Systèmes d'exploitation

Chapitre 2: Gestion des processus

Définition:

- Un processus est l'activité résultante de l'exécution d'un programme séquentiel, avec :
  - Ses données et sa pile d'exécution,
  - Son compteur ordinal (contrôleur de flux),
  - Son pointeur de pile (Stack Pointer) et les autres registres

  par un processeur.

A quoi sert un processus ?

- Il sert à faire plusieurs activités au même temps, par exemple : faire travailler plusieurs utilisateurs sur la même machine, chaque processus pense que la machine est lui affectée tout seul.

Exemple: Compiler tout en lisant son mail.

Problème: Un processeur ne peut exécuter qu'une seule instruction à la fois.

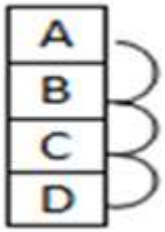
But : Partager un (ou plusieurs) processeur(s) entre plusieurs processus.

***Remarques*:**

- Ne pas confondre processus et processeur.
- Conceptuellement chaque processus a son propre processeur virtuel.
- En réalité, le vrai processeur bascule (commute) constamment d'un processus a l'autre.

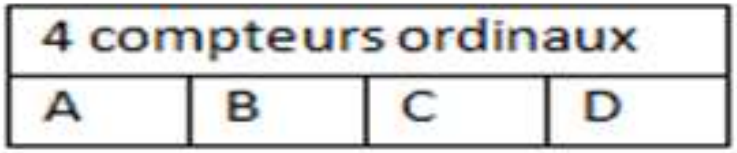
- **Exemples :**

a) 4 processus A, B, C, D avec un seul compteur ordinal bascule entre les processus



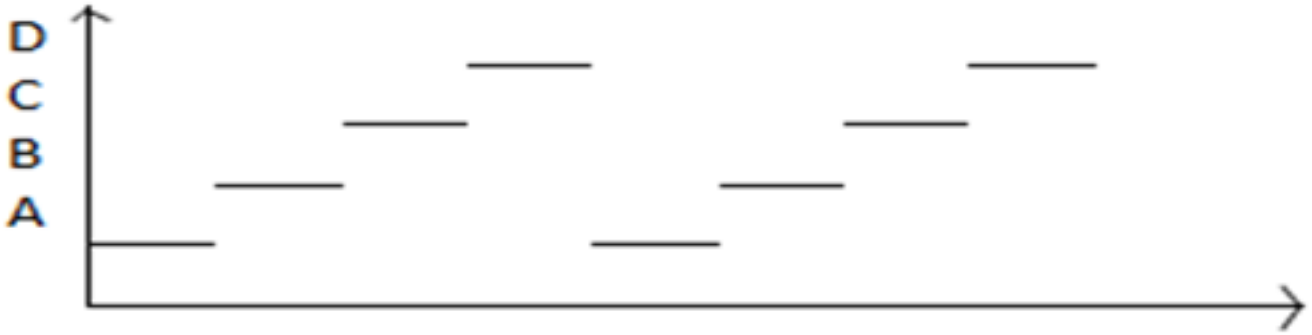
le processeur gère les 4 programmes : c'est la multiprogrammation.

b)



