Système d'exploitation

I. Organisation, Processus et Interruptions

Kévin Perrot

Aix Marseille Université

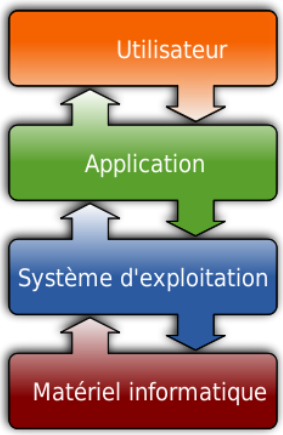
2017

Ce cours est (entre autres) basé sur les supports de Jean-Luc Massat en L3 informatique à Luminy.

Table des matières

1 Introduction

1.1 Qu'est-ce qu'un OS ?



GOLFTHEMAN

FIGURE 1 – Le système d'exploitation (*Operating System* OS) est un programme (lui-même composé de sous-programmes) qui gère l'utilisation des ressources d'un ordinateur. Il orchestre l'exécution de logiciels applicatifs (ceux que l'utilisateur emploie), en leur allouant des ressources (capacité de stockage des mémoires et des disques durs, capacité de calcul du processeur) de façon à ce que leur utilisation n'interfère pas avec d'autres demandes provenant d'autres logiciels.

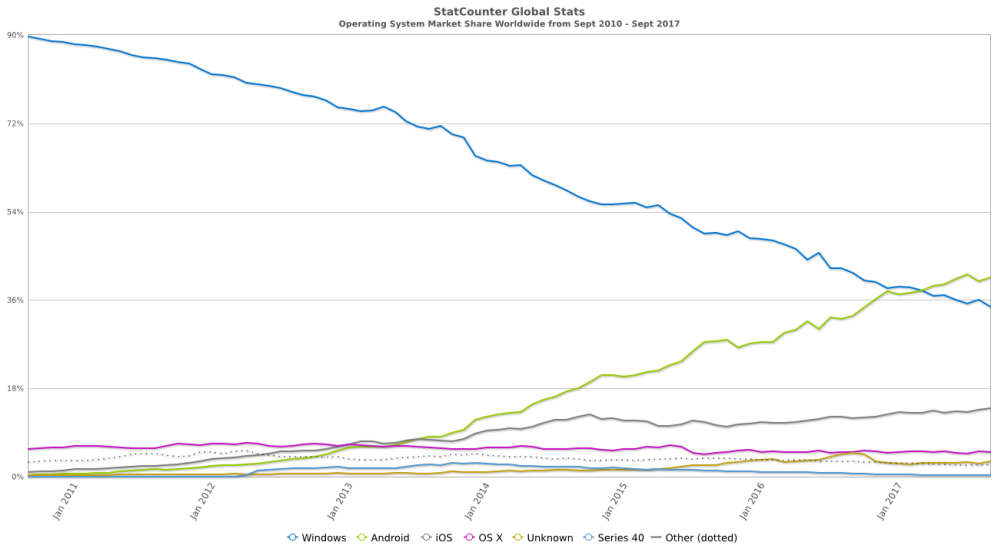


FIGURE 2 – Statistiques d'utilisation des systèmes d'exploitation.

1.2 Que se passe-t-il quand mon ordinateur démarre ?

Instruction : une opération élémentaire.

CPU : le processeur (*Central Processing Unit*) est le composant (hardware) qui calcule (qui exécute les instructions machines).

x86 : est un jeu d'instruction pour CPU, basé sur le CPU Intel 8086 de 1978.

ROM : mémoire morte (*Read-Only Memory*), est une mémoire informatique non volatile (qui ne s'efface pas quand on coupe le courant). La ROM est lente.

RAM : mémoire vive (*Random Access Memory*), est une mémoire informatique volatile utilisée pour stocker les données en cours de traitement. La RAM est rapide.

Séquence d'amorce (*Boot sequence*) des CPU x86.

- Exécution de l'instruction située à l'adresse `FFFFFFF0h`, généralement un `jump` qui pointe sur la première instruction du BIOS.
- BIOS (*Basic Input Output System*), situé sur la ROM de la carte-mère (fourni avec elle par le fabriquant).
  - *Power Good* : test du courant fourni par le transformateur.
  - POST (*Power-On Self-Test*) : vérifications (code du BIOS, mémoire, intégrité de la carte-mère, "*press F2 to enter setup*", initialisation de la RAM et des périphériques, affichage des configurations à l'écran).
  - Chargement de l'OS :
    - lit les disques dans l'ordre configuré (*setup*).
    - pour chacun, regarde s'il est amorçable (*bootable*), c'est-à-dire s'il contient un OS (indiqué par la présence de la *MBR boot signature* où il faut).
    - dès qu'il trouve un disque amorçable, il lui donne la main.
- Démarrage de l'OS :
  - Si n'y a qu'un seul OS, le lance.
  - Sinon, lance le chargeur d'amorçage (*boot loader*) (GRUB, LILO, rEFIt, Boot Camp, etc) qui propose à l'utilisateur de choisir l'OS qu'il souhaite lancer, parmi ceux rencontrés sur le disque.

2 Organisation d'un OS

2.1 Fonctionalités d'un OS

D'un point de vue de l'utilisateur, un OS doit en résumé assurer les fonctions suivantes.

- Gestion et conservation de l'information.
  Pour que tout utilisateur puisse créer, conserver, retrouver ou détruire les *objets* sur lesquels celui-ci désire effectuer des opérations.
- Gestion de l'exploitation de programmes.
  Le système d'exploitation est un logiciel de base, qui gère l'exécution d'autres logiciel appelé logiciels d'application (traitement de textes, compilateurs, jeux, etc).

2.1.1 Fonctions d'un OS

Pour bien effectuer cette gestion, l'OS a de nombreuses fonctions.

— Structuration de l'information.
Toute les informations sont stockées sous forme de *fichiers*.

— Transfert des données.
Entre l'unité centrale, le clavier, l'écran, l'imprimante, etc.

— Gestion de l'ensemble des ressources.
Pour que tout utilisateur ait accès aux ressources.

— Gestion du partage des ressources.
Le système doit répartir les ressources dont il dispose entre les divers usagers en respectant la règle d'*équité* qui empêche la *famine*. En particulier, il doit réaliser un ordonnancement des travaux qui lui sont soumis et éviter les *interblocages*.

— Abstraction matériel.
Offrir un service indépendant de l'implémentation matérielle : l'utilisation est identique, même si le matériel est différent.

2.1.2 Diversité d'utilisation et modularité

Les systèmes d'exploitation peuvent servir à de nombreuses tâches, et reposer sur des matériels aux contraintes différentes.

— Surveillance médicale : robustesse.

— Bases de données : efficacité accès à la mémoire.

— Séquençage de l'ADN : rapidité de calcul.

— Jeux vidéos : réactivité.

— Téléphone : faible consommation d'énergie.

— etc.

