Ordonnancement des Processus Interblocage

I. Exécution d'instructions par un ordinateur

I.1. Processus *versus* programme

Un **programme** est une succession d'instructions écrites dans un langage donné, qu'on peut stocker sur un support comme un disque dur. Par exemple, un exécutable est un fichier binaire constitué une suite d'instructions en langage machine compréhensibles par l'ordinateur sur lequel il devra s'exécuter.

Un processus en est une instance d'un programme en cours d'exécution, sa réalisation concrète lors de son exécution à un moment donné. À partir d'un même programme, on peut donc créer plusieurs processus, qui peuvent même tourner simultanément. On compare souvent le *programme* à une recette de cuisine et les *processus* à chacun des cuisiniers en train de préparer cette recette à un moment donné. On peut aussi faire le parallèle avec une classe d'objet et ses instances en POO.

C'est le noyau du système d'exploitation (OS) qui gère les processus, qu'il repère en leur affectant un identifiant entier unique (**PID**, *Process IDentifier*). Un processus peut créer de nouveaux processus, dits « **processus enfants** » (dont il sera le **processus parent**). Dans un OS POSIX (« de type UNIX ») comme Linux, au lancement un pseudo-processus numéroté 0 donne naissance au processus `init` de PID 1, qui va engendrer d'autres processus, et ainsi de suite. Cela crée une structure arborescente, une sorte de généalogie. Le PID du processus parent est noté **PPID** (*Parent PID*).

Voici quelques commandes à expérimenter sur un système Linux :

- La commande **`ps -ef`** (ou `ps aux`) permet de lister les processus tournant sur l'OS en indiquant notamment leurs PID et PPID. On peut rediriger sa sortie sur `head` pour ne voir que les premières lignes, ou vers `less` (ou `more`, voire `most` selon les installations, à quitter avec **`q`**) pour pouvoir se déplacer dans les lignes en sortie : `ps -ef | less`
  `ps -ef f` permet en outre de visualiser l'arborescence/ La sortie est une sorte de photographie de l'état du fonctionnement au moment de son exécution.
- **`top`** (ou `htop` s'il est disponible) permet de suivre en temps réel l'évolution des processus. (La touche `F` permet d'accéder aux options de `top`, notamment pour changer le critère de tri ou ajouter une colonne comme le PPID ; `C` permet de basculer vers l'affichage complet de la commande qui a créé le processus, `V` d'obtenir une vue arborescente et `Q` de Quitter `top`).

On peut utiliser `https://bellard.org/jslinux/vm.html?url=alpine-x86.cfg&mem=192` mais certaines commandes seront difficiles ou impossibles à mettre en œuvre sous cette version simplifiée et ce sera moins évident de faire tourner simultanément différents processus. Une version « live » d'un système GNU/Linux, par exemple obtenue en déposant une image ISO d'une distribution GNU/Linux (de préférence « légère ») sur une clé USB préparée avec Ventoy, ou une installation dans une machine virtuelle, via Virtualbox ou autre, reste une bien meilleure solution. L'idéal reste bien sûr de disposer réellement d'un tel système installé, comme au lycée.

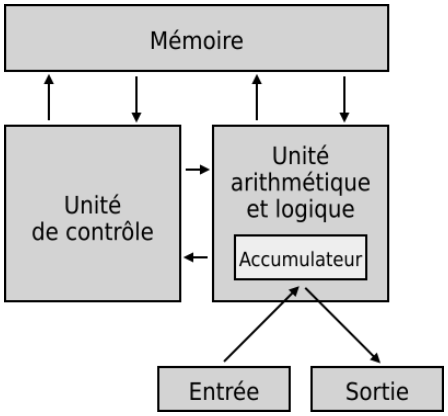
Noter que **chaque processus possède sa propre mémoire** virtuelle (que le programme voit comme s'il disposait de toute la mémoire à lui seul ; l a zone de mémoire physique associée lui a été allouée par l'OS, qui traduit de manière transparente pour le processus les adresses physiques en adresses virtuelles et réciproquement). Ceci interdit qu'un processus donné modifie ou même qu'il lise la mémoire d'un autre.

Un processus peut également **utiliser des ressources**, potentiellement de manière exclusive : fichiers ouverts, connexions réseau, etc.

Il peut également créer des *threads*, ou « flots/fils d'exécution » ou « tâches », parfois même appelés « processus légers ». Ce concept n'est pas spécifiquement au programme, mais on le rencontre fréquemment. Il suffit de savoir que ce qui le distingue d'un processus, c'est essentiellement que les *threads* générés par un même processus partagent le même espace mémoire, celui de leur processus.

