Gestion des processus

Exécution d'un programme sur un processeur

Nous avons vu l'année dernière que pour qu'un programme s'exécute sur le processeur, il devait être traduit (par un compilateur ou un interpréteur) en langage machine. Au début de l'exécution du programme, les instructions sont copiées en mémoire. Parmi les registres du processeur, il y en a 2 qui ont un rôle spécial : `IR` contient l'instruction courante et `IP` l'adresse de la prochaine. L'exécution d'une instruction se passe de la manière suivante :

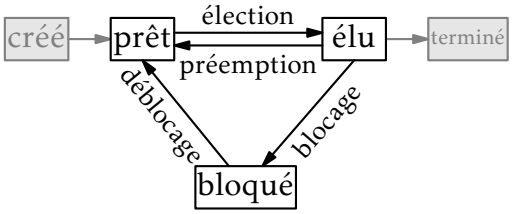
1) le contenu de la mémoire vive à l'adresse pointée par `IP` est copiée dans `IR` ;
2) l'instruction contenue dans `IR` est décodée, ce qui active le circuit électrique qui réalise l'opération visée ;
3) l'instruction décodée est exécutée et `IP` est mis à jour.

On continue ainsi jusqu'à ce que l'exécution se termine.
Sauf que ce fonctionnement correspond à un système mono-tâche. Le processeur ne peut rien faire d'autre tant que l'exécution du programme n'est pas arrivée à son terme. Depuis les années 70, les systèmes d'exploitation sont multi-tâches et permettent à plusieurs programmes de s'exécuter en parallèle. Mais comment faire cela avec un seul processeur ? Et même avec les processeurs modernes, comment faire s'il y a plus de programmes que de cœurs ?

Les processus

Un **processus** est une instance d'un programme en cours d'exécution. Un même programme peut être lancé plusieurs fois et dans ce cas, chaque instance est un processus différent.
Pour l'instant, considérons que le processeur ne possède qu'un seul cœur. Imaginons que l'utilisateur est en train de programmer en Python à l'aide de Thonny et que son programme actuel met beaucoup de temps avant de se terminer. En attendant, il utilise un navigateur Internet pour consulter les documentations, le tout en écoutant de la musique stockée sur son ordinateur. Le processeur doit donc exécuter au moins 3 processus en parallèle. Sauf qu'un seul peut s'exécuter à la fois. Pour donner l'illusion du parallélisme, le processeur va exécuter successivement quelques instructions de chaque processus avant de passer au suivant.
Les processus ne sont donc pas constamment en exécution. En fait, ils peuvent être dans 3 états :

- **Prêt** : le processus est prêt à être exécuté.
- **Élu** : le processus est en cours d'exécution.
- **Bloqué** : le processus a besoin d'une ressource pour continuer son exécution.



Pour comprendre l'état "bloqué", on peut prendre l'exemple suivant. Lorsqu'on appuie sur le bouton vert de Thonny, il va sauvegarder le fichier avant de l'exécuter. Il doit donc attendre que le fichier ait été écrit avant de pouvoir passer à la suite. Pour ne pas faire perdre du temps au processeur, le processus de Thonny passe dans l'état "bloqué", ce qui l'empêche d'être exécuté. Mais comment savoir quand débloquer Thonny ?

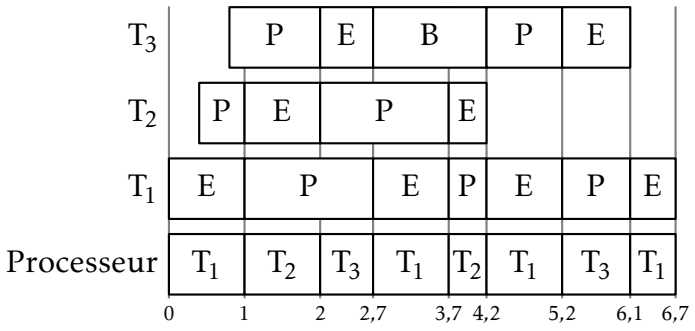
Les interruptions

Le système produit des **interruptions** (action sur le clavier, disque qui signale la fin d'une écriture, levée d'une exception…). Lorsque le processeur reçoit une interruption, il finit d'exécuter l'instruction en cours et active le **gestionnaire d'interruption**.

Ce dernier va regarder si un processus attendait cette interruption et si c'est le cas, il va le débloquer.
Pour l'exemple précédent de Thonny, lorsque le disque dur envoie l'interruption signalant la fin d'écriture, le gestionnaire d'interruption va faire passer Thonny dans l'état "prêt".
Afin d'assurer l'alternance entre les processus, le processeur génère de lui-même des interruptions, appelées **interruptions d'horloge** à intervalles réguliers. Pour les processeurs actuels, c'est de l'ordre de 100 ns. Dans ce cas là, le gestionnaire d'interruption va utiliser **l'ordonnanceur** qui va déterminer quel processus exécuter ensuite.

