Les processus (Chapitre 2)

Francesco Bronzino
ArchiSys

Sujets

1. Concept de processus
2. États d'un processus
3. Bloc de contrôle du processus
4. Ordonnancement
5. Algorithmes d'ordonnancement

Concept de processus

- Un ordinateur exécute une ou plusieurs tâches
  - Exemple : Vérifier la température d'une pièce
- Une tâche est exécutée au moyen d'un processus formel appelé **algorithme**.
- Un programme met en œuvre un algorithme au moyen d'instructions en langage machine
  - Il peut être écrit dans un langage de programmation et *compilé*.
- Un **processus** est un programme en cours d'exécution

Concept de processus

Initialement, chaque ordinateur exécutait un programme à la fois.

- Chargé au démarrage du système
- Ou exécuté séquentiellement (traitement par lots)

Les systèmes modernes sont dotés d'un système d'exploitation qui permet l'exécution simultanée de plusieurs processus.

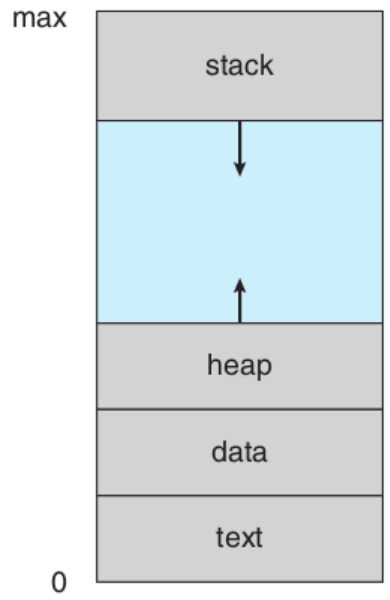
- Les ressources du système sont gérées par le système d'exploitation, qui les met à disposition par le biais d'appels système.
- Le système d'exploitation gère l'exécution des processus : **scheduling**.

Concept de processus

Un processus réside **en mémoire**

La structure d'un processus en mémoire est généralement divisée en plusieurs sections.

- **Section texte** : contient le code exécutable
- **Section des données** : contient les variables globales
- **Heap** : mémoire allouée dynamiquement au cours de l'exécution du programme
- **Stack** : mémoire utilisée temporairement lors des appels de fonctions



Processus *init*

Dans les systèmes d'exploitation modernes, un processus fondamental appelé **init** est lancé au démarrage du système.

- Il est exécuté jusqu'à l'arrêt du système

Le processus *init* démarre d'autres processus en arrière-plan :

- pour gérer les périphériques : réseau, antivirus
- Pour créer l'interface graphique ou l'interface graphique du terminal

Un processus démarré en arrière-plan par *init* est appelé **Service**.

- Les formats et les commandes pour les gérer diffèrent d'une distribution Linux à l'autre
  Commandes : `service` ou `systemctl`

Processus utilisateur

Les utilisateurs peuvent créer des processus pour effectuer leurs tâches

- Navigateur
- Éditeurs
- Programmes de serveur : serveur Web, serveur DNS

Concept de processus

Le système d'exploitation fournit des appels système pour :

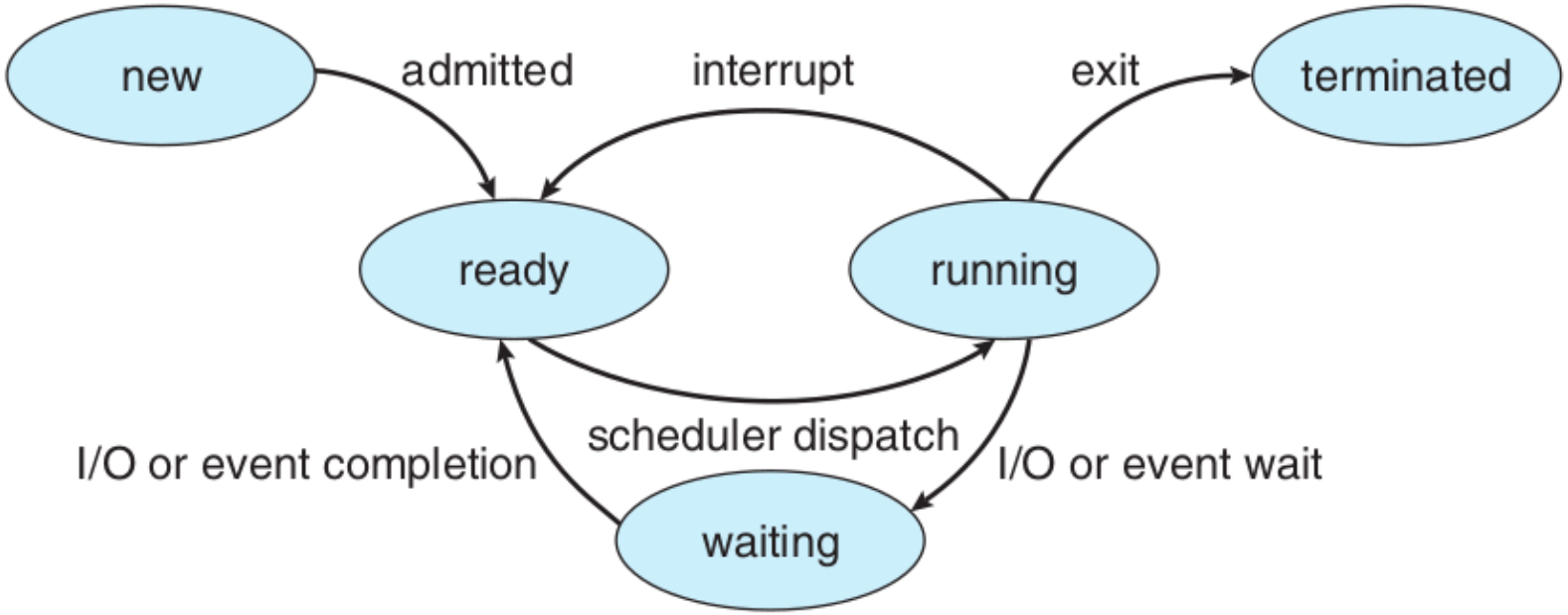
- Créer de nouveaux processus
- **Synchronisation :** Attendre que d'autres processus complete une tâche complexe.

Les processus sont identifiés par un **PID**.

- Le processus init a le PID 1 par définition

États d'un processus

Un processus peut se trouver dans plusieurs états



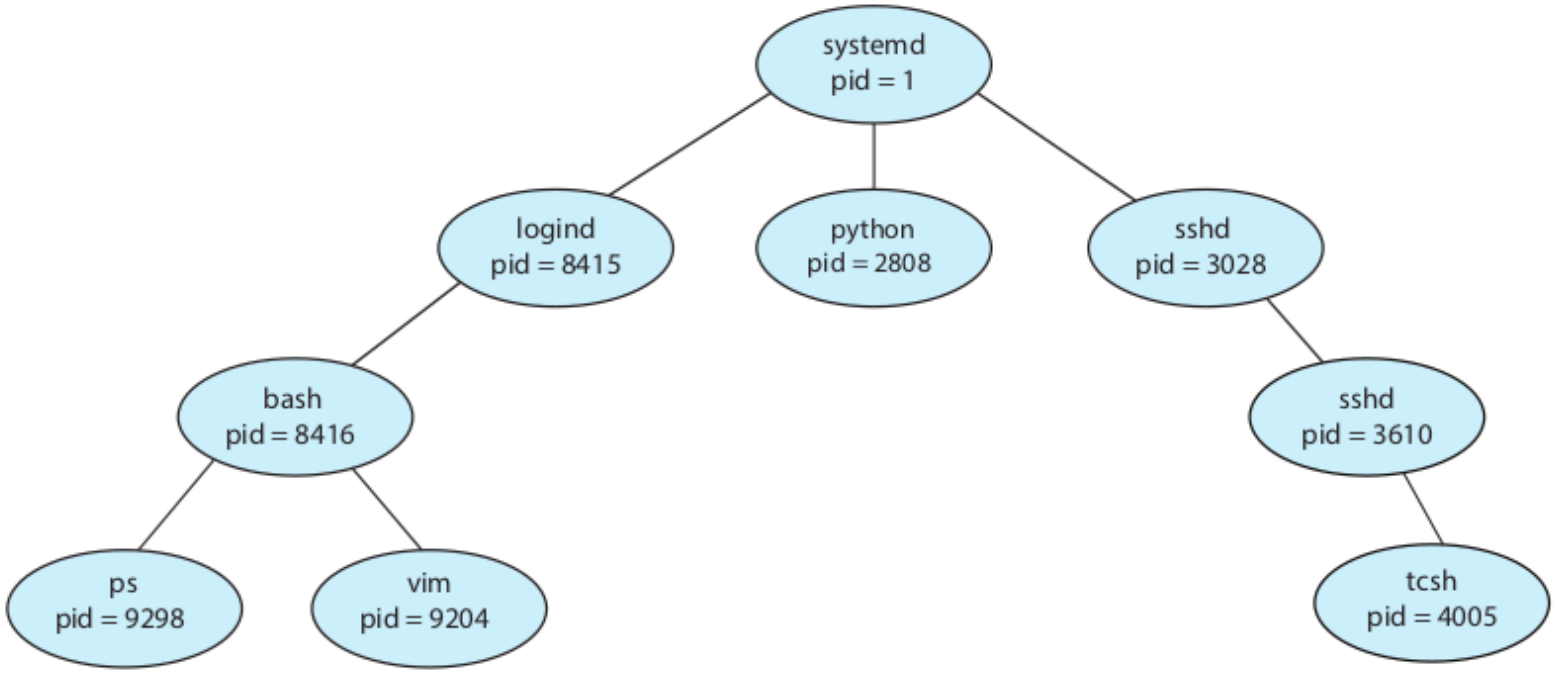
États d'un processus

À l'aide d'appels système, un processus peut créer un autre processus

- Le processus généré est **enfant** du processus générateur
- Une **arborescence de processus** est créée.
- Si le processus parent se termine, les enfants **ne sont pas** sont terminés
- Les processus *sans parents* deviennent des enfants du processus *init*

États d'un processus

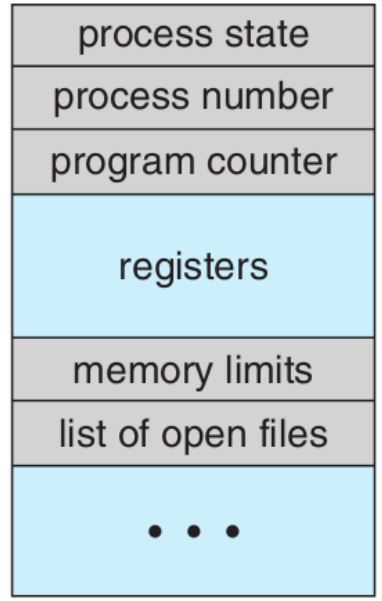
Exemple d'arbre de processus :



Process Control Block

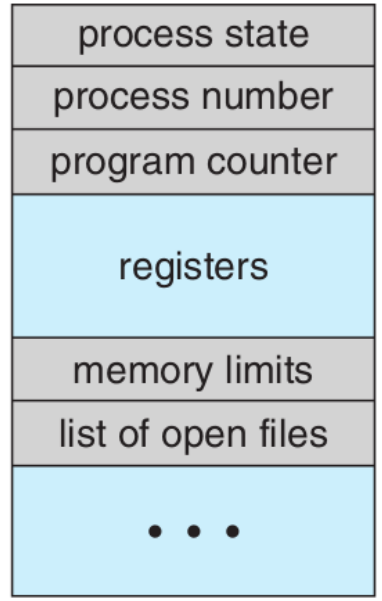
Chaque processus est représenté dans le système d'exploitation par un bloc de contrôle de processus (PCB) contenant les informations correspondantes

- **État du processus** : nouveau, prêt, en cours d'exécution, en attente, arrêté
- **Compteur de programme** : adresse de la prochaine instruction à exécuter
- **Registres du CPU** : permettent d'interrompre le processus



Process Control Block

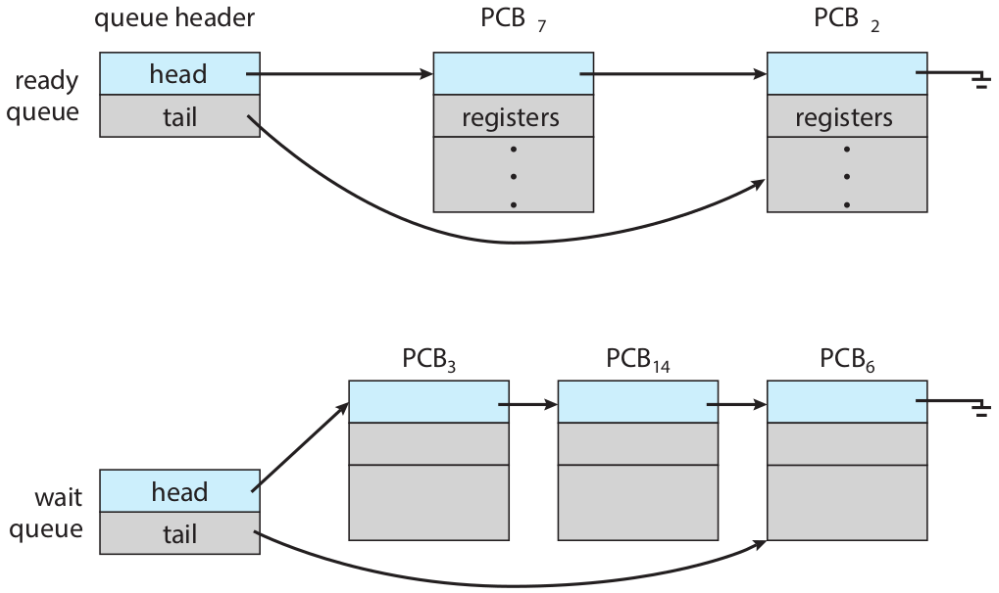
- **Informations sur l'ordonnancement** : priorités, ressources consommées
- **Informations sur la gestion de la mémoire** : pointeurs vers diverses zones de mémoire
- **Informations I/O** : fichiers ouverts, opérations en attente, etc...



Ordonnancement

Le **process scheduler** sélectionne un processus à exécuter parmi l'ensemble des processus disponibles.

- Maintient une file d'attente de processus prêts
- Maintient une file d'attente de processus en attente d'un événement. Exemple : exécution d'une action d'entrée/sortie



Ordonnancement

Les processus peuvent être classés en fonction du type de charge qu'ils génèrent et du goulot d'étranglement qui les limite.

