

Systèmes centralisés : TD1

Commutation

- Q : Comment les interruptions rendent-elles possible la multiprogrammation ?
- Q : Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ?

Justifiez votre réponse en expliquant le mécanisme de commutation de processus.

Systèmes centralisés : TD1

Comment les interruptions rendent-elles possible la multiprogrammation ?

- Les interruptions permettent la préemption du processeur au processus actif
 - L'ordonnanceur peut alors être exécuté pour déterminer un nouveau processus élu.
 - Les interruptions utilisées peuvent être des interruptions associées aux opérations d'E/S, ou bien explicitement programmées (interruption horloge).

Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ?

- Les sauvegardes/restaurations du contexte se font en mémoire vive
- Si la RAM est partagée, ce qui est, en général, le cas d'un biprocesseur, il est donc concevable qu'un processus s'exécute alternativement sur les deux processeurs.
- Mais cela peut poser des problèmes d'efficacité : défauts de cache

Systèmes centralisés : TD1

Ordonnancement

Dans les politiques d'ordonnancement, les notions **d'équité** (absence de famine) et de **priorité** sont à priori antagonistes.

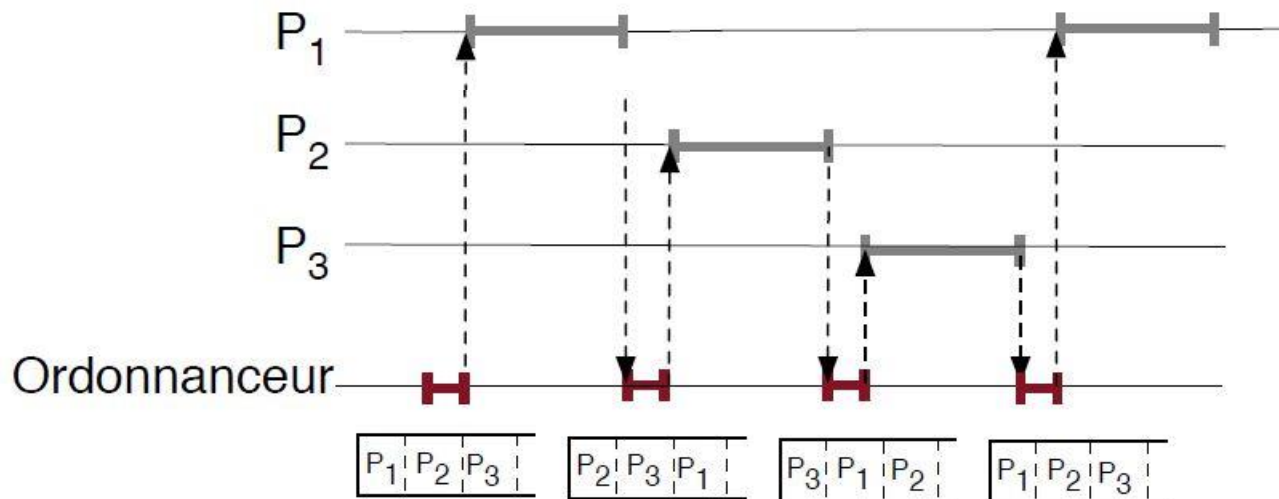
- Q : Comment introduire cependant une forme d'équité dans un système avec priorités ?
- Q : Présenter le scénario d'exécution du **tourniquet** avec un chronogramme
- Q : Programmer (en pseudo-code) l'algorithme de principe du tourniquet. On supposera que :
 - les processus sont identifiés par un entier (qui correspond à l'indice de leur descripteur dans la table des processus).
 - les processus sont gérés dans une file et que l'on dispose d'une opération commuter(courant , nouveau) qui sauvegarde le contexte d'exécution du processus actif (d'identifiant courant) pour installer/restaurer le contexte du nouveau processus
 - Et d'une opération armer_horloge (délai : entier), qui programme l'envoi d'un signal d'horloge après 'délai' ms

Systèmes centralisés : TD1

- Dans les politiques d'ordonnancement, les notions d'équité (absence de famine) et de priorité sont a priori antagonistes. Comment introduire cependant une forme d'équité dans un système avec priorités

Un moyen fréquemment employé est d'introduire le temps d'attente ou l'ancienneté dans le calcul de la priorité.