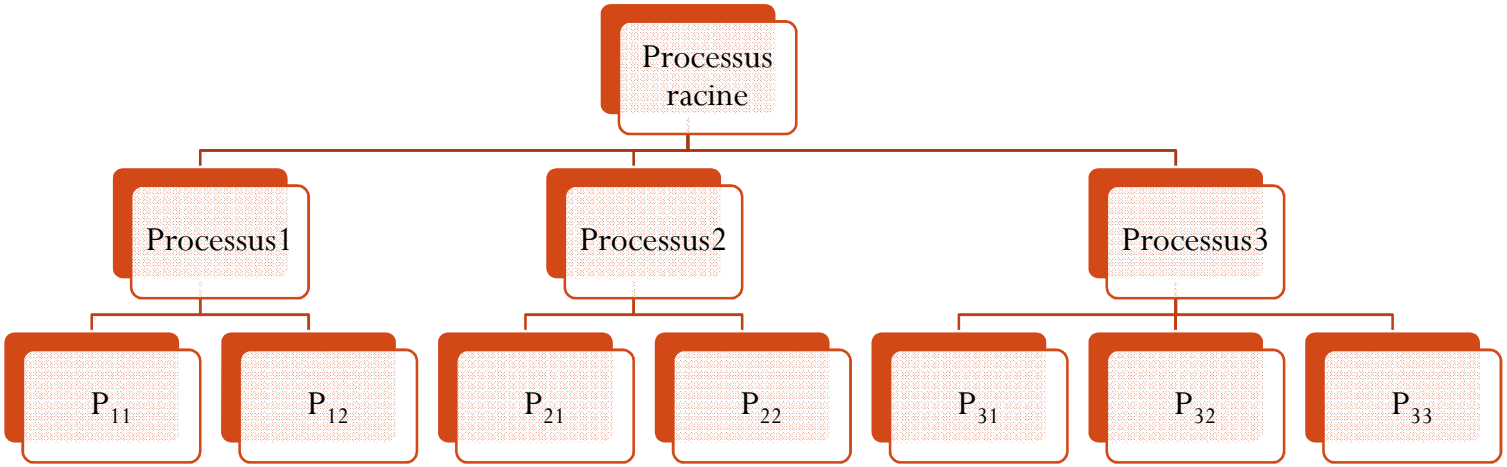
une abstraction de cette situation : 4 processus indépendants chacun avec son propre compteur ordinal.

c) Lors de l'exécution, on constate que sur un intervalle de temps assez grand tous les processus ont progressé mais qu'à un instant donné il n y a qu'un seul processus actif.



Hiérarchie des processus :

Les processus se reproduisent c-à-d un processus peut créer un ou plusieurs processus fils, on parle d'arbre hiérarchique des processus.



Remarques :

- Sous UNIX, les processus sont créés par l'appel système *fork()* qui permet de créer une copie conforme des processus appelant, le père et le fils s'exécutent en parallèle (Multitâches).

- Sous MS-DOS, le fils suspend le père (Monotâtche).

Exemple: Création des processus fils par fork()

```
main (int argc,char* argv[])
{
int pid ;
printf("%d",getpid()) ;
pid=fork();
if(pid==0)
{printf("%d",getpid());
 printf("%d",getppid());
}
else
{
printf("%d",getpid());
printf("%d",pid);
 }
}
```

Sous programme SP1

Sous programme SP2

Question:

Si la valeur de printf1 est 254 et la valeur de printf5 est 255, déduire les valeurs des autres printf.

Remarque :

fork() retourne un entier:
en cas de succès:
- 0 dans le fils.
- pid du fils dans le père.

en cas d'échec
- -1 dans le père.
- le fils n'est pas créé.

Application :

Créer trois processus:

- Chaque processus affiche son pid et le pid de son père,
- Le premier processus affiche la somme des entiers donnés comme arguments de la ligne commande,
- Le deuxième processus affiche la valeur maximale de ces éléments,
- Le troisième processus affiche la valeur minimale de ces éléments,

Types d'un processus :

1. Processus utilisateurs

Ce sont des processus qui sont lancés par un utilisateur (compilation, exécution, jeu, impression…).

2. Processus systèmes

Ce sont les processus lancés par le système d'exploitation (SE) pour la gestion des systèmes au démarrage de la machine (spouleur d'impression, surveillance des ports…).