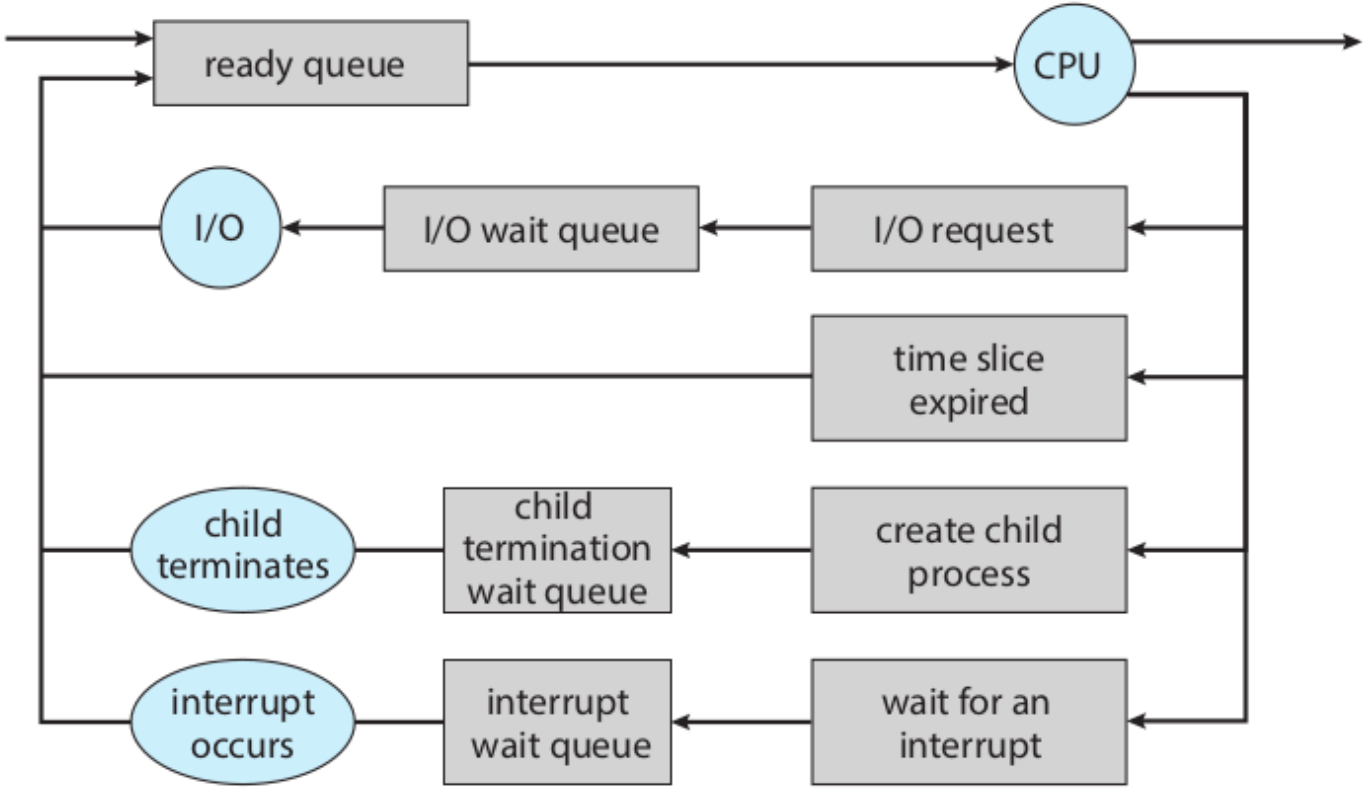
- **I/O Bound** : passe la majeure partie de son temps à effectuer des opérations d'I/O.
- **CPU Bound** : passe le plus clair de son temps à effectuer des opérations dans la CPU.

La tâche d'un ordonnanceur consiste à **optimiser** l'exécution des processus pour qu'ils s'exécutent dans le temps le plus court possible

- Il intervient plusieurs fois par seconde
- Il gère l'exécution à l'aide du mécanisme de **partage du temps**.

Ordonnancement

Chaque processus démarre à partir de la **file d'attente prête** et suit l'ordonnancement de la **queue** jusqu'à ce qu'il se termine.



Ordonnancement

L'ordonnanceur décide quelle CPU doit être allouée à quelle processus.
Lorsqu'il décide qu'une CPU doit être affectée à un autre processus, il doit :

- Sauvegarder l'état du processus en cours
  - Sauvegarder l'état du processus en cours afin qu'il puisse être restauré ultérieurement lorsque le processus lui-même sera à nouveau en mesure de s'exécuter
- Charger un nouveau processus en restaurant son état précédemment sauvegardé

Effectuer une sauvegarde d'état puis une restauration d'état correspondante, appelée **Context Switching**.

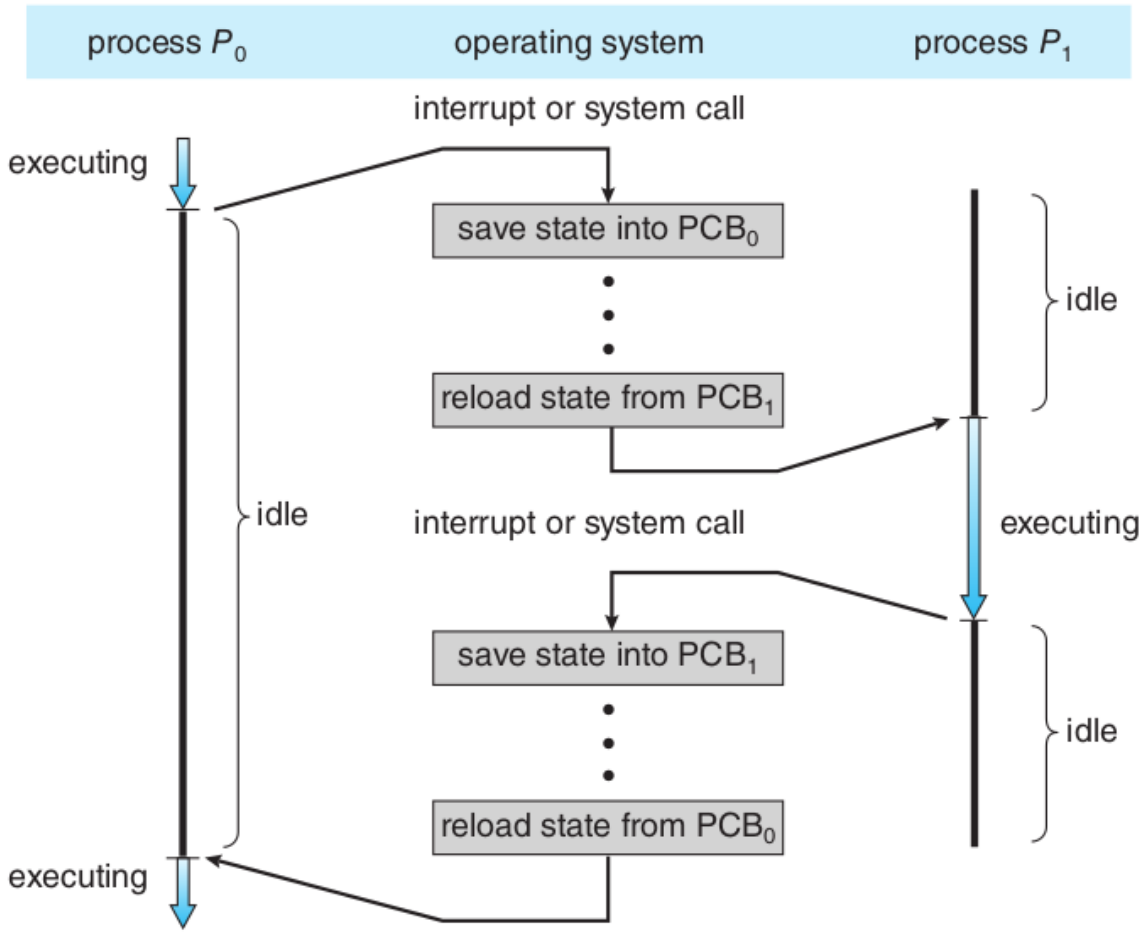
Ordonnancement

Le changement de contexte doit être rapide, car il s'agit d'une perte de temps qui ne sert à rien.

- Le système d'exploitation est optimisé pour effectuer cette action rapidement
- Actuellement de l'ordre de quelques microsecondes
- Dépend des caractéristiques du matériel et du processus, en particulier de la quantité de mémoire utilisée

Ordonnancement

Diagramme de changement de contexte :



Ordonnancement

Opération Yield : un processus indique au noyau qu'il n'a pas d'opérations à effectuer pour le moment.
Le noyau retire le processus de l'unité centrale et le remet dans la liste des processus prêts.

- C'est une façon de libérer l'unité centrale avant que le temps alloué n'expire.
- Particulièrement utilisé pour les processus en *temps réel*.

Algorithmes d'ordonnancement

L'objectif d'un système d'exploitation est de réduire le temps d'exécution des processus.

- Objectif ambigu : un travail long ou court doit-il être exécuté en premier ?
- Objectif complexe : le système d'exploitation ne sait pas si un processus est long ou court, s'il est intensif en termes d'unités de calcul ou d'entrées-sorties.

Il existe plusieurs **algorithmes d'ordonnancement** utilisés pour déterminer quel processus assigner à une CPU.

First-Come First Served

Le premier processus qui demande l'unité centrale l'obtient jusqu'à ce qu'il se termine.

Pro :

Contre :

First-Come First Served

Le premier processus qui demande l'unité centrale l'obtient jusqu'à ce qu'il se termine.

Pro : Simple

Contre : inefficace. Il est inefficace d'exécuter un processus long en premier.

Shortest-Job First Served

Le premier travail le plus court obtient l'unité centrale en premier

Pro :

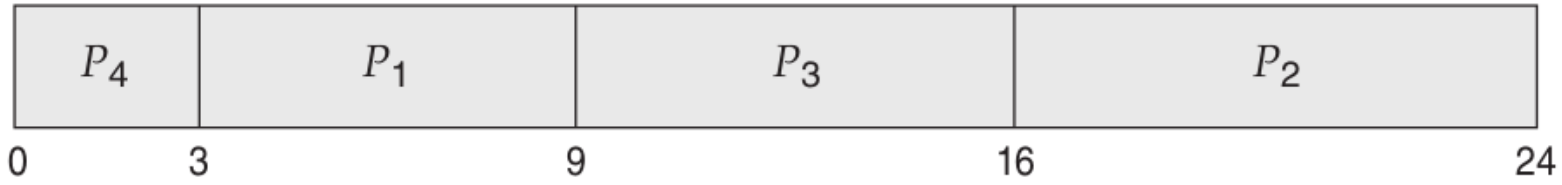
Contre :

Shortest-Job First Served

Le premier travail le plus court obtient l'unité centrale en premier

Pro : efficace. Le temps d'exécution moyen est réduit

Contre : Un processus long est autorisé à s'exécuter jusqu'à la fin.



Round Robin

À tour de rôle, chaque processus occupe l'unité centrale pendant une durée déterminée.

Pro :

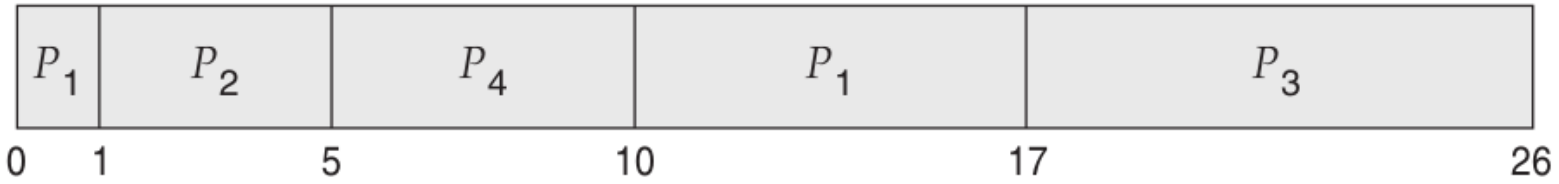
Contre :

Round Robin

À tour de rôle, chaque processus occupe l'unité centrale pendant une durée déterminée.

Pro : Simple et équitable

Contre : Pas de processus prioritaires



Priority Scheduling

Chaque processus a une priorité donnée par l'utilisateur. Le processus le plus prioritaire est exécuté

Pro :

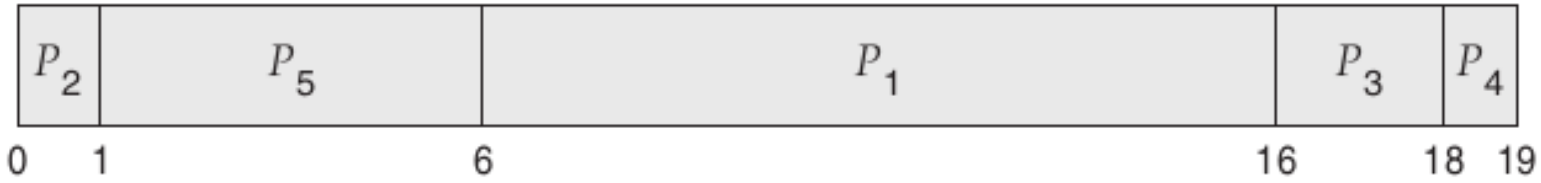
Contre :

Priority Scheduling

Chaque processus a une priorité donnée par l'utilisateur. Le processus le plus prioritaire est exécuté

Pro : Gère la priorité

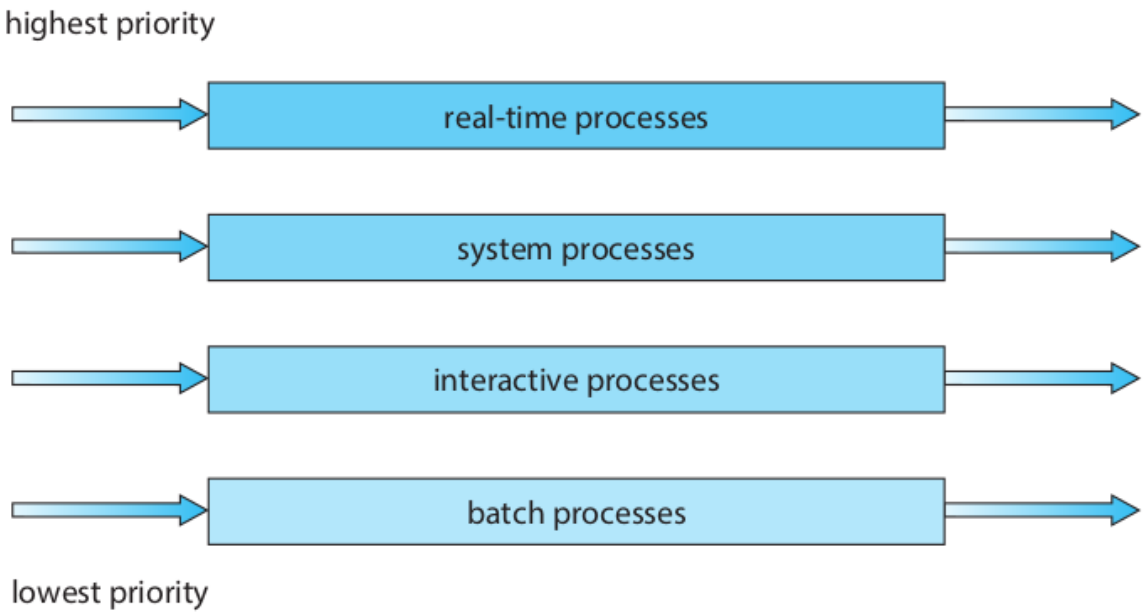
Contre : Un processus de faible priorité peut ne jamais être exécuté



Multi-Level Queue Scheduling

Il existe différentes files d'attente pour chaque niveau de priorité.

