Principes des Systèmes d'exploitation

IUT de Villetaneuse
D. Buscaldi

3. Gestion et Ordonnancement des Processus

Notion de Processus

- Programme ≠ Processus
  - un même programme peut avoir plusieurs exécutions simultanées
- **Programme**: description d'un traitement (statique)
- **Processus**: programme en cours d'exécution (dynamique)
  - Code
  - Données
  - Contexte d'exécution (IP, registres, état de la pile, fichiers ouverts…)

- Le Système d'exploitation doit:
  - Empêcher les processus de monopoliser l'UC
  - Empêcher les processus d'attendre des ressources pour un temps indéfini (*starvation* ou *famine* du processus)
  - Empêcher les processus d'interférer les uns avec les autres
  - Leur donner les moyens d'échanger des données entre eux (*IPC, inter-process communication*)

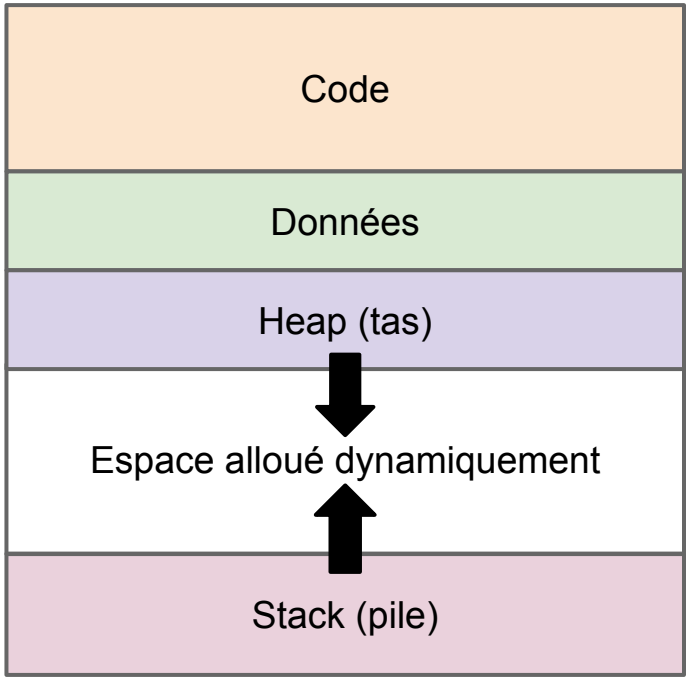
Création des processus

- Créer un processus nécessite:
  - Le nommer
  - Créer un Bloc de Contrôle de Processus (BCP) qui contient les informations nécessaires au SE pour le contrôle et l'exécution du processus
  - Déterminer sa *priorité*
  - Allouer les ressources

- Deux approches:
  - **Génération**: le processus est constitué à partir de l'exécutable indépendamment du processus demandeur
  - **Copie/recouvrement** (UNIX, Windows):
    - Copier un processus existant (celui qui demande la création)
    - Recouvrement de l'image avec celle d'un autre exécutable

Image du processus

- Code: le code de l'exécutable
- Données: variables globales et statiques utilisées par le programme et qui sont initialisées
- Heap (tas): géré par malloc et free pour réajuster sa taille, contient structures partagées par les libraries, les modules chargés dynamiquement par le processus
- Stack (pile): Appels à fonctions, variables locales, ...



Bloc de Contrôle des Processus

- 2 parties:
  - Table des processus
  - U (User) area
- Table des processus
  - Toujours en MC
  - Informations dont le système peut avoir besoin même quand le processus n'est pas actif:
    - identité, état, paramètres d'ordonnancement…
- U_area
  - Une zone par processus
  - Informations nécessaires seulement si le processus est actif
    - registres, pile noyau, descripteurs de fichers…
- /proc
  - Système de fichiers d'info sur les processus
  - Un répertoire pour chaque processus

```
$ cat /proc/6120/status
Name: gnome-panel
State:      S (sleeping)
Tgid: 6120
Pid:  6120
PPid: 5999
```

Le premier processus

- Au démarrage du système, il n'y a que le processus boot (*system_idle* pour Windows), identifié avec le numéro (**PID** ou Process ID) 0
- Initialisation du système (*start_kernel*()) puis création du premier processus: init n°1
- Puis, démarrage du scheduler et le main du boot rentre dans un boucle infini (*cpu_idle()*)

```
asmlinkage __visible void __init start_kernel(void) {
        … //beaucoup de procedures d'initialisation
        boot_cpu_init();
        page_address_init();
        pr_notice("%s", linux_banner);
        setup_arch(&command_line);
        mm_init_cpumask(&init_mm);
        …
        sched_init(); //démarrage du scheduler
        …
        rest_init(); //appelle cpu_idle()
}
```

Code:

https://github.com/torvalds/linux/blob/master/init/main.c

L'arbre des processus

```
$ pstree
init─┬─dirmngr
     ├─eclipse───java─┬─python2.7─┬─orted
     │                │           └─6*[{python2.7}]
     │                └─55*[{java}]
     ├─at-spi-bus-laun───2*[{at-spi-bus-laun}]
     ├─chrome─┬─31*[chrome───8*[{chrome}]]
     │        ├─chrome───9*[{chrome}]
     │        ├─2*[chrome───14*[{chrome}]]
     │        ├─chrome───33*[{chrome}]
     │        ├─chrome───7*[{chrome}]
     │        └─chrome───12*[{chrome}]
     ├─gnome-terminal─┬─5*[bash]
     │                ├─bash───pstree
     │                ├─bash───eclipse───java─┬─python2.7─┬─orted
     │                │                       │           └─
{python2.7}
     │                │                       └─{java
     │                ├─gnome-pty-helpe
     │                └─3*[{gnome-terminal}]
```

```
buscaldi@bichkek:~$ ps -ef
UID        PID  PPID  C STIME TIME CMD
root         1     0  0 août21 00:00:07 init [2]
…
root      5598     1  0 août21 00:00:11 /usr/sbin/gdm3
…
root      5635  5598  0 août21 00:00:00 /usr/lib/gdm3/gdm-simple-slave
…
root      5969  5635  0 août21 00:00:51 gdm-session-worker [pam/gdm3]
…
buscaldi  5999  5969  0 août21 00:00:37 gnome-session --session
gnome-fa
…
buscaldi  6120  5999  0 août21 00:03:29 gnome-panel
```

```
buscaldi@bichkek:~$ ps -f --ppid 6120
UID        PID  PPID  C STIME TIME CMD
buscaldi  6340  6120  6 août21 18:37:56 /usr/lib/icedove/icedove-bin
buscaldi  6364  6120  0 août21 02:25:05 /opt/google/chrome/chrome
buscaldi  8664  6120  0 août21 00:00:57 /usr/bin/perl
/usr/bin/shutter
buscaldi 30343  6120  0 août21 00:00:14 gnome-terminal
```

Fork

- **`pid_t fork(void);`** crée une copie du processus courant et renvoie:
  - 0 si on est dans le processus fils
  - -1 en cas d'erreur
  - le PID du fils (>0) si on est dans le père

- Un processus peut avoir zéro ou plusieurs fils, mais un seul parent

- Exécution en concurrence: L'ordre d'exécution est imprévisible!

