ReDS
Reconfigurable & embedded Digital Systems

heig-vd
HAUTE ÉCOLE D'INGÉNIERIE ET DE GESTION DU CANTON DE VAUD
www.heig-vd.ch

Systèmes d'exploitation (SYE)
Processus

Profs Daniel Rossier, Alberto Dassatti, Salvatore Valenza

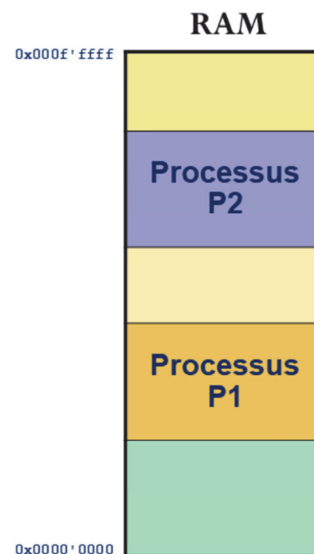
Version 3.3 (2017-2018)

Plan

- Définition d'un processus
- Création d'un processus
- Changements de contexte
- Etats et transitions

Définition d'un processus (1/7)

- ***Processus***
  - **Contexte d'exécution**
  - **Contexte mémoire**
- Monoprogrammation
  - Un seul processus en mémoire
- Multiprogrammation
  - Plusieurs processus en mémoire



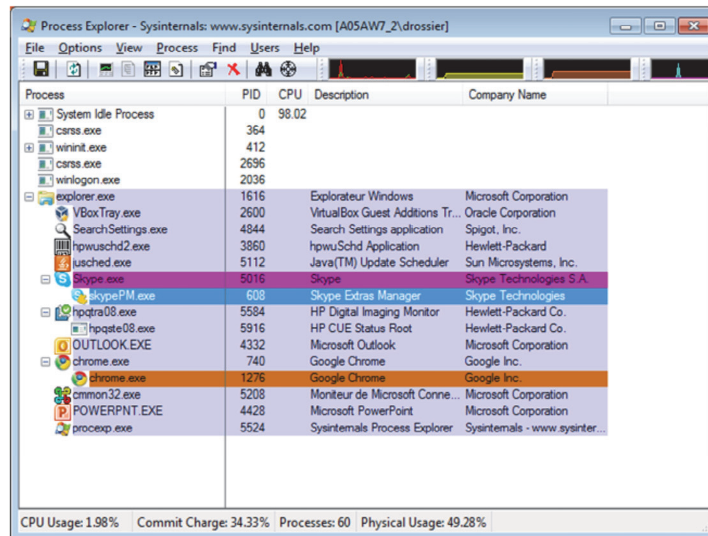
3 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

Dans une première approche, on peut dire qu'un **processus** représente une "application" qui tourne en mémoire. Il sera donc *chargé* en mémoire par le noyau et commencera son exécution; du point de vue d'un programmeur C par exemple, le point d'entrée du processus sera la **fonction *main()*** (bien qu'en réalité, ce n'est pas les instructions de cette fonction qui soient exécutées en tout premier. Il y en a d'autres auparavant qui sont *intégrées* à l'exécutable par l'éditeur de lien (*linker*). Dans ce cas de figure, le chargement d'un processus nécessite l'existence d'un fichier exécutable résidant sur disque dur (ou sur une autre mémoire de stockage) constituant ***l'image binaire*** du processus.

Un processus est donc associé à un **contexte d'exécution** et à **un contexte mémoire.** Le contexte d'exécution est "matérialisé" par les instructions qui s'exécutent, et comprend aussi les ressources logicielles manipulées par le processus, comme des objets de synchronisation, communication ou autres structures de données.

Dans un système multiprogrammé conventionnel, plusieurs processus peuvent être chargés en mémoire et tournés *à tour de rôle* sur le processeur, généralement durant un temps équivalent. Ce mécanisme de *partage du temps* du processeur donnera *l'illusion* à l'utilisateur que les processus tournent **en même temps**.