- Chaque file d'attente a son propre algorithme d'ordonnancement : RR, FCFS.
- Il existe un algorithme d'ordonnancement entre les files d'attente



Multi-Level Queue Scheduling

De cette façon, il y a de la flexibilité : vous pouvez avoir la priorité mais il n'y a pas de risque qu'un processus ne soit jamais exécuté.

Pro : Flexible
Contre : Complexe

Utilisé dans Linux, avec des variantes

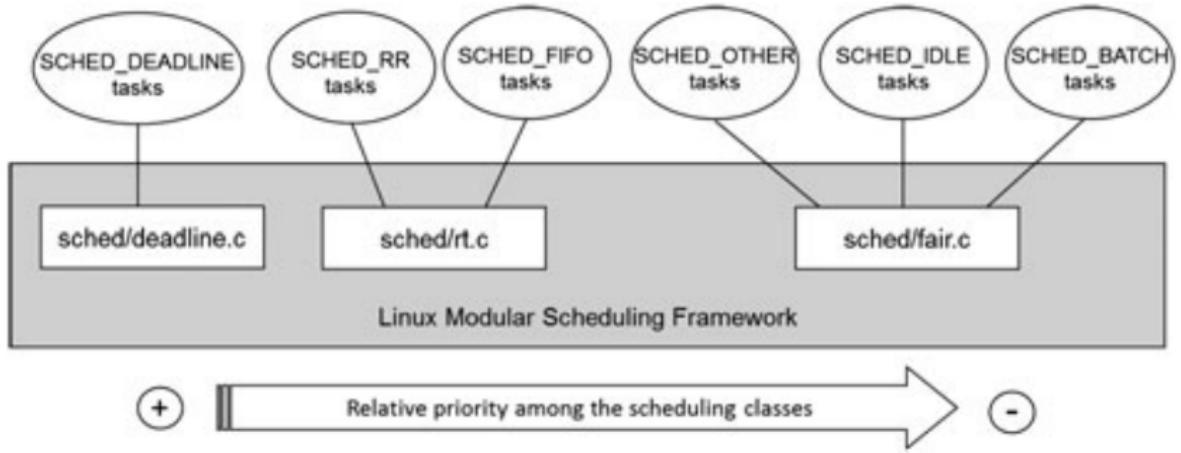
Linux: Completely Fair Scheduler

Sous Linux, l'algorithme d'ordonnancement est appelé **Completely Fair Scheduler**.

Les processus sont assignés à une **politique d'ordonnancement** par l'utilisateur, chacune avec des mécanismes différents

Le système exécute les processus de chaque politique, qui ont des exigences différentes

Les politiques peuvent gérer les priorités, les délais, etc...



Linux: Completely Fair Scheduler

La politique par défaut est `SCHED_OTHER` : ordonnancement en temps partagé avec priorité

Des appels système existent pour donner la priorité à un processus

`nice(2)`, `getpriority(2)`, `setpriority(2)`, `sched_setscheduler(2)`, `sched_getscheduler(2)`, `sched_setparam(2)`, `sched_getparam(2)`, and `sched_yield(2)`

Travailler avec les processus

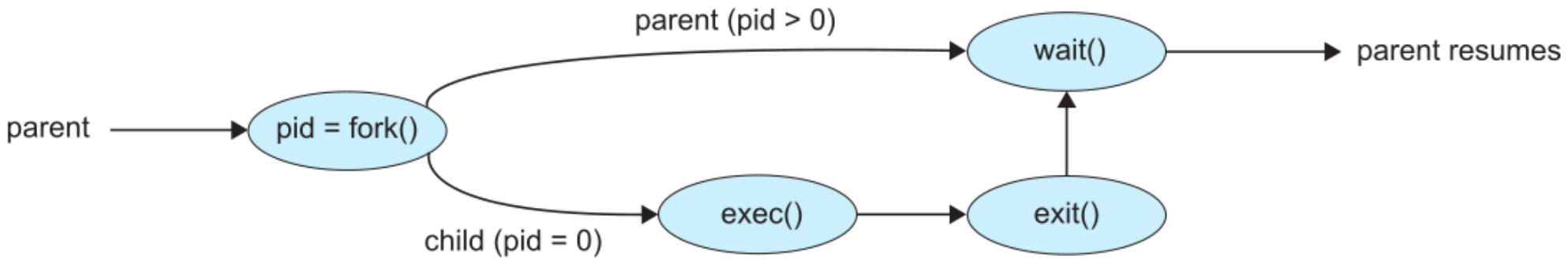
Sujets

1. Gestion des processus
2. Les fonction `fork`, `wait`, `exec`, `system`, `exit`
3. Autres fonctions
4. Commandes Bash pour les processus

Gestion des processus

Sous Linux, il existe 6 appels système principaux

- `fork` : crée un processus dupliqué
- `exec` : charge le code exécutable
- `wait` : attendre la fin du processus
- `signal` : capture un signal
- `kill` : envoie un signal
- `exit` : tue le processus en cours



Création d'un processus

Windows vs Linux :

Windows a un appel système complexe (`CreateProcessA`)

- Très verbeux
- Nombreux paramètres
- Très typé

```
BOOL  CreateProcessA(
  LPCSTR lpApplicationName,
  LPSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCSTR lpCurrentDirectory,
  LPSTARTUPINFOA lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
) ;
```

Création d'un processus

Windows vs Linux :

Windows a un appel système complexe (`CreateProcessA`)

- Très verbeux
- Nombreux paramètres
- Très typé

```
BOOL  CreateProcessA(
  LPCSTR lpApplicationName,
  LPSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment, ⬅️
  LPCSTR lpCurrentDirectory, ⬅️
  LPSTARTUPINFOA lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
) ;
```

Création d'un processus

Windows vs Linux :

Linux préfère les appels système simples :

- `fork` clone un processus
- `exec` exécute un fichier exécutable dans le processus en cours

Fonction `fork`

```c
#include <unistd.h>
pid_t  fork  (void) ;
```

Crée un nouveau processus enfant, en copiant complètement l'image mémoire du processus parent (données, stack, heap)

- Les deux processus évoluent de manière indépendante
- La mémoire est complètement indépendante entre le père et l'enfant
- Le code est généralement partagé entre le père et l'enfant
    - Code copié en écriture (copié lorsqu'il est modifié)

Note : `pid_t` est un alias pour un `int`, comme `size_t`

Fonction `fork`

- Tous les descripteurs de fichiers ouverts dans le processus parent sont dupliqués dans le processus enfant.
  - Copie complete !
- Le processus enfant et le processus parent continuent d'exécuter l'instruction suivant au `fork`.
- Valeur de retour :
  - Processus enfant : 0
  - Processus parent : PID du processus enfant
  - Erreur de fourche : PID négatif (père uniquement)

Fonction `fork`

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

void  enfant(void);
void  pere(void);

void    main(void) {
       pid_t   pid;
       pid = fork();
       if (pid == 0)
          enfant();
       else
          Padre();
}
void    enfant(void) {
      int  i=0;
      for(i=0;i<10;i++){
        usleep(200);
        printf("\tI am the children. i= %d\n",i);
      }
}
void  pere(void)
{
       int  i=1;
       for(i=0;i<10;i++){
        usleep(250);
        printf("I am the father. i= %d\n",i);
       }
}
```

Fonction `fork`

Observations

La valeur de retour du `fork` est cruciale

Un programme écrit en termes de `fork` n'est pas immédiatement compréhensible

- Opération atomique, mais effets complexes
- Il est possible de créer des arbres de processus complexes, avec un code complexe

Fonction `fork`

Fork Bomb: Un programme qui appelle le `fork` dans une boucle sans fin fait planter la machine en raison d'un trop grand nombre de processus.

```
#include <unistd.h>
int  main(void) {
      while(1)
           fork();
}
```

Fonction `fork`

Fork Bomb en Bash:

```
:(){ :|:& };:
```

ce qui équivaut à :

```
myfork() {
    myfork | myfork &
}
myfork
```

Fonction `fork`

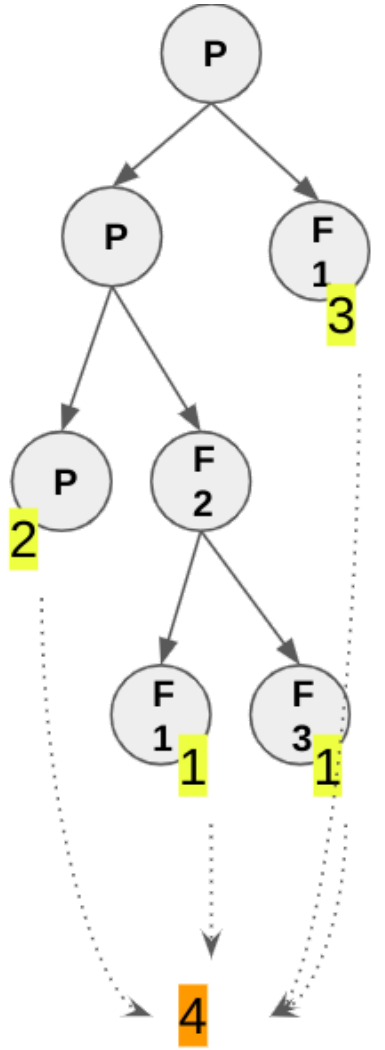
```
#include <stdio.h>
#include <unistd.h>
int  main(){
     if (fork()){
          if (!fork()){
               fork();
               printf("1 ");
          }
          else
               printf("2 ");
     }
     else
          printf("3 ");
     printf("4 ");
     return  0;
}
```

Fonction `fork`

```
#include <stdio.h>
#include <unistd.h>
int  main(){
     if (fork()){
          if (!fork()){
               fork();
               printf("1 ");
          }
          else
               printf("2 ");
     }
     else
          printf("3 ");
     printf("4 ");
     return  0;
}
```

Output:

```
2 4 3 4 1 4 1 4
```



Fonction `fork`

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

int  main(){
   printf("\n");
   if (fork() && (!fork())) {
     if (fork() || fork()) {
       fork();
     }
   }

   printf("2 ");
   return  0;
}
```

Fonction `fork`

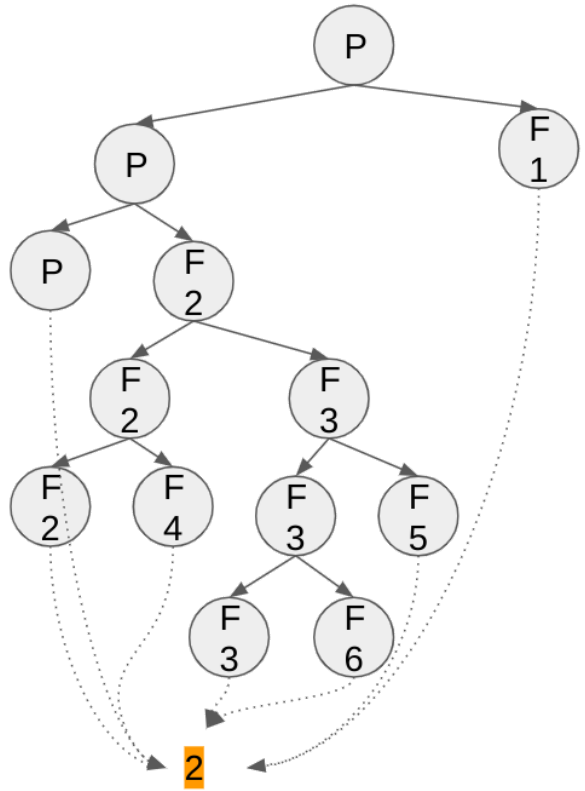
```c
#include <stdio.h>
#include <unistd.h>

int  main(){
   printf("\n");
   if (fork() && (!fork())) {
      if (fork() || fork()) {
         fork();
      }
   }

   printf("2 ");
   return  0;
}
```

Output:

```
2 2 2 2
```



Fonction `fork`

```c
#include <stdio.h>
#include <unistd.h>
int  main(int argc,char *argv[]){
   printf("A\n");
   fork();
   printf("B\n");
   fork();
   printf("C\n");
   return  0;
}
```

Output:

```
A
B
B
C
C
C
C
```

Fonction `fork`

```c
#include <stdio.h>
#include <unistd.h>
int  main(int argc,char *argv[]){
   printf("A ");
   fork();
   printf("B ");
   fork();
   printf("C ");
   return  0;
}
```

Output:

```
A B C A B C A B C A B C
```

Pourquoi ?

Fonction `wait`

```
#include <sys/wait.h>
pid_t  wait  (int *status)  ;
```

Attend la première terminaison de **un** enfant
argument `status` :

- Pointeur vers un entier ;
- S'il n'est pas NULL, spécifie le statut de sortie du processus fils (valeur retournée par le processus fils)

Valeur de retour :

- Le PID de l'enfant terminé
- 0 en cas d'erreur

Fonction `wait`

Cas :

- Si le processus n'a pas d'enfants : erreur
- Si le processus a des enfants qui se sont déjà terminés : retour instantané
- Si le processus a des enfants qui n'ont pas encore terminé : bloque l'appelant jusqu'à ce qu'il termine un enfant.

Fonction `wait`

La fonction `wait` *consomme* un enfant à la fois.
Après qu'un enfant ait été *retourné* au parent via une fonction `wait` :

- Le système d'exploitation libère les ressources du processus enfant
  - Le système d'exploitation conserve des informations sur les processus terminés qui n'ont pas encore fait l'objet d'un `wait`.
  - Trace de l'existence du processus
  - Valeur de retour et informations sur l'exécution

Fonction `wait`

Processus zombie : processus terminé dont le parent n'a pas encore effectué un `wait`.

- Une fois le `wait` est effectuée, le processus est définitivement mort et il n'en reste aucune trace.

Processus orphelins : processus lesquels le père est décédé.

- Si le père meurt, les enfants continuent l'exécution
- Ils deviennent les enfants du processus *init* ($PID = 1$)
- Périodiquement, *init* exécute `waits` pour consommer les orphelins morts.

Fonction `wait`

```
#include <sys/wait.h>
pid_t  waitpid(pid_t pid, int *status, int options) ;
```

Attend la première terminaison de :

- Tout enfant si `pid == -1` (comme `wait` classique)
- Tout enfant avec `pid` si `pid>0`.
- Tout enfant dont le **group ID** est le même que celui de l'appelant si `pid == 0`
- Un enfant dont le **group ID** est égal à `abs(pid)` si `pid <-1`

Fonction `wait

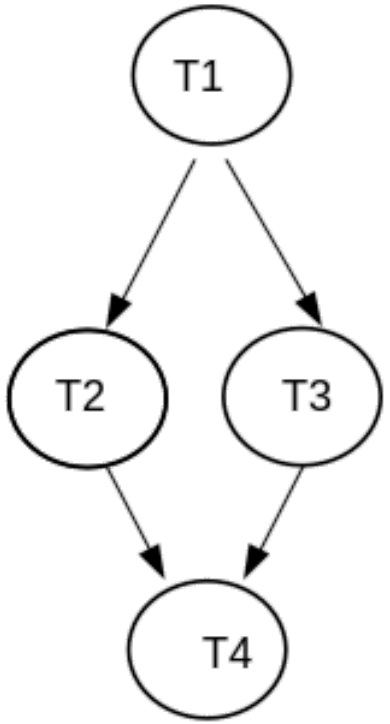
ID du groupe : Entier positif associé à un processus. Utilisé pour définir des groupes de processus créés par l'utilisateur. Utile pour garder l'ordre.

