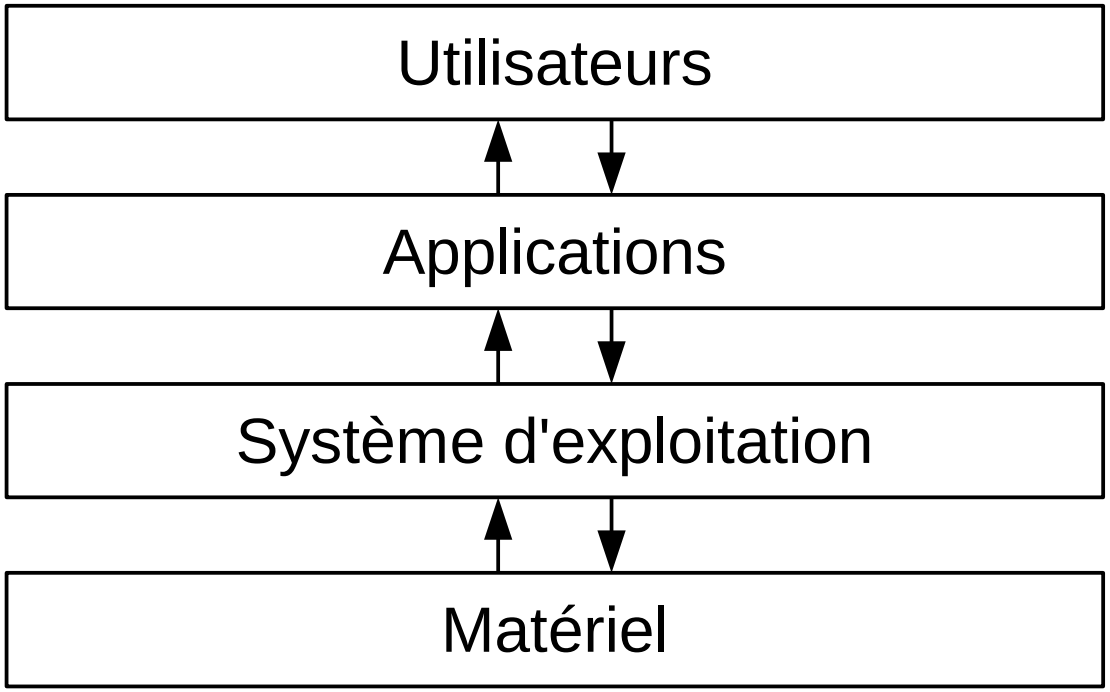
Programmation système

Plan du cours

0) Introduction
1) Les processus
2) Le stockage sur disque dur
3) Les tubes
4) Les signaux
5) La gestion de la mémoire
6) La gestion des ressources

Qu'est-ce qu'un système ?

- Programme central qui fait l'interface entre le matériel et les applications



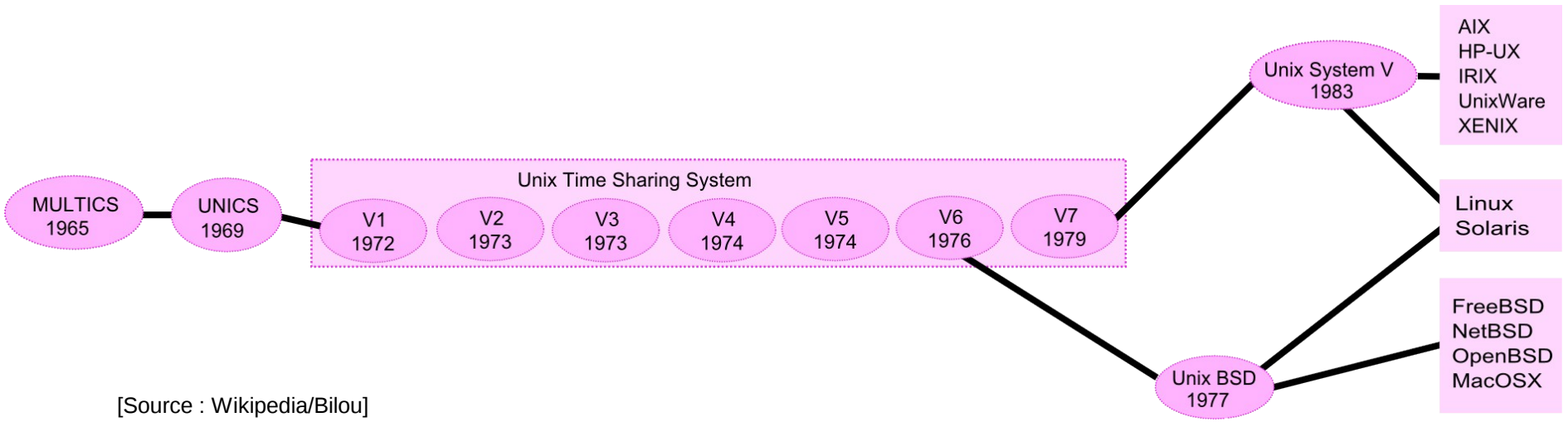
Rôle d'un système

- Masquer le matériel
- Gérer et partager les ressources
  - Processus (ordonnancement, communication)
  - Mémoire
  - Système de fichiers
  - Réseau