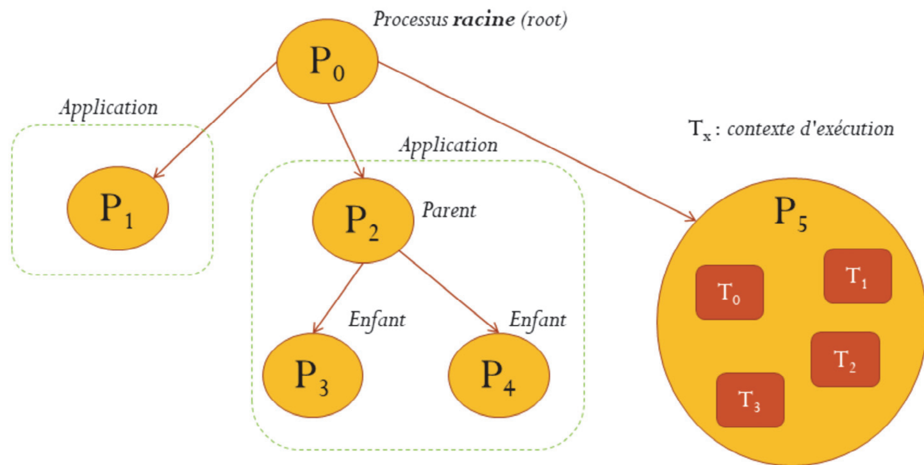
Définition d'un processus (2/7)



4 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

L'ensemble des processus tournant sur une machine peut être examiné avec des outils d'inspection (*Process Explorer, top, etc.*).

La figure ci-dessus montre la présence d'une affiliation entre les processus.



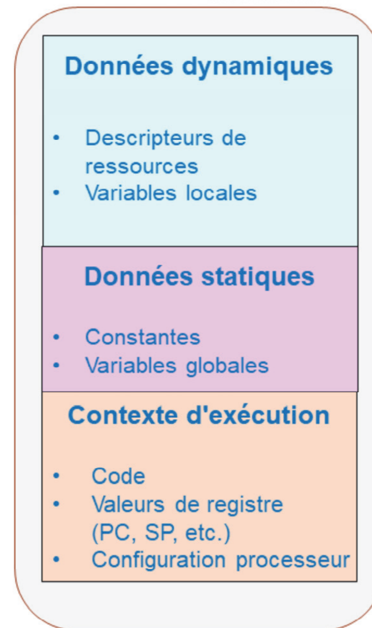
Dans un OS, les processus sont organisés de manière **hiérarchique**. Dès que le noyau a terminé la phase de *bootstrap* (démarrage noyau), ce dernier commence par lancer le tout premier processus du système. Généralement, il s'agit d'un processus particulier de type *init* ou *idle* auquel est rattaché d'autres processus (normalement un processus de type *shell*).
La notion même d'application n'a cependant aucun sens pour le noyau; une application peut être constituée d'un ou plusieurs processus, cela dépendra entièrement du développeur. Nous verrons par la suite que plusieurs contextes d'exécution (plus légers) peuvent également prendre place à l'intérieur d'un processus.

La création d'un nouveau processus s'effectue ainsi toujours à partir d'un processus existant, dans une relation parent-enfant. La terminaison d'un processus nécessitera **d'informer son parent**, et ce dernier pourra ainsi **récupérer l'état du fils** (en cours de terminaison) et de **restituer complètement les dernières ressources allouées** par le fils.

La terminaison prématurée d'un processus parent alors que celui-ci possède encore des enfants entraîne l'apparition de **processus *orphelins*** (*orphan*). Le noyau doit alors "rattacher" ces processus orphelins à un autre processus de niveau supérieur afin que celui-ci dispose à nouveau d'un parent. Ce nouveau parent adoptif peut être le processus racine (*root*). C'est opération est nécessaire, car lorsque le processus orphelin se termine, le processus parent doit libérer les dernières ressources comme mentionné précédemment. De cette manière, l'arborescence des processus reste *consistante.*

Définition d'un processus (4/7)

- **Caractéristiques**
  - **Image binaire**
    - Un fichier exécutable pouvant être chargé en mémoire.
  - **Contexte d'exécution**
    - Code (instructions machine)
    - Valeurs de registres, caches processeur
  - **Contexte mémoire**
    - Données (variables globales, locales, constantes)
  - **Ressources logiques**
    - Descripteurs de fichiers
    - Descripteurs d'objets de communication (IPCs)
    - …



6 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

Un processus est caractérisé par différents éléments dont son code, ses données, ses ressources logicielles/matérielles qu'ils utilisent durant son exécution, un ou plusieurs contextes d'exécution, etc.