I.2. Architecture de Von Neumann

Avant d'aborder la suite, il est important de se rappeler le principe du fonctionnement interne d'un ordinateur bâti sur le modèle de Von Neumann :



Il faut aussi connaître le principe de l'exécution d'un programme, notamment le rôle du **compteur ordinal** (registre PC *Program Counter*, ou **IP** *Instruction Pointer*, voire IAR *Instruction Address Register* ou CIR *Current Instruction Register*, noté **PC/IP** dans la suite) qui contient l'adresse mémoire de la prochaine instruction à exécuter.

Pour ces deux points, reprendre si besoin le cours de première, ou une activité comme celle du collègue David Roche de Bonneville sur https://dav74.github.io/site_nsi_prem/c9a.

II. Ordonnancement

II.1. Principe général

Quand le système d'exploitation « lance » un programme, il **charge le code binaire en mémoire vive**, à une adresse A, puis donne la valeur de A au registre PC/IP. Ainsi, l'exécution du processus peut commencer.

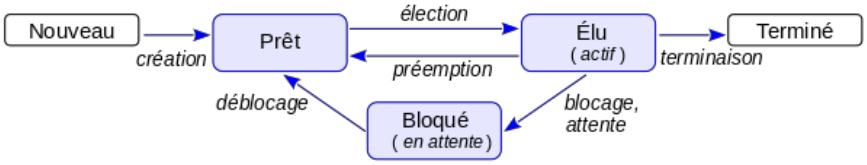
S'il ne pouvait faire que cela, la machine ne pourrait plus rien faire fonctionner d'autre avant la fin de l'exécution du processus actuel et elle serait définitivement « plantée » en cas de boucle infinie par exemple. Une autre situation problématique serait constituée par une phase dite d'attente active d'un processus, qui a besoin d'une valeur de retour d'un périphérique ou de l'utilisateur pour continuer son exécution (saisie, fin de lecture ou d'écriture d'un fichier, etc.), qui bloquerait longuement le fonctionnement complet de l'ordinateur.

Or, on sait que nos systèmes d'exploitation permettent d'exécuter plusieurs processus « simultanément », même s'il n'y a qu'un processeur (et ils savent faire tourner davantage de processus que de processeurs ou de cœurs, quand il y en a plusieurs). C'est l'**ordonnanceur**, une composante du noyau du système d'exploitation, qui permet cette **exécution concurrente**, en permettant l'utilisation du CPU par les différents processus à tour de rôle (c'est lui également qui permet une véritable **exécution parallèle** quand plusieurs processeurs ou cœurs sont disponibles, mais nous n'étudierons que les systèmes à processeur et cœur uniques).

Cela implique de pouvoir stopper l'exécution d'un processus à un moment donné avant la fin de son exécution, pour la reprendre plus tard. Cela fait notamment intervenir l'horloge et les interruptions, pour piloter les différents états des processus.

a. Cycle de vie d'un processus

Un processus peut être dans différents « **états logiques** », illustrés par le graphe suivant, ici en version simplifiée et qu'il faut absolument **connaître parfaitement**. Nous détaillons plus loin la manière d'opérer ces changements d'états.

