Système d'Exploitation

MEEF "Numérique et science informatique" (MEEF-NSI)

Fabien Rico

Univ. Claude Bernard Lyon 1

3 juin 2022

1. Introduction
    - Interface avec le matériel
2. Les processus
    - État d'un processus
    - Environnement
    - Communication entre processus
    - Les processus et python
3. Gestion de la mémoire
    - Rôle de la mémoire
    - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
    - Algorithmes d'ordonnancement
    - Avec préemption
    - Contraintes entre tâches
6. Configuration et services
    - Gestion des utilisateurs
    - Configuration du système
    - Rôles du système

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Introduction

Qu'est-ce qu'un système d'exploitation ?

Littéralement, *ce qui permet d'utiliser la machine.* Quatre grands rôles :

- Interface entre applications et matériel (*e.g.*, gestion des périphériques)
- Organisation (*e.g.*, des disques, de la mémoire, et des processus)
- Sécurité (*e.g.*, des données, du matériel)
- Interaction avec le ou les utilisateurs (*e.g.*, comptes, droits, installation)

Différents types de systèmes d'exploitation, pour différents usages :

- Systèmes « généralistes », multi-utilisateurs et multi-tâches : GNU-Linux (Debian, Ubuntu, Redhat. . .), Windows, Mac OS X. . .
- Pour l'embarqué (contraintes sur l'utilisation des ressources) : Windows Embedded Compact, Android. . .
- Pour le temps-réel (contraintes d'échéance) : RTLinux, RTAI.

Interface avec le matériel

Que se passe-t-il lorsque vous utilisez la commande Python `print("Message")`

- Le message est affiché sur la *sortie standard*
- Quelle est cette sortie standard ?
  - le terminal de commande qui a lancé le programme ;
  - une fenêtre de l'environnement de programmation ;
  - un fichier ;
  - la source de donnée d'un autre programme (pipe) ;
  - le fichier de logs d'un serveur ;
  - un système de logs centralisés d'un cluster ;
  - rien.

La même action pour un code peut donner des résultats différents en fonction de ce que le système défini comme *sortie standard*.
La sortie standard est un objet virtuel proposé par le système. On peut écrire des données dessus, mais c'est via le système que l'on décide de ce que ces données vont devenir.

Utilisateur

Matériel

Utilisateur

Applications

Matériel

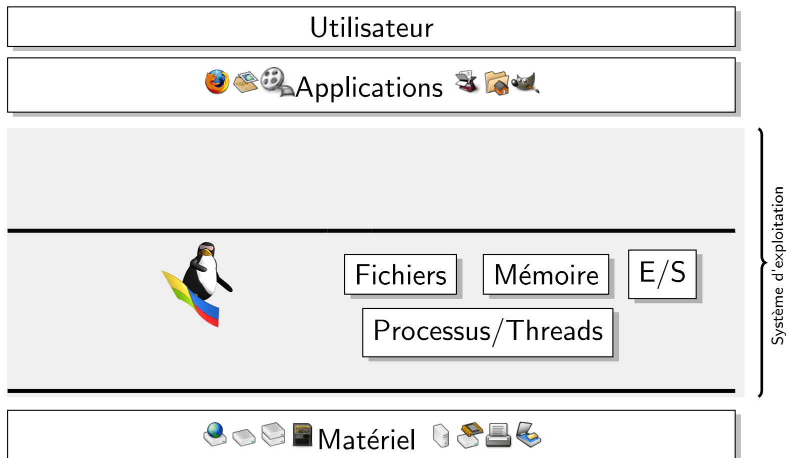
Utilisateur

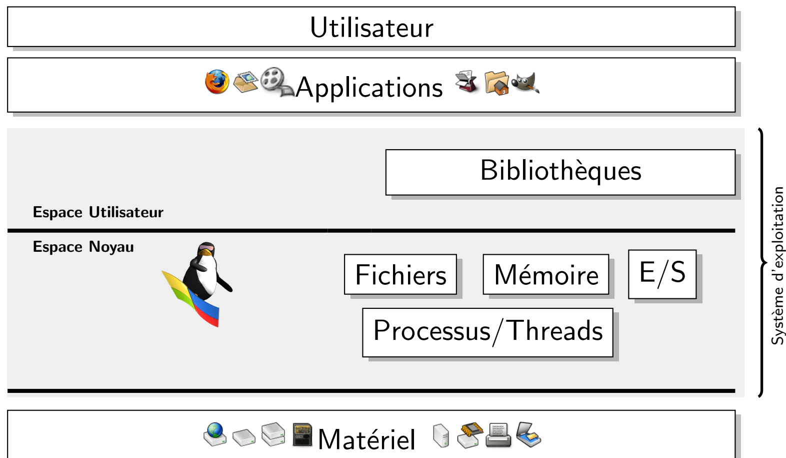
Applications

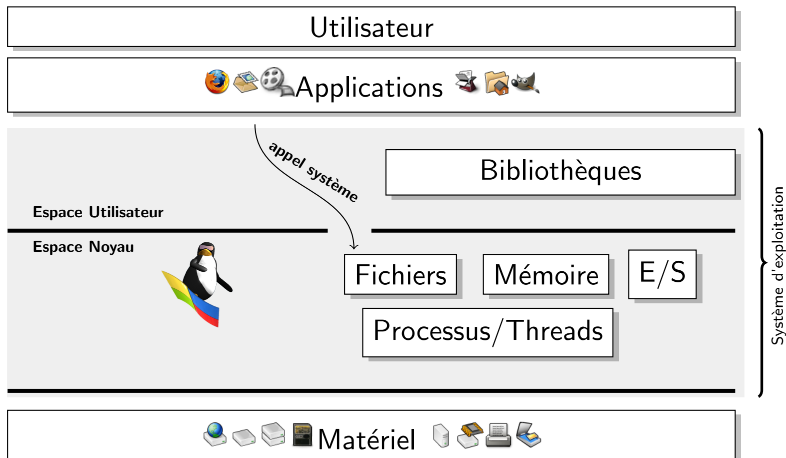
Fichiers | Mémoire | E/S

Processus/Threads

Matériel


Utilisateur
Applications
Fichiers
Mémoire
E/S
Processus/Threads
Système d'exploitation
Matériel


Utilisateur
Applications
Bibliothèques
Espace Utilisateur
Espace Noyau
Fichiers
Mémoire
E/S
Processus/Threads
Système d'exploitation
Matériel


Utilisateur
Applications
appel système
Bibliothèques
Espace Utilisateur
Espace Noyau
Fichiers
Mémoire
E/S
Processus/Threads
Système d'exploitation
Matériel

Fonctionnalités

En tant qu'interface, un système d'exploitation peut :

- tricher
  - proposer un fichier qui est un fait un objet virtuel sauvegardé ailleurs (disque réseau=*Cloud*) ;
  - enfermer le processus dans un environnement contraint (*sandbox*, *conteneur*) ;
  - proposer une machine virtuelle complète ;
- organiser
  - gérer les conflit d'accès ;
  - stopper un processus qui demande des données qui ne sont pas encore arrivées ;
  - débloquer ce processus lorsque les données arrivent ;
- assurer la sécurité
  - faire des vérifications de droits ;
  - refuser des opérations impossibles ;
  - retarder une modification qui pourrait endommager le matériel.

Notions de base

On voit apparaître dans ce qui précède certaines notions de base du système :

- **Rôle d'interface** : le système se place entre les logiciels et le matériel.
- **Objets virtuels** : le système définie des objets (fichier, processus, ...) qui sont manipulés par les logiciels.
- **Niveau de fonctionnement** : au moins deux niveaux de fonctionnenement, comportant des droits différents gérés pas le matériel :
  - **espace utilisateur**
  - **espace noyau**

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Processus

Définition (Processus)

Un processus est une instance (une exécution) d'un programme. C'est un ensemble de données gérées par le noyau qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, de le stopper et de le reprendre.

Processus

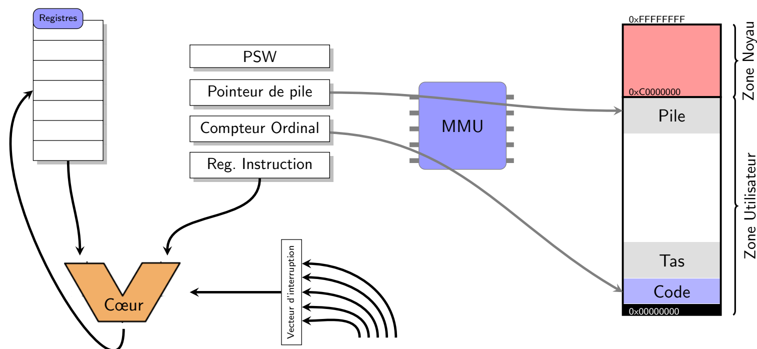
Définition (Processus)

Un processus est une instance (une exécution) d'un programme. C'est un ensemble de données gérées par le noyau qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, de le stopper et de le reprendre.

Son rôle est :

- d'identifier le programme ;
- de vérifier que le programme ne fait que ce qu'il a le droit de faire ;
- de lui permettre d'accéder aux fichiers et aux ressources ;
- d'isoler les programmes les uns des autres ;
- de libérer les ressources à la fin de l'exécution (fermer les fichiers, libérer toute la mémoire allouée, etc.) ;
- de permettre le passage d'une tâche à l'autre.

Processeur



Constituant d'un processus

- Position en cours dans le code du programme (Compteur ordinal)
- Contenu des registres du processeur qui l'exécute
- Informations sur la mémoire
- État (en cours, prêt, bloqué, ...)
- Ressources (fichiers, sockets, ...)
- Environnement (répertoire courant, id utilisateur, ...)
- Informations de gestion (id, pid, groupe, ...)
- ...

Sous linux ces informations peuvent se trouver dans le répertoire virtuel `/proc/<pid processus>`

État d'un processus

Rappel : Le système doit

- gérer l'accès au processeur
- gérer l'occupation de la mémoire

Définition (état d'un processus)

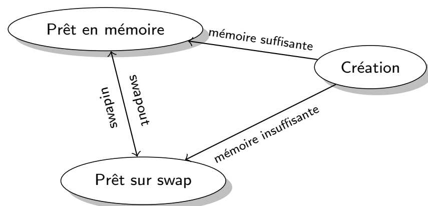
Le système doit donc gérer plusieurs listes de processus,

- ceux qui peuvent s'exécuter
- ceux qui sont en mémoire
- ceux qui sont bloqués car demandant une ressource occupée
- ...

