9 Le système

Les sections précédentes ont introduit la structure élémentaire des processeurs, et les prochaines sections montreront comment on est progressivement parvenu aux processeurs complexes actuels. Le but de cette section est d'introduire l'environnement du processeur, c'est-à-dire la structure d'un système complet. On illustrera notamment cette notion de système avec le PC.

Un système complet a deux composantes, l'une matérielle, l'autre logicielle. La composante matérielle comprend le processeur, les périphériques et les bus permettant à tous les composants de communiquer entre eux. La composante logicielle, le système d'exploitation, permet d'offrir à l'utilisateur une vision abstraite et simplifiée du fonctionnement du système matériel ; elle permet également de gérer l'ensemble des ressources du système matériel (processeur, mémoire, périphériques).

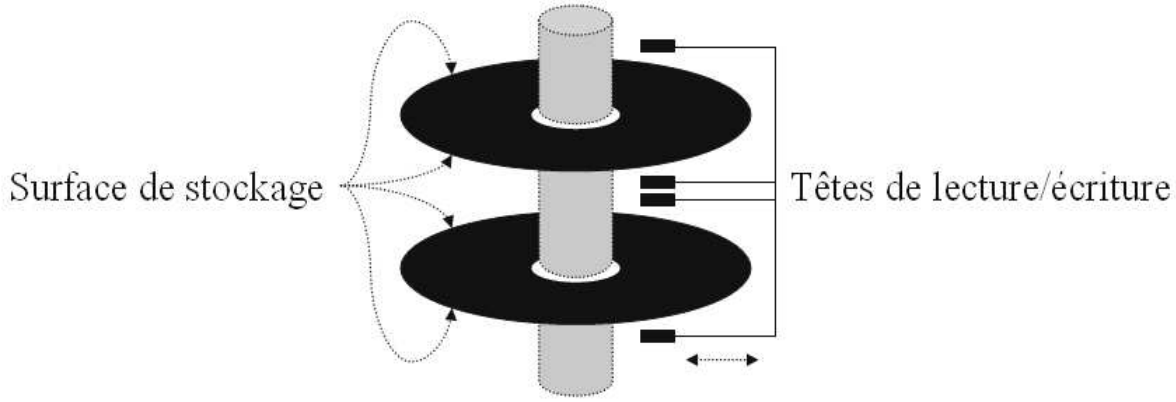
9.1 Les entrées/sorties

Le contrôleur de périphérique. Un périphérique d'entrée/sortie contient en fait deux parties : un appareil (clavier, écran, disque...) et un **contrôleur de périphérique**. Le contrôleur sert d'interface entre le périphérique et le processeur : il reçoit les requêtes du processeur et les transforme en commandes pour le périphérique, et réciproquement, il envoie les requêtes du périphérique au processeur. Il gère également les transferts de données entre le processeur et le périphérique. Les principales requêtes qu'un processeur envoie à un périphérique sont « lecture » et « écriture ». Selon les périphériques, ces requêtes peuvent correspondre à des commandes simples ou complexes.

On considère qu'il y a deux grandes catégories de périphériques : les périphériques par caractères et les périphériques par blocs. Dans un périphérique par blocs, on ne peut accéder à l'information que par blocs et chaque bloc possède une adresse. Dans un périphérique par caractère, on accède bien sûr à l'information caractère par caractère, mais on ne peut spécifier une adresse ni rechercher une information : on reçoit ou on envoie simplement un flux de caractères. Le clavier est un exemple de périphérique par caractères, tandis que le disque est un exemple de périphérique par blocs.

Exemple : le clavier.
Le clavier est l'un des périphériques les plus simples d'un système. Le clavier est relié à l'ordinateur par le biais d'un lien série (cas d'une interface RS-232). Lorsqu'une touche est frappée, le clavier transmet le caractère ASCII correspondant sur le lien, bit par bit. Au niveau de l'ordinateur, le lien série est relié à un circuit qui va effectuer la traduction d'une séquence de bits (8 bits et les bits de signalisation) en un caractère. Ce circuit est lui-même relié à un contrôleur associé au lien série. Lorsqu'un caractère est transmis, le contrôleur va le signaler au processeur qui va ensuite lire la donnée correspondante.

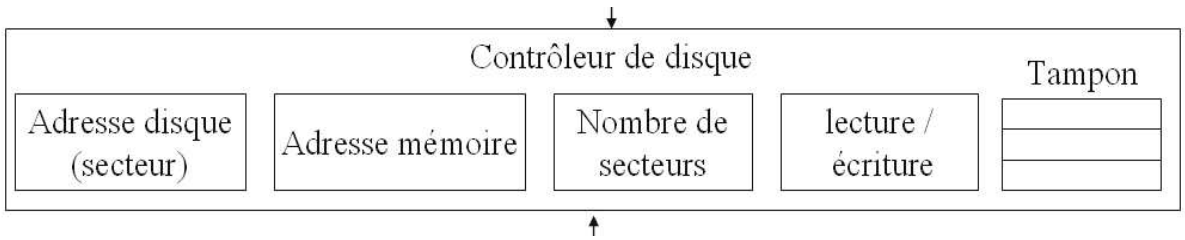
Exemple : le disque.



