COURS
UNIX

LA GESTION DES PROCESSUS

SOMMAIRE

1 La création de processus.

Le processus courant peut créer un processus fils en utilisant l'appel système fork (unistd.h). Cette primitive provoque la duplication du processus courant :

pid_t fork(void);

En cas d'échec de création de processus, fork retourne la valeur –1 et la variable errno est positionné suivant le type d'erreur survenue qui peut être :

- *le nombre maximal de processus pour l'utilisateur, ou dans le système, a été atteint.*

- *Le noyau n'a pas pu allouer suffisamment de mémoire pour créer un nouveau processus*

La création de processus se réalise en dupliquant le processus qui fait appel à la primitive fork.

Les deux processus disposent du même code programme, il est donc nécessaire de distinguer au retour de l'appel de fork si on se trouve dans le processus père ou dans le processus fils. La primitive fork, si aucune erreur ne s'est produite, retourne dans le fils la valeur 0 et dans le père le numéro du processus fils créé.

Processus PERE

```
#include …..

void       main
(void)
{
pid_t valeur;
……..
……..
valeur=fork()
printf("%d",va
leur):
……
}
```

Processus FILS

```
#include …..

void       main
(void)
{
pid_t valeur;
……..
……..
valeur=fork()
printf("%d",va
leur):
……
}
```

Lorsque le processus père appelle la fonction fork, le processus fils est créé, et les deux processus reprennent leur exécution au retour de l'appel système.

Ainsi, le père affiche la valeur retournée par fork soit la valeur:
Le fils affiche également la valeur retournée par fork, soit la valeur :

Il sera ainsi possible dans le source du programme de distinguer le traitement que doit effectuer le père, du traitement que doit réaliser le processus fils.

Exemple :

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

void main(void)
{
pid_t pid;
pid=fork();

