Chapitre 5

Le niveau du système d'exploitation

5.1 Systèmes d'exploitation multi-tâches

En plus de la gestion d'un système de fichiers, le rôle d'un système d'exploitation multi-tâches est de partager les différentes *ressources* (la mémoire, le processeur et les périphériques) entre différents processus. Ces processus apparaissent comme des *concurrents* pour accéder aux ressources, ce mot est employé aussi comme qualificatif d'un système d'exploitation multi-tâche.

Les appels systèmes fournissent ce qui est nécessaire à la gestion des fichiers (typiquement des appels `open()`, `read()`, `write()`, `close()`), mais aussi des appels pour lancer un autre (et peut-être indépendant) processus (`fork()`), ou encore d'envoyer un *signal* à un autre processus (`kill()`).

A un autre niveau, et l'utilisateur nén est pas réellement conscient, le système assure d'autres services comme par exemple empêcher un processus d'aller écrire à des adresses de mémoire qui ne lui ont pas été attribuées, ou encore qu'aucun processus ne monopolise une ressource.

Le système doit garder une table contenant des informations sur tous les processus actifs, avec des détails sur leur occupation en mémoire centrale, leur taille, leur propriétaire, etc. Chaque processus doit être identifié dans cette table d'une façon unique, cela peut être fait grâce à un numéro : le PID (*Process IDentifier*).

Lorqu'un système est multi-tâche, il n'y a plus qu'un petit pas à franchir pour en faire un système multi-tâche et multi-utilisateur. Chaque utilisateur accède à la machine grâce à un terminal, lequel est géré par un processus indépendant des autres processus. Le principal ajout pour avoir un système multi-utilisateur est le besoin de *sécurité*, pour que les utilisateurs puissent être assurés du caractère privé de leurs processus et de leurs fichiers s'ils le désirent.

5.1.1 Le partage du processeur (en anglais : *Processor scheduling*)

La question est : une fois que la ressource CPU a été donnée à un processus, comment le noyau fait-il pour la reprendre ? Comme nous l'avons déjà dit, chaque appel système redonne le processeur au noyau. En fait les appels systèmes sont exécutés par une instruction machine du niveau du processeur qui déclenche une *exception* (souvent le mnémonique de ces instructions dans les langages d'assemblage est TRAP) laquelle a un effet immédiat sur le niveau matériel de l'ordinateur. En particulier, le processus courant est interrompu (suspendu) et le processeur commence à exécuter les instructions qui sont à une adresse mémoire prédéterminée. Le système doit bien sûr s'assurer lors de l'initialisation faite à la mise en route que les instructions rangées à cette endroit sont bien celles nécessaire à son bon fonctionnement, et il doit aussi protéger cette zone mémoire contre toute écriture qui ne soit pas sous son contrôle. En résumé, un appel système est géré comme une interruption qui a été provoquée par le programme lui-même.

Les interruptions de l'horloge (en anglais : *Timer*) Le schéma précédent ne permet pas au noyau de récupérer le processeur si un programme fait des calculs intensifs, ou part dans une boucle

sans fin. Aussi, un dispositif matériel doit être prévu pour interrompre un processus qui ne fait pas d'appels système. Dans la mesure où les appels systèmes sont faits au moyen d'interruptions, il suffit en fait que ce nouveau dispositif matériel génére des interruptions à des intervalles de temps réguliers et programmés sous le contrôle du système d'exploitation.

5.1.2 La gestion de la mémoire (en anglais : *Memory management*)

Supposons que toute la mémoire physique soit occupée par différents processus, et qu'un utilisateur demande à charger et exécuter un nouveau programme. Dans ce cas, le système d'exploitation doit décider de déplacer (temporairement) un des processus qui est dans la mémoire centrale vers le disque (en anglais : *swap out*) pour faire de la place au nouveau programme. Plus tard ce programme qui a été déplacé sur le disque devra être rechargé en mémoire centrale (en anglais : *swapped in*). Sur un système fortement chargé, quand le nombre et la taille des processus dépasse largement les capacités de la mémoire centrale, la quantité de trafic entre la mémoire centrale et le disque, ainsi que la gestion de ce trafic, dégradent considérablement les performances du système. A la limite, le processeur ne fait plus que passer tout son temps à gérer le *swap* et ne fait plus avancer aucun processus, on parle alors d' *écroulement*. Avec plus de mémoire, il est possible qu'un système passe plus de temps à exécuter des instructions 'ùtiles" et donne ainsi l'impression que le processeur est plus rapide.