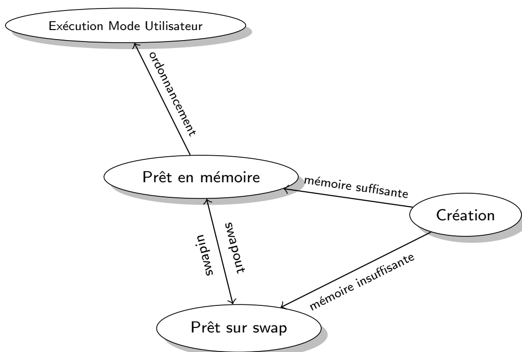
On parle *d'état du processus*

État d'un processus

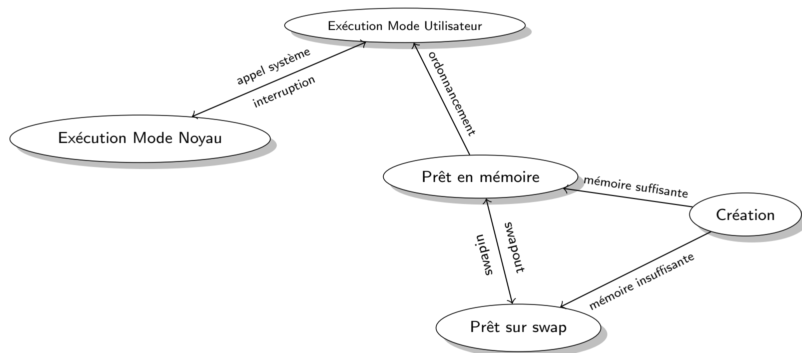
État d'un processus



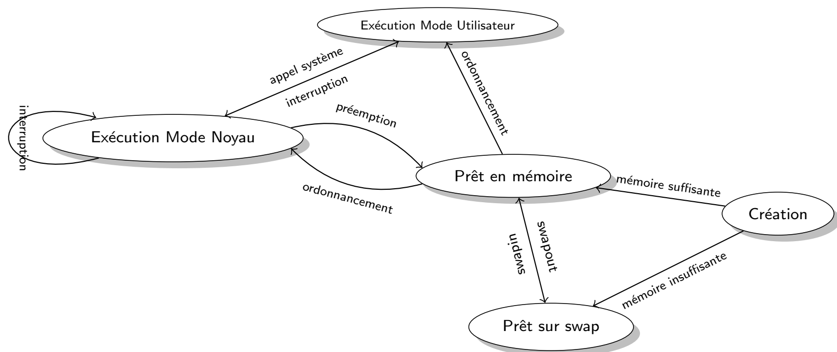
État d'un processus



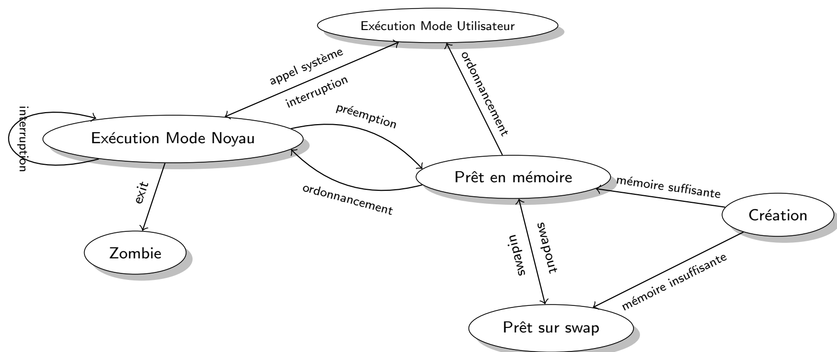
État d'un processus



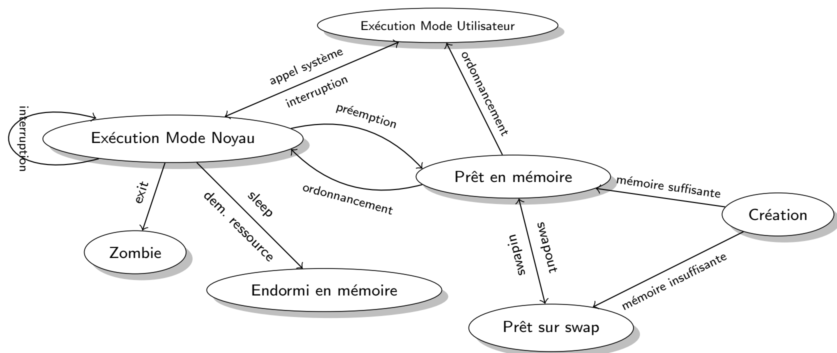
État d'un processus



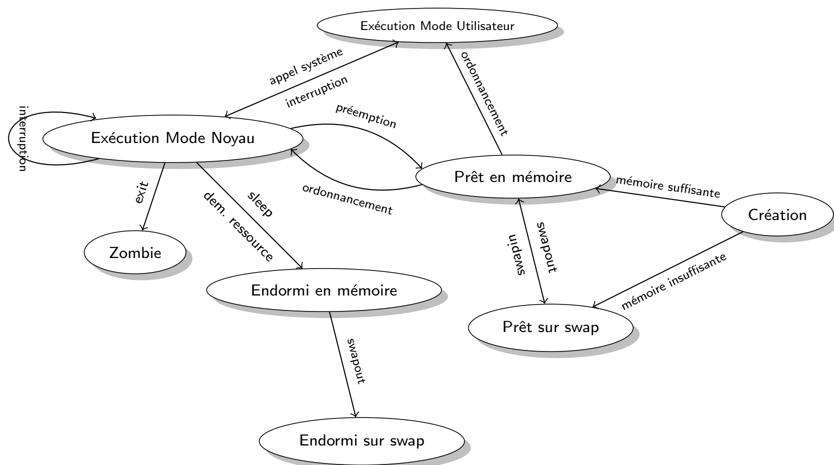
État d'un processus



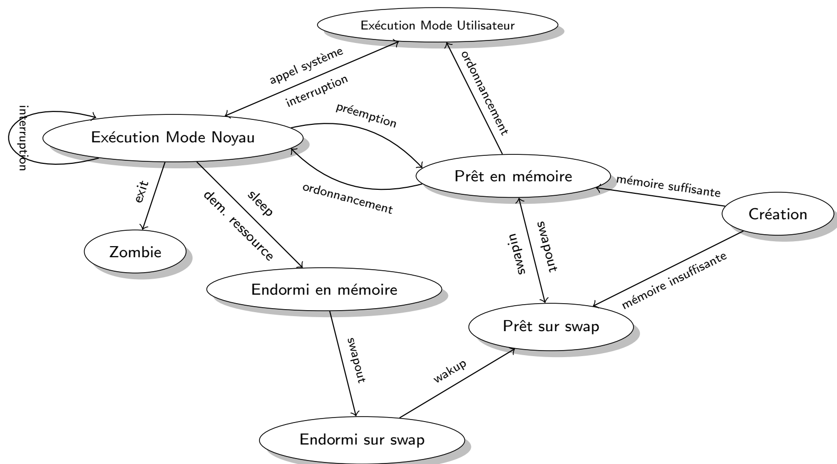
État d'un processus



État d'un processus



État d'un processus



Environnement

Les processus fonctionnent dans un environnement :

- Ils sont exécutés depuis un répertoire le *Working Directory* et toutes les actions en rapport avec le système de fichiers se font à partir de là.
- Ils sont la propriété d'un utilisateur et héritent de ses droits (avec possibilité d'escalade de droits).
- Ils ont une définition de *l'environnement linguistique* (langue, affichage des dates, ...)
- ...

La pluspart des informations sur l'environnement sont transmises par des variables héritées du processus qui les lance via des *variables d'environnement*

Variables d'environnement

- `PATH` : liste des répertoires où sont recherchés les exécutables
- `LD_LIBRARY_PATH` : liste des répertoires où sont recherchés les fichiers de bibliothèque au lancement des applications.
- `USER` : nom de l'utilisateur.
- `HOME` : répertoire de l'utilisateur sous Unix.
- `HOME_DRIVE` et `HOME_PATH` : disque et répertoire de l'utilisateur sous Windows.
- `PYTHONPATH`, `PYTHONHOME` pour la gestion de python (les *virtual env*).
- ...

Variables d'environnement (suite)

Ces variables peuvent être lues et modifiées sous Linux (`bash`)

```
# lire une variable
echo $VARIABLE
# lire toutes les variables
env
# modifier seulement pour une commande
VARIABLE=valeur commande
# exporter une variable dans l'environnement d'un
# processus pour affecter tous ses descendants
export VARIABLE=valeur
# ajouter l'export dans un script pour avoir un effet permanent
#   - pour un utilisateur
echo "export VARIABLE=valeur" >> $HOME/.bashrc
#   - pour tous les utilisateurs
echo "export VARIABLE=valeur" >> /etc/profile.d/mesvars.sh
```

Attention, modifier de manière permanente et erronée certaines variables comme `PATH` peut rendre le système inutilisable !

Variables d'environnement (suite)

Ces variables peuvent être lues et modifiées sous Windows

```
# lire une variable
echo %VARIABLE%
# lire toutes les variables
set
# exporter une variable dans l'environnement d'un
# processus pour affecter tous ses descendants
set VARIABLE valeur
# ajouter de manière permanente àl'utilisateur
setx VARIABLE valeur
# ajouter de manière permanente au système
setx /M VARIABLE valeur
```

Paramètres

Les processus ont des paramètres, ils permettent par exemple de spécifier les détails d'une commande.

- En ligne de commandes, les paramètres sont les mots passés après le nom de la commande. Par exemple :

  `cp fichierdepart /repertoire/arrive`
- Selon la commande, certains de ces paramètres sont obligatoires d'autre sont des options :
  - Pour `cp` « `-r` » est l'option « copier récursivement »c'est à dire le répertoire et tous les sous répertoires.

    `cp -r ./repertoire/depart /repertoire/arrive`
  - les options sont souvent reconnues grâce au caractère - sous linux et / sous windows.
  - elles peuvent aussi utiliser -- lorsque l'option est un mot de plusieurs caractères
  - les options peuvent prendre des paramètres `ls --sort size`
- Il existe des bibliothèques pour simplifier la lecture des paramètres. En python `argparse`.

Pour connaitre les fonctionnalités d'une commande, il existe la commande `man NOMDELACOMMANDE`

Communication entre processus

Les processus sont relativement isolés. Leurs codes ne peuvent pas se perturber car ils n'utilisent pas les mêmes espaces en mémoire. Mais il est parfois nécessaire de les faire communiquer.

- Pour prévenir d'un événement (une alarme) il y a les signaux.
- Pour échanger des données il y a des objets spéciaux proches des fichiers : *tube (pipe)*, *socket*, ...

Signaux

Définition (signaux)

Les signaux fonctionnent comme des alarmes, ils « dérangent » le processus pour lui faire exécuter une action prévue à l'avance. Il y en a un nombre fixe, prédéfinis.
On peut modifier l'action prévue mais ce n'est pas au programme.

Vous pouvez lancer un signal à un processus via la commande ou la fonction `kill`. Par exemple

```
# tue le processus de numéro 3452 en lui envoyant le signal 9
kill -9 3452
# stoppe tous les processus dont la commande est emacs en leur
# envoyant le signal SIGSTOP
killall -SIGSTOP emacs
```

Échange de données

L'échange de données est plus complexe, car il permet de communiquer toute forme de données entre processus sur des machines qui peuvent être distantes.
En TP vous utiliserez :

- Les *pipe*, uniquement locaux qui permettent de transmettre des données d'un processus à l'autre souvent via les entrées et sortie standards.
  `ls -l | grep toto`
  Le pipe « | » permet d'envoyer la sortie de la commande `ls` sur l'entrée de la commande `grep`
- Les *socket*, outils réseaux qui permettent à un processus donné de se connecter à un serveur distant pour lui transmettre des informations.
  `nc -l 8080` ouvre un serveur en écoute sur votre ordinateur, ce serveur accepte un client qui se connecte sur le port 8080
  `nc www.google.fr 80` cette commande se connecte sur le port 80 du serveur `www.google.fr` (c'est le serveur web).