Le disque est un périphérique nettement plus complexe que le clavier. Un disque comporte aujourd'hui plusieurs plateaux, chaque plateau contient un ensemble de pistes concentriques, les même pistes de différents plateaux forment un cylindre ; enfin, chaque piste comporte plusieurs secteurs. Pour accéder à une donnée, il faut donc préciser le cylindre, la piste et le secteur. Le contrôleur du disque offre une vue abstraite au système d'exploitation qui peut voir un disque comme une suite d'un grand nombre de secteurs contigus. Lors d'une requête en lecture, le système d'exploitation précise le numéro du premier secteur et le nombre de secteurs à lire. Le contrôleur de disque positionne la tête de lecture correspondante sur une piste ; lorsque le secteur requis passe sous la tête de lecture, la donnée est envoyée au contrôleur du disque qui signale sa disponibilité au processeur, et celui-ci vient ensuite la lire.

Les registres d'entrée/sortie. Chaque contrôleur dispose en général de plusieurs registres pour communiquer avec le processeur. Par exemple, pour le contrôleur du clavier, deux registres suffisent : l'un signalant qu'un caractère est disponible, l'autre contenant le caractère lui-même.

Exemple : les registres du contrôleur de disque.



Le registre « adresse disque » permet au processeur de spécifier le numéro du premier secteur à lire ; le registre « adresse mémoire » permet de spécifier au contrôleur l'emplacement des données à écrire sur le disque ; le registre « lecture/écriture » permet de spécifier la nature de l'opération ; le tampon permet de conserver les données en attendant leur lecture par le processeur ou leur écriture par le disque.

Selon les systèmes matériels, les registres d'entrée/sortie correspondent réellement à des zones mémoire spécifiques distinctes de la mémoire principale, ou à des adresses spécifiques en mémoire principale. Dans le premier cas, il faut des instructions spéciales pour accéder à ces zones mémoire, tandis que dans le second cas, on peut utiliser les instructions d'accès mémoire classiques.

Driver. Chaque contrôleur de périphérique a des commandes spécifiques. Comme un concepteur de système d'exploitation ne peut inclure dans son logiciel l'ensemble des commandes de l'ensemble des périphériques, il existe différentes couches logicielles pour réaliser l'interface entre le système d'exploitation et les périphériques. Du point de vue du système d'exploitation, les périphériques n'ont besoin que d'être lus ou écrits, ce qui constitue l'interface de plus haut niveau. Ensuite, la façon d'accéder à un périphérique diffère suivant qu'il s'agit d'un périphérique par blocs (plusieurs blocs adressables indépendamment) ou par caractères (flux de caractères) ; il existe donc une interface de bas niveau pour ces deux types de périphériques. Enfin, la couche logicielle de plus bas niveau est spécifique au périphérique : elle traduit des commandes générales des périphériques par blocs ou par caractères en commandes spécifiques au périphérique cible. Cette dernière couche logicielle s'appelle un ***driver*** ; elle est généralement fournie par le constructeur du périphérique et est insérée dans le système d'exploitation. À l'origine, le système d'exploitation ne contient donc qu'une vue abstraite du système matériel.

Accès à un périphérique. Il existe trois façons de communiquer avec un périphérique :

- en testant de façon continuelle le périphérique pour vérifier que des données peuvent être lues ou écrites par le processeur (***polling***) ;
- en interrompant le processeur lorsqu'un périphérique est prêt à lire ou écrire des données (**interruptions**) ;
- en établissant une communication directe entre deux périphériques (**DMA**).

Exemple : utilisation du polling pour l'accès à un clavier.
On suppose que l'on dispose de deux registres KBDR et KBSR dont le rôle est décrit ci-dessous :

- *KBDR: code de la touche frappée*
- *KBSR: registre d'entrée (zone mémoire)*
  - *KBSR[15] = 1 si une touche a été frappée mais KBDR non encore lu*
  - *KBSR[15] = 0 lorsque KBDR est lu*

On peut alors programmer une boucle qui attend continuellement qu'une touche ait été frappée pour lire le caractère correspondant. Le code correspondant est indiqué ci-dessous :

```
DEBUT       LDI   R1, @KBSR   ; Lecture du registre
            BRzp  DEBUT       ; d'état.
            LDI   R0, @KBDR   ; Lecture du code de la touche.
            BR    SUITE
@KBSR       .FILL xF400
@KBDR       .FILL xF401
```

Bien que le *polling* soit une façon simple d'accéder à un périphérique, le débit des données dans un périphérique est parfois beaucoup plus lent que la vitesse de fonctionnement d'un processeur, et le *polling* peut donc être très inefficace. On lui préfère en général la méthode par interruptions.