La fragmentation (en anglais : *Fragmentation*) Lorsqu'un programme se termine, la mémoire qu'il occupait devient disponible pour un autre processus. Cependant si le candidat au *swap in* est plus grand que léspace laissé par l'ancien, il ne peut être chargé dans cet emplacement. Si au contraire, il est plus petit, il va laisser une zone de mémoire inutilisée. Ainsi au fur et à mesure du chargement des programmes, la mémoire va contenir de plus en plus de petits *trous* éparpillés, aucun d'entre eux nétant assez grand pour accueillir une nouvelle tâche. Lorsque ceci arrive, le système d'exploitation doit compacter les programmes de façons à réunir les différents *trous*. Ainsi les programmes peuvent être déplacés dans la mémoire physique à leur insu.

La mémoire virtuelle (en anglais : *Virtual memory*) Les systèmes plus récents adoptent et implémentenent la *mémoire virtuelle*. Dans ce schéma, les programmes n'ont pas à être complètement en mémoire, et lorsque le besoin de place se fait sentir, ils n'ont pas besoin d'être complètement déplacés sur le disque. Chaque programme est découpé en unités qui ont toutes la même taille (les *pages*) qui peuvent être positionnées indépendamment les unes des autres dans la mémoire physique. Aussi longtemps que la partie active d'un programme est résidente en mémoire centrale, il peut se poursuivre. Cette méthode réduit considérablement l'activité de *swapping* et supprime le problème de la fragmentation. Cependant pour que le processus puisse voir la mémoire comme un tableau d'adresses consécutives alors que les pages vont être dispersées dans la mémoire physique, un dispositif doit exister entre le bus d'adresses du processeur et le bus d'adresses de la mémoire. Ce dispositif (l'*Unité de Gestion Mémoire*, (en anglais : *Memory Management Unit*) convertit les adresses logiques vues par le processus en adresses physiques vues par la mémoire centrale. Si une page nést pas présente en mémoire centrale, il va lever une interruption qui va donner la main au système d'exploitation afin que celui-ci ramène cette page en mémoire centrale.

Le partage du code (en anglais : *Code sharing*) Dans un système multi-tâches, il est fréquent qu'un même programme soit en cours d'exécution plusieurs fois à un instant donné : il a donné lieu à plusieurs processus. Cela peut être le cas pour un éditeur de texte, ou l'outil de consultation du courrier électronique. Pour éviter de gaspiller la mémoire en copiant plusieurs fois le code du programme qui est le même pour tous les processus, celui-ci peut-être partagé[1]. Lorsqu'une mémoire virtuelle est implantée, ceci est facile à réaliser.

[1] Par contre, les données manipulés par plusieurs processus qui partagent un code ne sont pas les mêmes, les zones qui contiennent les données de chaque processus restent indépendantes et protégées les unes par rapport aux autres.

5.1.3 La gestion des entrées/sorties (en anglais : *Input/output management*)

La mise en file d'attente des requêtes disque (en anglais : *Queuing disc requests*) Plusieurs processus peuvent être en attente d'informations en provenance d'un disque. Le système gère une file d'attente des requêtes sur un disque donné, et lorsqu'une requête est satisfaite, il a l'occasion de choisir la prochaine à traiter. Il peut simplement les traiter dans leur ordre d'arrivée avec une *file* (en anglais : First In First, FIFO), ou essayer d'optimiser les déplacements de la tête de lecture du disque dans la mesure où le temps de positionnement des têtes entre pour une grande part dans les temps d'attente d'entrées/sorties sur les disques magnétiques. De plus, lorsque certains processus sont suspendus parce qu'ils sont en attente de la terminaison d'entrées/sorties, d'autres processus qui sont en phase de calcul peuvent être activés et utiliser l'unité centrale.

