OS Réseaux et Programmation Système - C2

Rabie Ben Atitallah, LAMIH

rabie.benatitallah@univ-valenciennes.fr

Contributeurs :

Mikael Desertot

Processus

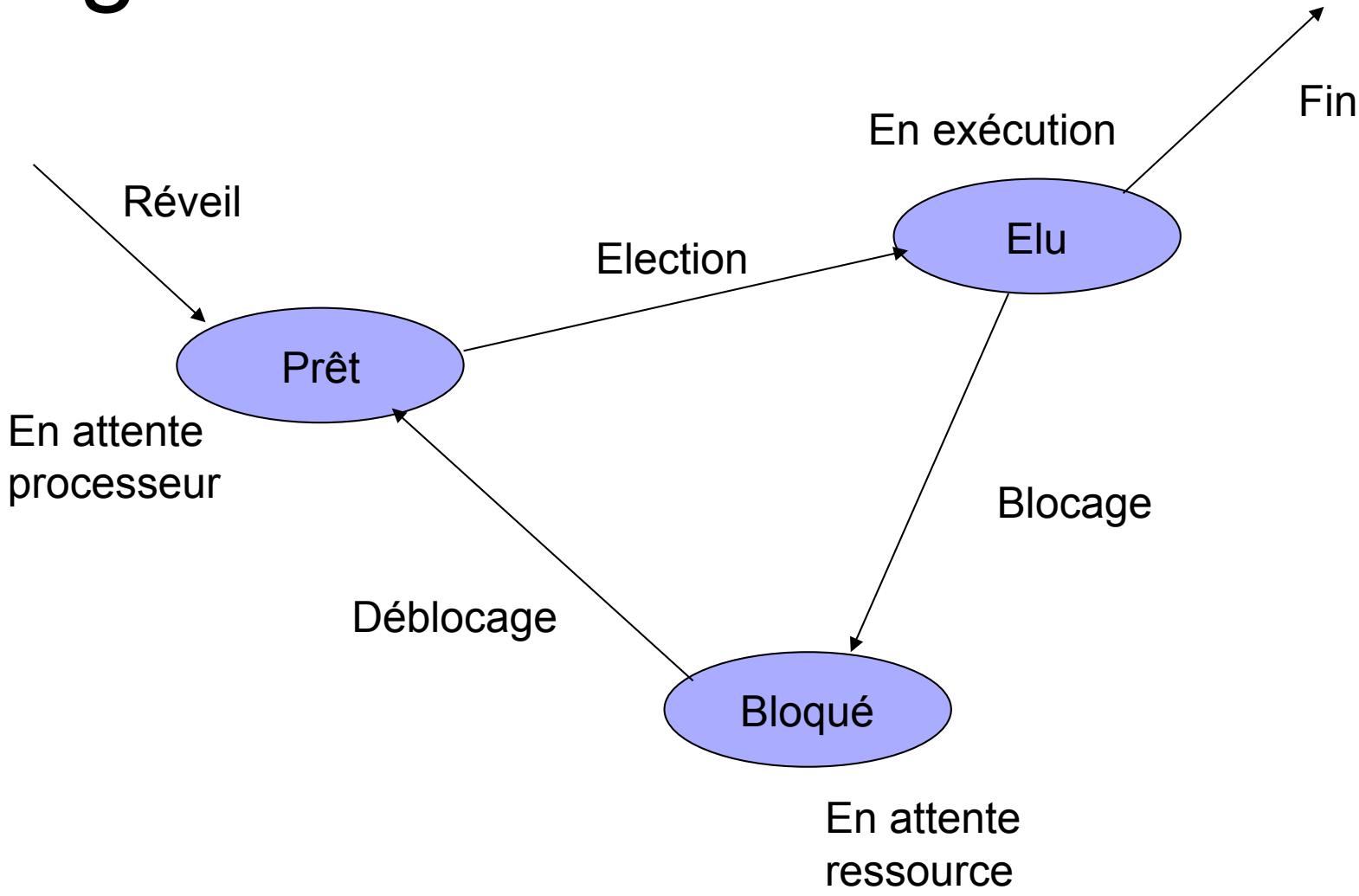
La notion de processus

- Le processus est l'entité d'exécution dans le système UNIX
  - Toute activité dans UNIX correspond à un processus
  - Un processus est l'image de l'état du processeur et de la mémoire au cours de lexécution d'un programme.
- Deux types de processus dans UNIX
  - Processus système
  - Processus utilisateur : correspondent à l'exécution d'une commande ou d'une application
- Identification
  - Chaque processus est identifié par un numéro :
    - le PID (Process IDentifier)
  - Un processus a une priorité en fonction de laquelle il est associé à l'UC

Etats d’un processus

- Lors de son exécution un processus est caractérisé par un état
  - Obtention du processeur et exécution -> élu
  - Attente d’entrée-sortie -> bloqué
  - En attente du processeur -> prêt