Malgré cette grande diversité, les systèmes comportent entre eux des parties très ressemblantes, voire identiques, et il serait donc très utile de pouvoir dégager celles-ci afin de profiter de certaines études partielles dans l'élaboration de portions plus complexes. C'est ce souci de rentabilisation qui a conduit à une conception modulaire des systèmes et de leurs différents constituants, techniques généralisable à tout logiciel développé sur une machine quelconque.

2.2 Structure interne des OS

2.2.1 Structure en couches

Un système informatique est composé de couches d'abstraction (*abstraction layers*), qui sont empilées, chaque couche superposée apportant de nouvelles fonctions de plus en plus élaborées et reposant sur les fonctions plus élémentaires assurées par les couches sous-jacentes.

Chaque couche est construite sur la couche précédente, et peut être utilisée sans connaître les couches inférieures : elle fait abstraction des détails qui la composent.

Chaque couche offre des services à la couche supérieure, et est cliente de la couche inférieure. Deux couches communiquent au travers d'interfaces bien définies.

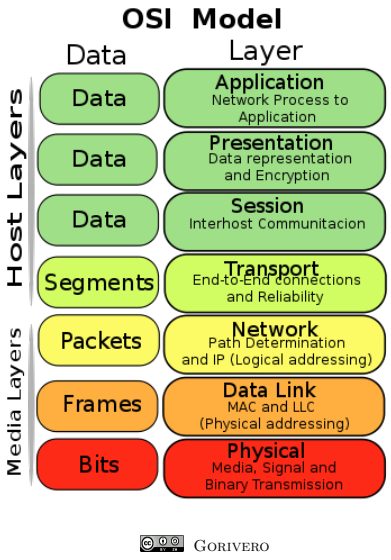


FIGURE 3 – Exemple de structure en couches : le modèle OSI (*Open Systems Interconnection*, utilisé pour le réseau).

Chaque couche est seule responsable de son fonctionnement interne. Les éventuelles modifications de ce fonctionnement ne doivent pas influencer les autres couches.

Autre exemple : le système de fichier. Vous disposez de primitives pour ouvrir, lire, écrire et fermer un fichier. On peut donc manipuler des fichiers sans devoir connaître le mécanisme interne des disques. Le système de fichier est pour vous une couche logicielle dont vous n'avez besoin de connaître que l'interface : comment s'en servir, elle vous permet de faire abstraction des détails propres à la mise en oeuvre des fichiers.

La structure en couche offre deux avantages dont il serait difficile de se passer.

— Indépendance pour la conception. L'utilisation d'une couche est complètement décrite par son interface, pas besoin d'en savoir plus.

— Indépendance pour la modification. On peut modifier une couche sans altérer les autres couches : il suffit de continuer à respecter l'interface. Ainsi, on peut améliorer ou corriger chaque couche sans avoir besoin de réécrire toutes les couches qui en dépendent.

Chaque couche peut être pensée comme un logiciel : un logiciel de niveau supérieur va utiliser les fonctionnalités d'un logiciel de niveau inférieur selon la description de son utilisation donnée par son interface.

2.2.2 Interfaces de programmation

Une interface de programmation (*Application programming interface* API) est accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur (la spécification). Mais l'interface, en elle-même, désigne un ensemble normalisé de *classes*, de *méthodes* ou de *fonctions* qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels.

Ces concepts sont ceux de la programmation orientée objet !

Exemple d'interface : Facebook `https://developers.facebook.com` !

2.2.3 Normes

Pour que les logiciels (à tous les niveaux) profitent à un maximum d'utilisateurs, de nombreuses normes et standards ont été établis.

— POSIX (*Portable Operating System Interface*)

— TCP/IP (*Transmission Control Protocol* et *Internet Protocol*)

— GNU C Library (`glibc`)

— OpenGL (*Open Graphics Library*)

— etc.

2.2.4 Interface et gestion des erreurs

Dans la spécification d'une interface, il y a deux façon de prendre en compte les éventuelles erreurs.

— Une variable d'erreur. Chaque procédure comporte un paramètre supplémentaire (code d'erreur) qu'elle peut modifier. La valeur finale de cette variable constitue un compte-rendu interprétable à un niveau supérieur.

— Interruptions (déroutements). A chaque cause d'erreur est associée une procédure de traitement spécifique. En cas d'erreur, un mécanisme que nous détaillerons ultérieurement (appelé interruption ou déroutement) déclenche automatiquement la procédure correspondante.

Dans tous les cas, le traitement d'une erreur consiste à revenir à un état stable du système où l'exécution puisse reprendre normalement, en perdant le moins d'information possible. La seconde méthode nécessite un mécanisme supplémentaire, mais elle sera préférable à la première pour deux raisons essentielles.

— Sécurité. Le caractère systématique du déroutement en cas d'erreur est supérieur au test de code qui peut être omis, provoquant ainsi une propagation d'erreur.

— Clarté. Le fait de pouvoir associer une procédure particulière à chaque cause d'erreur permet de séparer clairement le traitement des situations *normales* de celui des situations *exceptionnelles*.

Les méthodes de conception que nous venons de présenter et les concepts ou outils d'abstraction qui leur sont associés permettent, on l'a vu, grâce à la modularité ainsi acquise, de diviser et subdiviser un système informatique en plusieurs parties indépendamment modifiables ou même interchangeables. Cet aspect s'avère primordial, car les systèmes élaborés à l'heure actuelle sont de plus en plus importants et de plus en plus complexes. Si bien que leur réalisation est confiée à plusieurs personnes, à divers services, voire à plusieurs équipes. La cohérence du tout ne peut être facilement obtenue qu'à la condition d'avoir préalablement clairement défini les spécifications d'interface et l'arborescence des classes d'objets manipulés... mais seulement cela.

D'autre part, certaines portions du système peuvent être conçues de différentes façons en utilisant différentes stratégies. Dans ces conditions, les concepteurs devront tester celles-ci sur des critères de rapidité, d'optimisation d'occupation, d'utilisation de ressources, etc. Le respect des contraintes d'interfaçage autorisera la mise au point et l'évaluation de performance de ces parties par simple substitution sans affecter le reste du système.

A propos de ce cas de figure, les contraintes s'avèrent très strictes, la substitution d'un module par un autre module n'étant possible qu'à la condition que les autres modules n'y accèdent qu'en utilisant son interface. En particulier, un programme appelant un module ne devra en aucune façon exploiter des renseignements sur la réalisation interne du module. Une méthode efficace pour parvenir à ce but a été proposée par Parnas : laisser les programmeurs d'un module dans l'ignorance de la réalisation des autres.

