Les processus

Les processus

Un processus est un programme en cours d'exécution.

Un programme est donc un **élément passif** (un ensemble d'octets sur le disque) tandis qu'un processus est un **élément actif** (un programme en cours d'exécution).

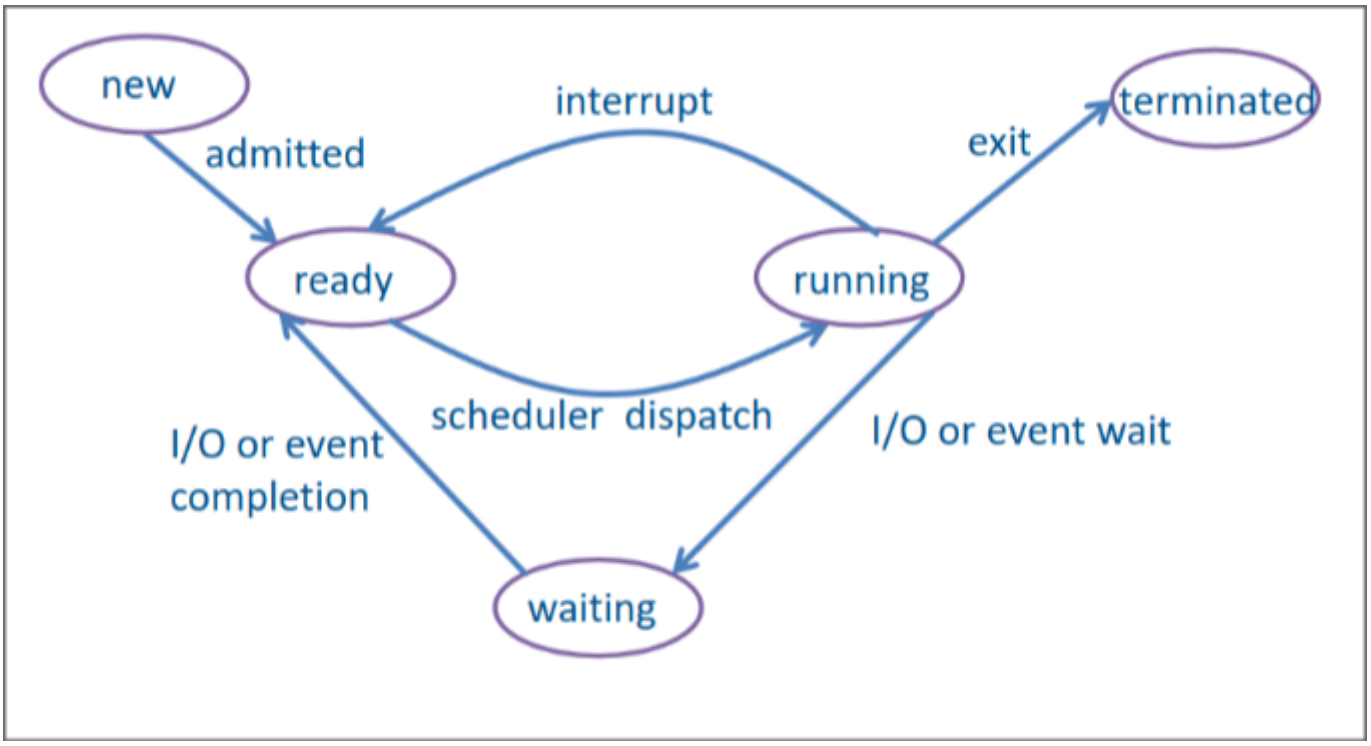
Que comporte un processus ?

- Le code du programme
- Le program counter (à quel instruction on est dans le programme, qui permet de savoir quelle sera la suivante) et les registres
- La pile (stack) et les données du programme

Informations concernant le processus

- PID (process ID) qui est l'identifiant du processus
- PPID (parent process id) qui est l'identifiant du processus parent
- Priorité du processus
- Temps CPU : temps consommé au CPU
- Tables des fichiers
- Etat du processus

Etat



- **new** correspond à un programme qui a été sélectionné pour être démarré, ses instructions ont été recopiées en mémoire par l'OS et un nouveau processus y a été attaché mais pas encore exécuté, son contexte d'exécution et ses détails n'ont pas encore été préparés.
- **ready** le processus a été créé et dispose de toutes les ressources pour effectuer ses opérations
- **running** le processus a été choisi par le scheduler pour tourner, il va donc exécuter ses instruction jusqu'a écoulement du temps imparti. Si il a besoin de plus de ressource, il passe dans l'état *waiting*, si il a terminé son exécution il passe en état *terminated* sinon il peut encore passé en *ready* si un processus de plus haute priorité arrive.
- **waiting** le processus est en attente d'un évènement (exemple appui d'un bouton ou écoulement d'un certain temps) ou de ressources (exemple lecture de disque). Le processus ne peut rien faire pour l'instant.
- **terminated** une fois que le processus est terminé (ou a été tué), il libère la totalité des ressources qu'il a détenues.

Vous pouvez avoir plus d'information sur ce sujet en consultant ce site.

Pour exécuter plusieurs processus

Le système alterne très vite entre les différents états pour donner l'illusion que plusieurs processus s'exécutent en même temps.

En somme on garde en mémoire les processus, le **scheduler** va choisir les processus à exécuter; lorsqu'un processus est en attente un autre processus va être sélectionné pour être exécuté. Le but du scheduler est de maximiser l'utilisation du CPU.

Le scheduler

Le scheduler va sélectionner le processus à exécuter, c'est lui qui va alterner entre les différents états de chaques processus.

Le scheduler utilise un algorithme précis et il doit être le plus rapide possible.

Le scheduler classifie les processus selon leur type :

- Processus CPU (calculs)
- Processus E/S (I/O, entrée sortie)

On va toujours vouloir privilégier les processus entrée-sorties, qui sont ceux qui dialogues avec l'utilisateur et qui vont donner l'illusion que les choses d'exécutent en même temps.

Changement de contexte

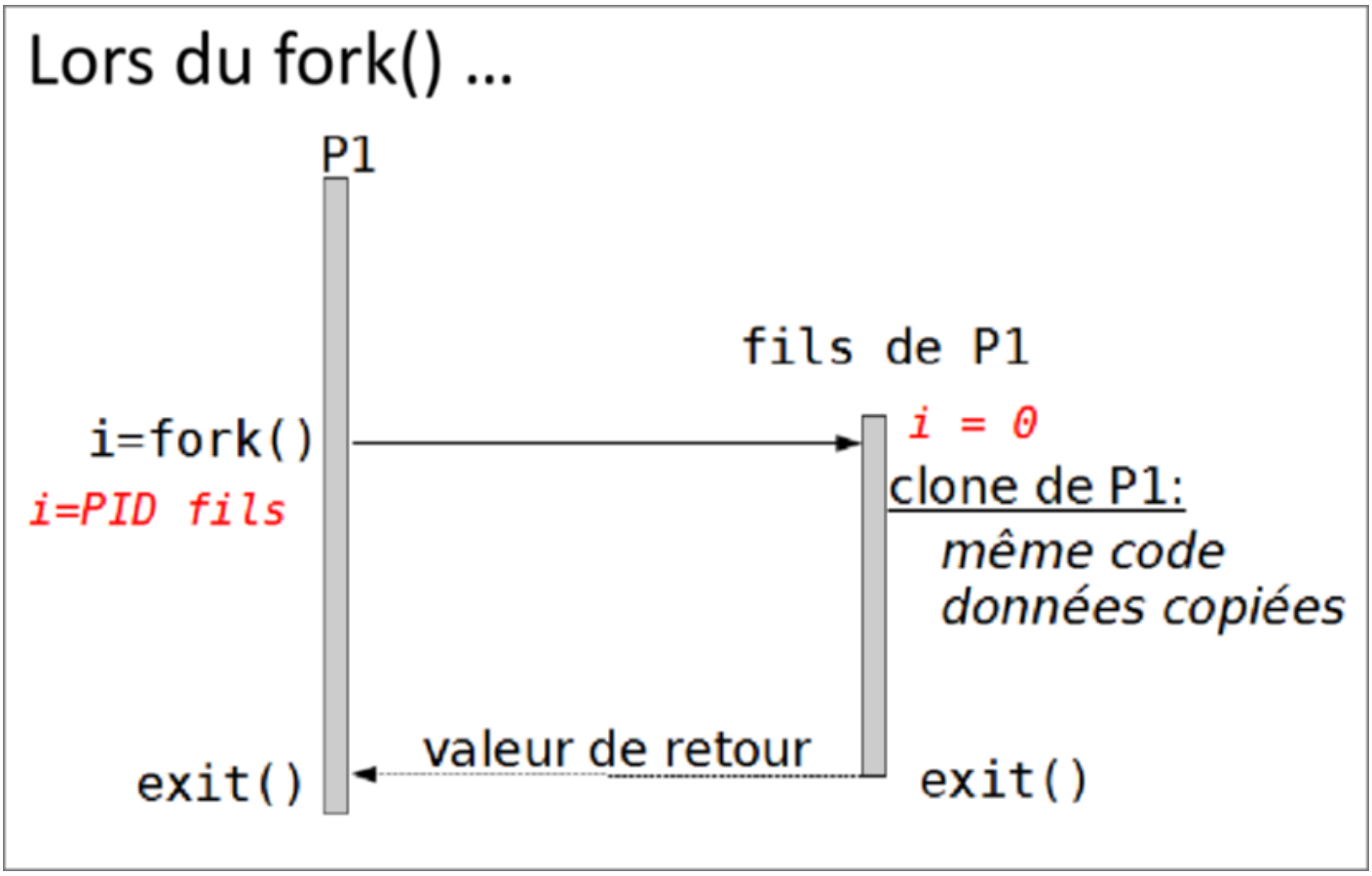
Pour changer de processus on doit pouvoir sauvegarder le contexte (les données) du processus précédent.

Le système va donc sauvegarder toutes les informations du processus pour pouvoir le redémarrer plus tard.

Ensuite le scheduler va sélectionner un autre processus et en charger les informations/contexte pour le démarrer.