Le tamponnage des entrées/sorties (en anglais : *Input/output buffering*) Lorsqu'un processus demande une lecture de 16 octets par exemple, il est interrompu pour que sa demande soit satisfaite par le système. Commes les disques opèrent sur des blocs d'une taille plus grande (disons, 512 octets), le système les conservent en mémoire centrale, et à la prochaine lecture il pourra répondre à la requête sans qu'il y ait physiquement une entrées/sortie disque. Lorsque le même principe est utilisé pour lécriture, cela implique que des informations néxistent que dans la mémoire centrale (et volatile !) pendant quelque temps. De temps en temps, il doit y avoir une *synchronisation* entre les informations *tamponnées* dans la mémoire centrale et le disque physique. Cette synchronisation doit aussi être faite bien sûr avant la mise hors tension de l'ordinateur ; ainsi avec ce genre de système, on ne doit pas intempestivement couper l'alimentation électrique d'une machine, mais on doit au contraire respecter une procédure d'arrêt.

5.1.4 Les appels systèmes du système UNIX

5.1.4.1 Gestion des fichiers (en anglais : *File Management*)

fd = creat(name,mode) ; — create a new file or truncate an existing file
fd = mknod(name,mode,addr) ; — create a regular, special, or directory i-node
fd = open(name,how) ; — open a file for reading, writing or both
s = close(fd) ; — close an open file
n = read(fd,buffer,nbytes) ; — read data from a file into a buffer
n = write(fd,buffer,nbytes) ; — write data from a buffer into a file
pos = lseek(fd,offset,whence) ; — move the file pointer somewhere in the file
s = stat(name,&buf) ; — read and return a file's status from its i-node
s = fstat(fd,&buf) ; — read and return an open file's status from its i-node
fd = dup(fd1) ; — allocate another file descriptor for an open file
s = pipe(&fd[0]) ; — create a pipe
s = ioctl(fd,request,argp) ; — perform special operations on special files

5.1.4.2 Gestion des répertoires et du système de fichiers (en anglais : *Directory and File System Management*)

s = link(name1,name2) ; — create a new directory entry, name2 for file name1
s = unlink(name) ; — remove a directory entry
s = mount(special, name, rwflags) ; — mount a file system
s = unmount(special) ; — unmount a file system
s = sync() ; — flush all disk blocks cached in memory to the disk
s = chdir(dirname) ; — change the working directory
s = chroot(dirname) ; — change the root directory

5.1.4.3 Protection (en anglais : *Protection*)

s = chmod(name, mode) ; — change the protection bits associated with a file
uid = getuid() ; — get the caller's uid
gid = getgid() ; — get the caller's gid
s = setuid(uid) ; — set the caller's uid
s = setgid(gid) ; — set the caller's gid
s = chown(name, owner, group) ; — change a file's owner and group
oldmask = umask(complmode) ; — set a mask used to mask off protection bits

5.1.4.4 Gestion du temps (en anglais : *Time Management*)

seconds = time(&seconds) ; — get the elapsed time in seconds since Jan. 1, 1970
s = stime(tp) ; — set the elapsed time in seconds since Jan. 1, 1970
s = utime(file, timep) ; — set the "last access" time for the file
s = times(buffer) ; — get the user and system times used so far

5.1.4.5 Gestion des processus (en anglais : *Process Management*)

pid = fork() ; — create a child process identical to the parent
s = wait(&status) ; — wait for a child to terminate and get its exit status
exit(status) ; — terminate a process execution and return exit status
size = brk(addr) ; — set the size of the data segment to "addr"
pid = getpid() ; — return the caller's process id

5.1.4.6 Gestion des signaux (en anglais : *Signals Management*)

oldfunc = signal(sig, func) ; — arrange for some signal to be caught, ignored etc.
s = kill(pid, sig) ; — send a signal to a process
residual = alarm(seconds) ; — schedule a SIGALARM signal after a certain time
s = pause() ; — suspend the caller until the next signal

5.2 Les exceptions et les interruptions

5.2.1 Les interruptions provoquées par les périphériques d'entrées/sorties

Le système d'exploitation gère les périphériques. Cela signifie qu'il demande aux périphériques d'exécuter des actions. Cependant, léxécution de ces actions nést pas immédiate. D'une façon ou d'une autre, le processeur doit savoir où en sont les exécutions de ces diverses actions.