- scénario d'exécution du tourniquet avec un chronogramme



Systèmes centralisés : TD1

- Programmer (en pseudo-code) l'algorithme de principe du tourniquet.

```
// dans le programme principal
quantum <- 10
associer (IT HORLOGE, ordonnancer)
armer_horloge (quantum)
...
// traitant associé à IT_HORLOGE
procédure ordonnancer()
    si F est non vide alors
        F.enfiler (id courant);
        id_elu := F.defiler ();
        commuter (id_courant, id_elu);
    finsi
armer_horloge (quantum)
```

Systèmes centralisés : TD1

Ordonnancement par loterie : Le principe de l'algorithme d'ordonnancement par loterie est simple :

- les tickets de loterie sont numérotés de 0 à Max
 - À un processus est accordé un certain nombre de tickets, sous forme de plages de numéros successifs
 - l'espace des numéros [0 .. Max] est ainsi virtuellement partitionné en intervalles successifs, et un processus est (virtuellement) associé à chaque intervalle ;
 - lorsque le nombre N est tiré, la liste des processus est parcourue, en cumulant le nombre de tickets attribués. Le premier processus pour lequel le cumul est $\geq N$ est élu
-
- Q : Adapter l'algorithme pour améliorer l'efficacité de la boucle de recherche (minimiser le nombre d'itérations).
 - Q : Montrer que cette adaptation n'altère pas les chances d'accès au processeur de chacun des processus.

Systèmes centralisés : TD1

- Il suffit de ranger les processus par ordre décroissant du nombre de tickets attribués. Le cumul croît alors le plus rapidement possible.
- Pour un processus donné, les chances de tirage sont proportionnelles à son nombre de tickets, et ne sont pas altérées par la position (ou de la configuration) de la plage de tickets.

Le recours aux choix aléatoires permet d'allouer une fraction de temps processeur à chaque processus, indépendamment de l'utilisation qu'en font les autres processus, et ce de manière effective et efficace.

Cependant, dans certaines situations, comme dans le cas de systèmes critiques, le caractère aléatoire de l'allocation peut s'avérer problématique.

Des adaptations déterministes de l'algorithme de la loterie ont ainsi été élaborées, comme l'ordonnancement par pas (stride scheduling).

Les versions récentes du système Linux fonctionnent sur cette base.

Systèmes centralisés : TD1

Ordonnancement par pas

- L'ordonnancement par pas conserve le principe d'émission et d'attribution de tickets aux processus.
- Le ticket est remplacé par un pas, inversement proportionnel à la priorité
- Pour chaque processus, les pas sont cumulés à chaque fois que le processus est élu. Le cumul des pas d'un processus est proportionnel au temps processeur consommé par ce processus.
- L'algorithme d'ordonnancement proprement dit consiste simplement, à l'échéance du quantum, à parcourir la liste des processus pour trouver (et élire) le processus ayant le cumul de pas le plus faible (en cas d'égalité, l'identifiant de processus est utilisé pour départager et rester déterministe)

Q : Comparer avec la loterie, quel est le coût de l'introduction du déterminisme ?