Autres arguments de `waitpid` :

- `status` comme dans `wait`.
- `options` : vérifie si la fonction est bloquante. Masque de bits.
  - 0 bloquant
  - `WNOHANG` : ne bloque pas si aucun enfant n'est déjà mort
  - Autres drapeaux pour n'intercepter que les enfants morts dans des conditions particulières

Fonction `wait`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
     pid_t pid;
     printf ("T1\n");
     pid = fork();
     if (pid == 0) {
          printf("T3\n");
          return 0;
     } else {
          printf("T2\n");
          wait((int *)0);
     }
     printf("T4\n");
     return 0;
}
```



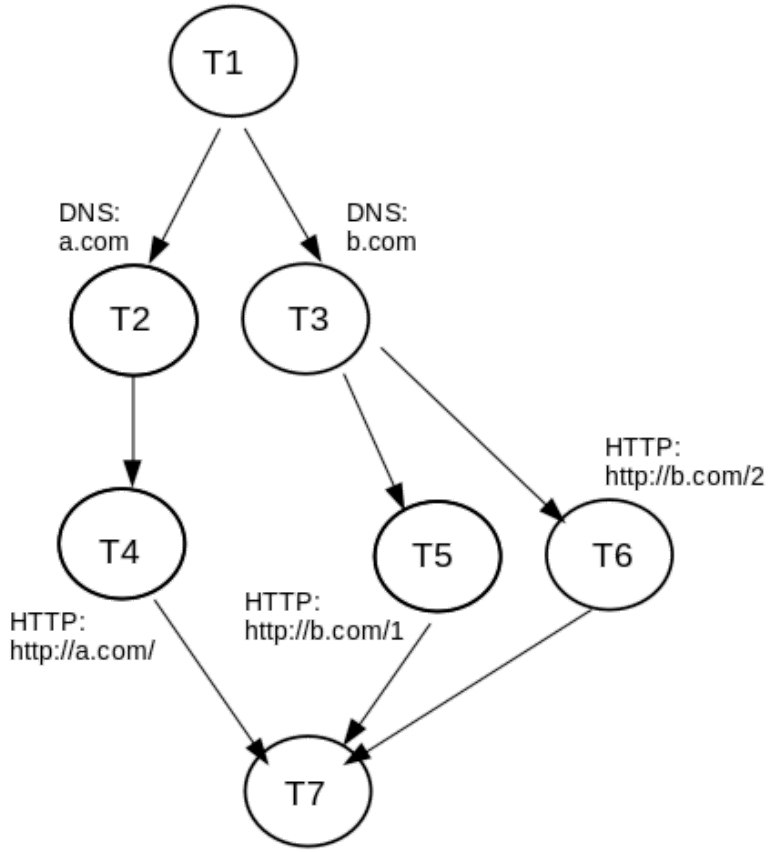
Fonction `wait`

Graphique permettant d'exécuter efficacement 3 requêtes HTTP vers des URL.

- `http://a.com/`
- `http://b.com/1`
- `http://b.com/2`

Avant chaque requête, une résolution DNS doit être effectuée. Deux URL ont le même domaine.

Note : Dans la réalité, le programmeur doit résoudre le problème de manière efficace. Il doit construire lui-même le graphe de précédence.



Fonction `wait`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main() {
     pid_t pid;
     printf ("T1 - Start\n");
     pid = fork();
     if (pid == 0) {
          printf ("T3 - DNS b.com\n");
          pid_t pid2;
          pid2=fork();
          if (pid2==0){
               printf ("T6 - HTTP http://b.com/1\n");
               return 0;
          }
          else{
               printf ("T5 - HTTP http://b.com/2\n");
               wait ((int *) 0); /* Attende T6 */
               return 0;
          }
     } else {
          printf ("T2 - DNS a.com\n");
          printf ("T4 - HTTP http://a.com/\n");
          wait ((int *) 0); /* Attende T3 - T5 */
     }
     printf ("T7 - Utilizzo i risultati\n");
     return 0;
}
```

Fonction `wait`

Et ce graphe ?

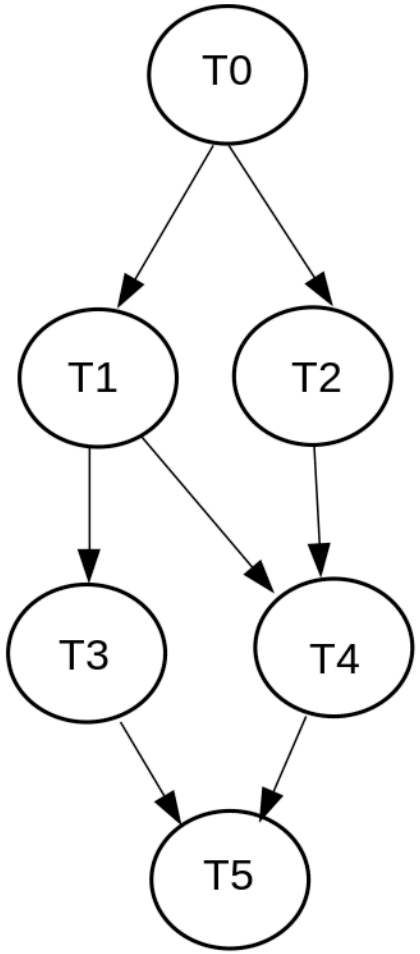
Fonction `wait`

Et ce graphe ?

Ce graphe est **impossible** à réaliser avec les seuls `fork` et `wait`.

T4 ne peut pas *attendre* T1. Il n'est pas son enfant !
En général :

- Seuls les enfants peuvent être *attendus*.
- Chaque enfant ne peut être attendu qu'une seule fois



Fonction `wait`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int  main() {
     pid_t pid;
     printf ("T0\n");
     pid = fork();
     if (pid == 0) {
          printf ("T2\n");
          wait ( ??? ) /* <--------- IMPOSSIBILE! */
          printf ("T4\n");
     } else {
          printf ("T1 -\n");
          wait ((int *) 0); /* Attende T4 */
     }
     return 0;
}
```

Fonction `exec`

La `fork` permet de dupliquer un processus.

- Utilisé lorsque l'enfant doit exécuter le même programme que son père
- Dans les programmes parallèles : serveur web, base de données

Le `exec` permet de changer la nature d'un processus en cours

- En chargeant et en exécutant un programme différent
- Utilisé lorsqu'un nouveau programme doit être lancé

Fonction `exec`

Lorsqu'un processus appelle une fonction `exec` :

- Le processus est remplacé **complètement** par le code contenu dans le fichier spécifié (le texte, les données, le stack, le heap sont remplacés).
- Le nouveau programme démarre à partir de sa fonction `main`.
- Le PID ne change pas

Fonction `exec`

Ce dont le processus hérite après un `exec` :

- Variables d'environnement
- PID et PPID
- répertoire de travail actuel, répertoire racine et répertoire personnel

Ce qui n'est pas hérité :

- Les fichiers ouverts s'ils ont le flag `close-on-exec`.
- Sinon, ils restent ouverts

La fonction `exec`

Il existe 7 versions de `exec`.

Elles ont la même fonction, la façon dont elles reçoivent les arguments varie.

```
#include <unistd.h>

int  execl(const char *pathname, const char *arg, ...)  ;
int  execlp(const char *file, const char *arg, ...)  ;
int  execle(const char *pathname, const char *arg, ..., char *const envp[])  ;
int  execv(const char *pathname, char *const argv[])  ;
int  execvp(const char *file, char *const argv[])  ;
int  execve(char *pathname, char *argv[], char* envp[])  ;
int  execvpe(const char *file, char *const argv[], char *const envp[])  ;
```

Fonction `exec`

Les fonctions avec `p` obtiennent le nom de l'exécutable, pas le chemin.

- Le système d'exploitation trouve l'exécutable dans les dossiers des programmes installés sur le système
- qui sont définis dans la variable d'environnement `PATH`.

Exemple :

```
execlp('ls') ;
execl("/usr/bin/ls") ;
```

Fonction `exec`

- Les fonctions avec `l` spécifient les **arguments** du nouveau programme via une liste d'arguments. Similaire à `printf`
- Les fonctions avec `v` spécifient les **arguments** du nouveau programme via un pointeur unique sur `char`. Equivalent à `argv` dans `main`
    - Le premier argument doit contenir le nom de fichier associé à l'exécutable en cours de chargement.
    - Le tableau de pointeurs doit être terminé par un pointeur NULL.
        - En l'absence de `argc`, ceci est utilisé pour communiquer la longueur du vecteur

Fonction `exec`

Exemple :

```
// Simple
execlp("cp", "file1", "file2") ;

// Générique
const char *args[4] ;
args[0] = "cp" ;
args[1] = "file1" ;
args[2] = "file2" ;
args[3] = NULL ;
execvp("cp", args) ;
```

Fonction `exec`

Les fonctions avec `v` reçoivent un vecteur de variables d'environnement. Ainsi, elles ne sont pas érodées par le processus parent.

- Les variables d'environnement sont spécifiées dans le dernier argument par un pointeur sur un pointeur `char`, terminé par un `NULL`.
    - terminé par un pointeur `NULL`.

```
char *const args[] = {"ls", "/tmp", NULL} ;
execv("/usr/bin/ls", args) ;

char *const envs[] = {"a=1", "b=2", NULL} ;
execle("/usr/bin/ls", args, envs) ;
```

Fonction `exec`

Observation :

- La fonction `execve` est un appel système.
- Les autres fonctions sont des fonctions de bibliothèque et invoquent `execve` après avoir correctement traité et ajusté les paramètres.

Fonction `exec`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#define MAXLINE 128
int  main() {
          char  buf[MAXLINE];
          pid_t  pid;
          int  status;
          printf("%% ");
          while  (fgets(buf, MAXLINE, stdin)  !=  NULL)  {
                    if  (buf[strlen(buf)  -  1]  ==  '\n')
                              buf[strlen(buf)  -  1]  =  0;
                    if  ((pid  =  fork())  <  0)  {
                              printf("fork error");  exit(1);
                    }  else  if  (pid  ==  0)  {  /*figlio */
                              execlp(buf,  buf,  NULL);
                              printf("can't execute: %s\n",  buf);
                              exit(127);
                    }  else
                    if  ((pid  =  waitpid(pid,  &status,  0))  <  0)
                              {printf("waitpid error");  exit(1);}
                    printf("%% ");
          }
          exit(0);
}
```

Qu'est-ce que ce code met en œuvre ?

Fonction `exec`

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int  main  (int argc, char ** argv)  {
     char  str[10];
     int  n;
     n = atoi(argv[1]) - 1;
     printf  ("%d\n", n);
     if  (n>0)  {
          sprintf  (str, "%d", n);
          execl  (argv[0], argv[0], str, NULL);
     }
     printf  ("End!\n");
     return  1;
}
```

Qu'est-ce qui est imprimé en exécutant `./prog 5` ?

Fonction `exec`

```
4
3
2
1
0
End !
```

Fonction `system`

Est une fonction de bibliothèque qui invoque une commande Bash et attend qu'elle se termine

- Utile pour appeler facilement des programmes externes dans un programme
- Combine : `fork`, `exec` et `wait`.

```
#include <stdlib.h>

int  system(const char *command) ;
```

Équivalent à un `fork` dont l'enfant exécute :

```
execl("/bin/sh", "sh", "-c", command, (char *) NULL) ;
```

Fonction `system`

Mise en oeuvre simple :

```
int  system(const char *cmd)
{
     int stat ;
     pid_t pid ;
     if (cmd == NULL)
          return(1) ;
     if ((pid = fork()) == 0) { /* Son */
          execl("/bin/sh", "sh", "-c", cmd, (char *)0);
          _exit(127);
     }
     if (pid == -1) {
          stat = -1; /* Error */
     } else { /* Father */
          while (waitpid(pid, &stat, 0) == -1) {
               if (errno != EINTR){
                    stat = -1;
                    break;
               }
          }
     }
     return(stat);
}
```

Fonction `exit`

Il existe plusieurs façons de mettre fin à un processus :

1. Mode standard

- A partir du `main`, un `return` est effectué

```
return status;
```

- La fonction `exit` est appelée

```
#include <stdlib.h>
void  exit(int status) ;
```

Tous les buffers (console et fichiers) sont *flushés*.

L'argument `status` est retourné au système d'exploitation

Fonction `exit`

Pour permettre ces opérations de nettoyage, toutes les fonctions de verrouillage enregistrées sont appelées via la fonction `atexit`.

```
void  foo(void)  {  printf('Exiting`) ; }

int main()
{
   atexit(foo) ;
   exit(10) ;
}
```

`Exiting` est imprimée

Fonction `exit`

2. Fonction `_exit`

```
#include <unistd.h>
void _exit(int status) ;
```

Se termine immédiatement sans vérifier les tampons.
Invoqué dans les processus **enfants** qui peuvent lire les *buffers* dans l'état intermédiaire des pères
Utilisé surtout après l'échec de `exec`.

- L'enfant ne doit pas exécuter d'instructions après `exec` !
- Les buffers peuvent contenir des données du père qui ne doivent pas être écrites par les enfants.

Fonction `exit`

Note :

- Le `_exit` est un appel système.
- La fonction `exit` est une fonction de bibliothèque. Elle nettoie et invoque ensuite `_exit`.

3. Terminaison anormale:

- Un **signal** non géré est reçu (nous le verrons).
- Le programme appelle `abort`.

