Cours 1 Système d'exploitation 2 L3

Production d'un programme

Avant d'être exécuté, un programme doit passer par plusieurs étapes. Au début, le programmeur crée un fichier et écrit son programme dans un langage source, le C par exemple. Un **compilateur** transforme ce programme en un module objet. Le module objet représente la traduction des instructions en C, en langage machine. Le code produit est en général relogeable, commençant à l'adresse 00000 et pouvant se translater à n'importe quel endroit de la mémoire en lui donnant comme référence initiale le registre de base. Les adresses représentent alors le décalage par rapport à ce registre.

On peut rassembler les modules objets dans des bibliothèques spécialisées, par exemple au moyen des commandes ar et ranlib (pour avoir un accès direct) sous Unix ou TLIB avec le compilateur C de Borland. On réunit ensuite les bibliothèques dans un répertoire, en général /usr/lib sous Unix.

Les appels à des procédures externes sont laissés comme des points de branchements. L'**éditeur de liens** fait correspondre ces points à des fonctions contenues dans les bibliothèques et produit, dans le cas d'une liaison statique, une image binaire. Certains systèmes, notamment Unix, autorisent des liaisons dynamiques et reportent la phase d'édition de liens jusqu'à l'instant de chargement. Il faut alors construire les objets et les bibliothèques d'une manière légèrement différente. Grâce à cette technique, on peut mettre les bibliothèques à jour sans avoir à recompiler et on encombre moins les disques.

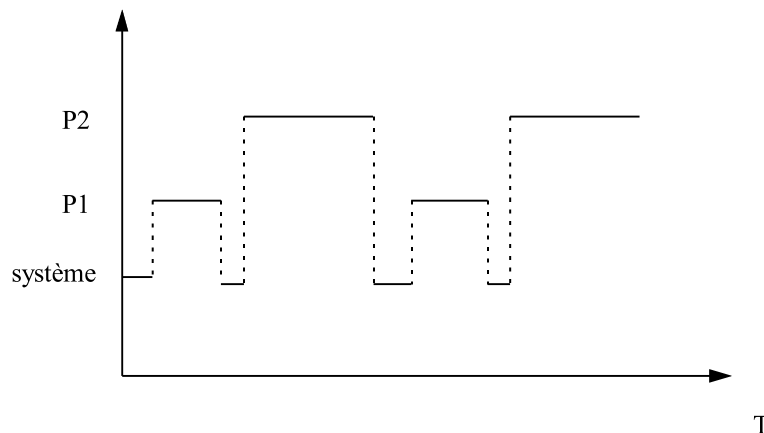
Le **chargeur** effectue les liaisons des appels système avec le noyau, tel que l'appel write par exemple. Enfin, il charge le programme en mémoire.

Outre le code compilé (le texte) et la zone de données initialisées, le fichier exécutable contient un certain nombre d'autres informations. Dans le cas du système Unix, il y a notamment un en-tête composé d'un nombre « magique » (un code de contrôle de type), de diverses tailles (texte, données,…), du point d'entrée du programme, ainsi qu'une table de symboles pour le débogage.

Les processus

Généralités

Les processus correspondent à l'exécution de tâches : les programmes des utilisateurs, les entrées-sorties,… par le système. Un système d'exploitation doit en général traiter plusieurs tâches en même temps. Comme il n'a, la plupart du temps, qu'un processeur, il résout ce problème grâce à un pseudo-parallélisme. Il traite une tâche à la fois, s'interrompt et passe à la suivante. La commutation des tâches étant très rapide, l'ordinateur donne l'illusion d'effectuer un traitement simultané.



Le multi-tâche

Les processus des utilisateurs sont lancés par un interprète de commande. Ils peuvent eux-mêmes lancer ensuite d'autres processus.

Les messages

Les messages forment un mode de communication privilégié entre les processus. Ils sont utilisés dans le système pédagogique Minix de Tanenbaum. Ils sont au cœur de Mach qu'on présente comme un successeur possible d'Unix et qui a inspiré Windows NT pour certaines parties. Par ailleurs, ils s'adaptent très bien à une architecture répartie et leur mode de fonctionnement est voisin des échanges de données sur un réseau.

