Chapitre 14

Gestion des processus et ressources

1 Systèmes d'exploitation multitâches

De nos jour les OS permettent d'exécuter plusieurs programmes en même temps :

- jouer à un jeu vidéo ;
- parler avec ses amis sur Discord ;
- utiliser un logiciel de streaming de type OBS ;
- et d'autres choses encore...

Lorsqu'on ouvre le gestionnaire des tâches de WINDOWS, voici ce qu'on obtient
Gestionnaire des tâches WINDOWS

Nom	PID	Statut	Priorité de base	Nom d'utilisateur	Temps pro...	Mémoire (plage de travail privée active)	Threads
ModernWarfare.exe	3268	En cours d'exécution	Haut	UGLi	00:03:38	4 595 776 Ko	94
MsMpEng.exe	4208	En cours d'exécution	Normal	Système	00:38:36	195 136 Ko	66
Discord.exe	11308	En cours d'exécution	Normal	UGLi	00:00:04	112 780 Ko	61
texstudio.exe	10220	En cours d'exécution	Normal	UGLi	00:00:11	102 604 Ko	6
Battle.net.exe	8448	En cours d'exécution	Normal	UGLi	00:00:05	93 412 Ko	21
Discord.exe	9788	En cours d'exécution	Normal	UGLi	00:00:03	82 948 Ko	45
obs64.exe	3596	En cours d'exécution	Normal	UGLi	00:00:05	56 528 Ko	40
SumatraPDF.exe	5868	En cours d'exécution	Normal	UGLi	00:00:01	43 812 Ko	6
Battle.net.exe	8068	En cours d'exécution	Normal	UGLi	00:00:02	43 172 Ko	58
SearchIndexer.exe	7772	En cours d'exécution	Normal	Système	00:01:02	41 700 Ko	76
explorer.exe	7852	En cours d'exécution	Normal	UGLi	00:00:12	40 268 Ko	120
Secure System	104	En cours d'exécution	N/D	Système	00:00:00	40 132 Ko	-
Discord.exe	12316	En cours d'exécution	Normal	UGLi	00:00:00	33 076 Ko	32
Battle.net.exe	9308	En cours d'exécution	Supérieure à la no...	UGLi	00:00:02	29 428 Ko	44
Taskmgr.exe	8336	En cours d'exécution	Normal	UGLi	00:00:03	24 208 Ko	23
Discord.exe	14524	En cours d'exécution	Supérieure à la no...	UGLi	00:00:00	23 688 Ko	33
Ighub_agent.exe	7944	En cours d'exécution	Normal	UGLi	00:02:14	21 064 Ko	109

Une partie des tâches en cours sur mon ordinateur ;

Peu importe le nombre de cœurs du CPU, la multitude des programmes en cours d'exécution montre que l'OS dispose d'un système pour gérer ces exécutions simultanées.
Puisque tous les programmes ne peuvent « tourner en même temps » on parle d'*exécution concurrente*.

2 Les processus

2.1 Exécution d'un programme seul (version théorique et simplifiée)

Un *exécutable* est un fichier qui contient des instructions en langage machine.
Lors de son ouverture

- l'OS charge le programme dans la RAM, à une certaine adresse-mémoire ;
- il écrit cette adresse dans le registre PC (*program counter*) du CPU.

À chaque cycle d'horloge, le CPU lit l'instruction à l'adresse courante, l'exécute, puis passe à l'instruction suivante jusqu'à la dernière.

2.2 En réalité

Dans le modèle précédent, un seul programme peut être lancé et l'OS doit attendre qu'il « rende la main » pour continuer.

Il n'est pas possible de lancer 2 programmes ou plus en même temps, on a affaire à un OS *monotâche*... Or nos OS actuels sont clairement multitâches.

2.3 Multitâche coopératif

Il consiste à laisser les programmes décider du moment où ils doivent rendre la main aux autres. Celui-ci pose cependant deux principales difficultés :

- il faut que le multitâche soit pris en compte lors de l'écriture des programmes ;
- en cas d'erreur le système tout entier peut se retrouver bloqué.