La commande `nc`, en plus d'ouvrir la socket en mode `TCP`, permet d'envoyer tout ce qui est écrit sur son entrée standard et affiche tout ce qui est reçu via le réseau.

Notions définies par le système

En ce qui concerne les processus, les notions importantes sont :

- *Les processus* : un programme en train de tourner.
- *L'état du processus* : son status à un instant donné.
- *L'environnement* : la description du lien entre le processus et le système.
- *Les paramètres* : informations données au lancement du processus.
- *Les canaux de communications* : outils du système pour permettre l'échange de données.
- *Les signaux* : moyen pour un processus ou le système de prévenir et dérouter un autre processus.
- *Interblocage* : risque pour plusieurs processus qui interragissent de se bloquer l'un et l'autre.
- *Assynchronisme* : fait de ne plus voir un programme comme une suite d'opérations qui se suivent mais comme des réponses à des évennements.

Module python en rapport avec les processus

Plusieurs modules sont utiles pour la gestion des processus par python. *Mais il faut savoir que toute interraction avec le système est un risque pour la portabilité*.

- Toute opération nécessitant une action du système doit passer par un appel système.
- Au départ chaque système avait sa propre liste d'appel, avec des paramètres spécifiques ou des resultats divers.
- $\Longrightarrow$ il y a donc un soucis pour faire des appels système dans un langage comme python qui doit fonctionner sur des systèmes différents.
- Il y a eu une tentative de normalisation : les normes POSIX utilisées par les unixes.

Mais tous les systèmes n'ont pas choisis de l'implémenter totalement ou d'utiliser la même version de la norme.

Module python en rapport avec les processus (digression)

Il y a plusieurs grandes familles notamment : les unixes (linux, bsd, macos. . .) et les MS windows.
Par exemple pour créer un processus, la norme POSIX définie dans les année 70 imposait 2 appels système :

- un pour créer une copie du processus courant `fork` ;
- un autre pour remplacer son code par un nouveau `exec`.

Cela était le choix efficace à cause du peu de mémoire disponible.

Windows défini dans les années 80 a choisi de le faire en une seule étape pour les deux opérations `createprocess`.

Mais comment faire un module de langage qui implémente le `fork` ?

Module python en rapport avec les processus (fin)

PYTHON va donc proposer plusieurs modules pour la gestion des processus.

- Le module `os` propose `fork` mais ne fonctionnera que sous unix.
- Le module `win32process` propose `createprocess` mais ne fonctionnera que sous windows.

La solution est de définir des modules plus spécialisés avec les primitives réellement utiles au programmeurs et faisables quel que soit le système :

- `multiprocessing` va permettre de gérer les processus dans le context du calcul parrallèle
  - lancer un groupe de processus ;
  - gérer l'echange de données entre eux.
- `subprocess` va permettre de gérer les processus dans le cas de l'appel d'une commande
  - lancer une commande avec ses paramètres ;
  - gérer les entrées sortie et code d'erreur.

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

À quoi sert la mémoire ?

- Stocker le code du programme

À quoi sert la mémoire ?

- Stocker le code du programme
- Stocker le code des fonctions partagées

À quoi sert la mémoire ?

- Stocker le code du programme
- Stocker le code des fonctions partagées
- Stocker les variables globales (Tas)

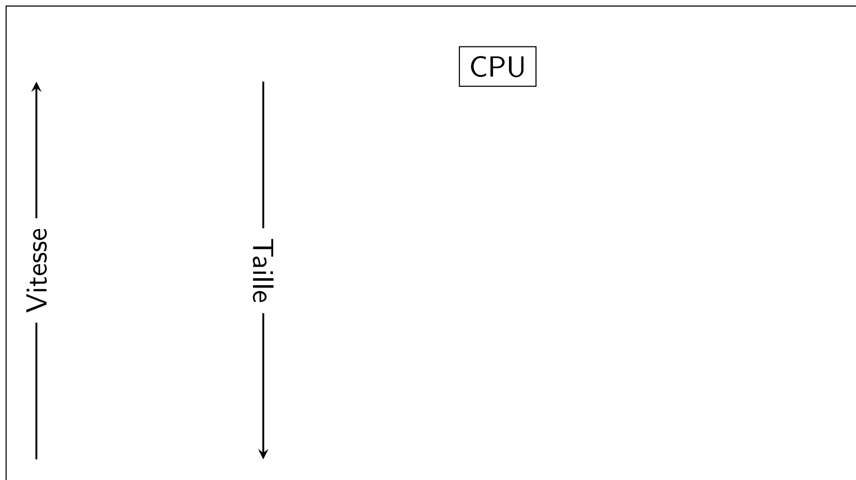
À quoi sert la mémoire ?

- Stocker le code du programme
- Stocker le code des fonctions partagées
- Stocker les variables globales (Tas)
- Stocker les variables locales (Pile)

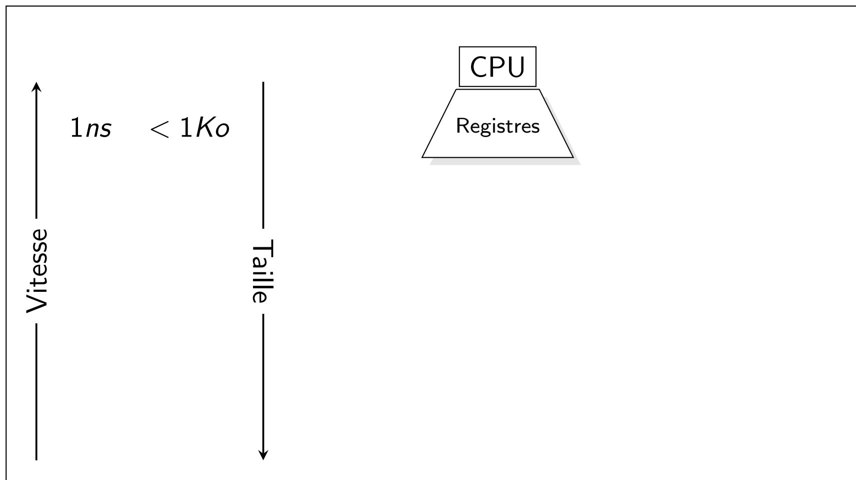
À quoi sert la mémoire ?

- Stocker le code du programme
- Stocker le code des fonctions partagées
- Stocker les variables globales (Tas)
- Stocker les variables locales (Pile)
- Une donnée pour pouvoir être traitée doit être dans un registre du processeur.