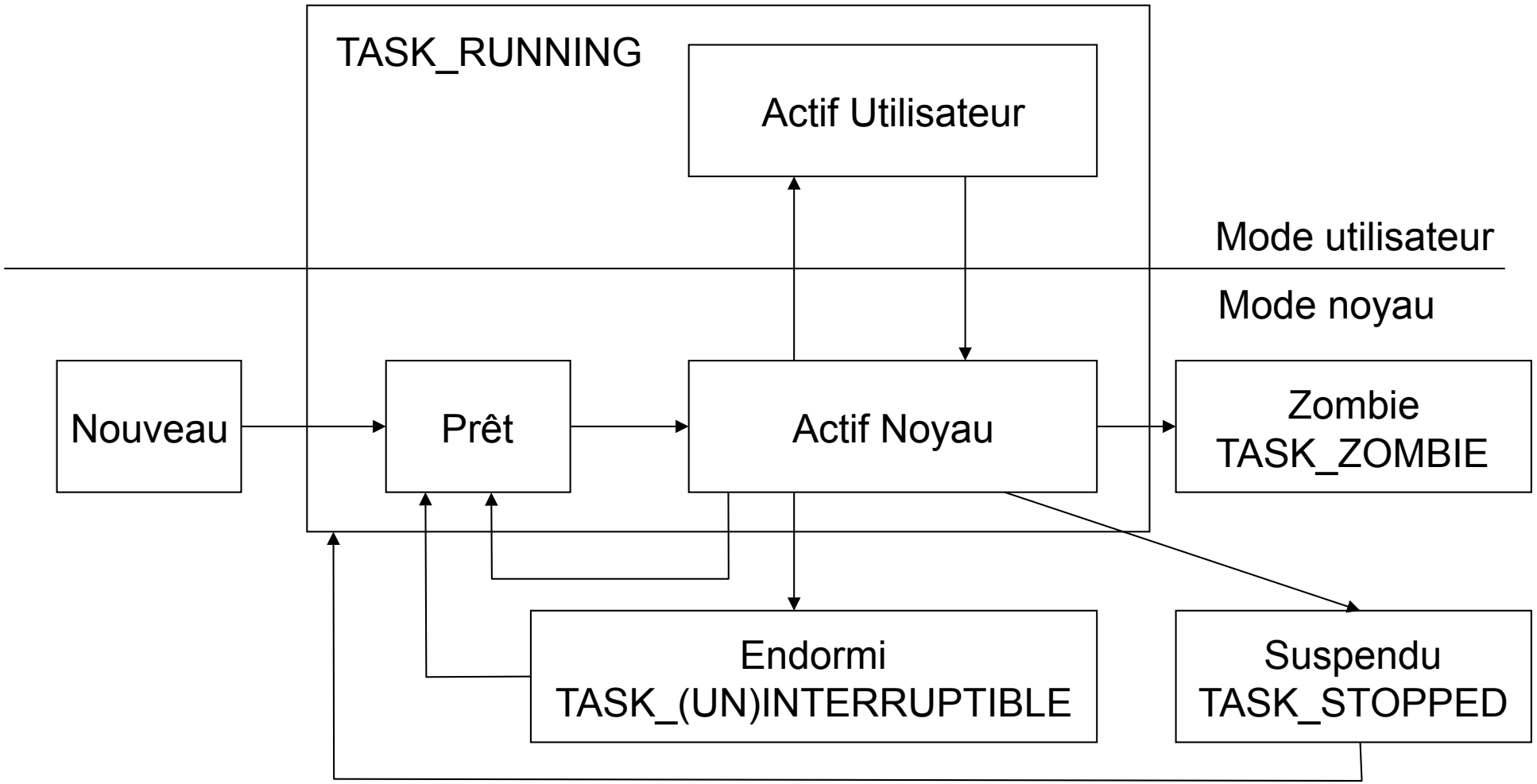
Diagramme d'états



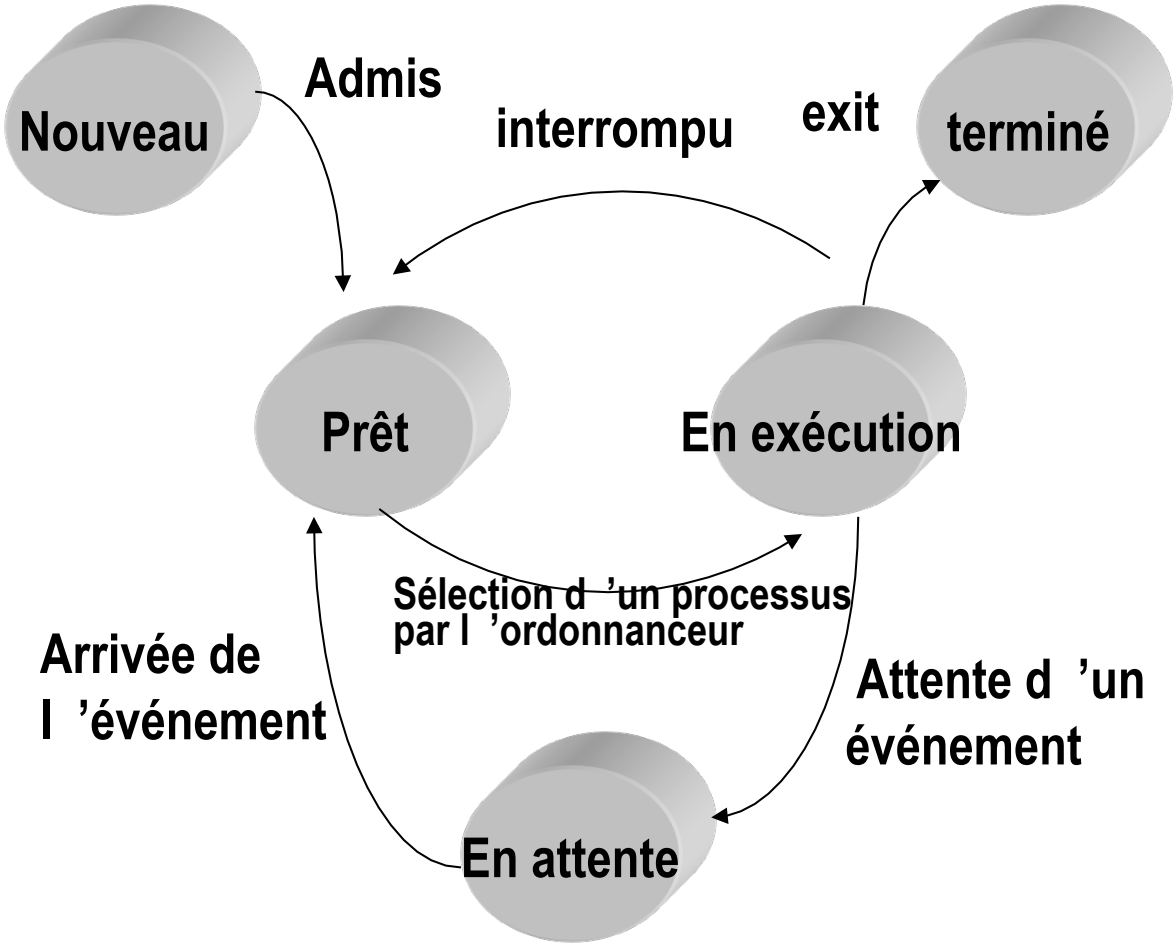
Etats d'un processus Linux (1)

- Evolue entre deux modes :
  - Utilisateur
  - Noyau
- Différents états
  - TASK_RUNNING
  - TASK_INTERRUPTIBLE / TASK_UNINTERRUPTIBLE
  - TASK_ZOMBIE
  - TASK_STOPPED

Etats d’un processus Linux (2)



États d'un processus (2)



Bloc de contrôle du processus (1)

- Le chargeur monte en mémoire centrale :
  - le code et les données du programme à exécuter
- Le système d'exploitation crée le bloc de control (BDC) du processus
- Référencement des BDCs dans une table des processus

Bloc de contrôle du processus (2)

Identificateur de processus
Etat du processus
Compteur ordinal Contexte pour reprise (registre, pointeurs, piles, …)
Chaînage selon les files de l'ordonnanceur
Informations mémoire
Ressources utilisées (fichiers ouverts, entrées-sorties)
Informations de comptabilisation (statistiques)

Création d’un processus

- La création d'un processus se fait par dédoublement :
  - Un processus est toujours créé par un autre processus par le mécanisme de fourche "fork".
    - On appelle le processus créateur processus père
    - Le processus créé est appelé processus fils .

