Processus, synchronisation et communication

LIF12-Systèmes d'Exploitation

Fabien Rico

Univ. Claude Bernard Lyon 1

séance 2

Jacques Bonneville	jacques.bonneville@univ-lyon1.fr	TP
Adil Khalfa	adil.khalfa@cc.in2p3.fr	TD + TP
Yves Caniou	yves.caniou@univ-lyon1.fr	TP
Dorra Boughzala	dorra.boughzala@ens-lyon.fr	TP

Objectifs du cours

Bilan des UEs de programmation LIF1, 3, 5, 7, 9

- Programmation séquentielle : une seule chose à la fois,

Parmi les objectifs de LIF12

Programmation *concurrente*.

- Écrire un programme qui fait plusieurs choses en même temps ou plusieurs programmes qui interagissent,
- Organiser ces différentes *choses* pour arriver au bon résultat,
- Résoudre les problèmes d'accès concurrent, de synchronisation,
- Appréhender les échanges de données.

Problématiques

Système

- Commencer, exécuter, terminer un programme
- Passage d'une tâche à l'autre, déroutement
- Sécurité

Problématiques

Système

- Commencer, exécuter, terminer un programme
- Passage d'une tâche à l'autre, déroutement
- Sécurité

Programmation

- Utiliser le multi-tâche
- Faire communiquer des tâches entre elles

1. Processus
   - Processus
   - Commutation
   - État d'un processus
2. Programmation
   - Création
   - Hiérarchie de processus
3. Signaux
   - Définition
   - Programmation

Concepts

Exécution d'un programme

Les tâches du système sont de :

- trouver le fichier sur le disque, en fonction de l'organisation des fichiers ;
- trouver de la place en mémoire ;
- charger le programme en mémoire ;
- trouver le point d'entrée du programme (main()) ;
- exécuter la première instruction du programme ;
- continuer, et suivre le déroulement du programme (*compteur ordinal* ou pointeur d'instruction) ;
- à la fin du programme, libérer les ressources qu'il occupe ;
- obtenir et traiter son résultat.

Concepts

Exécution d'un programme

Les tâches du système sont de :

- trouver le fichier sur le disque, en fonction de l'organisation des fichiers ;
- trouver de la place en mémoire ;
- charger le programme en mémoire ;
- **trouver le point d'entrée du programme (main()) ;**
- **exécuter la première instruction du programme ;**
- **continuer, et suivre le déroulement du programme (compteur ordinal ou pointeur d'instruction) ;**
- **à la fin du programme, libérer les ressources qu'il occupe ;**
- **obtenir et traiter son résultat.**

Il faut une notion qui définit une exécution d'un programme : le Processus.

Processus

Définition (Processus)

Un processus est une instance (une exécution) d'un programme. C'est un ensemble de données gérées par le noyau qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, de le stopper et de le reprendre.

Processus

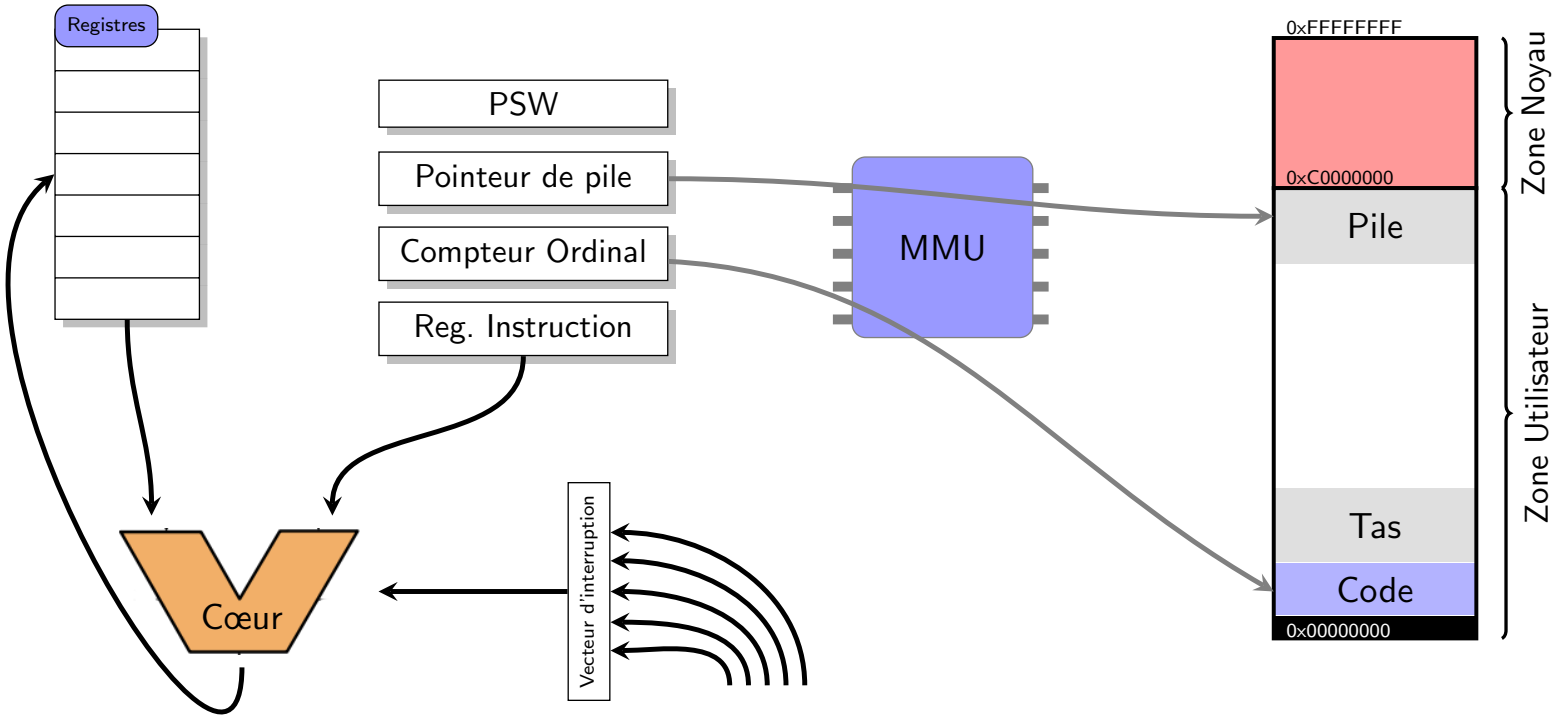
Définition (Processus)

Un processus est une instance (une exécution) d'un programme. C'est un ensemble de données gérées par le noyau qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, de le stopper et de le reprendre.

Son rôle est :

- d'identifier le programme ;
- de vérifier que le programme ne fait que ce qu'il a le droit de faire ;
- de lui permettre d'accéder aux fichiers et aux ressources ;
- d'isoler les programmes les uns des autres ;
- de libérer les ressources à la fin de l'exécution (fermer les fichiers, libérer toute la mémoire allouée, etc.) ;
- de permettre le passage d'une tâche à l'autre.

