Noyau d'un système d'exploitation
INF2610

Chapitre 2 : Processus

Département de génie informatique et génie logiciel

POLYTECHNIQUE MONTRÉAL
AFFILIÉE À L'UNIVERSITÉ DE MONTRÉAL

Automne 2016

Chapitre 2 - Processus

- Qu'est ce qu'un processus ?
- États d'un processus
- Hiérarchie des processus
- Processus UNIX-Linux
  - Création de processus
  - Remplacement d'espace d'adressage
  - Attente de la fin d'un processus fils
  - Terminaison de processus
  - Partage de fichiers entre processus

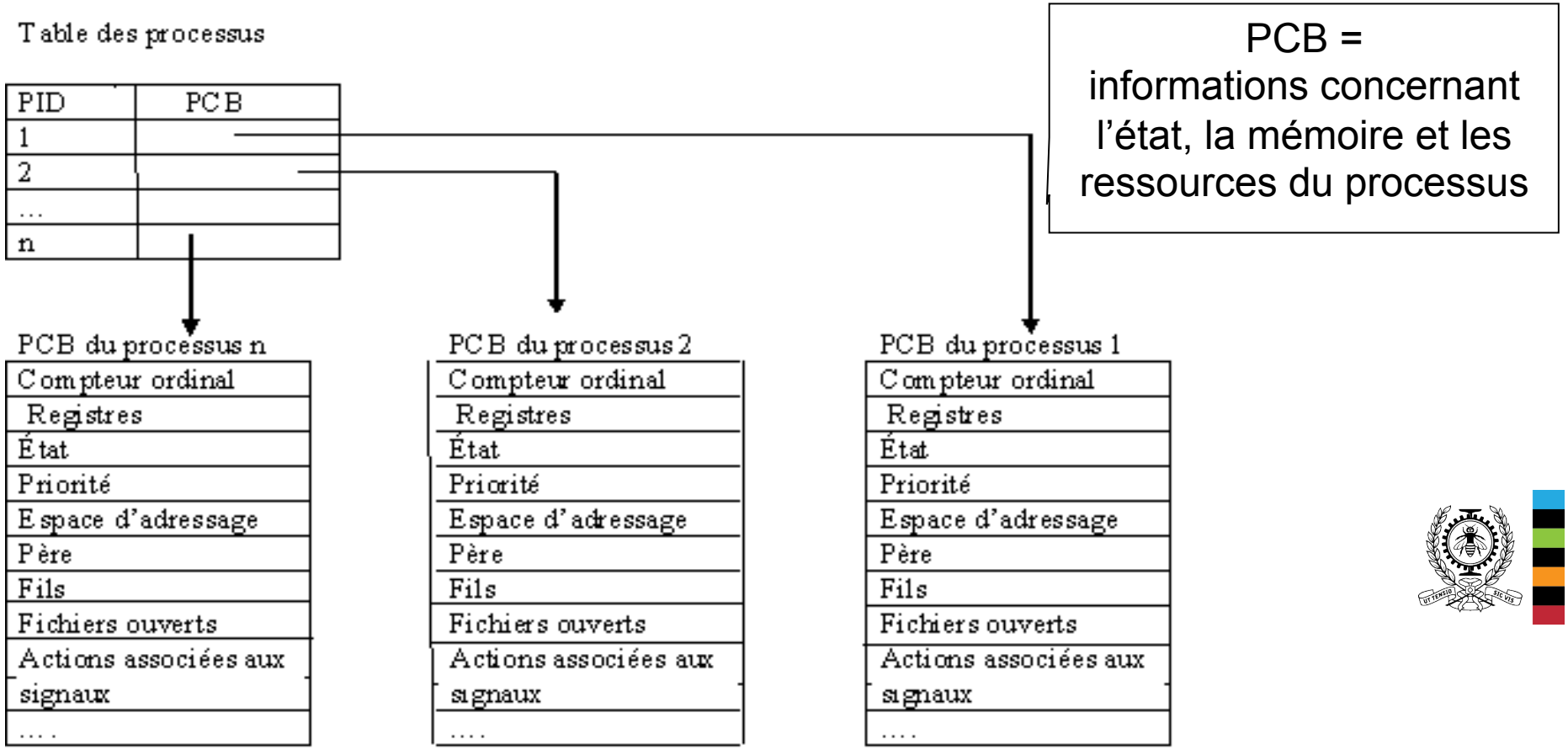
Qu'est ce qu'un processus ?

- Le concept processus est le plus important dans un système d'exploitation. Tout le logiciel d'un ordinateur est organisé en un certain nombre de processus (système et utilisateur).

- Un processus (cas d'un seul fil (thread)) est un programme en cours d'exécution. Il est caractérisé par:
  - Un numéro d'identification unique (PID);
  - Un espace d'adressage (code, données, piles d'exécution);
  - Un état principal (prêt, en cours d'exécution (élu), bloqué, …);
  - Les valeurs des registres lors de la dernière suspension (CO, PSW, Sommet de Pile…);
  - Une priorité;
  - Les ressources allouées (fichiers ouverts, mémoires, périphériques …);
  - Les signaux à capter, à masquer, à ignorer, en attente et les actions associées;
  - Autres informations indiquant le processus père, les processus fils, le groupe, les variables d'environnement, les statistiques et les limites d'utilisation des ressources….

Qu'est ce qu'un processus ? (2)
Table des processus

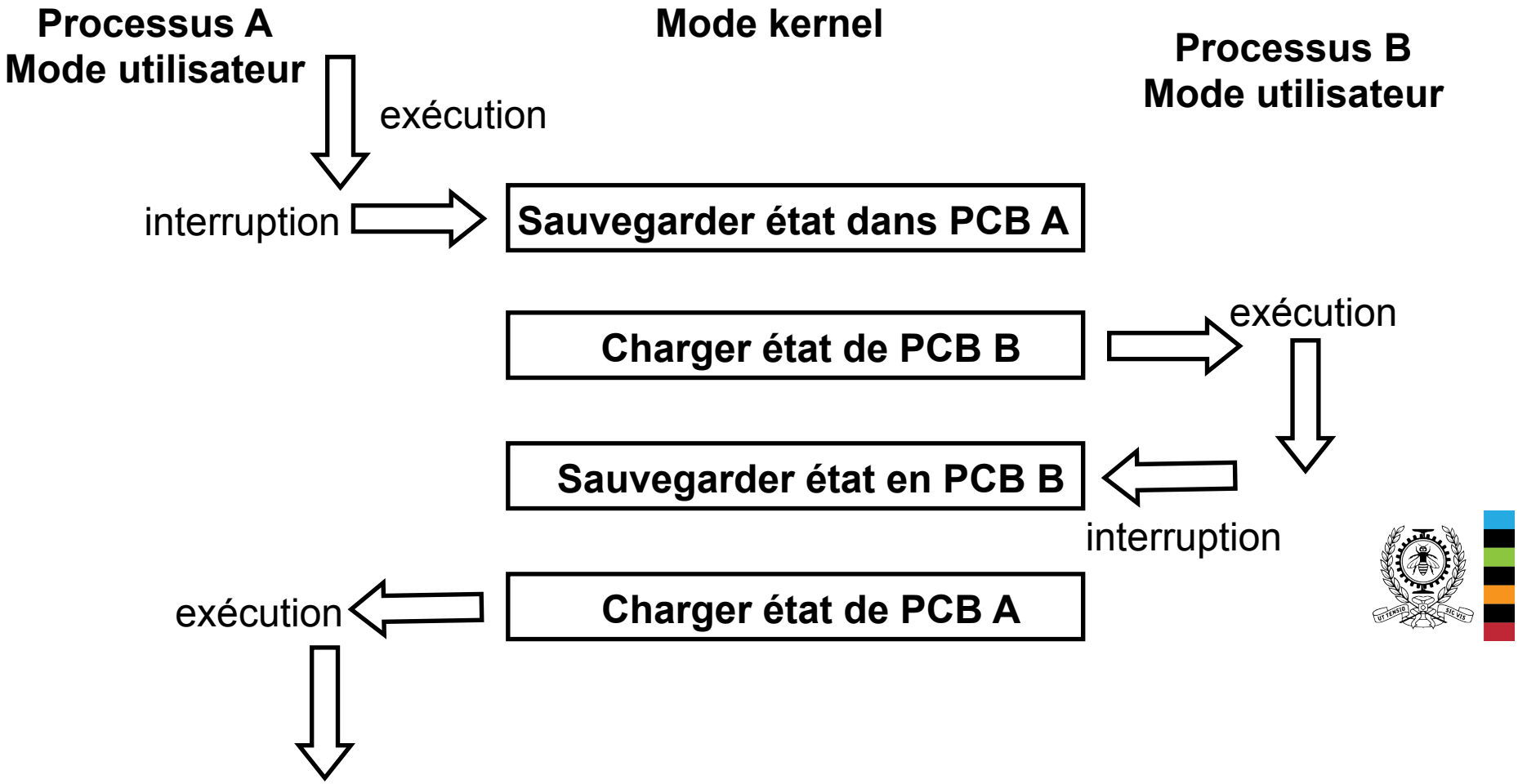
- Le système d'exploitation maintient dans une table appelée «table des processus» les informations sur tous les processus créés (une entrée par processus : Bloc de Contrôle de Processus PCB).