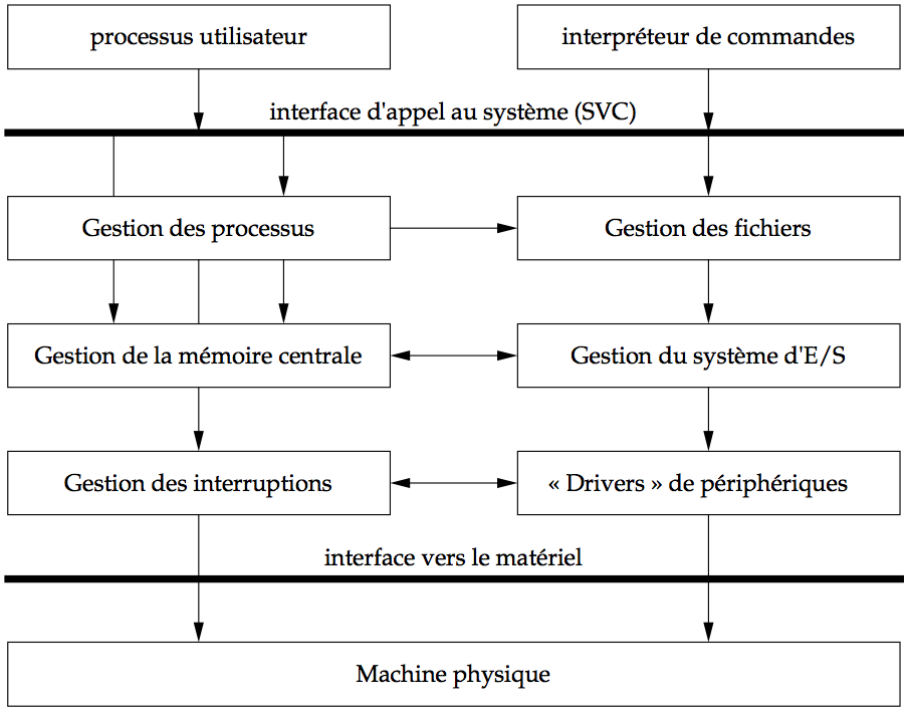
2.2.5 Composants d'un OS

L'étude des composantes permet de fixer les rôles de chaque couche logicielle et les rapports entre ces couches. Nous allons distinguer les modules suivants (Figure 4).

- Le *gestionnaire d'interruptions* récupère les interruptions matérielles et logicielles et applique le traitement approprié qui varie suivant la cause de ces interruptions.
- Les *pilotes de périphériques* (*drivers*) gèrent l'échange des données avec les périphériques. Chaque pilote connaît son périphérique et cache son mode d'utilisation aux couches supérieures du système. Ces drivers utilisent les interruptions car le dialogue asynchrone entre CPU et unités externes s'effectue au moyen des interruptions. En d'autres termes, le pilote envoie des ordres à son périphérique qui répond au bout d'un temps non défini par le biais d'une interruption.
- Le *système d'E/S* masque les drivers de périphériques. Il offre des fonctions d'E/S qui, bien qu'étant de bas niveau, ne distinguent pas de manière explicite la nature du périphérique. Ces E/S sont réalisées à partir (ou vers) des zones de la mémoire appelées des tampons (buffer). L'allocation de ces tampons passe donc par le gestionnaire de la mémoire centrale.
- La *gestion de la mémoire centrale* répond aux demandes d'allocation et de libération de zones mémoire. Dans une première approche, la mémoire virtuelle peut être vue comme une extension de la mémoire centrale qui est temporairement rangée sur disque. Ce déplacement d'une partie de la mémoire implique :
  - le retour à la demande des informations utiles et non présentes en mémoire centrale (c'est une opération d'E/S) ;
  - la sauvegarde sur disque des informations présentes mais inutilisées.
- Le *système de gestion des fichiers* (SGF) offre toutes les primitives nécessaires à la création, destruction, modification des fichiers se trouvant en mémoire secondaire.
- La *gestion des processus* répartit la ou les CPU entre les tâches qui en ont besoin. Ces tâches consomment de la mémoire et exploitent des fichiers.
- Les *processus utilisateur* (dont les *interpréteurs de commande* sont un exemple particulier) utilisent l'OS en lui adressant des requêtes en bonne et due forme. Ces requêtes permettent, au choix :
  - de lancer, de figer ou de tuer d'autres processus,
  - d'exploiter ou de modifier des fichiers,
  - d'allouer de la mémoire, etc.

2.3 Historique des OS

Cette section est issue du cours de Jean-Luc Massat (2012).



source : cours de Jean-Luc Massat (2012).

FIGURE 4 – Les composants et la structure d'un système.

L'historique est un moyen agréable de présenter les principaux concepts en partant de l'absence d'OS pour arriver aux systèmes répartis.

2.3.1 Systèmes monoprogrammés

Sur les premiers ordinateurs il n'existe pas d'OS à proprement parler. L'exploitation de la machine est confiée à tour de rôle aux utilisateurs ; chacun disposant d'une période de temps fixe. C'est une organisation en porte ouverte.

Au début des années 50, on voit apparaître le premier programme dont le but est de gérer la machine : c'est le *moniteur d'enchaînement des tâches*. Cet embryon d'OS a la charge d'enchaîner l'exécution des programmes pour améliorer l'utilisation de l'unité centrale (UC). Il assure également des fonctions de protection (vis à vis du programme en cours d'exécution), de limitation de durée et de supervision des entrées/sorties. Pour réaliser ces opérations, le moniteur est toujours présent en mémoire. Il est dit résident.

La fin des années 50 marque le début du *traitement par lots* (*batch processing*). Une machine prépare les données en entrée (lecture des cartes perforées à cette époque) tandis que la machine principale effectue le travail et qu'une troisième produit le résultat. Il existe donc un parallélisme des tâches entre lecture, exécution et impression. Les opérations d'E/S ne sont plus réalisées par la CPU, ce qui libère du temps de calcul.

2.3.2 Systèmes multiprogrammés

La multiprogrammation arrive au début des années 60. Elle est caractérisée par la présence simultanée en mémoire de plusieurs programmes sans compter l'OS lui-même. Cette caractéristique s'explique de la manière suivante : l'exécution d'un programme peut être vue comme une suite d'étapes de calcul (les cycles d'UC) et d'étapes d'E/S

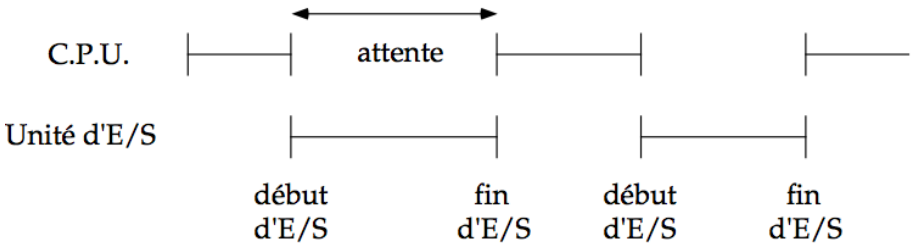


FIGURE 5 – Cycles de CPU et cycles d'E/S.

(les cycles d'E/S). Sur un système monoprogrammé, la CPU est donc inutilisée durant les cycles d'E/S.

