Processus et interblocage

1. Définition

Un **processus** est un programme *en cours d'exécution*.

La distinction est importante :

- un programme est un fichier statique (sous forme de code source qui sera interprété, ou compilé en un exécutable binaire) ;
- un processus, c'est ce programme *« vivant »* en mémoire, avec ses ressources propres.

> ✅ **Caractéristiques d'un processus**
>
> Chaque processus est identifié par :
>
> - son **PID** (*Process IDentifier*) : un numéro unique attribué par l'OS à sa création ;
> - son **PPID** (*Parent PID*) : le PID du processus qui l'a créé (*processus parent*) ;
> - l'**utilisateur** qui l'a lancé ;
> - les **ressources** qu'il consomme : mémoire vive, temps processeur.

2. Création et hiérarchie des processus

Sous GNU/Linux, tout processus est créé par un autre processus *via* un mécanisme appelé ***fork*** (littéralement : fourche). Le processus parent se divise en deux — il continue d'exister, et donne naissance au processus enfant qui vit alors sa propre vie.

> 💭 **Remarque : fork ??? ▼**
>
> En NSI, on ne demande aucune connaissance sur `fork()` (qu'on rencontre en C, langage dans lequel le noyau Linux est majoritairement écrit). Il suffit de comprendre l'idée d'arborescence et de parenté entre processus.

Il en résulte une **arborescence** de processus, dont la racine est `init` (ou `systemd` dans les distributions modernes), lancé par le noyau au démarrage avec le PID 1.

```
init (PID 1)
  │
  ├── bash (PID 412)
  │     │
  │     └── python3 (PID 891)
  │
  ├── firefox (PID 554)
  │
  └── ...
```

On peut visualiser cette arborescence avec la commande `pstree`.

3. États d'un processus

Le processeur ne peut exécuter qu'un seul processus à la fois (par cœur). Un processus ne s'exécute donc pas forcément en continu, autrement on ne pourrait pas écouter de la musique en rédigeant un document. L'OS (*Operating System*) alloue à chaque processus des **« tranches de temps »** (*time slices*) très courtes. Comme il fait alterner *(vraiment) très vite* les processus, l'utilisateur a *l'impression* que plusieurs processus fonctionnent en parallèle.

Tout l'enjeu est de trouver *comment* alterner *intelligemment* entre plusieurs processus concurrents : c'est l'**ordonnancement** (*scheduling*).

💭 **Remarque : ordonnanceur ▼**

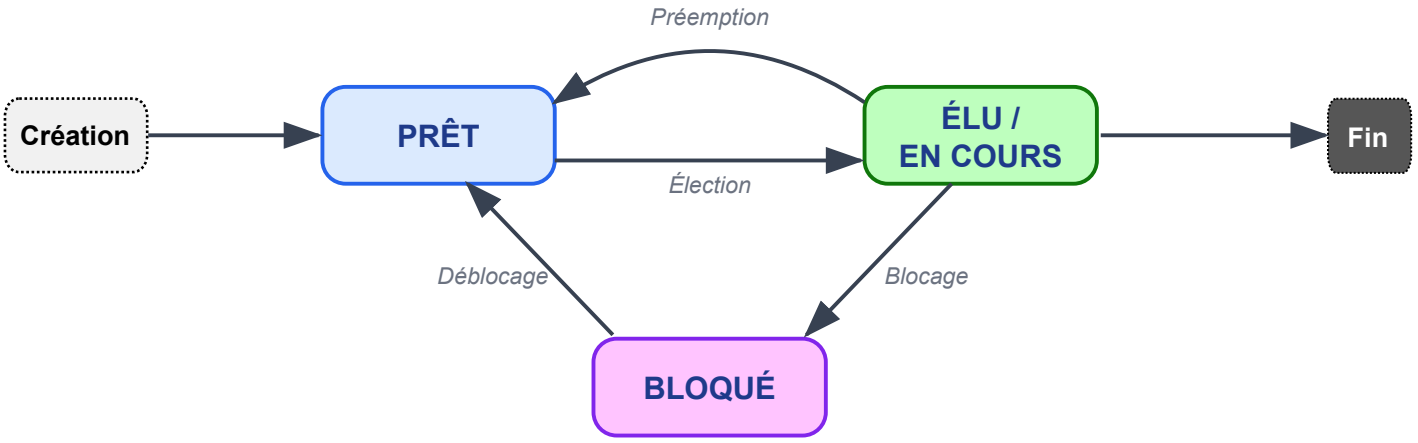
Pour la culture

L'ordonnanceur est un programme à part, chargé de l'ordonnancement. Son rôle est devenu de plus en plus important avec l'avènement des machines multi-processeurs, puis des processeurs multi-cœurs.

C'est spécialement vrai pour les serveurs, qui sont souvent des machines multi-processeurs, chaque processeur étant désormais multi-cœurs.