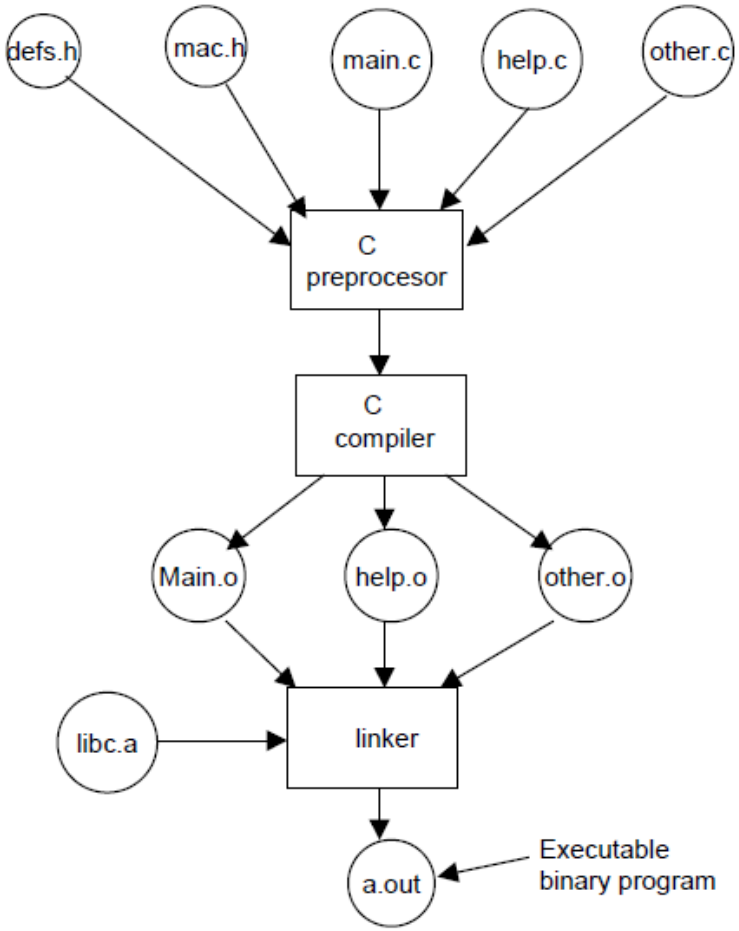
Qu'est ce qu'un processus ? (3)
Table des processus

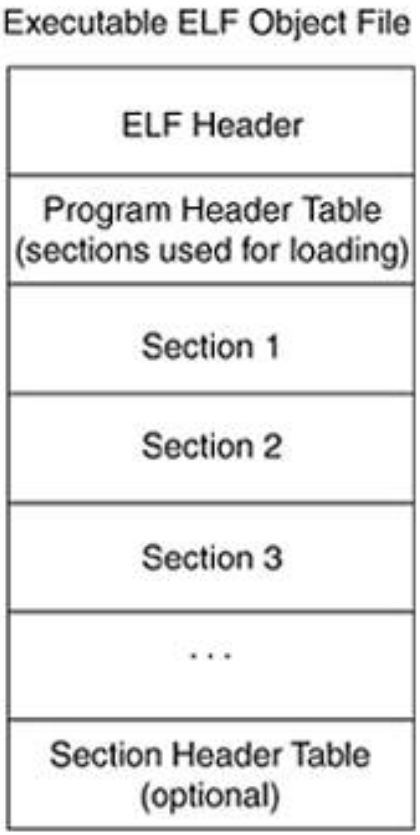
- Dans plusieurs systèmes d'exploitation, le PCB est composé de deux zones :
  - La première contient les informations critiques dont le système a toujours besoin (toujours en mémoire);
  - La deuxième contient les informations utilisées uniquement lorsque le processus est à l'état élu.

Qu'est ce qu'un processus ? (4)
Table des processus : Changement de contexte



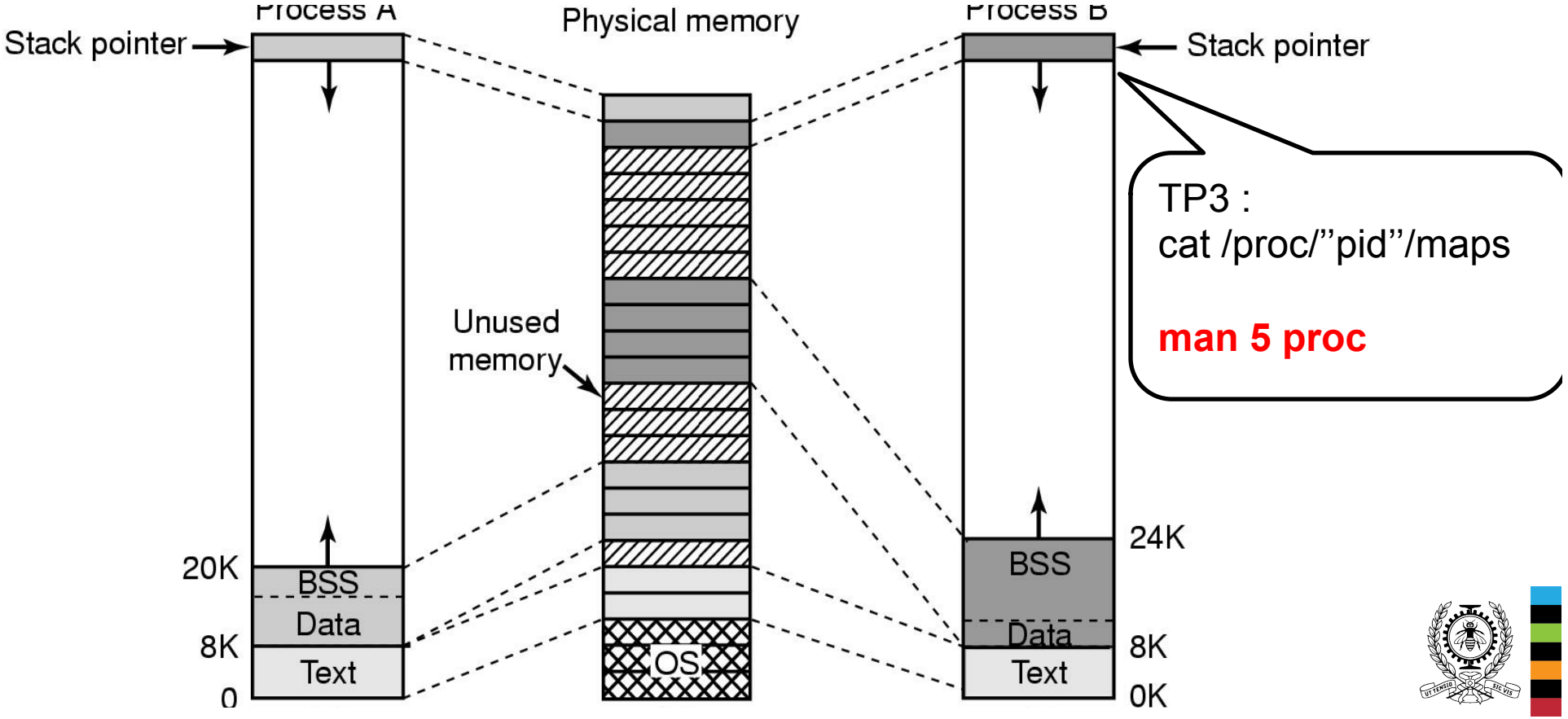
Qu'est ce qu'un processus ? (5)
Espace d'adressage virtuel