Famille de systèmes

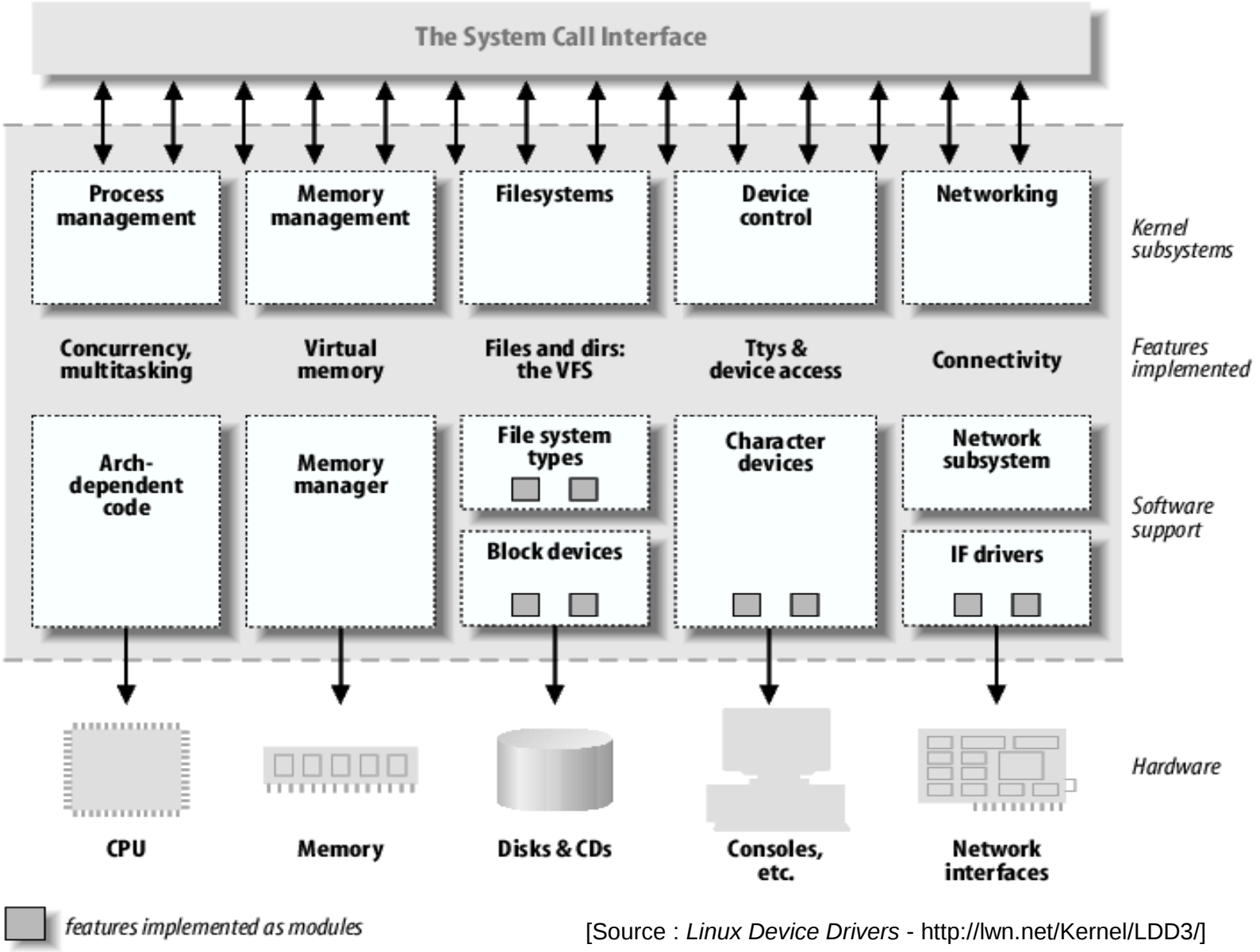
- DOS
- Windows
- UNIX
- AS400
- *etc.*

- AIX (IBM)
- Solaris, Open Solaris
- LynxOS (RTOS)
- QNX (RTOS)
- Linux
- OpenBSD, FreeBSD
- NetBSD
- MacOS X



[Source : Wikipedia/Bilou]