```
#include <stdlib.h>
void  abort(void) ;
```

Fonction `exit`

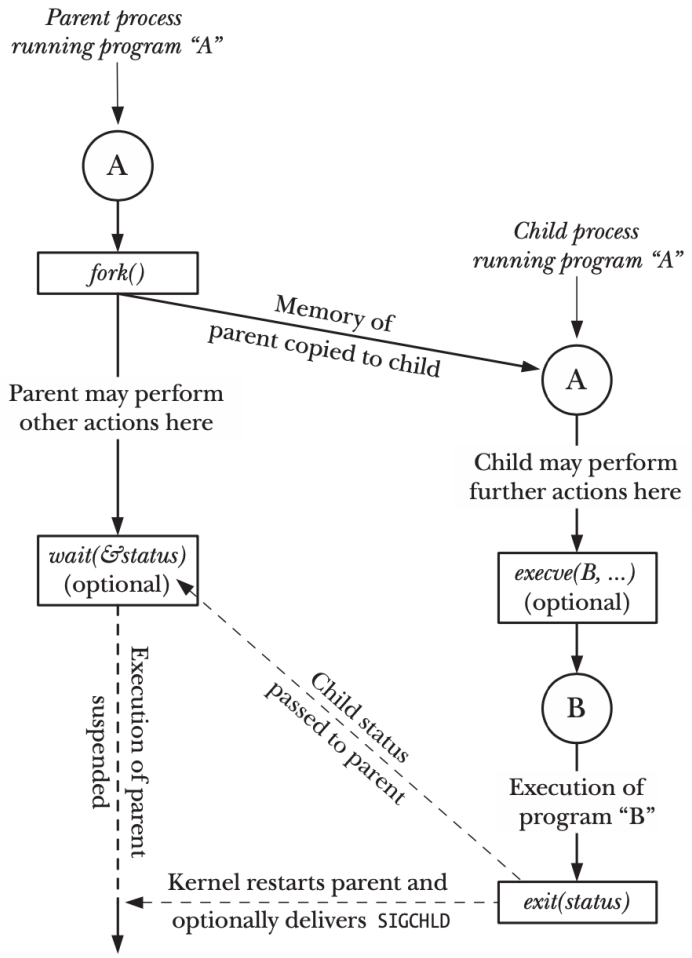
Quelle que soit la façon dont le processus se termine, le noyau effectue les actions suivantes :

- Suppression de la mémoire utilisée par le processus
- Fermeture des descripteurs ouverts

Statut d'un processus : collecté avec `wait()`, `waitpid()`

En résumé

- `fork` duplique le processus en cours
- `execve` transforme le processus en cours en un autre programme
- `exit` termine le processus en cours (égal à `return` de `main`)
- `wait` s'arrête jusqu'à ce qu'un processus enfant se termine



Autres fonctions

```c
#include <unistd.h>

pid_t  getpid(void) ;
pid_t  getppid(void) ;
```

La fonction `getpid()` renvoie le PID du processus appelant

La fonction `getppid()` renvoie le PID du **parent** du processus appelant.

Commandes Bash pour les processus

- `ps` : liste des processus du système
  - Par défaut, il n'affiche que les processus enfants du terminal courant.
    Avec l'option `a`, il affiche tout.
  - Par défaut, il n'affiche que les processus qui sont au premier plan (qui ont un shell).
    Avec l'option `x`, il affiche également les processus qui sont en arrière-plan.
  - Options utiles : `u` montre l'utilisateur propriétaire. `f` rend graphiquement la hiérarchie parent-enfant.
- `top` : montre les processus de manière interactive
- `htop` : comme `top` mais avec des graphiques améliorés

Commandes Bash pour les processus

- `which` : donne le chemin absolu vers un programme système

```
$ which ls
/usr/bin/ls
```

- `pgrep` : affiche le PID de tous les processus d'un programme donné

```
$ pgrep chrome
480492
480498
480505
```

Commandes Bash pour les processus

Exécution de processus enfants à partir de scripts bash :

- Une commande se terminant par `&` est exécutée en arrière-plan.
  - Un `fork` et un `exec` sont exécutés pour exécuter la commande.
  - N'exécute pas `wait`. Le script et le programme s'exécutent en parallèle
- Le script obtient le PID du processus avec `$!`.
  - Il est écrasé par chaque processus créé !
- Vous pouvez utiliser la commande `wait [PID]`.
  Attend le fils `PID` si spécifié, sinon n'importe quel fils

Commandes Bash pour les processus

Exemple : Combien de temps prend l'exécution de ce code ?

```
sleep 4 &
PID=$ !
sleep 2
wait $PID
```

Commandes Bash pour les processus

Exemple : Combien de temps prend l'exécution de ce code ?

```
sleep 4 & # Sleep s'exécute en arrière-plan
PID=$ !          # Le script récupère le PID
sleep 2
wait $PID # Le script attend la fin de sleep
```

Cela prend 4 secondes, pas 6

Le système de fichiers `/proc

Les systèmes Linux/POSIX exposent des informations sur les processus en cours via un système de fichiers virtuels.

- Monté automatiquement dans `/proc`.
- Permet à quiconque de connaître l'état des processus en cours d'exécution
- Via des lectures normales de fichiers

Le système de fichiers `/proc

Les informations concernant un processus PID se trouvent dans le **répertoire** `/proc/PID`

Le **fichier** `/proc/PID/status` contient diverses informations :

```
$ cat /proc/1566/status
Nom : cat
État : R (en cours d'exécution) R (en cours d'exécution)
Tgid : 5452
Pid : 5452
PPid : 743
...
VmPeak : 5004 kB
VmSize : 5004 kB
VmLck : 0 kB
VmHWM : 476 kB
VmRSS : 476 kB
```

Le système de fichiers `/proc'.

Le **sous-répertoire** `/proc/PID/fd` contient un lien pour chaque fichier ouvert par le processus

- Le nom de ces liens est le numéro du descripteur utilisé par le processus.
- Rappel : chaque fichier ouvert est identifié par un numéro

Exemple :

```
/proc/1968/1
```

Représente le `stdout` du processus 1968.

- Rappelez-vous : 0 est `stdin`, 1 est `stdout`, 3 est `stderr`.

Le système de fichiers `/proc

Autres **fichiers/sous-répertoires** du processus PID sous `/proc/PID`

File	Description (process attribute)
`cmdline`	Command-line arguments delimited by `\0`
`cwd`	Symbolic link to current working directory
`environ`	Environment list *NAME=value* pairs, delimited by `\0`
`exe`	Symbolic link to file being executed
`fd`	Directory containing symbolic links to files opened by this process
`maps`	Memory mappings
`mem`	Process virtual memory (must *lseek()* to valid offset before I/O)
`mounts`	Mount points for this process
`root`	Symbolic link to root directory
`status`	Various information (e.g., process IDs, credentials, memory usage, signals)
`task`	Contains one subdirectory for each thread in process (Linux 2.6)

Le système de fichiers `/proc

Le système de fichiers `/proc` fournit également de nombreuses informations sur le système et des possibilités de configuration

- `/proc/cpuinfo` : informations sur le processeur
- `/proc/meminfo` : informations sur la mémoire

Directory	Information exposed by files in this directory
/proc	Various system information
/proc/net	Status information about networking and sockets
/proc/sys/fs	Settings related to file systems
/proc/sys/kernel	Various general kernel settings
/proc/sys/net	Networking and sockets settings
/proc/sys/vm	Memory-management settings
/proc/sysvipc	Information about System V IPC objects

Sujets

1. Concept de signal
2. Signaux dans Linux
3. Appel système `sigaction`
4. Appel système `kill`
5. Appel système `raise`
6. Appel système `pause`
7. Appel système `alarme`
8. Considérations
9. Signaux dans le Shell

Concept de signaux

Dans presque tous les systèmes informatiques, il existe des **interruptions** :
Une interruption informe l'unité centrale qu'elle doit interrompre la tâche en cours afin d'effectuer une action contraignante.
Une interruption est générée par :

- un dispositif matériel qui souhaite informer le système d'un événement
- Des instructions particulières dans le code (par exemple, l'instruction `INT`)
    - Lorsqu'un processus appelle un appel système, il génère une interruption logicielle.

Concept de signal

Un **signal** permet de gérer des événements asynchrones qui interrompent le fonctionnement normal d'un processus.

- Il s'agit d'une interruption logicielle
- Notifie un événement à un processus spécifique

Ils peuvent être **générés** par

- le noyau pour communiquer des événements exceptionnels
  - Conditions d'erreur
  - Actions de l'utilisateur (par exemple, `CTRL+C` sur le clavier)
- un autre processus (s'il dispose des autorisations nécessaires) :
  - Permettre une communication primitive entre les processus
  - Utilisation de l'appel système `kill`

Concept de signal

Existe depuis les premières versions d'**Unix**

- Formalisé dans Unix 4

Au début, ils n'étaient pas fiables et étaient gérés au mieux.

- Ils pouvaient être perdus
- La gestion était compliquée
- Peu de configuration possible

Les signaux existent également dans **Windows**, bien qu'ils fonctionnent légèrement différemment

Les signaux sous Linux

Différents types de signaux existent sous Linux

- Dépend des versions de Linux
- La commande `kill -l` liste les signaux
  - 64 dans Ubuntu 20

Chaque signal possède un identifiant numérique et mnémonique.

- Les identificateurs de signaux commencent par les trois caractères `SIG`
- Par exemple, `SIGINT` est le signal d'interruption et porte le numéro 2.
- Les noms symboliques correspondent à un entier positif (`signal.h`)

Les signaux sous Linux

Chaque signal est généré par un événement spécifique dans le système d'exploitation, ou *manuellement* par un processus.
Un signal peut avoir les effets suivants sur un processus :

- Il est ignoré
- Il met fin au processus
- Il crée un **core dump** : un fichier qui contient l'état du programme à déboguer.
- arrête le processus
- Redémarrer le processus

Signaux sous Linux

Signaux ignorés par défaut:

- `SIGCHLD` : envoyé au parent lorsqu'un enfant se termine

Signaux qui terminent le processus par défaut:

- `SIGINT` : envoyé au processus en cours lorsque `CTRL+C` est pressé
- `SIGABRT` : envoyé par l'appel système `abort()`.
- `SIGFPE` : envoyé par une exception arithmétique

Signaux dans Linux

- `SIGHUP` : Envoyé à un processus si le terminal est déconnecté
- `SIGKILL` : moyen sûr de tuer un processus.
  Note : Vous ne pouvez pas créer de gestionnaire pour `SIGKILL`.
- `SIGSEGV` : Accès mémoire invalide
- `SIGTERM` : Signal de fin normalement utilisé. Généré par la commande `kill` par défaut.
- `SIGUSR1` et `SIGUSR2` : Générés uniquement par les processus utilisateurs, jamais par le système d'exploitation. Ils sont utilisés pour la communication entre les processus

Signaux dans Linux

Liste plus complète.

Le comportement par défaut peut être modifié :

- Ignorer un signal
- Pour le *manipuler* via un *manipulateur*.
- **PAS** pour induire une terminaison ou un Core Dump
- Sauf `SIGKILL` et `SIGSTOP`.

Name	Description	Default
SIGABRT	Abort process	Core
SIGALRM	Real-time timer expiration	Term
SIGBUS	Memory access error	Core
SIGCHLD	Child stopped or terminated	Ignore
SIGCONT	Continue if stopped	Cont
SIGFPE	Arithmetic exception	Core
SIGHUP	Hangup	Term
SIGILL	Illegal Instruction	Core
SIGINT	Interrupt from keyboard	Term
SIGIO	I/O Possible	Term
SIGKILL	Sure kill	Term
SIGPIPE	Broken pipe	Term
SIGPROF	Profiling timer expired	Term
SIGPWR	Power about to fail	Term
SIGQUIT	Terminal quit	Core
SIGSEGV	Invalid memory reference	Core
SIGSTKFLT	Stack fault on coprocessor	Term
SIGSTOP	Sure stop	Stop
SIGSYS	Invalid system call	Core
SIGTERM	Terminate process	Term
SIGTRAP	Trace/breakpoint trap	Core
SIGTSTP	Terminal stop	Stop
SIGTTIN	Terminal input from background	Stop
SIGTTOU	Terminal output from background	Stop
SIGURG	Urgent data on socket	Ignore
SIGUSR1	User-defined signal 1	Term
SIGUSR2	User-defined signal 2	Term
SIGVTALRM	Virtual timer expired	Term
SIGWINCH	Terminal window size changed	Ignore
SIGXCPU	CPU time limit exceeded	Core
SIGXFSZ	File size limit exceeded	Core

Les signaux sous Linux

Un processus peut définir un **signal handler**.

- Il s'agit d'une fonction qui est exécutée lorsque le processus reçoit le signal
- Si ce n'est pas le cas, le comportement par défaut est suivi

"Si et quand un signal se produit, exécuter cette fonction"

Les signaux sous Linux

Étapes de la vie d'un signal

1. Génération : par le noyau ou un processus
2. Délivrance : le signal est délivré le plus rapidement possible au processus.
    - Tant qu'un signal n'est pas délivré, il est *en attente*.
3. Traitement :
    - Le noyau lance la fonction de traitement du processus s'il y en a une.
    - Sinon, il exécute l'action par défaut pour ce signal (terminer ou ignorer).

Signaux sous Linux

Observation :

Les signaux ne sont pas mis en file d'attente.

Les signaux en attente pour un processus sont un *masque*.

- Si le même signal est généré plusieurs fois avant d'être délivré, il ne sera délivré qu'une seule fois.

Appel système `sigaction`

```
#include <signal.h>
int  sigaction(int signum, const struct sigaction *act,
               struct sigaction *oldact) ;
```

Modifie le comportement du processus en cours en fonction d'un signal particulier

Arguments :

- `signum` : signal pour le processus
- `act` : pointeur sur la structure définissant le traitement
- `oldact` : pointeur sur le comportement précédent. Peut être utilisé pour restaurer le comportement précédent

Retourne -1 s'il y a eu une erreur

Appel système `sigaction`

```
struct sigaction {
    void (* sa_handler )( int ) ;
    sigset_t sa_mask ;
    int sa_flags ;
    void (* sa_restorer )( void ) ;
} ;
```

- `1sa_handler` spécifie le comportement
  - Si fonction, spécifie un gestionnaire
  - Si `SIG_IGN` ignore
  - Si `SIG_DFL` rétablit le comportement par défaut

Appel système `sigaction`

```
struct sigaction {
    void (* sa_handler )( int ) ;
    sigset_t sa_mask ;
    int sa_flags ;
    void (* sa_restorer )( void ) ;
} ;
```

- `sa_mask` : signaux à bloquer lorsque le gestionnaire est en cours d'exécution Initialisé par la fonction de bibliothèque `int sigemptyset(sigset_t *set);``
- `sa_flags` : flags (non visible)
- `sa_restorer` : pour usage interne

Appel système `sigaction`

Exemple: créer une fonction pour ignorer un signal défini par l'appelant