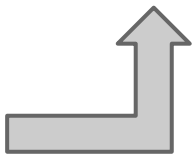
```
#include <unistd.h>
#include <sys/types.h>

pid_t fork ( ) ;
pid_t n ;
...
n = fork( ) ;
if ( n == 0 ) {
      /* fils; possibilité de recouvrement; */
} else {
      /* père ; n reçoit le pid du fils;
      possibilité d'attendre la fin du fils */
}
...
```

Fork - Exemple

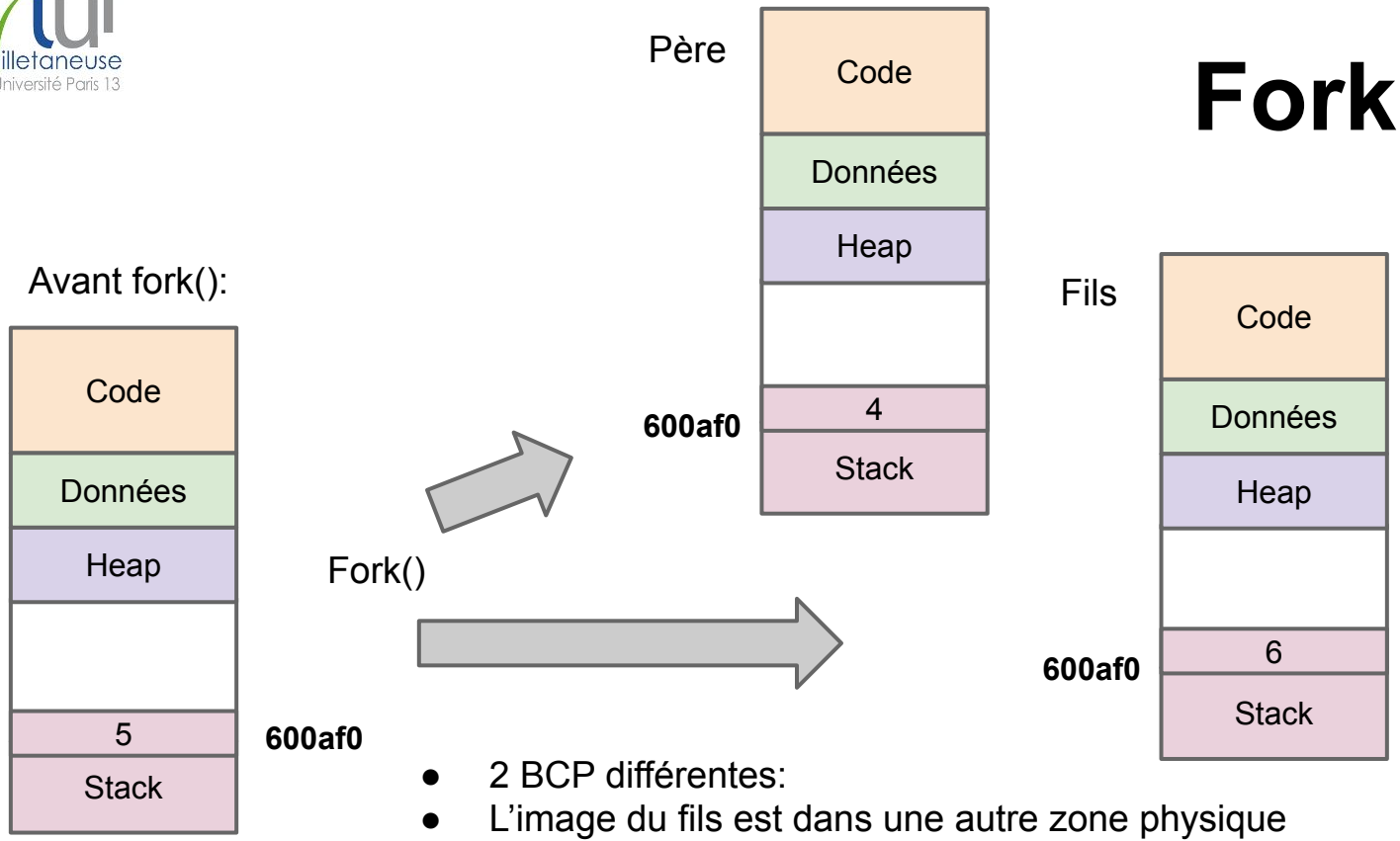
Valeur et adresse d'une variable dans le père et le fils:

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

int i = 5 ;

int main ( void ) {
      pid_t idf ;
      printf("Avant fork - Adr i: %x i= %d \n\n",&i, i);
      idf = fork() ;
      if ( idf == 0 ) {
            printf("FILS - Adr i: %x i= %d \n", &i, i );
            i++;
            printf("Après MODIF, FILS - Adr i: %x i= %d \n", &i, i );
      } else {
            sleep(1) ;
            /* pour que fils s'exécute d'abord */
            printf("\nPERE - Adr i: %x i= %d \n", &i, i);
            i--;
            printf("Après MODIF, PERE - Adr i: %x i= %d \n", &i, i);
      }
}
```

```
$> ./test
Avant fork - Adr i: 600af0 i= 5

FILS - Adr i: 600af0 i= 5
Apres MODIF, FILS - Adr i: 600af0 i= 6

PERE - Adr i: 600af0 i= 5
Apres MODIF, PERE - Adr i: 600af0 i= 4
```



Fork (suite)



- 2 BCP différentes:
- L'image du fils est dans une autre zone physique
- Le système gère les deux images aux adresses virtuelles identiques

Recouvrement

- Remplacer l'image mémoire du processus en cours par un nouveau processus
- Famille de fonctions **exec:**
  - execv, execvp, execve : nombre fixe de paramètres
  - execl, execlp, execle : nombre variable de paramètres
    - v: tableau
    - l: liste
    - p: avec PATH
    - e: avec environnement

```
/* spécificaton execl */
int execl (const char *ref, const char *arg0, …, const char *argN, NULL)
```

retour: toujours -1
(Si retour, il y a un erreur)

Référence du exécutable
(ex: “/bin/ls”)

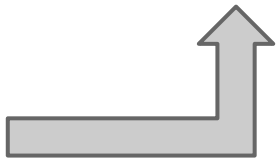
Liste des arguments du exécutable

```
/*Exemple:*/
execl (“bin/ls”, “ls”, “-l”, NULL)
```

Recouvrement - Exemple

```
$> ./test2
Et la liste?