Processeur


Registres
PSW
Pointeur de pile
Compteur Ordinal
Reg. Instruction
MMU
Cœur
Vecteur d'interruption
0xFFFFFFFF
0xC0000000
Pile
Tas
Code
0x00000000
Zone Noyau
Zone Utilisateur

Constituant d'un processus

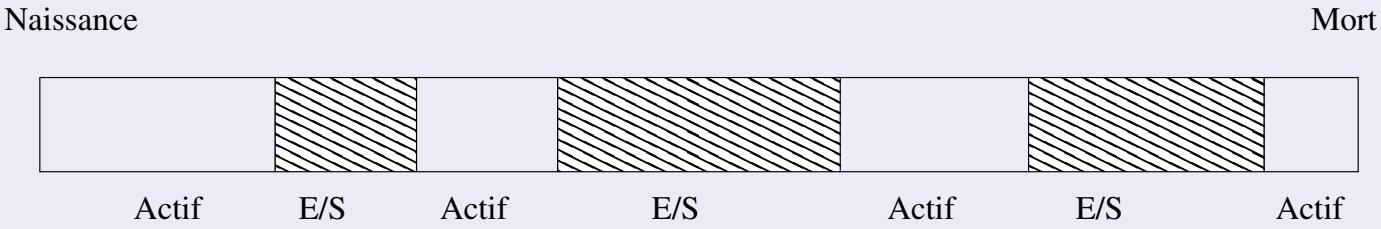
- Compteur ordinal
- Registres
- Pointeur de pile
- Mot d'état PSW
  mode d'utilisation, bits de condition, mode d'arrondi,. . .)
- Espace d'adressage
- État (en cours, prêt, bloqué...)
- Ressources (fichiers, sockets ...)
- Environnement (répertoire courant, id utilisateur. . .)
- Informations de gestion (id, pid, groupe, vecteur d'interruption,...)
- . . .

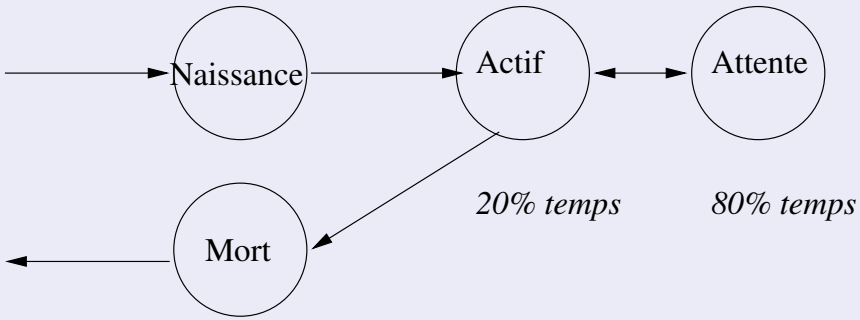
Mémoire d'un processus

Le processeur ne manipule jamais directement la mémoire. Pour chaque processus il utilise une mémoire virtuelle .

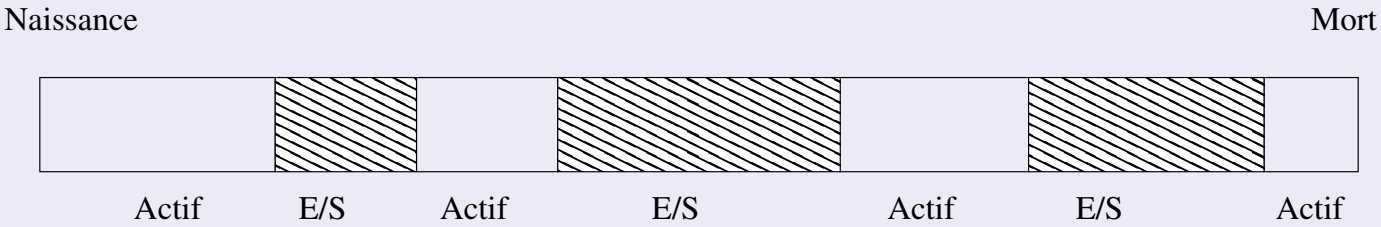
- Elle est de la taille maximum qu'il est possible d'adresser (4Go pour un 32bits)
- Elle est segmentée toujours de la même façon en fonction du système
- La mémoire est paginée, les adresses ne correspondent pas à la mémoire physique

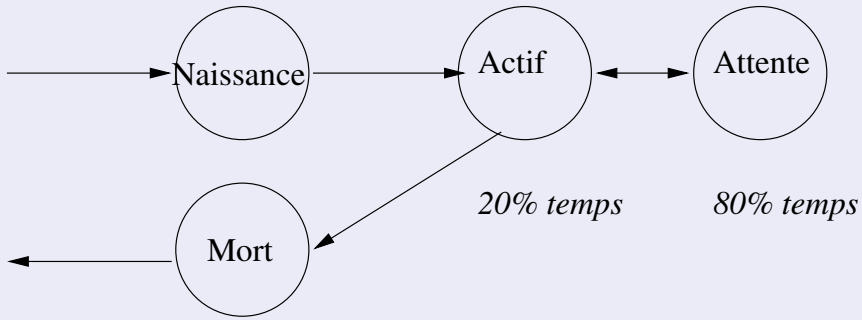
Vie d'un processus et multiprogrammation





Vie d'un processus et multiprogrammation





- les périphériques sont souvent plus lents que le processeur
- dans le cas de transfert par bloc (DMA,...), le processeur ne fait rien, si ce n'est attendre la fin du transfert...
- d'où l'idée simple : allouer le processeur à un autre processus !
- *c'est la commutation de processus*

État d'un processus

Rappel : Le système doit

- gérer l'accès au processeur
- gérer l'occupation de la mémoire

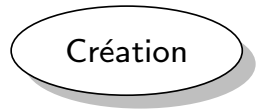
Définition (état d'un processus)

Le système doit donc gérer plusieurs listes de processus,

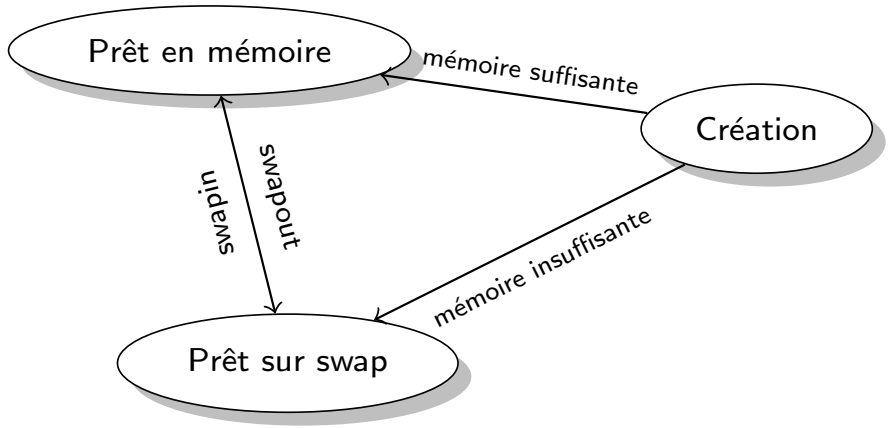
- ceux qui peuvent s'exécuter
- ceux qui sont en mémoire
- ceux qui sont bloqués car demandent une ressource occupée
- ...