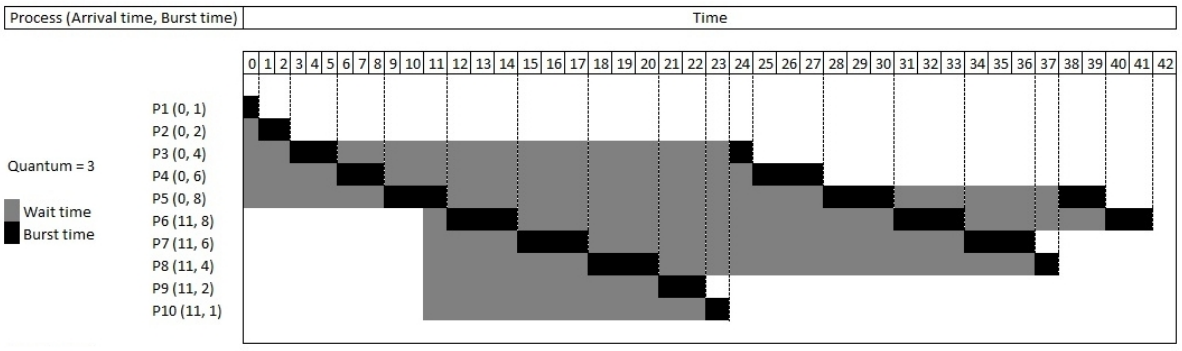


- Le processus naît initialisé à l'état « nouveau » (ou « *created* »).
- Ensuite, il est chargé en mémoire centrale, en attendant son exécution effective, on dit qu'il passe à l'état « **prêt** » (ou « *runnable* »). Il y a souvent de nombreux processus dans cet état, car la ou les places sur le processeur sont rares et les processus nombreux !
- Quand l'ordonnanceur le décide (le mécanisme d'*élection* est différent suivant la stratégie choisie), le processus passe à l'état « **élu** » (ou « **actif** », « *running* ») : il est **en cours d'exécution** par le processeur. L'ordonnanceur peut le ramener au bout d'un certain temps à l'état « prêt » pour céder la place à d'autres.
- À un moment donné, le processus « élu » peut aussi être interrompu et basculer à l'état « **bloqué** » (ou « **suspendu** »), s'il est **en attente d'une ressource** nécessaire pour poursuivre son déroulement : saisie utilisateur, réception de données du réseau ou lecture sur disque, etc. Une fois la ressource disponible, il rebasculera à l'état « prêt », en attendant d'être à nouveau élu par l'ordonnanceur pour terminer son travail.
- En fonctionnement normal, c'est uniquement après le statut « élu » qu'un processus bascule à l'état « terminé » quand il n'a plus rien à effectuer. Mais il arrive qu'il y parvienne directement depuis l'état « bloqué » voire « prêt » (demande d'arrêt brutal, etc. *Ces transitions ne sont pas représentées ci-dessus*). Une fois que le processus est dans cet état « terminé », l'OS peut libérer les ressources que le processus utilisait et informer son parent qu'il a terminé son cycle.

II.2. Stratégie de l'ordonnanceur

L'ordonnanceur choisit les processus à élire et la durée durant laquelle ils s'exécutent sur le processeur selon divers algorithmes, diverses stratégies, qu'on ne demande pas de connaître en détail. Il faut cependant savoir les mettre en œuvre quand on en a rappelé le principe.

Prenons l'exemple du **tourniquet** (***round robin***) : un processus est exécuté durant (au maximum) un temps déterminé (un **quantum**), puis c'est au suivant et ainsi de suite. Les PID des processus en attente d'exécution sont stockés dans une file circulaire. Illustration, avec un quantum de 3 unités de temps arbitraires (le couple suivant le processus indique, dans la même unité de temps, l'instant auquel il a été lancé puis sa durée) :



Une variante avec prise en compte de priorités peut être préférée selon le contexte et les objectifs. Ou alors un algorithme « *First-Come, First-Served* » avec des processus dans une file *FIFO* qui s'exécutent aussi longtemps qu'ils ne sont pas bloqués, ou « *Shortest Job First* » qui nécessite d'estimer le temps de traitement des processus pour prioriser le plus court, etc. (*voir les Travaux Pratiques*).

II.3. Commutation de contexte

Détaillons un peu à présent comment un processus peut passer de l'état « prêt » à l'état « élu » et inversement.

Un processus est caractérisé par principalement par les éléments suivant, qui constituent son **contexte d'exécution** :

- Son identifiant **PID**, celui de son processus parent **PPID**, ainsi qu'un **UID** identifiant de l'utilisateur propriétaire du processus.
- Un espace de **mémoire virtuelle**, contenant le code du programme, la pile pour les appels de fonctions (paramètres, variables locales et valeurs de retour), le tas pour les variables globales...
- L'état des **registres** du processeur sur lequel il s'exécute.
- Les **ressources** qu'il utilise (fichiers ouverts, connexions réseau...)

Quand un processus est suspendu, le système d'exploitation sauvegarde dans une zone mémoire dédiée (souvent la pile) son contexte d'éxécution [1]. Ensuite, celui du processus élu est récupéré, l'état des registres associé restauré, etc. Il peut alors reprendre son fonctionnement là où il en était (grâce au registre PC/IP notamment).

Cette **commutation de contexte** a un coût temporel, qui est pris en compte dans la stratégie adoptée par l'ordonnanceur, même si nous l'ignorons souvent dans les représentations comme celle donnée dans le paragraphe précédent, pour simplifier.

II.4. Interruptions

Une **interruption** [2] est un signal (désigné par un simple entier naturel) envoyé au processeur sur un canal dédié. Ce dernier interrompt alors le déroulement en cours et exécute un programme spécifique (chargé juste après le démarrage de l'OS) appelé **gestionnaire d'interruptions**, qui gère les actions à mener en fonction du type d'interruption.