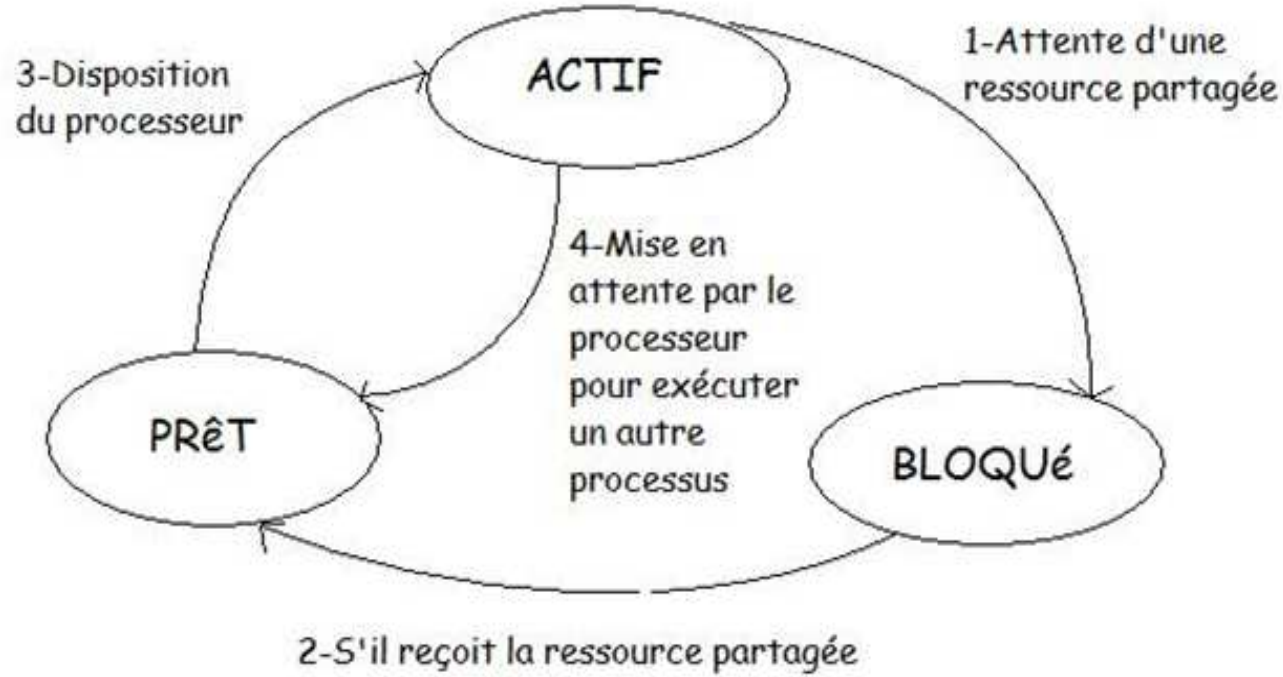
Etats d'un processus :

Actif : Si le processus dispose du processeur.

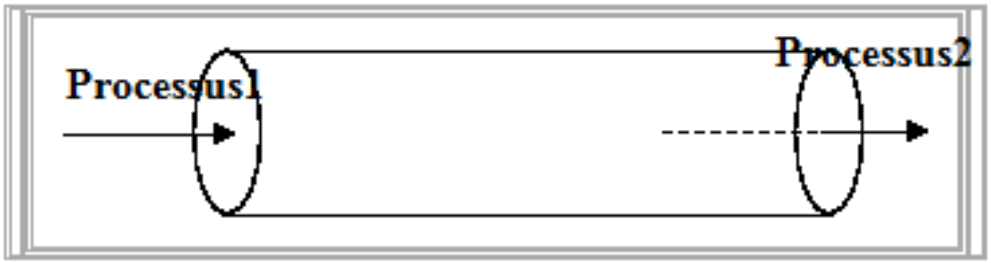
Bloqué : Si le processus attend une ressource partagée.

Prêt : Si le processus n'attend pas une ressource partagée et s'il ne dispose pas du processeur.

Diagramme de transition des états des processus:



Communication entre processus (tubes de communication)



Tube de communication

Un *tube* est un mécanisme de communication unidirectionnel entre deux processus.

Les tubes sont très pratiques car ils permettent de gérer à la fois les problèmes de **communication** (à chaque tube est associée une mémoire tampon à laquelle les deux processus vont accéder, l'un en écriture, l'autre en lecture), et les problèmes de **synchronisation** (la lecture dans un tube est bloquante jusqu'à ce que les données soit arrivées).