L'idée de base est d'utiliser ces temps d'attente pour exécuter un autre programme (Figure 5). Ce programme doit nécessairement être déjà présent en mémoire afin d'éviter l'E/S de chargement puisque justement on cherche à utiliser les temps morts d'E/S. La réalisation pratique de cette idée nécessite :

— des unités matérielles capables d'effectuer des E/S de manière autonome (libérant ainsi la CPU pour d'autres tâches) ;

— des possibilités matérielles liées à la protection de la mémoire et/ou à la réimplantation du code pour éviter qu'une erreur d'un programme influence le déroulement d'un autre.

Dans les années 60/70 les premiers *systèmes en temps partagé* (*time sharing*) sont disponibles. Ces systèmes sont directement liés à l'utilisation interactive des machines au moyen de terminaux vidéo. Ce mode d'utilisation impose un temps de réponse *acceptable* puisque les utilisateurs attendent devant leur terminaux. Pour garantir un bon temps de réponse moyen, le temps d'exécution de la CPU est découpé en tranches appelées des *quantas*. Ces quantas sont allouées aux programmes en cours d'activité. Le temps d'exécution de la CPU est donc partagé entre les programmes utilisateurs. Si le nombre d'utilisateurs n'est pas trop important et, sachant qu'un utilisateur moyen passe 90% de son temps à réfléchir et seulement 10% à exécuter une action, le temps de réponse reste acceptable et chaque utilisateur à l'impression d'avoir sa propre machine. Sur un plan matériel, le temps partagé est basé sur les possibilités suivantes :

— les programmes sont tous en mémoire ; le temps partagé implique donc la multiprogrammation ;

— le matériel doit permettre l'interruption d'un programme au bout de son quanta de temps pour passer la CPU à autre programme ;

— les temps de commutation d'un programme vers un autre doit être aussi faible que possible car durant cette étape la CPU est utilisée par l'OS au détriment des programmes utilisateurs.

Les systèmes répartis se développent durant les années 80 (Figure 6). Dans cette organisation, les données mais aussi les programmes sont réparties sur plusieurs machines connectées par un réseau. Les problèmes sont plus complexes puisqu'ils couvrent la communication, la synchronisation et la collaboration entre ces machines, mais ceci est une autre histoire... et un autre cours.

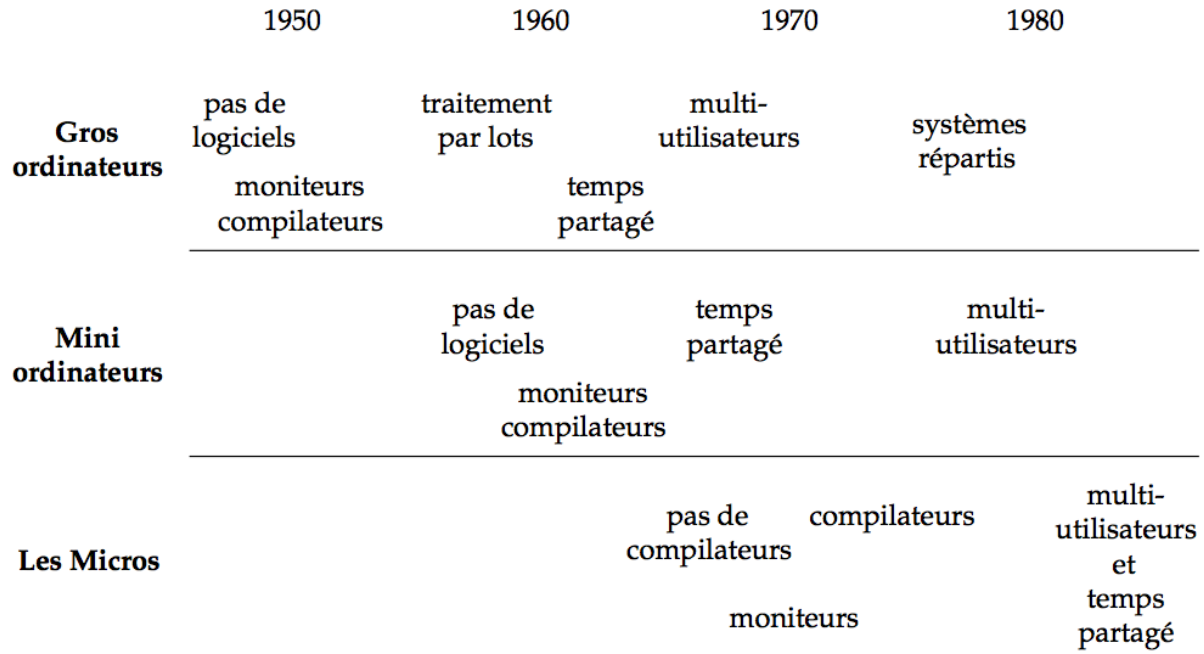
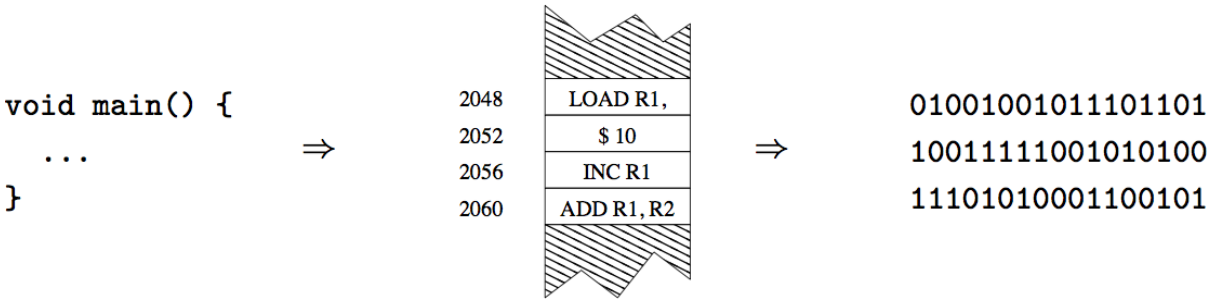


FIGURE 6 – Évolution des systèmes d'exploitation (Siberschatz et Galvin, 1994).



source : cours de Jean-Luc Massat (2012).

FIGURE 7 – Un programme.

3 Quelques notions

Un programme est une suite d'instructions rangées en mémoire (Figure 7).
Une instruction est une opération élémentaire, qui peut être

— ordinaire (mouvement de données, calcul, appel du système d'exploitation...) ;

— privilégiée (lancement des E/S, contrôle des interruptions, contrôle du matériel...).

Une machine est composée des éléments suivants.

— Processeur (CPU)

— Mémoire

— Organes d'E/S

L'état d'une machine est l'état de ses composants.