```
int  ignoreSignal  ( int sig )
{
     struct sigaction sa ;
     sa.sa_handler = SIG_IGN ;
     sa.sa_flags = 0 ;
     sigemptyset ( &sa.sa_mask ) ;
     return sigaction ( sig , &sa , NULL ) ;
}
```

Appel système `sigaction`

La fonction de gestion doit prendre un argument `int`.

- Lorsqu'elle est invoquée par le système d'exploitation, elle contient le numéro du signal

Et retourne `void`

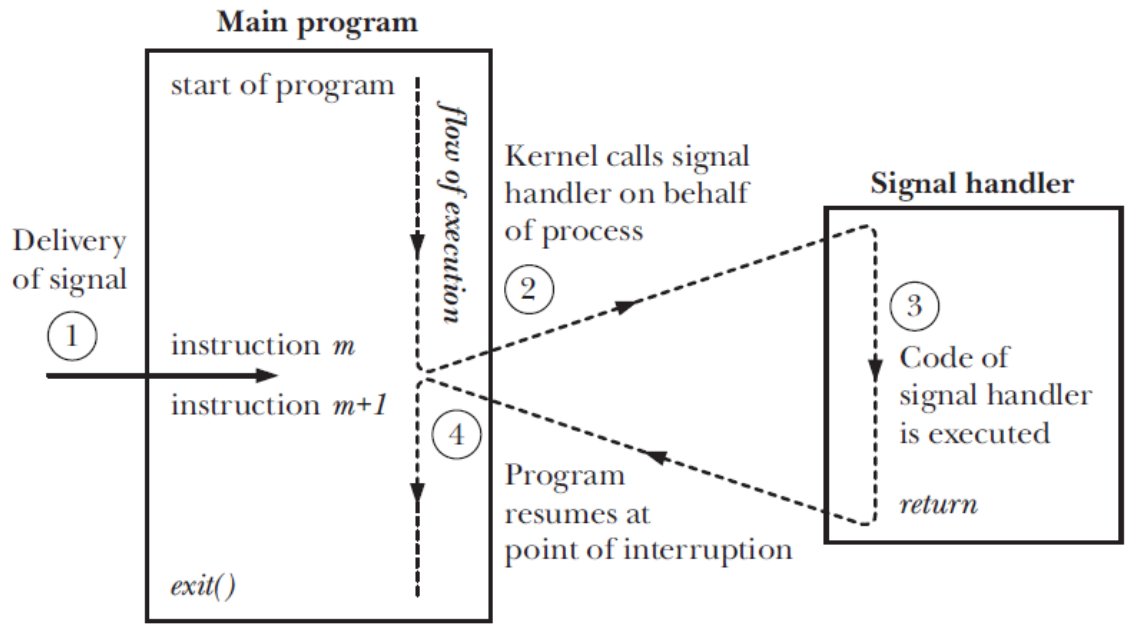
```
void  myHandler  ( int sig )
{
        /* Actions à effectuer lorsque le signal
   est délivré */
}
```

Appel système `sigaction`

Il est automatiquement invoqué par le noyau lors de la réception du signal

Le programme s'arrête, exécute le gestionnaire

Enfin, il reprend l'exécution à partir du point d'interruption



System Call `sigaction`

Exemple : créer un programme qui traite les signaux `SIGINT`, `SIGHUP` e `SIGTERM`

```
#include <signal.h>
#include <stdio.h>
void  func(int signum)
{
   printf("receive %d\n", signum);
}
int  main  (void)
{
   struct sigaction new_action, old_action;

   new_action.sa_handler = func;
   sigemptyset (&new_action.sa_mask); /* Si noti l'uso di sigemptyset */
   new_action.sa_flags = 0;

   sigaction (SIGINT, &new_action, NULL);
   sigaction (SIGHUP, &new_action, NULL);
   sigaction (SIGTERM, &new_action, NULL);

    while(1) ;
}
```

Pour mettre fin au programme, un signal doit lui être envoyé `SIGKILL`.

`pkill –KILL <nome prog>`

Appel de système `sigaction`

Il existe l'appel système de niveau inférieur `signal`.

```
#include <signal.h>
typedef  void  (*sighandler_t)(int)  ;
sighandler_t  signal(int signum, sighandler_t handler)  ;
```

Arguments :

- `sig` : le signal à traiter
- `handler` : spécifie le comportement. Il s'agit d'un pointeur vers une fonction.

Note : recommandé d'utiliser `sigaction`

Appel système `kill`

Envoie un signal à un processus ou à un groupe de processus

```
#include <sys/types.h>
#include <signal.h>
int  kill(pid_t pid, int sig)  ;
```

Arguments :

- `sig` : signal à envoyer
- `pid` :
  - if `> 0` : envoyé au processus identifié par pid
  - if `0` : envoyé à tous les processus du même groupe que le processus invoquant kill
  - if `0` : envoyé au groupe de processus identifié par `-pid`
  - si `-1` : non défini

Appel de système `sigaction`

Exercice : Créer un programme qui génère un processus enfant. Le père envoie au fils un signal `SIGURS1` toutes les secondes. L'enfant imprime le reçu.

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

void  handler(int signum){
   printf("Received\n");
}

int  main  (){
   pid_t  pid;
   struct sigaction action;

   pid = fork();
   if  (pid!=0){  /* Father */
      while(1){
          sleep(1);
          kill (pid, SIGUSR1);
      }
   } else{  /* Child */
      action.sa_handler = handler;
      sigemptyset (&action.sa_mask);
      action.sa_flags = 0;
      sigaction (SIGUSR1, &action, NULL);
      while  (1);
   }
}
```

System Call `raise`

```
#include <signal.h>
int  raise  (int sig) ;
```

Permet à un processus de s'envoyer un signal à lui-même.

En fait, il s'agit de

```
raise (sig)
```

est équivalent à :

```
kill (getpid(), sig) ;
```

Appel système `pause`

```
#include <unistd.h>
int  pause  (void) ;
```

Met le processus en pause jusqu'à l'arrivée d'un signal

Est utilisé pour implémenter l'attente passive d'un signal

Retourne après que le signal ait été capturé et que le gestionnaire ait été exécuté, retourne toujours -1

Appel système `alarm`

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds) ;
```

Implémentation d'un délai d'attente

Le système d'exploitation envoie un signal `SIGALRM` au processus après `secondes`.

S'il n'y a pas déjà eu de timeout, il renvoie 0.

Sinon, il renvoie le nombre de secondes avant l'expiration du dernier délai fixé.

Supprime l'ancien délai et insère le nouveau

Si `seconds` est 0, le timeout est désactivé.

Appel système `alarm`

Observations :

Le délai d'attente est géré par le noyau.

Le temps réel peut être légèrement plus long en raison du temps de réaction du noyau.

Appel système `alarm`

Exemple : Fonction `sleep` implémentée avec `alarm` et `pause`

```
static void myAlarm (int signo) {
     retour ;
}
void mySleep (unsigned int nsecs) {
     signal(SIGALRM, myAlarm)
     alarm(nsecs) ;
     pause () ;
}
```

Considérations

Un handler est un flux d'exécution concurrent

- Il peut démarrer à n'importe quel moment
- Pendant que le flux principal effectue une action

Important :.
Le gestionnaire ne doit pas modifier les variables globales qui sont également utilisées par le flux principal.

- Peut conduire à un état incohérent

Considérations

Exemple :

- Le flux principal lit une variable globale, y ajoute une valeur et écrase la variable.

```
1 int tmp = globalval;
2 tmp = tmp + 10;
3 globalval = tmp;
```

- Le hanlder effectue la même opération

Si le flux principal est interrompu à la ligne `2` par le signal, `globalval` est incrémenté de 10, alors qu'il devrait être incrémenté de 20

- Le flux principal et le gestionnaire auraient tous deux dû être incrémentés de 10.
- Le handler a cependant lu `globalval` lors d'une incrémentation par le flux principal.

Problème que nous verrons longuement dans le cas des programmes multithreadés

Considérations

Définitions :

- Les fonctions qui modifient des variables locales statiques (état interne), ou qui modifient des variables globales, ou qui appellent des fonctions qui ne sont pas réentrantes ne sont pas réentrantes
- **Fonction réentrantes :** peut être utilisée en toute sécurité dans plusieurs flux
- **Fonction pas réentrantes : NE PEUT PAS** être utilisée en toute sécurité dans plusieurs flux.

En général, dans les gestionnaires, il faut :

- N'appeler que des fonctions rentrantes
- Éviter de manipuler des variables globales utilisées par le flux principal

Considérations

La plupart des fonctions de la bibliothèque C ne sont pas réentrantes

- `printf`, `scanf`
- Elles ne doivent pas être appelées à l'intérieur d'un handler !

Certaines fonctions de la bibliothèque standard ne sont pas réentrantes, mais ont une version réentrante correspondante (désignée par `_r`), par exemple `strtok` (`strtok_r`), `readdir` (`readdir_r`), `rand` (`rand_r`).

Note : un programme peut recevoir un signal pendant que l'un de ses appels système (par exemple, `read`) est en cours d'exécution.
Le noyau interrompt l'appel système et exécute les gestionnaires.
Selon le cas, il reprend après le handler.

Signaux dans la shell

```
kill pid
```

Envoie un signal au processus `PID`.

Par défaut, il envoie `SIGTERM`.

Peut être spécifié avec les options `-KILL` (ou `-9`) `-INT` (ou `-2`).

```
pkill nom
killall nom
```

Même comportement, mais envoie le signal à tous les processus du programme `nom`.

Signaux dans la shell

Exercice : Ecrir un programme en C qui enregistre le nombre de `SIGTERM` qu'il a reçu. Lorsque vous appuyez sur `CTRL+C`, il imprime ce nombre et se termine. Nommez le programme `sample`.

Ecrir aussi un script bash qui envoie 10 signaux `SIGTERM` au processus.

Programme bash:

```
for i in $( seq 5) ; do
    pkill sample
done
```

Signaux dans la shell

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

int c;

void handler(int signum){
  if (signum==SIGTERM)
    c+=1;
  else if (signum==SIGINT){
    printf("Received %d SIGTERM\n", c);
    exit (0);
  }
}

int main (){
    struct sigaction action ;
    c=0;
    action.sa_handler = handler;
    sigemptyset (&action.sa_mask);
    action.sa_flags = 0;
    sigaction (SIGTERM, &action, NULL);
    sigaction (SIGINT, &action, NULL);

    while (1);
}
```

Signaux dans la shell

Lorsque vous appuyez sur `CTRL+C`, un `SIGINT` est envoyé au programme, qui imprime `c` et se termine

Script bash qui automatise toute la séquence : démarre le programme C, envoye des signaux, et termine

```
./sample &
PID=$!
for i in $( seq 5) ; do
    kill $PID
done
kill -INT $PID
```

Signaux dans le shell

Si `CTRL+C`, `SIGINT` est envoyé

- Le programme se termine s'il n'y a pas de gestionnaire

Si `CTRL+Z`, `SIGTSTP` est envoyé

- Par défaut, l'application est suspendue
- Et mise en arrière-plan par l'interpréteur de commandes
- A ce stade :
  - `fg` reprend l'exécution au premier plan
  - `bg` reprend l'exécution en arrière-plan

Signaux dans le shell

Très utile : si j'ai lancé une longue commande et que je veux utiliser l'interpréteur de commandes pendant qu'elle s'exécute.

```
$ ./longjob
^Z
[1]+ Stopped ./longjob
$ bg
[1]+ ./longjob &
$ free terminal
```

Signaux dans le shell

Lorsque je lance un programme en arrière-plan (`./job &`) et que je ferme le terminal, le signal de raccrochage `SIGHUP` est envoyé.

- Par défaut, le programme se termine
- Vous pouvez changer le comportement

Ou j'utilise la commande `nohup` , qui exécute une commande immunisée contre `SIGHUP`

```
nohup ./job
```

Utile si je lance des travaux sur un terminal distant et que je dois rentrer chez moi !
Alternative plus propre : la commande `screen` qui génère un terminal virtuel

Signaux dans le shell

Gestionnaire de signaux dans les scripts bash

```
command trap SIGNAL
```

Exécute la commande ou la fonction `command` si le script reçoit le signal `SIGNAL`.
Il existe un pseudo-signal supplémentaire, `EXIT`, appelé lorsque le script se termine

Exemple typique :

```
tempfile=/tmp/tmpdata
trap "rm -f $tempfile" EXIT
```

Signaux dans le shell

Créer un programme bash qui compte combien de signaux `SIGUSR2` il reçoit, et les affiche quand `CTRL+C` est pressé, et le nommer `sample.sh`

```
#!/bin/bash

count=0
function husr(){
     let count++
}
function hint(){
     echo "Received $count SIGUSR2"
     exit 0
}

trap husr SIGUSR2
trap hint SIGINT

while true; do
     sleep 1
done
```

Envoyer les signaux avec la commande : `bash pkill –USR2 sample.sh`.

Note : déclaration de fonction en Bash

Sychronisation

Sujets

1. Objectifs
2. Les *pipes*
3. Les *FIFOs*
4. Mémoire virtuelle partagée
5. Mémoire partagée avec `shmget`
6. Mémoire partagée avec `mmap`
7. Problèmes

Objectifs

Dans un système d'exploitation, plusieurs processus s'exécutent simultanément. Ils peuvent être classés en :

- **Processus indépendants** : ils n'influencent pas les autres processus et ne sont pas influencés par eux.
- **Processus coopératifs** : ils interagissent avec d'autres processus. Ils doivent utiliser des mécanismes appropriés pour ce synchroniser

Objectifs

Tous les systèmes d'exploitation fournissent des outils pour la **communication inter-processus**.

Ils sont généralement basés sur :

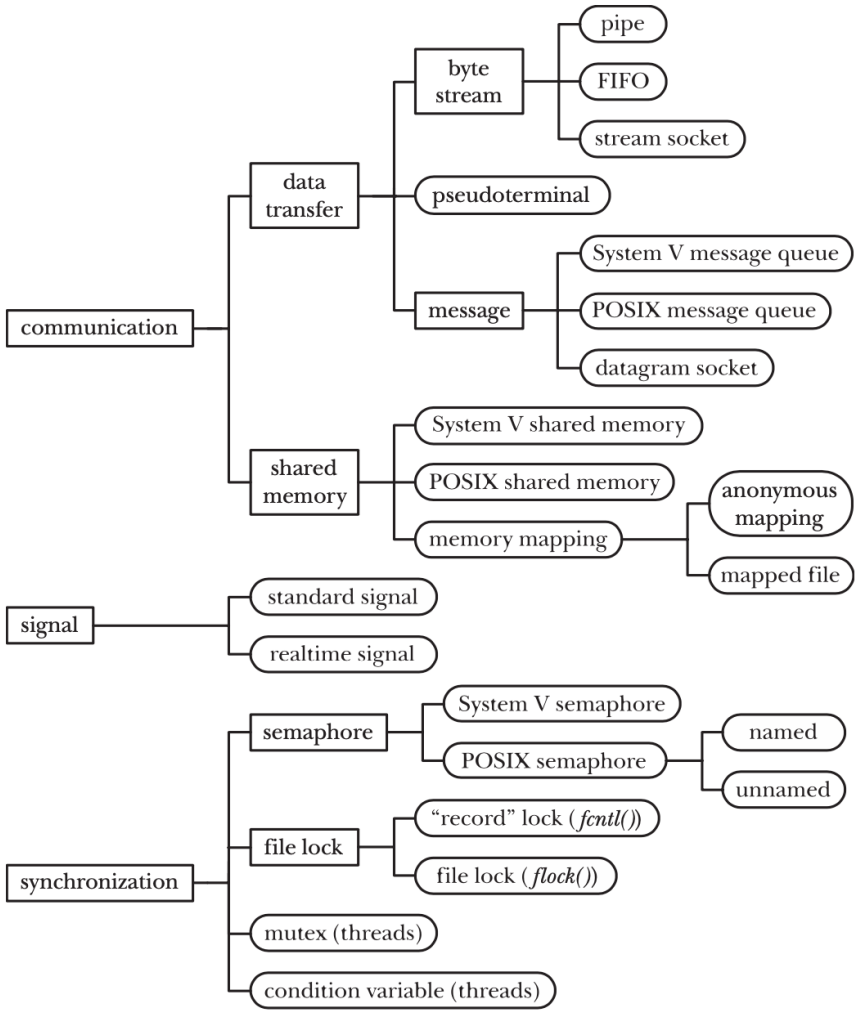
- **L'échange de messages**
- **Échange de données**
- **Mémoire partagée**

Objectifs

Chaque système d'exploitation utilise des mécanismes différents.