Le polling Si une méthode par interruption nést pas utilisée, l'alternative est que le système interroge (régulièrement) chaque périphérique pour connaître son état. Cependant cette technique gaspille beaucoup de temps du processeur dans des boucles d'interrogations des périphériques que l'on appelle des *attentes actives*. De plus, si cette boucle d'interrogation est mêlée à des activités de calculs, le processeur ne peut s'apercevoir que beaucoup plus tard qu'une opération a été terminée. Or certains périphériques demandent à être servi tout de suite[2], cela impose que le processeur puisse traiter la fin d'une opération d'entrées/sorties dès quélle est terminée. Cela conduit à la notion d'*interruption*.

[2]Plus précisemment dans un délai borné caractéristique du périphérique ; par exemple sur une ligne série, il faut que le processeur traite le dernier caractère arrivé avant que le suivant ne vienne lécraser.

Les interruptions En fait, une interruption, comme son nom l'indique va interrompre l'activité présente du processeur pour que celui-ci s'occupe d'une tâche plus urgente. Une interruption arrive d'une manière *asynchrone* par rapport au déroulement ordinaire des instructions d'un programme : elle nést pas prévue à l'avance.

On peut faire une analogie avec une personne tranquillement assise sur son canapé et en train de lire un livre (un processeur en train de de dérouler les instructions d'un programme). Lorsqu'un coup de sonnette à la porte d'entrée vient l'interrompre : il nétait pas prévu à l'avance par l'auteur du livre entre quelles phrases ce coup de sonnette allait se produire : *asynchronisme*. Notre lecteur exécute alors un programme associée à l'interruption 'çoup de sonnette" : il met un signet dans son livre (*sauvegarde du contexte*), va ouvrir la porte et parle avec son hôte,. . .—cette discussion peut à son tour être interrompue par un coup de téléphone— lorsque celui-ci repart, il peut alors reprendre (*restauration du contexte*) son livre et continuer sa lecture.

Ces interruptions permettent à un périphérique d'informer le processeur qu'il a terminé une opération d'entrée/sortie et qu'il est prêt pour la prochaine.

Très souvent les interruptions sont gérés par niveaux, certaines affaires urgentes sont plus urgentes que d'autres, il y a certaines actions que le processeur effectue tout de suite parce qu'il ne gagnerait rien à les remettre à plus tard (comme transférer le bloc lu par le disque de son tampon vers la mémoire centrale) et d'autres qui doivent être achevées dans le délai imparti défini par le périphérique.

Le mécanisme d'interruption commence par un aspect matériel (il faut qu'il y ait une sonnette à la porte d'entrée), qui prend la forme pour un processeur d'une ou plusieurs lignes d'entrées sur le processeur lui-même. Lorqu'un périphérique veut déclencher une interruption, il positionne ces lignes dans un certain état électrique. Ce mécanisme continue par une prise en compte de ces informations électriques au niveau de la micro-machine : avant de commencer une nouvelle instruction le processeur consulte ces lignes et si leur état signale une interruption, il exécute une série d'actions qui ont pour but de sauvegarder létat courant du processus et se met lui-même dans un état propre à exécuter les routines de *service de l'interruption* :

- sauvegarde dans la pile du compteur ordinal et du registre d'état,
- passage en mode *superviseur*,
- chargement du *vecteur d'interruption* dans le compteur ordinal.

Le routine d'interruption est alors exécutée, et cést une suite normale d'instructions. . .qui peut elle-même être interrompue par une interruption de niveau plus élevée. Cette suite d'instructions doit se terminer par une instruction spéciale (`IRET` sur les 80X86 (*RETurn from Interruption*)) qui ressemble à l'instruction qui termine léxécution d'une routine ordinaire (RET sur les 80X86 (*RETurn from subroutine*)), mais cette instruction doit aussi dépiler l'ancienne valeur du registre d'état :

- restauration du compteur ordinal et du registre d'état avec les valeurs conservées dans la pile (ce qui remettra éventuellement le processeur en mode utilisateur).