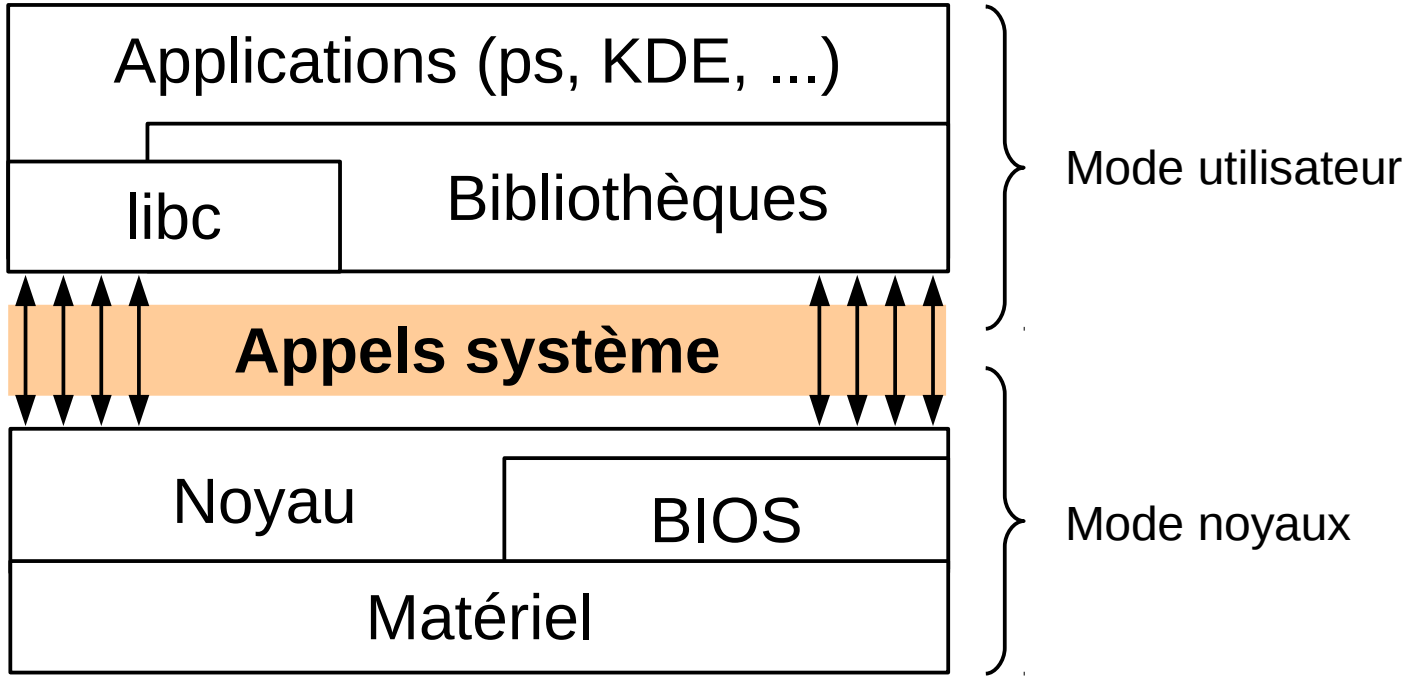
Fonctionnalités du noyau Linux



[Source : *Linux Device Drivers* - http://lwn.net/Kernel/LDD3/]

Les appels systèmes

- L'accès à la couche « système » à partir de la couche « applicative » se fait au travers de l'**API des appels système**.



Vous avez dit API ?

- API = Application Programming Interface

Programme

```
int a, b, c;
a = 2;
b = 3;
c = mult(a, b);
printf("%d", c);
```

API

```
int mult(int, int);
```

mult est une fonction qui prend en argument 2 nombres entiers et qui renvoie leur produit.

Librairie ou Système

Implémentation #1

```
int mult(int a, int b)
{
  return a * b;
}
```

Implémentation #2

```
int mult(int a, int b)
{
  int i, ret=0;
  for (i=1; i<=b; ++i)
    ret += a;
  return ret;
}
```

Systèmes « POSIX »

- POSIX = Portable Operating System Interface (uniX)
- Ensemble de standards de l'IEEE (Institute of Electrical and Electronics Engineers)
- POSIX définit notamment :
  - Les commandes shell de base (ksh, ls, man, …)
  - L'API (Application Programming Interface) des appels systèmes
  - Les extensions « temps réel »
  - L'API des threads (processus légers)

Objet du cours

- L'objet du cours de programmation système est la présentation de différents appels systèmes de la norme POSIX.
- Ces appels seront regroupés en grandes catégories :
    1) Les processus
    2) Le stockage sur disque dur
    3) Les tubes
    4) Les signaux
    5) La gestion de la mémoire
    6) La gestion des ressources