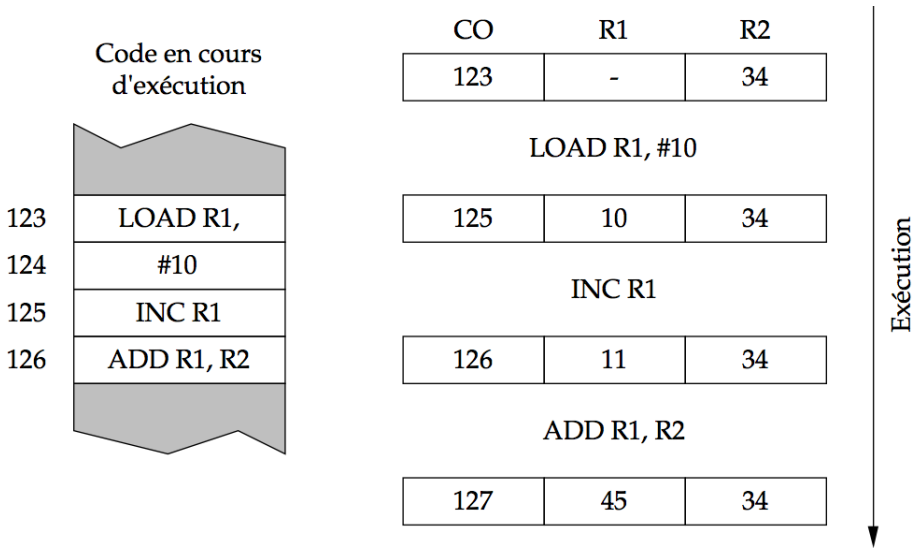
Un processeur est composé des éléments suivants.

— Unité de calcul (*Arithmetic and Logic Unit* ALU).

— Registres :

- Généraux (entier, flottant, adresse).
- Spécialisés (Mot d'État du Processeur MEP ou *Processor Status Word* PSW) :
  - Compteur ordinal (*Program Counter* PC). Contient l'adresse mémoire de l'instruction en cours d'exécution ou de la suivante.
  - Registre d'instruction (*Instruction Register* IR). Contient l'instruction en cours de traitement.
  - Registre d'état (*FLAGS register*). Ensemble de bits représentant des drapeaux.
  - Pointeurs de pile (*Stack Pointer* SP). Contient l'adresse du sommet de la pile.
  - Mode d'exécution (MODE). En mode *esclave* les instructions privilégiées sont interdites, et en mode *maître* les restrictions disparaissent.
  - Masque d'interruptions (*Interrupt Mask* IM). Indique le niveau de priorité du processus courant, qui est utilisé pour savoir si une interruption doit être prise en compte immédiatement, ou alors être retardée (masquée).
- Unité de contrôle, horloge, unité d'entrée-sortie.
- Bus. Ce sont des systèmes de communication partagés par plusieurs composants.

Une exécution est une évolution discrète d'un programme (Figure 8).



source : cours de Jean-Luc Massat (2012).

FIGURE 8 – Une exécution.

Un processus est un programme en cours d'exécution. Il est constitué des éléments suivants.

- Le code machine associé au programme.
- Un espace d'adressage en mémoire vive (pour le code machine, la pile, les données...).
- Des attributs de sécurité (propriétaire, permissions).
- Des ressources allouées.

Le contexte d'un processus comprend l'état de ses constituants, ainsi que l'état du processeur (le MEP/PSW).

4 Interruptions

4.1 Mécanisme

Le problème est le suivant. Pendant que votre ordinateur effectue des calculs, il doit en même temps répondre à de nombreux *stimuli* :

— un paquet arrive sur la carte réseau

— déplacement de la souris

— activation du clavier

— apparition d'une alarme

Ces évènements arrivent de façon asynchrone.

L'attente active n'est pas une bonne solution car

— le test explicite est pénible et peu sécurisé

— perte de temps CPU

Les interruptions permettent d'interrompre provisoirement le déroulement d'un programme en cours pour exécuter une routine considérée comme prioritaire. On associe à chaque cause d'interruption un numéro k qui l'identifie. On dispose également dans les adresses basses de la mémoire d'une table appelée le vecteur d'interruptions (`vi`). Les cases `vi`$[k]$ de cette table contiennent l'adresse de la routine à exécuter lors d'une interruption de cause k. Cette routine est appelée le traitant d'interruption de cause k.

Plus précisément, lors d'une interruption de cause k, la CPU effectue dès la fin de l'instruction en cours les actions suivantes :

1. sauvegarder la valeur du compteur ordinal et le mode d'exécution (dans une pile ou dans une case mémoire particulière suivant les C.P.U.) ;
2. passer en mode maître ;
3. placer dans le compteur ordinal la valeur `vi`$[k]$, c'est à dire l'adresse de la première instruction de la routine associée à l'interruption de cause k.

L'interruption est donc un mécanisme matériel puisque la sauvegarde et l'initialisation du compteur ordinal à partir du vecteur d'interruptions sont des opérations réalisées par la CPU. Le traitant représente la partie logicielle du mécanisme d'interruption. Il a (presque) toujours la structure suivante :

1. sauvegarder la valeur des registres de la CPU (dans un emplacement particulier de la mémoire). Cette étape est couramment appelée la sauvegarde du contexte ;
2. traiter la cause de l'interruption ;
3. restaurer la valeur des registres de la CPU et le mode du programme interrompu. C'est la restauration du contexte ;
4. placer dans le compteur ordinal la valeur préalablement sauvegardée.

De cette description on tire deux conclusions :

— les traitants d'interruption s'exécutent en mode maître (donc avec des droits étendus)

— l'exécution du programme interrompu n'est pas perturbée par le traitement de l'interruption.

L'étape 4 est souvent réalisée au moyen d'une instruction de la CPU qui provoque le retour au programme interrompu (RTI). Cette étape est appelée l'*acquittement de l'interruption*. Les principales utilisations du processus d'interruption sont les suivantes :

— Interruption logicielle (ou déroutement) provoquée par la CPU lors de la détection d'une situation *anormale*. Par exemple :
  — appel explicite de l'OS,
  — instruction incorrecte ou inconnue,
  — violation de privilège,
  — dépassement de capacité,
  — division par zéro,
  — tentative d'accès à une zone protégée de la mémoire.
— Interruption matérielle générée par une unité externe à la CPU afin de lui signaler l'apparition d'un événement extérieur. Par exemple :
  — fin d'une E/S,
  — impulsion d'horloge,
  — changement d'état d'un des périphériques,
  — présence d'une valeur intéressante sur un capteur.

Certaines CPU n'ont qu'une seule cause d'interruption. Dans ce cas, un *ou* logique de toutes les causes possibles sera effectué et le traitant d'interruption – qui est unique – devra au préalable tester les indicateurs pour connaître la cause.

Le mécanisme des interruptions est illustré sur la Figure 9.