La primitive fork()

- Création dynamique d'un nouveau processus
- S'exécute de manière concurrente
- Fils = duplication du père

```
#include <unistd.h>

pid_t fork (void);
```

La primitive fork() (2)

- La valeur de retour de la fonction FORK est
  - 0 pour le processus fils
  - égale au PID du fils chez le processus père.
  - -1 en cas d'erreur

- Le processus fils peut exécuter un nouveau code à l'aide des primitives de recouvrement EXEC.

La commande fork() (3)

- Le processus fils hérite de son père :
  - Le code à exécuter
  - Les données (**duplication**)
  - La priorité

- Ecrire un programme en C qui définit quel processus est en cours d'exécution

Solution

```
#include<stdio.h>
#include<unistd.h>

main() {
   int pid;
   pid=fork();
   if (pid==0) printf("je suis le
fils");
   else printf("je suis le père");
}
```

Identification de processus

- #include <unistd.h>
- PID du processus
  - int pid = getpid();
- PID du processus père
  - int ppid = getppid();
- UID identifiant utilisateur
  - int uid = getuid(void);
- GID groupe d'appartenance du processus
  - int gid = getgid();

Exemple

```
#include<stdio.h>
#include<unistd.h>

main() {
    int pid;
    pid=fork();
    if (pid==0){
          printf("je suis le fils, mon pid est %d
\n",getpid());
          printf("pid de mon père, %d\n", getppid());
    } else {
          printf("je suis le père, mon pid est %d
\n",getpid())
          printf("pid de mon fils, %d\n", pid);
    }
}
```

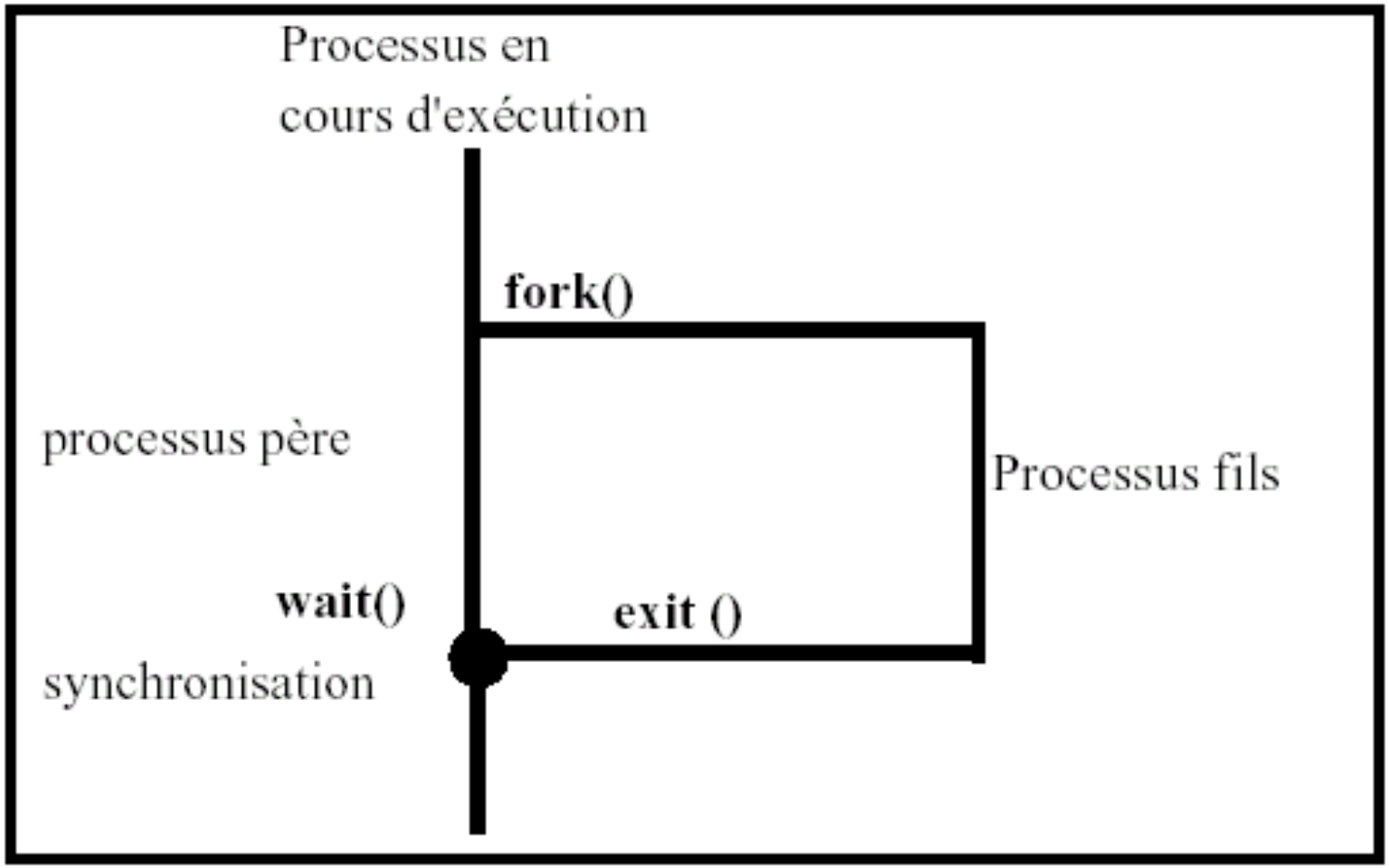
Terminaison processus

- Lors de la fin de l'exécution de son code
- Appel à la primitive exit()
  - #include<stdlib.h>
  - void exit(int status);
- Désallocation des ressources