http://www.freebsd.org/cgi/man.cgi?query=a.out&sektion=5

Qu'est ce qu'un processus ? (6)
Espace d'adressage virtuel



Mémoires virtuelles et mémoire physique

Qu'est ce qu'un processus ? (7)
Espace d'adressage virtuel

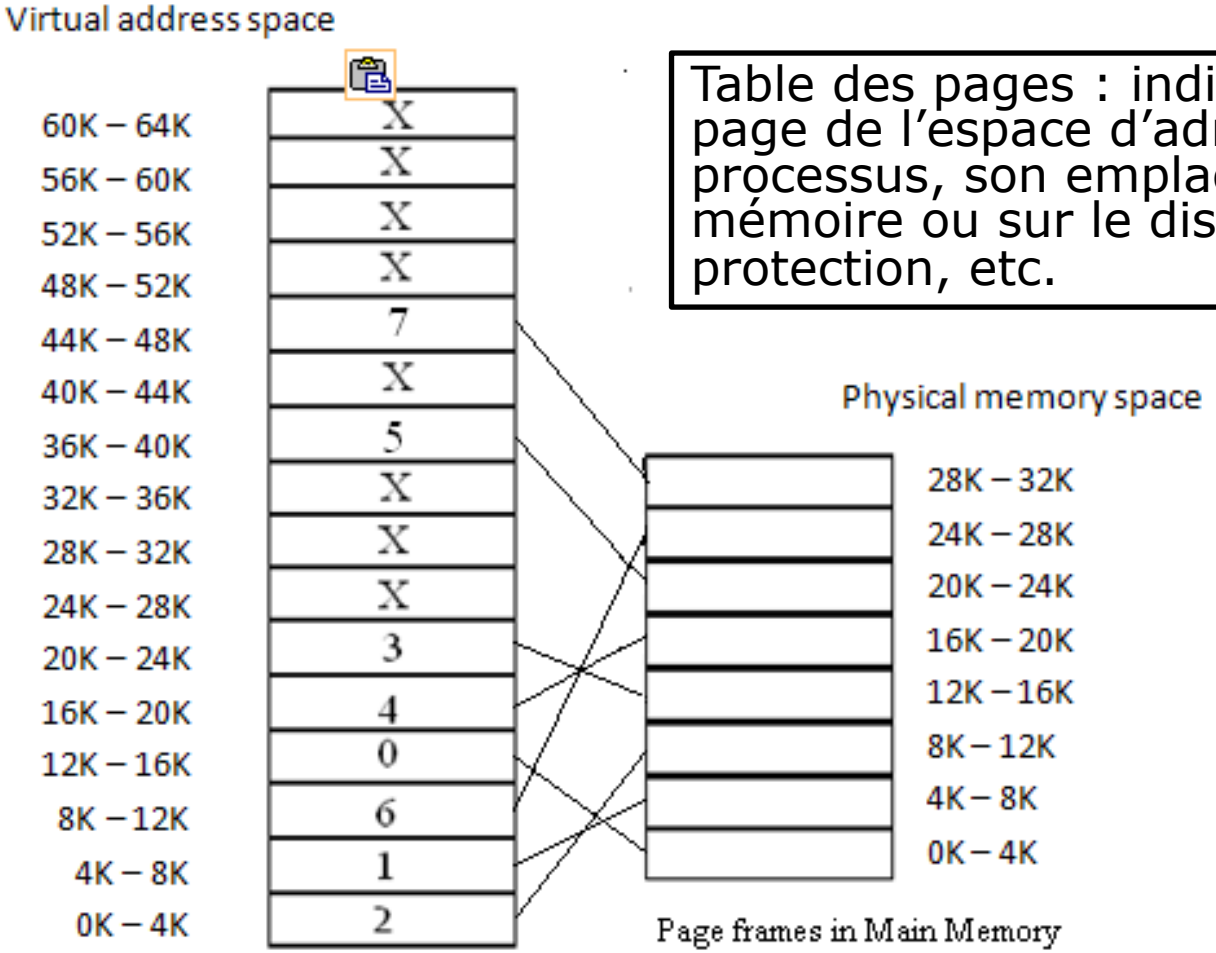
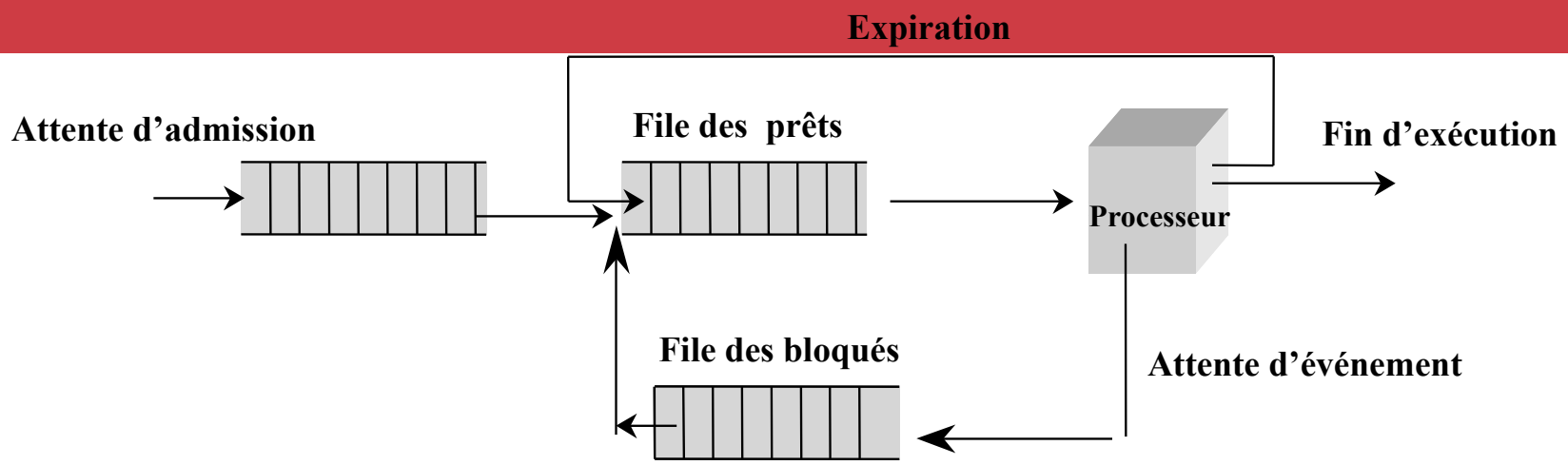
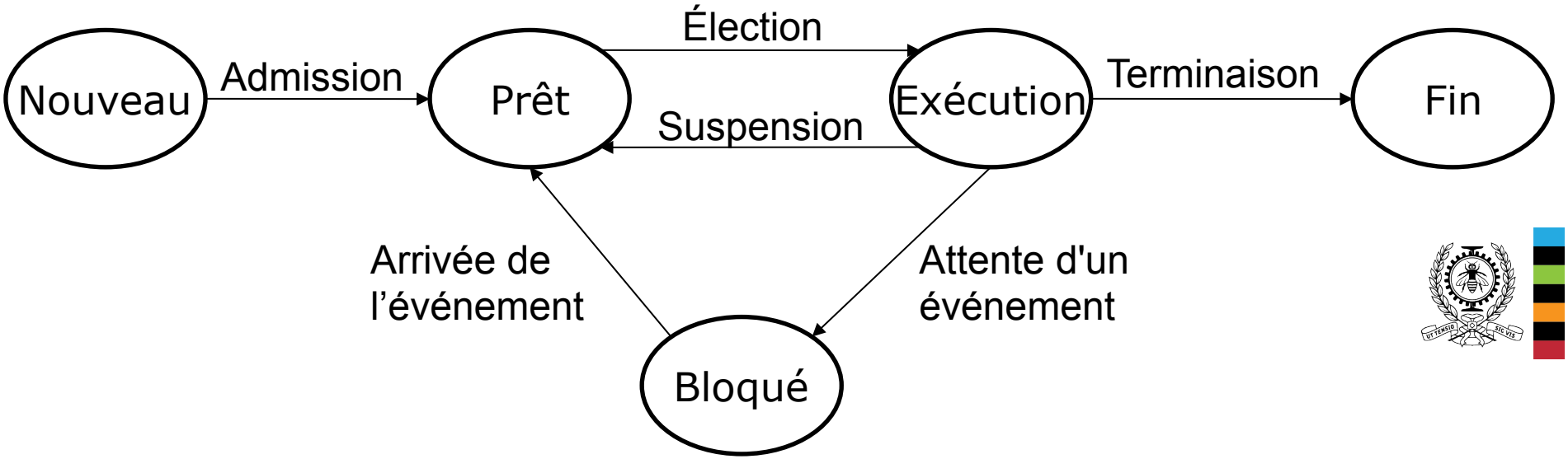


Table des pages : indique pour chaque page de l'espace d'adressage d'un processus, son emplacement en mémoire ou sur le disque, son code de protection, etc.