Création d'un tube de communication

- Les tubes Unix permettent de relier deux processus issus d'un même ancêtre, qui doit créer le tube à l'aide de l'appel système **pipe().**
- L'appel **pipe()** retourne deux descripteurs d'entrées/sorties, l'un pour écrire (l'entrée du tube), l'autre pour lire (la sortie du tube).
- On peut ensuite créer un ou plusieurs processus fils à l'aide de l'appel système **fork().**
- Les processus fils héritent alors des descripteurs et peuvent donc utiliser le tube.

En résumé :

- Prototype de pipe() est: int pipe(int fd[2])
- Elle créé un tube formé de deux descripteurs fd[0] et fd[1]
- Elle retourne 0 en cas de réussite
- Au programmeur de décider quelle est l'entrée et la sortie du tube
- Ensuite en général on créée des fils
- Par exemple un fils écrit dans fd[0] avec write :

  write(fd[0],bufentree,nbre_octets)

- L'autre lit les valeurs transmises avec read :

  read(fd[1],bufsortie,nbre_octets);

Problème : Comment savoir qu'il n'y a plus rien à lire ?

Exemple de communication entre deux processus:

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


int  main(int argc, char *argv[])
{
   int pipefd[2];
   int  cpid;
   char buf;


   assert(argc == 2);
```

```
if (pipe(pipefd) == -1)
        {
           perror("pipe");
           exit(EXIT_FAILURE);
         }

        cpid = fork();
        if (cpid == -1) {
           perror("fork");
           exit(EXIT_FAILURE);
        }

         if (cpid == 0)
{   /* Child reads from pipe */
           close(pipefd[1]);         /* Close unused write end */
          while (read(pipefd[0], &buf, 1) > 0)
             write(STDOUT_FILENO, &buf, 1);
             write(STDOUT_FILENO, "\n", 1);
           close(pipefd[0]);
           exit(EXIT_SUCCESS);
       }
```