4.2 Hiérarchie

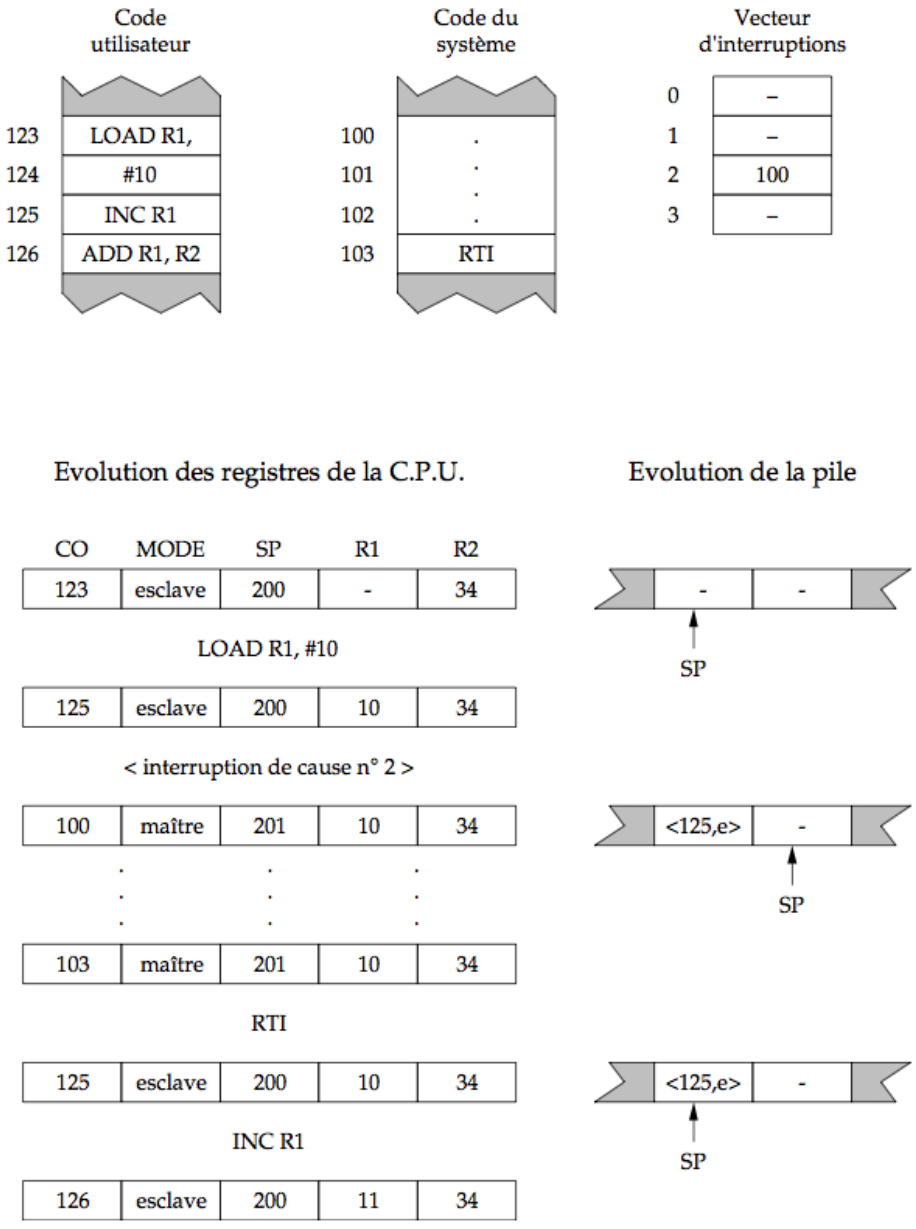
Les interruptions peuvent être hiérarchisées, c'est-à-dire classées par ordre de priorités respectives. Un traitant d'interruption peut donc être lui-même interrompu par une demande d'interruption de niveau de priorité supérieure. Il passe alors à l'état d'attente. La figure suivante représente l'activité des programmes dans le temps pour un système hiérarchisé à 8 niveaux où le niveau 0 est le plus prioritaire, le niveau 7 correspondant au programme d'arrière-plan (Figure 10).

Aussi, certaines opérations doivent être atomiques (elles ne doivent pas être interrompues), comme par exemple la sauvegarde ou la restauration du contexte doit toujours se faire entièrement (sinon l'on risque de perdre certaines informations).

4.3 Types d'interruptions

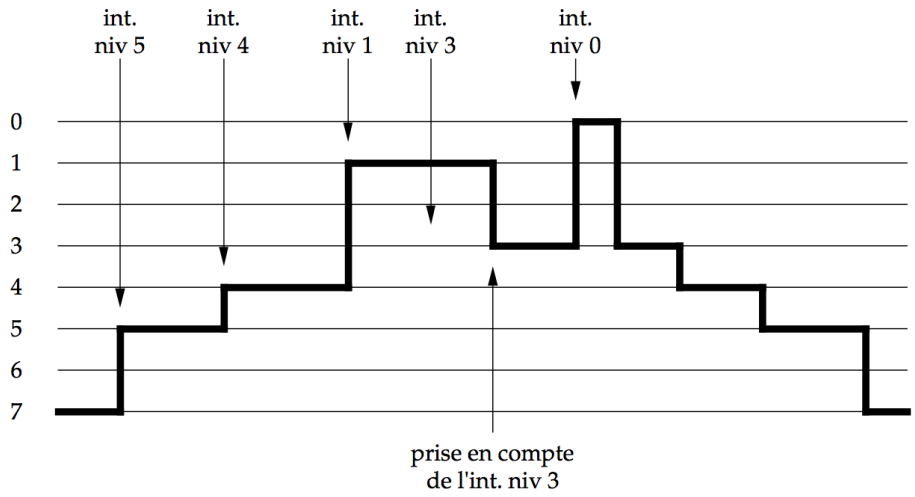
On distingue trois types d'interruptions.

— Interruption matérielle : réaction aux évènements extérieurs.
— Appel système (appel au superviseur, *SuperVisor Call* SVC, ou *Trap*) : appel explicite d'une routine système.



source : cours de Jean-Luc Massat (2012).

FIGURE 9 – Exemple d'interruption d'un processus.



source : cours de Jean-Luc Massat (2012).

FIGURE 10 – Hiérarchie des interruptions.

côté utilisateur

- préparer les arguments de la requête
- préparer le type de la requête
- SVC
- analyser le compte rendu de l'OS

côté OS

- sauver le contexte du demandeur
- vérifier la nature de la requête
- vérifier les arguments de la requête
- vérifier les droits du demandeur
- exécuter la demande
- restaurer le contexte du demandeur
- retour vers le demandeur

— Déroutement : traitement des erreurs et des situations anormales.

Les programmes utilisateur sont exécutés en mode esclave. Ainsi, toutes les actions demandant des droits étendus (en particulier ce qui concerne les E/S) passent par une requête en bonne et due forme à l'OS. Les appels système permettent l'utilisation depuis un programme utilisateur d'un certain nombre de routines système exigeant des droits étendus. Les appels système provoquent des interruptions, ce qui parait compliqué mais a un avantage de sécurité important : il existe un et un seul point d'entrée vers l'OS pour tous les processus utilisateur. Il est donc plus facile (pour le concepteur du système) de sécuriser l'appel des primitives système.