switch(pid)
  {
  case (pid_t)-1:
        perror("Création de processus");
        exit(2);
  case (pid_t)0:
        /* on est dans le fils */
        printf("Je suis le processus %d de père %d \n",getpid(), getppid());
        printf("fin du processus fils\n");
        exit(0);
  default :
        /*on est dans le père*/
        printf(Je suis le processus %d de père %d \n",getpid(), getppid());
        printf("fin du processus père\n);
  }
}
```

Il est possible lors de l'exécution de ce code que les messages affichés soient mélangés. En effet le noyau peut élire le fils avant que le père soit terminé ou inversement.

Dans le cas où le père se termine complètement avant que le fils soit élu, l'affichage du PPID par le fils donnera le résultat 1. En effet, un processus orphelin est adopté par le processus de PID 1.

2 Les ressources d'un processus.

Lorsqu'un processus crée un processus fils, un certain nombre de caractéristiques du fils sont héritées du père :

- les numéros réels et effectifs de l'utilisateur.
- le groupe réel et effectif de l'utilisateur.
- la priorité
- le répertoire courant.
- Les numéros de groupe et de session.

Par contre le processus fils n'hérite pas :

- De la valeur du PID.
- De la valeur du PPID.
- Des temps d'exécution (remis à zéro pour le fils).

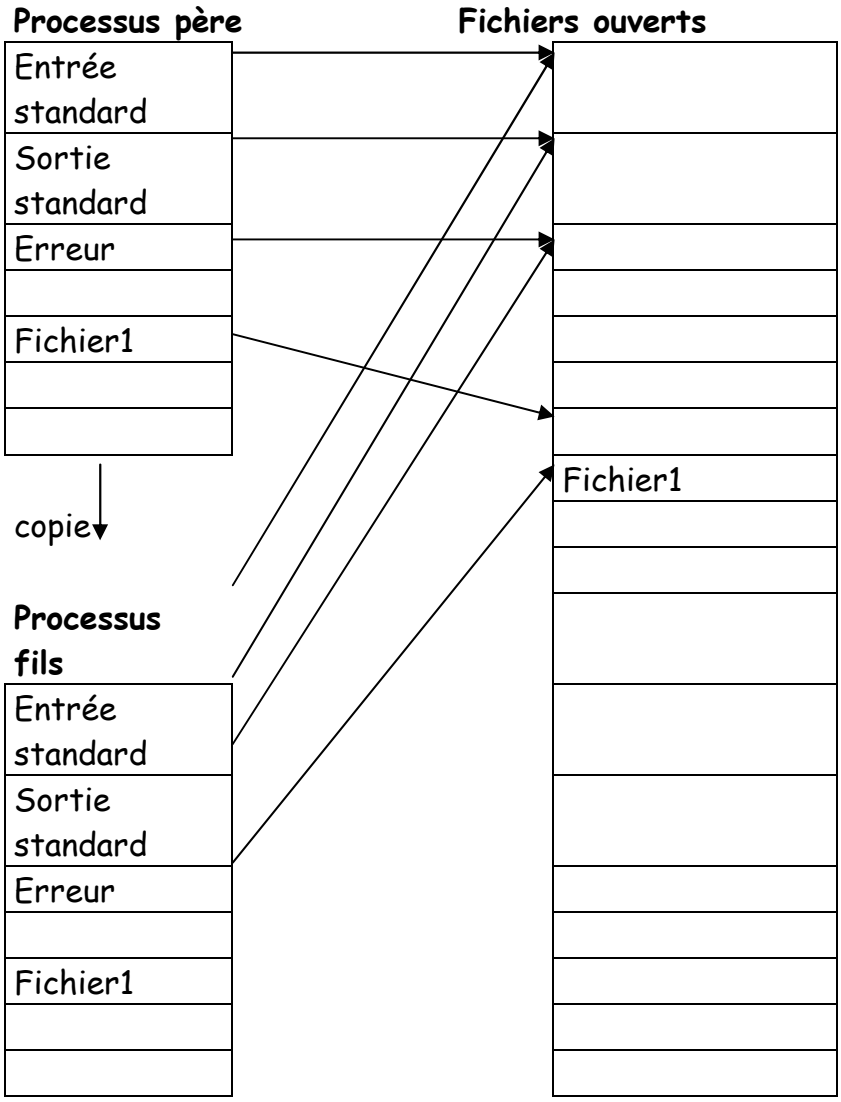
Le processus fils est pratiquement une copie conforme du processus parent :

- Les segments de données, de code et de pile du fils sont une copie des segments du père.
- La table des descripteurs de fichiers du fils est copie de celle du père.

Conséquences :

- Etant donné que le fils travaille sur une copie des données du processus père, toutes les modifications d'un objet par l'un des deux processus ne sont pas visibles par l'autre.

- étant donné que la table des descripteurs de fichiers du processus fils est une copie de la table du processus père, pour chaque fichier ouvert, le compteur de référence est incrémenté dans la table des fichiers ouverts. Ainsi les deux processus ont accès aux mêmes fichiers (ouverts avant le fork), et ils ont les mêmes offsets dans ces fichiers.


Processus père
Fichiers ouverts
Entrée standard
Sortie standard
Erreur
Fichier1
copie
Fichier1
Processus fils
Entrée standard
Sortie standard
Erreur
Fichier1

3 Synchronisation père/fils.

Nous avons déjà vu que lorsqu'un processus se termine, il passe à l'état zombie, jusqu'à ce que son père prenne connaissance de sa terminaison.

Si un processus père crée un processus fils qui se termine avant lui, le processus fils passera à l'état zombie jusqu'à la terminaison du père.

Le processus père doit donc prendre connaissance de la mort d'un fils.