```
else
 {          /* Parent writes argv[1] to pipe */
            close(pipefd[0]);       /* Close unused read end */
            write(pipefd[1], argv[1], strlen(argv[1]));
            close(pipefd[1]);          /* Reader will see EOF */
            wait(NULL);                /* Wait for child */
            exit(EXIT_SUCCESS);
           }
                }
```

Les Threads

- Les threads *(appelés aussi « processus légers »*, comme les processus, sont des mécanismes permettant à un programme de faire plus d'une chose à la fois.

- Comme les processus, les threads semblent s'exécuter en parallèle; le noyau Linux les ordonnance de façon asynchrone, interrompant chaque thread de temps en temps pour donner aux autres une chance de s'exécuter.

- Conceptuellement, un thread existe au sein d'un processus. Les threads sont une unité d'exécution plus fine que les processus.

Les Threads

- Nous avons vu comment un programme peut créer un processus fils.
- Celui-ci exécute immédiatement le programme de son père, la mémoire virtuelle, les descripteurs de fichiers, *etc*. de son père étant copiés.
- Le processus fils peut modifier sa mémoire, fermer les descripteurs de fichiers sans que cela affecte son père, et *vice versa*.
- Lorsqu'un programme crée un nouveau thread, par contre, rien n'est copié **:** le thread créateur et le thread créé partagent tous deux le même espace mémoire, les mêmes descripteurs de fichiers et autres ressources.

Les Threads

- Si un thread modifie la valeur d'une variable, par exemple, l'autre thread verra la valeur modifiée.
- De même, si un thread ferme un descripteur de fichier, les autres threads ne peuvent plus lire ou écrire dans ce fichier.
- Comme un processus et tous ses threads ne peuvent exécuter qu'un seul programme à la fois, si un thread au sein d'un processus appelle une des fonctions *exec*, tous les autres threads se terminent (le nouveau programme peut, bien sûr, créer de nouveaux threads).

Création de Threads

- Chaque thread d'un processus est caractérisé par un ***identifiant de thread***. Les identifiants de threads dans des programmes C ou C++ sont de type ***pthread_t*.**

- Lors de sa création, chaque thread exécute une ***fonction de thread***. Il s'agit d'une fonction ordinaire contenant le code que doit exécuter le thread. Lorsque la fonction se termine, le thread se termine également.

- Sous GNU/Linux, les fonctions de thread ne prennent qu'un seul paramètre de type *void** et ont un type de retour *void**. Ce paramètre est **l'*argument de thread***: GNU/Linux passe sa valeur au thread sans y toucher.

Création de Threads

- Un programme peut utiliser ce paramètre pour passer des données à un nouveau thread. De même, il peut utiliser la valeur de retour pour faire en sorte que le thread renvoie des données à son créateur lorsqu'il se termine.

- La fonction ***pthread_create*** crée un nouveau thread. Voici les paramètres dont elle a besoin:

  - Un pointeur vers une variable *pthread_t*, dans laquelle l'identifiant du nouveau thread sera stocké;

Création de Threads

- Un pointeur vers un objet d'attribut de thread. Cet objet contrôle les détails de l'interaction du thread avec le reste du programme. Si vous passez *NULL* comme argument de thread, le thread est créé avec les attributs par défaut.

- Un pointeur vers la fonction de thread. Il s'agit d'un pointeur de fonction ordinaire de type: *void* (*) (void*)*;

- item Une valeur d'argument de thread de type *void**. Quoi que vous passiez, l'argument est simplement transmis à la fonction de thread lorsque celui-ci commence à s'exécuter.

Création de Threads

- Un appel à *pthread_create* se termine immédiatement et le thread original continue à exécuter l'instruction suivant l'appel. Pendant ce temps, le nouveau thread débute l'exécution de la fonction de thread.

- Linux ordonnance les deux threads de manière asynchrone et le programme ne doit pas faire d'hypothèse sur l'ordre d'exécution relatif des instructions dans les deux threads.

- L'exemple ***threadcreate*** crée un thread qui affiche ***x*** de façon continue sur la sortie des erreurs. Après l'appel de *pthread_create*, le thread principal affiche des ***o*** indéfiniment sur la sortie des erreurs.

Création de Threads (Exemple 1)

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

/* Affiche des x sur stderr.  Paramètre inutilisé. Ne finit jamais. */
void* print_xs (void* unused)
{
  while (1)
    fputc ('x', stderr);
  return NULL;
}
/* Le programme principal.  */
int main ()
{
  pthread_t thread_id;
  /* Crée un nouveau thread. Le nouveau thread exécutera la fonction print_xs. */
  pthread_create (&thread_id, NULL, &print_xs, NULL);
  /* Affiche des ‘o’ en continue sur stderr. */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

```c
#include <pthread.h>
#include <stdio.h>
/* Paramètres de la fonction print.   */
struct char_print_parms
{
   /* Caractère à afficher. */
   char character;
   /* Nombre de fois où il doit être affiché. */
   int count;
};
/* Affiche un certain nombre de caractères sur stderr, selon le contenu de PARAMETERS, qui est un
pointeur vers une struct char_print_parms. */
void* char_print (void* parameters)
{
   /* Effectue un transtypage du pointeur void vers le bon type. */
   struct char_print_parms* p = (struct char_print_parms*) parameters;
   int i;
   for (i = 0; i < p->count; ++i)
     fputc (p->character, stderr);
   return NULL;
}
/* Programme principal.   */
int main ()
{
   pthread_t thread1_id;
   pthread_t thread2_id;
   struct char_print_parms thread1_args;
   struct char_print_parms thread2_args;
   /* Crée un nouveau thread affichant 30 000 x. */
   thread1_args.character = 'x';
   thread1_args.count = 30000;
   pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
   /* Crée un nouveau thread affichant 20 000 'o'. */
   thread2_args.character = ‘o’;
   thread2_args.count = 20000;
   pthread_create (&thread2_id, NULL, &char_print, &thread2_args);
   return 0;
}
```