On parle *d'état du processus*

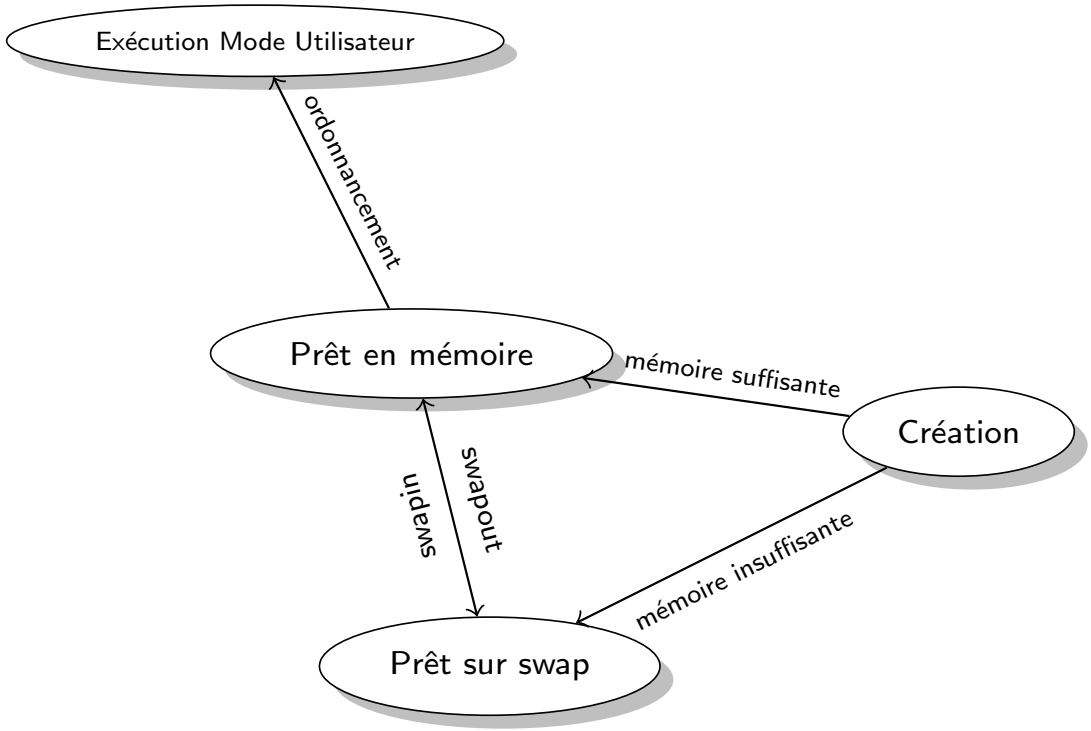
État d'un processus



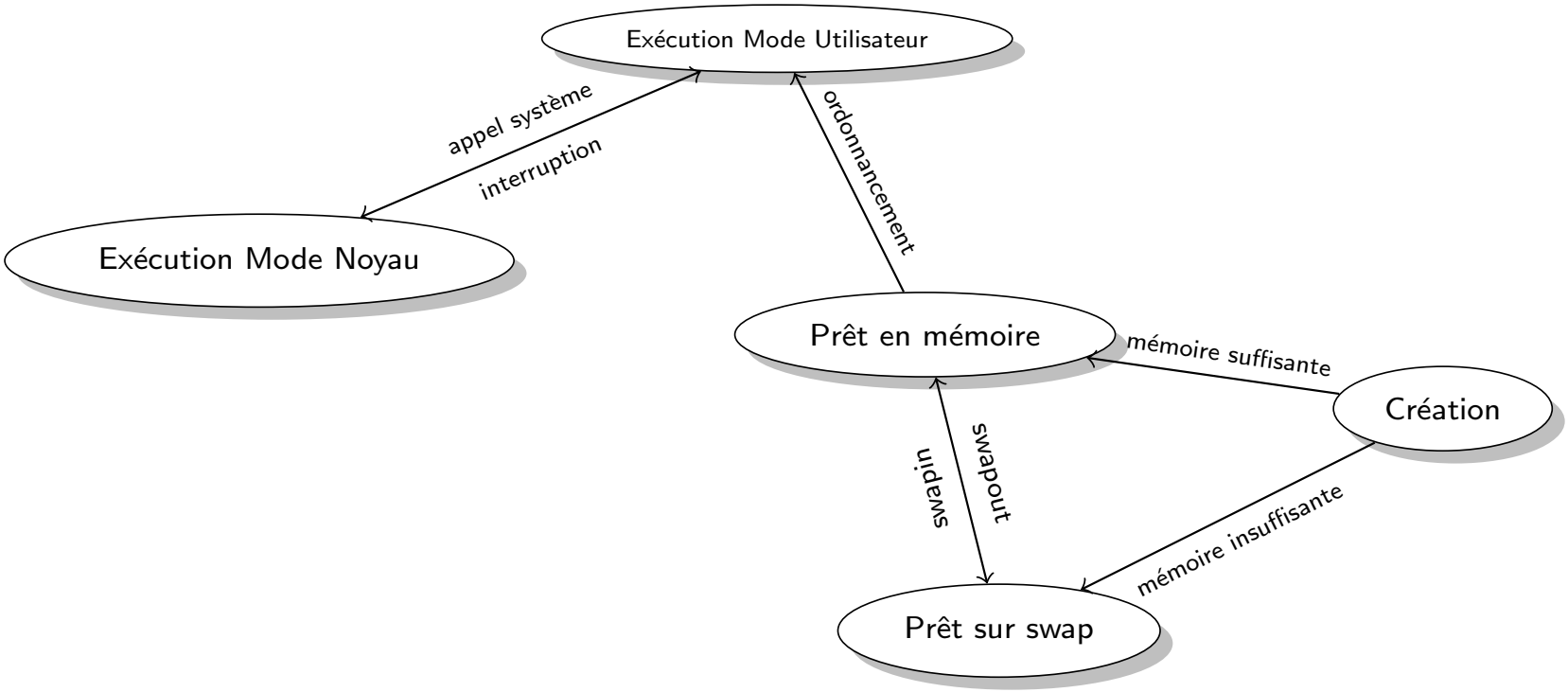
État d'un processus



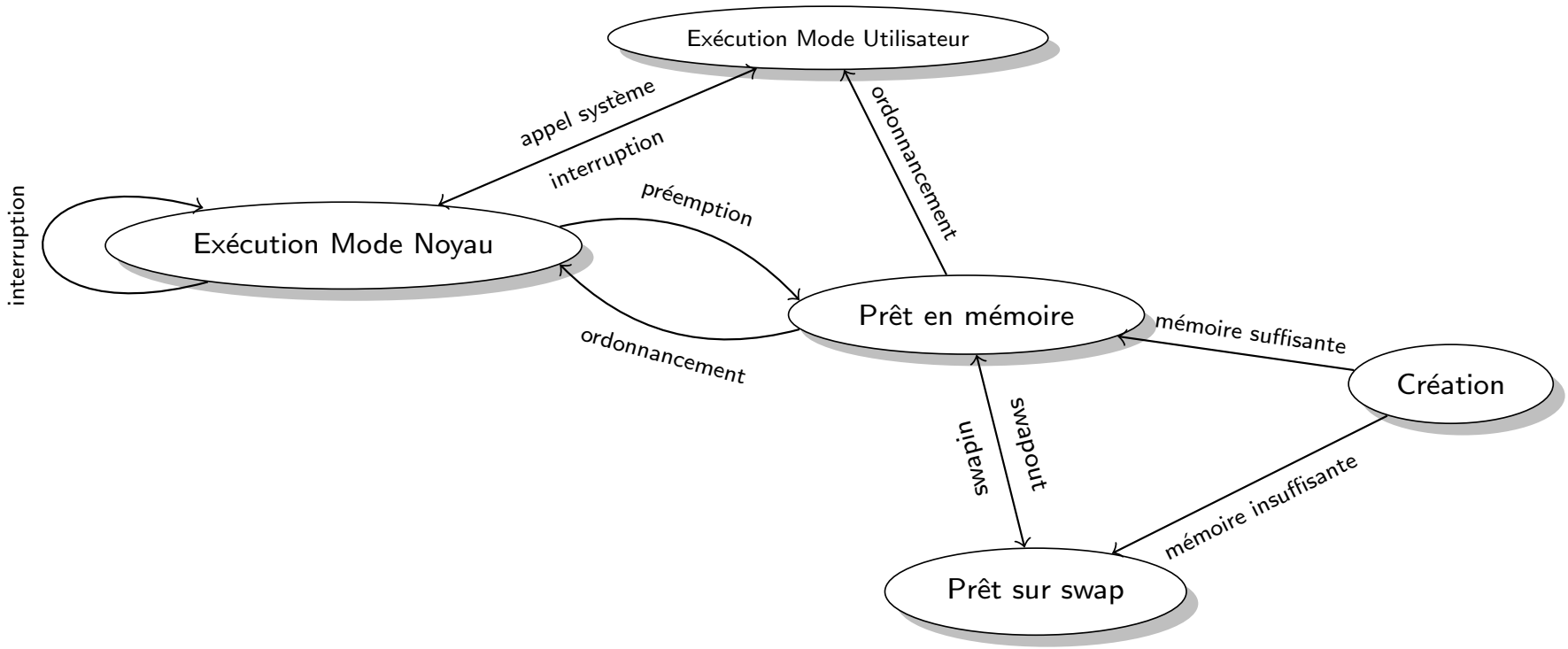
État d'un processus



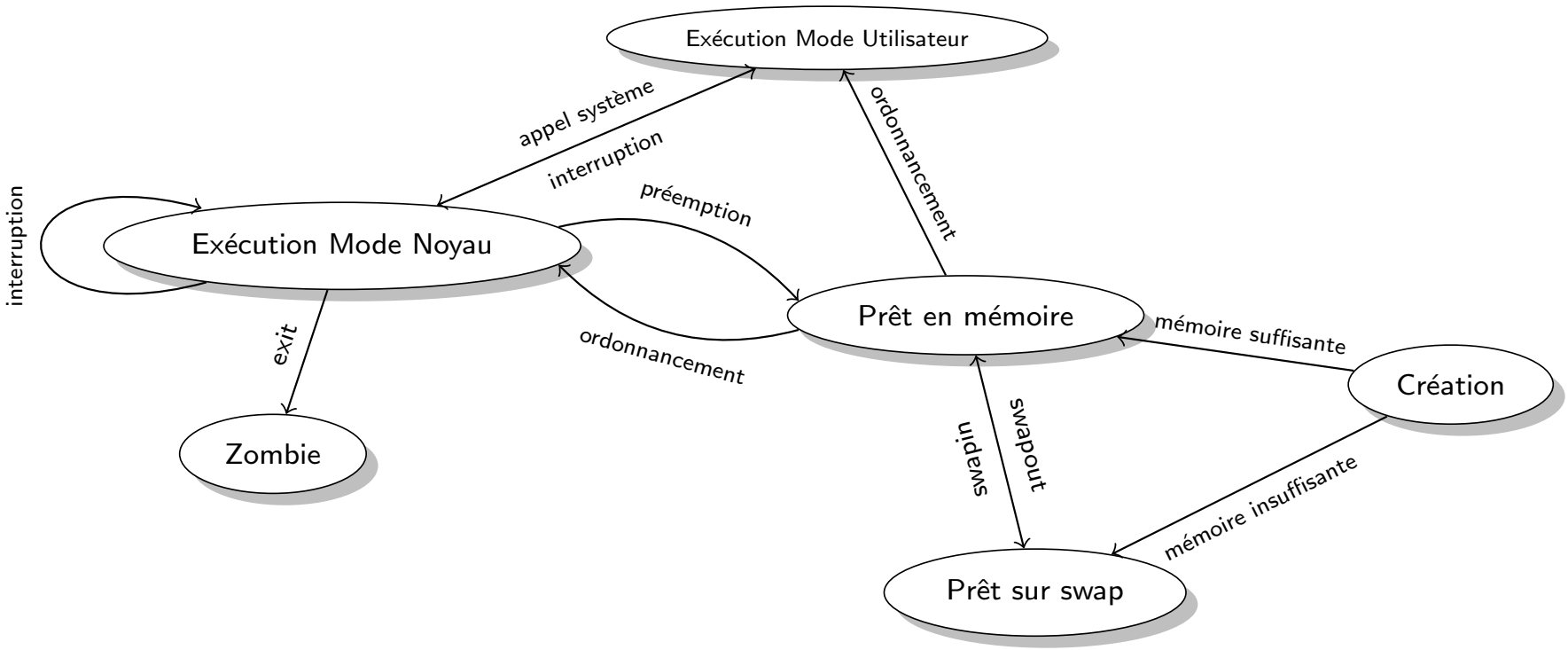
État d'un processus



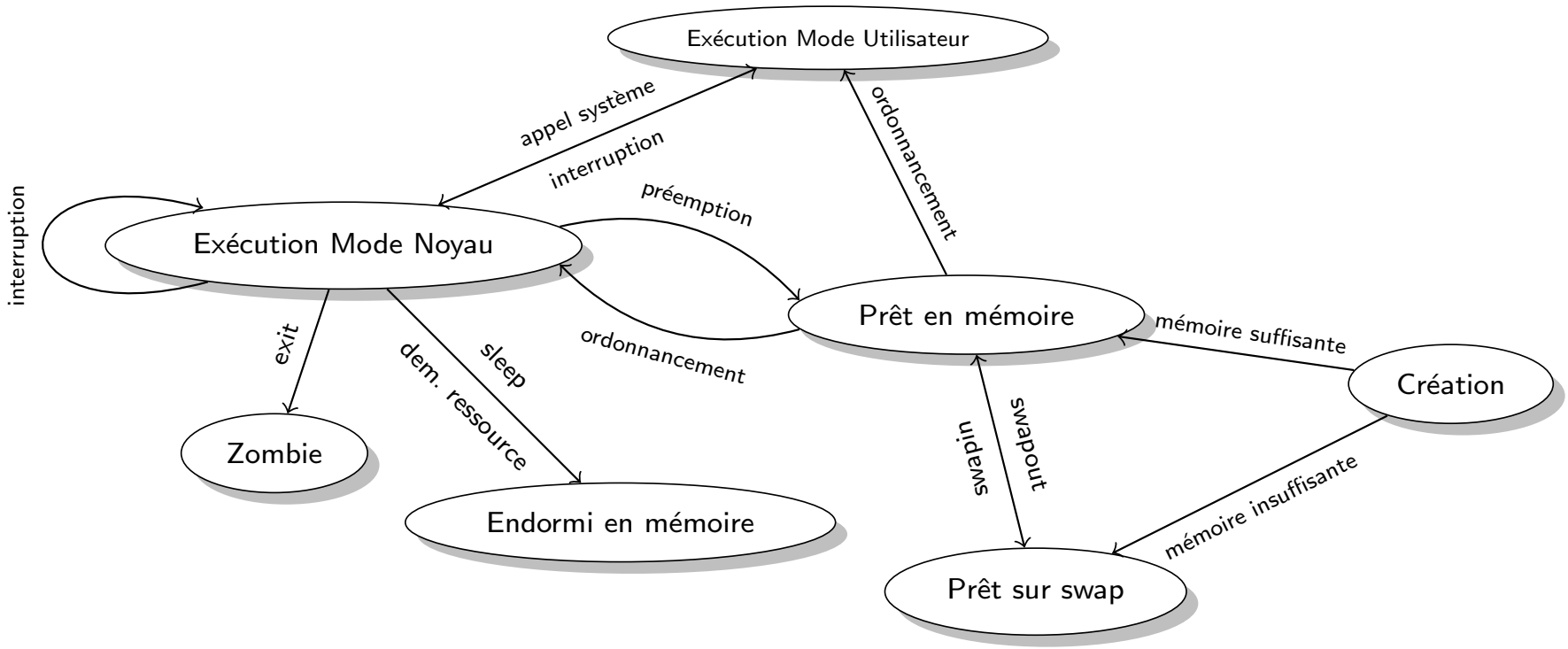
État d'un processus



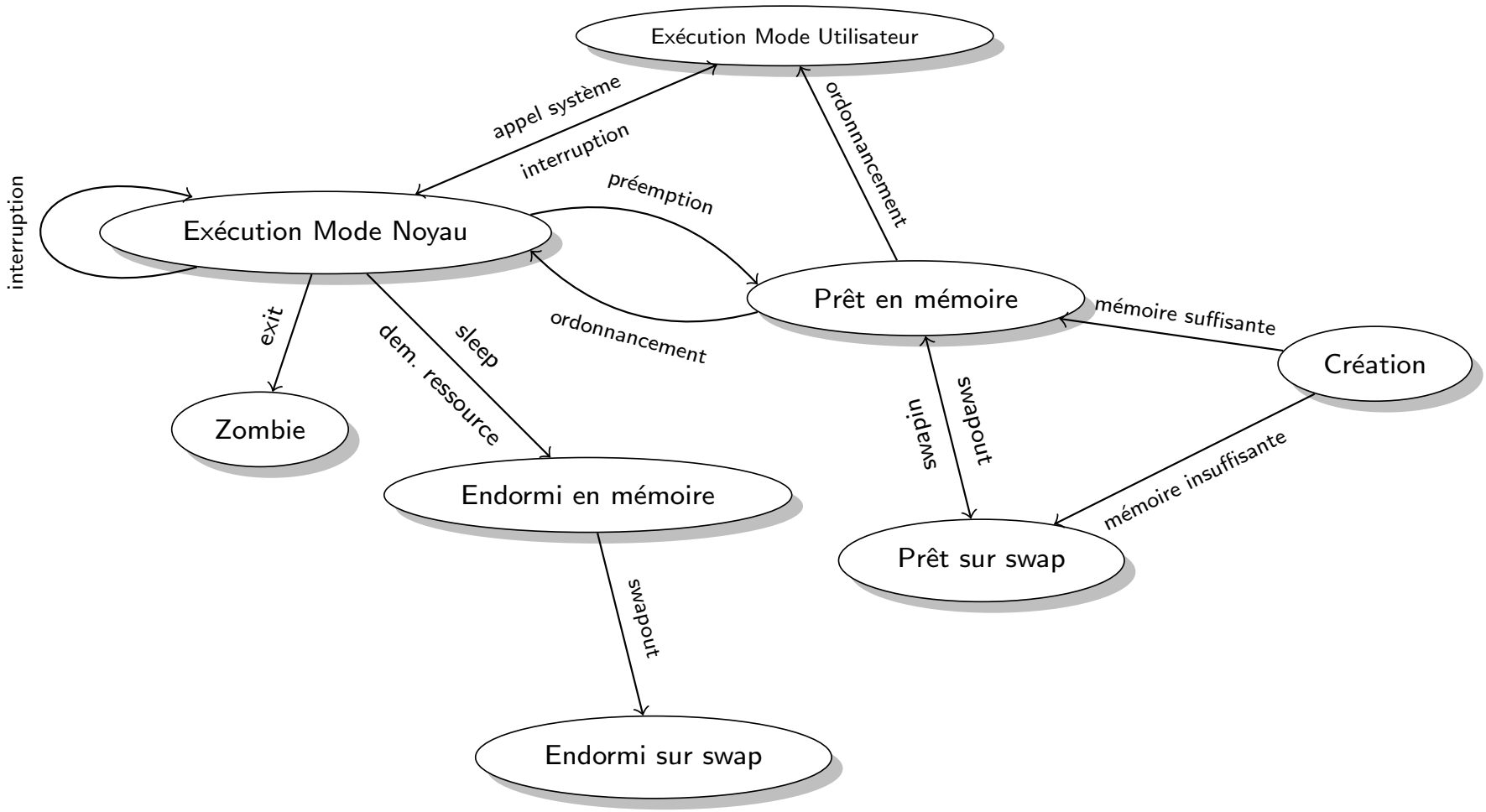
État d'un processus



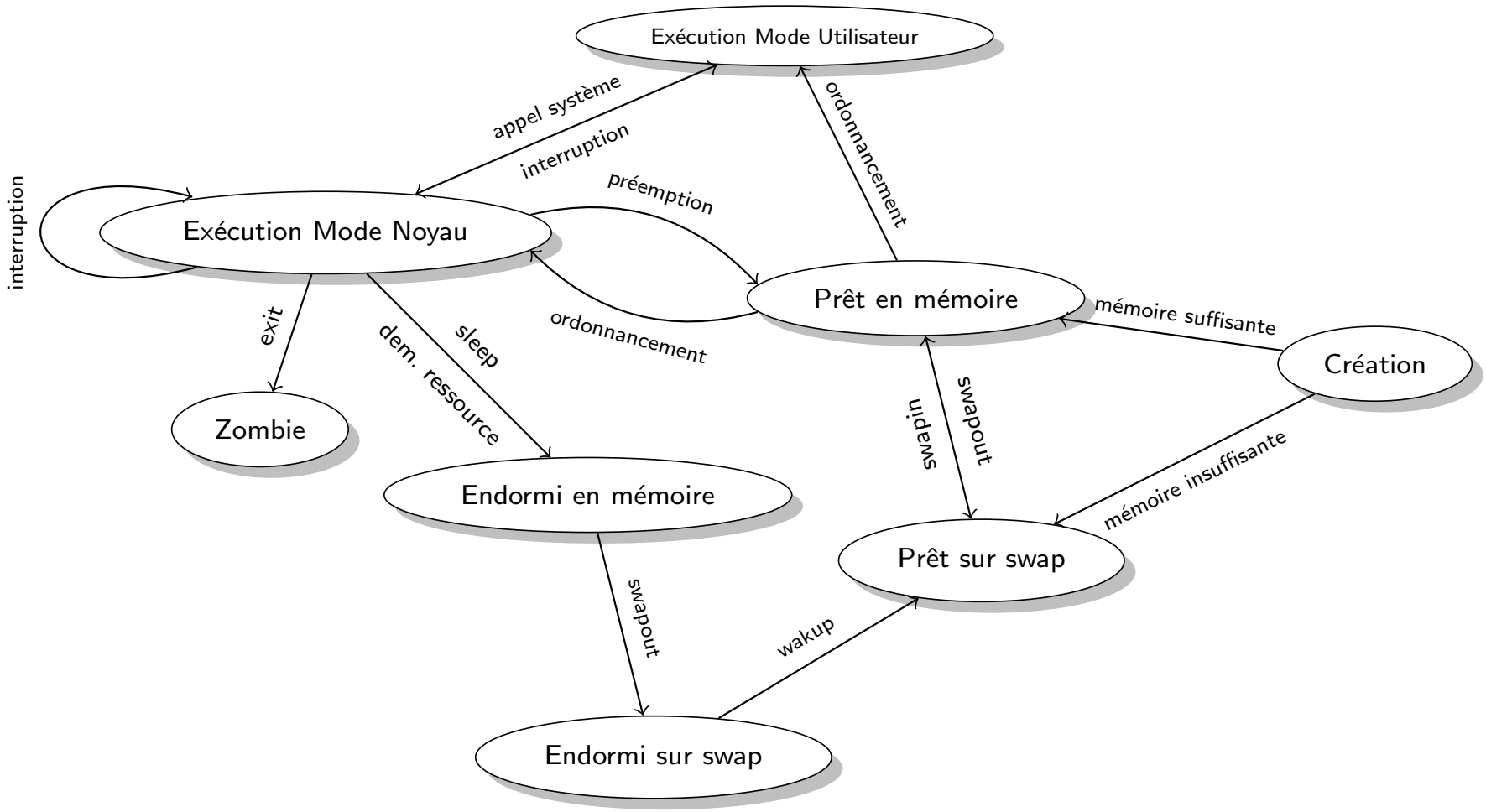
État d'un processus



État d'un processus



État d'un processus



Création de processus

Exemple (La commande `int system(const char * commande);`)

- Permet à un programme d'exécuter une autre commande
- Toute commande reconnue par le shell peut être utilisée
- La fonction stoppe le processus et attend la fin de la commande
- La fonction récupère et retourne le retour de la commande qui peut être traité

Travail à faire

Exemple (La commande `int system(const char * commande);`)

Les étapes sont :

- Le processus « A » crée un nouveau processus et stoppe
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

Les appels système

Comment créer un processus pour éxécuter un programme ?

- Créer un nouveau processus : **fork**() (Unix)
- Remplacer le processus courant par un autre : **exec**() (Unix)
- Les deux à la fois **CreateProcess** (api win32), **NtCreateProcess** (appel système)
- Terminer le processus avec **exit**(), **ExitProcess**()
- Attendre la fin d'un processus fils **waitpid**()