Il va ainsi faire cela tout le temps pour alterner entre tous les processus en attente, prêts et en cours pour maximiser l'utilisation du CPU et donner l'illusion que tout fonctionne en même temps.

Création d'un processus (fork)



Pour créer un processus on utilise l'appel système *fork*. Le processus créé par un fork est appelé le processus *fils*, et le processus qui a créé le *fils* est appelé le *père*.

Le processus *fils* est un clone de son *père*, toutes les données du premier sont recopiées dans le fils.

La fonction `fork()` en C va retourner un entier :

- `-1` si une erreur est survenue (comme souvent en C, une valeur négative veut dire qu'une merde s'est passée)
- `0` pour le processus fils
- Le PID du fils pour le processus père

Exemples en C

Exemple simple

Voici un autre exemple :

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

```
int main (void)
{
   /* Variable pour stoquer la réponse du fork */
   pid_t pid;

   /* Fork et mise du résultat dans la variable */
   pid = fork();

   /* Si le pid est 0, alors c'est le fils qui lit l'info */
   if (pid == 0) {
     printf("Je suis le processus fils\n");

   /* Si le pid est autre chose, alors c'est le père qui lit l'info */
   } else {
     printf("Je suis le processus père et mon fils est le : %d\n", pid);
   }

   /* Fin des deux processus */
   return EXIT_SUCCESS;
}
```

Va retourner quelque chose comme :

```
Je suis le processus père et mon fils est le : 243328
Je suis le processus fils
```

Exemple plus complexe

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

int main (void) {
  /* La valeur de i par défault est 5 */
  int i=5;
  pid_t ret;

  /* Ce code sera exécuté uniquement sur le père */
  printf("Avant le fork() ... \n");
```

```
  /* La valeur de retour sera 0 sur le processus fils, et le pid du fils sur le processus père */
  ret = fork();

  /* Le code à partir d'ici sera exécuté sur les deux processus */
  printf("Après le fork() ... \n");

  /* Sur le processus fils, i sera multiplié par 5 */
  if(ret == 0) {
    i*=5;

  /* Sur le processus père, i sera additioné de 5 */
  } else {
    i+=5;
  }

  /* Le code ici sera exécuté sur les deux processus */
  printf("La valeur de i est: %d\n", i);

  /* On retourne la valeur de succès d'exécution ce qui va tuer les deux processus */
  return EXIT_SUCCESS;
}
```

Va retourner :

```
Avant le fork() ...
Après le fork() ...
La valeur de i est: 10
Après le fork() ...
La valeur de i est: 25
```

Fin d'un processus

Un processus se termine quand il n'y a plus aucune instruction à exécuter ou lorsque l'appel système `exit(int)` est appellé (cette fonction permet de renvoyer une valeur entière au processus père).

wait et waidpid

Un processus père peut attendre la mort de son fils à l'aide des fonctions `wait()` et `waitpid()` et peut ainsi récupérer l'entier retourné par le `exit(int)` du fils.

La fonction `wait()` va simplement attendre la mort d'un fils (peu importe lequel) tandis que la méthode `waitpid()` va attendre la mort d'un processus fils déterminé.

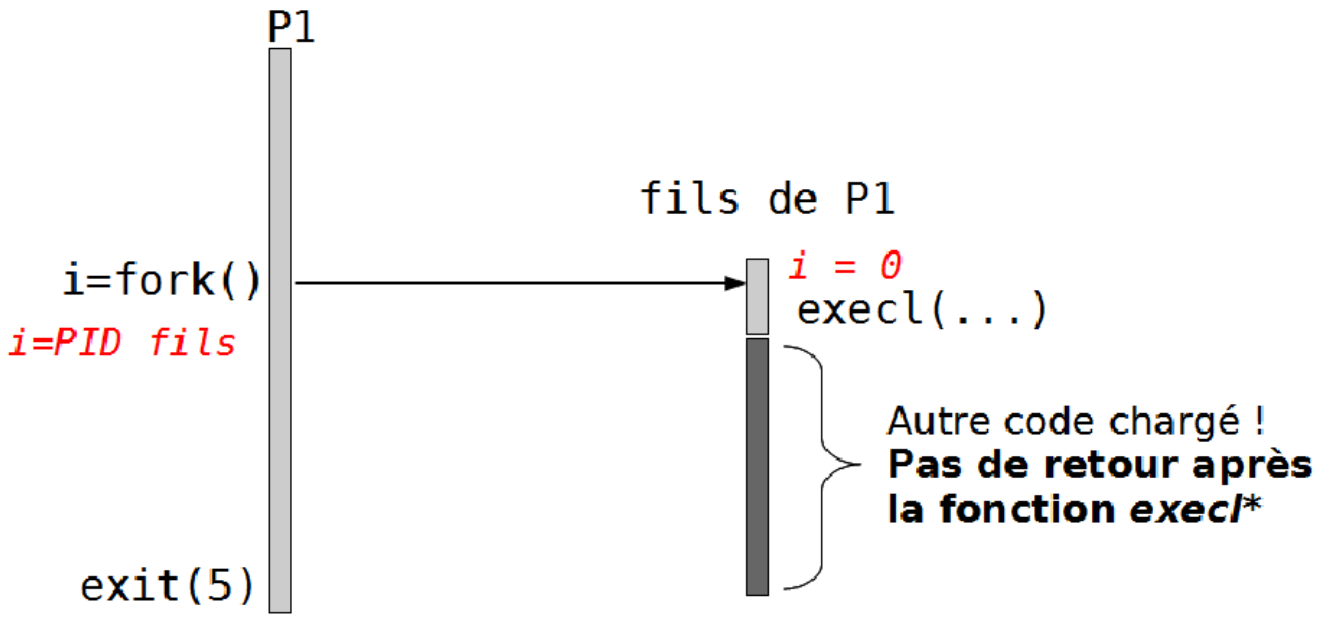
Les fonctions `wait` et `waitpid` retourne le pid du fils, il faut donc passer le pointeur d'une variable en argument pour récupérer les valeurs. Voici un exemple :

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

int main(void) {
  char chaine[100+1];
  int compteur = 0;
  pid_t pid_fils;

  /* On crée un nouveau processus */
  switch (fork()){
    /* Si le résultat est -1 c'est qu'il y a eu un problème */
    case -1:
      printf("Le processus n'a pas été créé.");
      exit(-1);

    /* Si on est le processus fils, on demande d'entrer une chaine de caractères */
    case 0:
      printf("Entrez une chaine de caractères : ");
      scanf("%100[^\n]%*c", chaine);

      /* On retourne la longueur de la dite chaine en exit */
      exit(strlen(chaine));

    /* Si on est le processus père, on attends la mort du fils et on récupère la sortie du exit dans une variable */
    default:
      /* On stoque le retour du exit dans une variable ainsi que le PID du fils */
      pid_fils = wait(&compteur);
      /* On extrait la longueur de la chaine depuis la sortie du wait avec WEXITSTATUS */
      printf("Enfant %d est mort. Compteur = %d", pid_fils, WEXITSTATUS(compteur));
```

```
  }


  return EXIT_SUCCESS;
}
```

execl

`execl` permet d'avoir de charger un autre dans le processus, une fois cette fonction execl exécuté le code du processus remplacé est perdu.



La fonction prends en paramètre, deux choses :

- Le **chemin vers le programme**
- Les **arguments du programme** ce qui commence par le chemin du programme (une deuxième fois) et qui termine par un NULL

Voici un exemple d'execl :

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


int main(void) {
  /* On crée un nouveau processus avec fork() */
  switch(fork()) {
```

```
    /* Si fork retourne -1 c'est qu'il y a eu un problème */
    case -1: printf("Erreur fork()\n");
             exit(-1);

    /* Si fork retourne 0 c'est que c'est le processus fils, on va donc exécuter la commande ls avec execl */
    case 0: printf("Je suis le fils\n");
            /* Execl va lançer la commande "ls -l" */
            /* Le premier paramètre est le chemin vers le programme */
            /* Le deuxième paramètre est le chemin vers le programme qui va être passé en argument */
            /* Le troisième paramètre est le flag "-l" qui sera passé en argument */
            /* Le NULL termine la liste des arguments */
            if(execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", "-l", NULL)) {
              /* Si le execl retourne -1, c'est qu'il y a eu une merde */
              printf("Erreur execl()\n");
              exit(-2);
            }
            printf("Ce code ne sera jamais exécuté car il est après le execl");

    /* Pour le processus père, on va simplement attendre que le fils ai terminé */
    default: wait(NULL);
             printf("Mon fils a terminé\n");
  }
  return EXIT_SUCCESS;