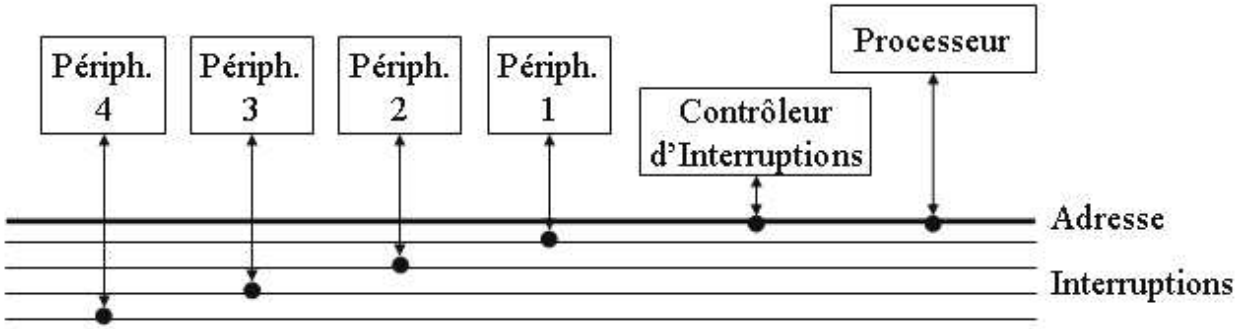
Lorsqu'un périphérique reçoit des données pour le processeur, il envoie au processeur un signal d'interruption. Si celui-ci peut être interrompu (par exemple, s'il n'est pas en train de communiquer avec un périphérique de plus haute priorité), il stoppe la tâche en cours, sauve son état en mémoire et appelle la routine système correspondant au numéro d'interruption (envoyé en même temps que le signal d'interruption). La routine système fait appel au driver

du périphérique, lit ou écrit les données sur le périphérique, puis signale au processeur que l'interruption est terminée, et que la tâche stoppée peut reprendre.

Exemple : utilisation des interruptions pour accéder au clavier.
On suppose ici que lorsque KBSR passe à 1, une interruption est déclenchée. Le processeur exécute alors la routine suivante qui se contente de lire la donnée dans le registre KBDR :

```
DEBUT       LDI   R0, @KBDR   ; Lecture du code de la touche.
            RTI
@KBDR .FILL .FILL xF401
```

Le fonctionnement des périphériques est généralement asynchrone. Plusieurs interruptions peuvent donc être déclenchées en même temps. Il existe souvent un contrôleur d'interruptions destiné à gérer les conflits entre les interruptions, et à déterminer si le processeur est capable d'accepter une interruption. Chaque contrôleur de périphérique dispose par exemple d'une ligne spécifique sur laquelle il signale son interruption, ou alors d'une ligne commune et de lignes d'adresses pour indiquer l'origine de l'interruption.



En cas de transfert de grandes quantités de données, par exemple d'un disque vers un autre disque, il peut être plus efficace de laisser communiquer entre eux les deux périphériques plutôt que de solliciter le processeur pour chaque ensemble de blocs. Ce type d'entrées/sorties est appelé DMA (*Direct Memory Access*), et il est généralement géré par un contrôleur de DMA (ce contrôleur peut être intégré au périphérique lui-même, mais il s'agit plus généralement d'un contrôleur du système). Le processeur indique au contrôleur de DMA quels sont les périphériques qui doivent communiquer, le nombre et éventuellement l'adresse des données à transférer, puis il initie le transfert. Le processeur n'est pas sollicité durant le transfert, et une interruption signale au processeur la fin du transfert.

Afin de limiter les risques de mauvaise utilisation d'un périphérique, un utilisateur ne peut jamais directement accéder aux zones mémoire correspondant aux registres des périphériques. Aussi, il dispose d'une instruction spéciale, appelée TRAP dans le LC-2, qui lui permet de spécifier la routine correspondant au périphérique à utiliser. Le numéro fourni en paramètre de l'instruction est en fait un index vers une table contenant la première adresse de chacune des routines système; on appelle ce paramètre le vecteur d'interruption.

9.2 Les processus

Le système d'exploitation, également appelé le noyau, permet d'accéder à l'ensemble des ressources matérielles de l'ordinateur. Lorsqu'une partie du système d'exploitation est exécutée sur le processeur, on dit que l'exécution se fait en mode noyau ; tout autre

programme est exécuté en mode utilisateur, qui ne permet notamment pas l'accès aux ressources matérielles de l'ordinateur.
Un processus est un programme en cours d'exécution et tout l'environnement système qui lui est associé (le code binaire du programme, l'espace mémoire destiné aux données, la pile du programme, les descripteurs de fichiers en cours d'utilisation par le programme...). Un des principaux buts des systèmes d'exploitation tel que Unix est de permettre la multiprogrammation, c'est-à-dire l'utilisation simultanée des ressources matérielles de l'ordinateur par plusieurs utilisateurs. Bien qu'un seul processus puisse être exécuté à la fois sur le processeur, le système d'exploitation permet de donner l'illusion aux utilisateurs qu'ils y accèdent simultanément. Pour ce faire, le système d'exploitation n'alloue le processeur à un processus que pendant un temps restreint appelé quantum. Lorsque le quantum de temps d'un processus est écoulé ou alors que le processus devient inactif, le système d'exploitation alloue le processeur à un autre processus. Auparavant, il sauve l'état du processus courant sur la pile du noyau (registres matériels, PC, pointeur de pile, PSW,...). Cette opération est appelée un **changement de contexte**.