De même, si le père se termine avant un fils, le fils est adopté par le processus 1.

3.1 La primitive wait.

La primitive système **wait** (sys/wait.h et sys/types.h) permet à un père de prendre connaissance de la fin d'un processus fils ou d'attendre la mort de celui-ci.

pid_t wait(int *status);

L'appel wait permet de :

- si le processus n'a pas de fils, wait retourne –1.
- si le processus a un fils ou plusieurs fils zombie, wait retourne le PID d'un des fils, et place à l'adresse de status l'état de terminaison du fils. Le processus fils est alors complètement détruit.
- Si le processus a un ou plusieurs fils en cours d'exécution, le processus père est bloqué jusqu'à la mort d'un des fils, wait retourne alors le PID du fils et place à l'adresse de status l'état de terminaison du fils.

3.2 La primitive waitpid.

La primitive waitpid suspend l'exécution du processus courant jusqu'à ce qu'un processus spécifié par le paramètre pid se termine.

pid_t waitpid(pid_t pid, int *status, int option);

- si pid est positif, il spécifie le numéro du processus fils à attendre.
- Si pid est nul, il spécifie tout processus fils du même groupe que le processus appelant.
- Si pid est égal à –1, il spécifie le premier fils à se terminer (wait).
- Si pid est inférieur à –1, il spécifie tout processus fils dont le numéro de groupe est égal à la valeur absolue de pid.

L'option, si elle est mentionnée peut prendre deux valeurs (ou logique):

WNOHANG : le processus appelant n'est pas bloqué si le processus demandé n'est pas terminé.

WUNTRACED : permet de prendre en compte les changements d'état d'un processus.

La fonction waitpid retourne :
- –1 en cas d'échec.
- 0 si en mode non bloquant si le processus n'est pas terminé.
- Sinon le numéro du processus fils terminé.

Et positionne à l'adresse de status l'état de terminaison du processus pris en compte.

3.3 Prise en compte de l'état de terminaison.

La variable placée à l'adresse status peut être consultée pour connaître l'état de terminaison d'un processus.

L'interprétation de cet état est effectuée grâce à des macro-instructions (sys/wait.h) :

WIFEXITED (status) : retourne une valeur non nulle si le processus s'est terminé normalement (return ou exit).

WEXITSTATUS (status) : retourne le code de retour du processus lors de sa terminaison.

WIFSIGNALED (status) : retourne une valeur non nulle si le processus fils a été interrompu par la réception d'un signal.

WTERMSIG (status) : retourne le numéro du signal ayant provoqué la terminaison du processus fils

WIFSTOPPED (status) : retourne une valeur non nulle si le processus est suspendu.

WSTOPSIG (status) : retourne le signal ayant causé la suspension du processus fils.