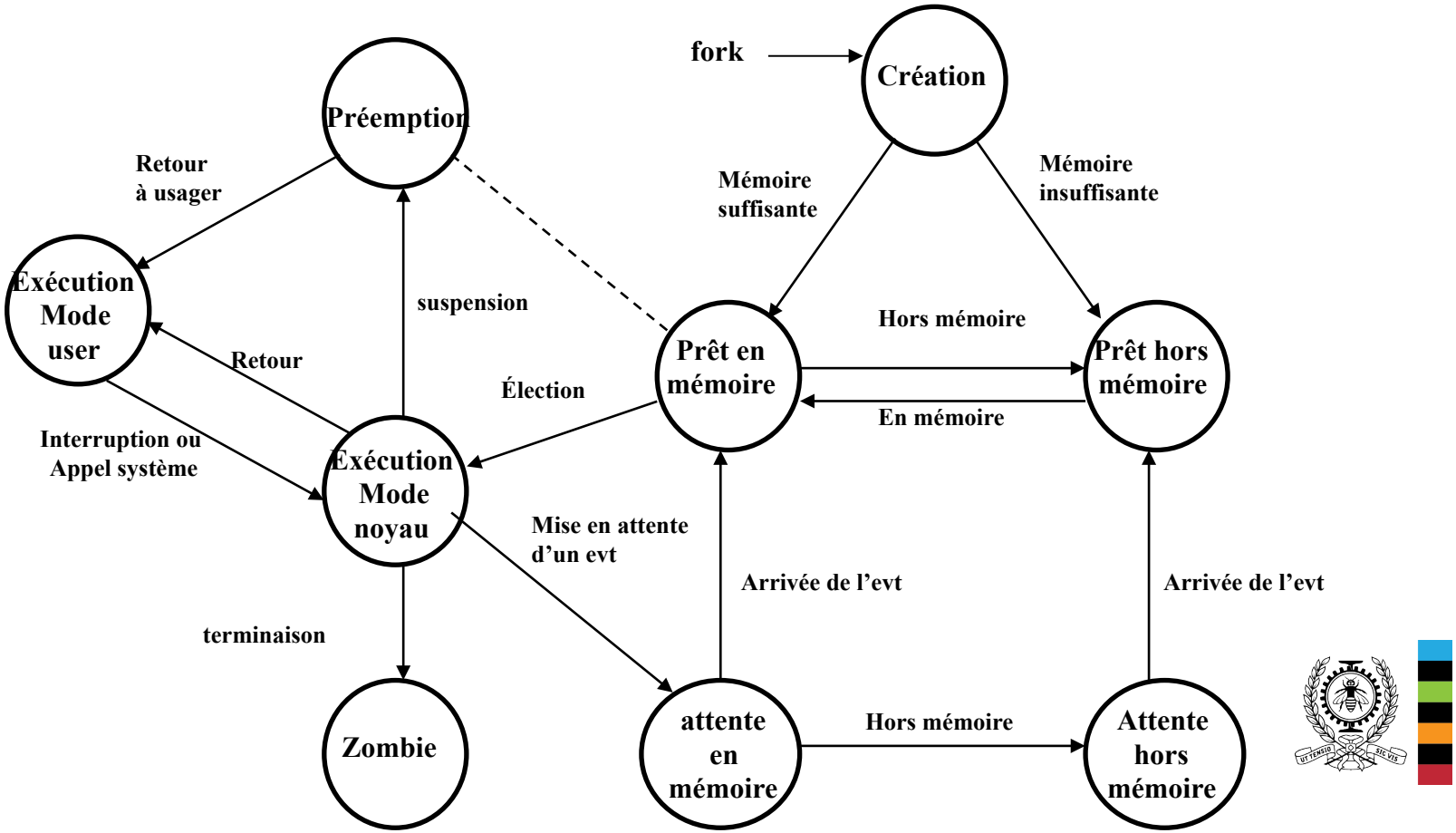
Mémoires virtuelles et mémoire physique

États d'un processus

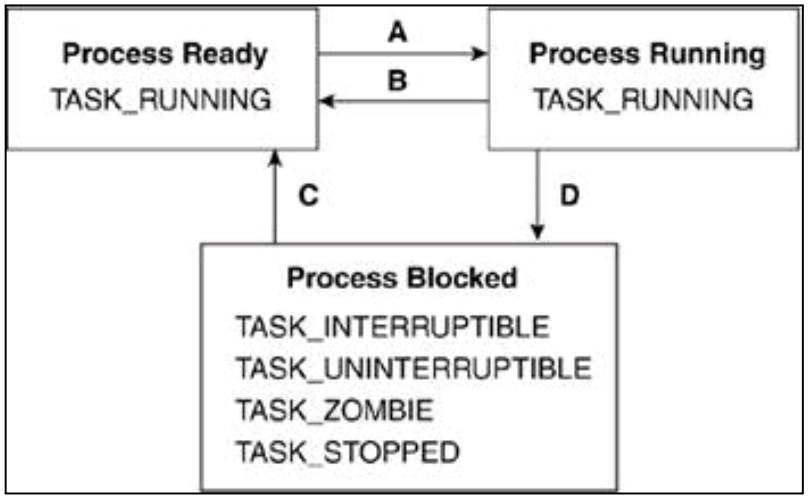


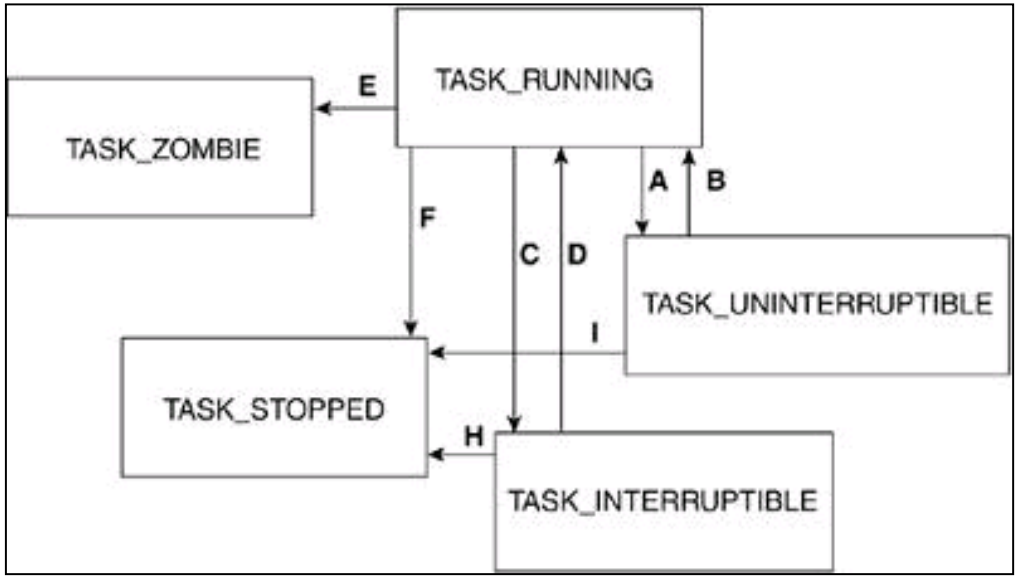


États d'un processus (2) : Unix



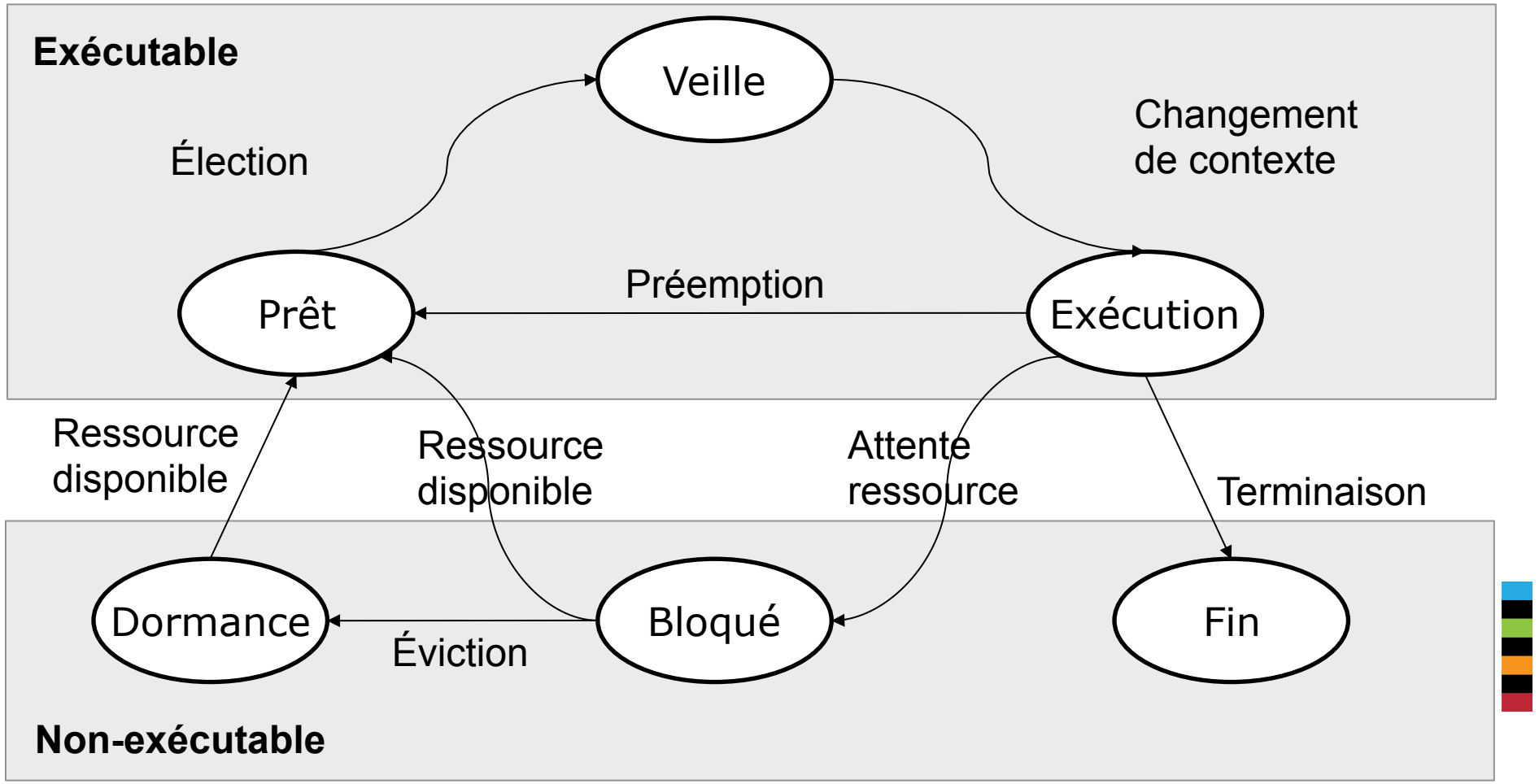
États d'un processus (3) : Linux





États : R → Task_Running
S (sleeping) → Task_Interruptible
D → Task_Uninterruptible,
T (being traced or stopped) → Task_Stopped ,
Z → Task_Zombie.

États d'un processus (4) : Windows



Hiérarchie des processus

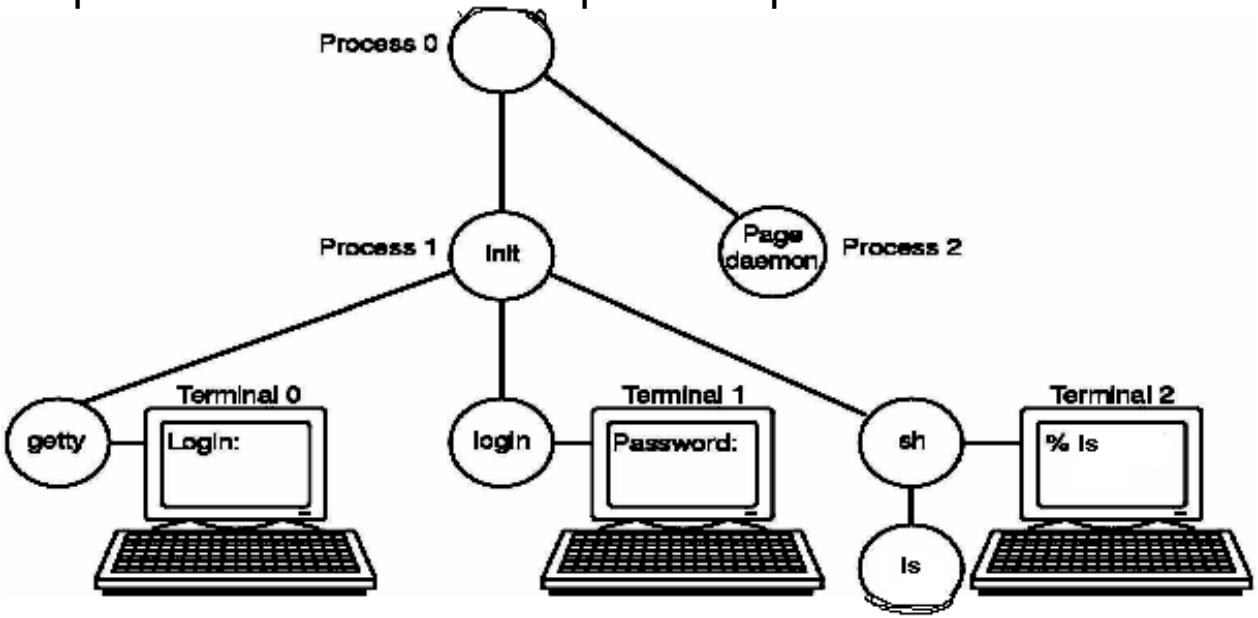
- Le système d'exploitation fournit un ensemble d'appels système qui permettent la création, la destruction, la communication et la synchronisation des processus.
- Les processus sont créés et détruits dynamiquement.
- Un processus peut créer un ou plusieurs processus qui, à leur tour, peuvent en créer d'autres.
- Dans certains systèmes (MS-DOS), lorsqu'un processus crée un processus, l'exécution du processus créateur est suspendue jusqu'à la terminaison du processus créé (exécution séquentielle).
- Dans Windows, les processus créateurs et créés s'exécutent en concurrence et sont de même niveau (exécution asynchrone mais pas de relation hiérarchique explicite gérée par le SE).