Synchronisation de processus

- wait()
  - Permet au processus d'attendre la terminaison d'un de ses fils
- waitpid()
  - Permet au processus appelant d'attendre de manière sélective la terminaison d'un de ses fils
- Mise en sommeil d'un processus
  - sleep(n) suspend l'exécution du processus appelant pour une durée de n secondes.
- Exercice : ordonnancer 2 processus : le processus fils doit s'exécuter avant le processus père

Synchronisation de processus (2)



Un petit exemple…

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

main() {            # Le fils s'exécute avant le père
     int pid;
     pid=fork();
     if (pid==0) {
            printf("je suis le fils");
            exit(0);
     }
     else {
            wait();
            printf("je suis le père");
     }
}
```

Un deuxième exemple…

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

main() {
  int status;
  int i = 0;
  int pid;

  pid=fork();
  if(pid==0) {
    printf("\nje suis le fils...");
    printf("\nChez moi, i=%d",i);
    i++;
    printf("\nMaintenant, chez moi, i=%d\n",i);
  }
  else {
    wait(&status);
    printf("\nje suis le pere...");
    printf("\nChez moi, i=%d\n",i);
  }
}
```

Exécutons notre deuxième exemple…

```
je suis le fils...
Chez moi, i=0
Maintenant, chez moi, i=1

je suis le pere...
Chez moi, i=0
```

- Attention, les données sont dupliquées !!!

Interprétation de "status"

- Macros définies dans <sys/wait.h>
  - WIFEEXITED(status)
    - Vrai si le fils s'est terminé par la commande exit()
  - WEXITSTATUS(status)
    - Code passé au fils lors de sa terminaison
  - WIFSIGNALED(status)
    - Vrai si le fils s'est terminé à cause d'un signal
  - WIFSTOPPED(status)
    - Vrai si le fils est stoppé temporairement

Exemple

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

main() {
  int status;
  int i = 0;
  int pid;

  pid=fork();
  if(pid==0) {
    printf("\nje suis le fils...");
    printf("\nChez moi, i=%d",i);
    i++;
    printf("\nMaintenant, chez moi, i=%d\n",i);
  }
  else {
    wait(&status);
    printf("\nje suis le pere...");
    printf("\nChez moi, i=%d\n",i);
    if (WIFEEXITED(status))
      printf("\nle code de retour de mon fils est : %d", WEXITSTATUS(status));
  }
}
```

Primitive exec (1)

- Permet de charger en mémoire un nouveau code exécutable
- Ecrase le code hérité au moment de la création

Primitive exec (2)

- int execl(const char* ref, const char*arg, …, NULL)
  - ref = chemin d'un argument depuis le répertoire courant
- int execlp(const char* ref, const char*arg, …, NULL)
  - ref = chemin d'un argument depuis le PATH
- int execle(const char* ref, const char*arg, …, const char* envp[])
  - ref = chemin d'un argument depuis le répertoire courant
  - envp = tableau de spécification des variables d'environnement
- int execv(const char* ref, const char*arg[])
  - ref = chemin d'un argument depuis le répertoire courant
- int execvp(const char* ref, const char*arg[])
  - ref = chemin d'un argument depuis le répertoire courant
- int execve(const char* ref, const char*arg[], const char* envp[])
  - ref = chemin d'un argument depuis le répertoire courant
  - envp = tableau de spécification des variables d'environnement

Exemple

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

main() {
     int pid;
     pid=fork();
     if (pid==0) {
            printf("je suis le fils");
            execlp("ls","ls","-l","/",NULL);
     }
     else {
            printf("je suis le père");
            wait();
     }
}
```

Threads (processus léger)

- Extension du modèles de processus
- Admettre plusieurs fils d'exécution indépendant dans un même espace d'adressage
- Allègement des opérations de commutation de contexte
- Pas de duplication d'espace d'adressage

Processus lourd et processus léger

Processus lourd

Processus léger

Niveau utilisateur

- Le noyau ignore les threads et ordonnance des processus classiques
- Une bibliothèque système gère l'interface avec le noyau

- Inconvénient
  - Au sein d'un même processus, un thread peu monopoliser le processeur a lui seul.

Niveau noyau

- Gère chacun des threads de manière indépendante
- Pas de blocage de processus si un de ses threads est bloqué
- Commutation de contexte un peu plus couteuse qu'au niveau utilisateur

Primitives de gestion

- Très semblables à celles permettant la gestion de processus.
- Threads identifiés de manière unique
- La primitive pthread_self()
  - Permet à un processus de connaître son identifiant

Création d'un thread

- int pt thread_create (pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);

- *thread = id du thread
- *attr = attribut spécifié au thread
- (*start_routine) = appel à exécuté
- *arg = arguments de l'appel

Terminaison d'un thread

- pthread_exit(void *ret)
  - Met fin au thread qui l'exécute
- Retourne le paramètre "ret"
- Récupération de la valeur de retour
  - pthread_join(pthread_t thread, void **retour)
  - Bloque le processus appelant

Attributs d’un thread

- Adresse de départ et la pile associée
- Politique d'ordonnancement associée
- Priorité associée
- Son attachement ou détachement

Exemple

```
#include <stdio.h>
#include <pthread.h>

pthread_t pthread_id[3];

void f_thread(int i) {
      printf("je suis le %d eme thread d'identité %d.%d\n", i, getpid(),
      pthread_self());
}

main() {
      int i;
      pid=fork();
      for (i=0; i < 3; i++)
                pthread_create(pthread_id[i], NULL, f_thread, i);
      printf("je suis le thread initial %d.%d\n", getpid(),
      pthread_self());
      pthread_join();
}
```