  /* Le switch n'a pas besoin de break car dans tous les cas, cela se fini par un exit, il ne peut donc rien y avoir
après */
}
```

Choix des processus

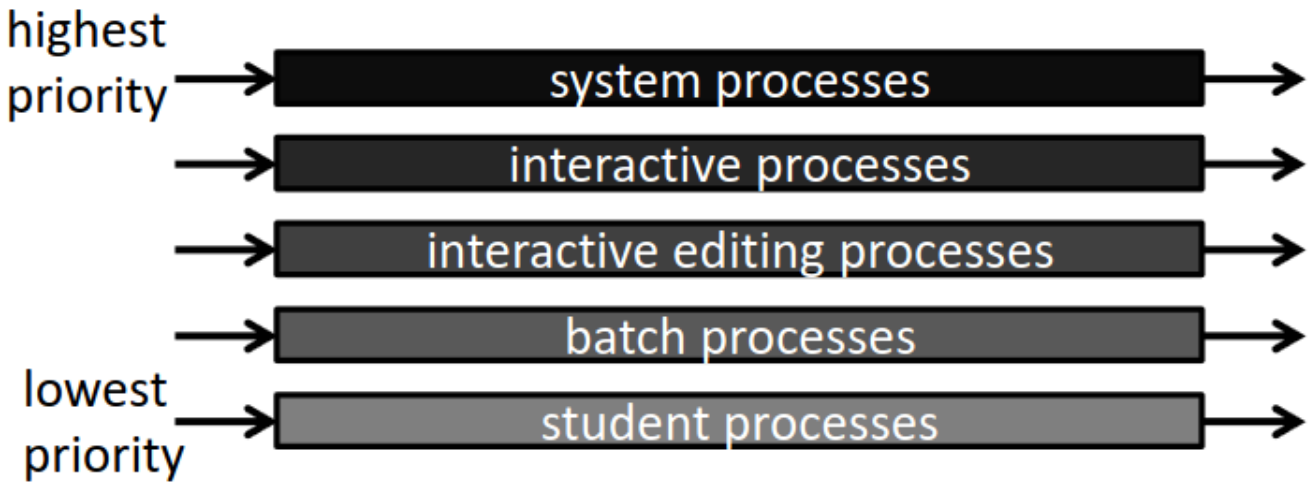
Le **scheduler** du système d'exploitation doit sélectionner les processus à démarrer pour maximiser l'utilisation du CPU (généralement entre 40% et 90%) pour avoir un débit (le nmobre de processus terminés par unité de temps) important (si les processus sont trop long, le début sera faible).

Algorithmes

- **FCFS (First-Come, First-Served)**, la file ici est une FIFO (first in, first out), c'est l'algorithme le plus simple à implémenter mais il peut être très long, car si le premier

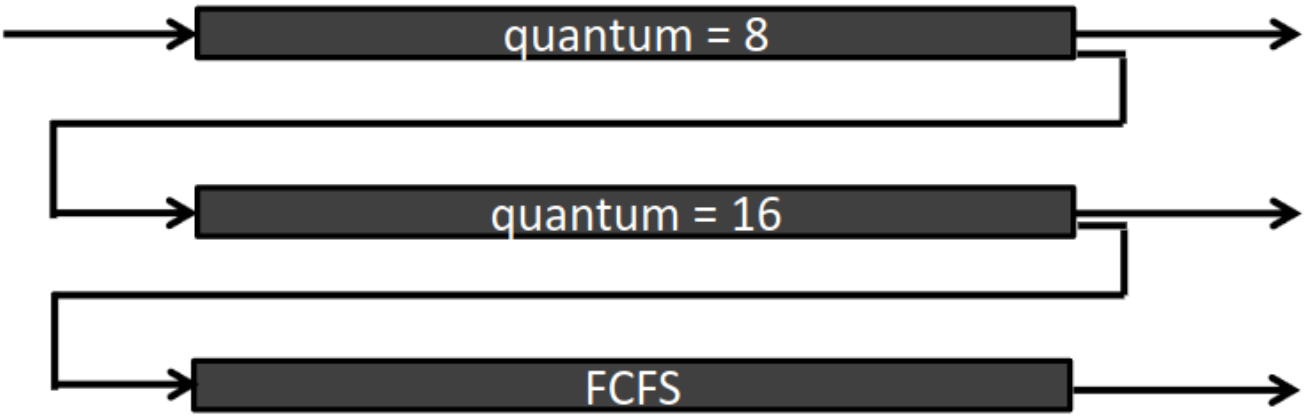
processus est long, il ralenti tous les processus qui suivent

- **SJF (Shortest-Job First Scheduling)**, est une amélioration du précédent, il ordonne les processus selon leur durée, ainsi les processus les plus rapides viennent au début et les plus lents à la fin. Cet algorithme est seulement possible si on sait à l'avance la durée du processus, mais aujourd'hui c'est rarement le cas.
- **Priorité**, on tient compte de la priorité d'un processus, ainsi les processus avec la priorité la plus élevée (nombre le plus petit) sont exécutés avant.
  - Cet algorithme peut être préemptif ce qui signifie qu'un processus qui tourne (running) peut être mis sur pause (en état ready) si un processus de plus haute priorité arrivé.
  - Cependant cela peut mener à de la famine car les si il y a continuellement des processus de plus haute priorité qui arrive.
    - Ce problème peut être résolu en combinant l'age et la priorité (ainsi les processus ayant attendu trop longtemps passe avant)
- **Round-Robin Scheduling (Tourniquet)**, les processus sont servi dans l'ordre d'arrivée et chaque processus reçoit le CPU pour un temps déterminé (appelé quantum), ainsi on va alterner entre chaque processus avec un temps donné (c'est donc un algorithme préemptif)
  - Si le quantum est trop grand, l'utilisateur·ice aura l'impression que le système lag car rien ne pourra être fait tant que le processus en cours est n'a pas fini son quantum
  - Si le quantum est trop petit, alors on va perdre en efficacité du CPU car beaucoup de l'énergie de calcul sera mise dans le fait d'échanger tous les processus tout le temps.
- **Multilevel Queue Scheduling**, qui s'agit d'avoir de files différentes suivant la nature du processus, une priorité et un mécanisme de scheduling propre est attaché à chaque file, il est ainsi possible d'avoir FCFS et Round-Robin sur des files différentes.



- **Multilevel Feedback Queue Scheduling**, les files sont plus dynamique (un processus n'appartient pas à une file et migrent d'une file à l'autre), chaque file a des caractéristiques précises (quantum, algorithme scheduling, etc).
  - Par exemple on peut dire qu'un processus va commencer dans une RR de quantum 8, si il n'a pas fini à la fin de son quantum il passe dans une autre file de priorité

mois élevée avec un quantum de 16 et si il n'a toujours pas fini il passe dans une priorité encore mois élevée en FCFS.



Choix de l'algorithme

Il n'y a pas un seul bon algorithme car chaque algorithme sert à remplir un but précis.

On peut évaluer ces algorithmes selon une certaine utilisation en utilisant des modèles mathématiques, des simulations, des implémentations et des tests.

Quel algorithme utilisé dans l'OS ?

Sous Windows, c'est un système à 32 niveaux de priorités (préemptif).

Linux en revanche utilise un autre système de scheduling appelé CFS, vous pouvez en apprendre plus dans cette vidéo.

La communication IPC

Il est nécessaire que les processus communiquent entre-eux (pour le partage d'information, la répartition des calculs, la modularité et la facilité). La communication inter-process sont très courrant sous UNIX et servent à résoudre ce problème.

Différentes options

- Fichiers, cependant c'est très lent et difficile à synchroniser
- Tube nommé ou non-nommé
- Files de messages
- Mémoire partagée, qui a l'avantage d'être très rapide
- Socket (échanges via le réseau) qui est universel

Les tubes

Les tubes sont des petits fichiers géré en file circulaire, ils sont si petit qu'ils sont souvent en cache (ce qui est donc très efficace). Si le message devient trop grand, il sera alors découpé en blocs.

Tubes non-nommés

Les tubes non-nommés sont des tubes temporaires, ils sont alloué via l'appel système `pipe()`

Il existe différents tubes standards :

- `stdin` tube de lecture (via le clavier, genre `scanf`)
- `stdout` tube de sortie (affichage à l'écran, genre `printf`)
- `stderr` est un tube de sortie pour les messages d'erreurs

Il est ainsi possible de rediriger ces tubes.

Opérations

- Ecriture dans le tube avec appel système `write(int h, char* b, int s)` (h étant le tube, s les premiers octets, et b le buffer)
- Lecture dans le tube avec appel système `read(int h, char* b, int s)`
- Fermeture du tube via `close(int h)`

Note les fonctions `read` et `write` retournent 0 si on tente d'écrire ou de lire un tube sans qu'il n'y a pas de processus à l'autre bout du tube.

Exemple

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

int main(void) {
  int tube[2];
  char buffer[255];

  /* On crée le tube et on note les identifiant entrée et sortie dans le tableau  */
  pipe(tube);

  /* On crée un nouveau processus */
  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Pour le processus fils */
    /* Le processus fils va lire le processus tube[0] pour avoir la lecture en écriture */
    /* Le buffer va être la variable où les données vont être écrites */
    /* Et enfin 's' est la taille que l'on va récupérer */
    case 0:
      /* Si le tube est vide, read va attendre que le tube soit rempli */
      read(tube[0], buffer, 254);
      printf("Message: %s\n", buffer);
      break;

    /* Pour le processus père : */
    /* Ici on écrit "salut à toi" dans le tube en écriture (tube[1]), le buffer va donc contenir le message */
    /* Le 's' va contenir la longueur du buffer */
    /* Ainsi le message va être envoyé au fils */
    default:
      strncpy(buffer, "salut a toi", 12);
      write(tube[1], buffer, strlen(buffer));
```

```
    /* Ici on attends que le processus fils meurt, sinon le read du fils retournera 0 car il n'y aura plus le processus à l'autre bout car le programme sera terminé */
    wait(NULL);
  }
  return EXIT_SUCCESS;
}
```

Redirections

Par défault les 3 tubes standard sont dirigé vers le stdout (ou stderr si configuré autremenet).