5.3 Gestion mémoire et protection

Cette section va nous permettre de préciser les notions déjà évoquées de *mémoire virtuelle*, d'*adresses logiques*, et d'*adresses physiques*. Le processeur, avec ses multiples modes d'adressages, calcule une *adresse effective*. Cette adresse effective se référe à un espace mémoire linéaire contigu dont les cellules sont indifférenciées. Dans ce schéma, les adresses logiques, vue par le processeur et le programme qu'il est en train d'interpréter, et les adresse physiques, vues par la mémoire centrale, sont identiques :

adresse physique = adresse logique

Cette relation peut se généraliser sous la forme :

adresse physique = map(adresse logique)

En fait, le besoin d'une gestion mémoire est soulevé par plusieurs raisons :

- un programme a besoin d'utiliser plus de mémoire que ce qu'il y a effectivement en mémoire centrale,

- plusieurs processus partagent le même espace de mémoire physique, et veulent tous voir un espace contigu numéroté à partir de 0,
- chaque processus veut être protégé en écriture (et peut-être même en lecture) des autres processus (en particulier, le système d'exploitation lui-même doit être protégé des processus utilisateurs).

Le premier point est résolu grâce à la notion de mémoire virtuelle[3], dont le principe est basé sur léxistence d'un deuxième niveau de mémoire : le disque magnétique.

Le deuxième point nécessite un mécanisme de translation dynamique aussi bien lors du chargement de nouveaux programmes que lors des phases de *compactage* déjà évoquées. Dans ce cas, la fonction qui associe une adresse physique à une adresse logique est simplement :

$$p = \text{base} + l$$

Quant au troisième point, il est résolu par des *attributs d'accès* sur chacunes des zones mémoires.

Que l'on utilise un mécanisme de mémoire virtuelle ou une simple translation dynamique d'adresse, il y a une correspondance entre les adresses logiques et les adresses physiques. Pour la simplification de cette fonction entre les deux espaces d'adresses, ce nést pas chaque adresse logique qui se voit attribuée une adresse physique, mais la correspondance se fait par zones contigues d'adresses logiques et physiques.

5.4 Exercices

Exercice 5.1 (Ordonnancement des tâches) *On considère un système multi-utilisateur, multi-tâches sur lequel trois processus sont actifs :*

processus A *un éditeur de texte, celui-ci est le plus souvent en attente de lecture de caractères sur le clavier, et on supposera que le traitement du caractère une fois tapé nécessite 1000 instructions,*

processus B *un programme de calcul intensif, qui ne fait pas d'entrée-sortie,*

processus C *un compilateur qui lit et écrit des fichiers et a besoin d'un bloc de données depuis le disque toutes les 50000 instructions, le disque répond à ces demandes en 30 millisecondes.*

On suppose que le processeur qui exécute ces différents processus exécute toutes ses instructions en 1 micro-seconde, et que si un processus ne fait pas d'appel système, il ne peut malgré tout pas être actif plus de 40 millisecondes à la suite. On suppose aussi que chaque exécution du système dure exactement 3 millisecondes.

Question 5.1.1 *Dessinez sur un diagramme l'activité des différents processus et du système d'exploitation. On fera apparaître en fonction du temps quel processus est actif.*

Question 5.1.2 *Est-ce que le processus de calcul (le processus B) ralentit considérablement les deux autres processus ?*

Question 5.1.3 *Sur un système monotâche (donc mono-utilisateur), que se passe-t'il lorsque léditeur de texte est actif ?*

[3]En fait, ce principe ne résoud le problème que pour les programmes qui ont une certaine localité d'accès à leurs données. Ainsi, pour un très grand tableau bidimensionnel, il peut y avoir une très grande différence de temps d'exécution entre un programme avec deux boucles imbriquées et le même programme où l'ordre d'imbrication des deux boucles est inversé.