écriture, exécution)
 - **3 catégories de sujets** (le propriétaire, son groupe et le reste des sujets)
- **Exemple → Unix**
 - 3×3 bits par fichier

```
$ ls -l
total 248
drwx----- 6 nico prof      4096 nov.  20 12:06 private
-rw----- 1 nico prof      2356 dec.   5 15:30 notes.ods
drwxrwx--- 8 nico prof      4096 dec.   4 17:31 doc
-rw-rw-r-- 1 joe  student     36 jan.  21 16:49 programme.c
-rwxrwxr-x 1 joe  student    996 jan.  28 10:53 programme
...
```

OPÉRATIONS SUR UN FICHIER

- **Appels systèmes de base**
 - **Création:** allocation espace + entrée **répertoire**
 - **Lecture:** pointeur de lecture
 - **Écriture:** pointeur d'écriture
 - **Repositionnement:** déplacer un pointeur
 - **Suppression:** retrait de l'entrée dans le répertoire
 - **Troncature:** vider mais garder l'entrée
- **Opérations composées**
 - Ex: copie, renommage
 - ➡ effectuées à partir des appels systèmes de base

OUVERTURE DE FICHIER

✗ Problème

- Nécessité d'accéder au **FCB** à chaque opération sur le fichier
- Le **FCB** est stocké dans le répertoire du périphérique
- Très coûteux en **accès disque** (donc en temps)!

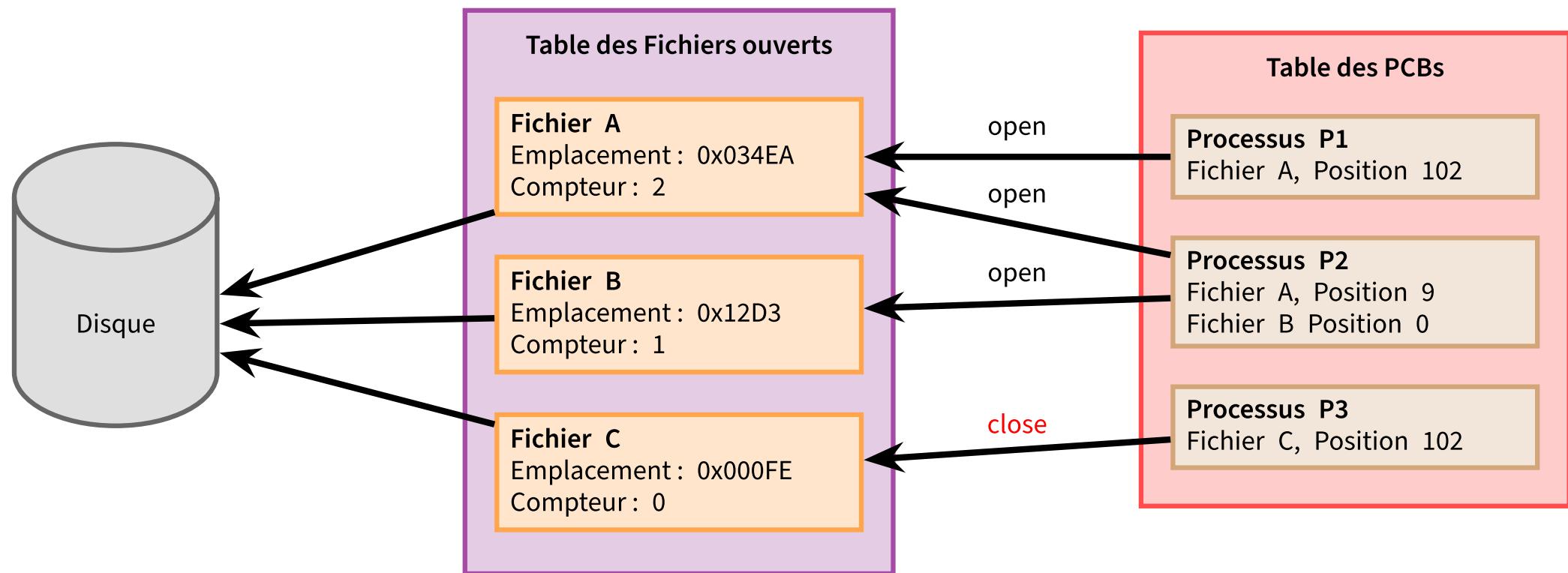
✓ Solution

- L'appel système **open** permet de **charger** le **FCB** en mémoire.
- L'**OS** impose que tout accès à un fichier soit **précédé** d'une ouverture.

TABLE DES FICHIERS OUVERTS

- La **table des fichiers ouverts** de l'**OS** contient l'ensemble des **FCB** des **fichiers ouverts**.
 - **ouverture** → chargement du **FCB** + ajout dans la table
 - **fermeture** → retrait de la table
 - ➡ les **FCB** sont chargés en **RAM**
 - ➡ pas d'impact sur le fichier!
- Gestion par l'**OS**
 - **implicite** → **open** implicite au premier accès
 - **explicite** → **exception** si le fichier n'a pas été ouvert avant
 - **une table de fichiers ouverts globale** avec compteurs
 - **une table par processus** → fermeture à la terminaison
 - ➡ possibilité d'interdire l'accès aux autres processus

TABLE DES FICHIERS OUVERTS



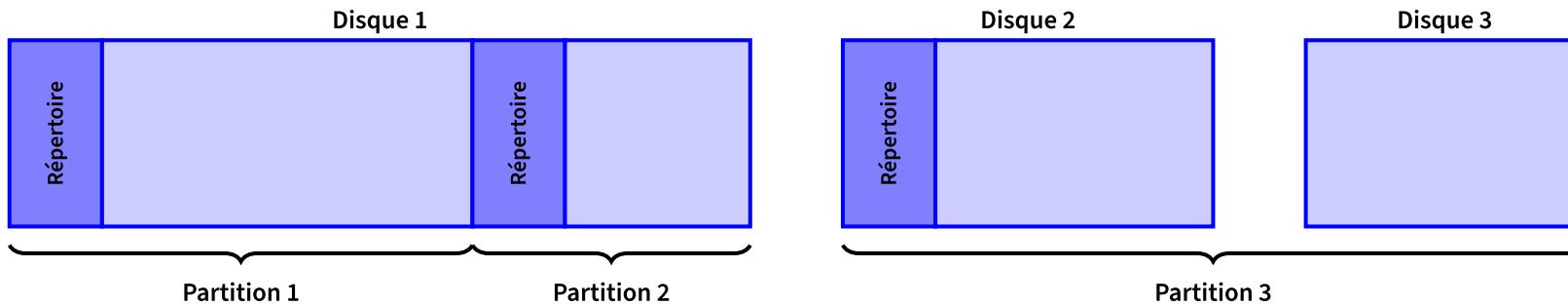
NOTION DE RÉPERTOIRE

- Le **répertoire** est la structure de **stockage des informations** des fichiers (les **FCB**) dans les **supports de stockage**.
 - ➡ entrée du répertoire = identifiant du fichier et/ou nom du fichier
 - ➡ contenu du répertoire = **FCB** des fichiers
- L'**OS** récupère les informations sur les fichiers dans **le répertoire**

STRUCTURE DES DISQUES

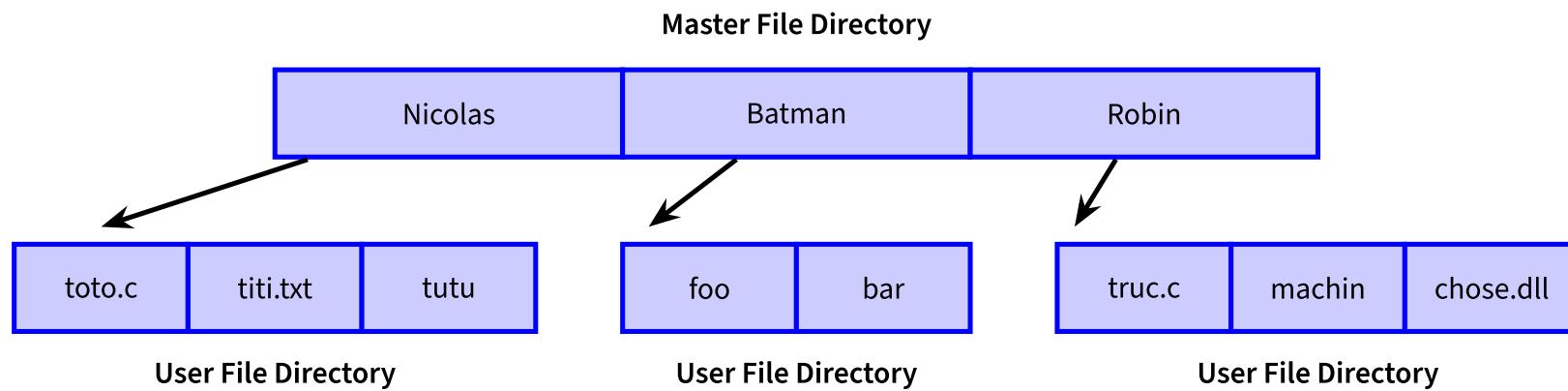
- **Disque** → Structure physique
- **Partition** → Structure logique (disque "virtuel")
 - Base : 1 disque = 1 partition
 - 1 disque = N partitions
 - 1 partition = 1 ou N disques (selon **OS**)
- **Répertoire** → Un répertoire par partition → l'ensemble des **FCB**
 - ➡ Nom/identifiant → **FCB**

STRUCTURE À 1 NIVEAU



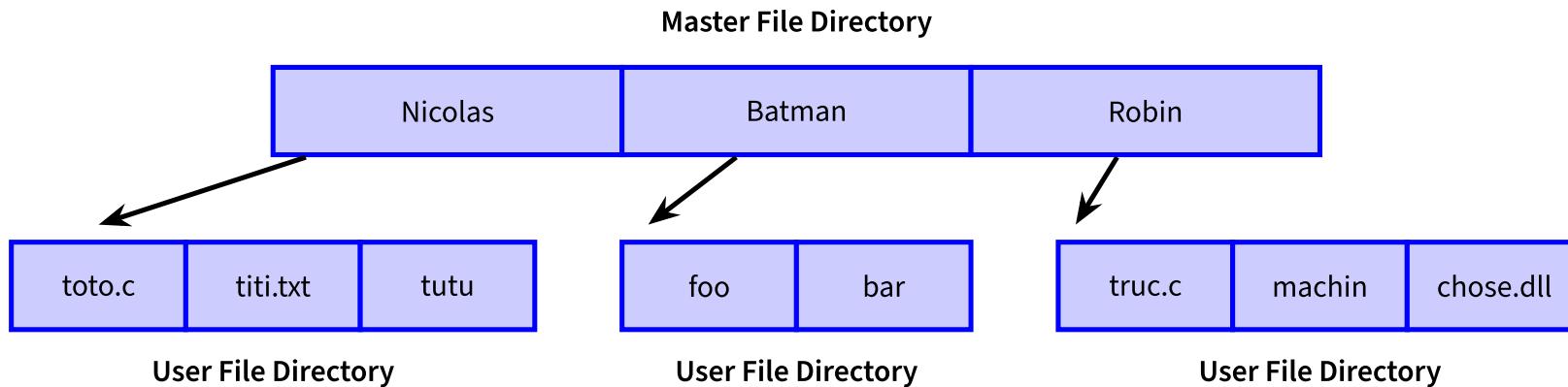
- Nom → **FCB**
 - Exemple: **MSDOS** et **Windows** → 11 octets (8 nom + 3 extension)
 - Exemple: **Unix** et **Mac** → 255 octets
- ✖ Taille du répertoire proportionnelle au nombre de fichiers
 - ➡ borner le nombre de fichiers ...
- ✖ Utilisateur: organiser les fichiers, unicité de nom, ...