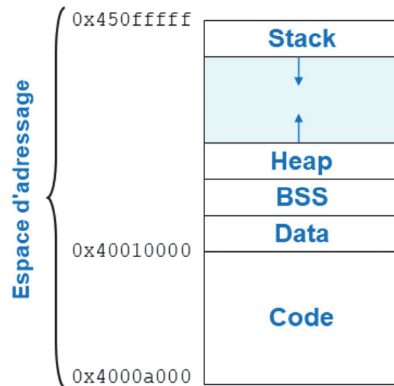
Un processus est chargé en mémoire à partir de son **image binaire**, c-à-d le fichier binaire (exécutable) résultant de l'édition des liens, après compilation. Ce fichier binaire est donc *structuré* d'une certaine manière de telle sorte que l'appel système effectuant la lecture du fichier et son chargement en mémoire puisse se faire correctement. Typiquement, cette structure de fichier correspond à un format standardisé: sous Windows, il peut s'agir de **COFF** (*Common Object File Format*), **ECOFF** (*Extended COFF*), ou de nos jours **PE** *(Portable Executable*). Sous Linux, il s'agit du format **ELF** (*Executable and Linkable Format*), successeur du format "**a.out**". Finalement, le format **Mach-O** est celui utilisé sous MacOS X.

Un fichier binaire (qui forme ainsi l'image binaire d'un processus) contient généralement une entête (*header*) suivi d'une liste de segments décrivant le processus; on trouvera des segments de **code**, de **données initialisées**, de **données non-initialisées**, d'**informations de *debug***, etc. Dans ce contexte la terminologie de *segment* est analogue à celle de *section.*

Le format de l'exécutable permet également de spécifier le type d'adresse utilisé (adresses relatives, absolues, symboliques).

Définition d'un processus (5/7)

- Sections (zones mémoire) d'un processus



- *Stack*
  - Variables locales d'une fonction
  - Arguments d'une fonction
  - Adresse de retour
  - Local au contexte d'exécution
- *Heap*
  - Allocation dynamique (*malloc, new, etc.*)
  - Global au(x) contexte(s) d'exécution
- *BSS*
  - *Block Started by Symbol*
  - Variables statiques **non**-initialisées
- *Data*
  - Variables statiques initialisées
  - Constantes

7 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

La notion de processus est fortement liée à celle d'espace d'adressage. En effet, chaque processus dispose de son propre espace d'adressage. L'organisation d'un processus en mémoire dépendra de l'OS et de l'ensemble des librairies systèmes: cette organisation consiste à placer les différentes zones mémoire (aussi appelées ***sections***) d'un processus, comme par exemple la pile, le tas, le code, etc.

La topographie d'un processus, c-à-d la configuration du processus en mémoire en terme de régions ou encore de **sections**, reste toujours la même: la section code (aussi appelé section ***text***) occupe les premières adresses, suivit de la section ***data*,** puis la section ***bss*** et la section ***heap***. Quant à la pile (*stack*), elle est généralement placée au sommet de l'espace d'adressage, et évolue de manière décroissante (on parle alors de pile descendante). Ainsi le tas évolue dynamiquement suivant des adresses croissantes, et va à l'encontre de la pile.

La section ***bss*** permet de stocker dans le fichier exécutable des variables non-initialisées, sans toutefois réserver déjà leur taille définitive. En effet, si l'on déclare un tableau de 1000 entiers, par exemple, il n'est pas utile de réserver 1000 x 4 octets dans le fichier. L'allocation de cette place mémoire s'effectuera donc durant le chargement de l'image binaire en mémoire.

Définition d'un processus (6/7)

- **PCB** (*Process Control Block*)
  - **Fiche signalétique** d'un processus
  - Structure de données gérée et visible **que par le noyau**
  - Contient toutes les informations système liées au processus
  - Utilisé lors de **changements de contexte**
- **Chaque processus** dispose d'un PCB **propre** et **unique.**

8 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

Chaque processus dispose d'une structure de données particulière, appelée ***PCB*** *(Process Control Block).* Le PCB est initialisé à chaque création d'un nouveau processus et contient toutes les informations relatives à ses ressources. C'est une structure gérée exclusivement par le noyau dont l'espace utilisateur n'a aucun accès.