4 Les primitives de recouvrement.

Un nouveau processus, créé par un appel à la primitive fork, est une copie conforme de son processus père, et exécute donc le même programme (même fichier source). Les primitives de recouvrement permettent à un processus d'exécuter un nouveau programme (nouveau fichier source).

La primitive **execve** (unistd.h) provoque l'exécution d'un nouveau programme.

int execve (const char *pathname, const char *argv[], const char *envp[]);

- **pathname** spécifie le nom du fichier à exécuter, qui doit être un programme binaire ou un script shell.

- **argv** indique les arguments du programme à exécuter : chaque élément du tableau doit pointer sur une chaîne de caractères représentant un argument. Le premier élément du tableau doit contenir le nom du programme. Le dernier élément doit contenir la valeur NULL

- **envp** spécifie les variables d'environnement du programme. Chacun des éléments doit contenir l'adresse d'une chaîne de caractères de la forme variable=valeur, et le dernier élément doit contenir la valeur NULL.

L'appel système execve provoque le recouvrement des segments de code, de données et de pile du processus courant par ceux du programme appelé. En cas de succès, il n'y a donc pas de retour.
Plusieurs fonctions offrent des variantes de execve :

int execl (const char *pathname, const char *arg, ..,NULL);
int execle (const char *pathname, const char *arg,..,NULL, const char * envp[]);

Pour ces deux fonctions les arguments des programmes sont directement indiqués.

int execv (const char *pathname, const char * argv[]);

Cette fonction est similaire à execve sans les variables d'environnement.

Les fonctions **execlp** et **execvp** sont similaires aux fonctions **execl** et **execv** sauf que le programme est recherché suivant la variable d'environnement PATH.

LES ENTRAILLES DES PROCESSUS SOUS UNIX

Le système d'exploitation gère les transitions entre les états. Pour ce faire il maintient une liste de processus éligibles et une liste de processus en attente. Il doit également avoir une politique d'activation des processus éligibles (ou politique d'allocation du processeur aux processus) : parmi les processus éligibles,faut-il activer celui qui est éligible depuis le plus de temps, celui qui aurait la plus forte priorité ou celui qui demande l'unité centrale pour le temps le plus court (à supposer que cette information soit disponible)?

Les processus sont classés dans la file des processus éligibles par **l'*ordonnanceur*** (en anglais ***scheduler***). C'est le *distributeur* (en anglais *dispatcher*) qui se chargera de leur activation au moment voulu.

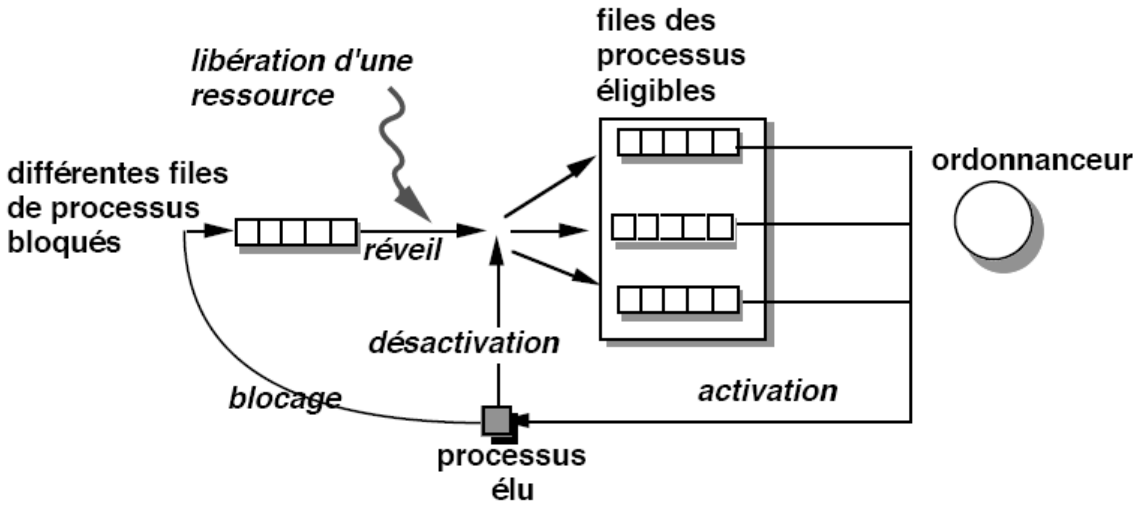