Création de processus. Un processus peut être créé soit au démarrage du système, soit par un autre processus. Par exemple les démons (email, *cron*,…) sont des processus créés au démarrage qui attendent des événements. La façon la plus classique de créer un processus en Unix est l'appel système *fork*. Un processus fils est créé qui est une copie exacte du processus père : mêmes programme, données, fichiers, variables d'environnement. Cependant, les deux processus ont des espaces d'adressage distincts : une modification dans une donnée du processus père n'affecte pas le processus fils. Cependant, comme la copie est une opération coûteuse, en pratique le processus fils pointe sur les données du processus père et la copie n'a lieu que pour les adresses mémoire où le processus fils désire écrire. Après le *fork*, on exécute en général l'appel système *exec* qui a pour effet de charger un nouveau programme dans le processus fils.

États d'un processus. Un processus peut avoir trois états possibles :

- en cours d'exécution
- prêt (le processus peut s'exécuter mais le processeur n'est pas disponible)
- bloqué (le processus attend un événement)

Par exemple, un processus peut être bloqué lorsqu'il communique avec un autre processus. En Unix, de processus peuvent communiquer par le biais de canaux, appelés **pipes** (e.g., « *ls – l | more* »). Afin d'assurer la synchronisation entre les deux processus, un processus qui lit un canal est bloqué tant que le canal est vide.

Signaux. Deux processus peuvent également communiquer par le biais de signaux. Le signal est analogue à une interruption logicielle : lorsqu'un processus reçoit un signal, il stoppe son exécution, et branche à une routine destinée à traiter ce signal ; en fin de traitement, le processus recommence son exécution à l'endroit où il s'était arrêté. Un processus peut en fait avoir plusieurs attitudes à la réception d'un signal : il peut ignorer, prendre fin, ou traiter le signal s'il dispose d'une routine ad hoc.

Exemples de signaux en Unix.

SIGABRT: fin du processus et core dump
SIGPIPE: écriture vers un pipe sans processus lecteur
SIGSEGV: accès à une adresse mémoire invalide
...

Appels système. Lorsqu'un processus effectue un appel système, c'est-à-dire qu'il appelle une routine du système d'exploitation, le processus ne s'arrête pas mais passe en mode noyau. Il dispose alors de droits beaucoup plus importants, et peut notamment accéder à l'ensemble des ressources matérielles. En outre, le processus dispose alors d'une autre pile, située dans l'espace mémoire du noyau.

Exemple d'appel système.
L'appel waitpid attend la fin d'un processus fils ; il dispose de 3 paramètres décrits ci-dessous.

```
waitpid(pid_child, status_child, stall)
```

- *pid_child : PID du processus fils (-1 si quelconque)*
- *status_child : statut du processus fils en fin d'exécution*
- *stall : appel bloquant ?*

Mis en oeuvre des processus. Au sein du noyau, chaque processus dispose d'une entrée dans une **table des processus**. Cette table contient les informations dont le système doit pouvoir disposer à tout moment sur l'ensemble des processus :

- Image mémoire (programme, données, pile…)
- Signaux (masques indiquant comment traiter des signaux)
- Informations utiles au séquencement des processus (priorité, temps processeur déjà utilisé,…)
- Informations générales (état du processus, PID, identification, événements en attente…)

Outre cette table, le système d'exploitation dispose également d'une seconde structure qui contient des informations uniquement utiles lorsque le processus est en cours d'exécution (et qui n'est donc pas nécessairement en mémoire à tout instant) :

- Registres matériels
- État de l'appel système (paramètres, résultat)
- Fichiers utilisés par le processus
- Pile noyau utilisée par le processus

Les *Threads*. Le fait que les processus aient tous un espace d'adressage distinct constitue parfois une forte limitation. Par exemple, dans le cas d'un traitement de texte, on veut pouvoir faire deux actions simultanées sur les mêmes données : entrer des caractères à l'aide d'un clavier et appliquer une vérification orthographique sur les derniers mots rentrés. Dans ce but, les systèmes d'exploitation intègrent des ***threads***. Un thread est un flot d'exécution ; un processus peut contenir plusieurs threads. Un thread est également appelé un « processus léger ». Les seules informations locales à un thread sont : le PC, les registres, la pile d'exécution et son état.

Implémentation d'un thread. Les threads peuvent être implémentés soit dans le noyau soit dans l'espace utilisateur. Implémenter les threads dans l'espace utilisateur permet de les ajouter à n'importe quel système d'exploitation existant, sans modifier le noyau. En outre, le passage d'un thread à l'autre est beaucoup plus rapide qu'un changement de contexte puisqu'il n'y a pas d'appel au noyau. En revanche, lorsqu'un thread engendre un appel système qui va bloquer le processus, l'ensemble des threads du processus seront également bloqués. Il s'agit

d'une très forte limitation de ce type d'implémentation puisqu'un des principaux intérêts des threads est de permettre à un thread d'utiliser le processeur lorsqu'un autre thread est bloqué.
Il existe aussi des implémentations hybrides des threads : à la fois dans le noyau et dans l'espace utilisateur. Chaque processus se voit attribuer plusieurs threads dans le noyau, et chaque thread noyau peut gérer plusieurs threads utilisateur. Lorsqu'un thread utilisateur est bloqué pour un appel système, un seul thread noyau est bloqué.
L'implémentation des threads pose d'autres problèmes complexes : la gestion des variables globales (elles sont visibles et donc modifiables simultanément par tous les threads), la gestion des signaux (quel(s) thread(s) reçoit le signal d'un processus ?), la gestion de la pile du processus...