Savoir comment répartir efficacement la charge de travail sur les *millions* d'unités de calcul des super-calculateurs (des méga-machines qui peuvent rassembler des dizaines de milliers de serveurs) est d'une importance **critique !**

À chaque instant, un processus peut se trouver dans un état précis parmi 5 états possibles. Ces états sont reliés entre eux par 6 transitions, selon le schéma ci-dessous :



Les 5 états

État	Signification
Création	Le processus vient d'être créé par le système. Il n'est pas encore prêt à s'exécuter : l'OS lui alloue de la mémoire et initialise ses structures de données.
Prêt (*ready*)	Le processus a toutes les ressources nécessaires et n'attend plus que le processeur. Il est dans la file d'attente de l'ordonnanceur, qui choisira son tour d'exécution.
En cours ou **élu** (*running*)	Le processus est en cours d'exécution : l'ordonnanceur lui a attribué le processeur. À un instant donné, sur un cœur, un seul processus peut être dans cet état.
Bloqué (*waiting*)	Le processus est en attente d'un événement externe : une entrée/sortie (saisie au clavier — pensez à ce qu'il se passe avec un `input()` en Python — lecture disque, trame réseau…), un accès à une ressource occupée, ou un signal. Il ne consomme pas de CPU.
Terminé (*terminated*)	Le processus a terminé son exécution (normalement ou suite à une erreur). L'OS libère les ressources qu'il utilisait (mémoire, descripteurs de fichiers…).

Les 6 transitions

Transition	De → Vers	Description
(entrée)	Création → Prêt	Une fois initialisé, le processus est placé dans la file des processus prêts et attend son tour.
Élection	Prêt → En cours	L'ordonnanceur choisit ce processus parmi tous les processus prêts et lui accorde le processeur. Le choix dépend de la politique d'ordonnancement (priorité, tourniquet…).
Préemption	En cours → Prêt	Le SE retire le processeur au processus en cours, par exemple à la fin d'un quantum de temps (tourniquet) ou parce qu'un processus plus prioritaire est prêt. Le processus retourne dans la file sans avoir terminé.
Blocage	En cours → Bloqué	Le processus demande une ressource indisponible (ex : lire un fichier sur disque, attendre une réponse réseau). Il se met en attente volontairement et libère le processeur.
Déblocage	Bloqué → Prêt	L'événement attendu s'est produit (données reçues, ressource libérée…). Le processus est remis dans la file des prêts ; il ne reprend pas immédiatement le processeur.
Fin	En cours → Terminé	Le processus a exécuté sa dernière instruction (ou a été tué — voir la commande `kill`). L'OS récupère toutes ses ressources et supprime son entrée dans la table des processus.

4. Observer les processus

4.1. ps : instantané

La commande `ps` (*process status*) affiche un instantané des processus actifs.

```
ps            # Processus de l'utilisateur courant dans le terminal
ps -fau       # Vue détaillée de tous les processus (avec arborescence)
```

Colonnes importantes : `PID`, `PPID`, `USER`, `%CPU`, `%MEM`, `COMMAND`.

4.2. top / htop : vue dynamique

`top` et `htop` affichent les processus en temps réel, triés par consommation CPU ou mémoire.

```
top           # Vue dynamique (quitter avec Q)
htop          # Version améliorée et plus lisible (si installé)
```

💭 **Remarque : tuer un processus ? ▼**

Si un processus est bloqué ou consomme trop de ressources, on peut forcer son arrêter avec les commandes suivantes.

```
kill PID            # Envoie un signal d'arrêt propre (SIGTERM)
kill -9 PID         # Arrêt forcé immédiat (SIGKILL) — à utiliser en dernier
```

Sur une machine *partagée* (plusieurs utilisateurs connectés à la même machine), on **ne** peut **pas** tuer les processus des autres utilisateurs… sauf si on est `root`, bien sûr ! Et avec un grand pouvoir viennent de grandes responsabilités 😉 !