STRUCTURE À 2 NIVEAUX



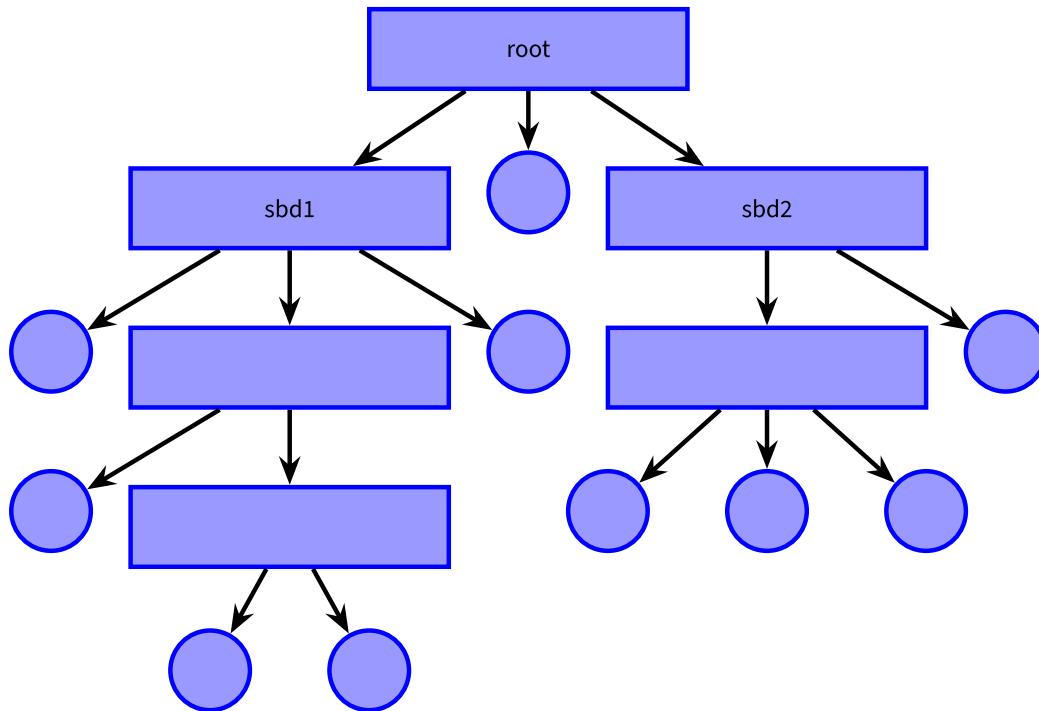
- Un répertoire par utilisateur
 - Identifiant + Nom → **FCB**
 - Répertoire des utilisateurs: **Master File Directory (MFD)**
 - ➡ Identifiant → **User File Directory**
 - Répertoire par utilisateur: **User File Directory (UFD)**
 - ➡ Nom → **FCB**

STRUCTURE À 2 NIVEAUX



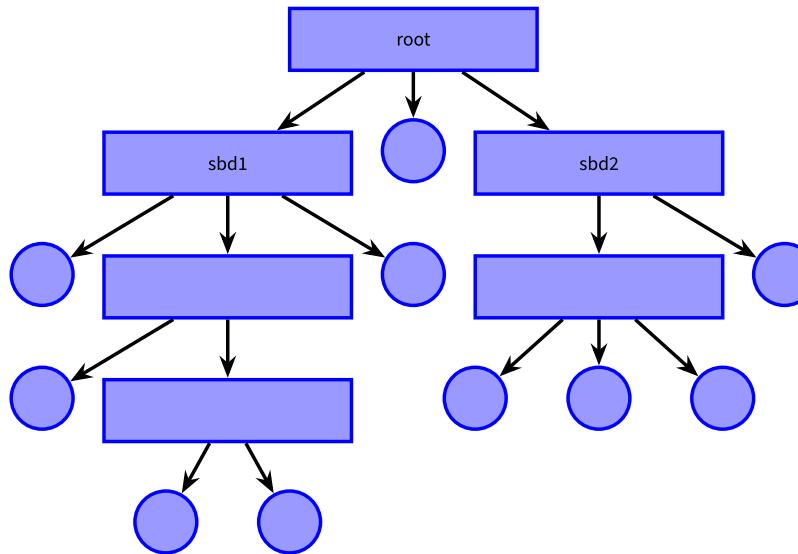
- ✓ Pas beaucoup plus coûteux en taille
- ✓ Utilisateur: organiser les fichiers, unicité de nom, . . .
- ✗ Taille des répertoires proportionnelle au nombre de fichiers
- ✗ Partage de fichiers

STRUCTURE ARBORESCENTE



- Généralisation de la structure à 2 niveaux:
 - ➡ Répertoire racine → **Master File Directory (MFD)**
 - ➡ **Sous-répertoires**, pouvant à leur tour jouer le rôle de **MFD**

STRUCTURE ARBORESCENTE

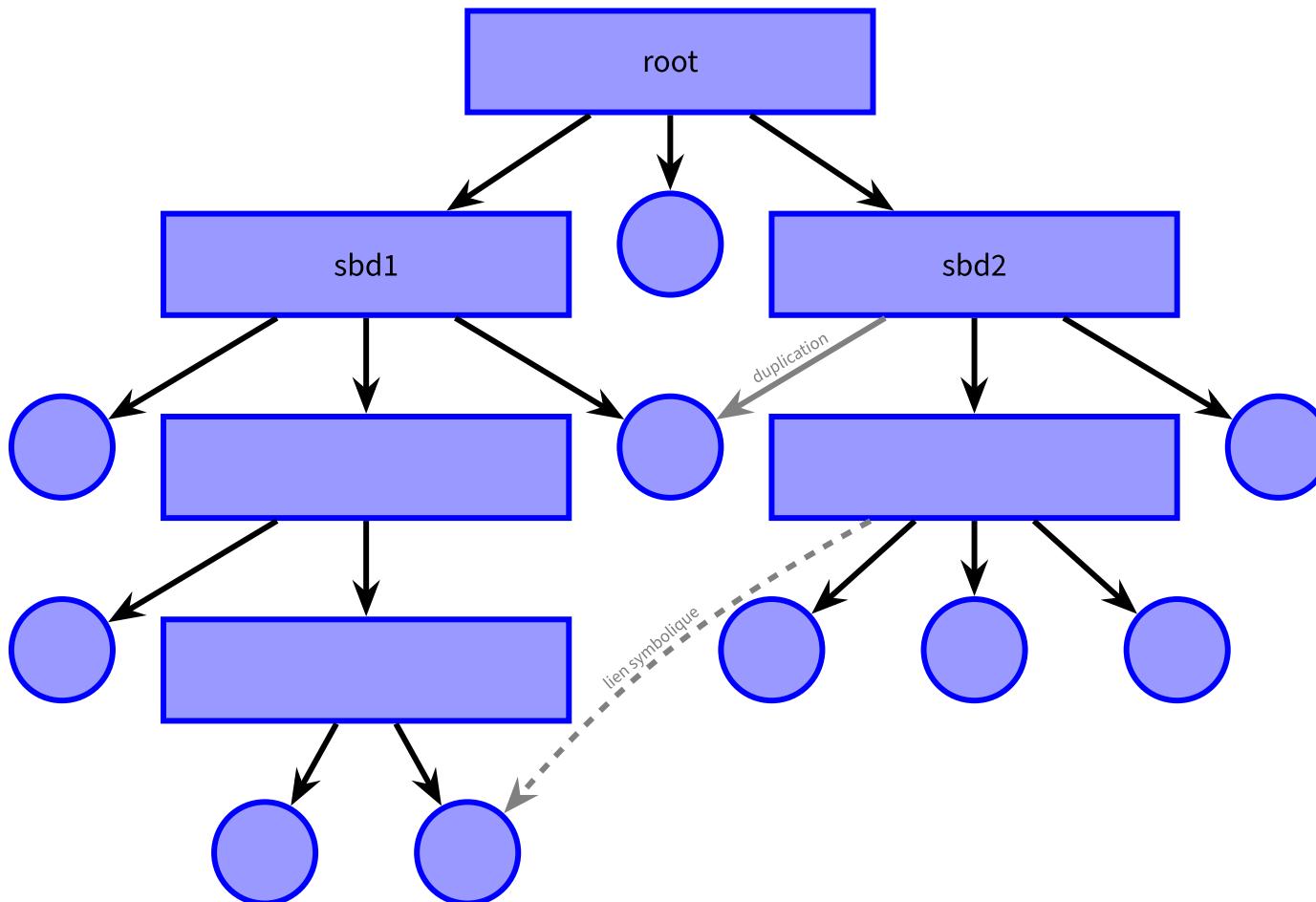


- **Fichiers**
 - Bit "répertoire" dans le **FCB**
 - Nom unique = chemin depuis la racine (chemin **absolu**)
- **OS** → Répertoire courant (par processus)
 - ➡ Recherche à partir du répertoire courant (chemin **relatif**)
 - ➡ Recherche par défaut (**PATH**)

STRUCTURE EN GRAPHE

- Généralisation de l'arbre avec des **liens**
 - ➡ Graphe **acyclique**
- **Liens:** référencer un fichier décrit dans un autre répertoire
 - ➡ bit "**lien**" dans le répertoire + chemin absolu
- **Extension:** duplication
 - ➡ **FCB** recopié → copie et original indiscernables
 - ➡ Compteur de liens (pour libérer l'espace sur le support physique)