Hiérarchie des processus (2) : UNIX

- Le programme amorce charge une partie du système d'exploitation à partir du disque (disquette ou CD) pour lui donner le contrôle. Cette partie détermine les caractéristiques du matériel, effectue un certain nombre d'initialisations et crée le processus 0.

- Le processus 0 réalise d'autres initialisations (ex. le système de fichier) puis crée deux processus : **init** de PID 1 et **démon des pages** de PID 2.

- Ensuite, d'autres processus sont crées à partir du processus init.



Hiérarchie des processus (3) : Windows

- Chaque processus pointe vers le processus parent (processus créateur).
- Si le parent se termine, il n'y a pas d'actualisation de cette information.
- Un processus peut pointer vers un parent inexistant. Cela ne cause aucun problème puisque rien ne dépend de cette information de parenté.
- Windows ne gère de lien qu'avec l'ID du processus parent, pas avec le parent du parent, etc.

Processus UNIX - Linux

- Chaque processus s'exécute de manière asynchrone et a un numéro d'identification unique (PID).
- L'appel système getpid() permet de récupérer le PID du processus : pid_t getpid();
- Chaque processus a un père, à l'exception du premier processus créé (structure arborescente).
- S'il perd son père (se termine), il est tout de suite adopté par le processus **init** de PID 1.
- L'appel système getppid() permet de récupérer le PID de son processus père.
- Un processus fils peut partager certaines ressources (mémoire, fichiers) avec son processus père ou avoir ses propres ressources. Le processus père peut contrôler l'usage des ressources partagées.

Processus UNIX - Linux (2)

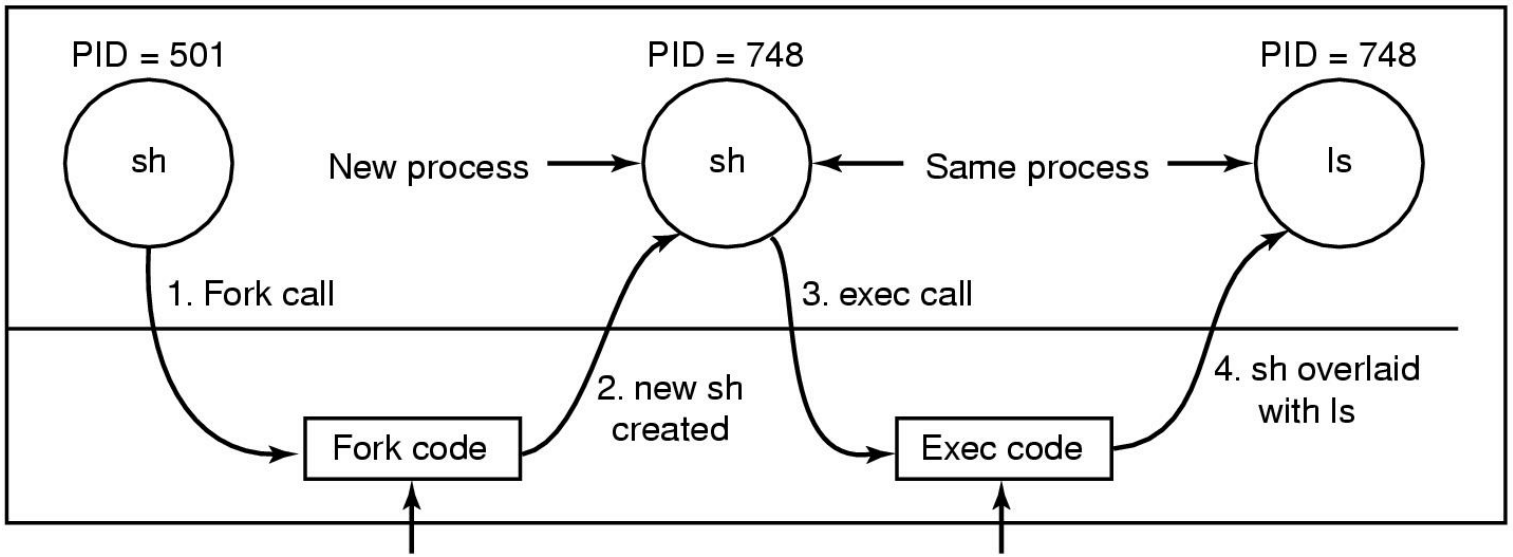
- Le processus père peut avoir une certaine autorité sur ses processus fils. Il peut les suspendre, les détruire (appel système **kill**), attendre leur terminaison (appel système **wait**) mais ne peut pas les renier.
- La création de processus est réalisée par duplication de l'espace d'adressage et de certaines tables du processus créateur (l'appel système **fork**).
- La duplication facilite la création et le partage de ressources. Le fils hérite les résultats des traitements déjà réalisés par le père.
- Un processus peut remplacer son code exécutable par un autre (appel système **exec**).

Processus UNIX –Linux (3) : Shell simplifié

```
 1   int main(int argc, char **argv)
 2   {
 3      char *cmd, *params;
 4      while(1) {
 5         display_prompt();
 6         read_command(&cmd, &params);
 7
 8         int pid = fork();
 9         if (pid < 0) {
10            printf("fork failed\n");
11            continue;
12         }
13         if (pid != 0) {
14            int status;
15            wait(&status);
16         } else {
17            execve(cmd, params, 0);
18         }
19      }
20   }
```

Processus UNIX – Linux (4) : Shell simplifié

Lancement de la commande ls



Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Création de processus : fork

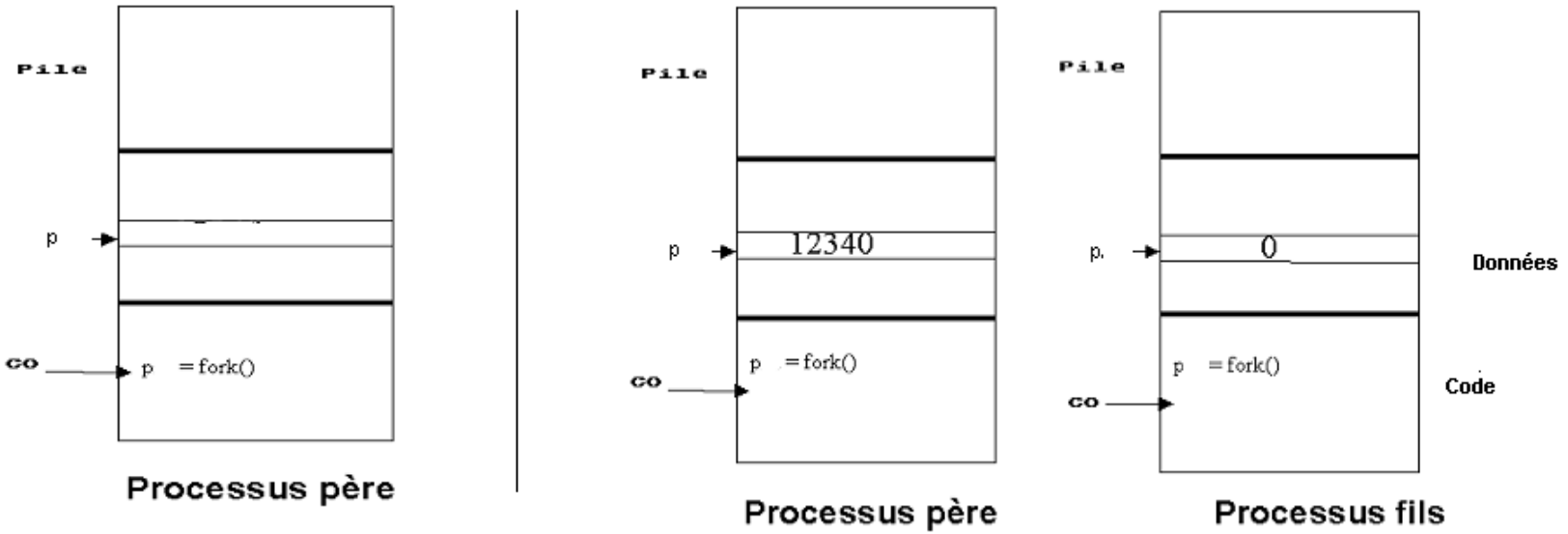
- L'appel système fork :
  - associe un numéro d'identification (le PID du processus);
  - ajoute puis initialise une entrée dans la table des processus (PCB). Certaines entités comme les descripteurs de fichiers ouverts, le répertoire de travail courant, la valeur d'umask, les limites des ressources sont copiées du processus parent;
  - duplique l'espace d'adressage du processus effectuant l'appel à fork (pile+données) → Principe « Copy-on-write ».
  - duplique la table des descripteurs de fichiers….