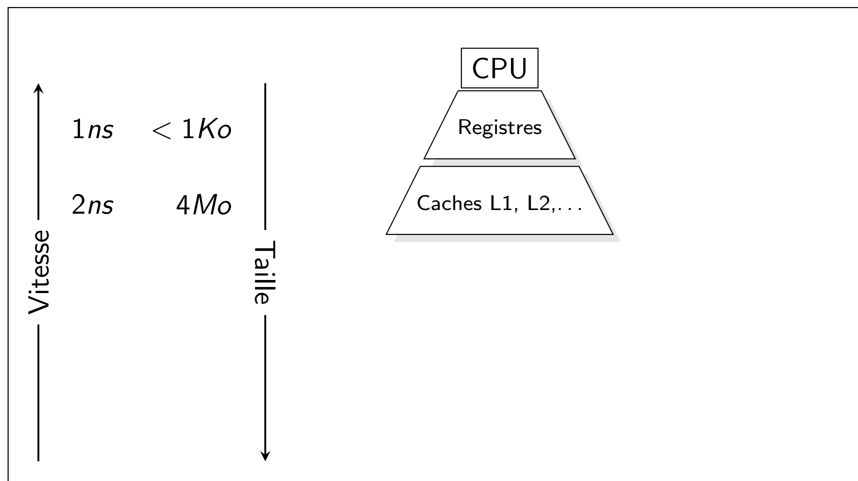
Hiérarchie des mémoires



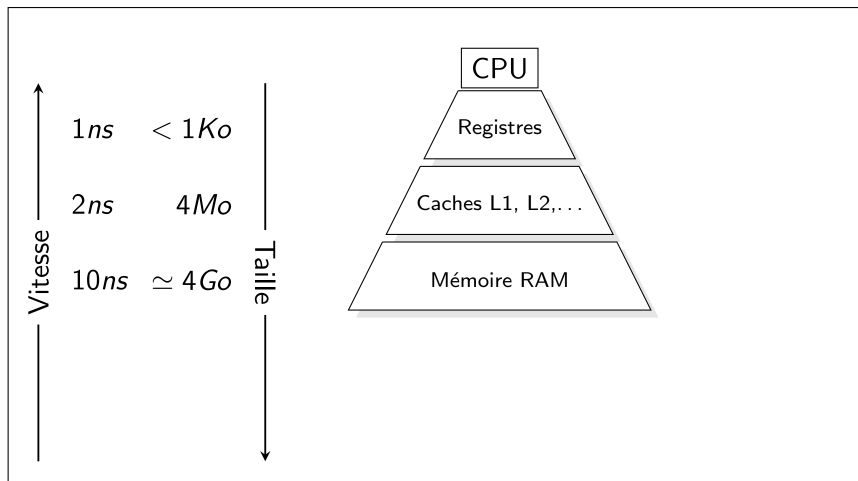
Hiérarchie des mémoires



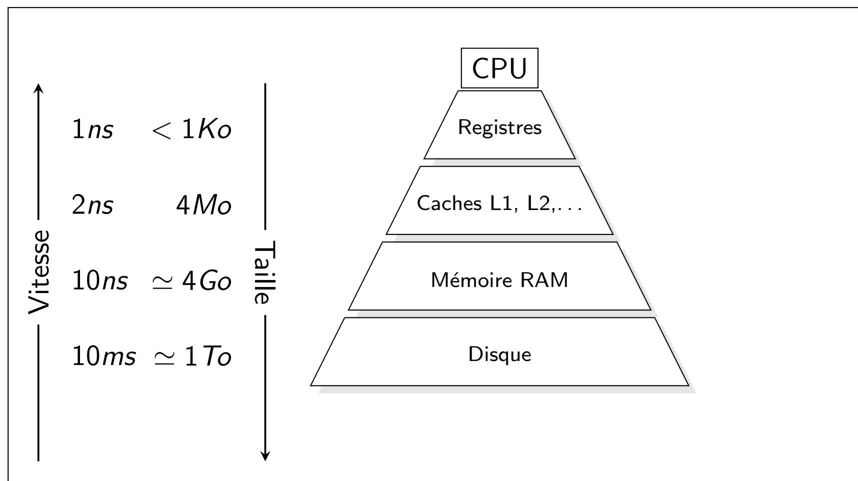
Hiérarchie des mémoires



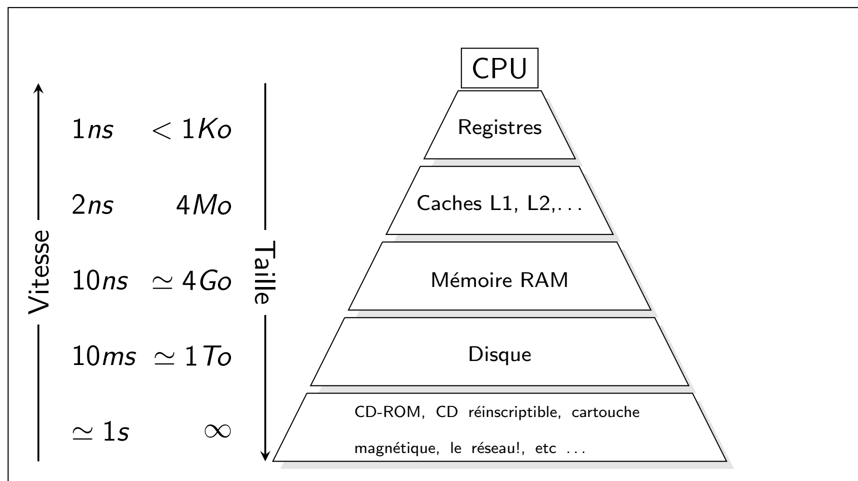
Hiérarchie des mémoires



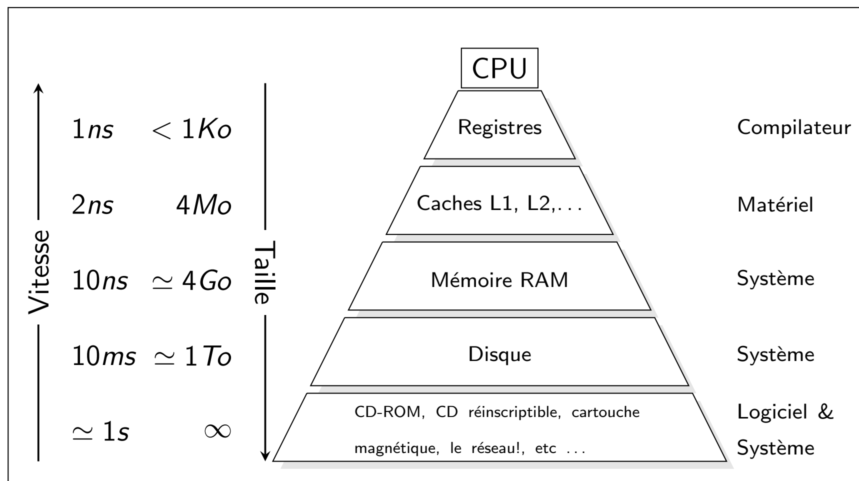
Hiérarchie des mémoires



Hiérarchie des mémoires



Hiérarchie des mémoires



Hiérarchie des mémoires

Vitesse		Taille		
$1ns$	$< 1Ko$	CPU / Registres	Compilateur
$2ns$	$4Mo$	Caches L1, L2,...	Matériel
$10ns$	$\simeq 4Go$	Mémoire RAM	Système
$10ms$	$\simeq 1To$	Disque	Système
$\simeq 1s$	∞	CD-ROM, CD réinscriptible, cartouche magnétique, le réseau!, etc ...	Logiciel & Système

Principe de localité : Les programmes tendent à utiliser des instructions et des données accédées dans le passé (localité temporelle) ou proches de celle-ci (localité spatiale)

Cercle *vertueux ?*

- Les programmeurs suivent naturellement le principe de localité.
- Donc ceux qui conçoivent les systèmes s'en aperçoivent et l'utilisent cette propriété (mécanismes de pages, overlay...).
- Donc pour garder de bonnes performances les programmeurs suivent ce principe.
- Donc ceux qui conçoivent les processeurs l'utilisent (pipeline, cache mémoire...)

Cercle *vertueux ?*

- Les programmeurs suivent naturellement le principe de localité.
- Donc ceux qui conçoivent les systèmes s'en aperçoivent et l'utilisent cette propriété (mécanismes de pages, overlay...).
- Donc pour garder de bonnes performances les programmeurs suivent ce principe.
- Donc ceux qui conçoivent les processeurs l'utilisent (pipeline, cache mémoire...)
- Donc optimiser un code revient souvent à améliorer sa localité
- ...
- Ce principe prend de plus en plus d'importance.

Le problème

- Chaque accès à une variable ou une fonction est un accès à la mémoire.
- Le compilateur doit faire la correspondance entre un nom qui est dans le code et une adresse demandée par les opérations du processeur.
- Mais le programme doit être portable, c'est à dire fonctionner sur des ordinateurs différents et en concurence avec d'autres processus.
- Le compilateur ne peux donc pas donner l'adresse de la case mémoire où se trouve réellement une variable.

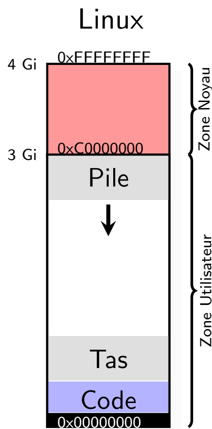
Historiquement, il y a plusieurs façon de régler le problème :

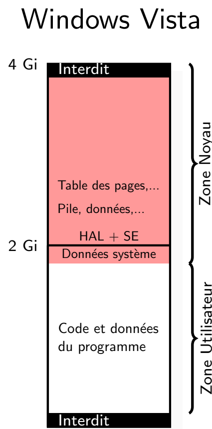
- translation d'adresse ;
- notions de segment ;
- mémoire virtuelle.

Mémoire virtuelle

- Idée : le processeur travaille avec des adresses mémoires différentes des adresses physiques.
- *Capacité d'adressage* : l'ensemble des adresses que l'on peut coder, par ex. 4Go sur un processeur 32 bits (0x00000000 - 0xFFFFFFFF), 16Eo pour un 64 bits.
- *Espace d'adressage* : partie utilisable par le processus.
- L'espace d'adressage est partitionné en segments dépendants du système d'exploitation.
- À chaque lecture mémoire, l'adresse physique est calculée.
- Ce calcul permet de faire des vérifications supplémentaires :
  - présence effective de l'adresse en mémoire ;
  - droits.
- Il faut un matériel dédié la *Memomy Management Unit*.

Espace d'adressage d'un processus





Pagination

Comment faire la correspondance par exemple sur un processeur 32 bits

- La MMU utilise une table de correspondance *@ logique* $\mapsto$ *@ physique*
- La table est stockée en mémoire
- Si on stocke tout, 2^{32} @ à conserver
- $\Rightarrow$ On rassemble les adresses en *pages* (par ex. 4096 octets)
- $\Rightarrow$ 1 Mo à stocker (par processus), la plupart du temps les cases de la table sont vides.
- Donc on utilise plusieurs niveaux
  Ob 0100 1110 0001 0101 1001 1001 0101 1001

Pagination

Comment faire la correspondance par exemple sur un processeur 32 bits

- La MMU utilise une table de correspondance *@ logique* $\mapsto$ *@ physique*
- La table est stockée en mémoire
- Si on stocke tout, 2^{32} @ à conserver
- $\Rightarrow$ On rassemble les adresses en *pages* (par ex. 4096 octets)
- $\Rightarrow$ 1 Mo à stocker (par processus), la plupart du temps les cases de la table sont vides.
- Donc on utilise plusieurs niveaux
  Ob 0100 1110 :0001 0101 1001 :1001 0101 1001

Pagination

Comment faire la correspondance par exemple sur un processeur 32 bits

- La MMU utilise une table de correspondance *@ logique* $\mapsto$ *@ physique*
- La table est stockée en mémoire
- Si on stocke tout, 2^{32} @ à conserver
- $\Rightarrow$ On rassemble les adresses en *pages* (par ex. 4096 octets)
- $\Rightarrow$ 1 Mo à stocker (par processus), la plupart du temps les cases de la table sont vides.
- Donc on utilise plusieurs niveaux
  Ob 0100 1110 :0001 0101 1001 :1001 0101 1001
  Ox 4e : 159 : 959

Pagination

Comment faire la correspondance par exemple sur un processeur 32 bits

- La MMU utilise une table de correspondance *@ logique* $\mapsto$ *@ physique*
- La table est stockée en mémoire
- Si on stocke tout, 2^{32} @ à conserver
- $\Rightarrow$ On rassemble les adresses en *pages* (par ex. 4096 octets)
- $\Rightarrow$ 1 Mo à stocker (par processus), la plupart du temps les cases de la table sont vides.
- Donc on utilise plusieurs niveaux

Ob 0100 1110	:0001 0101 1001	:1001 0101 1001
Ox 4e	: 159	: 959
`rep`	: `page`	: `offset`

rep index dans le répertoire des pages, *page* index dans la table des pages, *offset* position dans la page.

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages
- La page se trouve en mémoire physique à une adresse `0x?????000` le mot mémoire demandé est à l'adresse physique `0x?????959`

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages
- La page se trouve en mémoire physique à une adresse `0x?????000` le mot mémoire demandé est à l'adresse physique `0x?????959`
- Si on ne trouve pas la page, il y a interruption

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages
- La page se trouve en mémoire physique à une adresse `0x?????000` le mot mémoire demandé est à l'adresse physique `0x?????959`
- Si on ne trouve pas la page, il y a interruption
  - Si la page existe mais n'est pas en mémoire *Défaut de page*

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages
- La page se trouve en mémoire physique à une adresse `0x?????000` le mot mémoire demandé est à l'adresse physique `0x?????959`
- Si on ne trouve pas la page, il y a interruption
  - Si la page existe mais n'est pas en mémoire *Défaut de page*
  - Si la page n'existe pas ou est interdite le processus est signalé (`SIGSEG`)

Ex. de pagination (suite)

La MMU du pentium permet une table de page avec 2 indirections :

- Pour lire l'adresse `0x4e159959`
- La MMU contient un registre donnant l'adresse de la *table des répertoires de pages*.
- On trouve la table des pages à l'index `0x4e = 78` de la table des répertoires de page
- On trouve la page à l'index `0x159 = 345` de la table des pages
- La page se trouve en mémoire physique à une adresse `0x?????000` le mot mémoire demandé est à l'adresse physique `0x?????959`
- Si on ne trouve pas la page, il y a interruption
  - Si la page existe mais n'est pas en mémoire *Défaut de page*
  - Si la page n'existe pas ou est interdite le processus est signalé (`SIGSEG`)
- Pour changer de contexte, il faut changer le registre d'adresse du répertoire de page.

Remarque

- En réalité, plusieurs accès mémoire par accès demandé

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* ≪ Translation Lookaside Buffer ≫
  - Équilibre taille des tables/nombre d'indirections

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* ≪ Translation Lookaside Buffer ≫
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?
  - ...