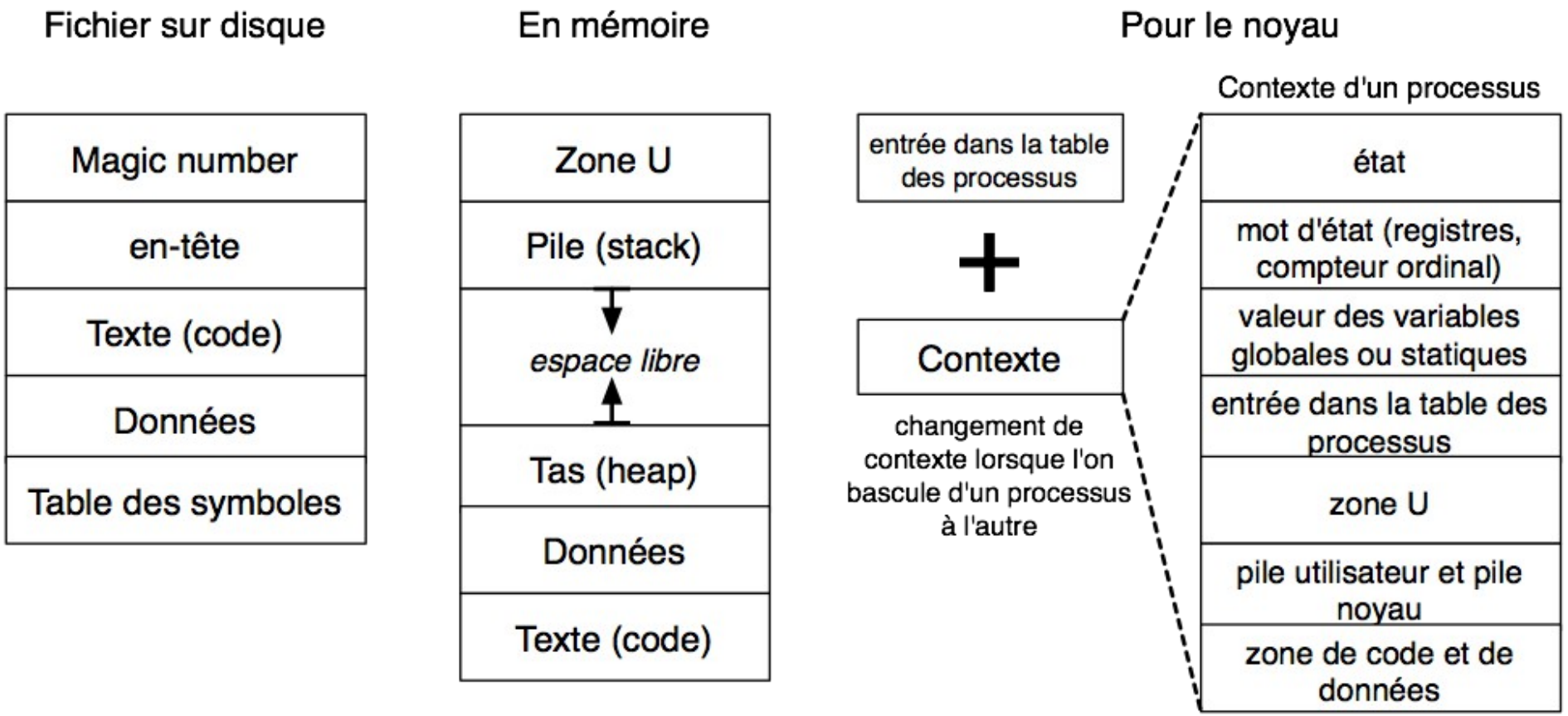
Chapitre 1
Les processus

Plan du cours

Notion de processus (1)

- Un processus est l'instance dynamique d'un programme
- Le programme contient du code et des données
- Il faut beaucoup plus d'informations pour décrire un processus dans le système
- Toutes les informations permettent au système d'exécuter plusieurs processus « en même temps » en passant de l'un à l'autre

Notion de processus (2)



Certaines de ces données (table des processus, zone U) sont détaillées dans la suite.

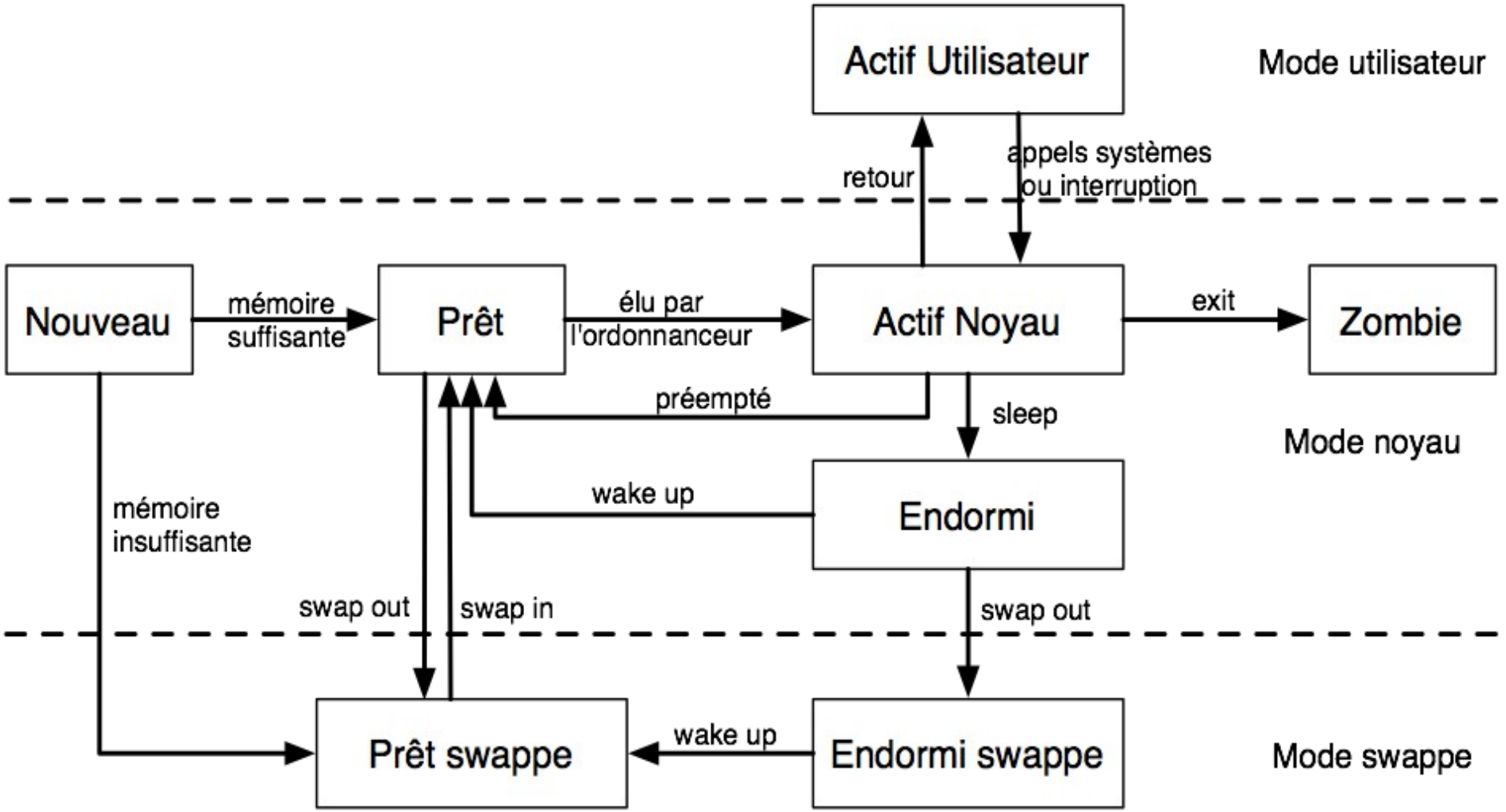
La Zone "U"

- Parmi les données caractérisant le processus
  - **uid** réel et effectif de l'utilisateur
  - Compteurs des **temps** (users et system) consommés
  - Masque de **signaux** (cf plus tard...)
  - **Code d'erreur** de la dernière erreur rencontrée pendant un appel système.
  - **Répertoire courant** (*pwd*) et la racine courante (*chroot*)
  - Table de **références sur descripteurs de fichier** (mais pas les descripteurs eux-mêmes)
  - **Limites** (man *ulimit*)
  - *umask* : **masque de création de fichiers** par défaut

Exécution des processus (1)

- Un processeur ne peut exécuter qu'un processus à la fois
  - Processeurs multi-cœurs
- Le système bascule sans arrêt d'un processus à l'autre pour permettre le multi-tâches
- Les processus peuvent se trouver dans différents états suivant qu'ils sont exécutés, en attente, endormis...