5 Processus

Cette section est issue du cours de Jean-Luc Massat (2012).

5.1 Definition

Un processus est un programme en cours d'exécution. Il faut bien faire la différence entre un programme qui est un fichier inerte regroupant des instructions de la CPU et un processus qui est un élément actif. Figeons un processus pour en observer ses composantes. Nous trouvons :

— des *données* (variables globales, pile et tas) stockées dans une zone de la mémoire qui a été allouée au processus ;

— la *valeur des registres* (généraux et spécialisés) de la CPU lors de l'exécution ;

— les *ressources* qui lui ont été allouées par le système d'exploitation (mémoire principale, fichiers ouverts, périphériques utilisés, etc.) ;

L'ensemble de ces composantes forme le contexte d'exécution d'un processus ou plus simplement le *contexte*.

5.2 État d'un processus

Un processus n'est pas continuellement en train de s'exécuter. Si la machine comporte n processeurs identiques, à un instant donné il y a au maximum n processus actifs. En fait, parmi tous les processus qui sont susceptibles de s'exécuter, seulement un petit nombre s'exécutent réellement. L'allocation de la CPU aux processus qui la réclament est appelée l'ordonnancement de la CPU.

L'état opérationnel d'un processus est un moyen de représenter les différentes étapes de ce processus telles qu'elles sont gérées par le système d'exploitation. La Figure 11 montre les divers états dans lesquels, dans une première approche intuitive, peut se trouver un processus :

— Initialement, un processus est connu du système mais l'exécution n'a pas débuté.

— Lorsqu'il est initialisé, il devient prêt à être exécuté (1).

— Lors de l'allocation de la CPU à ce processus il devient actif (2). Trois cas peuvent alors se présenter :

  — Le processus se termine (3).

  — Le processus est en attente (5) d'un événement et dès sa réception il redeviendra prêt (6).

  — Le processus est suspendu et se remet dans l'état prêt (4). Il y a *réquisition* ou *préemption* de la CPU. Dans ce cas, l'OS enlève la CPU au processus qui la détient.

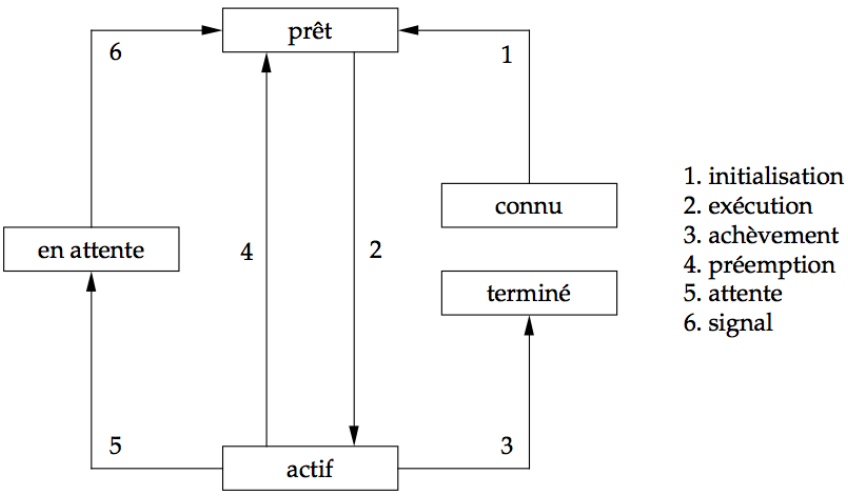


FIGURE 11 – États d'un processus.

La notion d'attente d'un événement mérite par son importance et sa complexité un petit exemple. Un éditeur de texte enchaîne continuellement la boucle suivante :

répéter
  lire un caractère;
  traiter ce caractère;
jusqu'à . . .;

Lorsque le processus éditeur est *actif* il adresse une requête à l'OS pour lui demander une opération d'E/S (la lecture d'un caractère). Deux cas se présentent :

— si il existe un caractère dans le tampon d'entrée, ce dernier est renvoyé par l'OS ;

— si le tampon d'entrée est vide, l'OS va *endormir* le processus en changeant son état. Lorsque l'utilisateur frappe une touche du clavier, l'OS (qui avait préalablement sauvegardé la demande de l'éditeur) *réveille* l'éditeur qui pourra ainsi devenir actif et traiter ce fameux caractère.

Plus généralement, toutes les opérations lentes (en comparaison de la vitesse de la CPU) provoquent un arrêt momentané du processus demandeur et une reprise ultérieure lorsque l'opération est terminée. C'est notamment le cas pour les opérations d'E/S. Le but de ce mécanisme est de récupérer le temps d'attente pour exécuter un autre processus sur la CPU.

5.3 Représentation d'un processus

Un processus est caractérisé dans le système par :

— un identificateur ou numéro (par exemple le PID pour *Process IDentification* dans le système UNIX) ;

— un état opérationnel (par exemple, un des cinq vus précédemment) ;

— un contexte ;

— des informations comme les priorités, la date de démarrage, la filiation ;

— des statistiques calculées par l'OS comme le temps d'exécution cumulé, le nombre d'opérations d'E/S, le nombre de défauts de page, etc.

Ces informations sont regroupées dans un bloc de contrôle de processus ou PCB (*Process Control Block*). Le système maintient donc un PCB pour chaque processus reconnu. Ce PCB est une mine d'informations. Il représente en fait la principale donnée manipulée par l'allocateur de la CPU.

Lorsqu'un processus quitte l'état *actif*, son PCB est mis à jour et la valeur des registres de la CPU y est sauvegardé. Pour que ce même processus redevienne actif, l'OS recharge les registres de la CPU à partir des valeurs sauvegardées dans le PCB, il change l'état et finalement il redémarre l'exécution du processus. Nous verrons plus tard dans quelles conditions un processus peut *perdre* la CPU.

Hormis les processus, le système maintient également un certain nombre de files qui regroupent les blocs de contrôle des processus. On trouve ainsi,

— la file des processus prêts,

— la file des processus connus,

— la file des processus en attente qui se décompose en

  — la file des processus qui attendent la disponibilité (de la ressource *unité d'E/S 0*, de la ressource *mémoire*, etc. . .)

— la file des processus qui attendent la fin d'une opération d'E/S (sur l'unité d'E/S 0 (le disque principal), sur l'unité d'E/S 1 (un terminal), etc...)

— etc...

La Figure 12 illustre le chemin suivi par notre éditeur de texte dans les files du système lors de sa demande d'un caractère sur l'unité d'entrée standard.