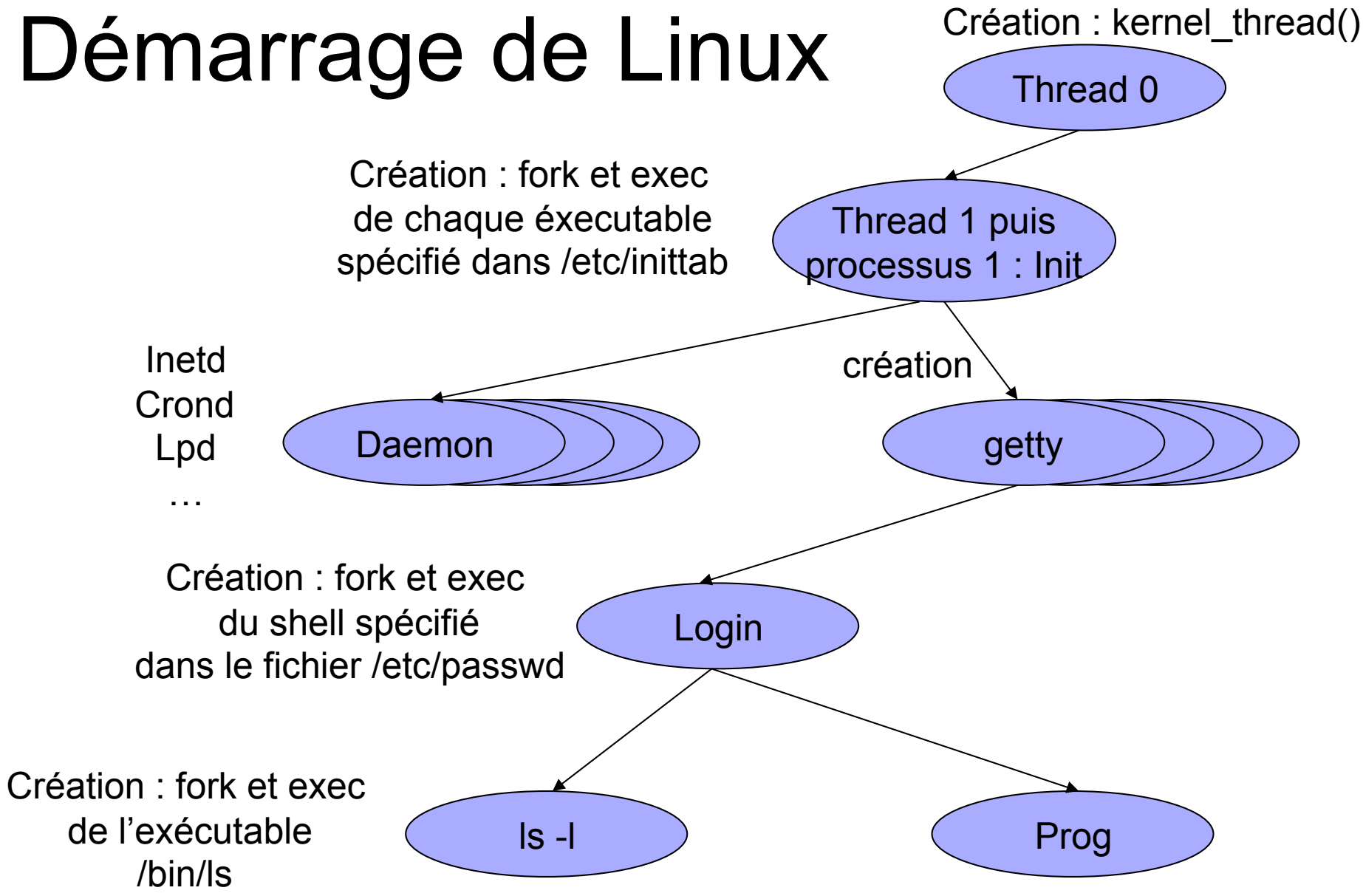
Exemple 2 (1)

```
#include <stdio.h>
#include <pthread.h>

int i;

void addition(){
      i = i + 10;
      printf("hello, thread fils %d\n", i);
      i = i + 20;
      printf("hello, thread fils %d\n", i);
}
```

Exemple 2 (2)

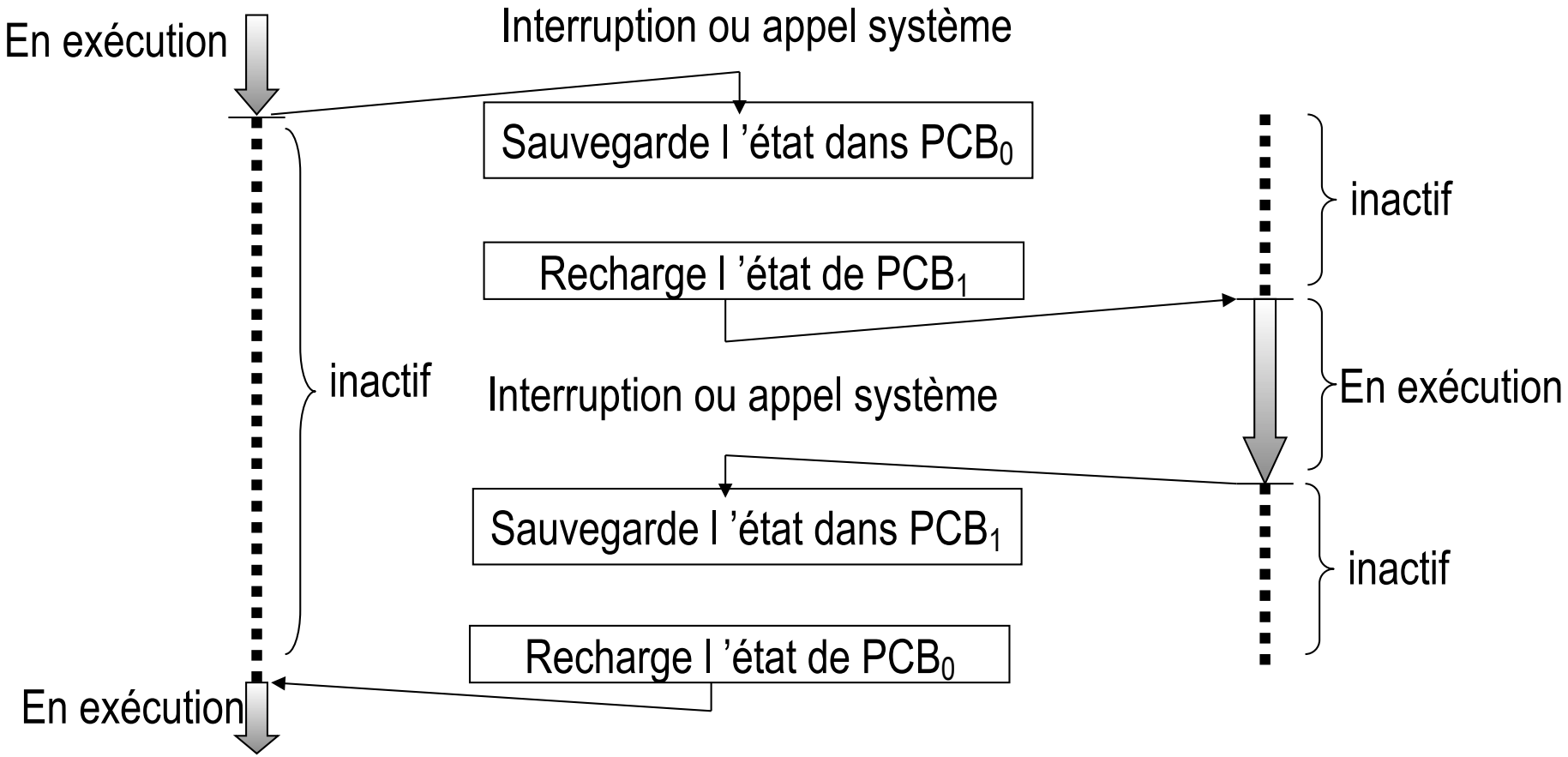
```
main() {
      pthread_t num_thread;

      i = 0;
      pthread_create(&num_thread, NULL, addition, NULL);

      i = i+ 1000;
      printf("hello, thread principal %d\n", i);
      i = i+ 2000;
      printf("hello, thread principal %d\n", i);
      pthread_join(num_thread, NULL);
}
```