C'est ce type de multitâche que l'on retrouve dans Windows 3.1 (1993) ou Mac OS 9 (1999).

2.4 Multitâche préemptif

Il consiste à octroyer un certain temps d'exécution au programme avant de reprendre la main de force en sauvegardant l'état du processus, au moyen d'une interruption programmable.
Ce type de multitâche se retrouve dans les systèmes d'exploitation Unix (1969) et aussi dans les systèmes plus grand-public depuis Windows 95 (1995) et Mac OS X (2001).

C'est sur la 2^{e} approche que nous allons nous concentrer.

2.5 Interruption

C'est un signal envoyé au CPU lorsqu'un évènement se produit : un disque dur a terminé d'écrire des octets, un clavier signale qu'une touche est pressée, *et cætera*.

Lors d'une interruption, le CPU arrête l'exécution du programme en cours.
Celle-ci est communiquée à un programme appelé *gestionnaire d'interruptions*, accompagnée d'une copie de l'ensemble des valeurs des registres du CPU.
En fonction de l'interruption, le gestionnaire passe la main à un autre programme.

2.6 Les interruptions d'horloge

Le CPU génère de lui-même des interruptions à intervalles de temps fixe.
De nos jours un CPU moderne les génère à une fréquence de 10 MHz, donc toutes les 100 nanosecondes (un dix-millième de milliseconde).

Ce sont elles qui permettent d'exécuter les programmes de manière concurrente.

Définition : exécutable

> Fichier binaire contenant des instructions en langage-machine directement exécutables par le CPU.

Définition : processus

> Un des programmes en cours d'exécution. En général il est décrit par
>
> - un *identifiant* unique ;
> - son *état* (effectivement en cours d'exécution, en attente ...) ;
> - la *mémoire* allouée par l'OS au programme ;

- les *ressources* utilisées par le programme (fichiers, connexions réseaux, matériels *et cætera*) ;
- les *valeurs des registres* du CPU.

Définition : PCB

Pour chaque processus, l'OS stocke sa description dans une structure de données appelée *PCB*, pour *Process Control Bloc*.

Définition : exécution concurrente

Deux processus s'exécutent de manière concurrente si les intervalles de temps entre le début et la fin de leur exécution ont une partie commune.

On ne parlera pas d'exécution *parallèle* (lorsque deux processus s'effectuent *réellement* en même temps).

2.7 Arborescence des processus

En général (c'est le cas avec les OS de type UNIX), l'OS crée au démarrage un processus de base qui à son tour va créer d'autres processus, appelés *processus fils*, et ainsi de suite.

Cela donne lieu à une *arborescence de processus*.

2.8 Processus légers ou thread

Les *threads* sont des processus particuliers crées par des processus qui se veulent eux-mêmes multitâches.
La principale différence entre thread et processus est que contrairement aux processus qui *a priori* sont isolés les uns des autres, les threads *partagent des variables globales*.
La commutation de contexte entre threads (voir plus loin) est également plus rapide qu'entre processus.

3 L'ordonnanceur

C'est le composant de l'OS chargé de *choisir l'ordre d'exécution* des différents processus.
De son point de vue les threads sont des processus à part entière.

Exemple de fonctionnement

1. un processus p1 est en cours d'utilisation ;
2. une interruption d'horloge survient ;
3. le *gestionnaire d'interruption* (GI) est appelé et sauvegarde l'état des registres du CPU à un endroit particulier de la mémoire ;
4. il fait appel à l'*ordonnanceur* qui *décide* à quel autre processus p2 il veut passer la main ;
5. le GI restaure les valeurs des registres du CPU qui on été sauvegardées la dernière fois que p2 a été interrompu ;
6. le GI rend la main : p2 reprend.

Les phases **2.** à **5.** s'appellent une *commutation de contexte*.