Ordonnancement de processus. Un des rôles essentiels du système d'exploitation est de gérer la principale ressource de l'ordinateur : le processeur. Le principe est de sélectionner à tout instant le prochain processus que doit exécuter le processeur. Les objectifs de l'ordonnancement de processus sont de maximiser l'utilisation du processeur, et d'assurer un équilibre entre les besoins des différents processus en cours d'exécution. Typiquement, l'exécution d'un processus apparaît comme un enchaînement de séquences d'exécution sur le processeur et d'accès aux entrées/sorties. Pour des questions d'efficacité, lorsqu'un processus est en attente d'entrées/sorties, il libère le processeur.
En général, le système d'exploitation maintient plusieurs listes de processus :

- ***Ready Queue*** : une liste des processus prêts à être exécutés et en attente.
- ***Wait Queue*** : une liste des processus en attente d'un événement (provenant par exemple d'un autre processus).
- ***Device Queues*** : une liste des processus en attente d'accès à un périphérique.

Plusieurs critères d'ordonnancement sont souvent utilisés simultanément :

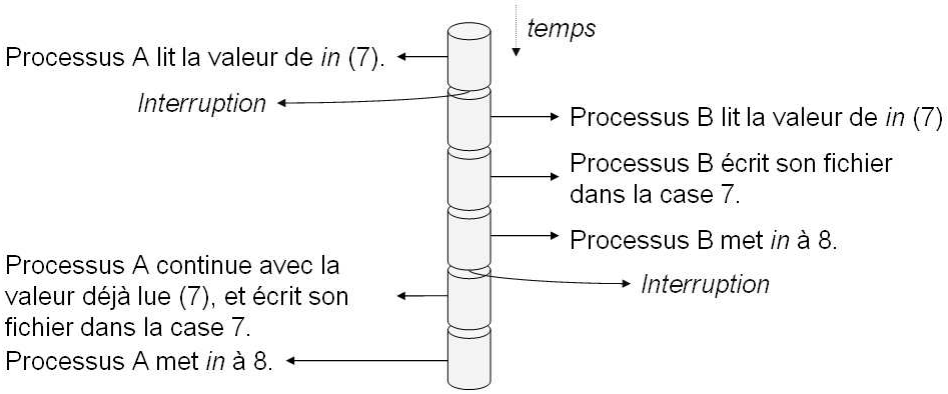
- le taux d'utilisation du processeur (on peut maximiser l'utilisation du processeur)
- le débit de terminaison des processus (on peut maximiser le nombre de processus terminés par seconde)
- le temps total d'exécution d'un processus (il faut équilibrer l'utilisation du processeur entre les différents processus)
- le temps d'attente d'un processus (lorsqu'il est prêt à être exécuté)
- le délai de réponse (dans le cas des commandes interactives).

Selon les applications de l'ordinateur (calcul intensif, applications interactives, temps-réel,...), différents algorithmes d'ordonnancement peuvent être préférés :

- Premier arrivé, premier servi.
- Plus court processus d'abord (cet algorithme suppose que l'on estime le temps d'exécution du processus).
- Utilisation de priorités (ces priorités sont déterminées en général statiquement par l'utilisateur, mais elles peuvent varier dynamiquement au cours de l'exécution, selon la charge ou la durée d'exécution du processus).
- Quantum de temps (on répartit l'utilisation du processeur en intervalles fixes de temps ; actuellement, ces intervalles ont une durée comprise entre 10 et 100 millisecondes ; la durée du quantum varie en général avec le nombre de processus en attente).
- Files d'attente multiples (on répartit les processus en fonction de leurs caractéristiques ; par exemple, on groupe les processus qui nécessitent beaucoup de calculs, et ceux qui nécessitent beaucoup d'entrées/sorties ; on peut appliquer des algorithmes d'ordonnancement différents pour chaque file d'attente).

Coopération entre processus. Dans un grand nombre de cas, des processus doivent partager des ressources communes, par exemple des zones mémoire ou des fichiers communs. Cette nécessaire coopération peut engendrer de nombreux conflits.

Exemple : conflits d'accès au répertoire des impressions.
Deux processus A et B veulent accéder au répertoire des impressions. Ce répertoire est un tableau dont chaque case contient le nom d'un fichier à imprimer ; les cas sont numérotés. Le répertoire comprend deux paramètres in (qui indique le numéro, dans le répertoire, de la première case libre) et out (qui indique le numéro, dans le répertoire, de la prochaine case à envoyer à l'imprimante). On considère le scénario suivant :



Aucune incohérence ne sera détectée, mais l'impression du fichier du processus B est perdue.

Ces **situations de concurrence** interviennent lorsque deux processus veulent accéder simultanément à une zone mémoire partagée (la variable *in* dans l'exemple précédent). Une séquence d'accès particulière peut alors engendrer un comportement incorrect du système d'exploitation et donc des processus. Une **section critique** correspond à la période d'un processus où il accède à une zone mémoire partagée. L'**exclusion mutuelle** est un mécanisme permettant d'assurer qu'un seul processus à la fois se trouve dans une section critique. Il existe un grand nombre de solutions pour régler ces situations de concurrence, mais il peut être difficile de trouver une solution efficace du point de vue de l'utilisation des ressources du système. Nous présentons ci-dessous deux des principales solutions : l'**attente active** (***busy waiting***) et les **sémaphores**.