Afin d'éviter qu'un processus accapare l'unité centrale, le système d'exploitation déclenche un temporisateur à chaque fois qu'il alloue l'unité centrale à un processus. Quand ce temporisateur expire, si le processus occupe toujours l'unité centrale (il peut avoir été interrompu par un événement extérieur ou s'être mis en attente), le système d'exploitation va le faire passer dans l'état prêt et activer un autre processus de la liste des processus éligibles. La désactivation d'un processus peut se faire sur expiration du temporisateur ou sur réquisition (en anglais **preemption**) du processeur central par un autre processus.

Un temporisateur peut être vu comme un système de compte à rebours ou de sablier. On lui donne une valeur initiale qu'il décrémente au rythme de l'horloge de l'ordinateur jusqu'à atteindre la valeur zéro. Ici, la valeur initiale du temporisateur est le **quantum** de temps alloué à l'exécution du processus. A l'expiration du temporisateur le système d'exploitation est immédiatement prévenu par une *interruption*.

D'autres événements sont signalés au système par des interruptions : c'est le cas des fins d'entrées-sorties. Le système est ainsi prévenu que le ou les processus qui étaient en attente sur cette entrée-sortie peuvent éventuellement changer d'état (être réveillés).

Ordonnancement

Dans un système multitâches, le système d'exploitation doit gérer l'allocation du processeur aux processus. On parle d'ordonnancement des processus.



Il existe deux *contextes* :

Le contexte *matériel* et le contexte *logiciel*. Le contexte matériel est la photographie de l'état des registres à un instant *t* : compteur ordinal, registre d'état, etc.

Le contexte logiciel contient des informations sur les conditions et droits d'accès aux ressources de la machine!: priorité de base, quotas sur les différentes ressources (nombre de fichiers ouverts, espaces disponibles en mémoires virtuelle et physique...).

Chaque fois que le processeur passe à l'exécution d'un nouveau processus il doit changer les configurations de ses registres. On parle de *changement de contexte*. Certaines informations (une partie du contexte logiciel) concernant les processus doivent rester en permanence en mémoire dans l'espace système. D'autres peuvent être transférées sur disque avec le processus quand celui ci doit passer de la mémoire au disque (quand il n'y a plus de place en mémoire pour y laisser tous les processus, par exemple). Il s'agit en général de tout le contexte matériel et d'une partie du contexte logiciel.

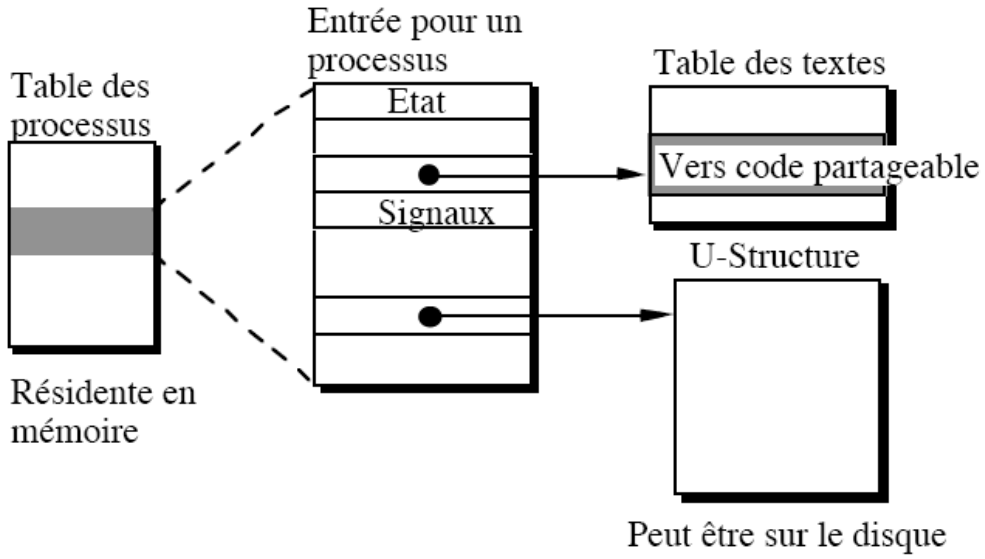