Démarrage de Linux

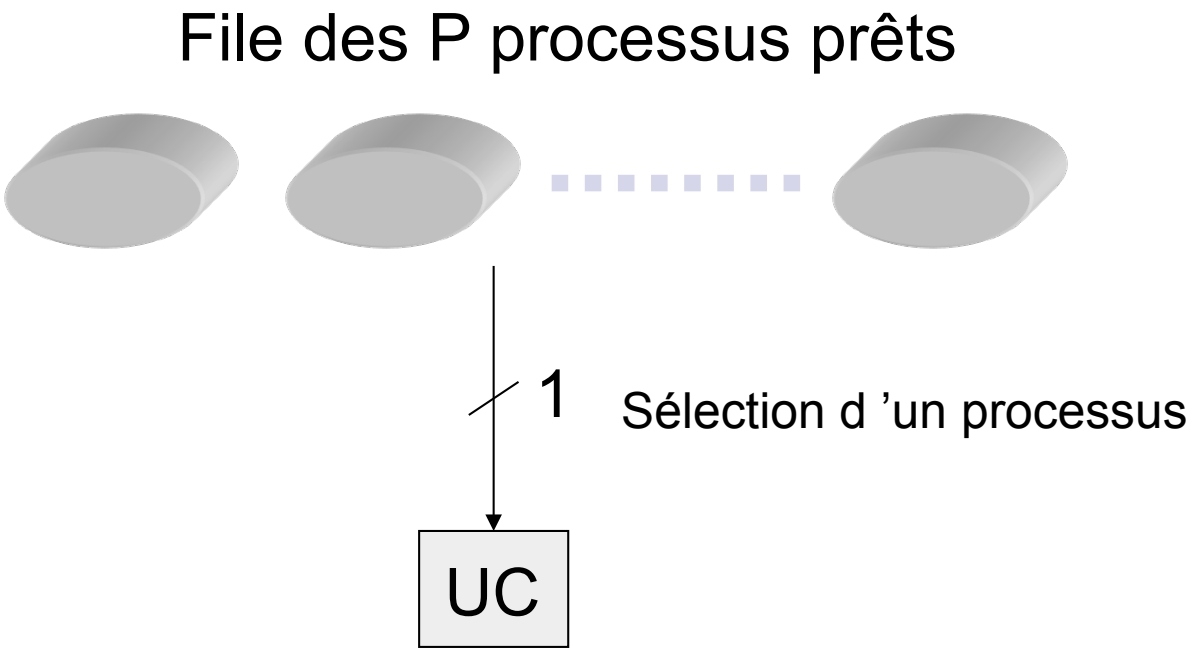


Commutation de l'UC entre processus

- Processus P0 SE Processus P1



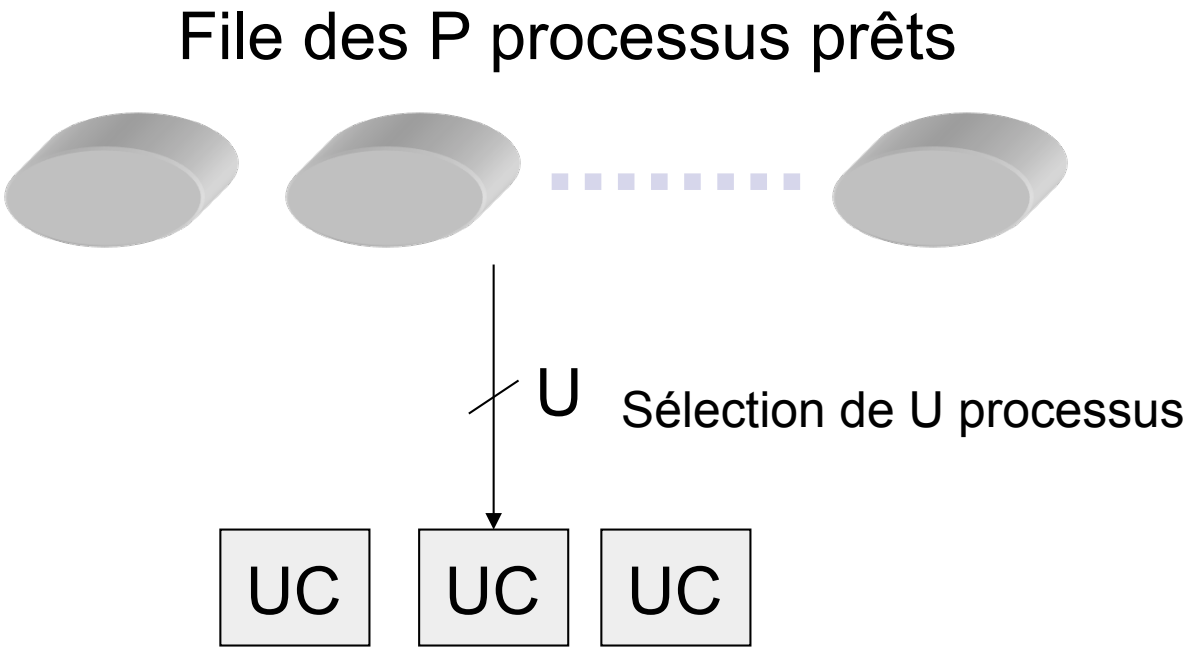
Ordonnancement des processus



PB : Quel processus dans la file d 'attente des processus prêts doit être affecté à l 'UC?