Les deux programmes ci-dessous illustrent le principe de l'attente active : chaque programme est exécuté par un processus différent, et les deux processus testent la même variable avant de rentrer dans une section critique, et si la variable ne contient pas la valeur adéquate, ils bouclent continuellement.

```
while(TRUE) {                  while(TRUE) {
  while (turn != 0);             while (turn != 1);
  critical_region();             critical_region();
  turn = 1;                      turn = 0;
  noncritical_region();          noncritical_region();
}                              }
```

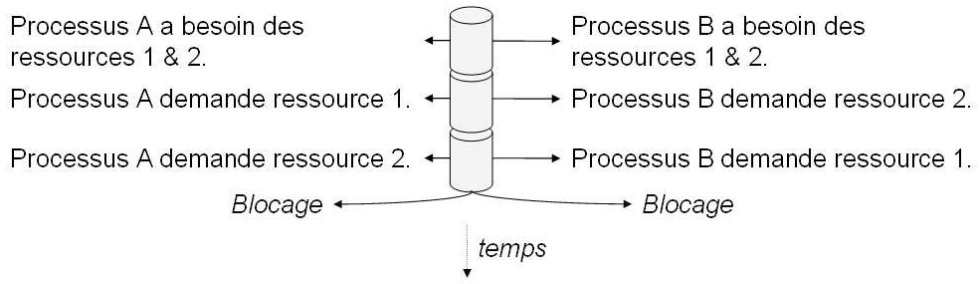
Un sémaphore est une variable entière positive ou nulle. Outre le sémaphore, on dispose en général de deux instructions, telles que *down* et *up*. *down* (sémaphore) a pour effet de décrémenter la valeur du sémaphore ; si cette valeur était égale à 0, alors le processus est bloqué (il n'utilise donc plus le processeur). *up* (sémaphore) a pour effet d'incrémenter la valeur du sémaphore ; si un processus était bloqué après une instruction *down*, il est débloqué.

Si plusieurs processus étaient bloqués, un seul peut être débloqué à la fois (un seul peut accéder au sémaphore). Une caractéristique très importante des instructions *up* et *down* est qu'elles sont **atomiques**, c'est-à-dire que l'action qu'elles conduisent ne peut être interrompue de façon à ce qu'aucun autre processus ne puisse accéder au sémaphore pendant cette action.
Le programme ci-dessous illustre le principe des sémaphores :

```
int semaphore = 1;
while(TRUE) {
   down(&semaphore);
   critical_region();
   up(&semaphore);
   noncritical_region();
}
```

L'introduction de l'exclusion mutuelle engendre un nouveau type de conflits : les **interblocages** (***deadlocks***). Un interblocage intervient lorsque deux processus A et B sont tels que A attend un événement de B et B attend un événement de A.

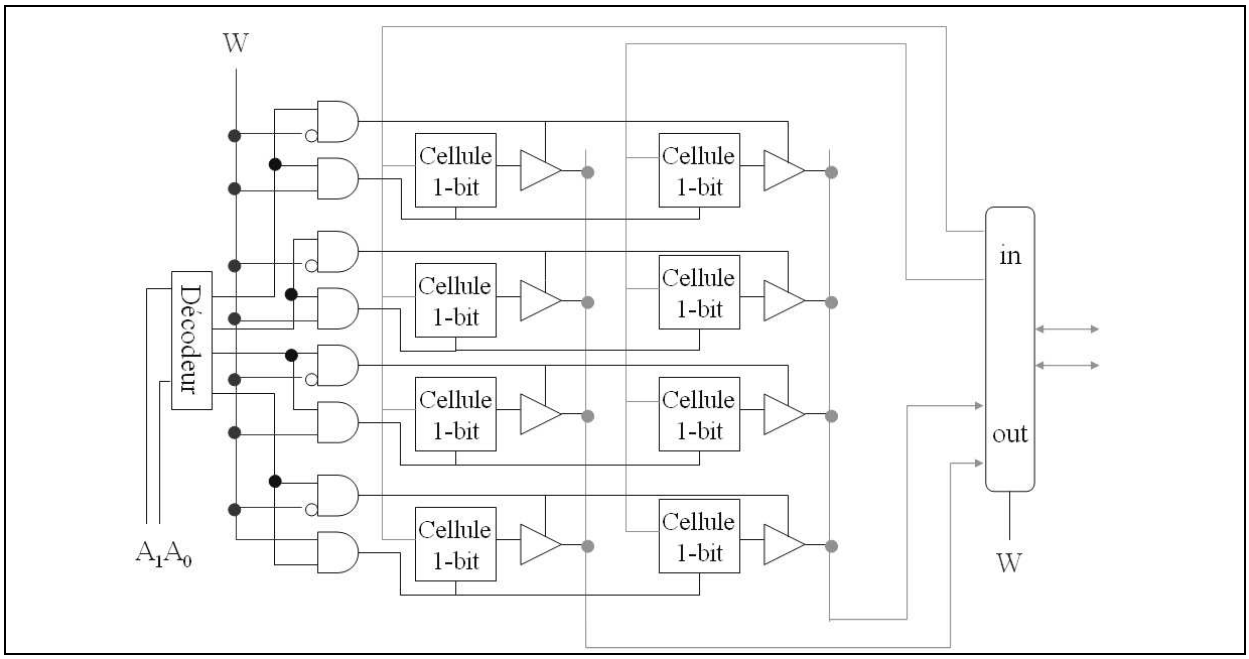
Exemple : interblocage lors de l'accès simultané à deux ressources.