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?
  - ...
- Les pages ne peuvent pas toutes être gérées de cette manière

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* ≪ Translation Lookaside Buffer ≫
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?
  - ...
- Les pages ne peuvent pas toutes être gérées de cette manière
  - besoin de pages contigue en mémoire (driver) ;

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* ≪ Translation Lookaside Buffer ≫
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?
  - ...
- Les pages ne peuvent pas toutes être gérées de cette manière
  - besoin de pages contigue en mémoire (driver) ;
  - accès directe pour le système ;

Remarque

- En réalité, plusieurs accès mémoire par accès demandé
  - Il faut un cache de traduction pour améliorer le *TLB* « Translation Lookaside Buffer »
  - Équilibre taille des tables/nombre d'indirections
- L'adresse d'une page n'utilise qu'une partie d'un mot mémoire (20 bits/32), les bits restant servent à stocker des informations :
  - La page peut-elle être mise en cache ?
  - La page existe ?
  - La page est-elle en accès lecture/écriture/exécution ?
  - La page est modifiée par rapport au disque (Dirty bit) ?
  - La page a-t'elle été accédée récemment ?
  - ...
- Les pages ne peuvent pas toutes être gérées de cette manière
  - besoin de pages contigue en mémoire (driver) ;
  - accès directe pour le système ;
  - ...

Avantage de la pagination

- Les processus ont tous un même espace mémoire.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.
  - Partage de zones mémoires contenant le code des bibliothèques partagées.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.
  - Partage de zones mémoires contenant le code des bibliothèques partagées.
- Copie à la demande de la mémoire d'un processus lors d'un `fork`.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.
  - Partage de zones mémoires contenant le code des bibliothèques partagées.
- Copie à la demande de la mémoire d'un processus lors d'un `fork`.
  - Le processus fils recopie la table des pages du processus père en marquant chaque page en lecture seule.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.
  - Partage de zones mémoires contenant le code des bibliothèques partagées.
- Copie à la demande de la mémoire d'un processus lors d'un `fork`.
  - Le processus fils recopie la table des pages du processus père en marquant chaque page en lecture seule.
  - C'est l'écriture qui déclenche réellement la copie.

Avantage de la pagination

- Les processus ont tous un même espace mémoire.
- Permet la séparation des processus.
- Permet le partage de mémoire :
  - Entre processus pour créer un moyen de communication.
  - Partage de zones mémoires contenant le code des bibliothèques partagées.
- Copie à la demande de la mémoire d'un processus lors d'un `fork`.
  - Le processus fils recopie la table des pages du processus père en marquant chaque page en lecture seule.
  - C'est l'écriture qui déclenche réellement la copie.
- Initialisation à zéro sur le même principe.

Multiprogrammation et mémoire

- En général, les cadres libres sont gérés de manières globales pour tous les processus.
- Une processus qui demande plus de mémoire aura plus de pages qui lui sont affectées.
- Il existe cependant des limites d'utilisation (ulimit, cgroups, ...)
- Comment être sur qu'un processus a suffisamment de mémoire ?

Quelle est la définition de « suffisamment de mémoire » ?

Ensemble de travail

Définition

Par le principe de localité, un processus sur une période courte utilise un sous ensemble limité de ses pages. C'est l'*ensemble de travail* (*working set*).

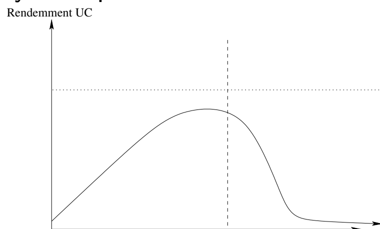
- Un processus qui accède à la mémoire va faire de nombreux défauts de page jusqu'à ce qu'il reconstitue son *espace de travail*.

Ensemble de travail

Définition

Par le principe de localité, un processus sur une période courte utilise un sous ensemble limité de ses pages. C'est l'*ensemble de travail* (*working set*).

- Un processus qui accède à la mémoire va faire de nombreux défauts de page jusqu'à ce qu'il reconstitue son *espace de travail*.
- Si le nombre de pages des ensembles de travail des processus prêts est supérieur à la mémoire physique, le système génère de nombreux défauts de pages. Il y a risque d'**écroulement** du système



Notions définies par le système

Les notions importantes sont

- *Principe de localité* : le fait qu'une donnée accédée est souvent proche dans l'espace ou le temps d'une données déjà accédée. C'est un principe à considérer lors de la programmation.
- *Mémoire* : mémoire disponible pour les processus.
- *Espace d'adressage* : ensemble des adresses que peut utiliser un processus.
- *Segment* : partie de l'espace d'adressage effectivement utilisé par un processus et ayant un rôle (système, code, données, . . .).
- *Droit sur les pages* : on attribu un rôle aux pages, si le processeur tente une utilisation en dehors du rôle, cela génère une exception traitée par le système.
- *Pages mémoire* : découpage de la mémoire en petits morceaux (les pages) qui sont affectées aux processus.
- *Swapping* : fait de déplacer une page de la mémoire au disque dur pour libérer de l'espace ou inversement pour donner des moyens de travail au processus.
- *Écroulement* : Lorsque le nombre de pages effectivement utilisées par l'ensemble des processus dépassent le nombre de cadres disponibles, le système va consacrer son temps à *swapper*.

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Les fichiers

L'une des tâches importantes du système est le stockage de l'information. Les fichiers permettent ce stockage.

- Ils ont une structure interne en relation avec le logiciel qui doit les utiliser.
- Ils ont souvent un type qui permet au système de savoir ce qu'il doit faire du fichier :
    - lancer un logiciel précis pour le lire ;
    - exécuter le fichier comme une commande ;
    - utiliser le fichier comme une bibliothèque de fonction ;
    - ...
- Ils sont classés dans une structure arborescente de répertoires.
- Un fichier est découpé en blocs de données qui sont les éléments que l'on peut lire ou écrire sur le disque.
- Le « disque » qui contient les fichiers est lui même organisé de manière à pouvoir retrouver le contenu du fichier à partir d'une structure de description (par exemple l'inode)

Type de fichier Unix

Traditionnelement on distingue 3 grands types de fichiers. Ils classe les fichiers selon la façon dont on peut les lire :

- Les *fichiers réguliers* ou *normaux* ceux qui contiennent des données, lisiblent par blocks.
- Les *catalogues* ou *répertoires*, les fichiers qui contiennent une liste de fichiers et servent a stocker la structure arborescente.
- Les *fichiers spéciaux*, ce sont des fichiers permettant d'accéder aux données gérées par le système sous la forme d'un fichier. Ils correspondent aux périphériques, socket, tubes...

Sous unix ≪ tout est fichier ≫, le système permet donc la lecture ou la modifications des objets du système via les appels systèmes de lecture et ecriture dans les fichiers (`open`, `read`,...). Les droits sont aussi gérés de la même manière.

Bien sur, le terme *fichiers réguliers* regroupent un grand nombre de choses différentes.

Typage des fichiers réguliers

Les *fichiers réguliers* ont différentes utilisations (exécutable, fichier texte, image, . . .) ; le rôle du système est de savoir quoi faire lorsque l'utilisateur demande une action sur le fichier.
Il y a 2 écoles :

- *Typage fort* : le fichier a un type défini par son nom (*extension*). C'est le cas par exemple sous DOS ou Windows.
- *Typage déduit* : le type du fichier dépend de son contenu ou de ses propriétés. C'est le cas sous Unix/Linux.
- *Type MIME* ou *Content-Type* : typage des données sur internet.
  - Le nom (donc l'extension) disparaît lors d'un envoi des données sur le réseau.
  - Les pages web ou les emails (pièces jointes) utilisent le type MIME écrit dans un entête.
  - Le navigateur ou le logiciel de lecture choisit le logiciel à appeler.

Typage fort

Tous les fichiers ont une extention qui peut être cachée mais est forcement présente.

- Un fichier exécutable doit se terminer par `.exe`, `.com` ou `.bin`.
- un script doit se terminer par `.bat`
- Pour le reste, le système reconnaît le logiciel permettant d'ouvrir le fichier en fonction de l'extension (tout est dans la base de registre `HKEY_CLASSES_ROOT`.

Typage déduit

Ce qu'il faut faire du fichier dépend de propriétées ou de sont contenu :

- Un fichier est présumé exécutable s'il a le droit d'exécution.
- On utilise souvent un code ou des directives placées en début de fichier :
  - le *nombre magique* au début du fichier **voir ici** ;
  - la première ligne des scripts comme `#! /usr/bin/env python3`
- Voir la commande `file` sous Unix.

En réalité, pour des raisons de compatibilités entre systèmes et pour éviter les problèmes, le typage est souvent mixte et la façon dont il est géré evolue souvent.

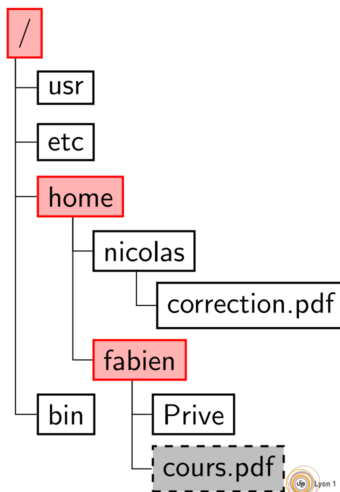
Certains fichiers sont en fait compatibles avec différents logiciels :

- un script python peut être à la fois un exécutable et un fichier de code ;
- un fichier `docx` ou `odt` est une archive ZIP contenant des fichiers de descriptions xml et les documents intégrés ;
- un fichier `jar` (exécutable java) est une archive contenant les fichiers `.class` et une description pour le lancement ;
- ...

L'organisation des répertoire

Les répertoires sont des fichiers spéciaux qui contiennent des fichiers ou d'autres répertoires

- Un répertoire ne fait partie que d'un autre répertoire (structure d'arbre).
- Chaque répertoire contient au moins deux autres : . lui même et .. son parent.
- Tout part de la racine :
  - Sous windows, il y a une racine par partition : `C:`, `D:`,...
  - Sous unix, il y a une racine unique : `/`.
- Pour accéder à un fichier on a besoin d'un chemin :
  - Chemin absolu `C:\User\fabien\cours.pdf`
  - Chemin relatif (par rapport au répertoire courant du processus qui utilise le nom) : `../nicolas/correction.pdf` pour un processus se trouvant dans `/home/fabien/`



Les fichiers sur le disque

Un disque n'est pas lu octet par octet mais par morceaux qu'on appelle *bloc*. Les fichiers sont alors découpés en blocs et écrits sur le disque.

Définition (Système de fichiers)

Le *système de fichiers* est la structure de données qui définit la façon de retrouver tous les blocs d'un fichier.

- Par exemple `NTFS`, `EXT`, `FAT`, `APFS`
- Chaque *partition* peut être formatée avec un système différent
- Les systèmes ont des propriétés différentes :
  - taille maximum des fichiers ;
  - vitesse d'accès selon les modèles d'utilisation ;
  - possibilité d'opération en ligne
  - ...