Le *bloc de contrôle d'un processus* (en anglais *Process Control Block* ou *PCB*) est une structure de données qui contient toutes les informations relatives au contexte d'un processus. Quand un processus est désactivé ou bloqué, toutes les information relatives à son contexte son sauvegardées dans son bloc de contrôle. Quand un processus est activé, on restaure son contexte à partir de son bloc de contrôle.

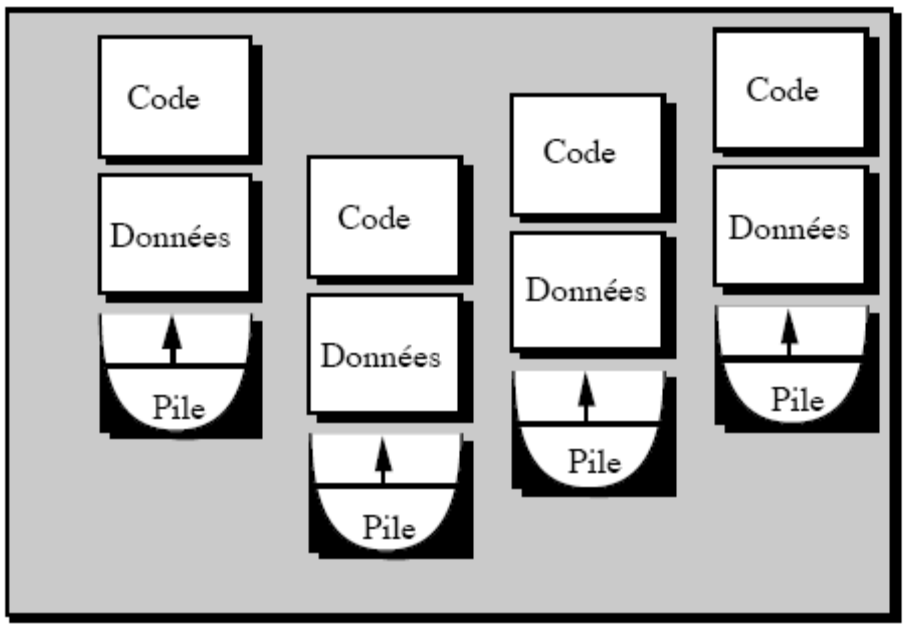
Les processus sont décrits dans une table des processus, une entrée decette table donne des informations sur l'état du processus et permet de retrouver les 3 régions : *code*, *data* et *stack*. Seules les informations dont le système a besoin en permanence sont dans la table des processus, les autres sont dans la *u-structure* qui peut être éventuellement passer sur le disque si nécessaire.

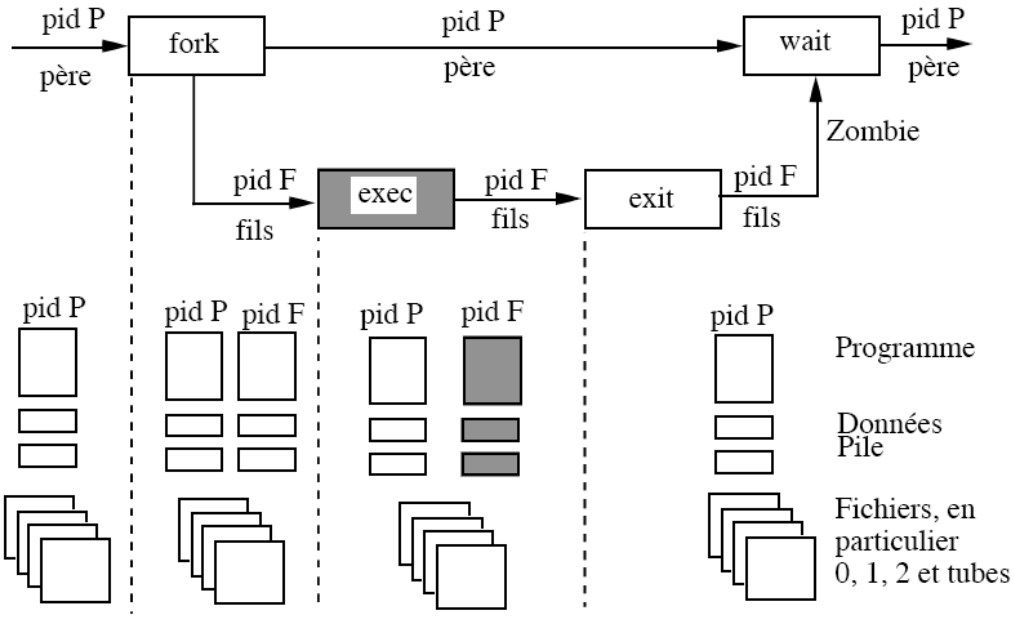
Unix alloue 3 régions en mémoire pour chaque processus:
• une région pour le code exécutable (text segment),
• une région pour la pile (stack segment),
• une région pour les données (data segment).

Unix mémorise pour chaque processus des informations de gestion du type:
• identificateur du processus (pid)
• configuration des registres (pile, compteur ordinal),
• répertoire courant, fichiers ouverts,

Remarque:
Le pointeur vers la table des *textes* (fichiers exécutables) donne, non pas le début du programme à exécuter mais un pointeur vers ce programme, ce qui permet à la fois le partage de code exécutable par plusieurs processus différents et le déplacement de celui-ci par le système.

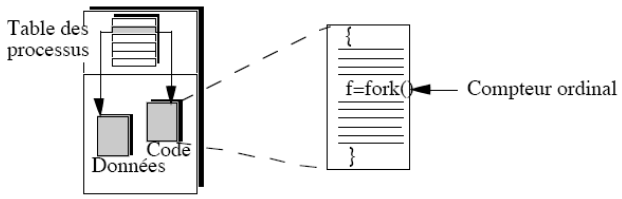




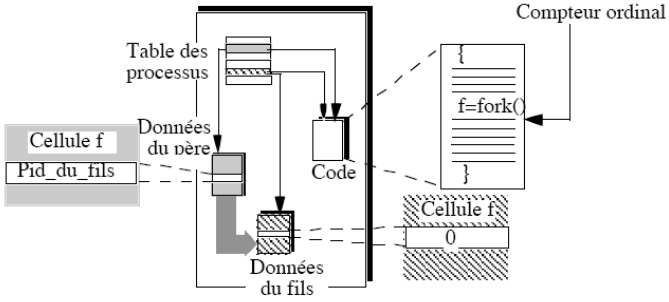


Remarque : le segment de code n'est pas dupliqué, mais partagé : deux po
inteurs différents désignent le même code exécutable.

• Avant fork :



• Après fork :



PRIORITE DES PROCESSUS

Un processus utilisateur démarre avec une priorité de base PBase (PBase=60). Toutes les secondes le système réévalue cette priorité et la pondère de façon à favoriser les processus interactifs, c'est à dire ceux qui font beaucoup d'entrées-sorties. Le mécanisme de pondération est décrit ci-dessous.

À chaque processus est associé un coefficient C dont la valeur est fonction du temps passé dans le processeur la dernière fois qu'il a été actif. C est initialisé ainsi!:
C= temps cpu consommé
La valeur C est recalculée toutes les secondes de la façon suivante:
C = C /2
et on attribue au processus la priorité P suivante, après avoir recalculé C:
P = PBase + C
Le processus peut modifier sa priorité en ajustant le paramètre nice, sa priorité deviendra:
P = PBase + C + nice

TRAVAUX PARTIQUES
GESTION DES PROCESSUS

1. Ecrire un programme qui ouvre deux fichiers, l'un en lecture, l'autre en écriture. Le processus père crée un processus fils, et père et fils recopient concurremment, caractère par caractère, le premier fichier dans le second.