9.3 La mémoire

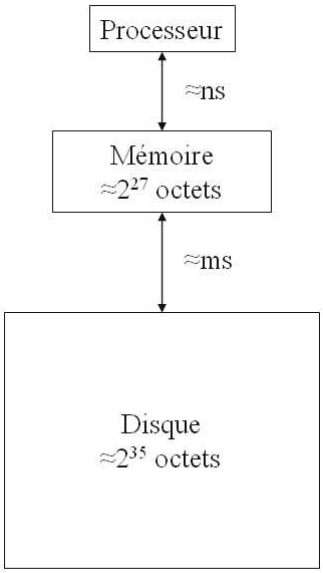
Mémoire Physique. La structure d'une mémoire physique est proche de celle d'un banc de registres, du moins dans ses principes plus que dans son implémentation. Une mémoire peut être vue comme un grand tableau de 2^n octets ; le numéro de ligne dans le tableau correspond à l'**adresse** dans la mémoire. Chaque ligne comporte alors 8 cellules de mémoire 1-bit dont les 8 bits sont envoyés simultanément sur la sortie de la mémoire en cas de lecture. Dans chaque colonne, les 2^n cellules 1-bit sont reliées à la même ligne 1-bit, comme pour un bus (mais l'implémentation est différente). Lorsque les n bits de l'adresse parviennent à la mémoire, ils sont envoyés à un décodeur, un circuit qui transforme une valeur sur n bits en une valeur sur 2^n bits où seul le bit x vaut 1, lorsque la valeur sur n bits est égale à x. De même, en cas d'écriture, le décodeur permet de sélectionner la ligne cible, et d'inhiber l'écriture dans les autres lignes.

Exemple : une mémoire de 4 lignes de 2 bits.



En pratique, des contraintes sur le nombre de pattes d'une puce imposent une structure différente pour les mémoires, notamment les DRAMs. Le nombre de pattes étant essentiellement déterminé par *n*, on agence la mémoire comme un tableau bidimensionnel de *n/2* lignes et *n/2* colonnes de cellules de 8 bits, afin de réduire le nombre de pattes. L'adresse est alors envoyée en deux étapes à travers *n/2* pattes.

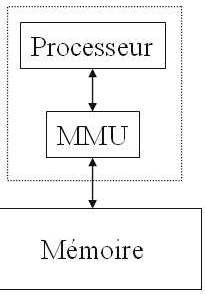
Mémoire virtuelle. L'espace mémoire que peut adresser un programme est déterminé par la taille des mots. Avec un mot de 64 bits, un processus peut disposer d'un espace mémoire de 2^{64} octets, soit 16 exa-octets. Bien entendu, pour des questions de coûts et de difficultés de conception, la taille de la mémoire physique ne peut être la même que celle de l'espace mémoire adressable. Cependant, beaucoup de programmes, notamment dans le domaine du calcul scientifique et de la modélisation numérique, exigent des volumes mémoire bien supérieurs à ceux de la mémoire physique (au plus quelques giga-octets sur une station de travail). Pour résoudre ce problème, la plupart des systèmes informatiques implémentent maintenant une **mémoire virtuelle**.

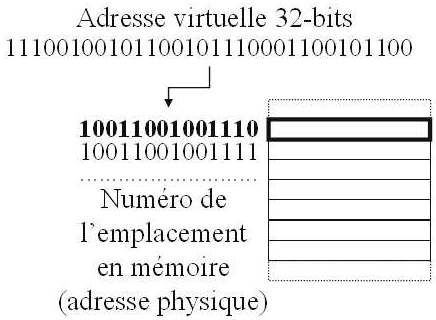


Hiérarchie Mémoire

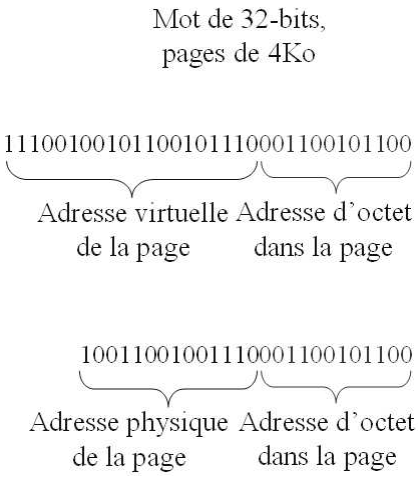
Le principe est de donner l'illusion à l'utilisateur qu'il dispose d'une mémoire de la même taille que celle de l'espace mémoire adressable. La mémoire physique est alors utilisée comme un tampon contenant les données et les parties du programme les plus fréquemment référencées. Si nécessaire, le disque peut être utilisé pour contenir les autres données et instructions.