Tout processus dispose de son propre PCB, et un PCB ne peut être associé qu'à un seul processus.

Le PCB est particulièrement utile lorsqu'il y a un changement de contexte entre processus, c'est-à-dire que le noyau "décide" de suspendre le processus en cours d'exécution pour donner la main à un autre processus. L'état du processus sortant pourra ainsi être sauvegardé "dans" son PCB (le PCB est mis à jour), et l'état du processus entrant sera récupéré à partir de son PCB.

Définition d'un processus (7/7)

- **PCB** – Une structure de données…
  - **Etat** du processus
  - **Espace d'adressage**
  - **Pointeur** vers le tas (*heap*)
  - **Références** vers les processus fils
  - **Descripteurs** des fichiers ouverts
  - **Contexte(s) d'éxécution**
    - Etat du contexte d'exécution
    - Pointeur sur instruction courante (PC)
    - Registres de données
    - Pile

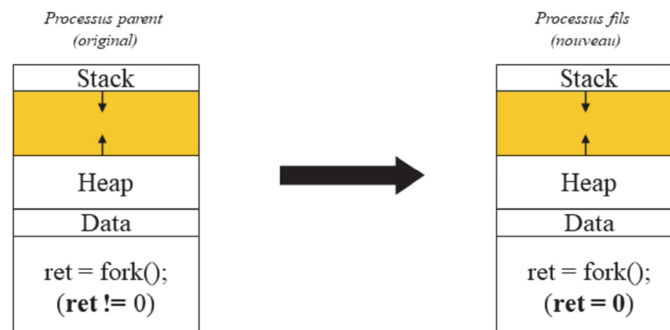
Etat	ID
PC	*Registres*
Priorité	
Pile (*Stack*)	
Tas (*Heap*)	
Fichiers ouverts	
Processus enfants	
Espace d'adressage	
Code de sortie	
…	

9 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

L'allocation du PCB s'effectue au tout début de la création d'un processus et est effectué dans le **contexte du processus parent**.

La terminaison d'un processus consiste à "informer" le processus parent; ce dernier peut récupérer un *code de sortie* – c'est l'argument de l'appel système *exit()* – afin de récupérer un état, puis le parent devra ***libérer*** la mémoire allouée pour son PCB (bien entendu, cette partie de code est pris en charge par du code noyau).

Nous verrons plus tard par quel mécanisme la communication entre fils et parent s'effectue.



L'appel système *fork()* permet de créer un nouveau processus. La création d'un processus consiste à **copier intégralement** les données du processus parent afin de pouvoir rapidement donner le contrôle au nouveau processus, de manière totalement indépendante du processus parent. Le processus fils peut ainsi poursuivre son exécution en *décidant* s'il doit remplacer son image binaire par une autre (dans le cas d'un lancement d'une nouvelle application par exemple), ou non. Dans ce dernier cas, l'image binaire étant la même, le code est identique à celui du parent: c'est pourquoi il est nécessaire de pouvoir *différencier* le contexte d'exécution dans lequel il s'exécute (contexte du parent ou contexte du fils), et cela dans le code même du processus. Le retour de l'appel système sert précisément à différencier ces deux contextes, comme nous allons l'expliquer.

Si la valeur retour de l'appel système *fork()* vaut **0**, cela signifie que "l'on se trouve dans le fils". Si la valeur est **supérieur à 0**, cela signifie que "l'on se trouve dans le parent", et cette valeur correspond à l'identification du fils (*PID: Process ID*). Cette valeur pourra être utilisée pour référencer le processus fils, notamment dans le cas d'une attente sur celui-ci (appel système *waitpid()*). Finalement, si la valeur est négative, cela signifie qu'il y a eu un problème à la création du processus fils (nombre maximum de processus atteint, dépassement de capacité mémoire, etc.). La raison exacte de l'échouement peut toujours être récupérée au travers de la variable globale *errno*.