3.1 États d'un processus

Un processus peut parfois être de lui-même en pause (`sleep` en Python).
De même il peut être en attente d'une ressource : par exemple lors de l'appel de `input`, le programme sollicite le clavier et attend une réponse.
Parfois aussi, des erreurs peuvent se produire (programme mal codé, problème matériel…).

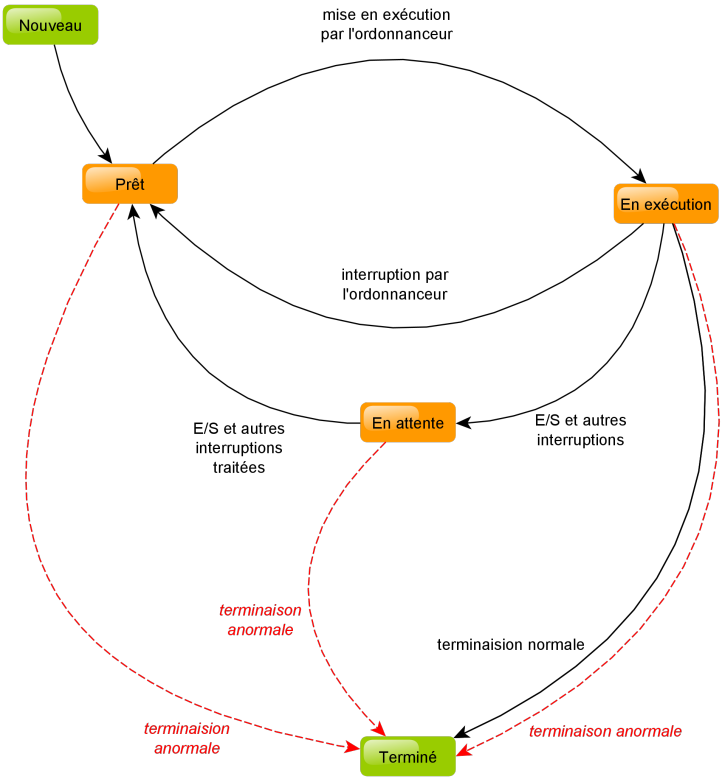
Toutes ces conditions amènent l'OS à définir l'*état* de chaque processus.

Les différents états

- **Nouveau :** état éphémère : l'OS vient de copier l'exécutable en mémoire et créé son PCB ;
- **Prêt :** le processus est dans l'ensemble des processus qui peuvent être choisis par l'ordonnanceur pour être exécutés : il attend son tour ;
- **En exécution :** le processus est effectivement en train de s'exécuter ;
- **En attente :** le processus est interrompu et est en attente d'un évènement externe : qu'une allocation mémoire soit effectuée, qu'une E/S soit activée, qu'un des périphériques dont il a besoin soit libre…
- **Terminé :** état éphémère : l'OS va libérer la mémoire allouée au processus ainsi que les ressources qu'il utilise et effacer son PCB.

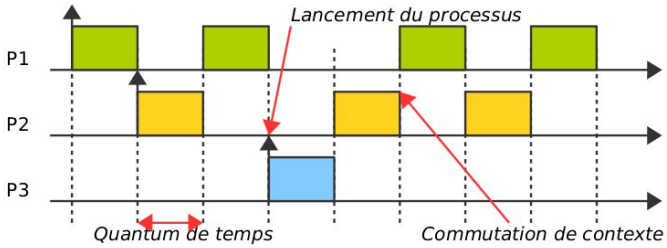
À tout moment une erreur peut conduire à la terminaison du processus.

3.2 Cycle de vie d'un processus



4 Les stratégies d'ordonnancement

4.1 Quel algorithme utiliser ?



On observe ici le déroulement *pseudo-parallèle* de 3 processus.

– P1 débute à $t = 0$, utilise 4 quanta, sur une durée de 8 quanta ;

- P2 débute à $t = 1$, utilise 3 quanta, sur une durée de 6 quanta ;
- P3 débute à $t = 3$, utilise 1 quantum, sur une durée de 1 quantum.