Processus et fichiers

- Les processus peuvent utiliser les fichiers :
  - les ouvrir/créer (`open`), lire ou écrire (`write`, `read`), les fermer `close` ;
  - explorer le système de fichiers avec les bibliothèques (`os` et `os.path`) ;
  - stocker des objets directement via des formats comme `JSON`.
- Ils sont toujours exécutés dans un *répertoire de travail* (parfois difficile à retrouver si on utilise un IDE).
- Ils ont toujours 3 fichiers spéciaux ouverts :
  - l'entrée standard `sys.stdin` ;
  - la sortie standard `sys.stdout` ;
  - la sortie d'erreur `sys.stderr`.

Notion de sérialisation

Le stockage de données dans un fichier pose souvent la question de sérialisation. C'est à dire ≪ Comment représenter dans un ensemble d'octets qui se suivent une structure de données présente en mémoire avec son organisation ? ≫.
Plusieurs possibilités sont utilisées pour résoudre le problème :

- méthode *ad-hoc* définie par le programmeur
- méthodes de sérialisation adaptées au langage (`serialyse` en php, `pickle` en python)
  - \+ capable de sérialiser facilement la plupart des objets ;
  - \- dépendant du langage voir plus.
- méthodes de sérialisation généralistes :
  - `XML` : +très expréssive, +lecture de flux, -gros surcout de données
  - `json, yml` : -juste l'essentiel (tableau, listes type primitifs), -il faut parser la totalité du fichier, +très compact.

Comment choisir ?

Résumé des notions sur les fichiers

- *Les fichiers* : ensemble de données que l'on peut lire et modifier de matière séquentielle ; associés à des flux dans les langage de prog.
- *Les répertoires* : fichiers spéciaux qui contiennent les noms et points d'accès à d'autres fichiers.
- *Les partitions* : disque ou partie de disque organisée de manière à pouvoir déposer les fichiers.
- *Le système de fichiers* : organisation des données sur le disque définie par le *formatage* de la partition. Il en existe plusieurs types en fonction de l'os ou des besoins des programmes.
- *Espace de nommage* : façons de représenter les fichiers sous une forme d'arbre (répertoire, sous repertoire...) afin de les retrouver.
- *Type de fichiers* : description du contenu qui permet d'associer à chaque fichier un programe ou une action.
- *Droits sur les fichiers* : liste d'actions autorisées ou refusées à un utilisateur ou groupe d'utilisateurs.

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.

Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.

Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.

Importance de la politique d'ordonnancement : cas d'école

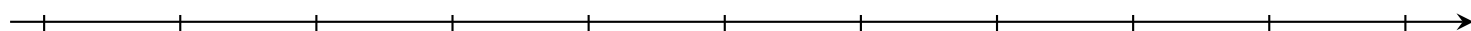
Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute

Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

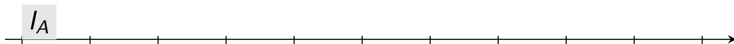
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
    - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

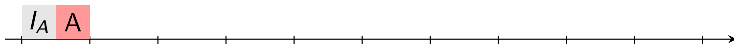
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

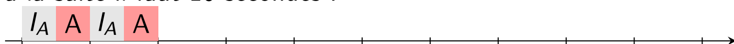
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

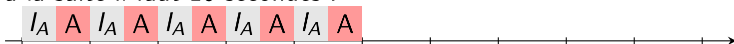
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

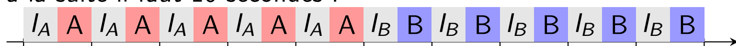
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

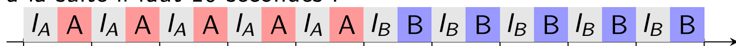
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

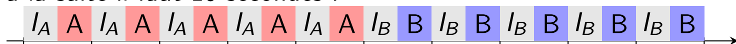
- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



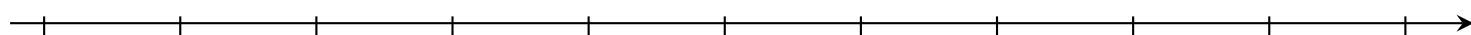
Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



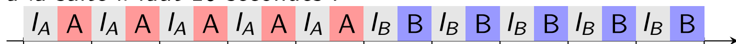
  - en même temps il faut $5,5$ secondes



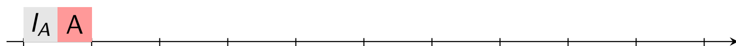
Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



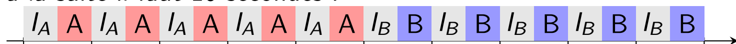
  - en même temps il faut $5,5$ secondes



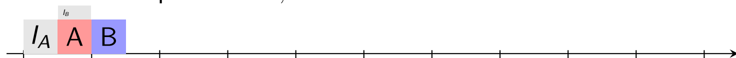
Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



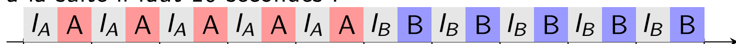
  - en même temps il faut $5,5$ secondes



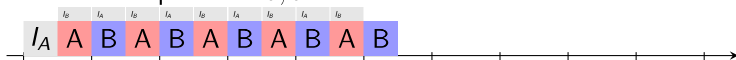
Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :



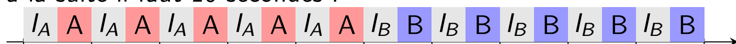
  - en même temps il faut $5,5$ secondes



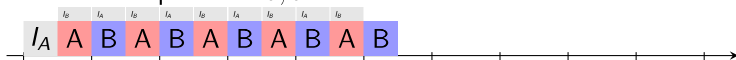
Importance de la politique d'ordonnancement : cas d'école

Le système doit gérer des tâches de calcul, et, à chaque instant, il doit choisir la tâche ayant accès au processeur.
Par exemple

- À un instant donné 2 processus (A et B) doivent s'exécuter.
- Chacun alterne $\frac{1}{2}$s de lecture sur le disque et $\frac{1}{2}$s de calcul.
- Chacun dure en tout 5 secondes.
- Si on les exécute
  - à la suite il faut 10 secondes :

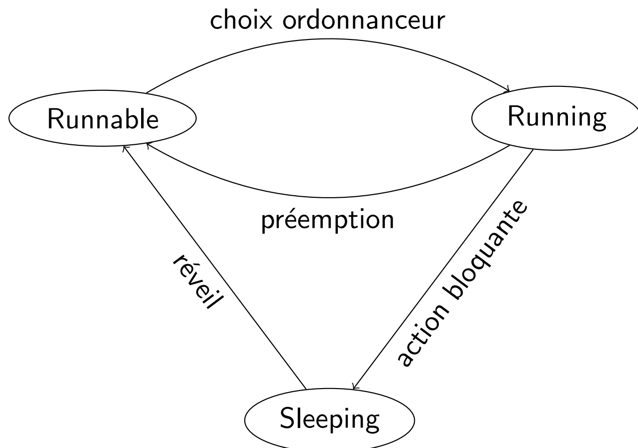


  - en même temps il faut $5,5$ secondes



- Si un processus est bloqué, il faut exécuter un autre processus

État de chaque processus



Un peu plus de réalisme

Idée simplifiée : ≪ Si un processus est bloqué, il faut exécuter un autre processus ≫. Pas si simple :

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.

Un peu plus de réalisme

Idée simplifiée : ≪ Si un processus est bloqué, il faut exécuter un autre processus ≫. Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.

Un peu plus de réalisme

Idée simplifiée : ≪ Si un processus est bloqué, il faut exécuter un autre processus ≫. Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).

Un peu plus de réalisme

Idée simplifiée : ≪ Si un processus est bloqué, il faut exécuter un autre processus ≫. Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)
  - Processus avec beaucoup d'entrées/sorties (analyse de disque)

Un peu plus de réalisme

Idée simplifiée : « Si un processus est bloqué, il faut exécuter un autre processus ». Pas si simple :

- Les tâches arrivent aléatoirement (sauf cas particulier ex. temps réel).
- On ne connaît pas *a priori* leur durée.
- Plusieurs tâches peuvent être disponibles, mais souvent aucune ne l'est.
- Elles ont des importances différentes (p. ex. swap).
- Elles peuvent avoir des dépendances.
- Il y a parfois des contraintes externes (pilote).
- Le comportement des tâches est différent :
  - Processus interactifs (court, temps de réponse important)
  - Processus avec beaucoup d'entrées/sorties (analyse de disque)
  - Long calculs (rare)

Ordonnancement

Définition (Ordonnancement)

La sélection dans le temps des processus ou threads qui peuvent accéder à un processeur est *l'ordonnancement*. Le but est de maximiser :

- Le débit (nombre de processus traités par unité de temps)
- Le taux utile (le taux du processeur effectivement par les tâches utilisateurs)

Il y a deux grands types d'ordonnancement en fonction de la possibilité d'interrompre une tâche :

- *Ordonnancement collaboratif* : les tâches ne sont pas interruptibles.
- *Ordonnancement préemptif* : le système peut interrompre une tâche à tout moment.

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame. . .
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame. . .
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;
  - Windows 95 Windows 98 utilisaient un mode collaboratif pour certaines tâches.

Préemptif vs collaboratif

- L'ordonnancement collaboratif est plus simple
  - Les tâches rendent la main lorsqu'elles sont finies ou lors de certains *appels systèmes particuliers* (ex `read()`, `sleep()`, `yield()` ou `SwitchToFiber()`)
  - Les tâches critiques sont protégées (écriture, périphérique)
  - Mais il est difficile d'assurer que le système reste réactif : si une tâche non interruptible entre dans une boucle infinie, là, c'est le drame...
  - $\Rightarrow$ non-adapté aux systèmes d'exploitation grand public
- L'ordonnancement préemptif évite ce problème
  - Une boucle infinie dans un programme ne compromet pas le système
  - Le système n'est pas bloqué par certaines tâches trop longues
  - Mais cela pose des difficultés pour les tâches critiques (driver, écritures)
- Par exemple :
  - MsDos et Windows 3.1 utilisaient un ordonnancement collaboratif ;
  - Windows 95 Windows 98 utilisaient un mode collaboratif pour certaines tâches.
  - Windows Nt, Xp et les suivants, MacOs, Unix, Linux...utilisent le préemptif.

Bonne politique d'ordonnancement

Il n'y a pas de politique d'ordonnancement meilleure que les autres, tout dépend du but

- PC Bureautique (web, traitement de texte,. . .)
  - But de l'ordonnanceur : donner une sensation de fluidité à l'utilisateur.
  - On privilégie les nouvelles tâches, celles qui n'ont pas eu accès au processeur pendant longtemps.
  - On désavantage les tâches de calculs long.
- Serveur de base de données
  - Les tâches ont des importances différentes
  - Les tâches peuvent s'exclurent (lecture en même temps qu'une écriture).
  - On peut privilégier certains groupes de tâches.
- Temps réel (multimédia, commandes d'une voiture, commandes de vol d'un avion...)
  - But de l'ordonnaceur : *garantir* que les traitements soient faits dans les temps.
  - Ensemble de calculs à réaliser et *deadlines* bien définis.
  - Il est nécessaire de pouvoir maîtriser la réalisation des tâches.