Création de processus (2/4)

- Soit le code C suivant :

```
3   int main(int argc, char *argv[])
4   {
5     int pid0, pid1;
6
7     printf("message0\n");
8     pid0 = fork();
9
10    if (pid0)
11    {
12      printf("message1\n");
13      pid1 = fork();
14
15      if (pid1)
16        printf("message2\n");
17
18    }
19    else
20      printf("message3\n");
21  }
22
```

- Combien y a-t-il de processus au maximum ?
- Quels sont les affichages possibles ?
  (aidez-vous d'un diagramme de séquence)

Création de processus (3/4)

- Appel système ***waitpid()***

```
$ man 2 waitpid

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Attente sur un processus
- Possibilité de récupérer l'état du processus
- **Nettoyage du PCB** (restitution mémoire) lorsque le processus a terminé

12 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

L'appel système ***waitpid()*** joue un rôle fondamental au niveau de la gestion des processus, car non seulement il permet de se synchroniser avec un autre processus, mais il permet de récupérer l'état d'un processus terminé et de **restituer la mémoire allouée pour le PCB**.

Lorsqu'un processus termine son exécution, il doit exécuter l'appel système *exit()* (de manière explicite ou implicite) qui comporte un code de sortie (entier signé) dans son argument. Cet entier est stocké dans le PCB du processus et peut être récupéré par le processus parent via l'appel système *waitpid()*, grâce à l'argument ***wstatus*** dont l'adresse est passée comme second argument.

Cela explique pourquoi le PCB ne peut être éliminé avant que le processus parent n'est pris connaissance du code de sortie. Le processus terminé se trouve alors dans un état de *terminaison* (voir état des processus).

L'exécution de *waitpid()* s'avère donc obligatoire afin de procéder à l'élimination définitive du processus (suppression du PCB).

Création de processus (4/4)

- Appel système *exec()*
  - Remplacement de l'image binaire du processus
  - Démarrage à une certaine adresse (point d'entrée)
    - La fonction *main()* est appelée.
  - Passage d'arguments
  - Famille de fonctions de librairie (Posix), ***execve ()*** **est l'appel système**

```
$ man 3 exec

int execve(const char *filename, char *const argv[], char *const envp[]);
int execl(const char *path, const char *arg, ... /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ... /* (char  *) NULL */);
int execle(const char *path, const char *arg, ... /*, (char *) NULL, char * const envp[] */);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

```
$ man 2 execve

int execve(const char *filename, char *const argv[], char *const envp[]);
```

13 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

En réalité, l'appel système ***exec()*** en soi n'existe pas, mais repose sur une famille de fonctions appartenant à la librairie standard C (*Posix*). Au final, ces fonctions finiront toutes par appeler le *vrai* appel système qui est la fonction ***execve()*** dont les arguments sont indiqués ci-dessus.

Ces différentes fonctions permettent de passer les arguments relatifs à l'image binaire (chemin de répertoire, nom de fichier, etc.) ainsi qu'aux arguments qui seront récupérés dans la fonction ***main()*** de l'image chargée.

Par la suite, nous appellerons "*appel système exec()*" une des fonctions mentionnées ci-dessus.

Le lancement d'une application constituée d'un processus se déroulera alors en **deux temps**: la **création** d'un nouveau processus avec l'appel système *fork()* effectué par le parent, et le **chargement** (et démarrage) de l'image binaire avec l'appel système *exec()* **exécuté par le fils**.

Changements de contexte (1/3)

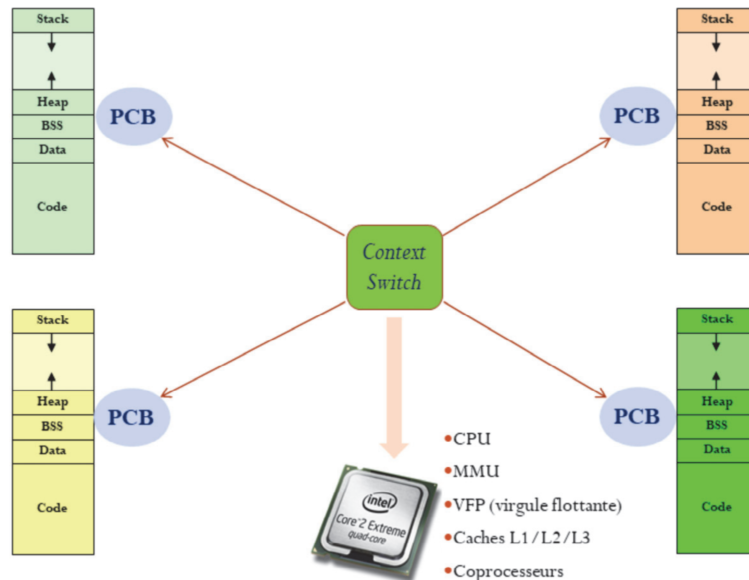
- Systèmes multiprogrammés
  - **Plusieurs processus** en mémoire