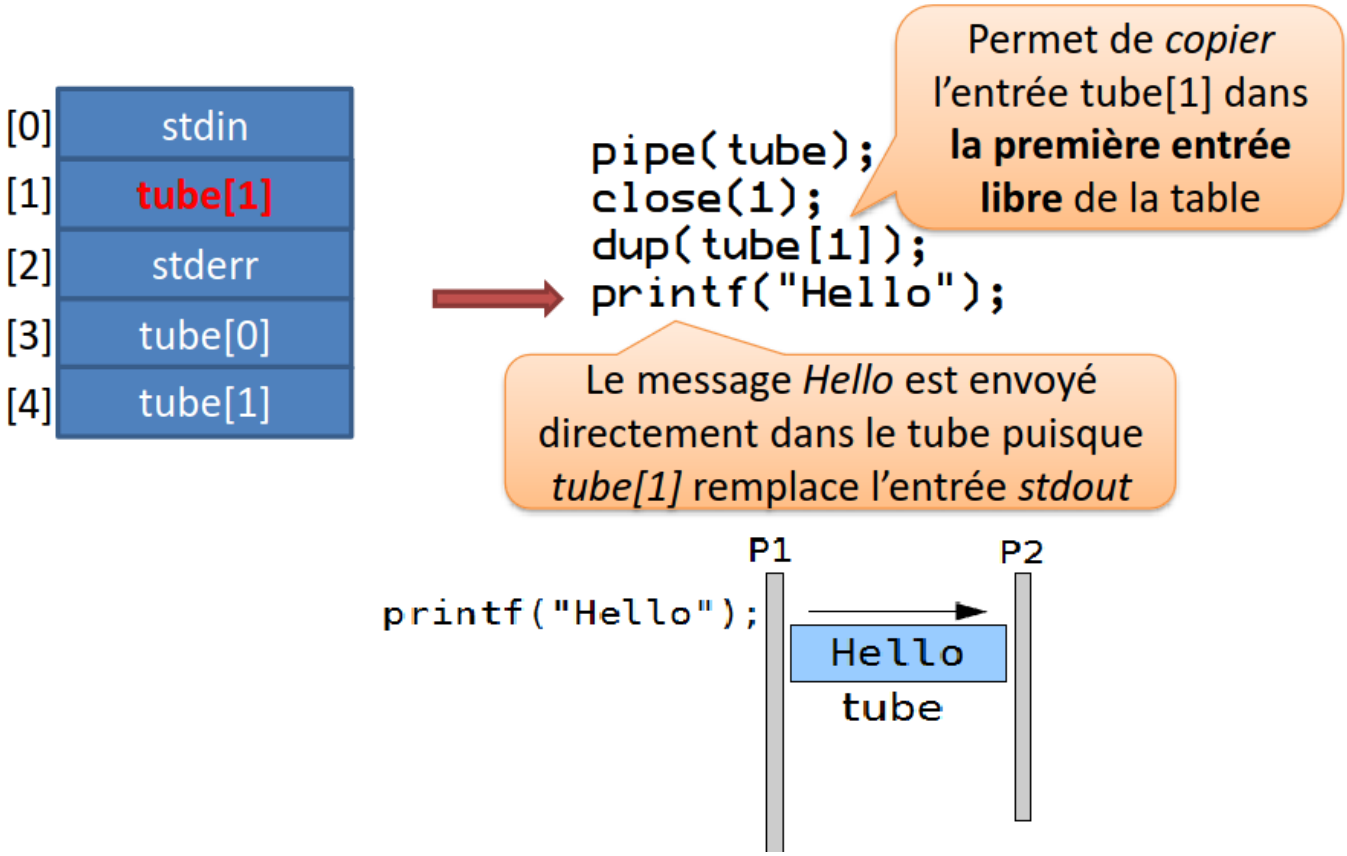
On peut également rediriger ces tubes, ainis ce qui était affiché à l'écran est alors dirigé automatiquement dans le tube ou peut être lu à partir d'un tube.

Utilisation en shell

```
# On liste les fichiers et on récupère toutes les lignes contenant "dia"
# grep prends comme entrée le résultat du ls
# C'est le shell qui va automatiquement rediriger le stdout du ls comme le stdin du grep
ls | grep "dia"
```

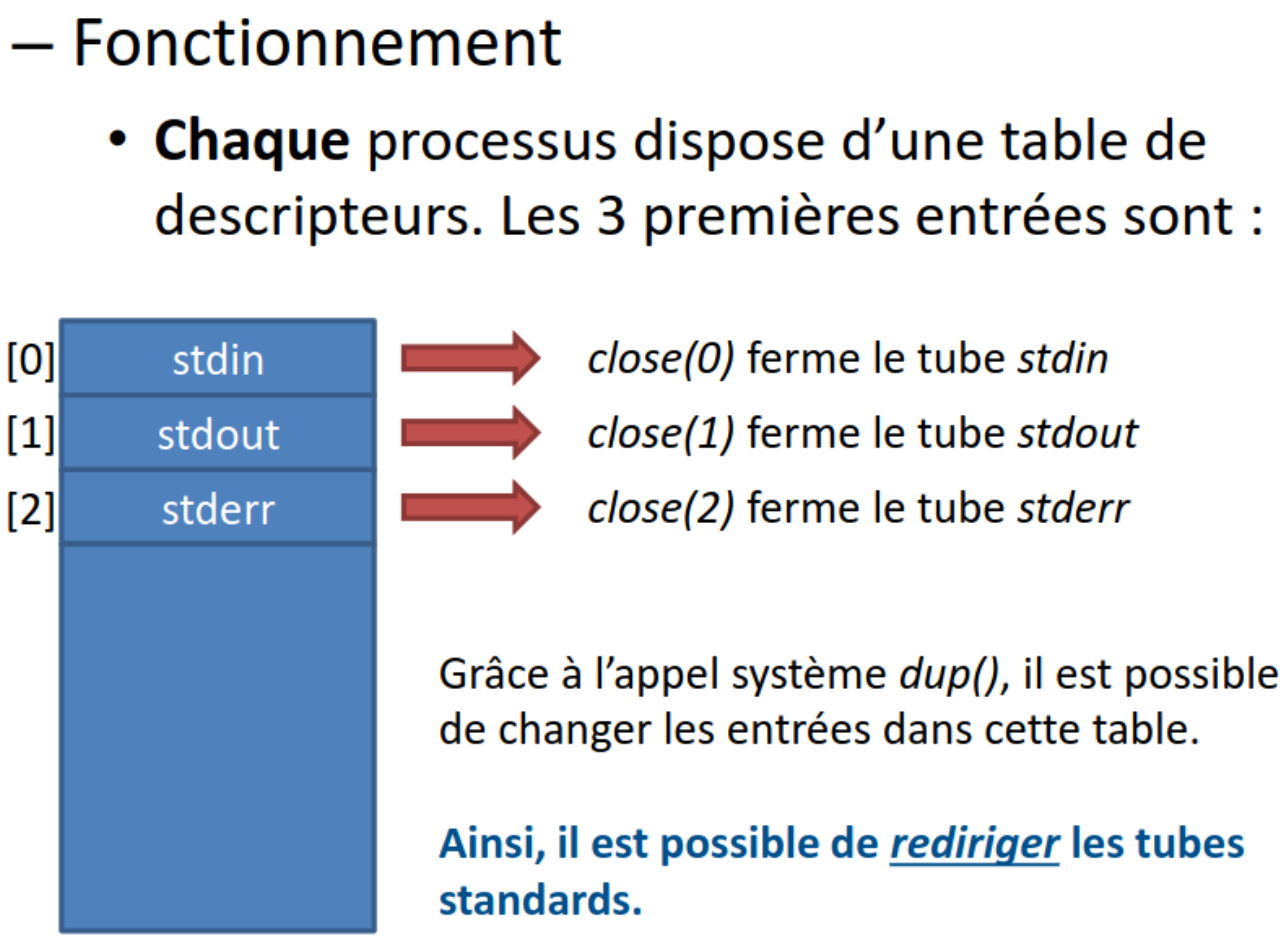
Fonctionnement

Voici un exemple de redirection :



Dans cet exemple :

1. On crée un tube
2. On ferme le stdout
3. On copie notre sortie de tube comme étant le stdout
4. On écrit dans le stdout → donc dans notre tube



Exemple en C

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

int main(void) {
  int tube[2];
  char buffer[255];

  /* On crée notre nouveau tube */
```

```
  pipe(tube);

  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Pour le processus fils */
    case 0:
      /* On ferme le stdin */
      close(0);
      /* On copie l'entrée du nouveau tube pour remplacer le stdin */
      dup(tube[0]);
      /* On lit depuis le stdin (on lit donc depuis le tube) */
      scanf("%[^\n]%*c", buffer);
      /* On affiche le message stdout */
      printf("Message: %s\n", buffer);
      break;

    /* Pour le processus père */
    default:
      /* On ferme le stdout */
      close(1);
      /* On copie la sortie du tube dans le stdout */
      dup(tube[1]);
      /* On print un message vers le stdout, qui a été redirigé vers le nouveau tube */
      printf("salut a toi\n");
      /* On force le printf a se faire maintenant */
      fflush(stdout);
      /* On attends que le processus fils meurre pour éviter de causer une erreur de lecture du tube */
      wait(NULL);
  }
  return EXIT_SUCCESS;
}
```

Autre exemple (avec execl)

Lorsque l'on redirige un pipe, le pipe reste redirigé si on exécute un autre programme par après avec `execl`, on peut donc passer l'output d'un programme dans un autre programme. Voici un exemple de pipe qui prends le résultat du ls et compte le nombre de lignes, c'est l'équivalent de `ls | wc -l`. Notez cependant que les path de ls et wc sont **très certainement** différent sur votre système, pour connaitre le PATH réel faites la commande `whereis ls` et `whereis wc`.

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

int main(void) {
  int tube[2];

  /* On crée un nouveau tube */
  pipe(tube);

  /* On crée un premier enfant */
  if (fork() == 0) {
      /* On ferme le stdout */
      close(1);
      /* On redirige la sortie du tube dans le stdout */
      dup(tube[1]);

      /* On ferme les tubes pour laisser uniquement le stdin et stdout */
      close(tube[0]);
      close(tube[1]);

      /* On exécute le ls */
      execl("/run/current-system/sw/bin/ls", "/run/current-system/sw/bin/ls", NULL);
      /* Puis ce que rien n'arrive après un execl le reste du code ne s'exécutera pas */
  }

  /* On crée un deuxième enfant */
  if (fork() == 0) {
      /* On ferme le stdin */
      close(0);
      /* On remplace le stdin par le tube[0] */
      dup(tube[0]);
      /* On ferme les tubes pour laisser uniquement les stdin et stdout */
      close(tube[0]);
      close(tube[1]);

      /* On exécute wc -l ça récupère le stdin du ls */
      execl("/run/current-system/sw/bin/wc", "/run/current-system/sw/bin/wc", "-l", NULL);
```

```
  }

  /* On ferme le tube[0] et tube[1] pour laisser uniquement le stdin et stdout */
  close(tube[0]);
  close(tube[1]);

  /* On attends la mort des fils pour mourrir aussi */
  wait(NULL);
  wait(NULL);
  return EXIT_SUCCESS;
}
```

Tubes nommés

Les tubes nommés sont permanent via des fichiers spéciaux dans le filesystem.

On peut en créer un en utilisant `mkfifo(const char* nom, mode_t mode)` (le nom préise le nom du tube et le mode précise les permissions).