- La valeur de retour est :
  - 0 pour le processus créé (fils).
  - le PID du processus fils pour le processus créateur (père).
  - négative si la création de processus a échoué (manque d'espace mémoire ou le nombre maximal de créations autorisées est atteint).

Création de processus (2) : fork

- Au retour de la fonction fork, l'exécution des processus père et fils se poursuit, en temps partagé, à partir de l'instruction qui suit fork.

- Le père et le fils ont chacun leur propre image mémoire mais ils partagent certaines ressources telles que les fichiers ouverts par le père avant le fork.



Création de processus (3) : Exemple 1

```
1   // programme tfork.c : appel système fork()
2   #include <sys/types.h>  /* typedef pid_t */
3   #include <unistd.h>                  /* fork() */
4   #include <stdio.h>                   /* pour perror, printf */
5   int a=20;
6   int main(int argc, char *argv) {
7       pid_t x;
8       // création d'un fils
9       switch (x = fork()) {
10      case -1: /* le fork a échoué */
11          perror("le fork a échoué !");
12          break;
13      case 0:   /* seul le processus fils exécute  ce « case »*/
14          printf("ici processus fils, le PID %d.\n ", getpid());
15          a += 10;
16          break;
17      default:  /* seul le processus père exécute cette instruction*/
18          printf("ici processus père, le PID %d.\n", getpid());
19          a += 100;
20      }
21      // les deux processus exécutent ce qui suit
22      printf("Fin du Process %d. avec a = %d\n", getpid(), a);
23      return 0;
24  }
```

Création de processus (4) : Exemple 1

```
jupiter% gcc -o tfork tfork.c
jupiter% tfork
ici processus père, le PID 12339.
ici processus fils, le PID 12340.
Fin du Process 12340 avec a = 30.
Fin du Process 12339 avec a = 120.
```

a du fils

a du père

```
jupiter% tfork
ici processus père, le PID 15301.
Fin du Process 15301 avec a = 120.
ici processus fils, le PID 15302.
Fin du Process 15302 avec a = 30.
jupiter%
```

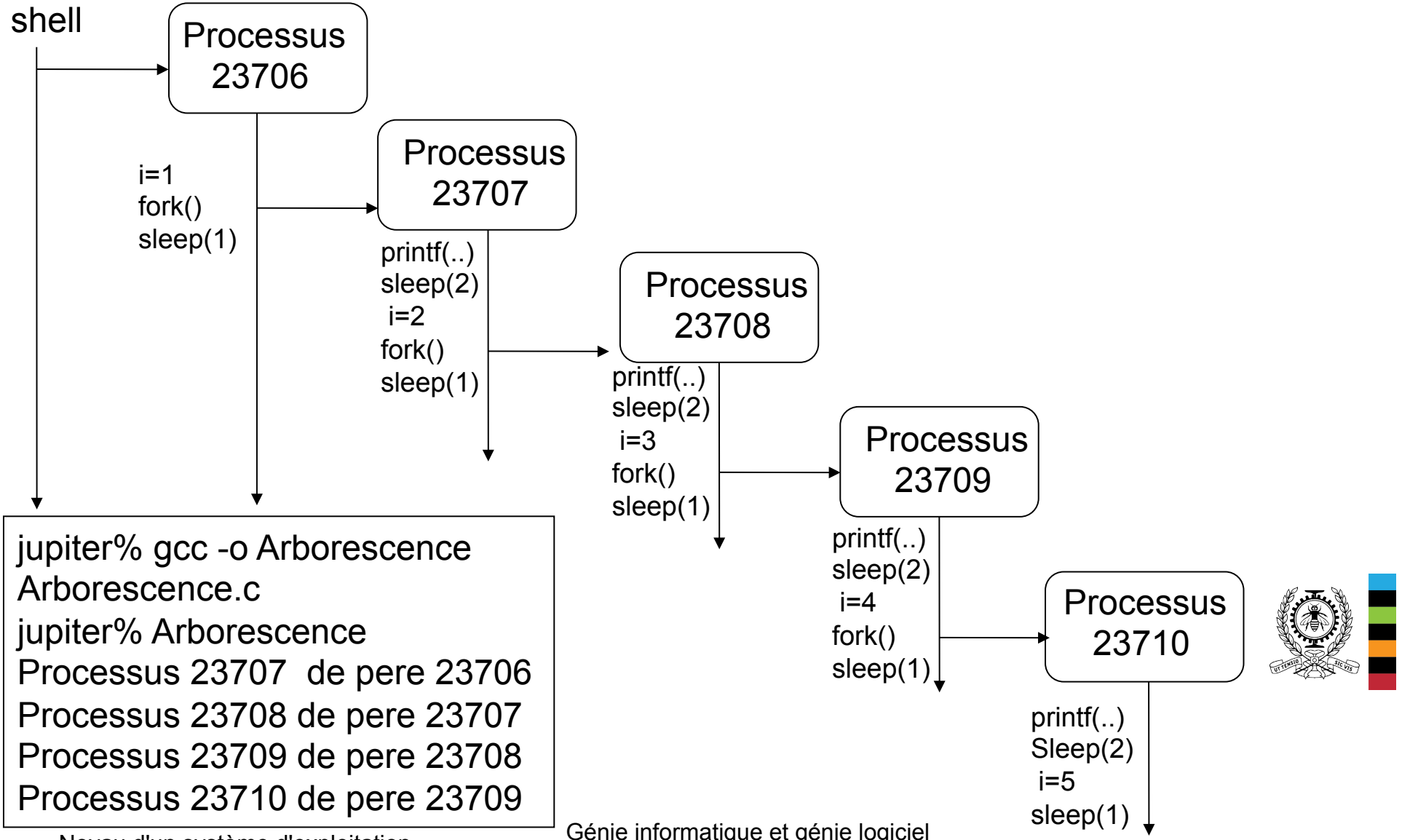
a du père

a du fils

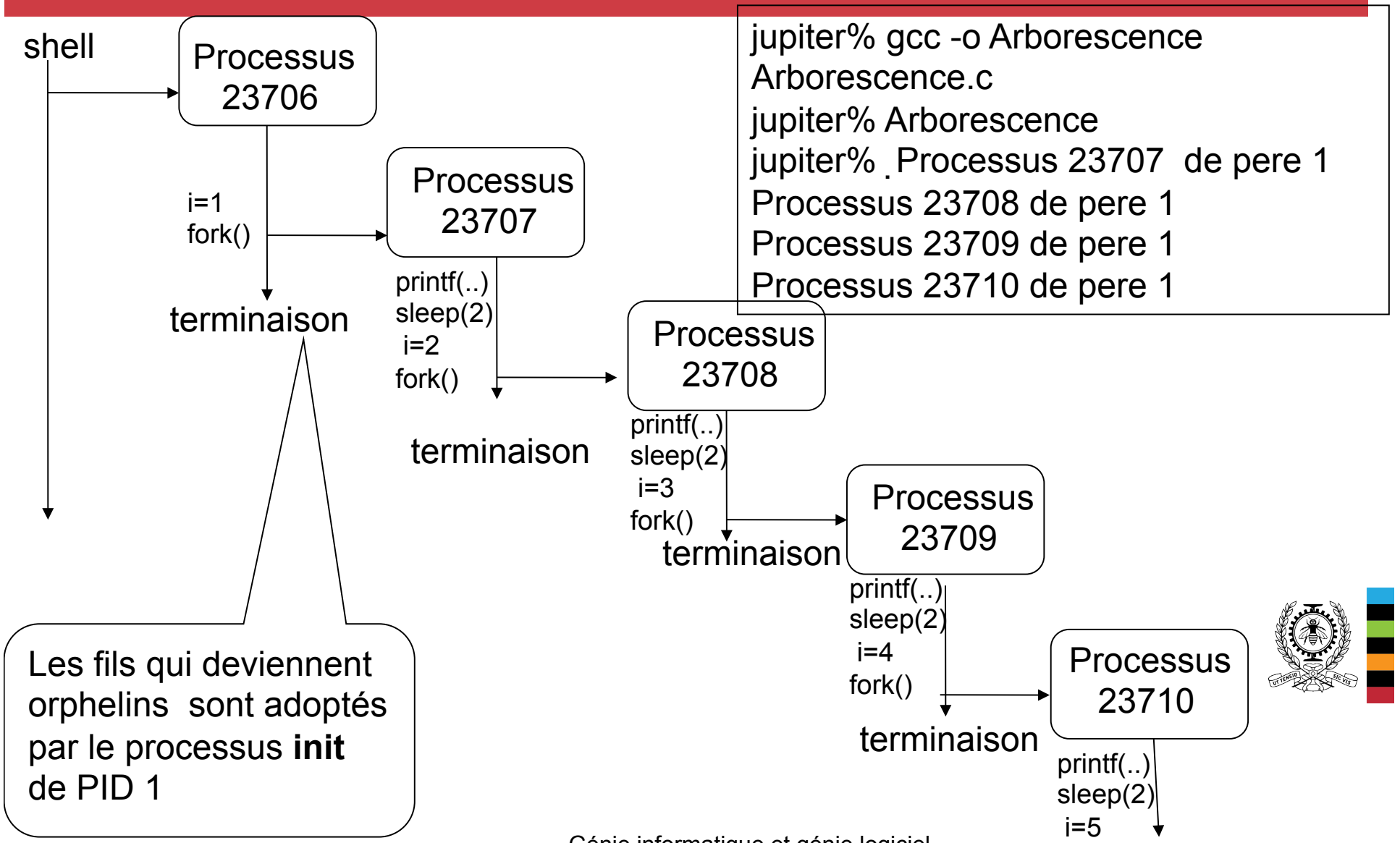
Création de processus (5) : Exemple 2

```
1  // programme tree.c : appel système fork()
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  int main(int argc, char **argv) {
6      pid_t p;
7      int i, n=5;
8      for (i=1; i<n; i++) {
9          p = fork();
10         if (p > 0)
11             break ;
12         printf(" Processus %d  de père %d. \n",  getpid(), getppid());
13         sleep(2);
14     }
15     sleep(1);
16     return 0;
17 }
```