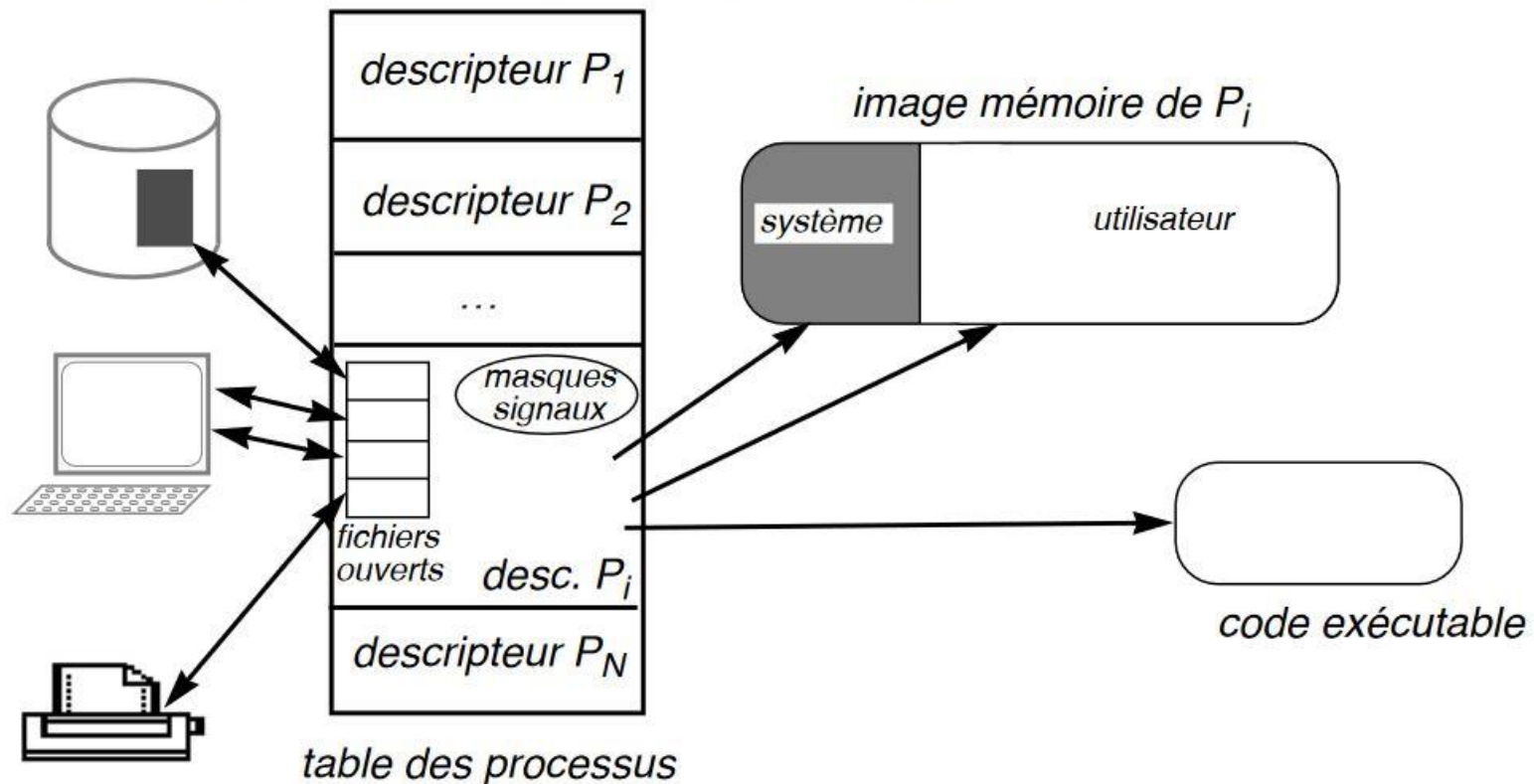
Systèmes centralisés : TD1

- L'ordonnancement par pas est plus lourd et moins souple que la loterie :
 - La recherche du cumul de pas le plus faible est coûteuse
 - Lors du lancement d'un nouveau processus, on doit évaluer/simuler ce qu'aurait été son cumul de pas s'il avait démarré en même temps que les processus existants (sinon, il monopoliserait le processeur jusqu'à rattraper les autres). On peut lui affecter un cumul de pas initial immédiatement inférieur au cumul du prochain processus à servir.
- L'allocation du processeur à un processus dépend des demandes des autres processus.
- Cependant, l'algorithme est équitable (tout processus finit par être exécuté),
- L'ordonnancement est déterministe, et prévisible.

Systèmes centralisés : TD1

Gestion des processus

Descripteur de processus (vue système)