STRUCTURE EN GRAPHE



TECHNIQUES DE PROTECTION

- Dans les systèmes **POSIX** on distingue :
 - **3 modes** (lecture, écriture, exécution)
 - **3 catégories de sujets** (le propriétaire, son groupe et le reste des sujets)

```
$ ls -l
total 248
drwx----- 6 nico prof      4096 nov.  20 12:06 private
-rw----- 1 nico prof      2356 dec.   5 15:30 notes.ods
drwxrwx--- 8 nico prof      4096 dec.   4 17:31 doc
-rw-rw-r-- 1 joe  student    36 jan.  21 16:49 programme.c
-rwxrwxr-x 1 joe  student    996 jan.  28 10:53 programme
...
```

Mode	Lecture (r)	Écriture (w)	Exécution (x)
Fichier	mode lecture	mode écriture	exécution du fichier
Répertoire	lister le contenu	créer, renommer et supprimer un fichier	accéder au répertoire et à son contenu

NOTION DE SYSTÈME DE FICHIER

- Comment stocker les informations (données et code) sur le disque
 - Comment les **organiser** ?
 - Comment y **accéder** ?
- Différence avec la **RAM**
 - Grande quantité de données
 - Accès lent (rapport 10^3 à 10^6)
- ➡ Définir une **norme** de gestion
 - Organisation de l'ensemble des **données** et des **périphériques**
 - Exemple: **Linux** → chaque périphérique est représenté par un fichier

LA VUE LOGIQUE

Système de fichiers (File System - FS)

- Un **système de fichiers** est un ensemble de **structures de données et de fonctions** qui permettent à un **OS** de **manipuler des fichiers**.

Unité logique

- Du point de vue de l'**OS**, le **FS** doit rendre des services qui ne **dépendent pas de son implémentation** (indépendant du support physique).
- Il doit répondre à des problèmes comme :
 - la diversités des supports de stockage ;
 - la sécurisation des données.

LA VUE PHYSIQUE

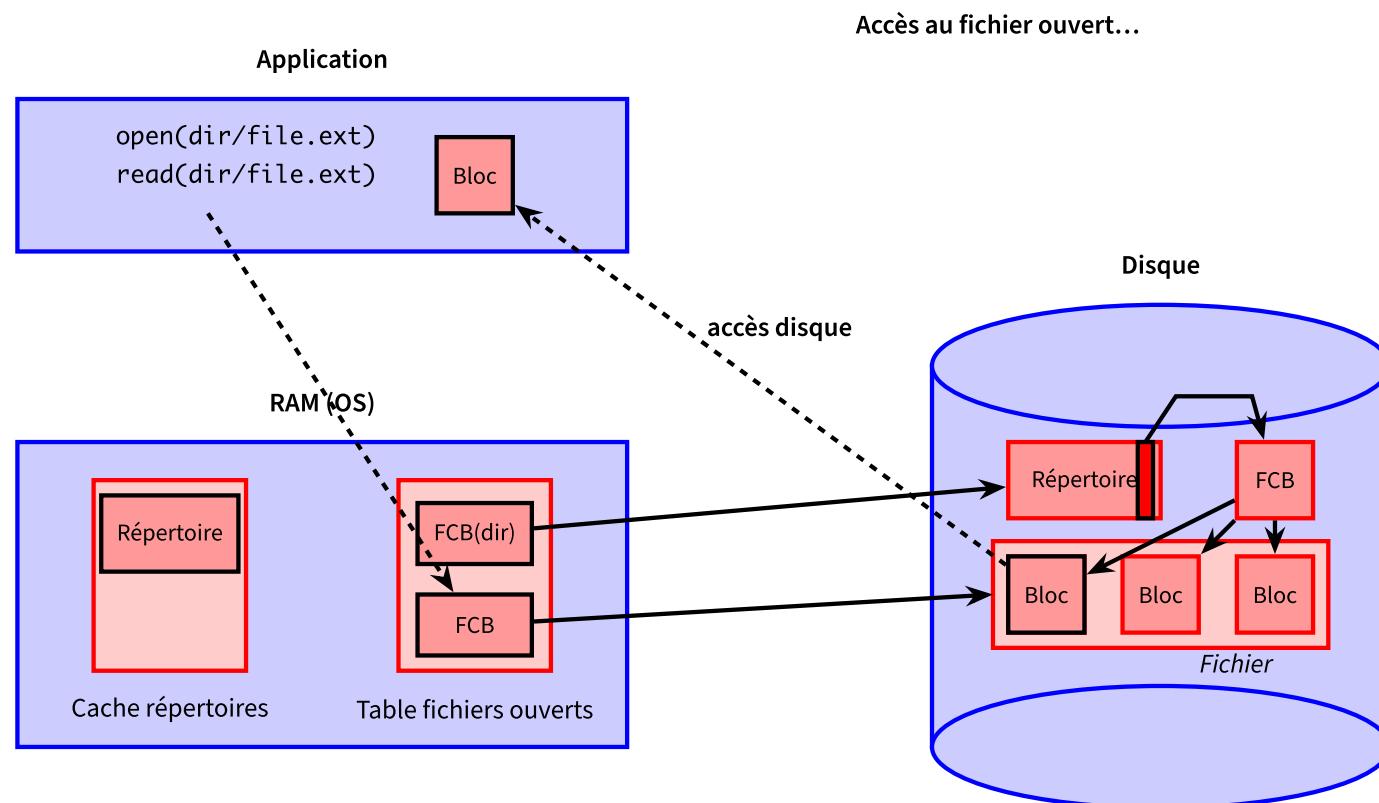
STRUCTURE D'UN SYSTÈME DE FICHIERS

- **Système logique**
 - Structure de répertoires
 - **FCB** + gestion de la protection
- **Système physique**
 - Fichiers → ensemble de blocs logiques
 - Bloc logique → blocs physique
 - Identification des blocs physiques selon support
- **Lien → pilote de périphérique**
 - **Appel système** (ex: chargement bloc **456**) → instruction matériel

MONTAGE DE RÉPERTOIRE

- **Le système de fichiers . . .**
 - ... associe des noms à des blocs logiques (**fichiers**)
 - ... associe des noms à des **répertoires** venus du disque!
- **Montage de répertoire**
 - Le montage consiste à positionner un répertoire dans le **FS**
 - Chargement du **FCB** par l'**OS** (disque → **RAM**)
 - Attribué à l'utilisateur du processus
 - Association dans le **MFD au niveau logique**
 - Les fichiers deviennent accessibles

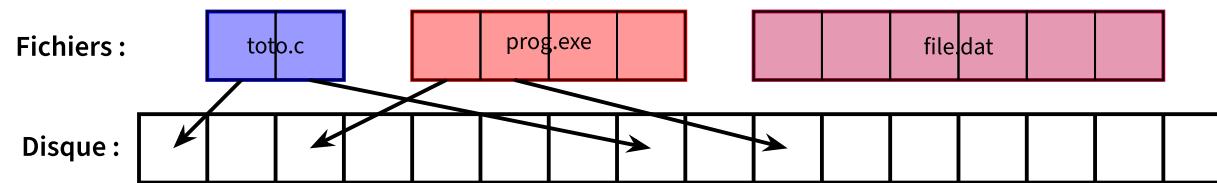
ACCÈS À UN FICHIER



ALLOCATION

Fichiers → blocs logiques → blocs physiques

➡ Choix des blocs physiques pour 1 fichier donné

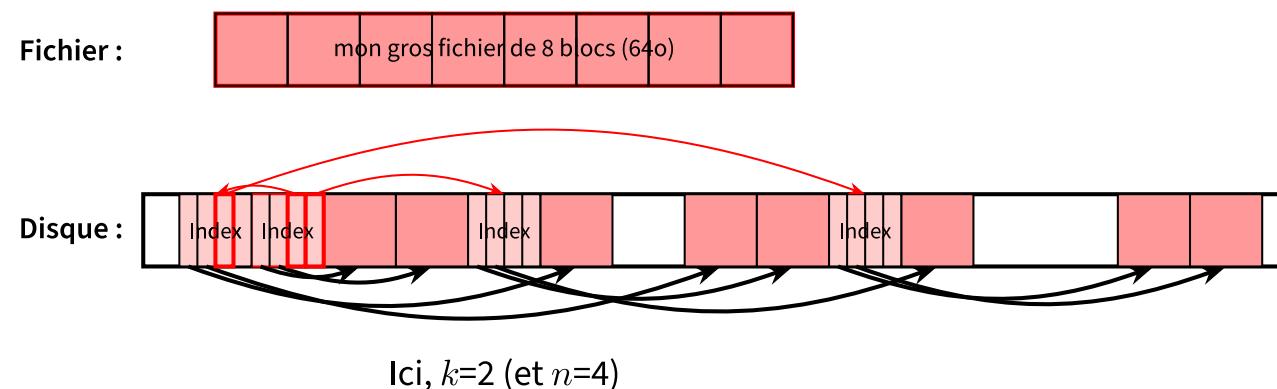


- **3 méthodes possibles**
 1. Allocation contiguë
 2. Allocation chaînée
 3. Allocation indexée

SCHÉMA COMBINÉ

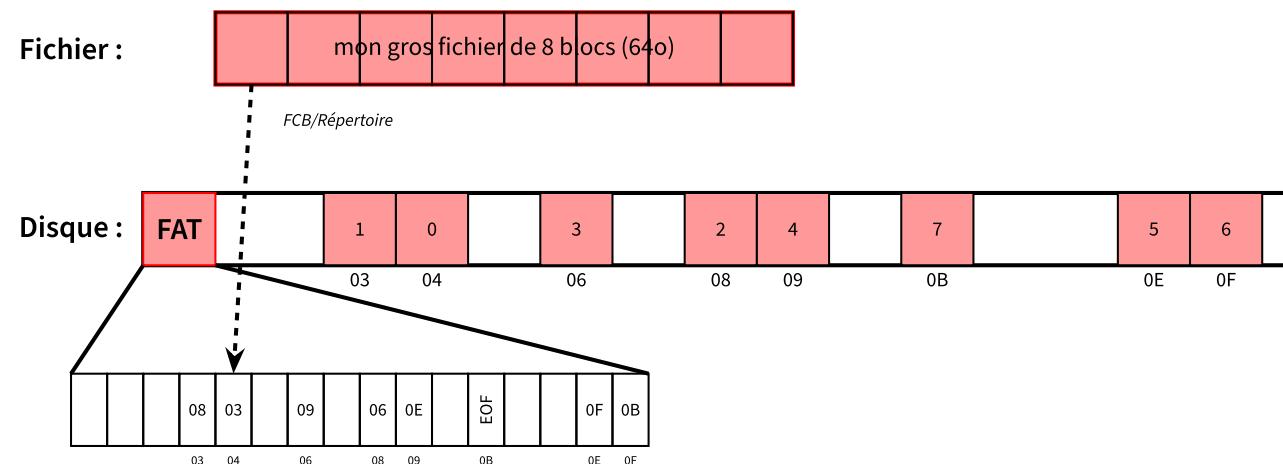
EX: LINUX EXTFS

- Combiner allocation chaînée et allocation indexée
- Index = k premiers blocs du fichier + $(n - k)$ blocs d'indirection
- ✓ Moins de perte pour les petits fichiers
- ✓ Accès rapide