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Ordonancement sans préemption

- **FIFO** (*First In, First Out*) - on exécute les tâches dans l'ordre d'arrivée.
- **SJF** (*Shortest Job First*) - on exécute les tâches arrivées en choisissant la plus courte d'abord :
  - comme une queue à la poste (traitement des lettres avant les autres tâches) ;
  - nécessite de connaître leur durée ;
  - possibilité de *famine*.
- **À priorité** - les tâches ont des importances, on choisit la plus importante ("la plus prioritaire") :
  - comme une queue à l'aéroport (1ère classe > femme enceinte > famille, ...) ;
  - le classement en priorité a de grandes conséquences ;
  - possibilité de *famine*.

Dans tous les cas, une tâche longue qui a accès au processeur le bloque pendant toute sa durée.

Exemple

	T_0	T_1	T_2	T_3
Durée	3	7	2	1
Arrivée	0	1	2	2,5
Priorité	1	5	3	6

FIFO: arrivées T_0, T_1, T_2, T_3 ; exécution T_0 (0 à 3) ; axe 0, 5, 10, 15

SJF: arrivées T_0, T_1, T_2, T_3 ; exécution T_0 (0 à 3) ; axe 0, 5, 10, 15

Priorités: arrivées T_0, T_1, T_2, T_3 ; exécution T_0 (0 à 3) ; axe 0, 5, 10, 15

Exemple

	T_0	T_1	T_2	T_3
Durée	3	7	2	1
Arrivée	0	1	2	2, 5
Priorité	1	5	3	6

FIFO

T_0 T_1 T_2 T_3

T_0 T_1

0 5 10 15

SJF

T_0 T_1 T_2 T_3

T_0 T_3

0 5 10 15

Priorités

T_0 T_1 T_2 T_3

T_0 T_3

0 5 10 15

Exemple

	T_0	T_1	T_2	T_3
Durée	3	7	2	1
Arrivée	0	1	2	2, 5
Priorité	1	5	3	6

FIFO
T_0 T_1 T_2 T_3
T_0 | T_1 | T_2
0 5 10 15

SJF
T_0 T_1 T_2 T_3
T_0 | T_3 | T_2
0 5 10 15

Priorités
T_0 T_1 T_2 T_3
T_0 | T_3 | T_1
0 5 10 15

Exemple

	T_0	T_1	T_2	T_3
Durée	3	7	2	1
Arrivée	0	1	2	2, 5
Priorité	1	5	3	6

FIFO : T_0 (0–3), T_1 (3–10), T_2 (10–12), T_3 (12–13)

SJF : T_0 (0–3), T_3 (3–4), T_2 (4–6), T_1 (6–13)

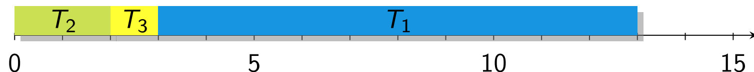
Priorités : T_0 (0–3), T_3 (3–4), T_1 (4–11), T_2 (11–13)

Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3
Durée	7	2	1
Arrivée	-2	-1	-0.5
Priorité	5	7	6



Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4
Durée	7	2	1	3
Arrivée	-2	-1	-0.5	2
Priorité	5	7	6	6

Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T_2'
Durée	7	2	1	3	2
Arrivée	-2	-1	-0.5	2	5
Priorité	5	7	6	6	7

- Il y a *famine*

Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T'_2
Durée	7	2	1	3	2
Arrivée	−2	−1	−0.5	2	5
Priorité	5	7	6	6	7

- Il y a *famine*
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus

Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T_2'
Durée	7	2	1	3	2
Arrivée	-2	-1	-0.5	2	5
Priorité	5	7	6	6	7

- Il y a *famine*
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus
- *Nouveau problème :* une tâche longue devient prioritaire puis bloque le processeur (mauvais temps de réponse).

Problème

Les deux dernières stratégies ne sont pas équitables :

- Si un processus est peu prioritaire (ou long pour SJF), il est bloqué par tous les autres

	T_1	T_2	T_3	T_4	T'_2
Durée	7	2	1	3	2
Arrivée	-2	-1	-0.5	2	5
Priorité	5	7	6	6	7

- Il y a *famine*
- *Idée de solution :* on peut augmenter la priorité des processus qui ne sont pas élus
- *Nouveau problème :* une tâche longue devient prioritaire puis bloque le processeur (mauvais temps de réponse).
- ⇒ il faut une préemption pour découper la tâche.

Algorithmes précédents

- FIFO ne peut pas être préemptif (la tâche qui serait élue en cas de préemption est déjà celle exécutée).
- Les algorithmes SJF et à priorité peuvent l'être. Par ex :

	T_1	T_2	T_3
Durée	10	2	1
Arrivée	0	1	2
Priorité	10	2	5

  - SJF

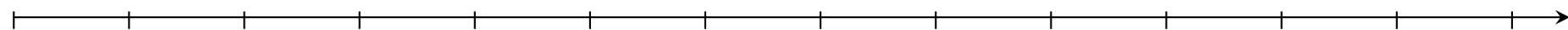
  - Avec priorités

Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

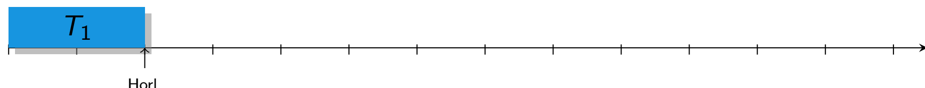


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1



Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

T_1 T_2

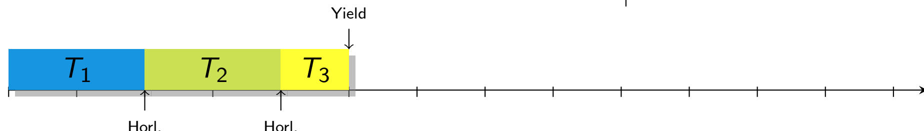
Horl. Horl.

Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison. . .).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

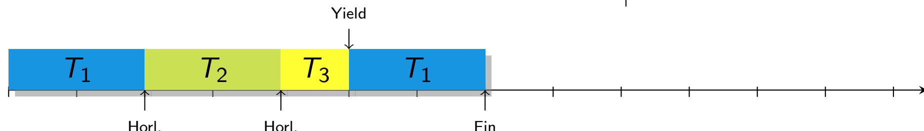


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

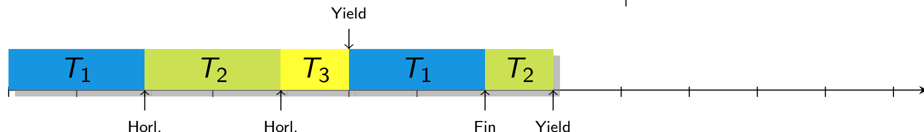


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison. . .).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

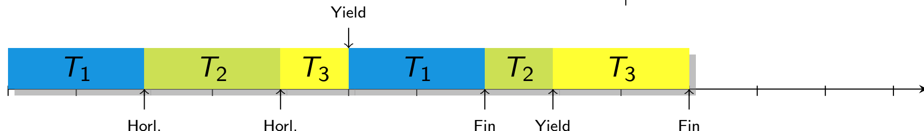


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

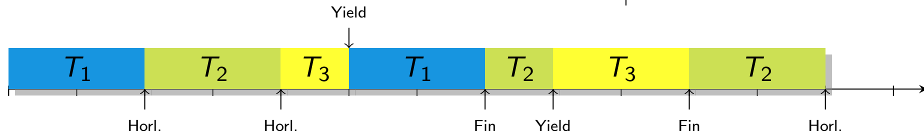


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1

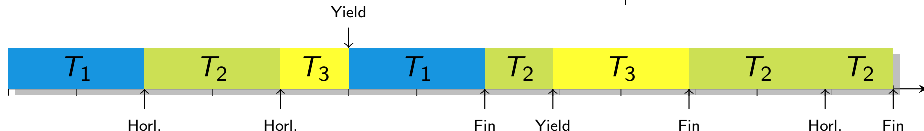


Tourniquet (Round Robin)

- On définit un *quantum de temps* : une durée maximum d'exécution.
- Les tâches sont élues selon leur ordre d'arrivée (FIFO).
- Elles s'exécutent jusqu'à :
  - expiration du quantum
  - ou suspension (yield) par la tâche elle-même (appel système, défaut de page, terminaison...).

Par exemple avec un quantum de $q = 2$:

	T_1	T_2	T_3
Durée	4	6	3
Arrivée	0	1	2
Yield	–	3	1



Mélanges d'algorithmes

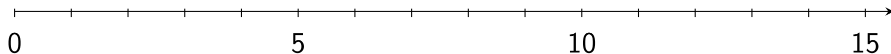
Priorité et Round Robin

- Le système gère plusieurs listes de tâches.
- Chaque liste correspond à une priorité.
- L'ordonanceur choisit une tâche dans la première liste non vide.
- Par exemple :
  - Processus critique (swap, I/O...).
  - Processus système.
  - Processus interactif.
  - Processus de calcul.
  - Tâche de fond.
- Pour une même liste, on utilise le **Round Robin**

Exemple

Avec un quantum de $q = 2$, ordonnancez (en RR) :

	T_1	T_2	T_3	T_4
Durée	6	3	4	3
Arrivée	0	2	3	8
Priorité	4	2	2	4



Contraintes

Pour le moment nous n'avons pas vu les dépendances entre tâches :

- Une tâche qui ne peut pas démarrer avant qu'une autre termine
  - Par exemple *producteur-consommateur* : certaines tâches consomment une ressource produite par d'autres et doivent donc attendre.
  - Plus précisément : quand deux processus dialoguent sur une socket, celui qui lit doit attendre que l'autre ait écrit quelque-chose.
- Deux tâches ne doivent pas s'exécuter en même temps (exclusion mutuelle).
  - Par exemple, deux écritures dans un fichier ne peuvent pas avoir lieu en même temps.

Le système doit proposer des moyens de bloquer des tâches.

Effet sur l'ordonnancement

Dès que l'on fait de la programmation multitâches, il faut tenir compte des contraintes

- Une tâche peut être ralentie car elle ne parvient pas à obtenir une ressource.
  - Exemple de *mars pathfinder* dont le système était réinitialisé suite à une *inversion de priorité*.
  - Exemple de votre ordinateur fortement ralenti lors d'une copie de disque.
- Des tâches peuvent se bloquer irrémédiablement lorsqu'elles demandent les même ressources : *dead-lock*

Dead Lock

	Voiture gauche	Voiture droite
Besoins	1 puis 2	2 puis 1

Figure – Rue à une voie

Si les deux voitures avancent, chacune bloque l'autre et ne peut plus la débloquer.
Un *dead-lock* apparaît :

- Lorsque des tâches réservent plusieurs ressources et que l'ordre de réservation permet un cycle.
- Lorsque des tâches utilisent une opération bloquante et attendent que les autres les débloquent.