- Le nouveau processus est un clone du premier : même *contexte*

fork() en détails

man **fork**

- pid_t **fork**(void);
- **fork**() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points . . .
- On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created.

Question ?

Que se passe-t'il avec le code suivant :

```
for (i=0; i<10; i++) {
   fork();
}
```

commande system

Exemple (res = system("emacs");)

- Le processus « A » crée un nouveau processus et stoppe.
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

commande system

Exemple (res = system("emacs");)

- **Le processus ≪ A ≫ crée un nouveau processus et stoppe.**
- Le nouveau processus ≪ B ≫ lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

commande system

Exemple (res = system("emacs");)

- **A utilise `fork()` pour se dupliquer et créer B,**
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

commande system

Exemple (res = system("emacs");)

- A utilise `fork()` pour se dupliquer et créer B,
  - **A et B sont des clones, pour l'instant,**
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

commande system

Exemple (res = system("emacs");)

- A utilise `fork()` pour se dupliquer et créer B,
  - A et B sont des clones, *pour l'instant*,
  - **les deux processus doivent être capables de se reconnaître**
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

fork() en détails

Tester le résultat de **fork**()

pour identifier le processus dans lequel on se trouve.

```
pid_t code= fork();
if (code == -1) {
   fprintf(stderr, "Le processus fils n'a pas
            pu être créé car : %s\n", strerror(errno));
   exit(1);
}
if(code == 0) {
   /* Code du fils */
}
else{
   /* Code du père */
}
```

commande system()

Exemple (res = system("emacs");)

- A utilise `fork()` pour se dupliquer et créer B
- Le nouveau processus « B » lance la commande
- A attend la fin de B
- A récupère le retour de B
- A reprend son cours

commande system()

Exemple (res = system("emacs");)

- A utilise `fork()` pour se dupliquer et créer B
  - A attend la fin de B
  - B lance la commande
  - B se termine
- A récupère le retour de B
- A reprend son cours

commande system()

Exemple (res = system("emacs");)

- A utilise `fork()` pour se dupliquer et créer B
  - **A attend la fin de B**
  - B lance la commande
  - B se termine
- **A récupère le retour de B**
- A reprend son cours

Fin d'un processus

- Lorsqu'il se termine un processus passe en état *zombi* : il n'est plus jamais prêt mais ses ressources ne sont pas totalement libérées
- Notamment le résultat (le retour de la fonction main()) est en attente
- Le père doit lire le résultat du processus par l'appel **waitpid**()
- Si le père meurt, le processus est adopté par `Init` qui libère les ressources sans lire le résultat

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

À quoi correspondent les arguments ?

waitpid() en détails