- Quand un changement de contexte peut-il survenir ?
  - **Terminaison** du processus (appel système *exit()*)
  - **Mise en attente** du processus
  - Sur décision de **l'ordonnanceur**

14 Cours SYE - Institut REDS/HEIG-VD - Gestion des processus

Comme nous l'avons évoqué à plusieurs reprises, un système multiprogrammé peut contenir plusieurs processus en mémoire, et des changements de contexte entre les processus peuvent intervenir à différents moments.

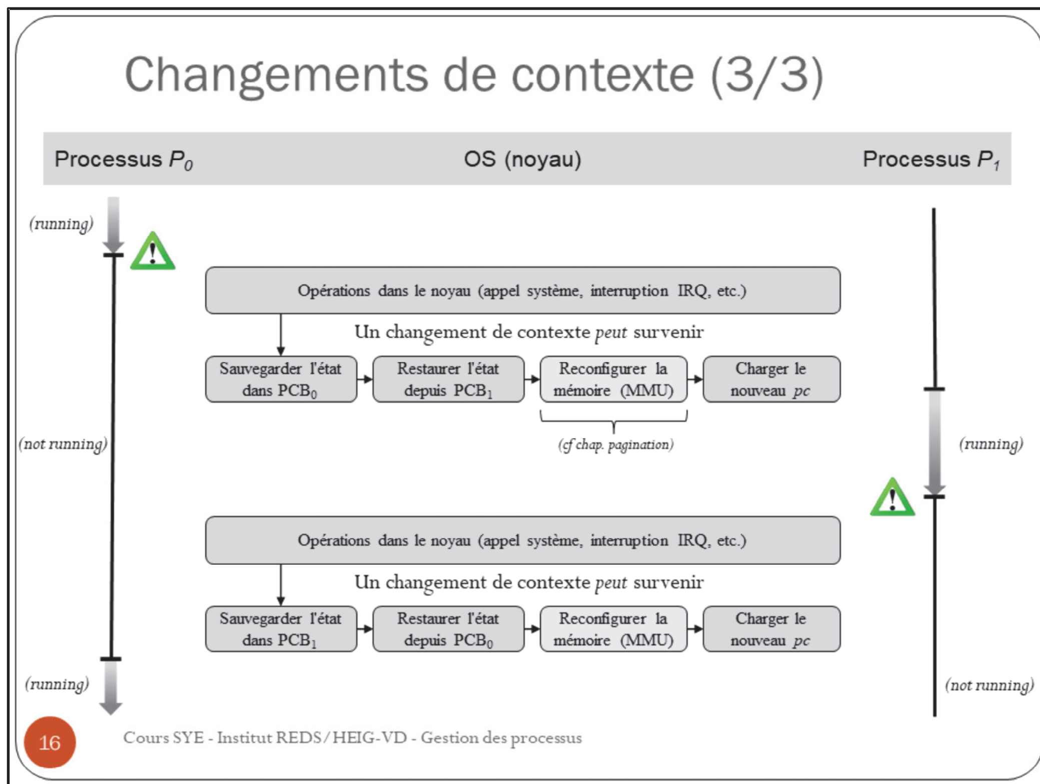
Un premier scénario conduisant obligatoirement à un changement de contexte est la **terminaison d'un processus**. L'exécution de l'appel système *exit()* permet de transmettre un code de sortie au processus parent, et d'informer le parent que le processus se termine. Si la fonction *exit()* n'est pas appelée explicitement, elle le sera de toute manière par le code s'exécutant à la sortie de la fonction *main()*; on se rappelle en effet que la fonction *main()* est appelée par du code *appondu* au moment du *linkage*, ce code appartenant généralement à un fichier objet nommé ***crt*** (*C runtime*).
Un autre scénario impliquant un changement de contexte est **lorsque le processus se met en attente**; il y a plusieurs raisons à cela. Par exemple, dans le cas d'une écriture, l'appel système *write()* peut faire intervenir le système de fichiers qui transmettra une requête au *driver* d'un disque dur, qui a son tour devra *attendre* sur la réponse du périphérique. Cette attente a pour effet de suspendre le processus. Par conséquent, l'ordonnanceur doit commuter sur un autre processus. **L'ordonnanceur** *décide* de changer de processus sur le CPU (intervention asynchrone du noyau).
Finalement, **l'ordonnanceur** peut en tout temps provoquer un changement de contexte, lorsque le temps alloué à un processus est consommé.