Les communications de messages se font à travers deux opérations fondamentales : envoie(message) et reçois(message). (send et receive). Les messages sont de tailles variables ou fixes. Les opérations d'envoi et de réception peuvent être soit directes entre les processus, soit indirectes par l'intermédiaire d'une boîte aux lettres.

Les communications directes doivent identifier le processus, par un n° et une machine, par exemple. On aura alors : envoie(Proc, Message) et reçois(Proc, Message). Dans ce cas la

communication est établie entre deux processus uniquement, par un lien relativement rigide et bidirectionnel. On peut rendre les liaisons plus souples en laissant vide l'identité du processus dans la fonction reçois.

Les communications peuvent être indirectes grâce à l'utilisation d'une boîte aux lettres (un « port » dans la terminologie des réseaux). Les liens peuvent alors unir plus de deux processus du moment qu'ils partagent la même boîte aux lettres. On devra néanmoins résoudre un certain nombre de problèmes qui peuvent se poser, par exemple, si deux processus essayent de recevoir simultanément le contenu d'une même boîte.

Les communications se font de manière synchrone ou asynchrone. Le synchronisme peut se représenter par la capacité d'un tampon de réception. Si le tampon n'a pas de capacité, l'émetteur doit attendre que le récepteur lise le message pour pouvoir continuer. Les deux processus se synchronisent sur ce transfert et on parle alors d'un « rendez-vous ». Deux processus asynchrones : P et Q, peuvent aussi communiquer de cette manière en mettant en œuvre un mécanisme d'acquittement :

P	Q
envoie(Q, message)	reçois(P, message)
reçois(Q, message)	envoie(P, acquittement)

Les fonctions de messages.

La mémoire partagée

On peut concevoir des applications qui communiquent à travers un segment de mémoire partagée. Le principe est le même que pour un échange d'informations entre deux processus par un fichier. Dans le cas d'une zone de mémoire partagée, on devra déclarer une zone commune par une fonction spécifique, car la zone mémoire d'un processus est protégée.

Le système Unix fournit les primitives permettant de partager la mémoire. NT aussi sous le nom de fichiers mappés en mémoire. Ces mécanismes, bien que très rapides, présentent l'inconvénient d'être difficilement adaptables aux réseaux. Pour les communications locales, la vitesse est sans doute semblable à celle de la communication par un fichier à cause de la mémoire cache. Lorsqu'il a besoin de partager un espace mémoire, le programmeur préférera utiliser des fils d'exécution.

Exclusions entre les processus

L'exclusion a pour but de limiter l'accès à une ressource à un ou plusieurs processus. Ceci concerne, par exemple, un fichier de données que plusieurs processus désirent mettre à jour. L'accès à ce fichier doit être réservé à un utilisateur pendant le moment où il le modifie, autrement son contenu risque de ne plus être cohérent. Le problème est semblable pour une imprimante où un utilisateur doit se réserver son usage le temps d'une impression. On appelle ce domaine d'exclusivité, une section critique.

L'attente active sur un verrou

La façon la plus ancienne de réaliser l'exclusion mutuelle sous Unix est d'effectuer un verrouillage sur une variable partagée. Cette opération de verrouillage doit être indivisible (atomique). Lorsque le verrou est posé sur la variable, les processus exécutent une attente active. Lorsqu'il se lève, un processus, et un seul, pose un nouveau verrou sur la variable et rentre dans la section critique. Il sait qu'il aura l'exclusivité d'une ressource, un fichier par exemple :

```
Processus 1                                Processus 2
while (pos_ver(var) == échec)              while (pos_ver(var) == échec)
        rien;                                      rien;
section_critique();                        section_critique();
lève_ver(var);                             lève_ver(var);
```

Attente active sur un verrou.