FILE ALLOCATION TABLE (FAT)

- Utilisé sous **MSDOS (Intel)** et **OS/2 (IBM)**
- Allocation indexée
- Liste chaînée des **index** des blocs en **début de chaque partition**



FILE ALLOCATION TABLE (FAT)

- **Avantages**

- ✓ **FCB** → adresse premier bloc = premier index
- ✓ Pas de fragmentation (allocation indexée)
- ✓ Allocation bloc simple
- ✓ Accès rapide (**FAT** chargée en cache puis accès direct disque)

- **Inconvénients**

- ✗ Fiabilité: **FAT** perdue → disque foutu!
→ doubler la **FAT** (sur 2 blocs distincts)

AUTRES SERVICES DES FS

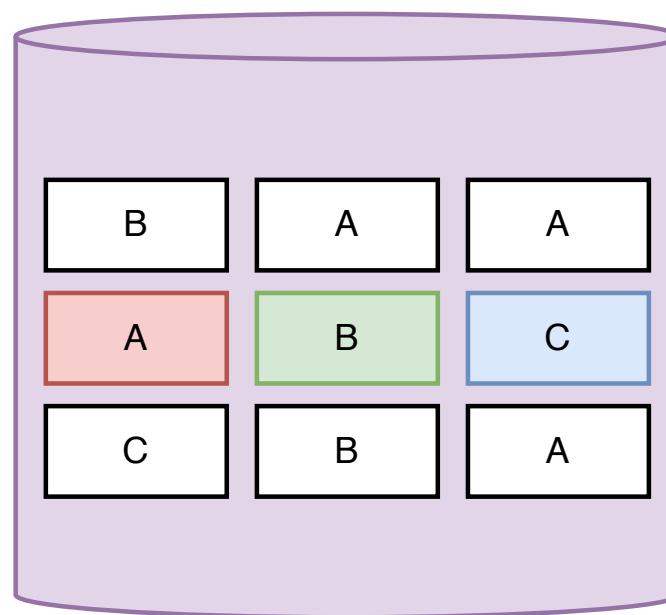
Les FS récents peuvent offrir des services supplémentaires:

- **Journalisation**
 - Actions d'écritures par transaction (**commit/rollback** en cas de problème)
 - Exemple : **NTFS, ext4**
- **Chiffrement**
 - Exemple : Windows Encrypted FileSystem (**EFS**)
 - Exemple : Linux **ext4, DM-Crypt LUKS** (bas niveau)
- **Compression**
 - Exemple : **ZFS, BrtFS, NTFS**

AUTRES SERVICES DES FS

Les FS récents peuvent offrir des services supplémentaires:

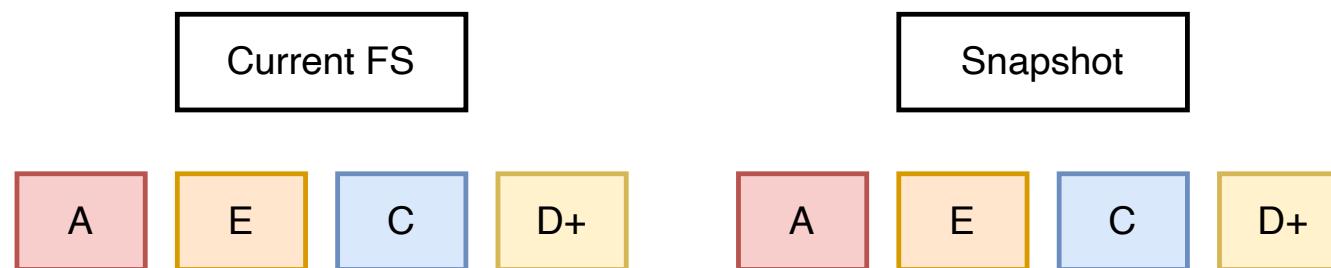
- **Déduplication**
 - Exemple : [BrtFS](#), [SDFS](#), [ZFS](#)



AUTRES SERVICES DES FS

Les FS récents peuvent offrir des services supplémentaires:

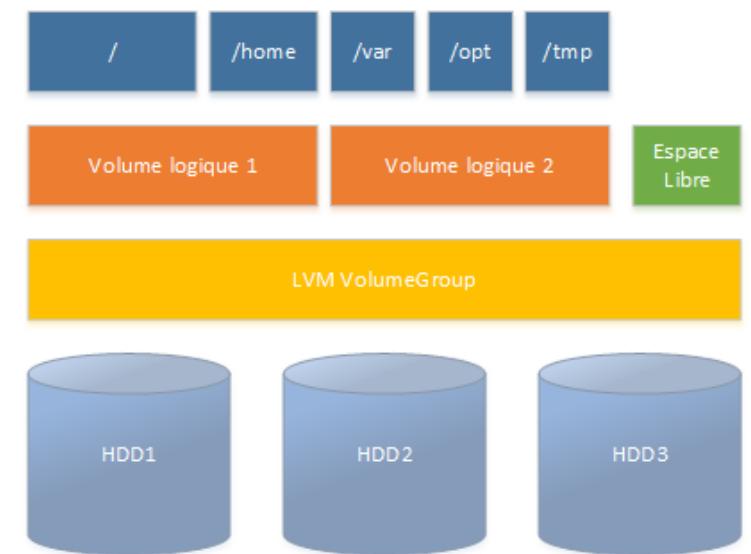
- **Clichés instantanés (snapshots)**
 - Technologie de **Copy-On-Write**
 - Exemple : **NTFS** (Volume Shadow Copy), Linux **LVM**
 - **Extension** : gestion de versions (snapshots continus)
 - Exemple : **ext3cow**, **tux3**, **NILFS**



VOLUME LOGIQUE

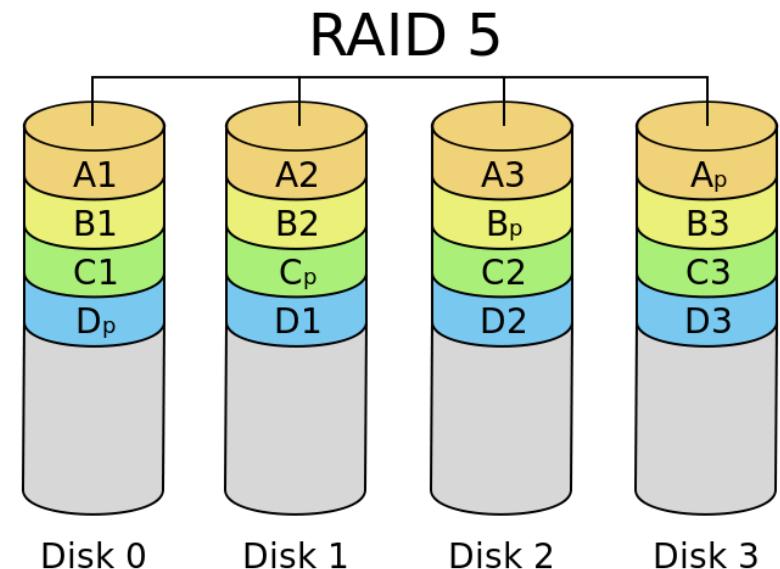
La gestion par **volume logique (LVM)** est une méthode d'allocation d'espace disque plus flexible que le partitionnement classique.

- regroupés en groupe de volumes équivalent à des pseudo disques durs.
- découpé en **volume logique** qui seront formatés et montés dans le **FS**.
- une manière de **virtualiser le stockage** qui permet beaucoup de flexibilité :
 - ➡ redimensionnement
 - ➡ tolérance aux pannes



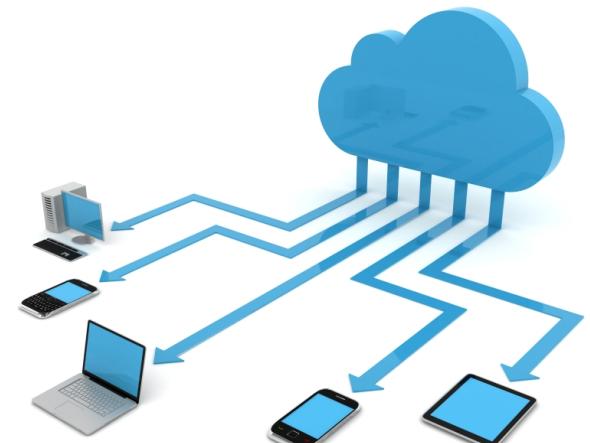
VOLUME LOGIQUE (RAID)

- Le **RAID** est un ensemble de **techniques de virtualisation** permettant de *répartir des données sur plusieurs disques durs*
- Le **RAID** permet d'améliorer :
 - ➡ les performances,
 - ➡ la sécurité
 - ➡ la tolérance aux pannes



LES SYSTÈMES DE FICHIER EN RÉSEAU

- Présentent des **ressources de stockage réparties sur un réseau**.
- Masquent les **FS** sous-jacent (Ex. : **NFS** de **Unix**, **CIFS** de **Microsoft**)
- Les nouvelles versions sont orientées **Cluster/Cloud** (Ex. : **Ceph**, **GlusterFS**, **GoogleFS/HadoopFS**, **RozoFS**)



SÉCURITÉ

⌚ Fonctionnalités principales d'un OS

SÉCURITÉ INHÉRENTE À L'OS

- Seul l'**OS** peut exécuter des **instructions privilégiées** (mode **Superviseur du CPU**)
 - Ex. : instructions de **bas niveau** (lecture/écritures de périphériques)
- Les **espaces mémoires** de chaque processus et du noyau sont **isolés** (**pagination, segmentation**)
- Un processus monopolisant le **CPU** ne bloque pas le système
 - **Timer matériel** déclenchant la préemption du **CPU**