pid_t **waitpid**(pid_t pid, int *status, int options);

- The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state. By default, waitpid() waits only for terminated children, but this behavior is modifiable via the options argument...
- The value of pid can be :
  - -1 meaning wait for any child process.
  - > 0 meaning wait for the child whose process ID is equal to the value of pid.
  - ...
- If status is not NULL, wait() and waitpid() store status information in the int to which it points.
- The value of options is an OR of zero or more ...

Utilisation de **waitpid**()

```
pid_t code= fork();
if (code == -1) {... /* traitement de l'erreur */}
if(code == 0) {/* Code du fils */
   ...
   exit(23);
}
else{
   /* Code du père */
   int status;
   ...
   waitpid(-1,&status, 0); /* Attente d'un fils */
   if (WIFEXITED(status)) {
      fprintf(stdout, "Le fils a retourné %d\n",
                        WEXITSTATUS(status));
   }
}
```

Hiérarchie de processus

Exemple

- en TP, après avoir compilé un programme,
- ./main
- que se passe-t-il ?
- Le shell utilise **fork**() pour créer un nouveau processus,
- le nouveau processus remplace les instructions du shell par celles de votre programme avec **exec**().
- Le shell attends que votre programme se termine avec **waitpid**().
- et avec ./main & ?

Hiérarchie de processus

Qui a créé le shell ? qui a créé le processus qui a créé le shell ?
Voir la commande pstree().