Différents algorithmes

- **FIFO** : les processus sont placés dans une file et exécutés dans cet ordre, puis remis dans la file
- **SJF** : (*Shortest Job First*) le processus le plus court est exécuté en premier
- **Round Robin** : Chaque processus aura la main pendant un certain temps, le même pour tous

Priorité

Il est possible d'attacher une priorité à chaque processus de sorte que

- les processus importants soient exécutés en premier ou plus souvent que les autres ;
- les processus peu importants ne soient exécutés que quand le CPU n'est pas beaucoup sollicité.

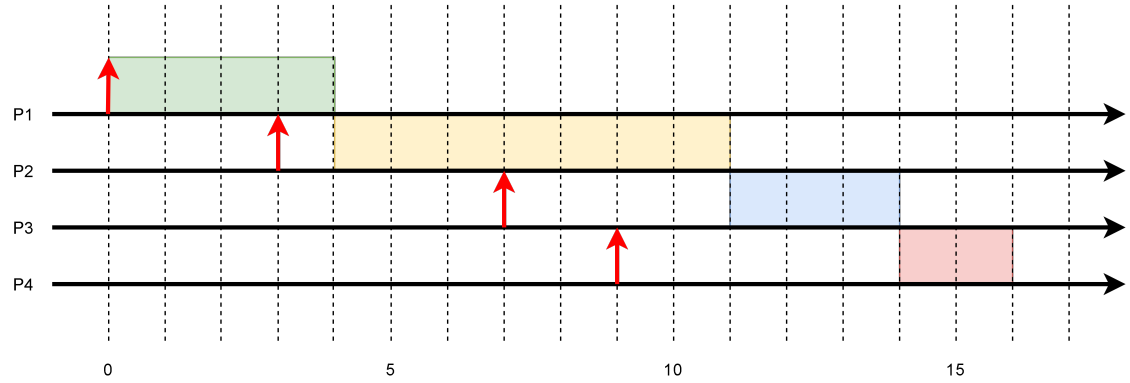
La plupart des OS multitâches actuels utilisent (pour partie) une combinaison des algorithmes précédents combinés avec des priorités.
Dans les systèmes UNIX, la priorité d'un processus s'appelle *niceness*.

4.2 Exemples détaillés

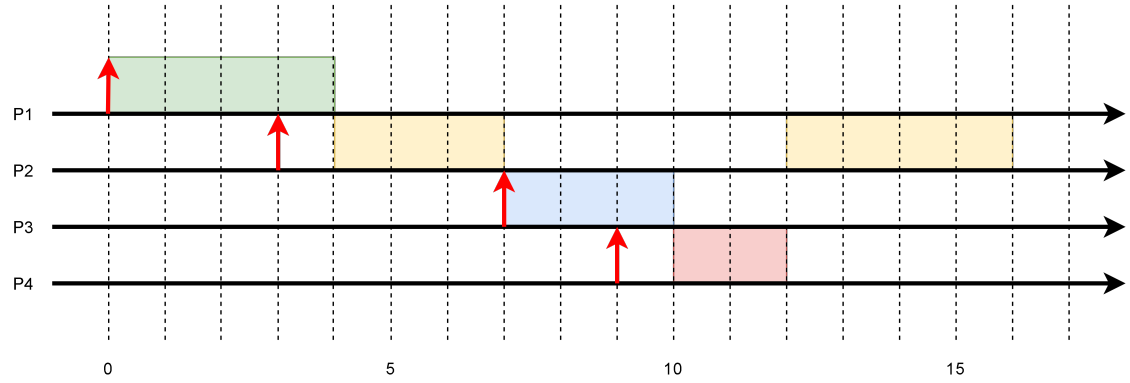
On considère les processus suivants

Processus	Départ	Durée	Priorité
P1	0	4	basse
P2	3	7	basse
P3	7	3	haute
P4	9	2	haute