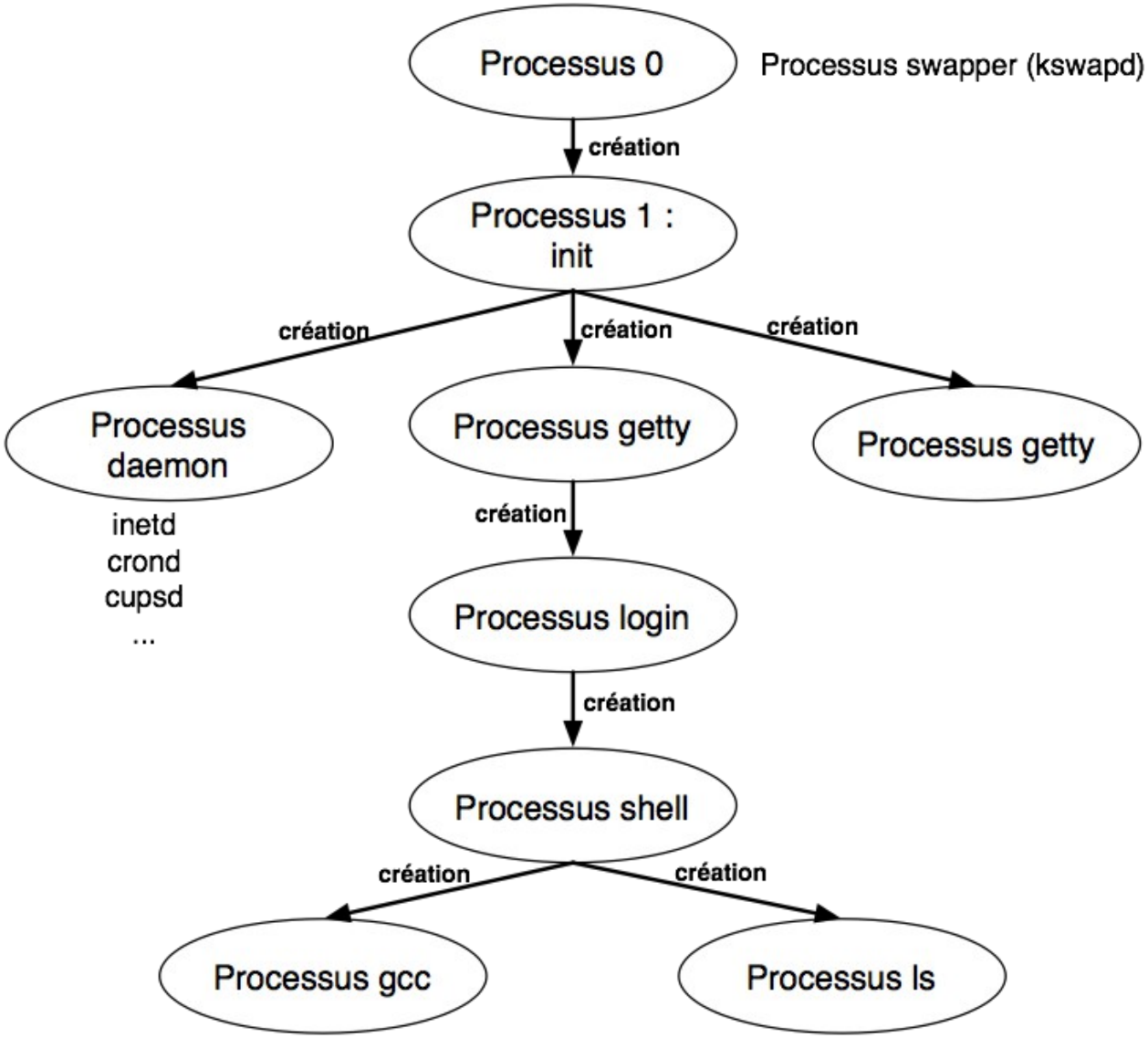
Exécution des processus (2)



Création des processus (1)

- Les processus ont tous une relation **père/fils** : **arborescence** de processus.
- Le processus initial est 0 (<*swapper*>) ; il donne naissance à <*init*> (1).
- Un nouveau processus est toujours créé comme fils d'un autre et n'importe quel processus peut donner naissance à un fils.
- Un processus père devrait attendre la mort de ses fils avant de se terminer.
- Si un père décède avant ses fils, ceux-ci sont adoptés par le processus <*init*> pour ne pas être orphelins.