AUTRES MÉCANISMES

- **OS multiutilisateurs**
 - plusieurs identités
 - les processus appartiennent à une identité
- **Privilèges**
 - des comptes avec des privilèges différents sur le système
 - ➡ root, administrator, ...
 - des processus privilégiés avec des niveaux de privilèges granulaires
 - ➡ capabilities sous Linux par exemple
- **Droits sur des objets (DAC)**
 - possibilité de positionner des droits sur les fichiers

PLAN

- Architecture des ordinateurs
- Structure et évolution des OS
- Fonctionnalités principales d'un OS
- Virtualisation

[Retour au plan](#) - [Retour à l'accueil](#)

OS ET VIRTUALISATION

Le **système d'exploitation** abstrait les ressources d'exécution en utilisant différentes stratégies :

1. **Multiplexer** une ressource (partager son utilisation).
 - ➡ **CPU** en **temps partagé**.
2. **Emuler** une ressource (créer une ressource purement logicielle).
 - ➡ **Mémoire virtuelle** swappée.
3. **Agréger** plusieurs ressources pour n'en présenter qu'une.
 - ➡ arborescence du **Virtual FileSystem** et **Volumes logiques**

AVANT LA VIRTUALISATION

Deux approches :

1. **Un gros serveur** portant de nombreuses applications

- ✗ problèmes de dépendances sur des librairies
- ✗ problèmes de maintenance
 - impact d'une opération d'un service sur un autre
- **solution** → limiter les dépendances entre applications

2. **Un serveur par application**

- ✓ Bonne isolation,
- ✗ mais quel gâchis (HDD, RAM, CPU) !
- **solution** → consolidation de serveur

NIVEAUX DE VIRTUALISATION

1. Machines virtuelles (émulation)
2. Conteneurs
3. Applications cloud native

LES MACHINES VIRTUELLES (ÉMULATION)

⌚ Niveaux de virtualisation

LES MACHINES VIRTUELLES

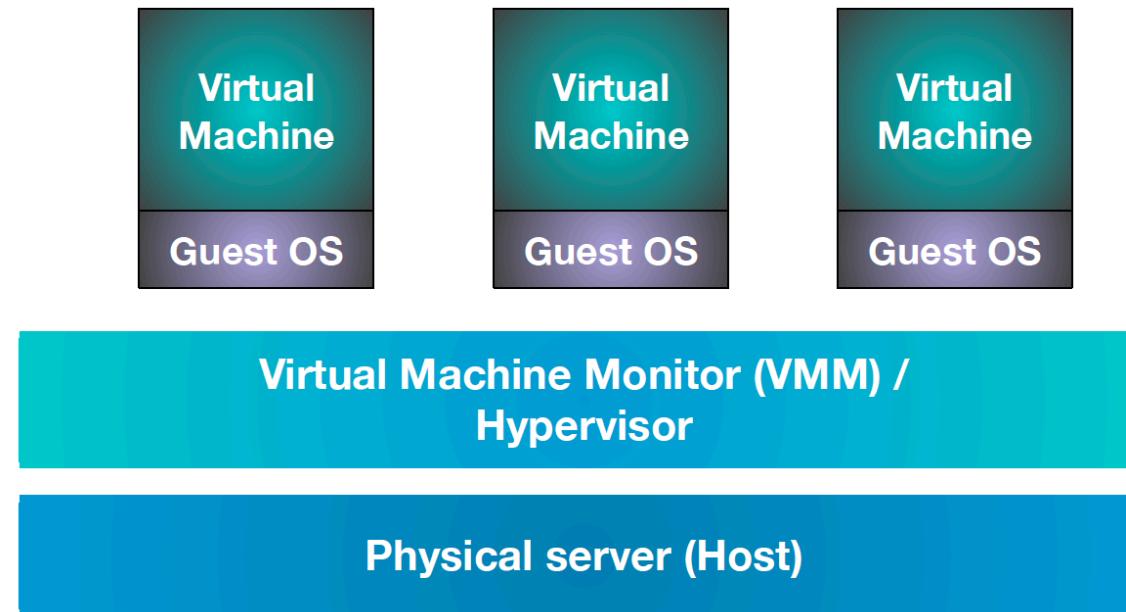
Une machine virtuelle (Virtual Machine - VM)

- un environnement qui fonctionne comme un ordinateur virtuel avec ses propres ressources (processeur, mémoire, disque, ...)

Les composants d'une VM

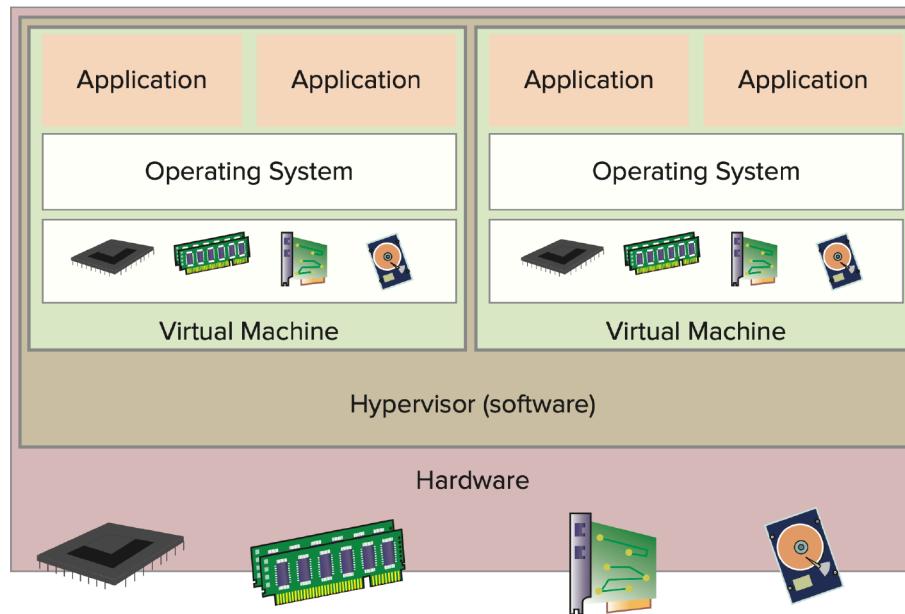
- une ou plusieurs applications.
- un système d'exploitation.
- un ensemble de périphériques virtuels.

LES HYPERVISEURS



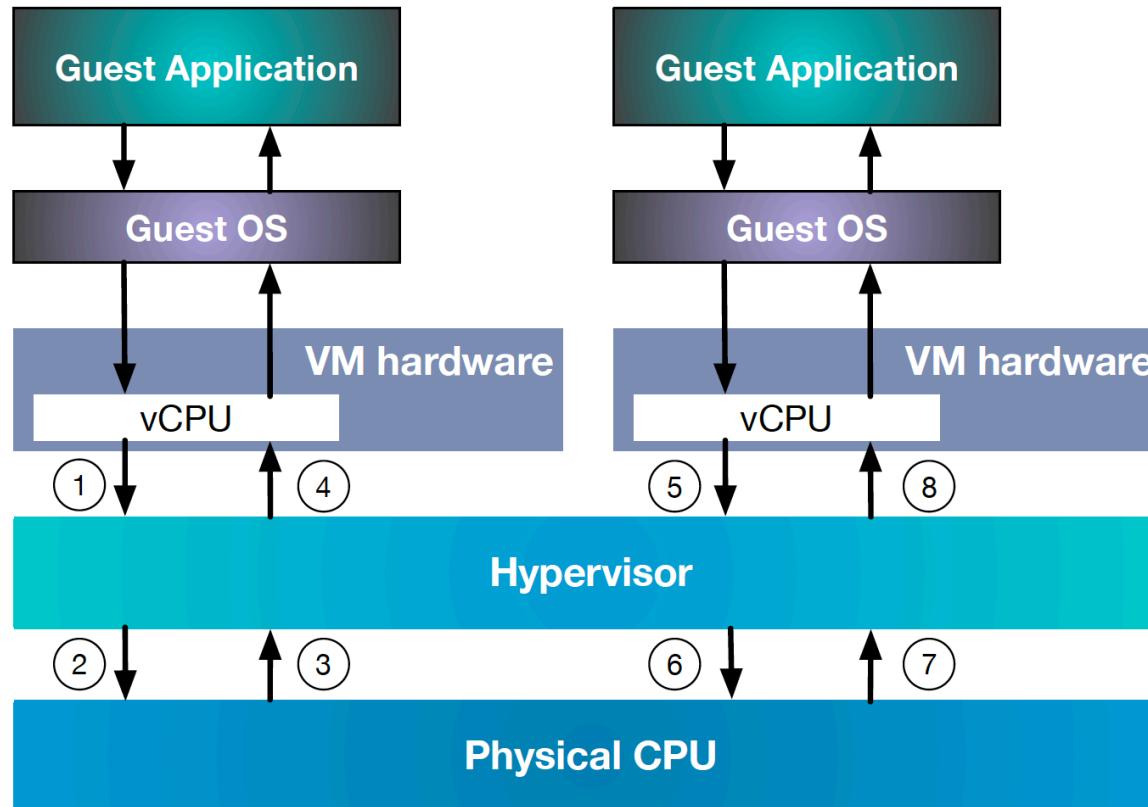
Un hyperviseur est un composant logiciel qui gère les interactions entre les machines virtuelles et le matériel sous-jacent.

LE RÔLE DES HYPERVISEURS



- Autorisez les **VM** à **partager les mêmes ressources** physiques.
- Présentez à chaque **VM une fraction** des ressources physiques.
- **Répond aux demandes** des **VM** invitées.