L'originalité de cette approche est que c'est un **événement extérieur qui sollicite le processeur** (de manière **asynchrone**, c'est-à-dire à un moment quelconque) et non lui qui demande un élément extérieur à un moment précis (par exemple de nombreuses fois à intervalles réguliers, jusqu'à ce qu'il soit en capacité de donner la réponse attendue, ce qui est très coûteux).

Une interruption peut être **matérielle** (IRQ, *Instruction ReQuest*) : fin d'écriture de données sur disque, signalement de paquets qui arrivent sur une carte réseau, etc. C'est le cas du signal envoyé **à intervalle régulier par l'horloge** du système (fréquence de l'ordre de 10Mhz, donc toutes les 100ns env.) qui permet à l'ordonnanceur de gérer ses quantum de temps accordés aux processus.

Il existe aussi des interruptions **logicielles** (*soft IRQ*), par exemple pour gérer les **erreurs de segmentation** (*segmentation fault* aka. « *segfault* », tentative d'accès à une zone de mémoire non autorisée) ou les **exceptions** (signalement d'erreur comme une division par zéro, qui conduit souvent à un « plantage » si elle n'est pas prise en compte de manière spécifique).

Les signaux des systèmes POSIX envoyés à des processus, permettent d'interrompre leur exécution, par exemple pour demander à tout moment une terminaison « contrôlée » (*core dump* préalable, à savoir sauvegarde de l'état de la mémoire et des registres, par exemple via le signal `SIGTERM`, le 15), ou brutale et inconditionnelle (`SIGKILL`, 9), ou une mise en pause (`SIGSTP`), ou d'informer qu'un de ses processus fils est

1. Plus de détails sur https://fr.wikipedia.org/wiki/Contexte_d%27ex%C3%A9cution
2. On s'en tient à l'essentiel ici, mais les curieuses et curieux trouveront plus de détails sur https://fr.wikipedia.org/wiki/Interruption_(informatique)

mort (`SGCHLD`), etc. On peut l'expérimenter avec **`kill -[n°_ou_SIGXXX] [n°_des_processus]`** (par exemple `kill -15 12586` ou `kill -KILL 12586`, ou `kill -SIGKILL 12586`) où le signal par défaut, en l'absence de précision, est `SIGTERM`.
`kill -L` affiche les correspondances entre code et entiers. Un développeur peut choisir comment le processus devra réagir à un signal, sauf si ce dernier ne peut être ignoré (`SIGKILL`).

Les mécanismes des exceptions (cf. blocs `try: ... except: ...` en Python) s'appuient sur des mécanismes similaires.

III. Dangers des exécutions concurrentes

III.1. Partage de ressources

Tout semble aller au mieux dans le meilleur des mondes : l'exécution effective des processus sur le processeur est régulièrement interrompue par l'ordonnanceur, mais elle reprend comme s'il n'y avait pas eu de pause, grâce à la récupération de leur état mémoire, des valeurs de registres qui les concernent et d'autres caractéristiques spécifiques. Ainsi, même s'il s'agit d'une exécution concurrente, l'utilisateur la perçoit comme si elle était véritablement parallèle.

Mais ceci se complique s'ils souhaitent utiliser des ressources partagées par d'autres processus : en effet, il y a un risque que pendant qu'un processus est retourné au statut « prêt », la ressource soit modifiée par un autre, qui lui est actif. C'est le plus souvent à proscrire. Il suffit de penser au cas d'une base de données dans laquelle plusieurs utilisateurs modifieraient le contenu d'une table simultanément (heureusement, les SGBD se chargent de l'interdire), ou simplement à un fichier texte lu par un ou des processus, pendant qu'un autre le modifie.

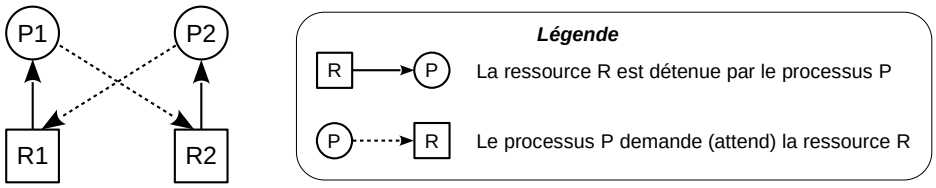
Quand on ouvre un fichier en écriture par exemple, on sait qu'il faut penser à le refermer ensuite. C'est ce qui permet à d'autres processus d'y avoir accès ensuite.