Création des processus (2)



Création des processus (3)

- Un processus dispose d'un **PID** (**P**rocessus **ID**) et d'un **P**PID (**P**arent Processus ID).
- Un PPID est un PID, de type **`pid_t`** (entier).
- Une simple commande « **`ps`** » permet de vérifier l'arborescence :

```
$> ps axo stat,ppid,pid,tty,user,comm
STAT  PPID    PID    TT       USER       COMMAND
S     0       1      ?        root       init
S     1       2      ?        root       keventd
Ss    1       816    ?        root       inetd
Ss+   1       1119   tty1     root       getty
S     23094   6851   ?        www-data   apache2
S+    14970   14974  pts/2    toto       gnuplot
R+    14956   17569  pts/1    toto       ps
```

Création des processus (4)

- Pour donner naissance à un processus, le père se clone et la nouvelle instance charge un nouveau code.
- Primitives systèmes :
  - **`fork()`** : permet à un processus de se cloner
  - **`exec()`** : remplace le code (en mémoire) du clone par celui du processus à exécuter (lu sur le disque). C'est une primitive dite de recouvrement.
  - **`wait()`** : permet au père d'être notifié de la mort d'un processus fils et de récupérer des informations sur cette terminaison. Appel bloquant.
- Le signal SIGCHLD permet également à un fils d'informer son père d'un changement.

Utilisation de `fork()` (1)

- **`fork()`** permet à un processus de se cloner.
- Toutes les données du processus (Zone U) sont dupliquées, à l'exception de son PID et de son PPID.
- Les descripteurs de fichiers ouverts du père sont dupliqués : le fils a donc les mêmes fichiers ouverts que son père.
- Les stats du fils (temps d'exécution, etc.) sont remises à 0.

Utilisation de `fork()` (2)

- Fichiers d'entête (headers) :

```
#include <sys/types.h>
#include <unistd.h>
```

- Le type `pid_t` est un entier.
- `pid_t fork(void);`
- Autres primitives utiles :

```
pid_t getpid(void);   /* Quel est mon PID ? */
pid_t getppid(void);  /* Quel est le PID de mon père ? */

char* getcwd(char* buf, size_t size);  /* Quel est mon ...
char* getwd(char* buf);                ... working directory ? */
```

Utilisation de `fork()` (3)

```
int main(int argc, char* argv[])
{
  pid_t child = fork(); /* à partir de là, le père et le fils
                           exécutent le même code */

  switch(child) /* la différence se fait ici */
  {
    case -1 :
       perror("fork");
       exit(errno);
    case  0 : /* code du fils */
      printf("Fils : mon PID est %d\n", getpid());
      printf("Fils : le PID de mon pere est %d\n", getppid());
      break;
    default : /* code du père */
      printf("Père : le PID de mon fils est %d\n", child);
  }

  printf("%d : Cette phrase s'affiche dans les 2 processus\n", getpid());

  return 0;
}
```

Gestion des erreurs systèmes (1)

- Les fonctions système renvoient un code de retour.
  - Exemple : **`fork()`** renvoie -1 en cas d'erreur
  - Exemple : **`getenv()`** renvoie NULL en cas d'erreur
- Pas de détail sur le type d'erreur dans code de retour
- Pour connaître la nature de l'erreur :
  - Variable globale **`errno`**
  - Définie dans **`<errno.h>`**
  - Si pas d'erreur, la valeur de **`errno`** ne veut rien dire
  - Propre à un appel système:
    - Voir le descriptif adéquat dans le *man*

Gestion des erreurs (2)

- Affichage du message d'erreur:
  - `void perror(const char *msg)`
- **`perror()`** affiche un message sur la sortie d'erreur standard qui décrit la dernière erreur rencontrée lors d'un appel système.
- Voir aussi le descriptif dans le *man*
- En programmation système (comme ailleurs), la gestion des erreurs est **obligatoire** !

Utilisation de `fork()` (3)

```
int main(int argc, char* argv[])
{
  pid_t child = fork(); /* à partir de là, le père et le fils
                           exécutent le même code */

  switch(child) /* la différence se fait ici */
  {
     case -1 :
        perror("fork d'exemple");
        exit(errno);
    case  0 : /* code du fils */
      printf("Fils : mon PID est %d\n", getpid());
      printf("Fils : le PID de mon pere est %d\n", getppid());
      break;
    default : /* code du père */
      printf("Père : le PID de mon fils est %d\n", child);
  }

  return 0;
}
```

La commande shell **strace**

- La commande shell **strace** permet de tracer les appels système (et les signaux).
  - Votre meilleure amie en progsys!
- Exemple :

```
$> strace cat /dev/null
[...]
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
[...]

$> strace cat /dev/lapinblanc
[...]
open("/dev/lapinblanc", O_RDONLY|O_LARGEFILE)=-1
                ENOENT (No such file or directory)
[...]
```

Utilisation de **`wait()`** (1)

- Après un **`fork()`**, le père peut se mettre en **`wait()`**.
- **`wait()`** est bloquant et attend n'importe quel fils.
- **`waitpid()`** est bloquant et peut attendre n'importe quel fils ou l'un d'eux en particulier.

```
#include <sys/types.h>
#include <wait.h>

/* Attente terminaison d'un fils, récupère des infos dans status */
pid_t wait(int* status);

/* Attendre un fils précis (ou n'importe lequel) */
pid_t waitpid(pid_t wpid, int* status, int options);
```

Utilisation de **`wait()`** (2)

- L'interprétation de **`status`** se fait avec des macros :
  - **`WEXITSTATUS, WCOREDUMP, ...`** **(cf man 2 wait)**

```
/* [...] dans le père, après un fork() */

int status = 0;
pid_t pid;

if ( ( pid = wait(&status) ) == -1) {
  perror("wait");
  exit(errno);
}

printf("mon fils %d s'est terminé avec le code %d\n",
            pid, WEXITSTATUS(status));

/* [...] */
```