2. Ecrire un programme qui permet de créer deux processus (un père et un fils). Le fils exécutera une temporisation (sleep) et se terminera normalement (exit) tandis le père devra attendre la terminaison du fils et récupérer la valeur de terminaison du fils.

3. Ecrire un programme similaire, mais le fils effectuant une boucle infinie. Envoyer un signal (commande kill) au processus fils, le père récupérant le numéro du signal.

4. Le but de ce TP est de concevoir en C une commande parexec qui prend en argument en ligne de commande un nom de programme prog suivi d'une liste arbitrairement longue d'arguments, et qui exécute prog en parallèle sur chacun des arguments. Par exemple, la commande:

./parexec md5sum fichier1 fichier2 ... fichierN

lancera en parallèle les commandes md5sum fichier1, md5sum fichier2, ..., md5sum fichierN.

a. Programmez parexec à l'aide de fork() et execlp(). parexec ne doit rendre la main que quand toutes les exécutions de prog se sont terminées. Que se passe-t-il si on tape Contrôle+C?

   i. Dans une boucle d'indice i de 2 à argc-1, on fera un fork. Si cette fonction renvoie 0, cela signifie qu'on est dans le fils. On fera alors un execlp(argv[1], argv[1], argv[i], NULL) qui exécute prog avec l'argument argv[i] en ligne de commande. Attention: argv[1] apparaît deux fois dans execlp (une fois en tant que programme à exécuter et une fois en tant qu'argument 0 du programme à exécuter). De plus la liste d'arguments doit obligatoirement se terminer par NULL.

   ii. Pour attendre que tous les fils se terminent, il suffit d'appeler argc-2 fois wait.

   iii. wait peut échouer avec l'erreur EINTR, auquel cas il faut renouveler l'appel.

iv. Structure du programme :

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>

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

/* test nombre argument */
if (argc <  )
{
        fprintf(stderr, "%s prog fichiers\n", argv[0]);
        return 1;
 }

  /* lancement des processus */
  for ( i= ;       ;     )
{

    pid_t p = fork();

    if ( p < 0 )
    {


     }

    if ( !p )
     {
      /* fils */
      execlp(                                  );

      /* on n'arrive ici que si execlp a Ã©chouÃ© */


    }

    /* pere */
  }
```

```
 /* attente de la terminaison des fils */
  for ( i=                )
    {
    int status;
    /* boucle jusqu'a  ce que wait reussisse */
    while ( wait(&status) == -1)
          {
      if (errno != EINTR)
         {
           /* vraie erreur */
           perror("wait");
           return 1;
         }
     }
  }

  return 0;
}
```