Voici mon fils
total 32
drwxr-xr-x 2 buscaldi users 4096 sept.  1 15:46 .
drwxr-xr-x 4 buscaldi users 4096 sept.  1 14:55 ..
-rwxr-xr-x 1 buscaldi users 7461 sept.  1 15:46 test2
-rw-r--r-- 1 buscaldi users  338 sept.  1 15:46 test2.c

$>
```

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

int main ( void ) {
      pid_t p ;
      p = fork() ;
      if ( p == 0 ) {
            execl("/bin/echo", "echo", "-e", "Et la liste?\n", 0);
      } else {
            sleep(1) ;
            fprintf(stderr, "Voici mon fils\n");
            /*stdout n'affiche rien (recouvrement anticipe sortie)*/
            sleep(1);
            execl("/bin/ls", "ls", "-l", "-a", 0);
            fprintf(stderr, "Fini");
      }
}
```



getpid(), getppid()

- On peut connaître l'identité du processus appelant et le père du processus appelant:

```
pid_t getpid(void); /*Identité du processus appelant*/
pid_t getppid(void); /*Identité du père du processus appelant*/
```

- Exemple:

```
int main ( void ) {
      pid_t idf ;
      idf = fork() ;

      if ( idf == 0 ) {
            printf("Proc %d fils de %d \n", getpid(), getppid() );
      } else {
            printf("Je suis le proc %d \n", getpid());
      }
}
```

```
$> ./test3
Je suis le proc 13723
Proc 13724 fils de 13723
```

Groupes, Sessions

- Les processus sont rassemblés en *groupes*
  - permet de contrôler la distribution de **signaux**: un signal envoyé à un groupe est envoyé à tous les membres du groupe
- Les groupes sont rassemblés en *sessions*:
  - un terminal de contrôle commun aux différents processus

- **Signal:** mécanisme asynchrone permettant d'alerter un processus
  - l'envoi d'un signal à un processus revient à positionner un bit à 1 dans son bloc de contrôle
  - Les signaux sont traités par le noyau juste avant de repasser en mode utilisateur
- *On va regarder les signaux en détail plus tard dans le cours*

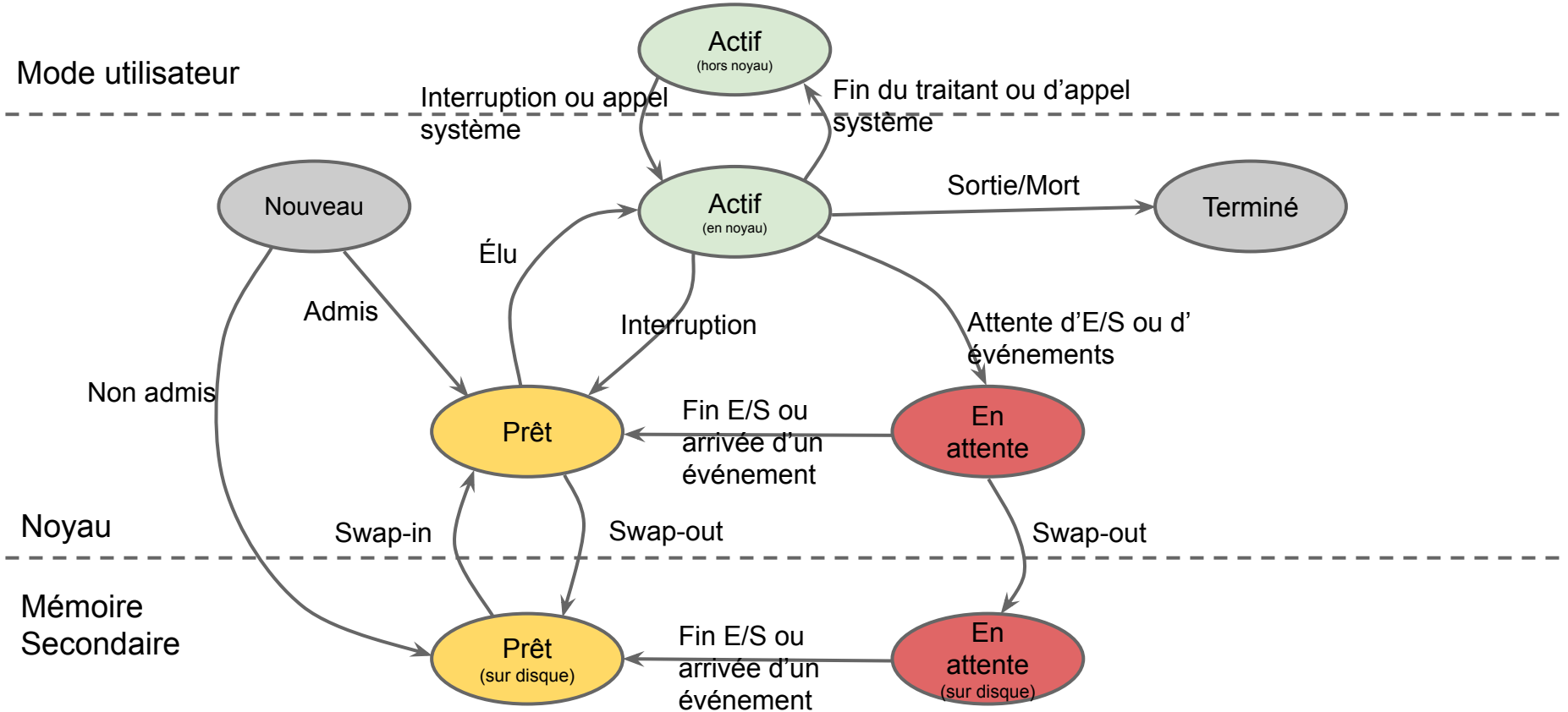
Groupes, Sessions

```
pid_t getpgid(pid_t); /*Identité du processus “leader” de groupe du processus en paramètre*/
pid_t getsid(pid_t); /*Identité du processus “leader” de la session qui inclut le processus en paramètre
                  /*(si paramètre=NULL, on considère le processus appelant*/
```

```
int main ( void ) {
        pid_t idf ;
        idf = fork() ;
        if ( idf == 0 ) {
                printf("Proc %d fils de %d \n", getpid(), getppid() );
                printf("GID du fils: %d \n", getpgid(0) );
                printf("Session du fils: %d \n", getsid(0) );
        } else {
                printf("Je suis le proc %d \n", getpid());
                printf("GID du pere: %d \n", getpgid(0));
                printf("Session du pere: %d \n", getsid(0));
        }
}
```

```
$> ./test3
Je suis le proc 5345
GID du pere: 5345
Session du pere: 3040
Proc 5346 fils de 5345
GID du fils: 5345
Session du fils: 3040
$> ps
 PID TTY          TIME CMD
 3040 pts/7    00:00:00 bash
 5645 pts/7    00:00:00 ps
$>
```