L'ordonnancement

Il existe plusieurs stratégie possibles pour l'ordonnancement. On peut choisir de passer au processus le plus prioritaire, au plus rapide, au dernier arrivé, à celui qui attend depuis le plus longtemps…
Linux peut utiliser plusieurs ordonnanceurs, dont un basé sur la stratégie du tourniquet (*round robin*). Les processus "prêts" sont placés dans une file d'attente. Le premier de la file est "élu" et exécuté pendant un certain laps de temps, appelé **quantum**. Si à la fin du quantum le processus n'a pas terminé son exécution, il est interrompu et on le place à la fin de la file. On passe alors au processus suivant.
Lorsqu'un processus passe de "bloqué" à "prêt", il est placé à la fin de la file.
Dans la figure ci-contre, on peut voir 3 processus s'exécuter sur un processeur. On prend un quantum comme unité de mesure. Le processus T_3 se retrouve dans l'état "bloqué" à 2,7 parce qu'il attend une ressource, qui semble être relâchée par T_2 à la fin de son exécution. Au total, T_1 a pris 3,6 unités de temps, T_2 1,5 et T_3 1,6.



Si un processeur possède plusieurs cœurs, il peut exécuter plusieurs processus en même temps. Il peut y avoir une unique file pour tous les cœurs ou des files pour chaque cœurs.

Changement de contexte

Le processeur passe donc très régulièrement d'un processus à un autre. Mais pour qu'un processus puisse continuer son exécution, il faut que les registres aient les mêmes valeurs à la fin de son exécution et à sa reprise. Il faut donc sauvegarder ces valeurs lorsque le processus est mis en pause.
Le système d'exploitation garde en permanence un certain nombre d'informations sur les processus dans une structure appelée **bloc de contrôle de processus**, ou PCB pour *Process Control Bloc*. Ces informations s'appellent **le contexte** du processus. On y retrouve principalement :

- Le PID (*Process ID*) qui est un entier qui identifie le processus.
- L'état du processus (élu, prêt ou bloqué).
- Une sauvegarde des registres lors de la dernière interruption.
- La plage d'adresses allouée au processus pour son exécution.
- La liste des ressources utilisées : fichiers ouverts, connexions réseaux en cours, périphériques utilisés…

Lorsque le processeur passe d'un processus A à un processus B, il doit donc sauvegarder l'état des registres pour A dans le PCB, charger celui de B et reprendre son exécution. On dit que c'est un **changement de contexte**.

Les threads

Parfois, un programme peut lui même faire plusieurs tâches en même temps. Par exemple, un navigateur Internet peut en même temps lire une vidéo, télécharger un fichier, afficher une nouvelle page et exécuter du code javascript sur un autre onglet. Pour cela, il peut générer plusieurs **fils d'exécution**, ou *threads*. Ce sont des sous-processus, ou processus légers. Ils partagent la même zone mémoire, alors que deux processus différents ont des zones différentes. À notre niveau, nous pourrons considérer que ce sont des processus comme les autres.
L'utilisation de threads permet de faire de la **programmation concurrente**. L'idée est de découper le programme en plusieurs sous-programmes qui s'exécutent en parallèle. Cela pose de vrais défis, puisque tous les problèmes ne se prettent pas très bien à ce genre d'approche. Cela pose aussi des problèmes de synchronisation entre les différents threads, ce qui peut entraîner des blocages.

Les processus en Linux

Pour visualiser les processus s'exécutant sur un ordinateur sous Linux, il faut utiliser la commande `ps`. Il existe de très nombreuses options permettant d'obtenir plus ou moins d'informations.
Voici un extrait obtenu avec `jslinux`.

```
[rootlocalhost root]# ps -efH
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 14:59 ?        00:00:00 /bin/sh /sbin/init
root        41     1  0 14:59 ?        00:00:00   dhcpcd
root        45     1  0 14:59 ?        00:00:00   /usr/bin/sh /bin/startx
root        64    45  0 15:00 ?        00:00:00     xinit /etc/X11/xinit/xinitrc
root        65    64  7 15:00 ?        00:00:13       /usr/libexec/Xorg :5 -auth
root        91    64  2 15:00 ?        00:00:05       fluxbox
root       113    91  1 15:00 ?        00:00:02         xterm
root       115   113  0 15:00 pts/0    00:00:00           bash
root       137   115  0 15:03 pts/0    00:00:00             ps -efH
```