Ordonnancement des processus

Cas d'une machine parallèle avec U UC



Algorithmes d'ordonnancement

- Un bon algorithme doit être :
  - Équitable
    - s'assurer que chaque processus reçoit sa part du temps CPU
  - Efficace
    - utiliser le temps processeur à 100%
  - Réactif
    - minimiser le temps de réponse en mode interactif
    - minimiser le temps d'attente en traitement par lots
  - Performant
    - maximiser le nombre de travaux effectués en une heure

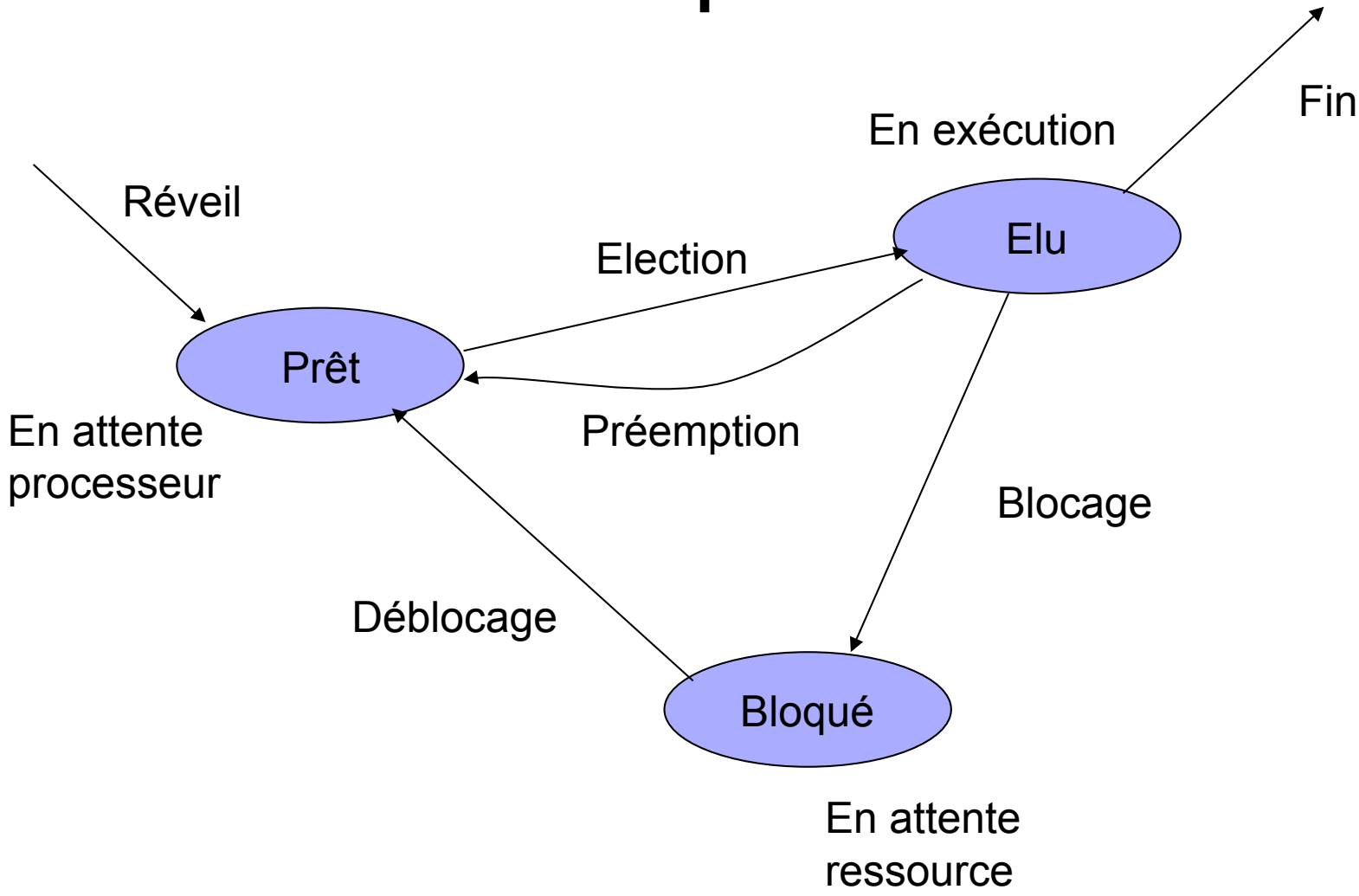
Types d’ordonnancement

- Ordonnancement sans réquisition (non préemptif)
  - Exécution jusqu’à terminaison d’un processus
  - Adaptés aux systèmes à traitement par lot
  - Non adaptés aux systèmes interactifs et multi-utilisateurs