COMMENT LE CPU EST VIRTUALISÉ

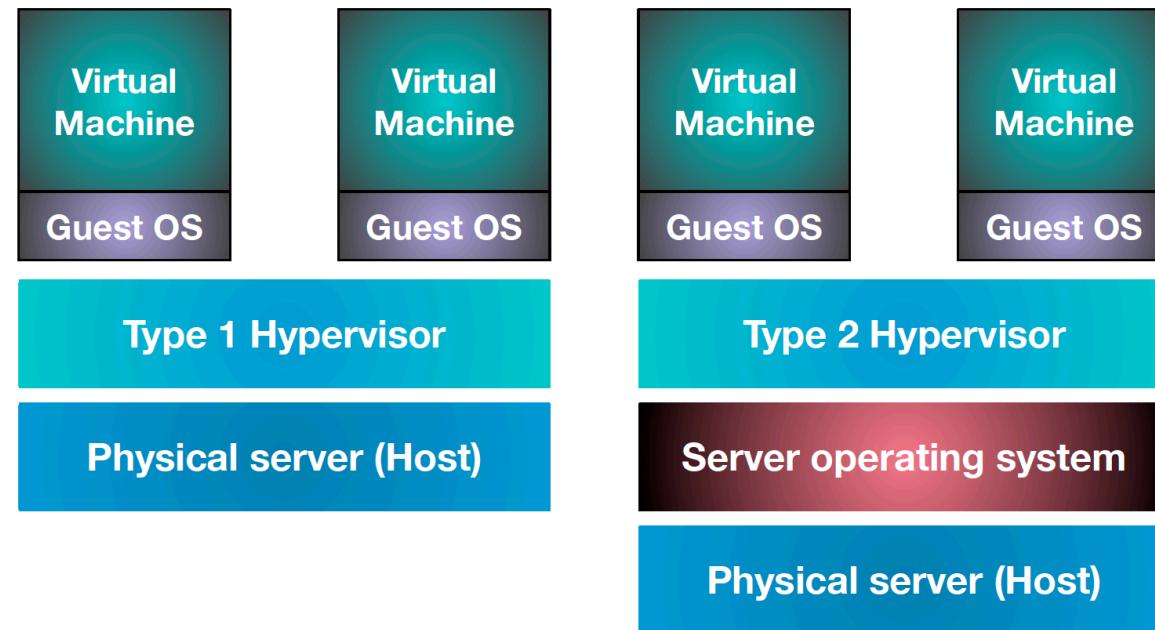


L'**hyperviseur** planifie des **tranches de temps** sur la **CPU physique** pour exécuter les instructions des **CPU virtuelles**.

COMMENT LE CPU EST VIRTUALISÉ

- Les serveurs → **plusieurs processeurs** avec **plusieurs cœurs**.
- Chaque **VM** peut avoir plusieurs **vCPU**.
- L'**hyperviseur** ne mappe pas statiquement **vCPU** à un cœur.
 - ➡ une **VM** ne peut pas avoir plus de **vCPU** que le nombre total de cœurs physiques.
 - ➡ le nombre de **vCPU** sur toutes les **VM** peut dépasser le nombre de cœurs physiques.

LES TYPES DE HYPERVISEURS



- **Type 1** → s'exécute sur le matériel hôte ([Microsoft Hyper-V](#))
- **Type 2** → s'exécute en tant qu'application sur l'OS hôte ([VirtualBox](#))

LES AVANTAGES DES VMS

- ✓ Plusieurs serveurs virtuels dans un serveur physique.
 - meilleure utilisation des ressources.
 - coûts inférieurs.
 - gestion simplifiée du data-center.
- ✓ Création facile de VMs à partir de modèles/clones.
 - un serveur virtuel peut être opérationnel en quelques minutes/heures.
 - ➡ un serveur physique peut avoir besoin de semaines.
- ✓ Migration facile des VMs.
 - temps d'arrêt réduits ou minimisés.

LES CONTENEURS

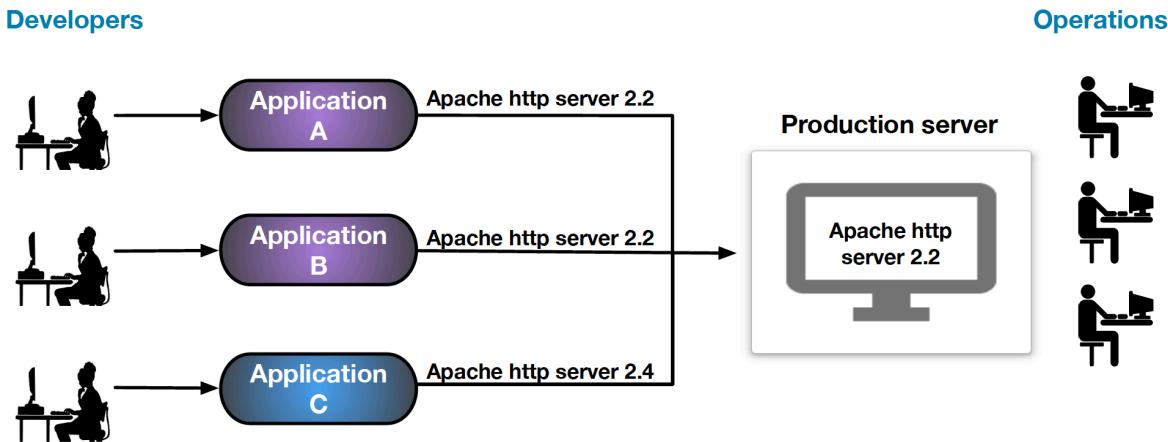
⌚ Niveaux de virtualisation

POURQUOI LA CONTENEURISATION ?

Gabriel N. Schenker : Utiliser une VM pour packager une seule application revient à utiliser « *un bateau gigantesque pour transporter un camion chargé de bananes* ».

- **Machines virtuelles**
 - une bonne solution pour **packager un microservice**.
 - le microservice est **intégré** à la machine virtuelle.
 - **déploiement facile** sur n'importe quelle machine.
- ... cependant, les **VMs sont lourdes** :
 - ✗ elles contiennent un **OS complet**.
 - ✗ le **démarrage** ou la **migration** d'une **VM** peut prendre du temps.

POURQUOI LA CONTENEURISATION ?

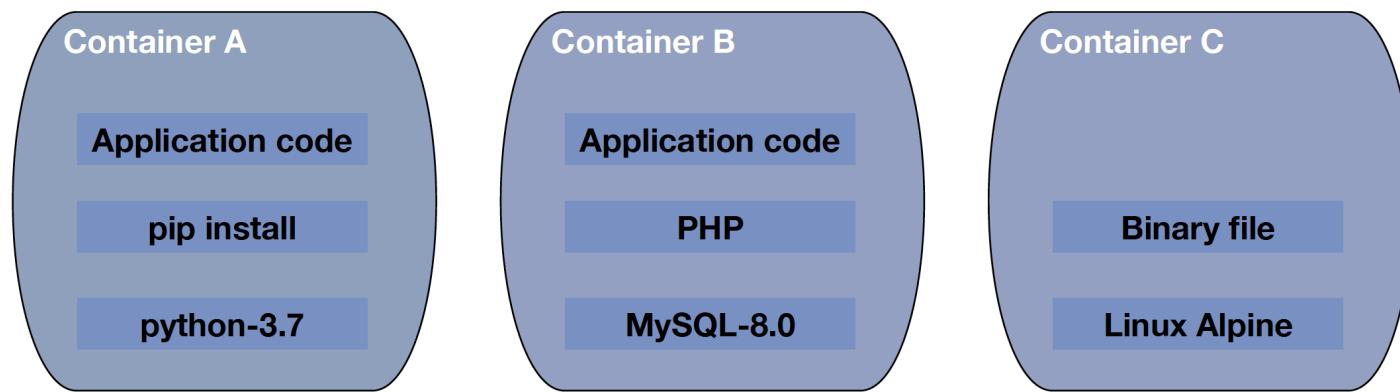


- ✖ Chaque **application** est livrée avec un "**package différent**".
- ✖ **Déploiement** dépend de l'application.
- ✖ Gestion de **dépendance** difficile en production.

Solution

- ➡ Les **conteneurs** standardisent le **développement** et le **déploiement**.
- ➡ Une excellente solution dans **un environnement cloud**.

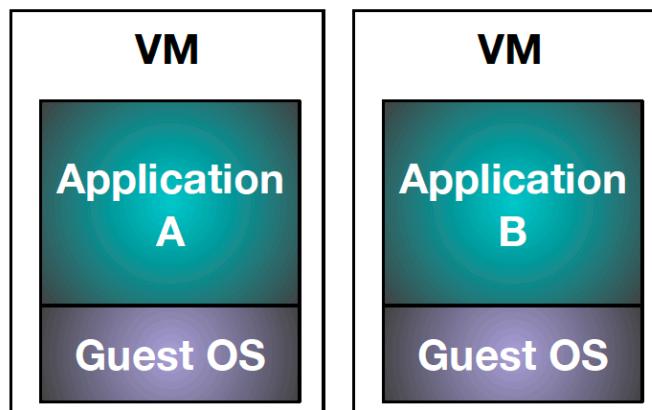
C'EST QUOI UN CONTENEUR ?



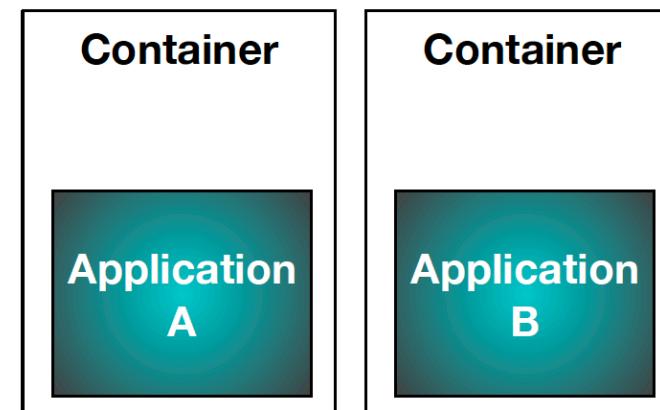
Une unité logicielle regroupant le **code** et toutes ses dépendances pour une **exécution** rapide et fiable de l'application d'un **environnement à un autre**.