Les processus non-només sont liés entre père et fils, tandis qu'ici les processus nommés peuvent être utilisé par des processus qui bien que sont complètement indépendant l'un de l'autre.

Un processus peut ouvrir un tube en utilisant `open(const char* nom, int flags)` (qui est bloquant par défaut tant que le tube n'est pas ouvert des deux cotés), les flags définissent le mode d'ouverture (écriture, lecture ou les deux bien que cela ne soit pas recommandé).

On peut écrire dans un pipe avec `write(int fd, char* buf, int size)` et lire avec `read(int fd, char* buf, int size)`

On peut enfin fermer un tube avec `close(int fd)`

Mémoire partagée

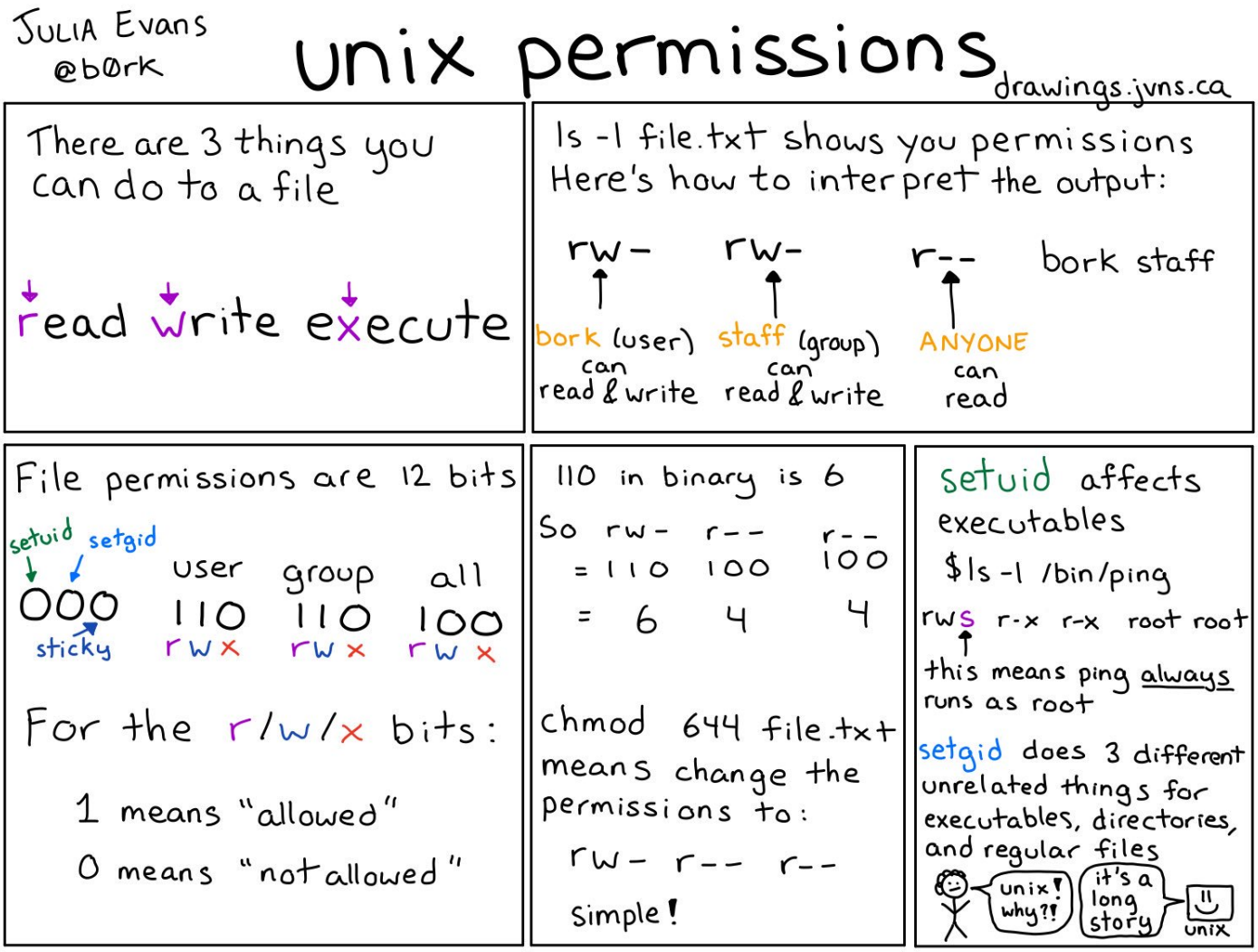
La mémoire partagée est un moyen très commun pour partager des informations entre processus, la zone de mémoire est commune à plusieurs processus. La taille est complètement configurable (comme avec malloc) et après un fork, le processus fils hérite de la mémoire partagée.

Shmget - Allocation

L'allocation se fait via `int shmget(key_t key, int s, int fl)` où

- La clé est l'identifiant de la mémoire partagée
- `s` est la taille en octets
- `fl` est le flag de permission sur la zone

Petite note sur les permissions



Les permissions se font via un code tel que `0664` :

- Le premier `0` indique que le nombre est en octal et non pas en décimal. Ainsi `0644` c'est `110 110 100` en binaire, et `777` est `1 100 001 001` en binaire.
- Premier `6` → est le propriétaire signifie que le propriétaire peut lire et écrire(read (1) write (1) execute (0) = 110 = 6)
- Deuxème `6` → est le groupe qui peut lire et écrire également (read (1) write (1) execute (0) = 110 = 6)
- Enfin le `4` → les autres utilisateurs peuvent seulement lire (read (1) write (0) execute (0) = 100 = 4)

Shmat - Récupération de pointeur

L'appel `shmat` permet de récupérer un pointeur vers la zone mémoire partagée. Sa signature de méthode est la suivante : `char* shmat(int shmid, char* addr, int flags)` où

- `char*` est le pointeur retourné
- `int shmid` est l'identifiant retourné par shmget

- `char* addr` est l'addresse souhaitée (généralement positionnée à 0 pour laisser le système choisir)
- `int flags` pour les paramètres de restriction (par exemple SHM_{RDONLY} donne un pointeur en lecture seule)

Shmdt - Détacher la zone

L'appel `shmdt` (qui prends en argument le pointeur) va détacher la zone mémoire sans pour autant la libérer.

Shmctl - Gérer la zone

L'appel `int shmctl(int shmid, int cmd, struct shmid_ds* ds)` permet de gérer la zone de mémoire.

- `shmid` est le descripteur de la zone retourné par shmget
- `cmd` détermine l'opération souhaitée (pour supprimer on utilise `IPC_RMID` mais il existe également IPC_{STAT} pour avoir des informations, IPC_{SET} pour modifier les valeurs associées, etc)
- `ds` contient les données en rapport avec les commandes STAT et SET

Exemple

Disons que l'on veut faire 2 programme, 1 premier écrit dans la zone mémoire et le deuxième la lit :

- Premier programme :

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
  int shmid;
  char* ptr;

  /* On alloue une zone de mémoire partagée avec l'identifiant 2324, une taille de 1024 octets, et une permission totale pour tout le monde */
  shmid = shmget(SHM_KEY, K, 0777|IPC_CREAT);
```

```
  /* Récupère un pointeur vers la zone de mémoire partagée */
  ptr = shmat(shmid,NULL,0);

  /* On copie une chaine de caractère dans la mémoire partagée */
  strcpy(ptr, "Hello !\n");

  /* On détache la zone mémoire (ce qui ne la libère pas mais permet qu'un autre processus l'utilise) */
  shmdt(ptr);

  /* On ferme le programme */
  return EXIT_SUCCESS;
}
```

- Deuxième programme :

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <string.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <stdio.h>

#define SHM_KEY 2324
#define K 1024

int main(void) {
  int shmid;
  char *ptr;

  /* On récupère la zone mémoire avec l'identifiant, la taille et le flag */
  shmid = shmget(SHM_KEY, K, 0777);

  /* Si le shmid retourné est < 0 alors c'est que la zone n'a pas été trouvée */
  if (shmid < 0) {
    printf("Erreur SHM\n");
    exit(-1);
  }
```

```
  /* On récupère le pointeur de la mémoire partagée */
  ptr = shmat(shmid, NULL, 0);

  /* On print le contenu de la mémoire partagée */
  printf("sa %d", IPC_CREAT);
  printf("Contenu : %s\n", ptr);

  /* On détache la mémoire du programme */
  shmdt(ptr);

  /* Le shmctl IPC_RMID va détruire la zone mémoire */
  shmctl(shmid, IPC_RMID, NULL);

  return EXIT_SUCCESS;
}
```

Commande ipcs pour lister les mémoires partagées

Si vous souhaitez voir la liste des zones partagées on peut utiliser la commande `ipcs`.

```
[snowcode@snowcode:~]$ gcc mempar.c

[snowcode@snowcode:~]$ ./a.out

[snowcode@snowcode:~]$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000914 4          snowcode   777        1024       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

Synchronisation

Lorsque plusieurs processus coopèrent, ils doivent souvent intéragir entre eux, ils doivent parfois attendre qu'une opération soit effectuée par un autre processus pour travailler.

Il faut donc avoir des mécanismes qui permettent d'envoyer des événements aux processus (un processus doit pouvoir attendre l'évènement).

Types de synchronisation

Sous UNIX, les mécanismes suivants sont mis en oeuvre pour la synchronisation :

- Les signaux
- Les sémaphores

On parlera de **point de synchronisation** lorsqu'un processus attend un autre.

Les signaux

Un signal est un événement capturé par un processus, c'est aussi un mécanisme simple utilisé par le système d'exploitation pour signaler aux processus une erreur (SIGILL, SIGFPE, SIGUSR1, SIGUSR2, etc).

Exemple

Voici par exemple un programme dont la fonction `sighandler` est appellée lorsque le signal SIGUSR1 est déclenché :

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

void sighandler(int signum);

/*
  Ce programme va lier la fonction sighandler au signal SIGUSR1
  Ce qui signifie que lorsque l'on lance le programme (qui contient une boucle infinie), lorsque l'on lance le signal
```

```
via "pkill -SIGUSR1 a.out" (par exemple)
  La fonction sighandler va être appellée et "SIGUSR1 reçu" va donc s'afficher à l'écran.
*/
int main(void) {
  /* Si on remplace ici SIGUSR1 par SIGINT et que l'on fait CTRL+C, on va appeller la commande sighandler */
  if(signal(SIGUSR1, sighandler) == SIG_ERR) {
    printf("Erreur sur la gestion du signal\n");
    exit(-1);
  }

  while(1) {
    sleep(1);
    printf("Hello\n");
  }

  return EXIT_SUCCESS;
}

void sighandler(int signum) {
  printf("SIGUSR1 reçu\n");
}
```

Opérations

Il existe plusieurs opérations différentes sur les signaux :

- `signal` et `sigset` qui lient un signal à une fonction. `signal` la lie une seule fois, tandis que `sigset` la lie continuellement.
- `alarm` déclenche le signal SIGALARM au processus courrant.
- `pause` suspend le processus jusqu'a la réception d'un signal
- `kill` envoie un certain signal au processus dont le PID est donné.