Dans les vieilles versions d'Unix, la variable de verrouillage correspondait souvent à un fichier sans droits que chacun des processus tente de créer. Lorsqu'un processus termine sa section critique, il détruit le fichier de verrouillage, laissant ainsi sa place à d'autres. On créera le verrou dans /tmp , par exemple, de manière à ce que tout le monde puisse le détruire. Enfin, pour que ce procédé fonctionne, il est nécessaire que tous les processus coopèrent.

```
void main()
{
    int desc;                  /* descripteur fichier de ressource */
    int desc_verr;  /* descripteur fichier verrou */
    char i;
                               /* attente active */
    while ((desc_verr = creat("/tmp/verrou", 0000)) == -1)
            ;
    section_critique();
    close(desc_verr);                /* fermeture du verrou */
    unlink("/tmp/verrou");           /* destruction du verrou */
}
```

int creat(char * ref_fichier, int mode) rend le descripteur du fichier ouvert en écriture, ou -1 en cas d'échec. mode définit les droits d'accès, 0777 p. ex. Le mode réel est obtenu par un ET binaire avec le paramètre cmask positionné par la fonction

int umask(int cmask)

int close(int desc_fichier) ferme le fichier considéré. Cette fonction rend 0 en cas de succès et –1 en cas d'échec.

int unlink(char * ref_fichier) détruit le fichier dont le nom est passé en paramètre. Cette fonction rend 0 en cas de succès et –1 en cas d'échec.

Au niveau du processeur, les variables de verrouillage se réalisent par une instruction de type Test and Set Lock (TSL).

On n'utilise plus désormais cette façon de procéder. On préfère la fonction flock(int fd, int opération) qui a les mêmes fonctionnalités. Elle permet à un processus de verrouiller l'accès à un fichier de descripteur fd avec un verrou exclusif. Dans ce cas, opération est égal à LOCK_EX, et le processus attendra s'il y a déjà un processus ou passera si la voie est libre. On retire le verrou avec la même fonction. L'opération est LOCK_UN. Le fichier d'en-tête à inclure est <sys/file.h>.

Les sémaphores

Généralités

Le concept de sémaphore permet une solution élégante à la plupart des problèmes d'exclusion. Ce concept nécessite la mise en œuvre d'une variable, le sémaphore, et de deux opérations atomiques associées P et V. Soit séma la variable, elle caractérise les ressources et permet de les gérer. Lorsqu'on désire effectuer un exclusion mutuelle entre tous les processus par exemple, il n'y a virtuellement qu'une seule ressource et on donnera à séma la valeur initiale de 1.

Lorsqu'un processus effectue l'opération P(séma) :

- si la valeur de séma est supérieure à 0, il y a alors des ressources disponibles, P(séma) décrémente séma et le processus poursuit son exécution,
- sinon ce processus sera mis dans une file d'attente jusqu'à la libération d'une ressource.

Lorsqu'un processus effectue l'opération V(séma) :

- si il n'y a pas de processus dans la file d'attente, V(séma) incrémente la valeur de séma,
- sinon un processus en attente est débloqué.

P(séma) correspond donc à une prise de ressource et V(séma) à une libération de ressource. Dans la littérature, on trouve parfois d'autres terminologies, respectivement, wait(séma) et signal(séma) , ou get(séma) et release(séma).

Le problème des producteurs et des consommateurs

Ce problème est un des cas d'école favoris sur les processus. Deux processus se partagent un tampon de données de taille N. Un premier processus produit des données et les écrit dans le tampon. Un second processus consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture. Initialement, le tampon est vide. La synchronisation de ces deux processus peut se réaliser grâce à des sémaphores.

Les processus producteur et consommateur doivent accéder de manière exclusive au tampon, le temps d'une lecture ou d'une écriture. Un sémaphore d'exclusion mutuelle est donc nécessaire. D'autre part, on peut considérer que les ressources du processus producteur sont les emplacements vides du tampon, alors que les emplacements pleins sont les ressources du processus consommateur. Au départ, le tampon étant vide, les ressources de consommation sont nulles, alors que les ressources de production correspondent à la taille du tampon.