Création de processus (6) : Exemple 2



Création de processus (7) : Exemple 2 sans «sleep»



Remplacement d'espace d'adressage : exec

- Le système UNIX/Linux offre une famille d'appels système **exec** qui permettent à un processus de remplacer son code exécutable par un autre spécifié par **path** ou **file**.

```
int execl(const char *path, const char *argv, …….);
int execlp(const char *file, const char *argv, …….);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- Ces appels permettent d'exécuter de nouveaux programmes.
- Le processus conserve, notamment, son PID, l'espace mémoire alloué, sa table de descripteurs de fichiers et ses liens parentaux (processus fils et père).
- En cas de succès de l'appel système exec, l'exécution de l'ancien code est abandonnée au profit du nouveau.

- En cas d'échec, le processus poursuit l'exécution de son code à partir de l'instruction qui suit l'appel (il n'y a pas eu de remplacement de code).

Remplacement d'espace d'adressage (2) : Exemple 3

```
 1  // programme test_exec.c
 2  #include <unistd.h>
 3  #include <stdio.h>
 4  #include <stdlib.h>
 5  #include <string.h>
 6  #include <errno.h>
 7  int main ()
 8  {          char* arg[] = {"ps", "-f", NULL};
 9             printf("Bonjour\n");
10             execvp("ps", arg);
11             printf("Echec de execvp\n");
12             printf("Erreur %s\n",strerror(errno));
13             return 0;
14  }
```

Attente de la fin d'un processus fils : wait

- Attendre ou vérifier la terminaison d'un de ses fils :
  - pid_t wait (int * status); // Attendre après n'importe lequel des fils
  - pid_t waitpid(int pid, int * status, int options);//attendre pid spécifié

- wait et waitpid retournent :
  - le PID du fils qui vient de se terminer,
  - -1 en cas d'erreur (le processus n'a pas de fils).
  - dans le paramètre **status** des informations qui peuvent être récupérées au moyen de macros telles que :
    - WIFEXITED(status) : fin normale avec exit
    - WIFSIGNALED(status) : tué par un signal
    - WIFSTOPPED(status) : stopé temporairement
    - WEXITSTATUS(status) : valeur de retour du processus fils (exit(valeur))

- waitpid(pid, status, WNOHANG) vérifie seulement la terminaison sans bloquer ; il retourne 0 en cas de non terminaison (pas d'attente).

Attente de la fin d'un processus fils (2) : Exemple 4

```c
// programme parent.c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
int main( ) {
  int p, child, status;
  p = fork();
  if (p == -1)
    return -1;
  if(p > 0) { /* parent */
    printf ("père[%d], fils[%d]\n", getpid(), p);
    if ((child=wait(NULL)) > 0)
        printf("père[%d], Fin du fils[%d]\n", getpid(), child);
  printf("Le père[%d] se termine \n", getpid());
  } else { /* enfant */
    if ((status=execl("/home/user/a.out", "a.out", NULL)) == -1)
      printf("le programme n'existe pas : %d\n",status);
    else
      printf("cette instruction n'est jamais exécutée\n");
  }
  return 0;
}
```

Le père attend la fin du fils

```c
// programme fils.c
#include <stdio.h>
#include <unistd.h>
int main() {
  printf("fils [%d]\n", getpid());
  return 0;
}
```

```
jupiter% gcc fils.c
jupiter% gcc -o parent parent.c
 jupiter% parent
  père[10524],  fils[10525]
  fils [10525]
  père[10524] Fin du fils[10525]
  Le père[10524] se termine
 jupiter%
```

Le fils change de code exécutable

Attente de la fin d'un processus fils (3) : Exemple 5

```
// programme test de execvp : texecvp.c
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
int main (int argc , char * argv[]) {
    if (fork() == 0) { // il s'agit du fils
        // exécute un autre programme
        execvp(argv[1], &argv[1]) ;
        fprintf(stderr, "invalide %s\n ",  argv[1]);
    } else if(wait(NULL) > 0)
        printf("Le père détecte la fin du fils\n");
    return 0;
}
```

```
jupiter% gcc –o texecvp
texecvp.c
jupiter% texecvp date
mercredi,  6 septembre
2000,
16:12:49 EDT
Le père détecte la fin du fils

jupiter% texecvp ugg+kjù
invalide ugg+kjù
Le père détecte la fin du fils

jupiter% gcc -o fils fils.c
jupiter% fils
ici programme fils [18097]
jupiter% texecvp fils
 ici programme fils [18099]
 Le père détecte la fin du
fils
```

Terminaison de processus

- Un processus se termine par une demande d'arrêt volontaire (appel système **exit**) ou par un arrêt forcé provoqué par un autre processus (appel système **kill**) ou une erreur.

  void exit(int vstatus);

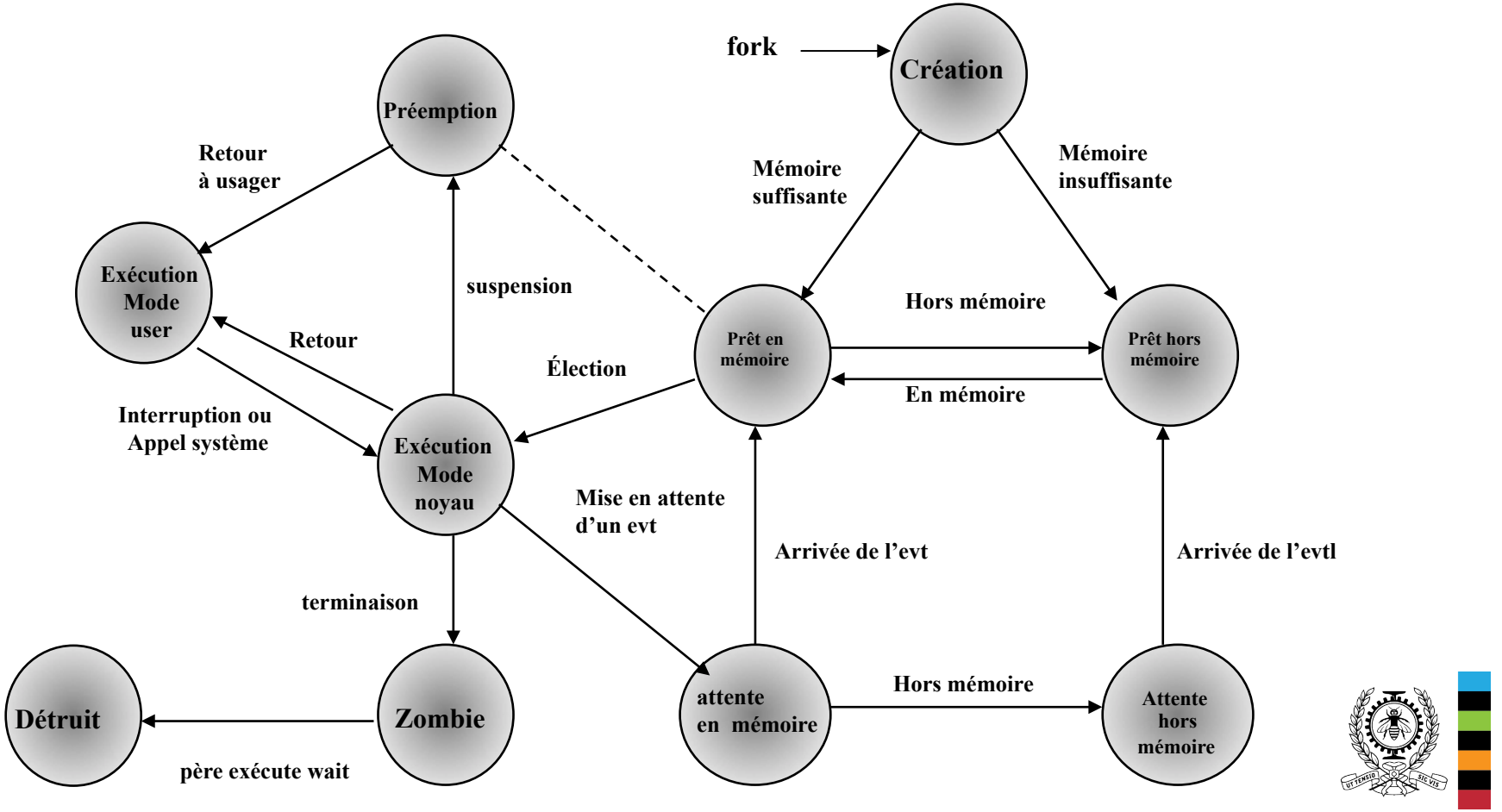
- Lorsqu'un processus fils se termine :
  - son état de terminaison est enregistré dans son PCB,
  - la plupart des autres ressources allouées au processus sont libérées.
  - le processus passe à l'état zombie (<defunct>).

- Son PCB et son PID sont conservés jusqu'à ce que son processus père ait récupéré cet état de terminaison.

- Les appels système wait(status) et waitpid(pid, status, option) permettent au processus père de récupérer, dans le paramètre status, cet état de terminaison.

- Que se passe-t-il si le père meurt avant de récupérer ces informations?

Terminaison de processus (2)



Terminaison de processus (3) : Exemple 6