- résultat : tous les processus sont créés . . .
  par un autre processus
- Il faut un premier processus ancêtre de tous les autres

Exemple (linux)

- Sous linux, Init est le processus à l'origine de tous les autres
- C'est lui qui lance tous les processus système
- Sa configuration /etc/inittab ou systemd, /etc/init.d, /etc/rc.d, /etc/systemd/ permet de gérer les ≪ services ≫du système

Question ?

Donnez des exemples de « service ».

L'ordre a-t'il une importance ?

Communication entre processus

Le système sépare les processus :

- pour éviter les perturbations
- pour assurer la protection des données
- pour faciliter la gestion

Donc *sans faire appel au système, un processus ne peut rien faire pour agir sur un autre*

Mais la communication entre processus est nécessaire

- certains processus doivent communiquer *e.g.* serveur
- tout processus nécessite un moyen d'être contacté (afin de pouvoir l'annuler par exemple)
- pour des raisons de performance, certains doivent fonctionner sans ces sécurités (virtualisation)

Exemple

Lorsqu'on appuie sur une touche :

1. Le matériel prévient le système
2. qui prévient le processus « serveur graphique »
3. qui prévient le logiciel concerné
4. qui calcule une nouvelle image
5. et demande au serveur graphique de l'afficher

Exemple

Lorsqu'on appuie sur une touche :

1. Le matériel prévient le système
2. qui prévient le processus « serveur graphique »
3. qui prévient le logiciel concerné
4. qui calcule une nouvelle image
5. et demande au serveur graphique de l'afficher

Il faut être capable :

- De prévenir un processus que quelque chose s'est passé
- D'échanger des informations avec ce processus

Cas similaires

De manière générale

- Pour prévenir, il faut des alarmes (sonnerie, sirène, réveil) qui dérangent et vous obligent à faire une action prédéfinie
- Pour communiquer, il faut des moyens de communication (téléphone, lettre/mail, média, ...) qui véhiculent l'information

Cas similaires

De manière générale

- Pour prévenir, il faut des alarmes (sonnerie, sirène, réveil) qui dérangent et vous obligent à faire une action prédéfinie
- Pour communiquer, il faut des moyens de communication (téléphone, lettre/mail, média, ...) qui véhiculent l'information

Pour communiquer avec le système, le matériel utilise :

- Des *interruptions* pour prévenir et dérouter le processus en cours. À chaque interruption correspond une action prédéfinie
- Des zones mémoires pour échanger les informations, l'accès à ces zones est sécurisé par le système (cela fait partie du mode noyau)

Cas similaires

De manière générale

- Pour prévenir, il faut des alarmes (sonnerie, sirène, réveil) qui dérangent et vous obligent à faire une action prédéfinie
- Pour communiquer, il faut des moyens de communication (téléphone, lettre/mail, média, ...) qui véhiculent l'information

Pour communiquer avec le système, le matériel utilise :

- Des *interruptions* pour prévenir et dérouter le processus en cours. À chaque interruption correspond une action prédéfinie
- Des zones mémoires pour échanger les informations, l'accès à ces zones est sécurisé par le système (cela fait partie du mode noyau)

De la même manière, pour communiquer, les processus disposent :

- De *signaux* qui sont des alarmes qui stoppent le fonctionnement du processus pour lui faire exécuter une fonction *gestionnaire de signal*
- De *tubes*, *sockets*, fichiers, partages mémoires, ...

Signaux

Définition

Un signal est un moyen de prévenir un processus d'un évènement :

- c'est un message simple : un petit nombre de messages codés
  - SIGINT=Ctrl−c,
  - SIGSTOP=Ctrl−z,
  - SIGKILL=**kill** −9 ...,
  - SIGSEGV=erreur de segmentation,
  - ...
- c'est un évènement asynchrone : il peut arriver n'importe quand
- il doit être traité donc déroute le processus (fonctions prédéfinie)
- la communication est limitée : type du signal, émetteur...

Par défaut un signal a un comportement défini. Pour changer ce comportement il faut mettre en place une *fonction d'appel* avant l'arrivée du signal

Signal et programmation événementielle

Un certain nombre d'entre vous ont déjà manipulé une notion proche : *l'évènement* dans la programmation d'IHM (évènement en `GTK`, `EventListener` en `java`, `slot` et `signaux` en `Qt`, boucle principale d'évènement en `SDL`. Par exemple en javascript :

```
document.getElementById("myBtn").addEventListener("click", displayDate);
```

qui associe la fonction displayDate au fait de clicker sur le bouton document.getElementById("myBtn").

Point communs

- Mise en place de *fonction d'appel* : des fonctions qui seront appelées automatiquement lorsque l'évènement survient
- On ne maîtrise pas les paramètres ni le retour de la fonction d'appel (dans cet exemple il n'y a pas de paramètres)
- On ne maîtrise pas le moment où l'évènement arrive.

Signal et programmation événementielle

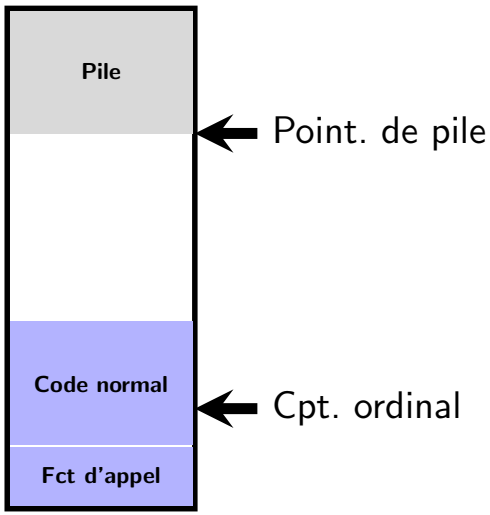
Mais il y a quelques différences :

- Le signal s'applique sur le processus entier :
  - en événementiel, on applique des traitements en fonction du signal et du widget concerné (le bouton, le menu,...) $\Rightarrow$ gestion précise
  - avec des signaux, on doit changer les traitements en fonction des phases du programmes

Signal et programmation événementielle

Mais il y a quelques différences :