VM vs CONTENEUR

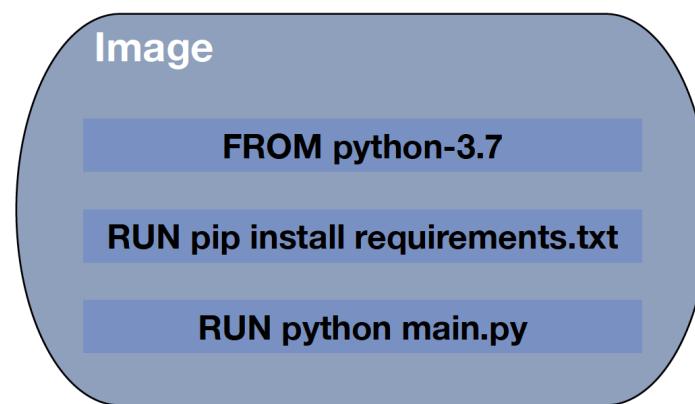
Virtual machines



Containers

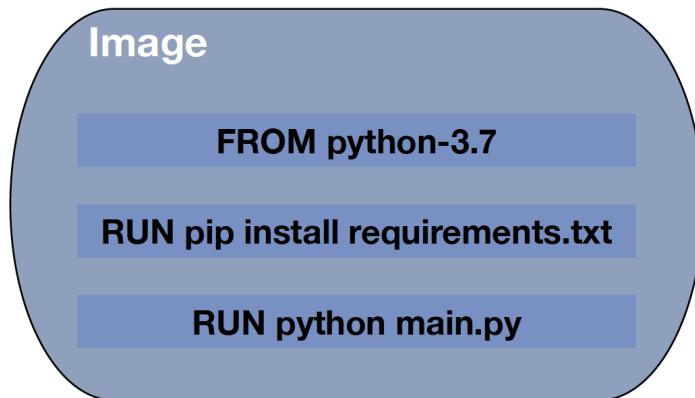


DOCKER



- Une plate-forme ouverte pour le **développement**, la **livraison** et l'**exécution** d'applications.
- Il offre la possibilité de **packager** et d'**exécuter** une application dans un **environnement isolé** appelé **conteneur**

DOCKER

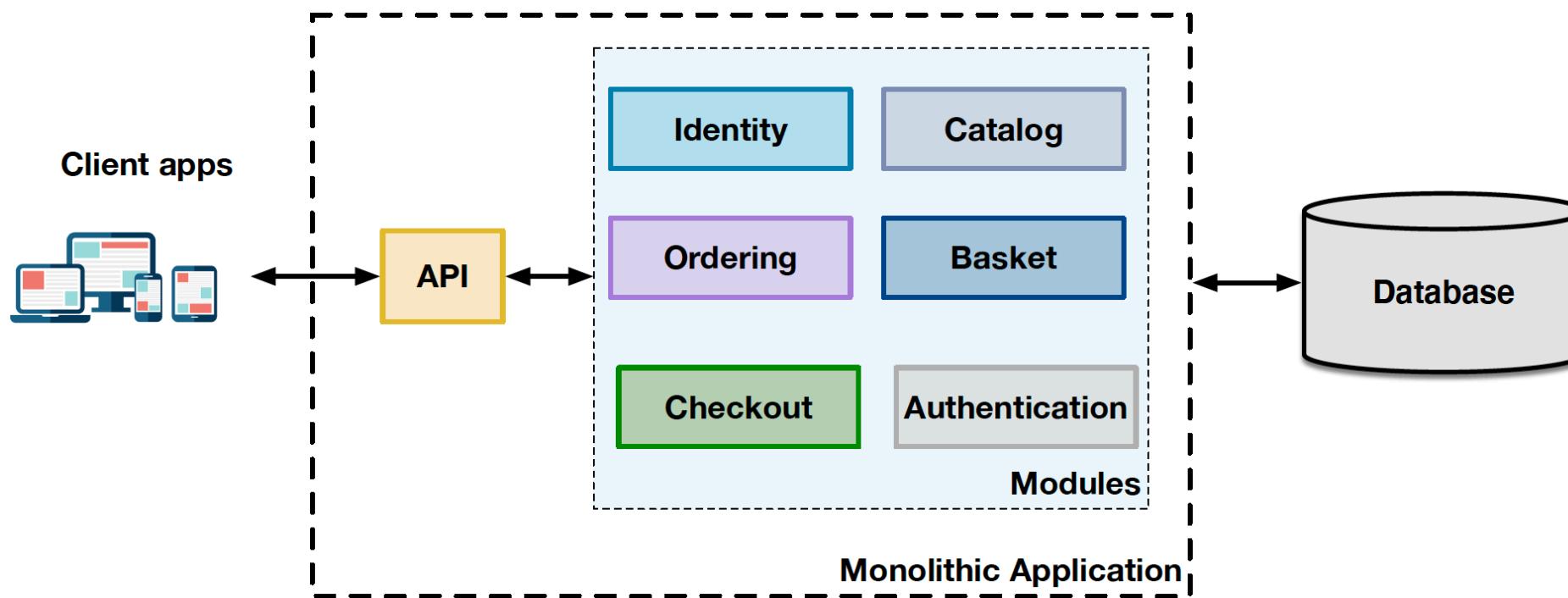


- Docker crée et exécute un conteneur à partir d'une image.
- Image → modèle en lecture seule avec des instructions pour créer et exécuter un conteneur.
- Image contient tous les fichiers nécessaires pour exécuter une application dans un conteneur.

APPLICATIONS CLOUD NATIVE

⌚ Niveaux de virtualisation

APPLICATIONS TRADITIONNELLES



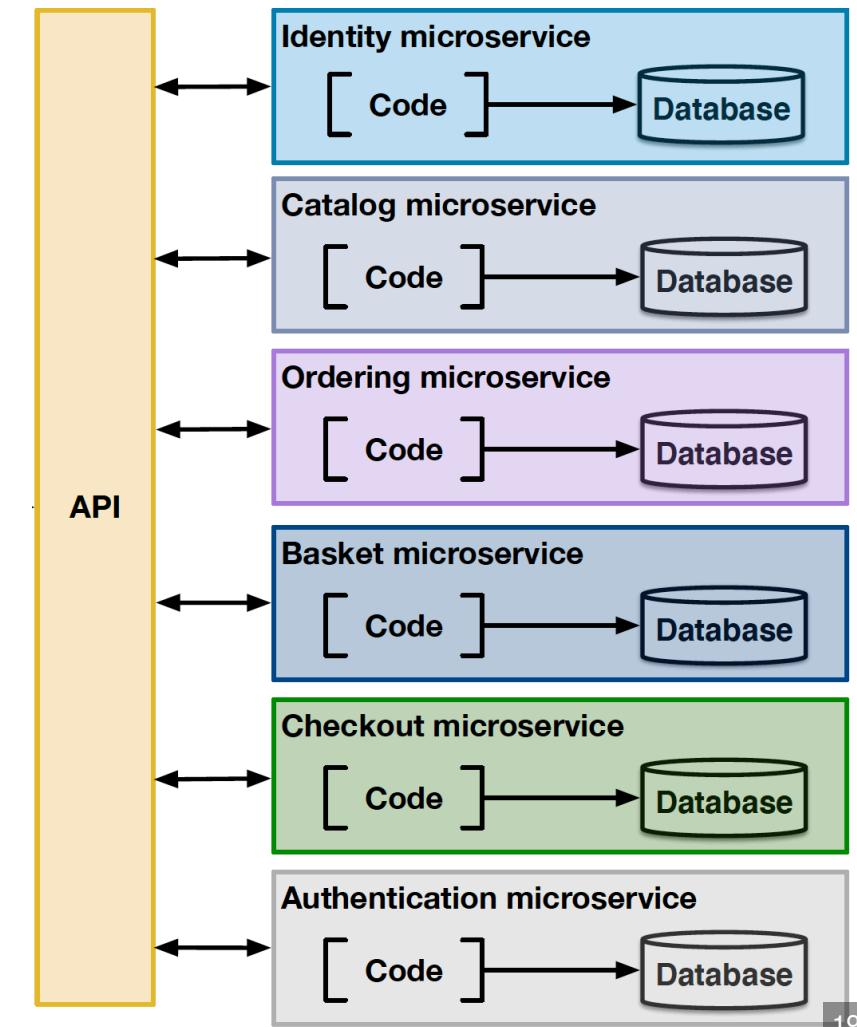
- Les **applications traditionnelles** ont **une architecture monolithique**.
- Toute la **logique métier** est intégrée dans **une seule application**.

LES INCONVÉNIENTS

- ✗ Le **code** est **difficile à maintenir**.
- ✗ La **base de données** est **difficile à maintenir**.
- ✗ Chaque modification entraîne le **déploiement de l'ensemble de l'application**.
- ✗ Une **erreur** dans un module affecte **l'ensemble de l'application**.
- ✗ **Difficile** d'adhérer aux **pratiques DevOps**.
 - **DevOps** : modularité, livraison, déploiement continu, isolation des pannes ...

ARCHITECTURES MICROSERVICES

- Application : **microservices** faiblement couplés
- Chaque **microservice** est un processus indépendant.



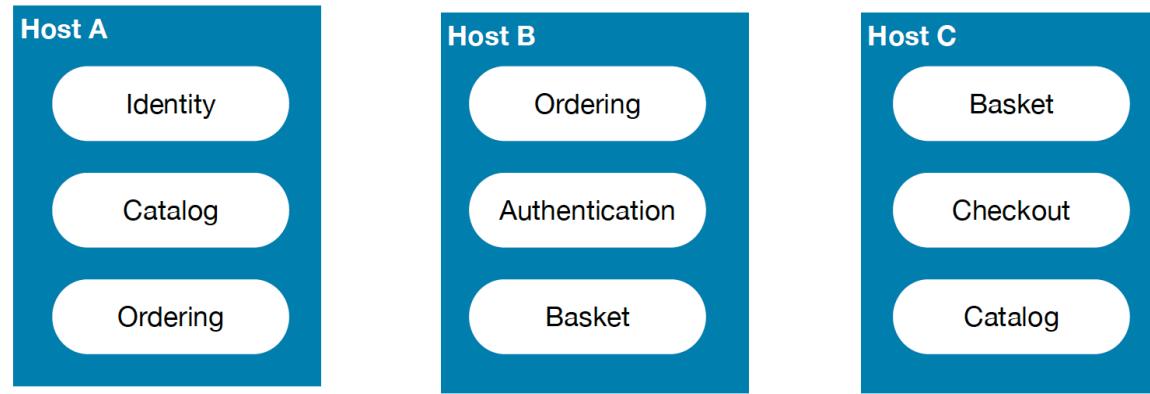
LES AVANTAGES

- ✓ Respect des principes DevOps.
- ✓ Différents microservices, différentes technologies.
- ✓ Microservices évoluent indépendamment les uns des autres.
- ✓ BDD décentralisées :
technologie de BDD personnalisée pour chaque microservice.
- ✓ Bon choix architectural pour une application cloud native.