Systèmes centralisés : TD1

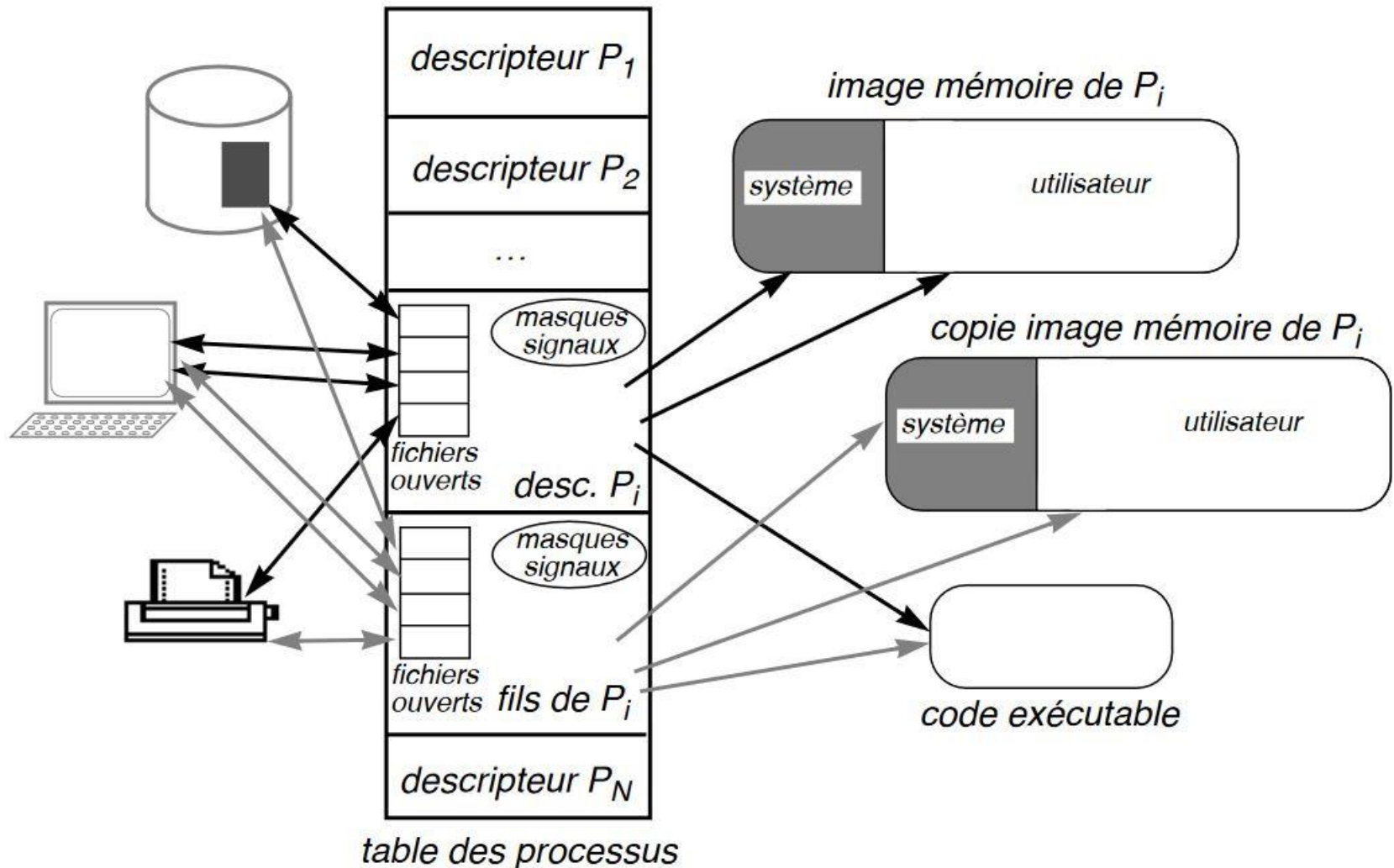
pid_t fork()

- Crée dynamiquement un nouveau processus (fils)
- Qui s'exécute de façon concurrente avec son père
- Hérite de ses attributs :
 - Même code
 - Une copie de la zone de données
 - Environnement d'exécution
 - Priorité
 - Descripteurs ouverts (fichiers, pipes, ...)
 - Traitement des signaux

```
    spid = fork();  
    // père  
    // spid = id du fils  
    // fils  
    // spid = 0  
    spid = -1  
    erreur
```

Systèmes centralisés : TD1

Table des processus après fork



Systèmes centralisés : TD1

```
1  /* test_fork.c : 3 fork successifs - combien de processus ? arborescence ? */
2  #include <stdio.h>          /* entrées sorties */
3  #include <unistd.h>        /* primitives de base : fork, ...*/
4
5  const char tvert[]="\033[32m";
6  const char tjaune[]="\033[33m";
7  const char tbleuc[]="\033[36m";
8  const char tblanc[]="\033[0m";
9
10 int main() {
11     int pid;
12
13     pid = fork();
14     printf("%sRetour fork = %d. Processus %d, Pere %d\n", tjaune, pid, getpid(),
15           getppid());
16     pid = fork();
17     printf("%sRetour fork = %d. Processus %d, Pere %d\n", tvert, pid, getpid(),
18           getppid());
19     pid = fork();
20     printf("%sRetour fork = %d. Processus %d, Pere %d\n", tbleuc, pid, getpid(),
21           getppid());
22     printf("%s\n", tblanc);
23     return 0;
24 }
```