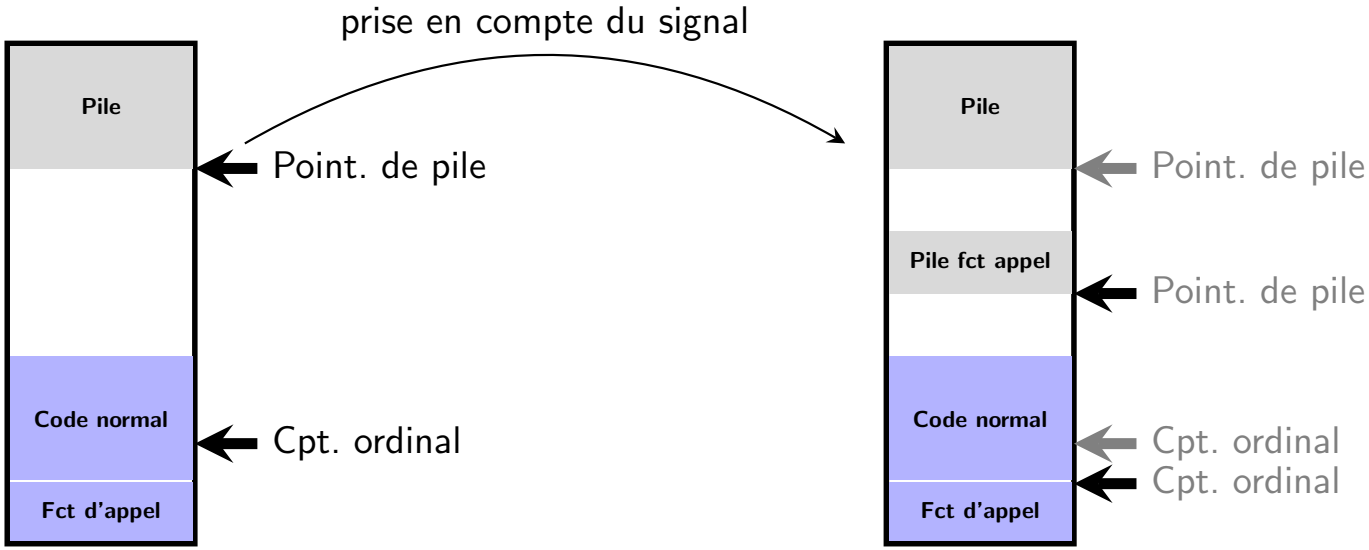
- Le signal s'applique sur le processus entier :
  - en événementiel, on applique des traitements en fonction du signal et du widget concerné (le bouton, le menu,...) ⇒ gestion précise
  - avec des signaux, on doit changer les traitements en fonction des phases du programmes
- Le traitement est immédiat, on interrompt et déroute le processus



Signal et programmation événementielle

Mais il y a quelques différences :

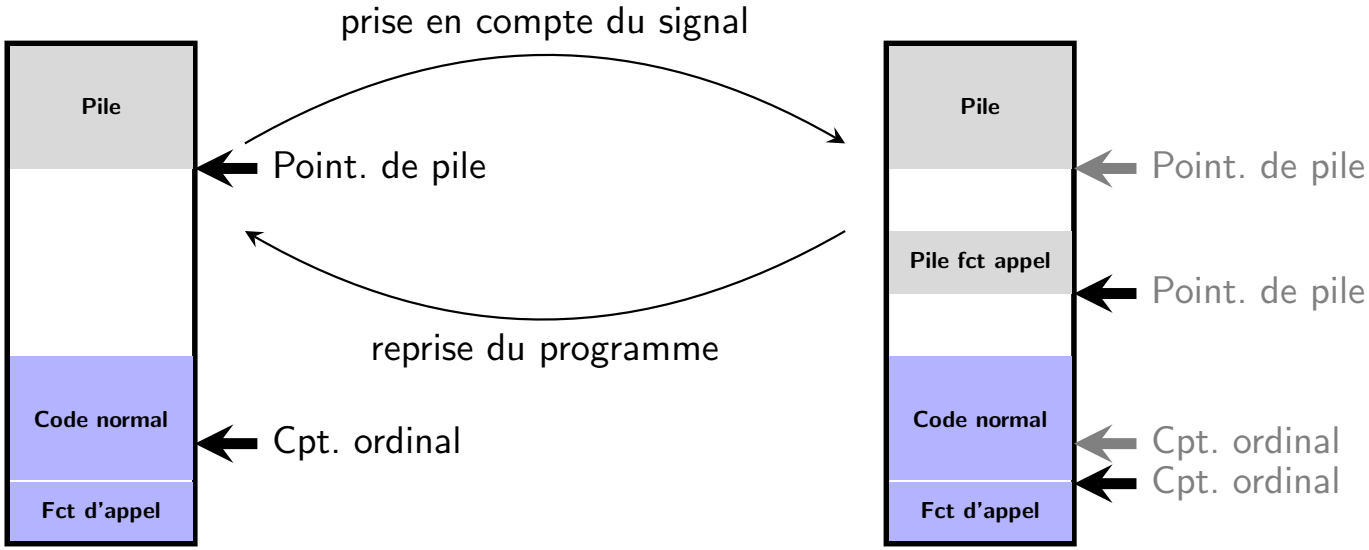
- Le signal s'applique sur le processus entier :
  - en événementiel, on applique des traitements en fonction du signal et du widget concerné (le bouton, le menu,...) ⇒ gestion précise
  - avec des signaux, on doit changer les traitements en fonction des phases du programmes
- Le traitement est immédiat, on interrompt et déroute le processus



Signal et programmation événementielle

Mais il y a quelques différences :

- Le signal s'applique sur le processus entier :
  - en événementiel, on applique des traitements en fonction du signal et du widget concerné (le bouton, le menu,...) ⇒ gestion précise
  - avec des signaux, on doit changer les traitements en fonction des phases du programmes
- Le traitement est immédiat, on interrompt et déroute le processus



Fonctions

Pour envoyer un signal à un processus :

- en shell : **kill** −s **signal** pid ou autre (<ctrl−C>, <ctrl−Z> ...)
- par programme : int **kill** (pid_t pid , int sig);
- signal : voir <**signal**.h>

Fonctions

Pour envoyer un signal à un processus :

- en shell : **kill** −s **signal** pid ou autre (<ctrl−C>, <ctrl−Z> ...)
- par programme : `int kill ( pid_t pid , int sig );`
- signal : voir <**signal**.h>

Changer le comportement par défaut

- Pour traiter un signal (version simple, d'origine) :

```
typedef void (* sighandler_t )( int );
sighandler_t signal( int signum, sighandler_t handler );
```

- Pour traiter un signal (version BSD) :

```
int sigvec( int sig , struct sigvec *vec, struct sigvec *ovec);
```

- Pour traiter un signal (version POSIX : à utiliser) :

```
int sigaction ( int signum, const struct sigaction * act ,
                                    struct sigaction *oldact );
```

Fonctions

Pour envoyer un signal à un processus :

- en shell : **kill** −s **signal** pid ou autre (<ctrl−C>, <ctrl−Z> ...)
- par programme : int **kill** (pid_t pid , int sig);
- signal : voir <**signal**.h>

Changer le comportement par défaut

- Pour traiter un signal (version simple, d'origine) :
  typedef void (∗ sighandler_t)(int);
  sighandler_t **signal**(int signum, sighandler_t handler);
- Pour traiter un signal (version BSD) :
  int **sigvec**(int sig , struct **sigvec** ∗vec, struct **sigvec** ∗ovec);
- **Pour traiter un signal (version POSIX : à utiliser) :**
  int **sigaction** (int signum, const struct **sigaction** ∗ act ,
  struct **sigaction** ∗oldact);

Exemple :

La fonction d'appel :

```
void fonction(int s) {
  fprintf(stderr, "M'enfin !!\n");
}
```

On met en place le traitement qui sera déclenché lorsque l'évènement arrive :

```
int main(int argc, char *arv[]) {
  sighandler_t old;
  old = signal(SIGINT, fonction);
  if (old == SIG_ERR) {
    perror("signal");
    exit(1);
  }
  while(1) {
    fprintf(stdout, "rrrrrr..rrrr..\n");
    sleep(rand()%5);
  }
  return 0;
}
```