Utilisation de **wait()** (3)

```
int main(int argc, char* argv[])
{
  int status = 0;
  pid_t returnCode;
  pid_t child = fork();
  switch (child)
  {
    case -1:
      perror("fork"); exit(errno);
    case 0:  /* dans le fils */
      printf("Fils : mon pid est %d\n", getpid());
      break;
    default: /* dans le père */
      printf("Pere : le pid de mon fils est %d\n", child);
      returnCode = wait(&status);
      if (returnCode != -1)
        printf("mon fils %d s'est terminé avec le code %d\n",
               returnCode, WEXITSTATUS(status));
  }
  return 0;
}
```

Utilisation de **exec( )** (1)

- **exec( )** est une primitive de **recouvrement** qui remplace le code à exécuter du processus par un autre.
- **exec( )** lance un processus avec des arguments, comme on le ferait sur la ligne de commande.
- La famille des **exec( )** permet simplement de spécifier les arguments du processus fils de différentes façons.

```
#include <unistd.h>

/* famille de exec() */
int execl(const char *path, const char *arg, ... , NULL);
int execlp(const char *file, const char *arg, ... , NULL);
int execle(const char *path, const char *arg, ..., NULL,
           char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvP(const char *file, const char *search_path,
           char *const argv[]);
```

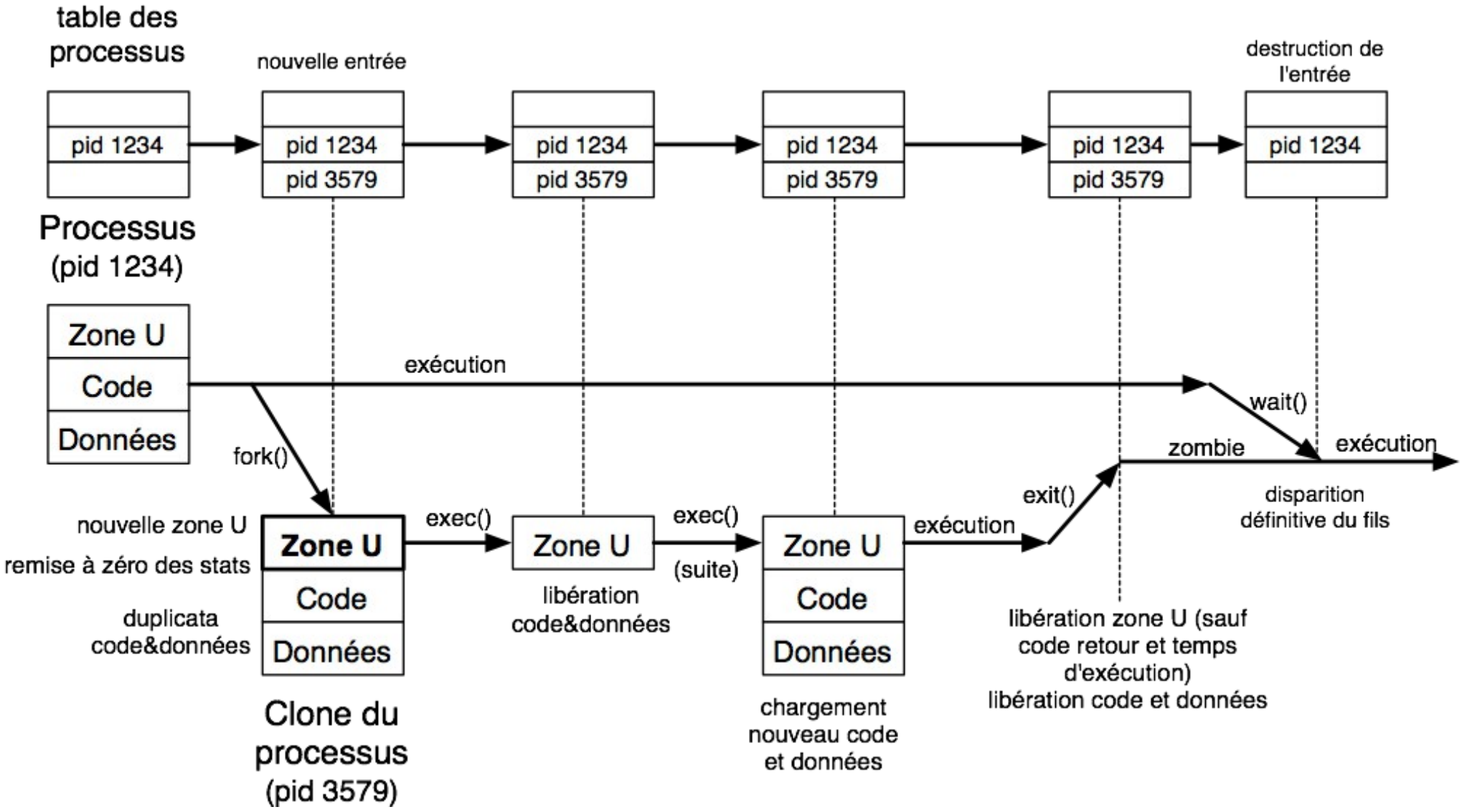
Utilisation de **exec( )** (2)

```
int main(int argc, char* argv[])
{
  int status = 0;
  pid_t returnCode;
  int error = 0;
  pid_t child = fork();
  switch (child)
  {
    case -1:
      perror("fork"); exit(errno);
    case 0:  /* dans le fils */
      printf("Remplacement du code, ici exécution de <ls -l>\n");
      error = execl("/bin/ls", "ls", "-l", NULL);
      printf("Ceci ne s'affichera jamais, sauf si le exec() a échoué\n");
      fprintf(stderr, "Le execl() a échoué avec le code %d\n", error);
      exit(errno);
    default: /* dans le père */
      printf("Je suis le père, le PID de mon fils est %d\n", child);
      if (wait(&status) == -1) { perror("wait"); exit(errno); }
  }
  return 0;
}
```

Mort des processus

- Un processus ne disparaît pas tout de suite à la fin de son exécution.
- En effet, quand un processus se termine (appel à **`exit()`**), il devient alors **zombie**.
- Il ne disparaitra que lorsque le père en aura pris connaissance (dans le **`wait()`** ou via un signal).
- <init> est toujours en **`wait()`**, de telle sorte qu'il permet aux zombies adoptés de se terminer.