Systèmes centralisés : TD1

```
hamrouni@behemot:~/SEC19/cours$ ./tf
Retour fork = 26479. Processus 26478, Pere 30258
Retour fork = 26480. Processus 26478, Pere 30258
Retour fork = 0. Processus 26479, Pere 26478
Retour fork = 0. Processus 26480, Pere 26478
Retour fork = 26481. Processus 26478, Pere 30258

Retour fork = 26482. Processus 26479, Pere 26478
Retour fork = 26483. Processus 26480, Pere 26478

Retour fork = 26484. Processus 26479, Pere 26478

Retour fork = 0. Processus 26481, Pere 1
Retour fork = 0. Processus 26484, Pere 1

hamrouni@behemot:~/SEC19/cours$ Retour fork = 0. Processus 26482, Pere 1

Retour fork = 0. Processus 26483, Pere 1
Retour fork = 0. Processus 26485, Pere 26482

Retour fork = 26485. Processus 26482, Pere 1
```

Systèmes centralisés : TD1

Père

Données

Descripteurs ouverts

Traitement des signaux

```
retour = fork();  
if retour < 0 exit(1);
```

```
if retour == 0 { // fils  
    ...  
}
```

```
else { // père  
    ...  
}
```

Fils

Copie des données
(modifiées : sur écriture)

Descripteurs ouverts

Traitement des signaux

```
if retour == 0 { // fils  
    ...  
}
```



Systèmes centralisés : TD1

pere_fils.c

```
/* Illustration des primitives Unix : Un père et ses 3 fils */
#include <stdio.h> /* entrées sorties */
#include <unistd.h> /* primitives de base : fork, ...*/
#include <stdlib.h> /* exit */
#define NB_FILS 3 /* nombre de fils */
int main() {
    int fils, retour ;
    int duree_sommeil = 30 ;
    printf("\nJe suis le processus principal de pid %d\n", getpid());
    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ; /* Tester systématiquement le retour des
                           appels système */
        if (retour < 0) { /* échec du fork */
            printf("Erreur fork\n") ;
            exit(1) ; /* exit avec une valeur != 0 en cas d'erreur */
        }
    }
}
```


Systèmes centralisés : TD1

```
/* fils */
if (retour == 0) {
    printf("\n Processus fils numero %d, de pid %d, de
           pere %d.\n", fils, getpid(), getppid()) ;
    sleep(duree_sommeil) ;
    printf("\n Fin du processus fils numero %d\n", fils);
    /* Important : terminer un processus par exit */
    exit(EXIT_SUCCESS) ; /* Terminaison sans erreur */
}
/* pere */
else {
    printf("\nProcessus de pid %d a cree un fils numero
           %d, de pid %d \n", getpid(), fils, retour) ;
}
}
return EXIT_SUCCESS ;
}
```

Qui passe 3 fois dans la boucle for ? Durée d'exécution du père ?

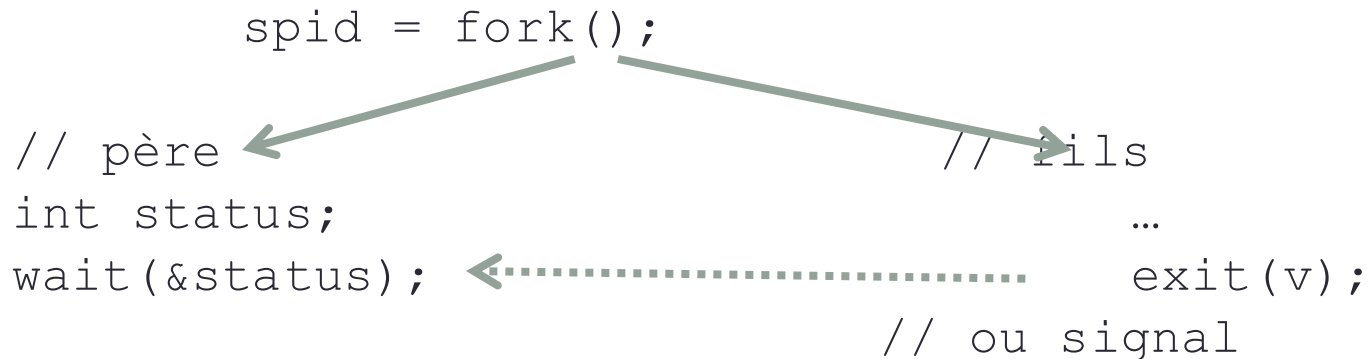
Systèmes centralisés : TD1

pid_t wait (int *status)

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

Attente **bloquante** (du père, suspension) de la fin d'un fils

```
        spid = fork();  
  
// père    ←-----→ // fils  
int status;  
wait(&status);    ...  
                  exit(v);  
                  // ou signal
```