États d'un processus



Terminaison d'un processus

- Terminaison demandée par le processus
  - exit(), _exit()
    - différence: exit() vide les buffers
  - Main terminé

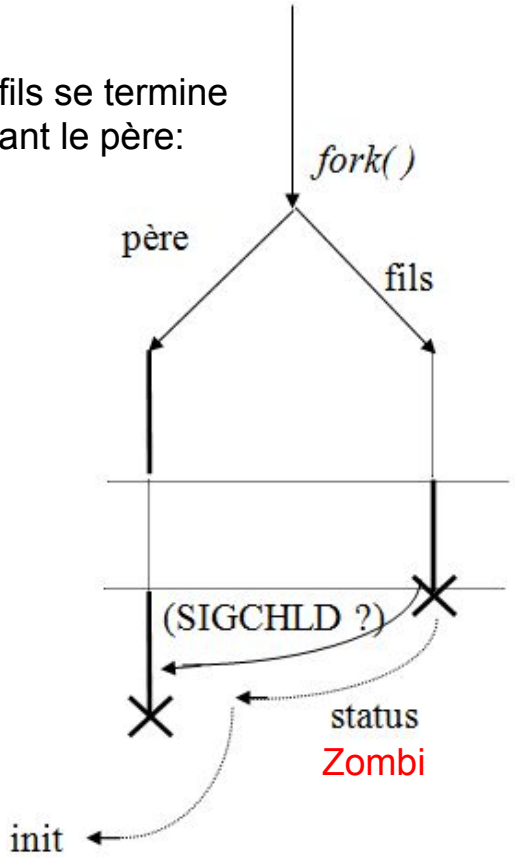
- Terminaison à la suite d'un événement
  - division par zéro
  - erreur de segmentation
  - mort du leader de session
  - etc…

Terminaison d'un processus

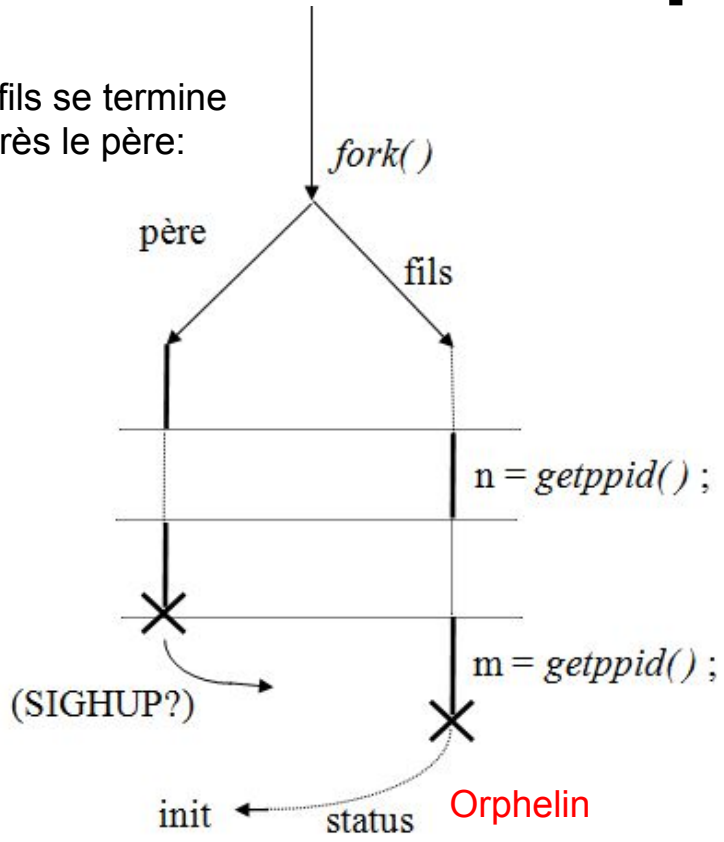
- Libération des ressources : mémoire, verrous, …
- Fermeture des fichiers ouverts
- Si fin de la session:
  - signaler l'événement aux fils (signal SIGHUP)
- Rattachement des processus fils (orphelins) au processus init,
- Informer le père (signal SIGCHLD)
  - sauvegarde du code de retour
  - devenir zombie:
    - Processus achevé mais qui dispose toujours de son PID
    - Son Bloc de Contrôle existe encore
    - Il sera définitivement éliminé quand le père fait **`wait()`** ou **`waitpid()`** (Primitives de synchronisation)

Terminaison d'un processus

le fils se termine avant le père:

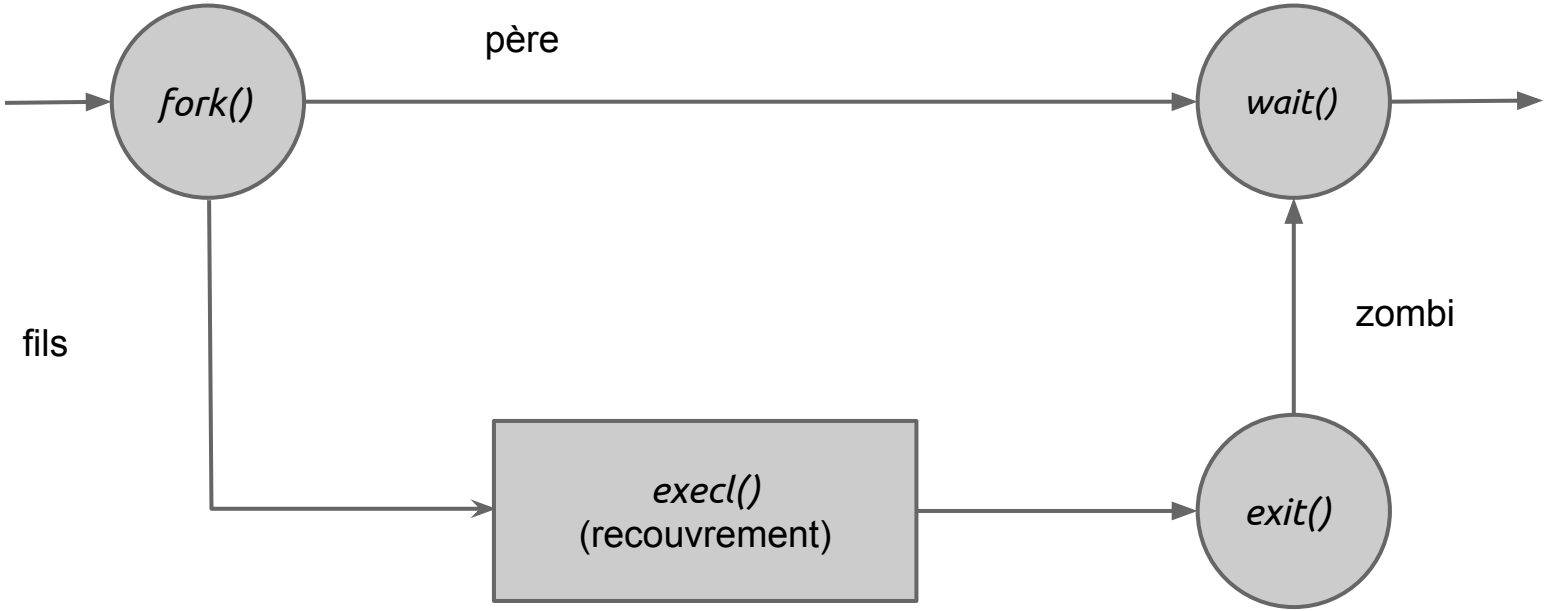


le fils se termine après le père:



n= pid du père
m= 1 (pid d'init)

Synchronisation père-fils



wait()

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

pid_t  wait(int *ptr_status) ;
```

- Valeur de retour:
  - -1 si aucun fils
  - pid d'un fils zombie (s'il y en a)
  - aucun des fils n'est terminé:
    - Le père se bloque
- Paramètre `ptr_status` reçoit le code de retour (si *ptr_status == NULL) alors le code de retour n'est pas récupéré

wait() - exemple

```
#define  nb_fils  3
main() {
        pid_t  n[nb_fils] , fini ;
        int  i , j ;
        for ( i = 0 ; i < nb_fils ; i++ )
          if (  (n[i]= fork() ) == 0 ) {
              printf("\t\t Fils %d commence : \n ",i);
              sleep(2) ;
              exit(0)  ;  // Fils se terminent.
          } else   printf("\n\t Pere a cree le fils %d de pid: %d \n",i,n[i]);

           // Seul le pere execute la suite
          printf( "\n  L'ordre de terminaison : ") ;
          for ( i = 0  ; i < nb_fils ; i++ )  {
                fini = wait(0);
                j = 0 ;
                while ( n[j] != fini  ) j++ ;
                printf(" %d ", j );
          }
          printf("\n ");
}
```

wait() - exemple

```
$> ./testwait

        Pere a cree le fils 0 de pid: 19579
                Fils 0 commence :

        Pere a cree le fils 1 de pid: 19580
                Fils 1 commence :

        Pere a cree le fils 2 de pid: 19581

                Fils 2 commence :
    L'ordre de terminaison :  0  1  2
```

waitpid()

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

- Valeur de retour:
  - -1 pas de fils
  - 0: échec
  - pid d'un fils zombie (s'il y en a)
- Paramètre ***pid:***
  - -1 : le père attend la mort d'un des fils
  - 0 : le père attend la mort d'un des fils du même groupe de l'appelant
  - >0 : le père attend la mort du fils avec PID *pid*
- Paramètre options:
  - 0 bloquant
  - 1 non bloquant

waitpid() - exemple

```
main ( ) {
  pid_t   pid[2] ;
  int i , status;

  for ( i = 0 ; i < 2 ; i++ )
     pid[i]=fork()
     if ( pid[i] == 0 ) {
          sleep (2*i+1) ;   /* Au lieu du code des fils */
          break ;        /* Evite fork pour fils */
     }
  if ( i == 2 )              /* seulement père   */
  {  pid[0] = waitpid ( pid[0], &status , 0 ) ;
        if (  pid[0] < 0  || status != 0  )
          fprintf(stderr, “erreur”) ;
  }
  printf ( " %d se termine \n ",i ) ;
          /* Pere aura i = 2  */
  exit(0) ;
}
```

bloquant