DÉPLOYER DES MICROSERVICES

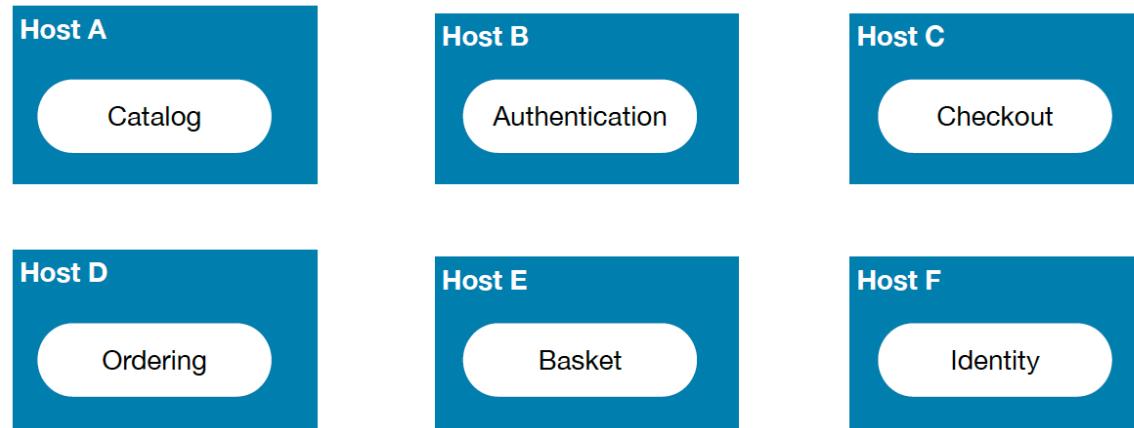
- **Sans virtualisation**
 - plusieurs instances de service par **hôte**.
 - ➡ chaque **hôte physique** exécute plusieurs instances de service.
 - une seule instance de service par **hôte**.
 - ➡ chaque **hôte physique** exécute une seule instance de service.
- **Avec virtualisation**
 - une seule instance de service par **machine virtuelle (VM)**.
 - ➡ un service s'exécute dans une machine virtuelle.
 - une seule instance de service par **conteneur**.
 - ➡ un service s'exécute dans un conteneur.
 - de nombreuses **VM/conteneurs** s'exécutent sur le même hôte physique.

PLUSIEURS INSTANCES PAR HÔTE



- ✓ Utilisation efficace des ressources.
- ✗ Isolement limité des instances de service.
- ✗ Difficile de limiter les ressources qu'un service utilise.
- ✗ Déploiement difficile :
les services sont écrits dans différents langages et frameworks.

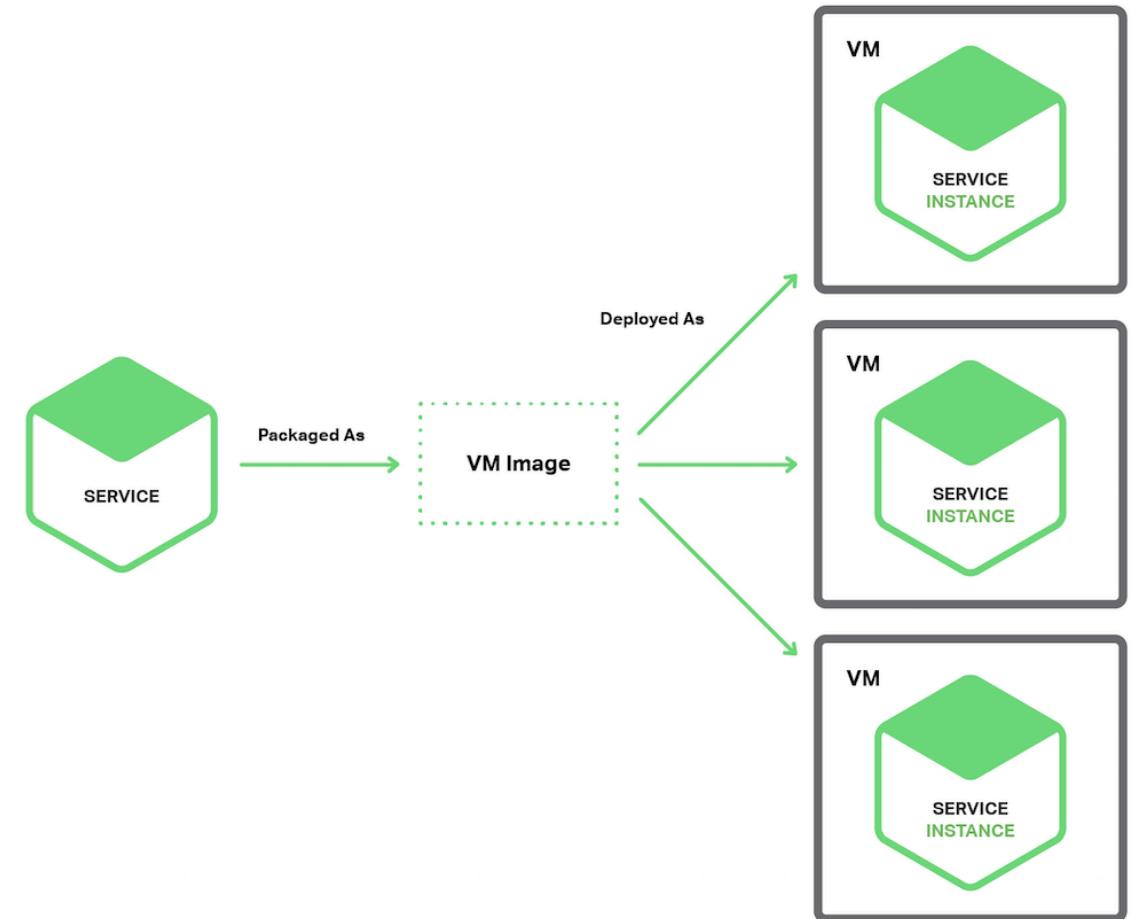
INSTANCE DE SERVICE PAR HÔTE



- ✓ **Isolement** entre les instances de service.
- ✓ **Monitoring facile** des ressources utilisées par un service.
- ✓ **Déploiement facile** :
pas d'exigences contradictoires entre deux prestations de service.
- ✗ **Utilisation inefficace** des ressources :
toutes les ressources d'un hébergeur sont dédiées à un seul service.

INSTANCE DE SERVICE PAR VM

- Chaque microservice est installé en tant que VM.
- Solution adoptée par Netflix.

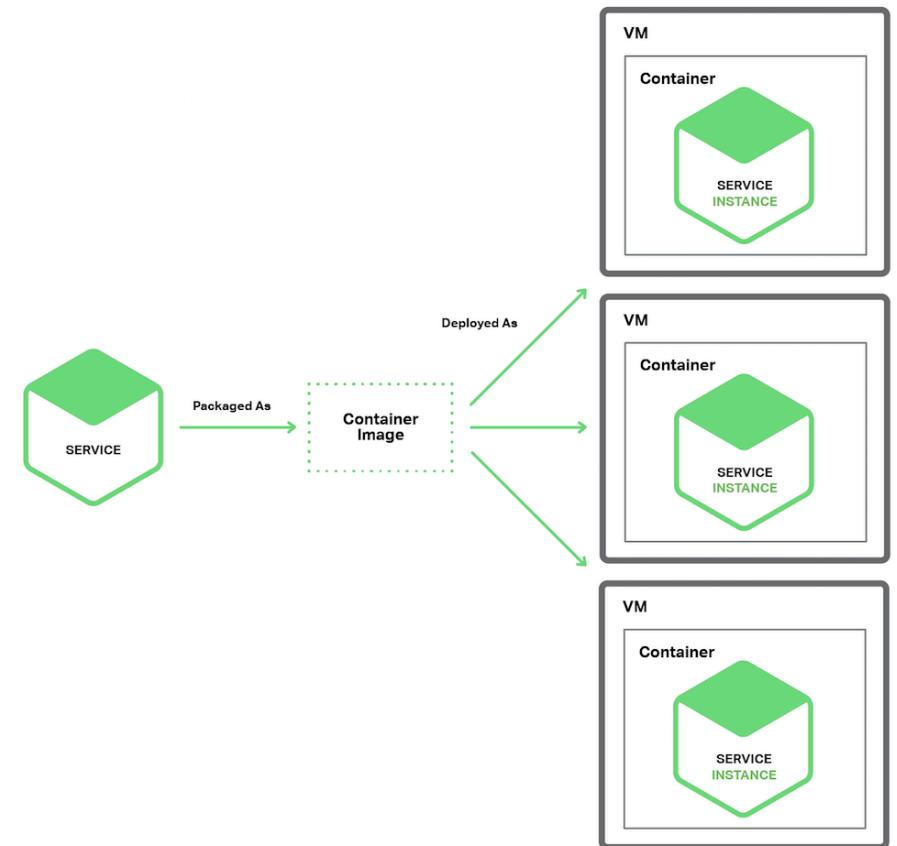


INSTANCE DE SERVICE PAR VM

- ✓ **Isolement** complet.
- ✓ **Quantité fixe de ressources par VM** (CPU, mémoire, ...).
- ✓ **Les détails technologiques sont cachés dans la VM.**
 - ✓ tous les services sont démarrés et arrêtés de la même manière.
- ✓ Infrastructure **Cloud mature**
 - ✓ équilibrage de charge et autoscaling.
- ✗ Les **images de VM sont lentes** à créer et à instancier.
 - ✗ Une machine virtuelle contient généralement un OS complet
 - ✗ Des solutions existent pour générer des images légères

INSTANCE DE SERVICE PAR CONTENEUR

- **Microservice** installé sous forme d'image de **conteneur (VM légère)**.
➡ ils contiennent **ce qu'il faut pour exécuter le service**
- **Les conteneurs** peuvent s'exécuter sur une **VM** ou un **serveur physique**.
- Tous **les avantages des VM** et ... sont **rapides à construire**



Source : <https://www.nginx.com/>

SERVERLESS DEPLOYMENT

- La **serverless computing platform** (**Google, Microsoft Azure, ...**) exécute autant d'instances du service que nécessaire.
- **Aucune visibilité** sur l'infrastructure sous-jacente.
 - ✓ pas besoin de configurer l'infrastructure sous-jacente.
 - ✓ payez à la demande, au lieu de payer pour l'infrastructure.
 - ✗ inapte aux services de longue durée.
 - ✗ aucun contrôle sur les performances.

ON-DEMAND COMPUTING

La **virtualisation** à la base du **Cloud Computing**

- **Cloud privé**
 - mutualisation des ressources entre départements
 - la **DSI** devient un prestataire de service
- **Cloud public (Amazon, MS Azure, ...)**
 - nouveau modèle économique: *Pay As You Go*
- **Cloud hybride**
 - en cas de dépassement de capacité des ressources propres
 - ➡ extension au **cloud public**

MERCI

[Version PDF des slides](#)

[Retour à l'accueil](#) - [Retour au plan](#)