Dans Linux, il existe de nombreux mécanismes.

- Historiquement en couches
- Hérités de System V (UNIX)
- Fait partie des normes POSIX



Objectifs

Nous voyons :

- **Échange de messages :** Signaux (déjà vu)
- **Échange de données :** *Pipe* et *FIFO*
- **Mémoire partagée :** utilisation des appels système `shmget` et `mmap`.

Les *pipes*

Les *pipes* sont la forme la plus ancienne et la plus largement utilisée d'IPC introduite dans Unix.

- Ils permettent l'échange de données entre les processus
- Modèle **producteur-consommateur**.
- Ils sont utilisés avec le même appel système que les fichiers : `read`, `write`.
- Ils résident dans la mémoire
- Ils ne sont pas persistants : lorsque les processus se terminent, tout ce qui reste est détruit.

Les *pipes*

Limitations

- Ils sont half-duplex (communication à sens unique)
- Ils ne peuvent être utilisés qu'entre des processus ayant un "ancêtre" commun.

Comment surmonter ces limitations

- Le *FIFO* (ou *pipe nommé*) peut être utilisé entre plusieurs programmes
  - Ils sont identifiés par un nom

Les *pipes*

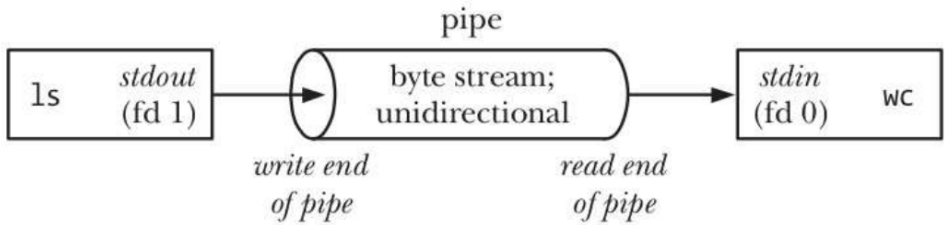
Les *pipes* sont couramment utilisés dans le shell, pour rediriger `stdout` et `stdin`.

Exemple:

```
ls | wc -l
```

Pour ce faire, l'interpréteur de commandes

- Utilise deux `fork` et `exec` pour créer les processus `ls` et `wc`.
- Crée un *pipe* pour connecter le `stdout` de `ls` avec le `stdin` de `wc`.



Les *pipes*

Définition

Les pipes sont un **flux d'octets**

- Les octets sont écrits/lus
- Pas seulement des caractères imprimables

Ils sont **unidirectionnels** :

- Ils ont une entrée et une sortie

Ils ont une **capacité limitée** :

- Les données en file d'attente (écrites mais pas encore lues) ne peuvent pas dépasser un seuil.
- Seuil configurable : 65 KB par défaut
  - Peut être modifié avec `fcntl(fd, F_SETPIPE_SZ, size)`

Les *pipes*

Création:

```
#include <unistd.h>
int  pipe  (int filedes [2])  ;
```

Renvoie deux descripteurs de fichiers via l'argument fd (passé par référence)

- `fd[0]` est ouvert à la lecture
- `fd[1]` est ouvert en écriture
- La sortie de `fd[0]` est l'entrée de `fd[1]`.

Les *pipes*

I/O on *pipes*:

Les fonctions `read` et `write` sont utilisées

- La valeur de retour est le nombre d'octets lus/écrits.

Lecture :

- La fonction `read` est bloquante jusqu'à ce qu'au moins un octet soit lu.

Écriture :

- Si le *pipe* est plein, le `write` est bloquant.

Les *pipes*

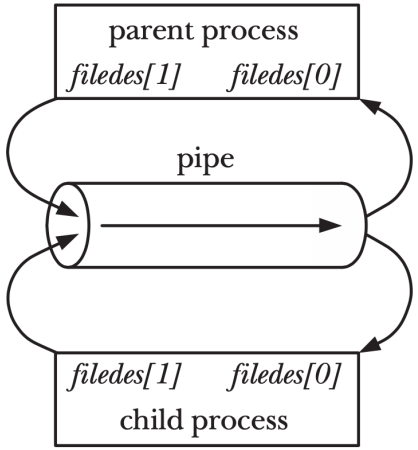
Note: si j'écris dans une *pipe* qui n'a pas de *lecteur* (`fd[0]` a été fermé), le processus reçoit le signal `SIGPIPE`.
Fin du processus, s'il n'y a pas de **Signal Handler** approprié

Les *pipes*

Partage entre processus :

Pour utiliser une pipe entre plusieurs processus :

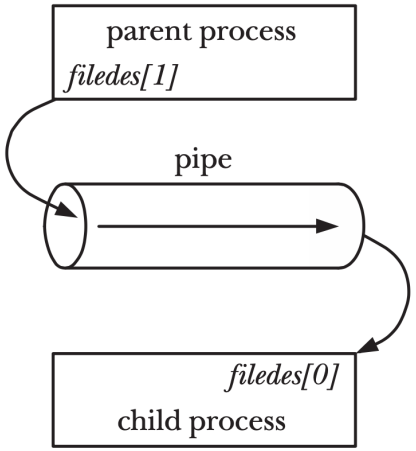
- Le processus parent crée la pipe et obtient les deux `fd`.
- Il fait un `fork`
- Les deux processus peuvent accéder au *pipe* en utilisant les deux `fd`.



Les *pipes*

Fermeture d'une pipe

- Habituellement, un processus (par exemple, le père) écrit, et un autre (par exemple, le fils) lit.
- Techniquement, il est possible pour un processus de lire et d'écrire.
  - Cependant, cela crée des problèmes de synchronisation
- Chaque processus ferme `fd` qu'il n'utilise pas



Les *pipes*

Exemple:

```
int pfd [2] ;
pipe ( pfd ) ;
switch ( fork () ) {
case -1 : exit(1)
case 0 :
     /* Enfant */
     close ( pfd [1]) ;
     /* Peut maintenant lire */
     break ;
default :
     /* Parent */
     close ( pfd [0]) ;
     /* Écriture sur la pipe */
     break ;
}
```

Les *pipes*

Fermeture d'une pipe:

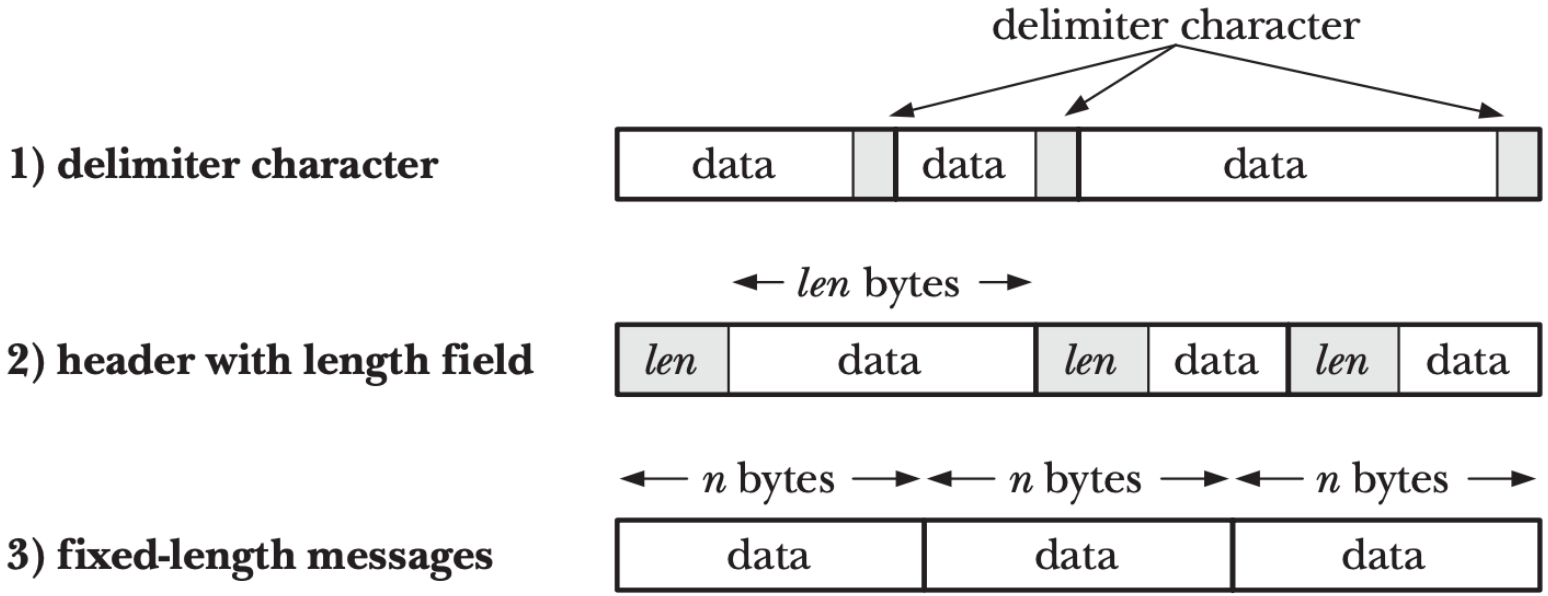
Le `read` bloque jusqu'à ce qu'au moins 1 octet soit lu

Si le `read` retourne 0, cela signifie qu'il n'y a pas de `fd` ouvert pour l'écriture.

- La *pipe* est mort

Messages sur les pipes

Il existe plusieurs stratégies pour échanger des messages via *pipe*



Les *pipe*

Exercice: créer un programme avec deux processus. Le processus père reçoit une chaîne de caractères de la ligne de commande et la transmet au processus fils via un *pipe*. Le fils reçoit la chaîne et l'imprime.

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#define MAXLINE 1024

int  main(int argc, char *argv[])
{
        int pfd[2], status;
        char line[MAXLINE];

        pipe(pfd);
        if (fork() > 0) {
                close(pfd[0]);
                write(pfd[1], argv[1], strlen(argv[1]));
                wait(&status);
        } else {
                close(pfd[1]);
                read(pfd[0], line, MAXLINE);
                printf("Received: %s\n", line);
        }
        exit(0);
}
```

Les *pipes*

Exercice: créer un programme avec deux processus. Le processus parent reçoit le nom d'un fichier à partir de la ligne de commande et transmet son contenu au processus fils via un *pipe*. Le fils reçoit le contenu et l'imprime.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXSIZE 1000

int  main(int argc, char * argv[]){
     int  pfd[2], status;

     pipe(pfd);
     if(fork()>0){
          close(pfd[0]);
          FILE * fp;
          char  line[MAXSIZE];

          fp=fopen(argv[1],"r");
          while(fgets(line, sizeof(line), fp)!=NULL)
             write(pfd[1], line, sizeof(line));
          close(pfd[1]);
          wait(&status);
          exit(0);
     }
     else {
          char buffer [MAXSIZE];
          close(pfd[1]);
          while (read(pfd[0], buffer, sizeof(buffer)) > 0)
          exit(0);
     }
}
```

Les *FIFO*

Les pipes "normaux"

- Ne peuvent être utilisés que par des processus ayant un "ancêtre" commun.
- Raison : seul moyen d'hériter de descripteurs de fichiers.

Les pipes nommés ou *FIFO*

- Ils permettent à des processus non apparentés de communiquer
- Ils utilisent le système de fichiers pour "nommer" la pipe
- Le *FIFO* est un type de fichier
  - La macro `S_ISFIFO` après un `stat` retournera `true`.
- La procédure de création d'un fifo est similaire à la procédure de création de fichiers.

Les *FIFO*

```c
#include <sys/types.h>
#include <sys/stat.h>
int  mkfifo(const char *path, mode_t mode);
```

Crée un *FIFO* à partir du `pathname` spécifié.
L'argument `mode` fonctionne comme dans `open` (`O_RDONLY`, `O_WRONLY`, `O_RDWR`, etc.).
Valeur de retour : 0 en cas de succès, -1 en cas d'erreur

Utilisation:

Comme les fichiers et les *pipes* : via `read` et `write`.
Tout processus ayant les permissions pour `pathname` peut l'utiliser

Les *FIFO*

Opening:

Ouvrir un fichier sans le flag `O_NONBLOCK`.

- Si le fichier est ouvert en lecture, l'appel se bloque jusqu'à ce qu'un autre processus ouvre la FIFO en écriture.
- Si le fichier est ouvert en écriture, l'appel se bloque jusqu'à ce qu'un autre processus ouvre la FIFO en lecture.

Fichier ouvert avec le flag `O_NONBLOCK`.

- Si le fichier est ouvert en lecture, l'appel revient immédiatement
- Si le fichier est ouvert en écriture et qu'aucun autre processus ne l'a ouvert en lecture, l'appel renvoie un message d'erreur

Les *FIFO*

Input/Output:

Avec `read` et `write`

Les données dans le *FIFO* sont mises dans un buffer par le noyau.

Important:

Un *FIFO* a un nom de chemin, mais ce n'est qu'une esquive pour permettre à différents processus d'y accéder.

Lorsqu'un *FIFO* est fermé (ou que les processus se terminent), le nom du fichier persiste dans le système de fichiers, mais il ne contient aucune donnée.

Les *FIFO*

Vous pouvez facilement créer et utiliser des *FIFOs* dans Bash :

```
mkfifo monfifo
tr 'aeiou' 'AEIOU' < myfifo &
man 2 pipe > myfifo
```

Les *FIFO*

Exercice: créer un programme qui lit à partir d'une FIFO et imprime le contenu en majuscules.

```
#include <stdio.h>
#include <ctype.h>
#include <sys/stat.h>
#include <string.h>
#include <stdlib.h>

int  main(int argc, char * argv[]){
     int  i, n, l ;
     FILE * f ;
     char  buffer[512] ;

     i = mkfifo("myfifo", S_IRWXU) ;
     if (i<0){
          printf("Impossible de créer la FIFO") ;  /* Il se peut qu'elle existe déjà */
     }

     f = fopen("myfifo", "r") ;
     if (f==NULL){
          printf("Impossible d'ouvrir la FIFO") ;
          exit(1) ;
     }
     while(fgets(buffer,sizeof(buffer),f)!=NULL){
          l = strlen(buffer) ;
          for (i=0 ; i<l ; i++)
               putc(toupper(buffer[i]), stdout) ;
     }
}
```

Texte avec : `echo "hello world" > myfifo`

Mémoire virtuelle partagée

Chaque processus dispose d'un espace d'adressage virtuel dédié

- Il existe une table de pages par processus
- Isolation de la mémoire entre les processus
  - Essentiel pour la sécurité
  - Ne permet pas le partage de la mémoire

Pour partager la mémoire, une ou plusieurs pages doivent être partagées.

- Le système d'exploitation fournit des appels système à cette fin

Mémoire virtuelle partagée

Il existe deux séries d'appels système pour la mémoire partagée entre les processus sous Linux :

- `shmget shmat shmdt ftok`
- `mmap munmap shm_open shm_unlink`.

L'approche `mmap` est plus moderne et plus flexible

Sous Windows, vous utilisez l'appel système `CreateFileMapping`.