Récapitulatif de la vie d'un processus



Partage des données entre père et fils

- Les **données** (les variables) sont dupliquées, (exception : les descripteurs de fichiers ouverts).
- La mémoire **n'**est **pas partagée** entre père et fils : les modifications dans un processus ne sont pas visibles dans l'autre.
- Duplication par *copy-on-write* (optimisation non négligeable).
- Pour utiliser de la mémoire partagée, voir les API du système (**`shm_open()`**, **`shm_unlink()`**...).

Arguments de **`exec()`** et de la ligne de commande : **`argc`** et **`argv`**

- **`int main(int argc, char* argv[])`** : pourquoi ?
- **`argc`** : nombre d'arguments sur la ligne de commande.
- **`argv`** : les arguments ; tableau de **`argc`** chaîne de caractères.
- **`argv[0]`** : nom du programme (**`argc`** est **toujours** >= 1).

```
int main(int argc, char* argv[])
{
  int i = 0;
  for (i = 0; i < argc; ++i)
    printf("argument %d : %s\n", i, argv[i]);
  return 0;
}
```

```
/home/prof>./toto -alF ab 12
argument 0 : ./toto
argument 1 : -alF
argument 2 : ab
argument 3 : 12
/home/prof>
```

Arguments de **`exec()`** et de la ligne de commande : **`argc`** et **`argv`**

- **`exec()`** reçoit des arguments.

```
pid_t child = fork();
switch (child)
{
  case -1:
    perror("fork"); exit(errno);
  case 0:  /* dans le fils */
    error = 0;
    printf("Remplacement du code, ici exécution de <ls -l>\n");
    error = execl("/bin/ls", "ls", "-l", NULL);
    printf("Ceci ne s'affichera jamais, sauf si le exec() a échoué\n");
    fprintf(stderr, "Le execl() a échoué avec le code %d\n", error);
    exit(errno);
  default: /* dans le père */
    printf("Je suis le père, le PID de mon fils est %d\n", child);
    if (wait(&status) == -1) { perror("wait"); exit(errno); }
}
```

Utilisation de l'environnement (1)

- L'environnement est un ensemble de clés/valeurs.
  - Nom de la *variable d'environnement*, valeur en char*
- Exemples de variables : **PS1**, **USER**, **SHELL**, …
- L'environnement du père est hérité par le fils.

```
#include <stdlib.h>

/* récupère la valeur d'une clef */
char* getenv(const char *name);

/* définit une paire clef/valeur */
int setenv(const char *name, const char *value, int overwrite);

/* reçoit une chaîne du type "clef=valeur" et exécute
   setenv("clef", "valeur", 1)*/
int putenv(const char *string);

/* supprime une clef (et donc sa valeur) */
void unsetenv(const char *name);
```

Utilisation de l'environnement (2)

- L'environnement hérité est celui précédant le **`fork()`**.

```
int main(int argc, char* argv[])
{
  pid_t child;
  int status;
  pid_t returnCode;
  setenv("toto", "titi", 1); /* modification de l'environnement, avant le fork */
  child = fork();
  switch (child)
  {
    case -1 :
      perror("fork"); exit(1)
    case 0 : /* dans le fils */
      printf("toto = %s\n", getenv("toto")); /* affiche "titi" */
      break;
    default:
      if (wait(&status)) {
       perror("wait"); exit(errno);
      }
  }
  return 0;
}
```

Processus léger

- **`fork()`** crée un nouveau **processus** concurrent.
- Un processus peut aussi se scinder en plusieurs *fils d'exécution (thread* en anglais*)* encapsulés dans le même processus.
- Les *threads* s'exécutent de façon concurrente mais partagent leur mémoire: technique légère pour faire du parallélisme.
- Utilisation d'API comme les threads POSIX.
- Vu en détail l'an prochain

Les threads POSIX

- **`<pthread.h>`**
- **`pthread_create`** crée un nouveau thread correspondant à l'exécution d'une fonction donnée.
- Un thread se termine sur l'exécution de **`pthread_exit()`** ou quand il arrive en fin de fonction.
- Synchronisation (attente) : **`pthread_join()`**.

Exemple de code

```
void* fonction_du_thread(void* donnees)
{
  int* x = (int*)donnees;
  printf("valeur %d\n", *x);
  *x = 4; /* je change la valeur (ici pas de conflit possible donc pas de mutex) */
  /* le code du thread s'arrete ici, equivalent a pthread_exit() */
  return 0;
}

int main(int argc, char* argv [])
{
  int valeur = 0;
  pthread_t t;
  int error = pthread_create(&t, NULL, &fonction_du_thread, &valeur);
  if (!error) {
    printf("le thread a bien ete cree, j'attends qu'il ait termine\n");
    error = pthread_join(t, NULL);
    if (error) {
      fprintf(stderr, "pthread_join : %s\n", strerror(error));
      exit(error);
    }
  printf("valeur doit valoir 4 : %d\n", valeur);
  return 0;
}
```