```
// programme deuxfils.c
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
void fils(int i);
int main() {
    int status;
    if (fork()) { // création du premier fils
        if (fork() == 0) // création du second fils
            fils(2);
    } else
        fils(1);
    if (wait(&status) > 0)
        printf("fin du fils%d\n", status >> 8);
    if (wait(&status) > 0)
        printf("fin du fils%d\n", status >> 8);
    return 0;
}
void fils(int i) {
    sleep(2);
    exit(i);
}
```

```
jupiter% gcc -o deuxfils deuxfils.c
jupiter% deuxfils
fin du fils1
fin du fils2
jupiter% deuxfils
fin du fils2
fin du fils1
```

Terminaison de processus (4) : Exemple 7

```
// programme exec_wait.c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
int main() {
    pid_t p = fork();
    if (p != 0) {
        execlp("./a.out", "./a.out", NULL);
        printf("execlp a échoué\n");
        exit(-1);
    } else {
        sleep(5);
        printf("Je suis le fils\n");
        exit(0);
    }
}
```

Après le fork, le père remplace son espace d'adressage par celui du programme wait_child.c

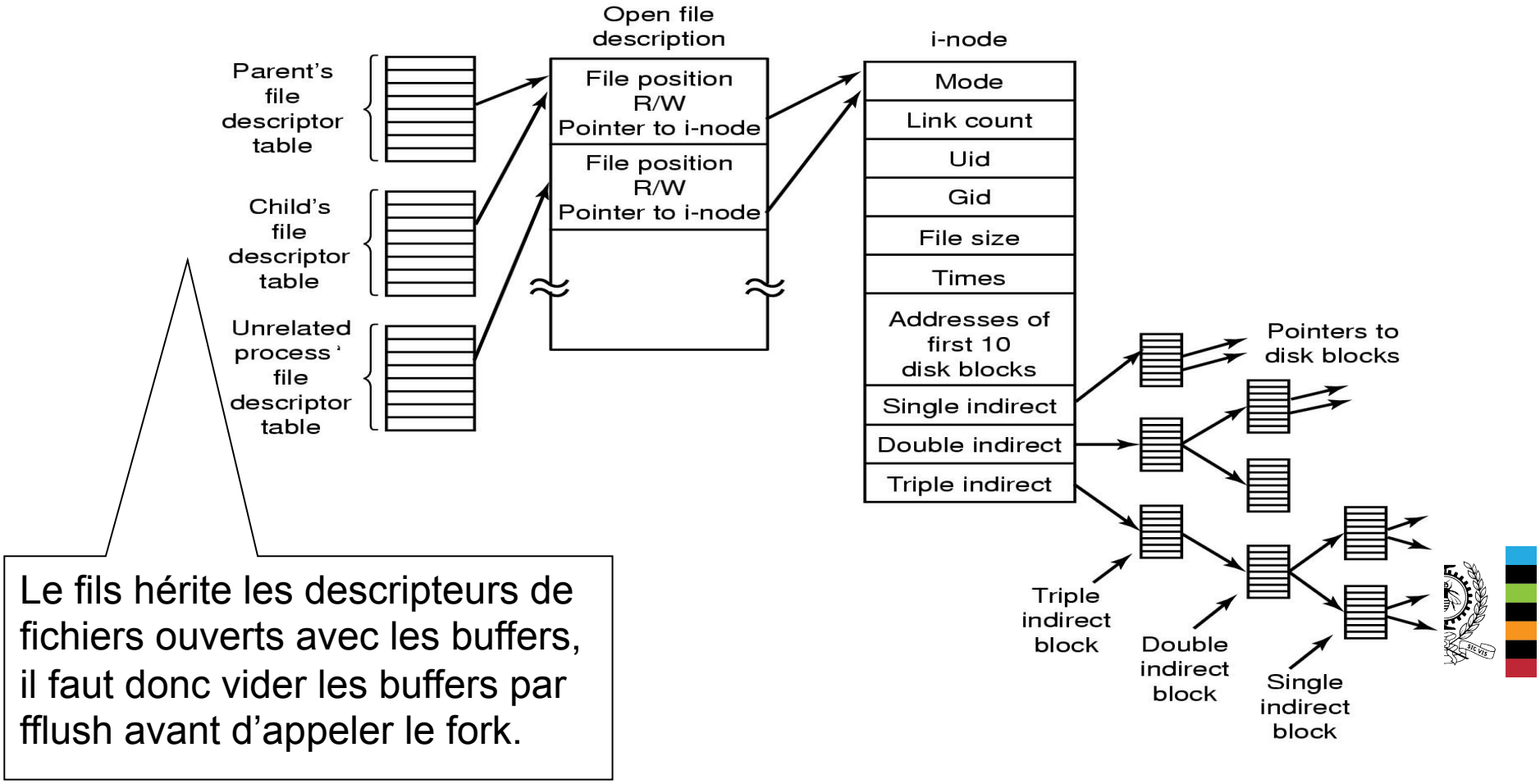
Les liens père /fils sont maintenus

```
// programme wait_child.c
#include <unistd.h>
#include <stdio.h>
#include <stlib.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
    printf("J'attends le fils\n");
    wait(NULL);
    printf("Le fils a terminé \n");
    exit(0);
}
```

```
jupiter$ ./exec_wait
J'attends le fils...
Je suis le fils
Le fils a terminé
```

Partage de fichiers entre processus père et fils

- Le fork duplique la table des descripteurs de fichiers du processus père.



Le fils hérite les descripteurs de fichiers ouverts avec les buffers, il faut donc vider les buffers par fflush avant d'appeler le fork.

Exemple 8 : printf avant fork

```
#include <unistd.h>            /* pour write */
#include <stdio.h>             /* pour printf */
int main( ) {

    printf(" ici 1er printf de %d ", getpid());
    write(1," ici 1er write  ",16);
    printf(" ici 2eme printf de  %d ", getpid());
    fork();
    write(1," ici 2eme write  ", 17);
    printf("end of line printf de %d\n", getpid());
    write(1, " ici 3eme write \n",17);
    return 0;
}
```

```
jupiter$./printfwrite
 ici 1er write   ici 2eme write   ici 1er printf de 11243  ici 2eme printf de  11243 end of line printf
de 11243
 ici 3eme write
ici 2eme write   ici 1er printf de 11243  ici 2eme printf de  11243 end of line printf de
11244
 ici 3eme write
jupiter$
```

Exemple 8 : printf avant fork (2)

```
#include <unistd.h>            /* pour write */
#include <stdio.h>             /* pour printf  et fflush*/
int main( ) {

    printf(" ici 1er printf de %d ", getpid());
    write(1," ici 1er write  ",16);
    printf(" ici 2eme printf de  %d ", getpid());
    fflush(stdout);
    fork();
    write(1," ici 2eme write  ", 17);
    printf("end of line printf de %d\n", getpid());
    write(1, " ici 3eme write \n",17);
    return 0;
}
```

```
jupiter$./printfwrite
 ici 1er write   ici 1er printf de 11258  ici 2eme printf de  11258  ici 2eme write  end of line
printf de 11258
 ici 3eme write
 ici 2eme write  end of line printf de 11259
 ici 3eme write
jupiter$
```

Passage de paramètres à un programme

int main(int argc, const char *argv[]);

int main(int argc, const char *argv[], const char *envp[]);

- argc: nombre de paramètres sur la ligne de commande (y compris le nom de l'exécutable lui-même)
- argv: tableau de chaînes de caractères contenant les paramètres de la ligne de commande.
- envp: tableau de chaînes de caractères contenant les variables d'environnement au moment de l'appel, sous la forme variable=valeur

Passage de paramètres à un programme

Quelles sont les variables d'environnement d'un processus ?

```
#include <stdio.h>
#include <stdlib.h>
// char *chaine1;
int main(int argc, const char *argv[], const char *envp[]) {
    int k;

    printf("Paramètres:\n");
    for (k = 0; k < argc; k++)
        printf("%d: %s\n", k, argv[k]);

    printf("Variables d'environnement:\n");
    for (k = 0; envp[k] != NULL; k++)
        printf("%d: %s\n", k, envp[k]);

    return 0;
}
```

Passage de paramètres à un programme

- **int main(int argc, const char*argv[])**
- **int execv (const char* path, const char* com[])**

  argv ← com

- **int execl (const char* path, const char* com0, const char* com1,…,)**

  argv [0] ← com0 , argv [1] ← com1, ….,

Exercice 1

Donnez l'arborescence de processus créés par ce programme ainsi que l'ordre de l'affichage des messages.

```
// chaine_fork_wait.c
int main()
{   int i, n=3;
      pid_t pid_fils;
      for(i=1; i<n;i++)
      {   fils_pid = fork();
          if (fils_pid > 0)
          {  wait(NULL);
               break;
          }
      }
      printf("Processus %d de pere %d\n", getpid(), getppid());
      return 0;
}
```

Exercice 2

Donnez, sous forme d'un arbre, les différents ordres possibles d'affichage de messages (chaque chemin de l'arbre correspond à un ordre possible).

```
int main()
{   printf("message0\n");
      if (fork())
      {   printf("message1\n");
            if (fork())
                  printf("message2\n");
            else exit(0);
      } else printf("message3\n");
      return 0;
}
```

Ajouter une ligne de code pour forcer l'ordre d'affichage suivant :
message0; message3; message1;message2

Lectures suggérées

- Notes de cours: Chapitre 3
(http://www.groupes.polymtl.ca/inf2610/documentation/notes/chap3.pdf)

- Chapitre 3 (pp 41- 50)

M. Mitchell, J. Oldham, A. Samuel - Programmation Avancée sous Linux- *Traduction : Sébastien Le Ray* (2001) **Livre disponible dans le dossier Slides Automne 2016 du site moodle du cours.**