Mémoire partagée avec `shmget`

```c
#include <sys/shm.h>
int  shmget(key_t key, size_t size, int shmflg);
```

Crée un segment de mémoire partagée.
Arguments :

- `key` : identifiant défini par l'utilisateur. Utilisez `IPC_PRIVATE` si vous êtes anonyme (utilisé uniquement avec `fork`)
- `size` : taille de la mémoire partagée
- `shmflg` : flags et permissions. `IPC_CREAT` créer si inexistant.
  - Utilisation typique `IPC_CREAT | 0666`.

Valeur de retour :

- Un identifiant de la zone créée
- -1 en cas d'échec

Mémoire partagée avec `shmget`

```
#include <sys/shm.h>
void  *shmat(int shmid, const void *shmaddr, int shmflg);
```

Met en correspondance le segment de mémoire virtuelle dans l'espace d'adressage du processus.
Arguments :

- `shmid` : identifiant retourné par `shmget`
- `shmaddr` : si non nul, la mémoire est mappée à `shmaddr` (arrondi à la *taille de la page*).
- `shmflg` : drapeau. `SHM_RDONLY` correspond à lecture seule

Valeur de retour :

- L'adresse virtuelle du segment mappé
- -1 en cas d'échec

Mémoire partagée avec `shmget`

```
#include <sys/shm.h>
int  shmdt(const void *shmaddr);
```

Supprime dans le mappage le segment à l'adresse virtuelle `shmaddr`.

Valeur de retour :

- 0 en cas de succès
- -1 en cas d'échec

Mémoire partagée avec `shmget`

```
#include <sys/ipc.h>
key_t  ftok(const char *pathname, int proj_id);
```

Crée une `clé` à partir d'un chemin, en s'assurant que :

- Deux chemins donneront toujours des clés différentes
- Le même chemin et le même `proj_id` donneront toujours les mêmes clés.
- Empêche différents programmes ayant choisi la même clé d'utiliser la même mémoire.

Arguments :

- `pathname` : le chemin
- `proj_id` : utilisé pour créer la `clé`. Ne doit pas être nul.

Valeur de retour :

- key en cas de succès
- -1 en cas d'échec

Mémoire partagée avec `shmget`

Utilisation : Mémoire partagée avec l'enfant créé via `fork`.

```
int shmid = shmget(IPC_PRIVATE, 1*sizeof(int), IPC_CREAT | 0666);
void * shm = shmat(shmid, NULL, 0);
if (fork()){ /* Pere */
    ...
} else { /* Enfant */
    ...
}
shmdt(shm) ;
```

Mémoire partagée avec `shmget`

Utilisation: Mémoire partagée entre deux processus indépendants

- Processus créateur

```
key_t key = ftok(path, proj) ;
int shmid = shmget(key, size, IPC_CREAT | 0666)) ;
void * data = shmat(shmid, NULL, 0) ;
...
shmdt(data) ;
```

- Processus utilisateur

```
key_t key = ftok(path, proj) ;
int shmid = shmget(key, size, 0666)) ;
void * data = shmat(shmid, NULL, 0) ;
...
shmdt(data) ;
```

Mémoire partagée avec `shmget`

Exercice: Créez deux programmes qui partagent la mémoire avec `shmget`.
Le premier programme vous permet d'écrire une chaîne de caractères en mémoire, tandis que le second vous permet de la lire.

Program 1:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int  main(){
     key_t  key  =  ftok("test",123);
     if  (key  <0){printf("Error. Does the file for the exist?");  exit(1);}
     printf("Key: %d\n",  key);

     int  shmid  =  shmget(key,1024,0666|IPC_CREAT);
     char  *str  =  (char*)  shmat(shmid,(void*)0,0);
     while(1){
          printf("Input Data : ");
          scanf(" %s",  str);
     }
     shmdt(str);
     return  0;
}
```

Mémoire partagée avec `shmget`

Program 2:

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
int  main(){
     key_t  key  =  ftok("test",123);
     if  (key  <0){printf("Error. Does the file for the exist?");  exit(1);}
     printf("Key: %d\n",  key);

     int  shmid  =  shmget(key,1024,0666|IPC_CREAT);
     char  *str  =  (char*)  shmat(shmid,(void*)0,0);
     while(1){
          printf("Press enter to read");
          getchar();
          printf("Data: %s\n\n",  str);
     }

     shmdt(str);
     return  0;
}
```

Mémoire partagée avec `mmap`

```
#include <sys/mman.h>
void  *mmap(void *addr, size_t length, int prot, int flags,
       int fd, off_t offset);
```

Crée une zone de mémoire partagée.

Arguments :

- `addr` : si non nul, la mémoire est mappée à `addr` (arrondi à la *taille de la page*)
- `length` : taille
- `prot` : peut être : `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`, `PROT_NONE`
  - C'est-à-dire que la page peut être lue, écrite, exécutée, mais qu'elle n'est pas accessible.
  - Normalement, `prot = PROT_READ|PROT_WRITE`.
- `flags` détermine si les changements sont visibles ou non par les autres processus
  - `MAP_ANONYMOUS, MAP_SHARED, MAP_PRIVATE`
  - Normalement `flags=MAP ANONYMOUS | MAP SHARED`
- `fd offsets` est utilisé pour faire correspondre la mémoire à un fichier

Mémoire partagée avec `mmap`

```
#include <sys/mman.h>
void  *mmap(void *addr, size_t length, int prot, int flags,
        int fd, off_t offset);
```

Valeur de retour :

- L'adresse virtuelle du segment mappé
- -1 en cas d'échec

Mémoire partagée avec `mmap`

```
int  munmap(void *addr, size_t length);
```

Supprime le mappage et rend la mémoire disponible à l'adresse `addr`.

Mémoire partagée avec `mmap`

.

Utilisation: il y a trois façons d'utiliser `mmap`

1. Zone de mémoire anonyme : utilisée avec `fork`

```
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0) ;
```

2. Zone de mémoire mappée au fichier :

```
fd = open('/home/wontoniii/file.txt', O_RDWR|O_CREATE) ;
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0) ;
```

3. Zone de mémoire mappée au fichier temporaire :

```
fd = shm_open('temporary.txt', 'rw') ;
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0) ;
```

Mémoire partagée avec `mmap`

Zone mémoire anonyme :

- La zone mémoire n'a pas de nom
- Seul un enfant né avec une `fork` peut y accéder
- Facile à utiliser

```
void* shmem = mmap(NULL, size, PROT_READ | PROT_WRITE,
                   MAP_PARTAGÉE | MAP_ANONYME, -1, 0) ;

if (fork()){ /* Père */.
    ...
} else { /* Fils */
    ...
}
munmap(shmem, size) ;
```

Mémoire partagée avec `mmap`.

Zone de mémoire mappée à un fichier:

- Un fichier est utilisé comme conteneur
- Le contenu de la zone partagée sera sauvegardé dans un fichier.
- Plusieurs programmes peuvent accéder à la mémoire partagée
  - Le chemin d'accès doit être identifié
- Efficace mais lent à cause du disque

Mémoire partagée avec `mmap`

Zone de mémoire mappée à un fichier:

Le fichier doit être suffisamment grand pour contenir la région mappée

```
#include <unistd.h>
int  truncate(const char *path, off_t length)  ;
int  ftruncate(int fd, off_t length)  ;
```

S'assure que le fichier ouvert `fd` ou le fichier `path` est au moins long de `length`.

- Si nécessaire, le fichier est tronqué
- Si nécessaire, il est étendu et rempli avec les caractères `'\0` (ou `0X00` en hexadécimal).

Mémoire partagée avec `mmap`

Zone de mémoire mappée à un fichier:

Flux typique :

```
fd = open(path, O_RDWR|O_CREATE) ; /* Pas un fopen()*/
ftruncate(fd, size) ;
void* shmem = mmap(NULL, size, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0) ;
...
munmap(shmem, taille) ;
```

Mémoire partagée avec `mmap`.

Zone de mémoire mappée dans un fichier temporaire

- Il est parfois nécessaire d'avoir un fichier temporaire identifiable
  - Pour une utilisation par plusieurs programmes
- Une zone de mémoire mappée sur un fichier est inutilement lente.
  - Si les données de la zone mémoire n'ont pas besoin de survivre

Vous pouvez utiliser les fonctions `shm_open` et `shm_unlink`.

- Elles créent et suppriment un fichier temporaire
- Qui est situé dans le dossier `/dev/shm/` qui a monté un FS temporaire (`tmpfs`)
- Les données sont **en mémoire**

Mémoire partagée avec `mmap`

Zone de mémoire mappée au fichier temporaire

```
#include <sys/mman.h>
int  shm_open(const char *name, int oflag, mode_t mode) ;
int  shm_unlink(const char *name) ;
```

Crée et supprime des zones de mémoire temporaire.

- Sémantique analogue à `open` et `unlink`.
- Opère sur les zones temporaires de mémoire/fichier
- `name` est un nom de fichier, pas un chemin complet
  - Tous les emplacements de mémoire sont des fichiers sous `/dev/shm/`.

Mémoire partagée avec `mmap`

Zone de mémoire mappée au fichier temporaire

Flux typique :

```
int fd = shm_open(nome, O_RDWR, 0);
ftruncate(fd, size);
void * mem = mmap(NULL, size, PROT_READ | PROT_WRITE,
                  MAP_SHARED, fd, 0);
...
munmap(shmem, size);

/* Pas obligatoire */
shm_unlink(nome)
```

Mémoire partagée avec `mmap`

Exercice: Créez deux programmes qui ont une mémoire partagée avec `mmap` et `shm_open`.
Le premier programme vous permet d'écrire une chaîne en mémoire, tandis que le second vous permet de la lire.
Program 1:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* For O_RDWR */
int  main(){
     int  fd;
     char  * mem;

     fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
     ftruncate(fd, 512);
     mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     while(1){
          printf("Input data: ");
          scanf("%s", mem);
     }
     munmap(mem, 512);
}
```

Mémoire partagée avec `mmap`

Programma 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h> /* For O_RDWR */
int  main(){
     int  fd;
     char  *  mem;

     fd = shm_open("mymem", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
     ftruncate(fd, 512);
     mem = (char *) mmap(NULL, 512, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     while(1){
          printf("Press enter to read data");
          getchar();
          printf("Data: %s\n\n", mem);
     }
     munmap(mem, 512);
}
```

Nota: Faut compiler avec `gcc prog.c –lrt –o prog`

pour inclure `librt, libposix4 – POSIX.1b Realtime Extensions library`

Problèmes

L'utilisation de la mémoire partagée est complexe

- Préférez les *pipes* ou *FIFO* lorsque c'est possible.

La mémoire partagée pose des problèmes de synchronisation et de "conditions de course".

- Des processus concurrents peuvent lire des données dans un état incohérent
- Alors qu'un autre processus les modifie
- Comme nous l'avons déjà vu avec les signaux

Problèmes

Exemple:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

La variable A est augmentée de 100.

- La diminution de 50 est perdue

Problèmes

Pour surmonter ces problèmes, il existe des **techniques de synchronisation**.

- Elles empêchent un processus d'être interrompu pendant qu'il effectue une opération critique
- Elles permettent à un processus d'attendre qu'une condition se produise.

On verra aprés...

UNIX Sockets

UNIX Sockets

Les **Sockets** sibt un outil de communication inter-processus pour l'échange de données entre différents **nœuds d'un réseau**.

Utilisation similaire à celle des *pipes* et des *FIFO*.

- Identifiée par un *descripteur de fichier*.
- Accédé avec les appels système `read` et `write`.

Contrairement aux *pipes* et aux *FIFOs*

- Ils connectent des nœuds différents
- Ils sont créés différemment avec des appels système dédiés

Types de sockets

Il existe quatre types de sockets :

- **UNIX Sockets** : permet la communication entre les processus du même nœud
- **Stream Sockets** : permet la communication via TCP
- **Datagram Sockets** : permet la communication via UDP
- **Raw Sockets** : permet la communication via des paquets IP bruts.

Basé sur le modèle **client/serveur**.

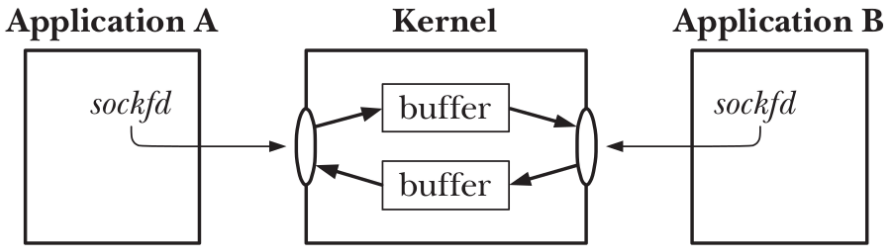
UNIX Socket

Communication entre les processus d'un même noeud

- Conceptuellement **très similaire** à un *pipe* ou *FIFO*.

Différence

- Elles utilisent le modèle **client/serveur**.
- Un serveur écoute
- Un client contacte le serveur et entame la communication

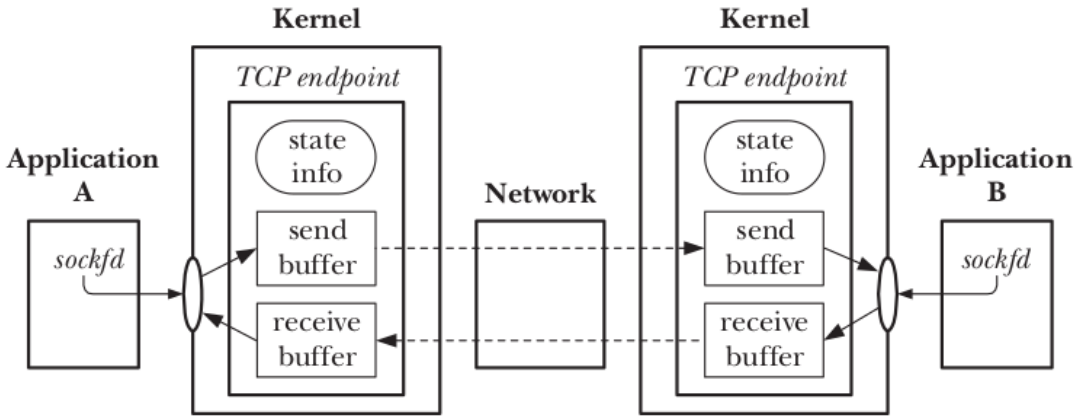


Stream Socket

Communication via TCP

- Service orienté connexion
- Le client et le serveur communiquent via un flux d'octets

Similaire à un *pipe* ou *FIFO* entre **différents noeuds**



Datagram Socket

Communication via UDP

- Le client et le serveur échangent des **messages**
- Service peu fiable
  - Perte possible de paquets

Différences

- Datagram sockets :
  - Orientée vers les **messages**
  - Non fiable
- Socket UNIX et Stream Socket :
  - Orienté vers le **flux**
  - Fiable

Sockets actives/passives

Les sockets sont basées sur le modèle **client/serveur**.

Une **socket passive** attend les connexions entrantes

- Implémente un serveur

Une **socket active** est effectivement connectée à un autre nœud

- Permet l'échange de données
- Utilisé par un client pour communiquer avec le serveur
- Également utilisé par le serveur, **après** avoir accepté une nouvelle connexion