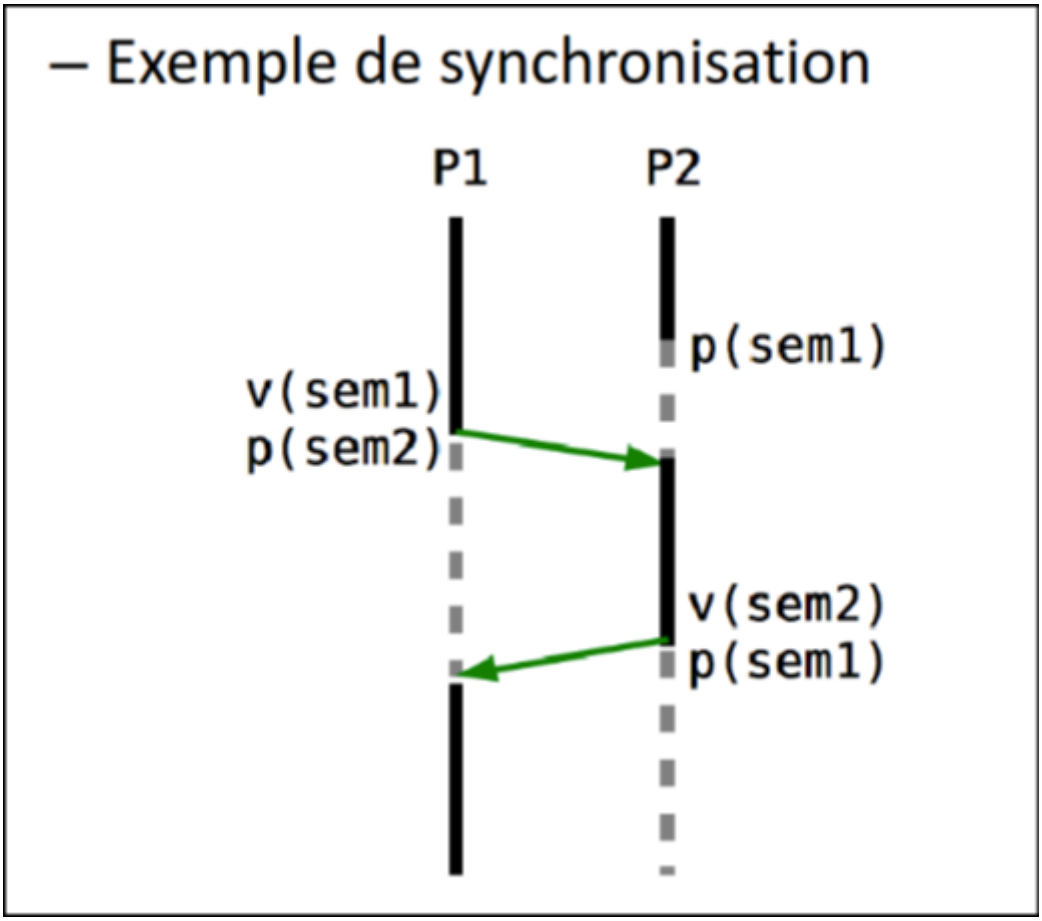
Les sémaphores

Un sémaphore est une variable entière en mémoire qui *excepté pour son initialisation* est accédée uniquement au moyen de fonction atomiques (ne pouvant pas être décomposée) `p()` et `v()`.

La fonction `p(sem)` va vérifier que la valeur est plus grande que zero, si c'est le cas, la variable est décrementée et l'exécution continue, si ce n'est pas le cas, alors il attend que ce soit le cas.

La fonction `v(sem)` va simplement incrémenter la variable de 1, et va ainsi réveiller tous les processus qui attendrait ce sémaphore.

Ces foncitons ne sont pas présente dans C de base il faut importer les fichiers `semadd.h` et `semadd.c` depuis l'espace de cours.



Exemple

Voici un exemple d'un programme qui communique avec un processus fils via 2 sémaphores. Il est intéressant de noter que généralement un processus ne va faire qu'une seule opération par sémaphore (par exemple que des `p()` sur sem1 et que des `v()` sur sem2 ou inversément)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "semadd.h"

#define SEM1 12345
#define SEM2 23456

/*
  Ce programme va créer 2 sémaphores et 2 processus (un père et un fils).
  Le fils et le père vont tous les deux exécuter une boucle sauf qu'a chaque itération ils vont s'attendre l'un l'autre.
```

Ainsi le père attends le sémaphore du fils (sem2) qui est émit lorsque le fils a fini son itération
De même le fils va ensuite attendre le sémaphore du père (sem1) qui est émit lorsque le père a fini son itération

Si on exécute ipcs -s lors de l'exécution du programme, on peut voir la liste des sémaphores créés.

Contrairement aux signaux, on peut créer nos propres sémaphores tandis que les signaux eux sont défini par le système d'exploitation.
*/

```
int main(void) {
  int sem1, sem2, i;

  /* Création des sémaphores */
  sem1=sem_transf(SEM1);
  sem2=sem_transf(SEM2);

  /* Création des deux processus */
  switch(fork()) {
    case -1:
      printf("Erreur fork()\n");
      exit(-1);

    /* Boucle du fils */
    case 0:
      printf("Je suis le fils %d\n", getpid());
      for(i=0;i<5;++i) {
        printf("[FILS] Valeur de i : %d\n",i);
        sleep(5);
        v(sem2); /* Envois du sémaphore (2) au père */
        p(sem1); /* Attente du sémaphore (1) du père */
      }

    /* Boucle du père */
    default:
      for(i=0;i<5;++i) {
        p(sem2); /* Attente du sémaphore (2) du fils */
        printf("[PERE] Je suis le père\n");
        sleep(5);
        v(sem1); /* Envois du sémaphore (1) au fils */
      }
```

```
  }

  return EXIT_SUCCESS;
}
```

Semget - Allocation de sémaphores

L'allocation se fait via `int semget(int key, int nb, int flag)`, où

- La valeur retournée est un descripteur "semid"
- La clé est la valuer qui identifie le sémaphore
- Les flags définissent les permissions, comme pour les mémoires partagées `IPC_CREAT` permet de demander la création des sémaphores

On peut aussi simplifier l'allocation à partir d'une clé en utilisant `int sem_transf(int key)`, cette fonction n'est **pas officielle** mais le fichier est disponible sur HELMo Learn.

Semctl - Gestion de sémaphores

On peut gérer les sémaphores (nottament pour libérer la mémoire) en utilisant `int semctl(int semid, int semnum, int cmd, union semun attr)` où

- semid est le descripteur du sémaphore
- semnum identifie le sémaphore (généralement c'est 0 si il n'y en a qu'un)
- cmd identifie la commande (IPC_{SET}, GETVAL, SETVAL, IPC_{RMID} ou IPC_{STAT}).
- `union semun attr` est une "union" (un type de structure où chaqun des éléments partagent la même zone mémoire, ainsi ce ne peut être qu'un seul élément à la fois, un peu comme une enum en Rust). Il faut généralement définir cette structure soi-même en revanche.

Semop - Faire les opérations sur les sémaphores

`int semop(int semid, struct sembuf* sops, unsigned nsops)` est la fonction qui est derrière les fonctions `p()` et `v()`.

- semid est le descripteur du sémaphore
- sops est un tableau de structures semfus (contenant l'opération)
- nsops est le nombre d'éléménts du tableau sops

Sections critiques

C'est bien beau la synchronisation sauf que la coopération entre plusieurs processus pose également des problème si deux processus concurrents souhaite modifier les même données au même moment.

Définition section critique

On peut donc mettre en place une **section critique**, c'est un ensemble d'instructions qui devraient être exécutées du début à la fin sans interruption.

Une section critique est indispensable lorsque l'on traite des données partagée afin qu'elle soit protégée et que ces données partagées ne deviennent pas incohérente.

Par exemple, si on fait par exemple une liste chainée, elle risque de ne plus être cohérente après plusieurs modifications.

On ne peut cependant pas empêcher la concurrence entre les processus. Pour cela on va mettre en place des protections avant toute modification pour s'assurer qu'un autre processus n'est pas déjà en train de modifier la zone partagée.

Variable partagée

Celle ci consiste à partager une variable entre plusieurs processus, qui est initiallement définie à 0. Avant d'entrer dans le processus, on boucle sur la valeur de cette variable.

Si la variable est différente de 0 on boucle (et on attends). Ensuite on place la variable à 1 avant de commencer la section critique puis on la remet à 0 une fois que cela est fini.