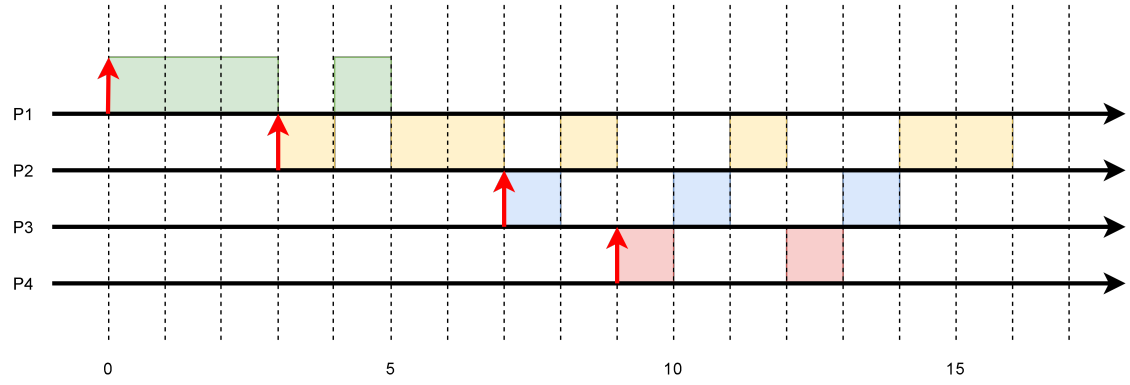
FIFO sans priorités



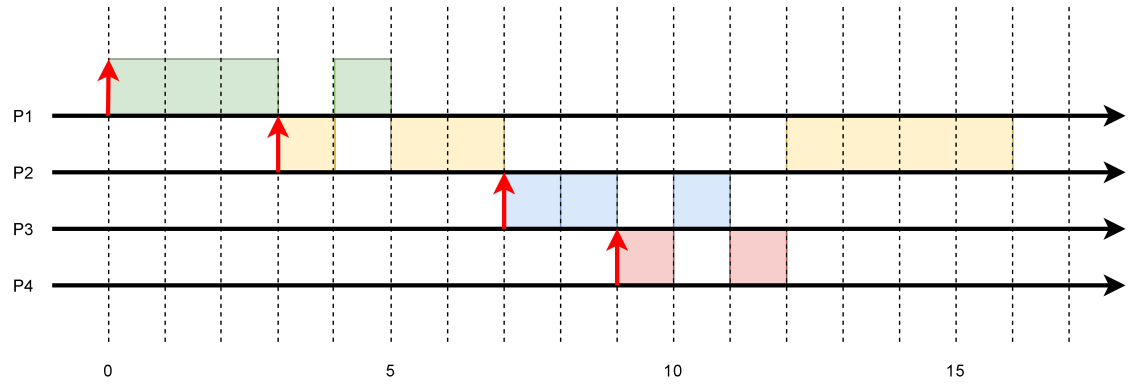
FIFO avec priorités



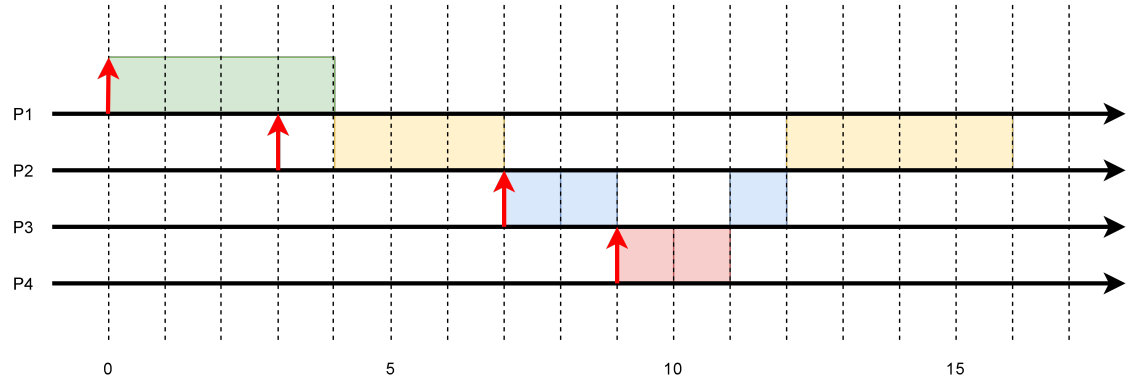
Round Robin sans priorités



Round Robin avec priorités



SJF sans priorités (et aussi avec dans ce cas particulier)