Un changement de contexte au niveau d'un processus est toujours une opération coûteuse (de l'ordre de la milliseconde) car elle fait intervenir beaucoup d'opérations sur le matériel. En particulier, le processeur doit être (re-)configuré afin que celui-ci se retrouve dans le contexte du processus *entrant*. Ses registres, les registres des coprocesseurs éventuels, la MMU (nécessaire pour la gestion mémoire), les mémoires *caches* et les structures internes du noyau doivent être modifiées en conséquence.

L'état actuel du processus *sortant* sera stocké dans son PCB; de même, l'état du processus *entrant* sera récupéré depuis son PCB.

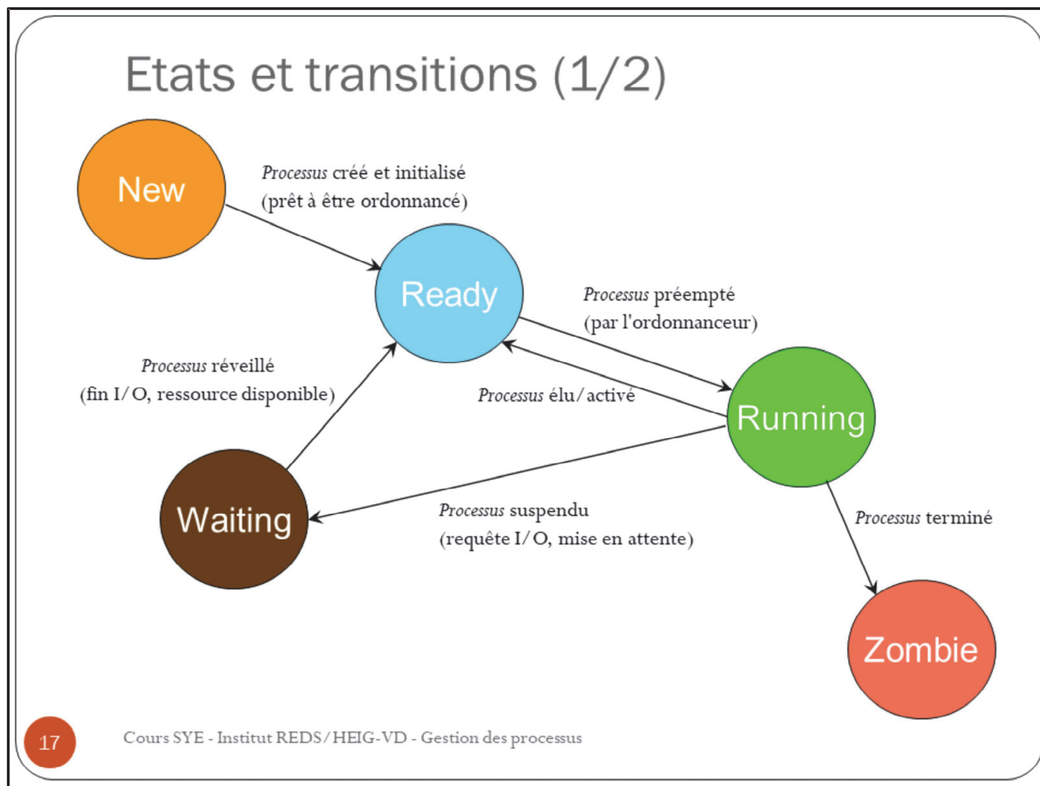
On verra par la suite que la politique de l'ordonnancement jouera un rôle prépondérant dans l'optimisation des performances en gardant le nombre de changement de contexte aussi faible que possible.



Lors d'un changement de contexte, le noyau et le matériel sont fortement sollicités car il s'agit de sauvegarder l'état **du contexte d'exécution et du contexte mémoire.** Typiquement, l'ensemble des registres du processeur contiennent des valeurs liées au contexte d'exécution; il s'agit de copier les valeurs de ces registres dans le PCB du processus sortant. De même, il s'agit de **préserver** les pointeurs courants de la pile, du tas, et des autres régions du contexte mémoire appartenant à ce processus.