```
while (i != 0);
i = 1;
/* Section critique ici */
i = 0;
```

Problème

Un gros problème peut survenir si un processus reviens dans l'état ready (par exemple avec la fin de son quantum de temps) entre l'instruction while et l'instruction de `i = 1`.

Ainsi l'autre processus peut lui aussi entrer en section critique et peut lui aussi avoir son quantum de temps qui expire durant celui ci.

Ainsi on peut donc arriver dans une situation ou plusieurs processus sont dans une section critique en même temps (ce qui est justement la chose à éviter).

Ainsi, cette méthode de protection n'est pas fiable.

En plus de cela, utiliser une boucle while comme ceci consomme inutilement du CPU.

> Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020 à 2:25:50.

Par alternance

La protection par alternance consiste de manière similaire à la méthode précédente à avoir une variable partagée mais ou chaque processus attends une valeur différente.

Ainsi, par exemple un programme 1 pourrait avoir le code suivant :

```
while (tour != 0);
/* Section critique ici */
tour = 1;
```

Et un programme 2 pourrait avoir le code suivant :

```
while (tour != 1);
/* Section critique ici */
tour = 0;
```

Ainsi lorsque tour est à 0, le programme 1 peut exécuter sa section critique, une fois qu'elle a fini le programme 2 peut exécuter la sienne, et une fois que le progrmame 2 à fini, le programme 1 peut recommencer.

Problèmes

Cette méthode de protection est fiable, contrairement à la précédente. Cependant elle souffre tout de même d'assez gros problèmes...

Premièrement, elle est assez difficile à gérer, surtout si il y a plus de deux processus à synchroniser.

Et deuximèment, comme la précédente, elle est assez peu efficace car utiliser une boucle while ainsi consomme inutilement du CPU.

> Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020 à 2:33:20.

Par fichier

La protection par fichier consiste à ouvrir et créer un fichier (appellé "lock file") en mode exclusif (c'est à dire qu'un seul processus peut accéder au fichier à la fois) pour annoncer que la section critique commence.

Puis enfin à supprimer le fichier une fois que la section critique est terminée.

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define FIN_SECTION_CRITIQUE 1
#define DEBUT_SECTION_CRITIQUE -1

int quid(int op, char* nom, int essais) {
  int i;

  /*
   * Quand on débute la section critique, on crée un fichier dit "lock file" en mode exclusif,
   * si cela n'est pas possible c'est qu'une section critique est déjà en cours
   */
  if(op == DEBUT_SECTION_CRITIQUE) {
    for(i=0;i<essais;++i) {
      /* Tenter d'écrire un fichier en mode exclusif (un seul processus a accès au fichier à la fois) et renvoyer 0 en cas de succès */
      if(open(nom,O_WRONLY|O_CREAT|O_EXCL) >=0) {
        return 0;
      }

      /* Si cela n'a pas fonctionné, réessayer dans une seconde */
      else if(i<essais) {
        sleep(1);
      }
    }
  }
```

```
  /*
   * A la fin d'une section critique on supprime le lock file
   */
  if(op == FIN_SECTION_CRITIQUE) {
    /* Suppression du fichier et retourne 0 en cas de succès */
    if(unlink(nom) == 0) {
      return 0;
    }
  }

  /* Retourne -1 en cas d'erreur ou dans le cas où tous les essais ont échoués */
  return -1;
}



int main(void) {
  printf("Attente section critique\n");
  quid(DEBUT_SECTION_CRITIQUE, "program.lock", 5);

  /* Section critique ici */
  printf("Début section critique\n");
  sleep(5);

  printf("Fin section critique\n");
  quid(FIN_SECTION_CRITIQUE, "program.lock", 5);

  return EXIT_SUCCESS;
}
```

Problèmes

Cette solution est tout à fait fonctionnelle et fiable, cependant le fait de devoir gérer un fichier peut rendre les choses un peu compliquée, de plus cela ralenti les choses. Car pour chaque accès au fichier, le processus devra passer en état **waiting**, puis **ready**, puis de nouveau **running**.

> Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020 à 2:37:50.

Synchronisation hardware

La synchronisation hardware consiste à utiliser des instructions assembleurs pour protéger une section critique.

Voici un pseudo-code de démonstration :

```
boolean TestAndSet (boolean target) {
    /* On copie la valeur de target */
    boolean rv = target;

    /* On met target à true */
    target = true;

    /* On retourne la copie de la valeur initiale */
    return rv;
}
```

Ainsi pour l'utiliser il suffirait de faire ceci :

```
/* On attends que le lock (variable partagée initialement à false) soit mis à false pour continuer */
while (TestAndSet(lock));

/* Section critique ici */

/* On met le lock à false une fois terminé */
lock = false;
```

Ainsi lorsque lock est à false, TestAndSet va la mettre à true et retourner false ce qui va donc faire passer la boucle et entrer en section critique. Une fois cette dernière terminée, le lock retourne à false.

En revanche si lock est à true, TestAndSet va retourner true et par conséquent rester dans le while, en attente jusqu'a ce que la variable soit à false.

Problèmes

Cette méthode est fiable mais le problème avec celle ci c'est l'utilisation du `while` qui va une fois de plus consomer du CPU pour simplement attendre.

Il est toute fois bon de noter que cette méthode est utilisée par le système d'exploitation pour gérer d'autes systèmes de protection tel que les sémaphores.

> Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020 à 2:47:00.

Sémaphore

Les sémaphores permettent de très simplement protéger une section critique, voici un exemple :

```
#include "semadd.h"
#include "sys/sem.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define KEY_SEM1 12345
#define KEY_SEM2 12346

int main(void) {

  int sem1, sem2;

  /* On crée 2 sémaphores */
  sem1 = sem_transf(KEY_SEM1);
  sem2 = sem_transf(KEY_SEM2);

  /* On crée un nouveau processus */
  switch (fork()) {
    case -1:
      printf("Quelque chose s'est mal passé lors de la création du processus...\n");
      return EXIT_FAILURE;

    /* Pour le fils */
    case 0:
      /* Attente du père */
      printf("En attente du père\n");
      p(sem1);

      /* Section critique */
      printf("Section critique du fils commence\n");
```

```
      sleep(3);

      /* Annonce au père qu'il a fini */
      printf("Section critique du fils se termine\n");
      v(sem2);

      break;



    /* Pour le père */
    default:
      /* Section critique */
      printf("Début de la section critique du père\n");
      sleep(3);

      /* Annonce au fils qu'il a fini */
      printf("Fin de la section critique du père\n");
      v(sem1);

      /* Attends le fils avant de supprimer les sémaphores */
      p(sem2);
      semctl(sem1, IPC_RMID, 0);
      semctl(sem2, IPC_RMID, 0);
  }

  return EXIT_SUCCESS;
}
```

Comme vu précédemment, les p et v des sémaphores sont des actions unitaires, il n'y a donc pas de risque que le processus soit arreter au millieu. L'utilisation des sémaphores est la manière recommandée de gérer des sections critiques.

> Pour plus d'infomration voir la vidéo de la séance 3 du cours d'OS 2020 à 2:52:00.

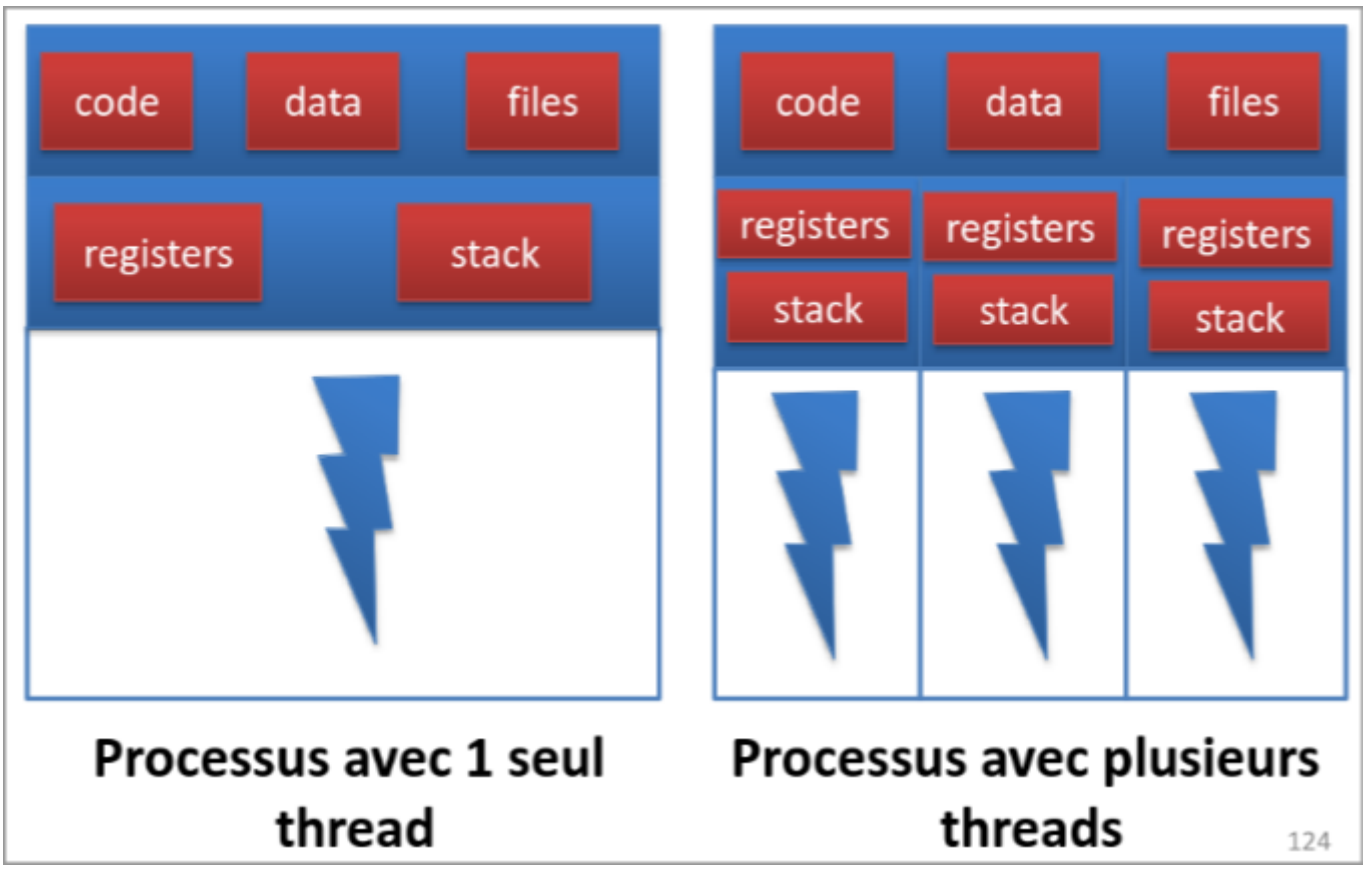
Les threads

Les processus que l'on a vu n'avait qu'un seul fil d'exécution (monothread) mais il est possible d'avoir un processus avec plusieurs fils d'exécutions (multithread).