5 L'interblocage

5.1 Processus et ressources

En réalité les algorithmes d'ordonnancement utilisés par les OS actuels font que l'exécution concurrente des processus est *non déterministe* : l'utilisateur de l'OS ne peut pas prédire quel processus sera exécuté à chaque instant.

Or les processus utilisent des *ressources*, et bien souvent celles-ci sont utilisables *de manière exclusive*.

Cela veut dire que lorsqu'un processus acquiert l'accès à une ressource, les autres processus ne peuvent y accéder avant que ledit processus n'ait libéré la ressource.

5.2 Interblocage

On dit qu'il y a *interblocage* lorsque des processus concurrents s'attendent mutuellement.

Dans un article de 1971, Edward Coffman a établi les conditions d'un interblocage, qui portent désormais son nom ?

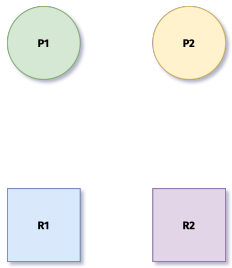
Propriété : conditions de Coffman

- Au moins une ressource doit être conservée par un processus en mode exclusif.
- Un processus doit conserver une ressource et en demander une autre.
- Une ressource ne peut être libérée que par le processus qui la détient.
- *Attente circulaire* : Chaque processus doit attendre la libération d'une ressource détenue par un autre qui fait de même.

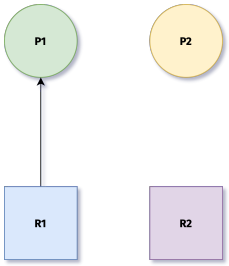
Dans le cadre de ce chapitre, la condition **3.** est toujours vérifiée.

5.3 Exemple Minimaliste

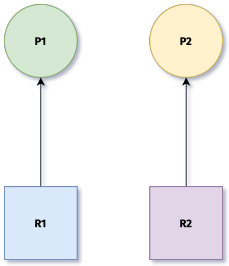
- On considère 2 processus P1 et P2 et 2 ressources R1 et R2



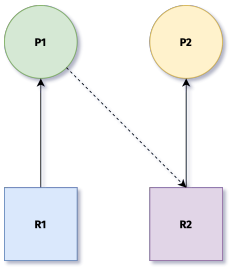
- P1 demande et obtient l'accès à R1



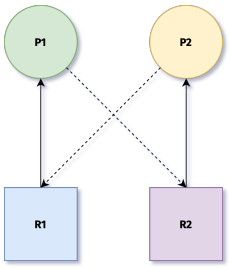
– P2 demande et obtient l'accès à R2



– P1 demande sans l'obtenir l'accès à R2



– P1 attend, P2 demande sans l'obtenir l'accès à R1



– P1 attend que P2 libère R2, qui attend que P1 libère R1 : li y a interblocage.

5.4 Solutions à l'interblocage

- **Ne rien faire :** laisser l'utilisateur gérer lui-même le problème (à l'aide de `kill` sous LINUX ou CTRL+ALT+SUPPR pour lancer le gestionnaire des tâches sous WINDOWS).
- **Détecter :** l'ordonnanceur suit l'allocation des ressources et l'état des processus et redémarre un ou plusieurs d'entre eux si besoin.
- **Éviter :** des algorithmes/recettes sont mis en œuvre pour supprimer une des 4 conditions de Coffman nécessaires à la survenue de l'interblocage.