C'est le système d'exploitation qui va permettre de mettre en oeuvre cette mémoire virtuelle, mais un support architectural est nécessaire ; ce support est appelé une MMU (*Memory Management Unit*). Le processeur n'a pas connaissance de la distinction entre mémoire physique et mémoire virtuelle, il considère qu'il dispose d'une mémoire de la même taille que celle de l'espace mémoire adressable. Il utilise donc des adresses virtuelles. Cependant, lorsqu'il effectue une requête mémoire, il faut alors trouver l'emplacement, i.e., l'adresse, dans la mémoire physique, d'une donnée dont on ne connaît que l'adresse virtuelle. Une donnée présente en mémoire physique a donc deux adresses : son adresse virtuelle, et son emplacement en mémoire physique, c'est-à-dire une adresse physique. Le rôle de la MMU est d'assurer la traduction des adresses virtuelles (des requêtes du processeur) en adresses physiques.

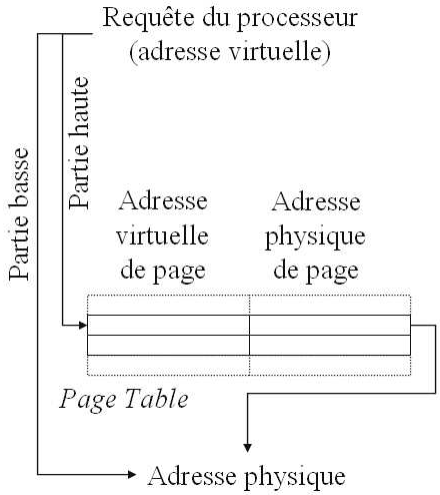




Cependant, conserver une traduction pour chaque octet de la mémoire physique serait très inefficace. Aussi, le système d'exploitation considère que la mémoire est découpée en **pages**, une page étant un bloc d'octets contigus (la taille d'un bloc varie de 512 octets à 64 ko selon les systèmes d'exploitation). L'adresse d'un octet comporte alors deux parties : la position de l'octet dans la page et l'adresse de la page ; la position de l'octet dans la page est commune à l'adresse physique et à l'adresse virtuelle ; en revanche, il existe une adresse physique de la page et une adresse virtuelle de la page. La MMU a donc pour but de traduire les adresses des pages.



Les traductions de l'ensemble des pages présentes en mémoire physique sont contenues dans une table (*Page Table*), elle-même stockée dans la mémoire physique. Cependant, même avec le mécanisme des pages, la taille de la table grandit rapidement avec la taille du mot. Aussi, on utilise maintenant des tables à deux niveaux et seule une partie de la table est contenue en mémoire physique.



Outre la traduction physique/virtuelle, la table des pages est utilisée par le système d'exploitation pour d'autres fonctions :

- pour la protection entre processus ; elle stocke les informations sur les droits en lecture, écriture et exécution ;
- pour une utilisation efficace de la mémoire physique ; dans ce but, elle stocke des informations sur la fréquence d'utilisation des pages, ces informations étant utilisées par l'algorithme de remplacement des pages implémenté dans le système d'exploitation ;
- pour l'algorithme d'ordonnancement des processus ; en effet, la table des pages contient l'ensemble des adresses virtuelles utilisées par le processus ; elle permet donc de savoir si une page est présente ou pas en mémoire physique ; par conséquent, lorsqu'un processus fait appel à une page qui n'est pas présente en mémoire physique et qui doit être chargée depuis le disque, l'algorithme d'ordonnancement doit bloquer le processus et le mettre en attente, c'est-à-dire faire un changement de contexte, afin de ne pas bloquer le processeur pendant le chargement de la page (plusieurs millisecondes) ; cette situation est appelée une faute de page.

Bibliographie

[1] Intel 4004, http://www.intel.com/intel/intelis/museum/exhibit/hist_micro/hof/hof_main.htm.

[2] Intel Pentium 4, http://www.intel.com/home/desktop/pentium4.

[3] Intel Itanium, http://www.intel.com/itanium.

[4] C.E. Shannon, *The synthesis of two-terminal switching circuits*, Transactions of the American Institute of Electrical Engineers, 28, 1, 59-98, 1949.

[5] S. B. Furber, D. A. Edwards and J. D. Garside, *AMULET3: a 100 MIPS Asynchronous Embedded Processor*, Proceedings of ICCD'00, Austin, Texas.

[6] Yale N. Patt, Sanjay J. Patel, *Introduction to Computing Systems, from bits and gates to C and beyond*, McGraw Hill International Editions, 2001.

[7] Jean-Michel Muller, *Arithmétique des ordinateurs. Opérateurs et fonctions élémentaires*, Masson, 1989.

[8] David A. Patterson et John L. Hennessy, *Organisation et Conception des Ordinateurs: L'interface Materiel/Logiciel*, chez Dunod.

[9] John L. Hennessy et David A. Patterson, *Architecture des Ordinateurs, une Approche Quantitative*, 2ème édition, chez Morgan Kaufmann.

[10] A. S. Tanenbaum, *Modern Operating Systems*, 2ème édition, chez Prentice Hall