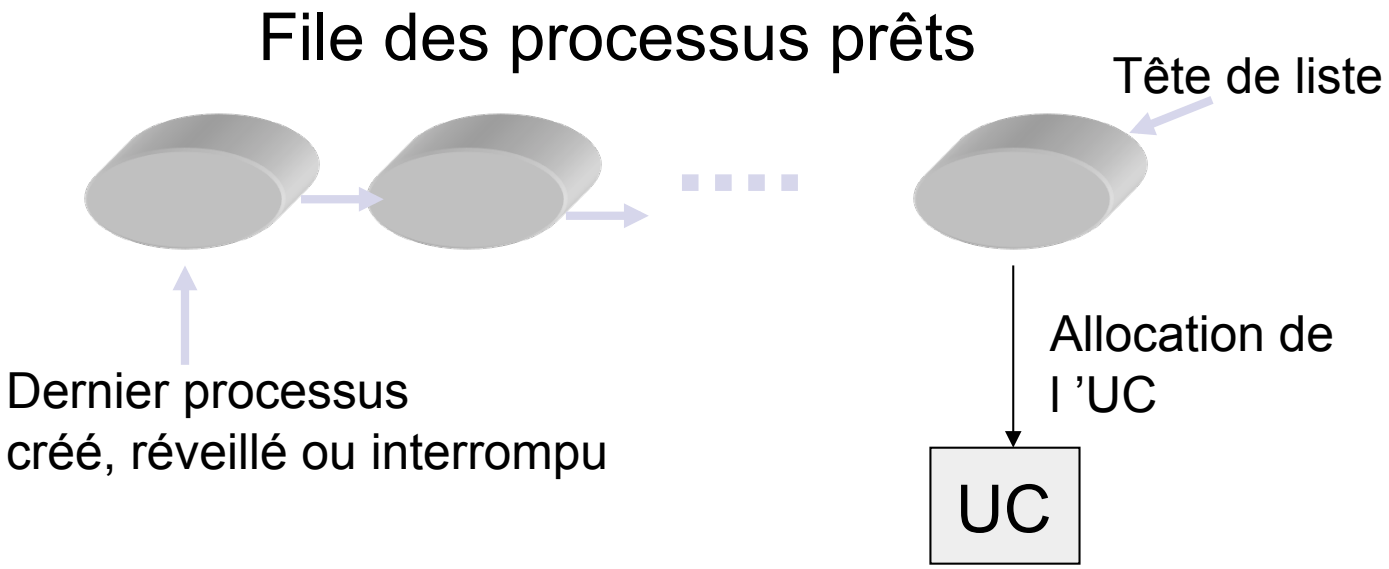
Types d'ordonnancement

- Ordonnancement sans réquisition (Collaborative Scheduling)
  - Exécution jusqu'à ce qu'un processus « passe la main »
- Ordonnancement avec réquisition (Preemptive Scheduling)
  - Suspension du processus en exécution
    - appels systèmes
    - interruptions matérielles
    - Allocation de l'UC par quantum de temps

Election / Préemption



Algorithmes d ’ordonnancement

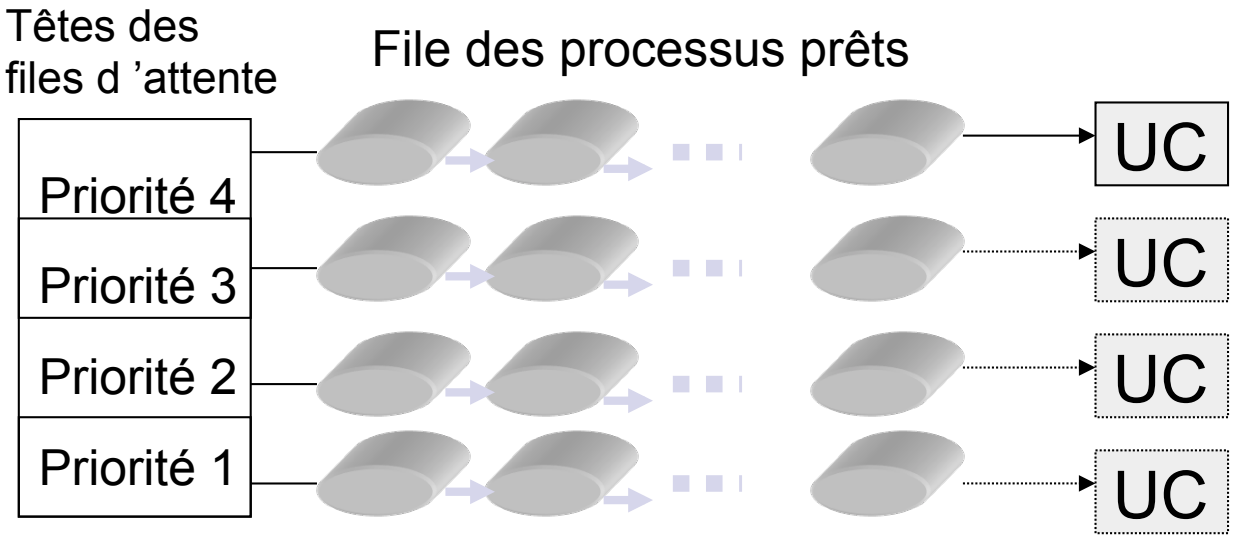


PB: Durée du quantum ?

- Un quantum trop petit provoque trop de commutations de processus
  changement de contexte devient coûteux
- Un quantum trop élevé augmente le temps de réponse

Algorithmes d ’ordonnancement

- Ordonnancement avec priorité
  - Idée de base : chaque processus possède une priorité et on lance le processus ayant la plus grande priorité



Algorithme d ’ordonnancement à classes de priorité

Pour éviter que les processus de priorité élevée monopolisent l ’UC, l ’ordonnanceur diminue leur priorité à chaque interruption horloge

Algorithmes d'ordonnancement

- Ordonnancement du plus court d 'abord
  - suppose la connaissance des temps d 'exécution
  - les travaux sont disponibles simultanément
  - adaptés aux traitement par lot
- Intérêt : minimise le temps moyen d 'exécution.
  - Considérons 4 travaux A,B,C et D dont les temps respectifs d'exécution sont a,b,c et d .
    - le travail A se termine au bout du temps a
    - le travail B se termine au bout du temps a+b
    - le travail C se termine au bout du temps a+b+c
    - le travail D se termine au bout du temps a+b+c+d
    - le temps moyen d'exécution est (4a+3b+2c+d)/4

La commande ps

- Permet d'obtenir des infos sur les processus du système
- L'option –l affiche pour chaque processus :
  - L'identité du propriétaire (UID)
  - L'identité du processus (PID)
  - L'identité du processus père (PPID)
  - L'état du processus
  - Des infos relatives à la priorité du processus (C/PRI/NI)
  - L'adresse du processus (ADDR) et sa taille (SZ)
  - Le terminal de contrôle du ps (TTY)
  - La raison de sa mise en sommeil s'il est endormi (WCHAN)

```
  F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY          TIME CMD
000 S   501  2515  2513  0  73   0    -   690 wait4  pts/1    00:00:00 bash
000 R   501  2547  2515  0  74   0    -   772 -      pts/1    00:00:00 ps
```