```
$> ./testwaitpid
 0 se termine
  2 se termine
$>  1 se termine
```

Ordonnancement

- Quand le CPU change de contexte pour un autre processus il faut sauvegarder et charger les nouveaux états (BCP)
- Le temps du Context-switch est un surcoût:
  - cela ne fait pas progresser le programme / calcul

- L'**ordonnanceur** (scheduler) doit ordonnancer les processus (travaux/jobs) et assurer des propriétés comme:
  - éviter les famines (tous les processus auront un jour la main)
  - l'équilibrage des charges (contexte multi-coeur)
  - garantir un temps de réponse raisonnable aux processus interactifs
  - éviter autant que possible le cache trashing

Types d’ordonnancement

- Ordonnancement à court terme :
  - seulement parmi les processus prêts en MC
  - plusieurs fois par seconde

- Ordonnancement à long terme :
  - parmi tous les processus prêts (MC + MS)
  - ordonnancement des tâches
  - 1 fois en qq minutes ou à la fin / à l’arrivée d’une tâche

Politiques d'ordonnancement

- Ordonnancement non-préemptif:
  - On attend qu'un processus termine ou fasse un appel bloquant
  - Très risqué (le processus peut rentrer dans un boucle infini)

- Ordonnancement préemptif :
  - L'ordonnanceur peut interrompre à tout moment un processus pour permettre à un autre de s'exécuter.
  - Une quantité de temps définie (*quantum*) est attribuée à chaque processus

Mesures d’efficacité

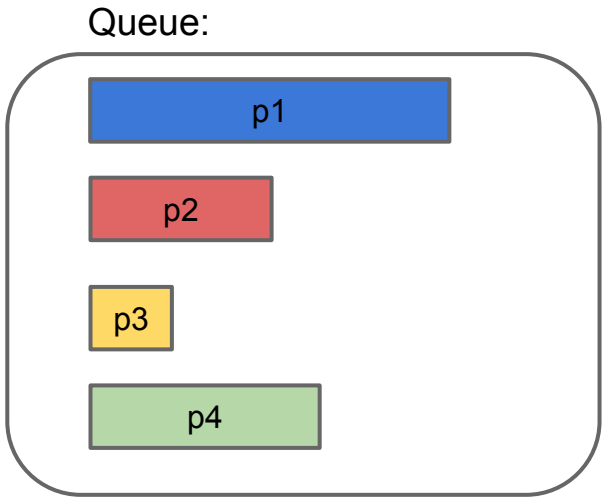
- Temps de rotation (turnaround time):
  - Combien de temps le processus reste dans le système avant d’être completé
  - T completion - T arrivée

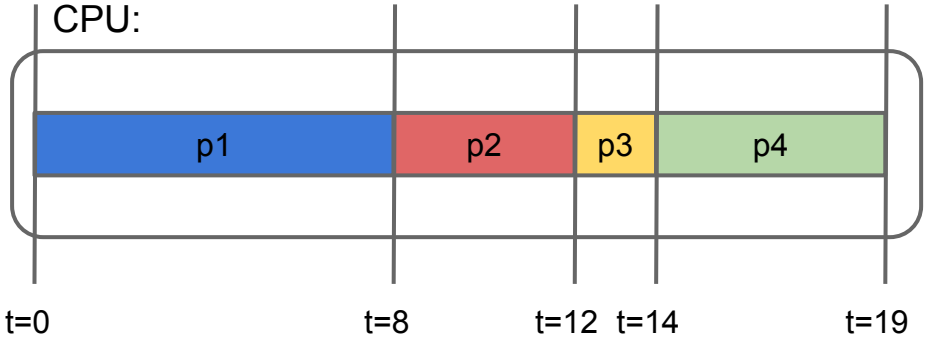
- Temps d’attente:
  - Combien de temps le processus reste en attente de la resource
  - T obtention - T arrivée

Algorithmes d’ordonnancement

- Non-préemptifs:
  - **FCFS** (First Come, First Served - premier arrivé, premier sorti)
  - **SJF** (Shortest Job First)
    - Privilège le plus court entre les processus en attente
    - Problème: difficulté de connaître la durée en avance
- Préemptifs:
  - **SRT** (Shortest Remaining Time)
    - L'ordonnanceur compare la valeur estimée de temps de traitement restant à celle du processus en cours d'ordonnancement
    - Si ce temps est inférieur, le processus en cours d'ordonnancement est préempté
  - **RR** (Round Robin o Tourniquet)