❑ Macros

- ❑ WIFEXITED (status) est vrai si le fils s'est terminé avec exit
- ❑ WEXITSTATUS (status) renvoie la valeur du exit
- ❑ WIFSIGNALED (status) est vrai si le fils a été tué par un signal
- ❑ WTERMSIG (status) renvoie le numéro du signal ayant tué le fils

Systèmes centralisés : TD1

pere_fil_wait.c

```
// fils
// le dernier fils ne s'endort pas
if (fils < NB_FILS) { sleep(duree_sommeil); }
exit(fils); // pour illustrer WEXITSTATUS
}
else { ... }
}

// Après la boucle de création : Attendre la fin des fils
for (fils = 1; fils <= NB_FILS; fils++) {
    if ((fils_termine = (int) wait(&wstatus)) > 0) {
        if WIFEXITED(wstatus) {
            printf("\nMon fils de pid %d s'est arrete avec exit
                %d\n", fils_termine, WEXITSTATUS(wstatus));
        } else if WIFSIGNALED(wstatus) {
            printf("\nMon fils de pid %d a ete tue par le signal
                %d\n", fils_termine, WTERMSIG(wstatus));
        }
    }
}

printf("\nProcessus Principal termine\n"); return EXIT_SUCCESS; }
// A l'exécution, tuer fils1 et fils2 : kill -2 fils1 ; kill -9 fils2
```

Systèmes centralisés : TD1

➤ **exec** : **int execl(...), execv(...), execlp, execvp**

➤ famille de primitives permettant le lancement de l'exécution d'un programme externe (commande, exécutable)

➤ Le programme externe recouvre le processus appelant : pas de retour au processus appelant sauf en cas d'échec du exec

➤ **int execl** (char* ref, char* arg0, ..., NULL)

➤ **int execlp** (char* ref, char* arg0, ..., NULL) // p : path

➤ ref : référence de la commande = [chemin/]nom

➤ arg0 : nom symbolique, souvent = ref, mais pas forcément

➤ **int execv** (char *ref, char *argv[]) **int execvp** (,,)

Systèmes centralisés : API en langage C

test_execl.c

```
/* test de execl */
#include <stdio.h> /* entrées sorties */
#include <unistd.h> /* primitives de base : fork, ...*/
int main() {
    int retour;
    printf("Execution de la commande ls\n");
    retour = execl("ls", "ls", "-l", NULL);
    /* partie suivante exécutée seulement si le execl échoue */
    printf("Echec de l'exécution - Retour = %d\n", retour);

    printf("\nExecution de la commande ls avec chemin complet\n");
    retour = execl("/bin/ls", "ls", "-l", NULL); printf("Echec de
        l'exécution - Retour = %d\n", retour);
    return retour;
}
```

=> Compiler, exécuter et expliquer le résultat

Systèmes centralisés : API en langage C

test_execlp.c

```
/* test de execlp */
#include <stdio.h> /* entrées sorties */
#include <unistd.h> /* pimitives de base : fork, exec, ...*/
int main() {
    int retour;
    printf("\nExecution de la commande ls avec execlp\n");
    retour = execlp("ls", "ls", "-l", NULL);
    printf("Echec de l'exécution - Retour = %d\n", retour);

    printf("\nExecution de la commande ls avec chemin complet\n");
    retour = execl("/bin/ls", "ls", "-l", NULL); printf("Echec de
        l'exécution - Retour = %d\n", retour);
    return retour;
}
```

=> Compiler, exécuter et expliquer le résultat

Systèmes centralisés : API en langage C

test_execv.c

```
#include <stdio.h> /* entrées sorties */
#include <unistd.h> /* pimitives de base : fork, ...*/
int main(int argc, char *argv[]){
    int retour = 0;
    if (argc > 1) {
        printf("Execution de %s .. avec execv\n", argv[1]);
        retour = execv(argv[1], argv+1);
        printf("Echec de execv - Retour = %d\n",retour);
        printf("\nExecution de la commande avec execvp\n");
        retour = execvp(argv[1], argv+1);
        printf("Echec de execvp - Retour = %d\n", retour);
    } return retour;
}
```

./testexecv ls -l ⇒ ?

./testexecv /bin/ls -l ⇒ ?

Systèmes centralisés : TD1

Projet mini_shell

R1 : ?

Répéter

Afficher l'invite (prompt)

IN : invite chaîne

Lire une commande

OUT : commande tableau de chaînes

Exécuter la commande

IN : commande ; OUT : fin booléen

jusqu'À fin

R2 : Comment exécuter la commande ?

Si commande interne alors ...

sinon ?