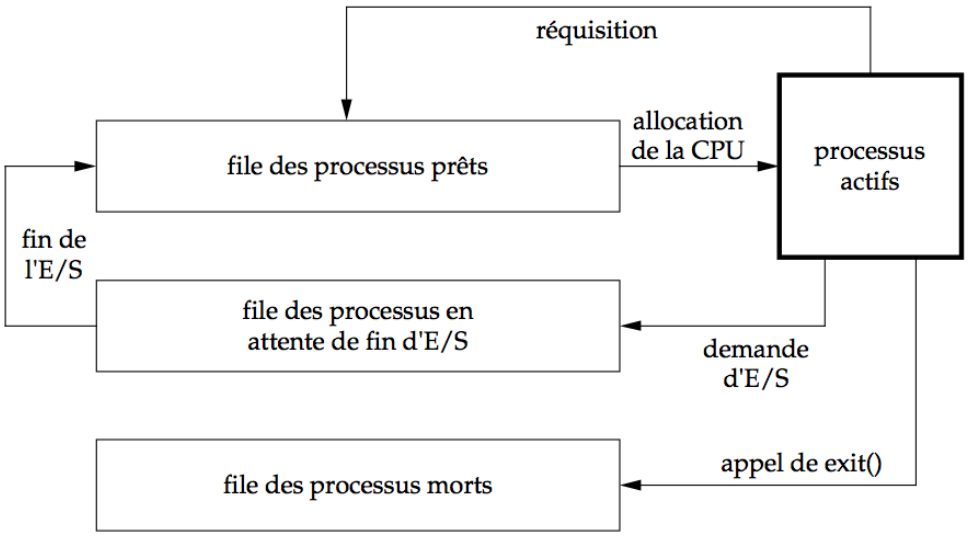


FIGURE 12 – Files du systèmes pour la gestion des processus.

5.4 Gestion des processus

Les processus sont les principaux éléments actifs du système. Dans ce cadre, il est logique que la création d'un nouveau processus soit demandée par un processus. Il existe donc une filiation entre processus père et le(s) processus fils. Lors du démarrage de la machine, l'OS lance un processus qui est orphelin puisqu'il n'a pas de père. Ce processus est souvent appelé init. Le premier rôle de ce processus est de lancer des fils qui auront chacun une fonction dans l'organisation générale de la machine. Par exemple,

— un processus pour gérer les E/S asynchrones avec les terminaux,

— un processus pour gérer les connexions au système avec demande et vérification d'un nom d'utilisateur et d'un mot de passe,

— un processus pour gérer l'allocation de la CPU aux processus !

— etc...

Ces processus font partie du système d'exploitation. Ils s'exécutent donc avec des droits étendus. Nous les appellerons les processus système ou démons (*daemons*) par opposition aux processus utilisateur. Le système d'exploitation est donc composé d'un noyau résident qui ne s'exécute que sur demande explicite (interruptions et déroutements) et d'un ensemble de processus système qui ont chacun une fonction précise à assurer. Ce découpage présente deux avantages :

— la partie résidente du système est réduite en taille ce qui permet d'éviter une trop grande consommation de mémoire par le système ;

— les processus systèmes ne sont pas forcément toujours prêts ou même toujours présents en mémoire ce qui permet – encore une fois – de réduire la mémoire et le temps CPU consommé par l'OS au détriment des processus utilisateur.

Si le système est organisé à base de plusieurs processus, des logiciels d'application peuvent également adopter cette structure. Si c'est le cas, il est nécessaire et même vital de fournir des outils permettant une communication et une synchronisation aisée entre les processus d'une même application. De plus, cette structuration à base de processus coopératifs est la seule capable d'utiliser facilement une structure matérielle multi-processeurs en associant un processus différent à chaque processeur.

Nous avons parlé de la création d'un processus mais sa disparition est une étape importante ! Elle à lieu sur demande d'un processus étranger (système ou père) ou sur sa propre demande sous la forme d'un suicide. Ce dernier cas correspond à l'appel de la fonction standard `exit()` du langage C.

5.5 Poids lourds et poids légers

Nous avons évoqué plus haut les avantages liés à la structuration des applications sous la forme de processus coopératifs. Mais cette structure comporte également des inconvénients :

— elle implique une communication massive entre les processus ce qui engendre un coût non négligeable de la part du système ;

— elle augmente le nombre de commutations de contexte (c-à-d la sauvegarde et la restauration du contexte d'un processus interrompu) provoquant de ce fait une perte de temps de CPU.

La notion de thread et de systèmes multi-threads vise à régler ce type de problème. Dans les systèmes multi-threads un processus est défini comme un ensemble de threads. Un thread (aussi appelé processus de poids léger ou *LightWeight Process* LWP) est un programme en cours d'exécution qui partage son code et ses données avec les autres threads d'un même processus. Bien entendu, les piles sont propres à chaque thread pour éviter que les appels de fonctions et les variables locales ne se mélangent (Figure 13).

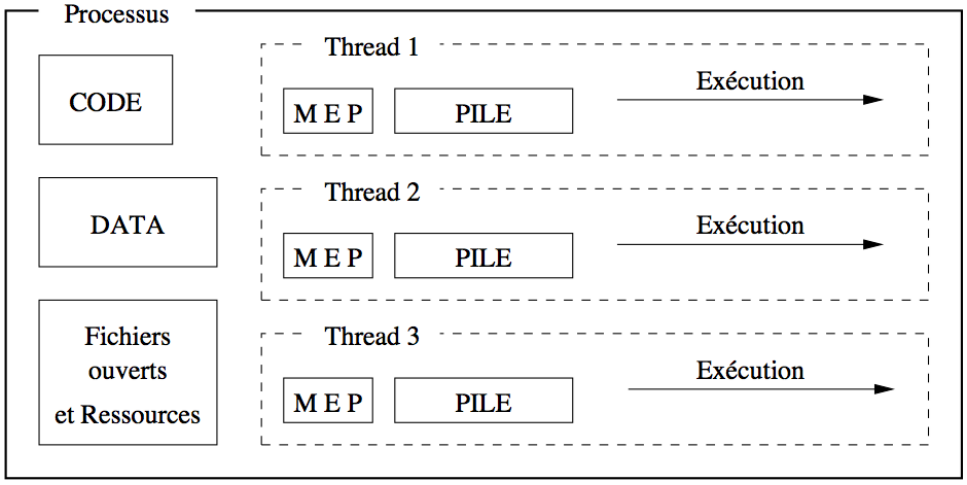


FIGURE 13 – Un processus est composé d'un ensemble de threads.

Cette solution présente plusieurs avantages :

— si un processus ne comporte qu'un seul thread nous revenons au modèle classique ; les systèmes multi-threads sont donc plus généraux ;

— il n'y a plus à mettre en place une communication entre les threads d'un même processus puisqu'ils agissent tous sur les mêmes données ;

— le temps de commutation entre les threads d'un même processus est réduit car le contexte est le même, et seuls les registres de la CPU doivent être sauvegardés ;

— en associant un (ou plusieurs) thread(s) à chaque processeur on peut facilement exploiter une structure multi-processeurs.