Notions

- **Multitâche :** À tout moment, le système doit exécuter plusieurs tâches. Lorsqu'il y a concurence, il doit arbitrer.
- **Ordonancement :** Lorsqu'il y a un accès à une ressource limité (processeur, ...), le système doit choisir l'ordre dans lequel les tâches peuvent accéder à la ressource.
- **But de l'ordonnancement :** Il n'y a pas de bonne manière d'ordonnancer, il faut s'adapter aux besoins.
- **Priorité :** Les tâches peuvent avoir des importances différentes
- **Équité :** Le fait pour le système de permettre à chaque tâche d'accéder à la ressource de manière équitable (pas forcement judicieux).
- **Famine :** Le fait pour une tâche de ne pas pouvoir accéder à la ressource ou de pouvoir être retardée sur un temps arbitrairement long.
- **Dépendance entre tâches :** Le fait pour une tâche de devoir être éxécuté après la fin d'une autre ou ne pas l'être en même temps qu'une autre.
- **Dead lock :** Plusieurs tâches ayant des dépendances peuvent se bloquer irrémédiablement.

Plan

1. Introduction
   - Interface avec le matériel
2. Les processus
   - État d'un processus
   - Environnement
   - Communication entre processus
   - Les processus et python
3. Gestion de la mémoire
   - Rôle de la mémoire
   - Organisation de la mémoire
4. Fichiers et système de fichiers
5. Tâches et ordonnancement
   - Algorithmes d'ordonnancement
   - Avec préemption
   - Contraintes entre tâches
6. Configuration et services
   - Gestion des utilisateurs
   - Configuration du système
   - Rôles du système

Interface avec l'utilisateur

Que voit l'utilisateur ?

- Le logo au démarrage, et quelques bizarreries :

- Un système de configuration et d'installation.
- Un système de gestion des services et des logs.

Pour les utilisateurs, le système doit :

- imposer des limites (*e.g.*, droits d'accès) ;
- permettre la programmation « avancée » (*e.g.*, processus, client/serveur) ;
- fournir des outils pour administrer la machine.

Utilisateurs

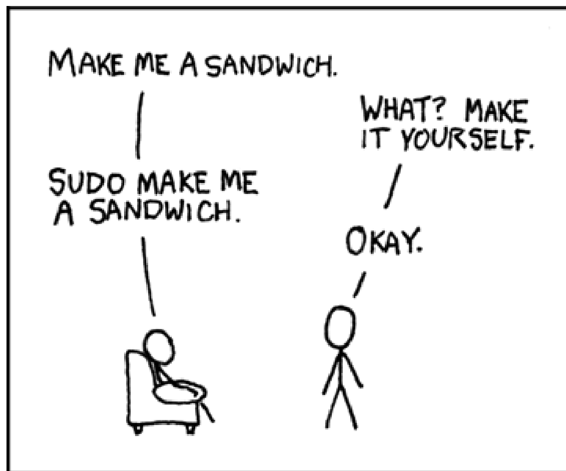
- Un ordinateur peut être accédé par plusieurs « utilisateurs » :
  - Ordinateur de la fac : tous les étudiants Lyon 1 + la DSI
  - Votre ordinateur personnel : vous (quand vous travaillez, ...ou pas !) et vous (quand vous installez des logiciels ou reconfigurez le système)
  - Un « utilisateur »(informatique) n'est pas forcément une personne physique !

Utilisateurs : cas d'utilisation

- Utilisateur d'un PC individuel (un seul utilisateur physique)
  $\rightsquigarrow$ Séparer les tâches quotidiennes et les tâches dangereuses. Interdire aux applications classiques (e.g. navigateur web) de modifier la configuration du système.
- Utilisateurs du PC familial (quelques utilisateurs qui se font confiance)
  $\rightsquigarrow$ Une configuration différente par utilisateur (exemple : fond d'écran différent, bookmarks du navigateur, ...)
- Utilisateurs d'un PC partagé (utilisateurs ne se faisant pas confiance *a priori* les uns aux autres)
  $\rightsquigarrow$ Gestion des droits (interdiction pour un utilisateur de lire/écrire les fichiers des autres, impossibilité d'agir sur un processus appartenant à un autre utilisateur, ...).

Tous les systèmes modernes pour PC (Linux, Windows, Mac OS) proposent du multi-utilisateur.

sudo



Utilisateurs : administrateurs et non-privilégiés

- Certains utilisateurs ont tous les droits : les administrateurs (`root` sous Unix/MacOS, `Administrator` sous windows).
  $\rightsquigarrow$ À n'utiliser que quand on en a vraiment besoin (ne jamais travailler en mode admin pour des tâches du quotidien comme naviguer sur le web, compiler un programme, ...). À ne pas utiliser pour ouvrir une session graphique.
- Les autres ne peuvent faire que ce qui leur est autorisé.
  $\rightsquigarrow$ À utiliser tout le temps !
- Pour passer administrateur temporairement :
  - Linux/MacOS : `sudo` *`commande`* ou `su -` sous Linux
  - Windows : clic droit → « run as different user »ou `runas.exe`

Droit

Chaque processus a un utilisateur, il ne peut faire que les actions autorisées à cet utilisateur.

- Sous windows les autorisations sont déclarées sous forme de liste de droits accordés ou refusés (ACL). Par exemple, pour un fichier :
  - droit de lecture, modification, contrôle total ...
  - les groupes ou utilisateurs qui ont le droit de faire ces actions.
- Sous unix, version plus ancienne et simpliste de ces listes :
  - 3 groupes : utilisateur propriétaire, groupe propriétaire, autres ;
  - 3 actions : lire, modifier, exécuter.
- Certains programmes ne sont pas éxécutés par l'utilisateur qui les a lancés, ils peuvent donc avoir plus de droits.
- Certains programmes doivent être exécutés par le compte administrateur pour fonctionner.

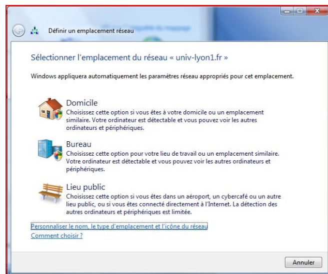
Configuration

- Le système conserve les configurations de la machine :
  - dans une base de données, *la base de registre* sous windows ;
  - dans des fichiers, ceux du *répertoire* `/etc` sous unix.
- Ces configurations disent :
  - comment démarrer ;
  - quels programmes lancer au démarrage *les services* ;
  - qui sont les utilisateurs ;
  - comment sont configurés les logiciels ;
  - ...

Modification des configurations

- L'idéal est de savoir directement modifier la base de configuration, mais ce n'est pas facile.
- Il existe des interfaces de configuration.
  - « Configuration du système »
  - « Panneau de configuration »
  - ...

  Mais elles ont un côté *boîte noire* et il n'y a pas d'uniformisation
- Il y a des assistants (*wizzard*), très efficaces pour faire une configuration courante mais impossible à adapter.



Installation

Les logiciels sont un ensemble de fichiers avec :

- des exécutables ;
- des bibliothèques de fonctions ;
- des configurations propres ;
- ...

Pour les installer, il y a souvent des *paquets*, c'est-à-dire

- des archives avec l'ensemble du nécessaire ;
- des script d'installation ou de désinstallation ;
- des dépendances.

Par exemple :

- sous Linux paquets `rpm` (Fédora/Redhat...) ou `deb` (Debian, Ubuntu) ;
- sous Windows paquets `msi` ;
- sous MacOS paquets `pkg` ;
- pour Python paquets `whl` pour pip.

Gestionnaire de paquets

On n'installe plus directement les paquets car :

- cela pose des problèmes de mise à jour ;
- cela pose des problèmes de gestion des dépendances.

Pour cela on utilise des gestionnaires de paquets qui sont capables :

- de trouver les logiciels sur des dépots publics ;
- de connaître et suivre les dépendances ;
- de gérer les mises à jour.

Par exemple `apt` pour Debian/Ubuntu et `pip` pour Python.

Surveillance du matériel

Le système surveille le matériel :

- pour vérifier les conditions d'utilisations (température,...) ;
- pour monitorer la charge (cpu, mémoire, I/O, ...) ;
- pour déclencher des actions automatiques (mise en veille, extinction des périphériques...) ;
- pour détecter des changements comme l'insertion d'un nouveau périphérique

L'utilisateur d'un ordinateur personnel peut configurer ces réactions

- gestion de la batterie ;
- action lors du branchement d'un téléphone sur une prise USB ;
- choix d'un profil de consommation CPU.

Actions automatiques

Le système doit s'adapter à l'environnement :

- connexions aux réseaux (authentification sur les réseaux wifi, récupération d'adresses IP, ...) ;
- mise à jour des logiciels (pour connaître et corriger les failles ou ajouter des fonctionnalités) ;
- synchronisation de l'heure ;

Ces actions sont souvent transparentes et ignorées par les utilisateurs, mais une mauvaise configuration peut avoir des conséquences :

- Si la date n'est pas correcte, les certificats web seront tous invalidés et donc la plupart des sites seront inutilisables.
- Si vous n'êtes pas sur le réseau local, vous ne pourrez certainement pas utiliser les périphériques locaux (imprimante, partages, ...).
- Une mauvaise gestion des connexions wifi permet à un pirate de lire vos données ou obtenir votre mot de passe.

Gestion des données

Le système peut avoir des actions pour préserver les données

- chiffrement d'un disques ;
- sauvegarde automatique, conservation d'un état journalier ;
- synchronisation d'un disque avec un serveur central ;
- conservation de la base de configuration pour « revenir à un point de restauration ».

De même ces services sont le plus souvent « offerts » par les éditeurs de systèmes d'exploitation à leurs utilisateurs. Ces derniers sont en général heureux de les avoir ou les ignore. Jusqu'au jour ou il y a un problème :

- perte des données irrécupérable car il y a chiffrement ;
- vol de données sur le serveur central ;
- mauvaise utilisation des points de restorations.

Sécurité informatique

Le système doit assurer un minimum de sécurité des communications :

- surveillance et limitation des logiciels (antivirus, gestion des droits,...) ;
- outils d'isolations (sandbox) ;
- pare feux ;
- base de certificats (qui permet aux logiciels de vérifier l'identité de sites distants) ;
- logs des évènements pour assurer la traçabilité.

Ces outils sont souvent totalement invisibles et peu de gens connaissent leur signification. Ce sont les administrateurs dans les entreprises ou les éditeurs des systèmes qui font des configurations génériques. Or, ces derniers ont souvent un choix a faire entre facilité d'utilisation et sécurité. Quelques exemple de choix couramment fait et douteux :

- désactivation de la vérification de certificats (risque de vol de mot de passe) ;
- services ouverts sans chiffrement avec un mot de passe unique dans une entreprise.

Notions

- **configuration :** ensemble de descriptions locales pour adapter un système à son environnement ou des logiciels aux systèmes.
- **configuration, interface de configuration, assistant :** ces 3 choses sont différentes, la configuration permet de faire n'importe quel choix mais reste compliquée à maitriser. L'interface est une aide graphique, mais elle est forcement limitée. L'assistant applique des configurations standardisées (comme le niveau 0 d'un support téléphonique).
- **paquets :** ensemble des fichiers et scripts de configuration nécessaires pour installer un logiciel.
- **Actions du système :** voir plus haut