Du point de vue du matériel, le contexte d'exécution pouvant solliciter également les coprocesseurs arithmétiques, graphiques et autres, il s'agit de préserver également l'état de ces unités. De plus, il faut également synchroniser les antémémoires (mémoires *caches*), c-à-d qu'il faut mettre à jour la RAM avec le contenu en *cache* avant de commuter sur un autre processus.

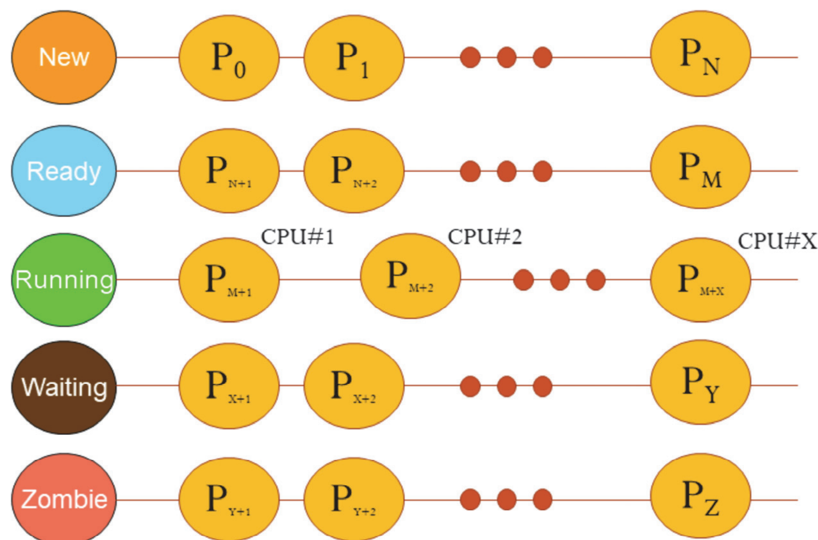
Finalement, comme nous le verrons dans le chapitre portant sur la gestion mémoire, l'unité *MMU (Memory Management Unit)* intervenant fortement dans les accès mémoire doit être **reconfigurée** pour permettre au processus entrant de pouvoir accéder son espace d'adressage correctement.



L'état ***new*** est utilisé lorsque le processus est en phase d'initialisation. Son PCB existe, mais toutes les structures de données ne sont pas encore initialisées. Le processus n'est donc pas encore *ordonnançable*.

Lorsqu'un processus se trouve dans l'état ***waiting***, et qu'il est en attente sur l'une ou l'autre ressource physique ou logique, il ne peut en **aucun cas** revenir directement sur l'état ***running***, une fois la ressource disponible. La transition par l'état ***ready*** permet de garantir que l'ordonnanceur reste maître du choix du processus à exécuter.

L'état ***zombie*** est utilisé lors de la terminaison d'un processus. Un processus dans cet état ne peut plus être ordonnancé et le processus parent peut consulter le code de sortie (généré par l'appel système *exit()*) avant de libérer définitivement le PCB (et donc de supprimer le processus du système).



En principe, il existe dans le noyau une file de processus par état (à l'exception de l'état *running* auquel peut être associé plusieurs processus que si la machine est multi-cœur). Les processus dans l'état ***ready*** sont des processus prêts à être ordonnancés. Les mouvements de cette file dépendra en particulier de l'ordonnanceur.

Lorsqu'un processus se trouve dans la file des processus dans l'état ***waiting***, il se peut que celui-ci reste bloqué longtemps. C'est pourquoi on peut associer aux processus de cette file un *watchdog* qui signalera à l'utilisateur une durée anormale d'attente (*timeout*). L'utilisateur pourra ainsi décider s'il faut *tuer* le processus ou non. Ce mécanisme permet d'offrir une solution très simple pour la détection de *deadlock* (interblocage) par exemple, ou l'indisponibilité définitive d'une ressource physique.

Références

- A. Silberschatz et al.:
  "**Operating Systems Concepts**", 8th edition, Wiley

- Andrew Tanenbaum
  "**Systèmes d'exploitation**", 3ème édition