b. Modifiez parexec pour qu'il prenne un argument supplémentaire entre prog et fichier1: le nombre maximum d'instances de prog à lancer en parallèle. Quand ce nombre est atteint, parexec doit attendre qu'un des appels se termine avant d'en lancer un nouveau.

i. Écrivez un petit programme compte qui effectue un compte à rebours seconde par seconde à partir d'un nombre passé en argument. Sur chaque ligne, compte affichera le nombre de secondes restantes ainsi que son pid (afin de l'identifier facilement en cas d'exécutions multiples). Une commande telle que ./parexec ./compte 2 4 5 6 7 devrait alors vous permettre de tester facilement votre nouveau parexec. Pour compte, vous aurez besoin des fonctions sleep et getpid.

Activité supplémentaire

Lorsque le shell reçoit le nom d'une commande à exécuter en premier plan, il se duplique (création d'un processus fils), le père attend la terminaison de son fils. Le fils est quant à lui chargé d'exécuter la commande.

On se propose d'écrire un shell minimum capable de lancer des commandes en premier plan :

1. Ecrire un programme « mon_shell » qui attend l'entrée de commandes (censées exister), pour chaque commande entrée, le processus se duplique et son fils exécute la commande en utilisant les primitives de recouvrement, le processus père (mon_shell) attendant la fin de l'exécution de la commande.

Nous allons modifier le programme précédent pour que notre shell identifie les redirections, pour qu'il puisse exécuter des commandes du type ***cat fichier1 > fichier2***.

2. Inclure une fonction dans le programme précédent permettant de déterminer le nom de la commande, le nom du fichier1 et du fichier2.

Pour pouvoir effectuer la redirection au sein du processus fils, la méthode consiste à :
- ouvrir le fichier2 dans le bon mode
- rediriger la sortie standard du fils vers ce fichier
- exécuter la commande par une primitive de recouvrement.

3. Ecrire le code du fils permettant de réaliser les différentes étapes.

4. Modifier le programme pour qu'il puisse réaliser des redirections du type ***cat fichier1 >> fichier2***.