Les colonnes correspondent respectivement à l'utilisateur du processus, son numéro, le numéro de son parent, le pourcentage d'utilisation du processeur, l'heure de départ, le terminal dans lequel il a été lancé (? s'il de provient pas d'un terminal), le temps d'utilisation du processeur et la commande appelée. On remarque que le premier processus créé est `init` qui lance tout le système d'exploitation. C'est lui qui lance, entre autres, `dhcpcd` (pour se connecter au réseau) et `startx` (serveur graphique).
L'affichage se fait en une seule fois. Pour avoir une vision en temps réel de l'état des processus, on peut utiliser la commande `top` :

```
[rootlocalhost root]# top
top - 15:29:56 up 30 min,  0 users,  load average: 0.00, 0.01, 0.02
Tasks:  24 total,   1 running,  23 sleeping,   0 stopped,   0 zombie
%Cpu(s): 13.0 us, 21.7 sy,  0.0 ni, 65.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    434.1 total,    381.3 free,     27.2 used,     25.6 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    398.3 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  219 root      20   0    8432   3124   2680 R  15.0   0.7   0:00.21 top
    1 root      20   0    3136   2516   2148 S   0.0   0.6   0:00.77 init
   41 root      20   0    1944   1472   1196 S   0.0   0.3   0:00.04 dhcpcd
   65 root      19  -1  106480  22432   7576 S   0.0   5.0   0:15.96 Xorg
```

Pour tuer un processus, il faut utiliser la commande `kill PID` ou `kill -9 PID`. La première commande est similaire au fait d'appuyer sur la croix en haut d'une fenêtre pour la fermer. La deuxième tue immédiatement le processus et toutes les données non sauvegardées sont perdues.

Gestion des ressources

Comme nous l'avons vu, les processus utilisent tous de la mémoire, mais sur des plages différentes. Cela ne pose donc pas de problèmes. Mais certaines ressources ne peuvent pas être partagées de la même manière. C'est le cas de la plupart des ressources matérielles, comme la carte son. Si un processus utilise le micro, un autre ne peut pas jouer de la musique Les fichiers aussi peuvent poser problème. Si un processus est en train d'écrire dans un fichier il faut faire attention à ce que d'autres ne soient pas en train de le lire ou d'écrire.
Pour cela, on utilise des **verrous** qui permettent d'assurer l'exclusivité de l'accès à une ressource. Les processus demandent d'accès à une ressource. Une fois qu'elle est libre, ils la vérouillent et aucun autre processus ne peut l'utiliser tant qu'ils ne l'ont pas relâchée.

L'interblocage

Ce système de verrous peut provoquer des **interblocages** (*deadlock*) où deux processus attendent mutuellement une ressource utilisée par l'autre sans relâcher la leur.
Voici un exemple où deux processus A et B utilisent des ressources R et S.

Si le processeur exécute A1 et B1, alors A2 et B2 vont faire entrer les deux processus dans l'état "bloqué". Aucun des deux ne va libérer sa ressource et ils ne pourront jamais se débloquer.

Processus A		**Processus B**	
étape A1	demande R	étape B1	demande S
étape A2	demande S	étape B2	demande R
étape A3	libère S	étape B3	libère R
étape A4	libère R	étape B4	libère S

Par contre, si le processeur exécute A1 puis A2 avant B1, alors B sera bloqué mais pas A qui pourra libérer les ressources et permettre à B de s'exécuter.
En 1971, Edward Coffman Jr a énoncé les conditions nécessaires à la survenue d'un interblocage :

- Au moins une ressource doit être conservée dans un mode non partageable.
- Un processus doit maintenir une ressource et en demander une autre.
- Une ressource ne peut être libérée que par le processus qui la détient.
- Chaque processus doit attendre la libération d'une ressource détenue par un autre qui fait de même.

Pour éviter cela on peut utiliser diverses stratégies :

- **La prévention** : on oblige le processus à déclarer à l'avance la liste de toutes les ressources auxquelles il va accéder.
- **l'évitement** : on fait en sorte qu'à chaque étape il reste une possibilité d'attribution de ressources qui évite le deadlock.
- **la détection/résolution** : on laisse la situation arriver jusqu'au deadlock, puis un algorithme de résolution détermine quelle ressource libérer pour mettre fin à l'interblocage.

On peut aussi régler le problème en ne donnant pas directement accès aux ressources. Sous Linux, ce sont des processus spéciaux, appelés *daemons* qui accèdent aux ressources. Ainsi, pour accéder à la carte son, les autres processus passent par le daemon concerné. Il est le seul à interagir avec la carte, ce qui permet à plusieurs processus d'utiliser le micro ou la sortie son en même temps.