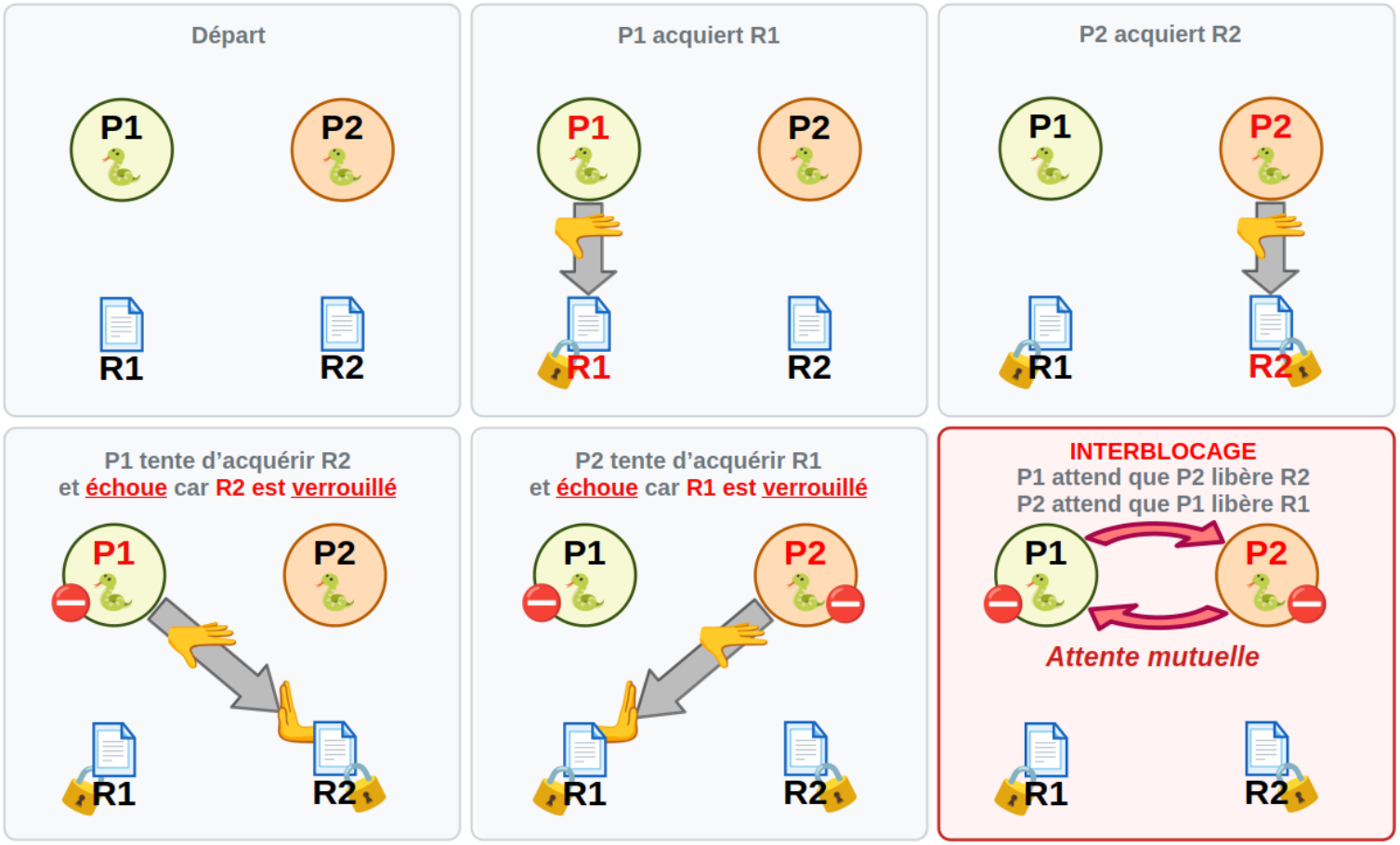
5. Interblocage (deadlock)

5.1. Définition

Un **interblocage** (ou *deadlock*) survient quand deux processus (ou plus) s'attendent mutuellement indéfiniment, chacun détenant une ressource dont l'autre a besoin.

Résultat : aucun des processus ne peut avancer. Le système se fige, sans erreur visible.

Illustration



Deux processus P1 et P2, deux ressources R1 et R2 (par exemple, deux fichiers ouverts en écriture exclusive) :

1. P1 demande et obtient R1.
2. P2 demande et obtient R2.
3. P1 *tente* d'acquérir **R2** et *échoue* car **R2 est verrouillé**.
4. P2 *tente* d'acquérir **R1** et *échoue* car **R1 est verrouillé**.
5. P1 attend que P2 libère R2, P2 attend que P1 libère R1, mais…
6. … **ni l'un ni l'autre ne libère quoi que ce soit → INTERBLOCAGE 💀 !**

5.2. Analogie

Deux personnes petit-déjeunent : l'une prend le beurre, l'autre la confiture, et chacun tente de prendre ce que l'autre a déjà, sans jamais lâcher ce qu'il tient.

Une analogie *inadaptée* serait celle de deux automobilistes têtus s'engageant sur un pont à sens unique : ni l'un ni l'autre ne peut avancer, et aucun ne veut reculer. Dans cette situation, une même ressource (le pont) aurait été acquise par deux processus (les automobilistes) au « même » moment. L'ordonnanceur d'un OS garantit que cela ne peut pas arriver…

5.3. Conditions nécessaires (Coffman, 1971)

Pour la culture (ce point est hors-programme).

Un interblocage ne peut se produire que si les quatre conditions suivantes sont *simultanément* réunies :

1. **Exclusion mutuelle** : une ressource ne peut être utilisée que par un seul processus à la fois.
2. **Possession et attente** : un processus détient au moins une ressource et en attend une autre.
3. **Pas de préemption** : on ne peut pas forcer un processus à libérer une ressource.
4. **Attente circulaire** : il existe un cycle d'attente entre les processus.

> ⚠️ **Comment l'éviter ?**
>
> Il suffit de briser *une* des quatre conditions. En pratique :
>
> - imposer un **ordre d'acquisition** des ressources (toujours prendre R1 avant R2) ;
> - utiliser des **délais d'attente** (*timeouts*) : si une ressource n'est pas disponible pendant n secondes, on libère ce qu'on détient et on réessaie.

6. Liens avec d'autres notions

- L'**ordonnancement** est au cœur de la gestion des processus par l'OS.
- Les **ressources** partagées (fichiers, mémoire, réseau) sont la source des interblocages.
- La **complexité** intervient dans l'évaluation du coût des algorithmes d'ordonnancement.