    - Sélectionne le processus qui attend depuis le plus longtemps
    - Préemption: après un quantum spécifié ou:
      - Terminaison
      - Accès E/S

FCFS



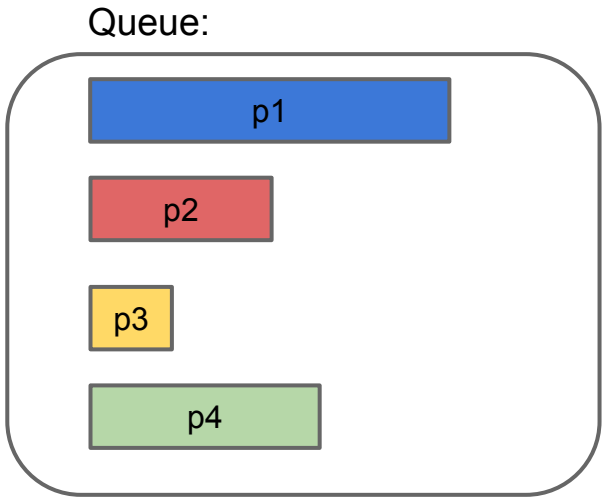


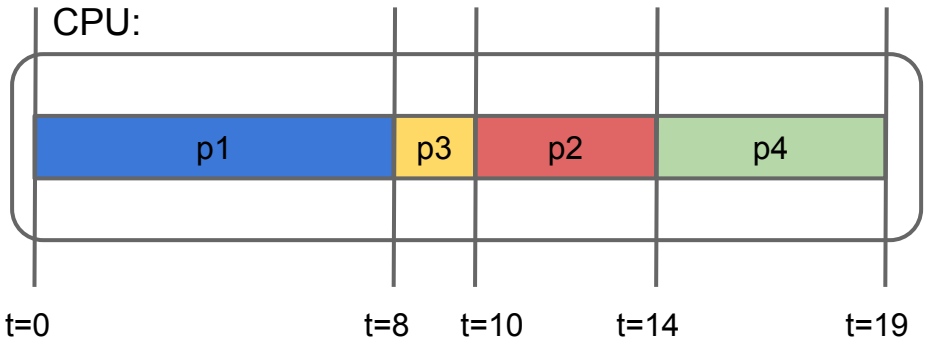
Processus	Temps d'arrivée	Durée
p1	0	8
p2	1	4
p3	2	2
p4	3	5

Temps de rotation (turnaround) moyen:
((8) + (12-1) + (14-2) + (19-3))/4 =45/4 = 11,25

Tempŝ d'attente moyen:
((0) + (8-1) + (12-2) + (14-3))/4 =28/4 = 7

SJF



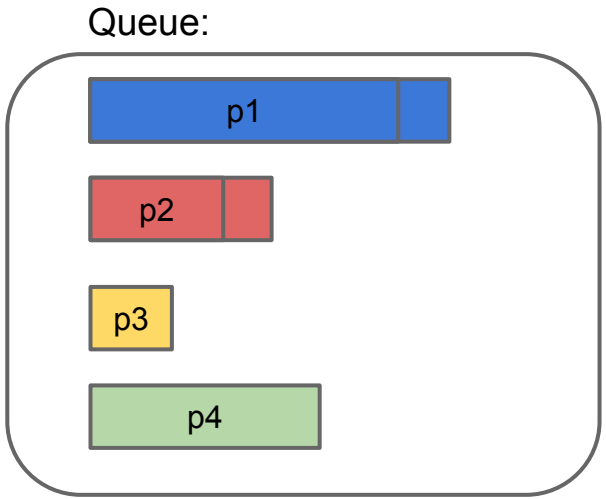


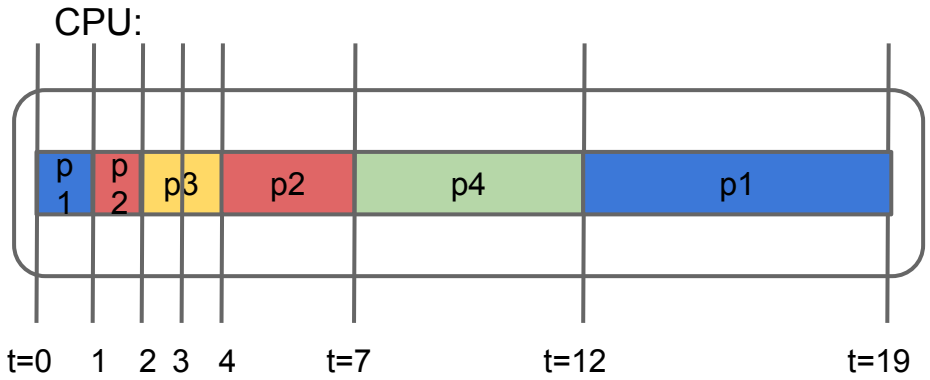
Processus	Temps d'arrivée	Durée
p1	0	8
p2	1	4
p3	2	2
p4	3	5

Temps de rotation moyen:
((8) + (10-2) + (14-1) + (19-3))/4 =45/4 = 11,25

Temps d'attente moyen:
((0) + (8-2) + (10-1) + (14-3))/4 =26/4 = 6,5

SRT



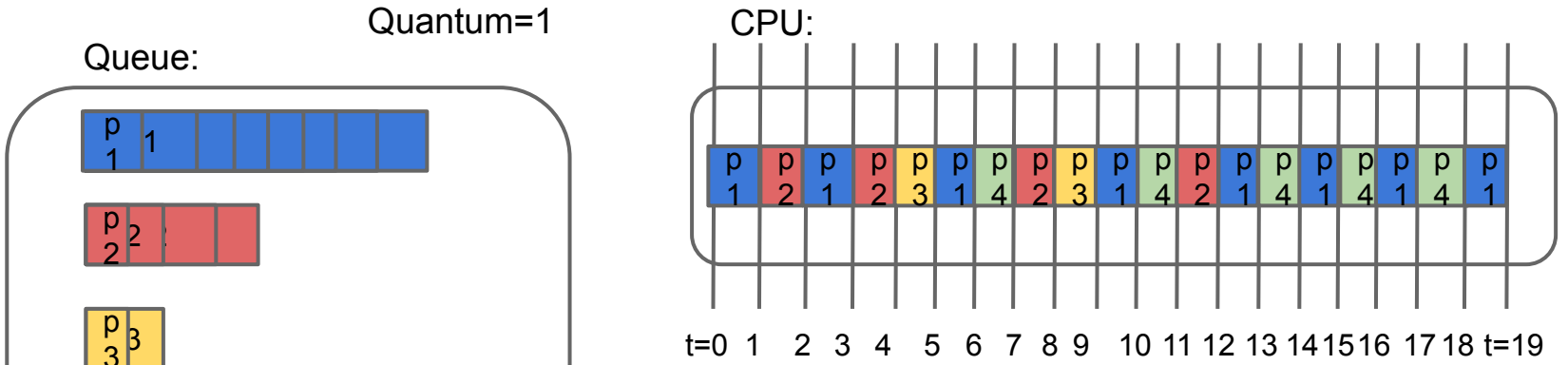


Processus	Temps d'arrivée	Durée
p1	0	8
p2	1	4
p3	2	2
p4	3	5

Temps de rotation moyen:
((19) + (7-1) + (4-2) + (12-3))/4 =36/4 = 9

Temps d'attente moyen:
((0+(12-1)) + (0+(4-2)) + (0) + (7-3))/4 =17/4 = 4,25

Round Robin (Tourniquet)



Processus	Temps d'arrivée	Durée
p1	0	8
p2	1	4
p3	2	2
p4	3	5

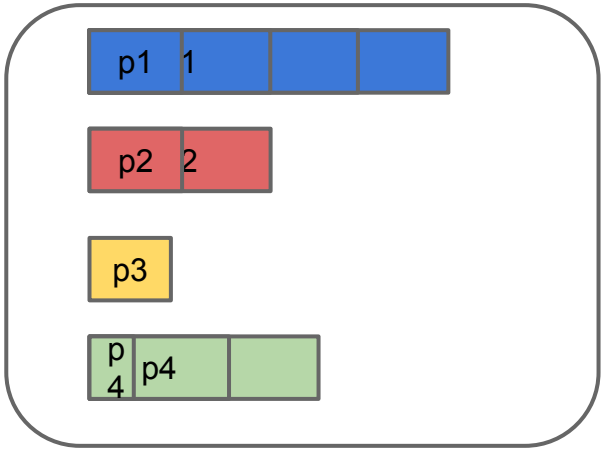
Temps d'attente moyen:
((0+1+2+3+2+1+1+1) + (0+1+3+3) + (2+3) + (3+3+2+1+1))/4 =33/4 = 8,25

Temps de rotation moyen:
((19-0) + (12-1) + (9-2) + (18-3))/4 =52/4 = 13

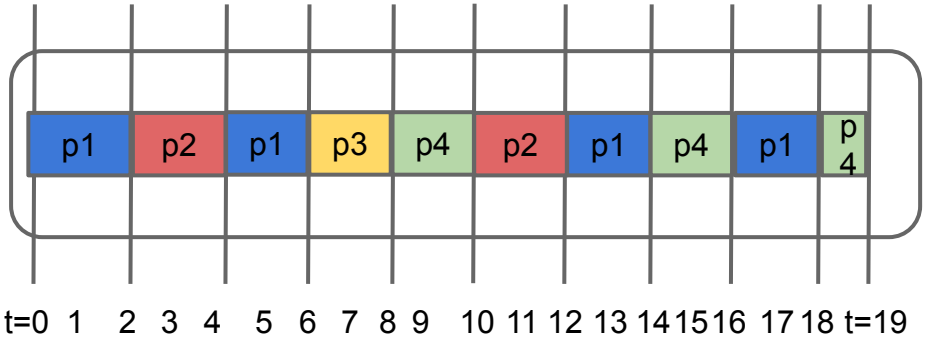
Round Robin (Tourniquet)

Quantum=2

Queue:



CPU:



Quantum court: trop de changements de contexte (*overhead*)
Quantum long: temps de réponse allongé -> devient FIFO

Temps d'attente moyen:
((0+2+6+2) + (1+6) + (4) + (5+4+2))/4 =32/4 = 8

Temps de rotation moyen:
((18-0) + (12-1) + (8-2) + (19-3))/4 = 51/4 = 12,75

p1	0	8
p2	1	4
p3	2	2
p4	3	5

Priorités

- Critères:
  - type de processus (système/utilisateur, compil/edit/…, premier plan/ arrière plan, ….)
  - profil du processus
  - taille du processus
  - nb de fichiers ouverts
- Priorité statique
  - principe : exécuter la première tâche de la priorité la plus haute
  - peut entrainer la famine : processus de priorité basse jamais exécuté
- Priorité dynamique
  - augmenter la priorité avec temps d'attente
  - baisser la priorité avec temps UC utilisé

Priorités

- On peut avoir une file par priorité
  - Linux: 140 files
  - Files séparés pour plusieurs processeurs aussi:
    - Linux 2.6 introduces l'*affinité*:
    - Les processus ont un champ de bits (modifiable) qui indique les CPU utilisables
      - Les fils héritent le même champ, donc ils ont la tendance à rester sur le même processeur
    - Si un processeurs est surchargé, le SE peut distribuer la charge

- Dans chaque file une discipline différente peut être appliquée

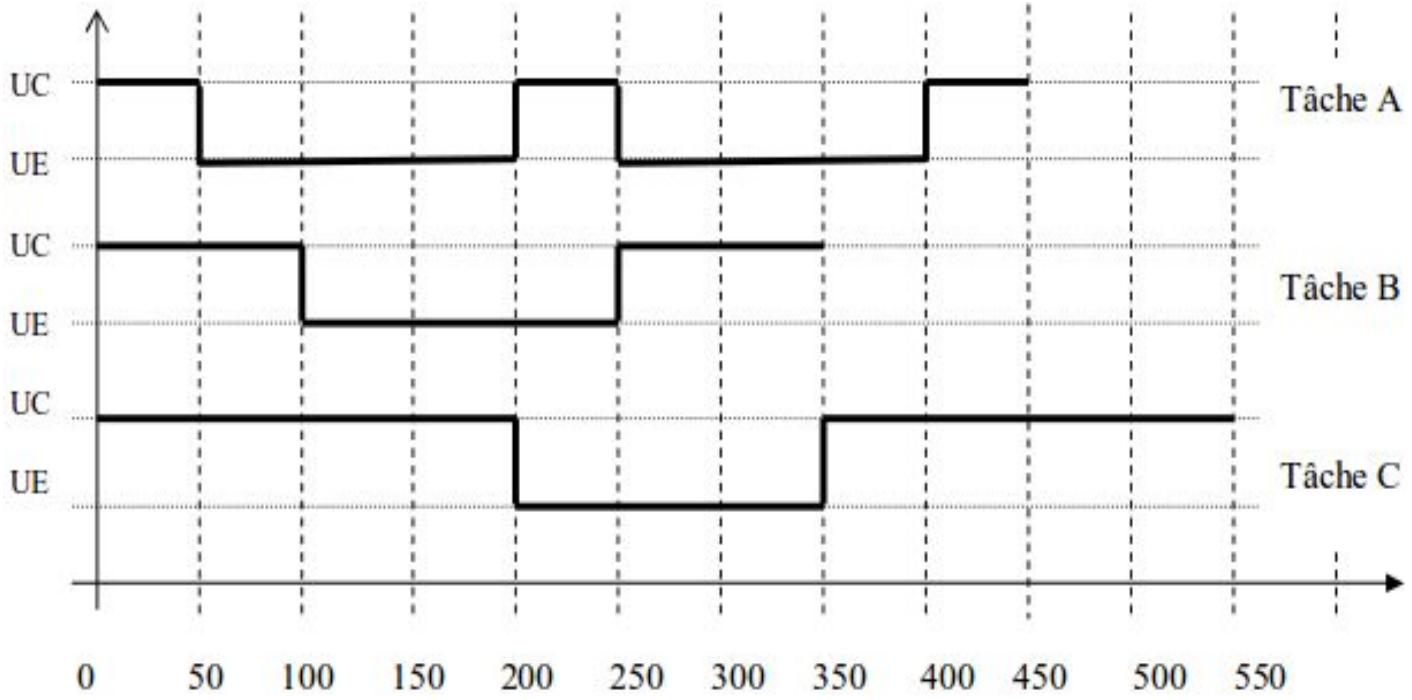
Ordonnancement en Linux

- 3 algorithmes:
  - *SCHED_OTHER* est l'ordonnancement universel temps-partagé par défaut, utilisé par la plupart des processus (PS)
  - *SCHED_FIFO* et *SCHED_RR* sont prévus pour des applications temps-réel qui nécessitent un contrôle précis de la sélection des processus prêts (TR)
    - par défaut tout processus est dans la classe PS, seul l'administrateur peut créer ou affecter des proc en classe TR

Ordonnancement PS

- 2 éléments pour chaque processus:
  - Priorité de base (statique) : p_base
  - Compteur de tranche (mesure du temps non utilisé) : compte
- Choix du processus à exécuter : le + grand compte
- Formule du compte:
  - compte = compte/2 + p_base
  - On le recalcule quand un des processus épuise son compte
- Possibilité de modifier la priorité de base avec la commande “ `nice` ”
  - Augmentation possible uniquement par root

p_base:
A=15, B=20, C=25



À chaque 10 ms, on diminue de 1 la valeur du compte du processus en cours d'exécution dans le processeur

Tâche	0	200	300	350	400	400	450	500	550	650	700	850	900
A	15	15	15	10* → 150	10* 100	20* 100	20* 50	20	20	20 →	15* 150	15 →	Fin
B	20	20 →	10* 150	10* 100	10* 50	25* 50	25	25	25 →	Fin			
C	25 →	5* 150	5* 50	5 →	0	25 →	20	15 →	Fin				
T. Active	C	B	A	C		C	C	C	B	A	---	A	

Temps de réponse :
A : 900 ms.
B : 650 ms.
C : 550 ms.

⇨ Temps de réponse moyen : 2100/3 ms. = 700 ms.

getpriority

- Pour connaître la priorité (statique) d'un processus:

```
#include <sys/resource.h>
int getpriority(int which, id_t who);
```

- *which* peut être:
  - PRIO_PROCESS, PRIO_PGRP, or PRIO_USER
- *who* respectivement doit être:
  - un pid, un gid ou un uid

Nice, taskset

- **`nice`** pour modifier la priorité statique d'ordonnancement:

```
$> nice -n commande
```

- **`taskset`** pour connaître (ou modifier, en tant que root) l'affinité d'un processus en exécution:

```
$> taskset --cpu-list -p pid
```