De manière générale, il est parfois nécessaire de « verrouiller » une ressource à un moment donné, pour en garantir un usage exclusif par l'un des processus. La partie correspondante du programme qu'on appelle une **section critique**, sera la plus courte possible pour ne pas mobiliser inutilement cette ressource.

III.2. Interblocage (*deadlock*)

a. Principe

Prenons un exemple simple : un processus P1 a un accès exclusif à une ressource R1 (on dit parfois qu'il la détient ou qu'elle la verrouille) et un processus P2 détient une ressource R2. Si, pour avancer dans leur déroulement, P1 a besoin de l'accès à R2 et P2 a besoin de l'accès à R1, les deux vont se retrouver indéfiniment coincés sur le statut « bloqué », en l'attente de leur seconde ressource. En effet, seul l'autre processus peut la libérer, or il attend lui-même une autre ressource indisponible pour avancer...

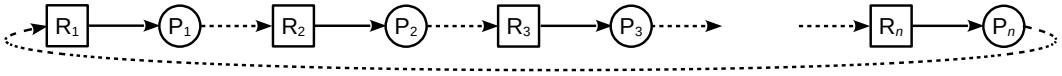


Un **interblocage** (***deadlock*** en anglais) désigne ce type de situation, dans laquelle plusieurs processus sont tous bloqués, mais ou l'événement qui pourrait débloquer chacun d'eux doit être produit par l'un des autres processus bloqués. Ainsi, la situation ne peut plus progresser sans intervention extérieure aux processus.

b. Conditions de Coffman

E. Coffman a mis en évidence quatre conditions nécessaires et suffisantes pour qu'il y ait un interblocage :

- **Exclusion mutuelle** : il existe une ou plusieurs ressources auxquelles un processus au plus peut avoir accès à un moment donné.
- **Possession et attente** (*Hold and wait*) : il existe un processus qui détient une ressource et qui en a besoin d'une autre pour sortir de l'état bloqué.
- **Pas de préemption (ou de réquisition)** : seul le processus qui détient une ressource peut la libérer lui-même.
- **Attente circulaire** : il existe une chaîne de dépendances, du type du schéma ci-dessus ou plus généralement, si les P_k sont des processus et les R_k des ressources, « P_1 détient R_1 et demande R_2, détenue par P_2 qui demande R_3, détenue par P_3 qui demande R_4, etc., détenue par R_{n-1} qui demande P_n, détenue par R_n qui demande P_1 :



(Ces conditions sont parfois énoncées sous leur forme nécessaire, mais pas suffisante, en indiquant par exemple pour la seconde qu'un processus qui détient une ressource *peut* en requérir une autre...)

Illustrons ce principe avec le « banquet des philosophes proposé par E. Dijkstra » (retrouver si besoin le descriptif *sur Wikipédia* d'où est tirée l'illustration) :

- **Exclusion mutuelle** : chaque fourchette ne peut être utilisée que par un seul philosophe à la fois.
- **Possession et attente** (*Hold and wait*) : un philosophe affamé va détenir la fourchette à sa gauche, puis attendre de pouvoir détenir également celle qui est à sa droite.
- **Pas de préemption (ou de réquisition)** : seul un philopsophe peut décider de reposer une fourchette qu'il détient.
- **Attente circulaire** : dans le cas où tous les philosophes deviennent affamés simultanément, chacun va prendre et donc détenir de manière exclusive la fourchette à sa gauche, puis requérir celle à sa droite. Mais celle-ci est détenue par son voisin de droite, lui-même en attente, etc. En faisant ainsi le tour de la table, on revient au philosophe de départ.

c. Stratégies

Pour éviter de se retrouver dans une telle situation, diverses approches sont envisageables...

- **Ignorer** ! La « stratégie de l'autruche » est utilisée par beaucoup d'OS. En effet, le coût des approches suivantes est élevé.
- **Détecter, remédier** : on les laisse se produire, puis on tente de les détecter (présence de cycles dans le graphe des dépendances qu'on a représenté ci-dessus, on reviendra sur ces notions dans l'année), puis de les supprimer.
- **Éviter dynamiquement** : les ressources sont allouées avec précaution, par exemple avec « l'algorithme du banquier », en refusant l'allocation d'une ressource tant qu'elle conduirait à un état du système considéré comme « non sûr ».
- **Prévenir** : il s'agit d'éviter que l'une des quatre conditions de Coffman puisse survenir.

On peut retenir que s'il est possible d'un point de vue théorique de les éviter complètement, c'est souvent irréaliste en pratique car trop coûteux.