Les threads sont en somme des sortes de "mini processus".

Avantages

Contrairement aux processus il est beaucoup plus rapide d'en créer un nouveau, également les threads d'un même processus partagent les informations. En plus sur un système avec plusieurs coeurs l'exécution des threads d'un même processus peut se faire en parallèle ce qui offre une performance intéressante.



Exemple

Par exemple on pourrait avoir un thread utilisé pour une saisie de texte, un autre thread pour l'affichage et encore un dernier thread pour vérifier les informations reçues.

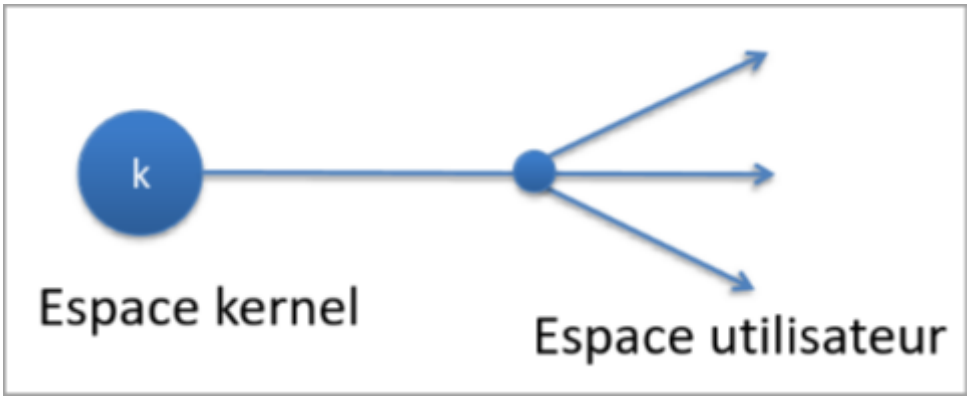
Modèles d'implémentations

Les threads peuvent être implémentés à deux niveaux :

- Dans **l'espace kernel**, il est alors pris en charge nativement par le système d'exploitation au même titre que les processus
- Dans **l'espace utilisateur**, il est alors supporté au travers de libraries externe

Les threads peuvent être implémentés selon plusiuers modèles :

Plusieurs à un

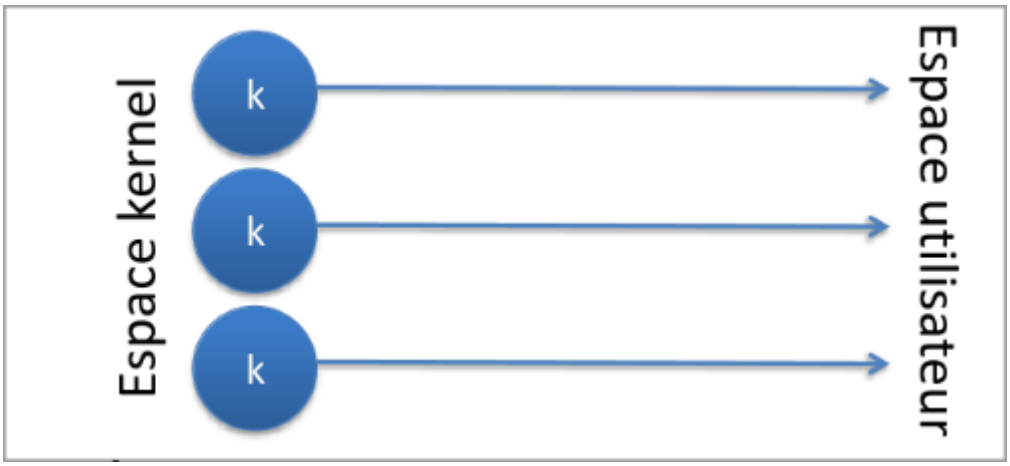


Dans ce modèle les threads sont supporté par une librarie externe, le système d'exploitation n'en a donc aucune connaissance et ne vois que le processus.

L'avantage est que sa création est rapide, cependant les inconvénients sont que un seul thread (le processus) est vu par le système, le scheduler du système n'est donc pas adapté. Si un thread réalse une opération bloquante, cela risque d'empécher tous les autres threads de travailler.

Enfin cette implémentation n'est plus vraiment courrante car elle n'est pas adaptées aux CPU multi-coeurs.

Un à un



Dans ce modèle chaque thread utilisateur est attaché à un thread kernel. Ainsi les threads sont complètement géré au niveau du système d'exploitation.

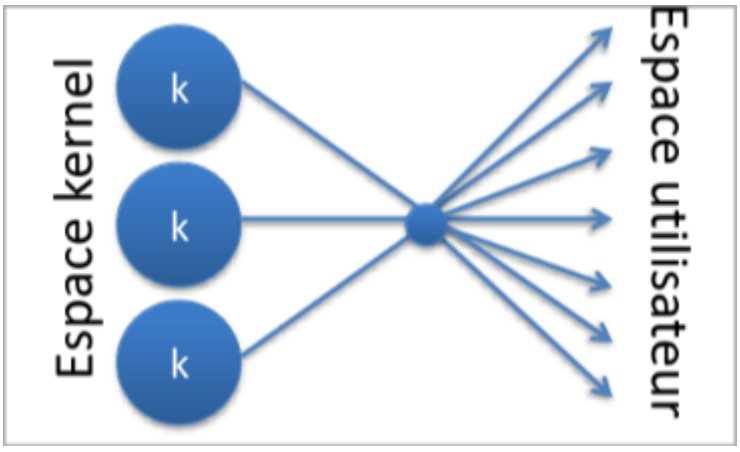
Cela a l'avantage de créer un scheduling plus avantageux et d'être compatible avec les processeurs multi-coeurs.

Cependant ce modèle est couteux pour le système car c'est lui qui doit tout gérer.

C'est ce modèle qui est nottament utilisé dans Linux. Voici par exemple la liste des threads associés au processus Firefox sur mon système.

```
[snowcode@snowcode:~]$ ps -T -p 608449
    PID    SPID TTY          TIME CMD
 608449  608449 tty1     00:00:00 Web Content
 608449  608453 tty1     00:00:00 IPC I/O Child
 608449  608455 tty1     00:00:00 Socket Thread
 608449  608456 tty1     00:00:00 HTML5 Parser
 608449  608457 tty1     00:00:00 JS Watchdog
 608449  608458 tty1     00:00:00 Backgro~Pool #1
 608449  608459 tty1     00:00:00 Timer
 608449  608463 tty1     00:00:00 RemVidChild
 608449  608464 tty1     00:00:00 ImageIO
 608449  608465 tty1     00:00:00 ImageBridgeChld
 608449  608466 tty1     00:00:00 RemoteLzyStream
 608449  608467 tty1     00:00:00 ProcessHangMon
 608449  608468 tty1     00:00:00 ProfilerChild
```

Plusieurs à plusieurs



L'idée du plusieurs à plusieurs est de créer un *pool* de thread au quel les threads utilisateurs vont être assigné à la volée au cours de l'exécution.

De cette façon cela combine les avantages des deux modèles précédents. Cependant ce modèle est assez peu courrant car il nécessite que le sysètme d'exploitation soit construit autour de ce modèle car il est plus complexe à gérer que les autres.

Problèmes

Il y a quelques difficultés à considérer pour les threads. Par exemple :

- Que se passe-t-il en cas de `fork()` dans un thread ? Certains OS vont dupliquer tous les threads, d'autres ne vont pas le faire.
- Et avec `execl()` ? L'appel execl remplace le code du processus pour charger celui d'un autre. Ainsi le code remplace tous les threads du processus
- Pour terminer l'exécution d'un thread il y a deux possibilités, dans tous les cas il faut faire très attention pour la libération des ressources
  - Le faire de manière **asynchrone**, un thread demande la terminaison d'un autre (cela est cependant rare)
  - Le faire de manière **différée**, chaque thread vérifie régulièrement s'il doit continuer ou s'arrêter
- Quand un signal est envoyé à un processus, quels threads recoivent le signal ? Tous, certains ou un en particulier ? Cela dépends du type de signal et cela est encore une fois pas comment dans tous les OS.

Librarie

Pour créer des threads dans les systèmes UNIX il existe la librarie standard `pthread` dont voici quelques fonctions intéressantes :

- `pthread_attr_init` qui permet de fixer certains attributs, mais pas utile dans le cours
- `pthread_create` pour créer et démarrer un nouveau thread
- `pthread_join` pour attendre la mort d'un thread
- `pthread_exit` pour terminer l'exécution d'un thread, cette fonction permet aussi de retourner une valeur de retour à pthread join via un pointeur générique `void*`

Exemple

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

void* thread1(void* args) {
  int i;
```

```
  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 10; i++) {
    printf("[THREAD 1] %d\n", i);
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le pointeur */
    *resultat += i;
  }

  return resultat;
}

void* thread2(void* args) {
  int i;

  /* On trasforme le void* args en pointeur de int avec un cast */
  int* resultat;
  resultat = (int*)args;

  for (i = 0; i < 24; i++) {
    printf("[THREAD 2] %d\n", i);
    /* On ajoute le nombre courrant au résultat, attention à ne pas oublier de déréferencer le pointeur */
    *resultat += i;
  }

  return resultat;
}

int main(void) {
  pthread_t tid1, tid2;

  /* On initialise les résultats à 0 */
  int resultat1 = 0;
  int resultat2 = 0;

  /* Création des threads auquels on passe les pointeurs vers les variables resultat1 et resultat2 */
  pthread_create(&tid1, NULL, *thread1, &resultat1);
  pthread_create(&tid2, NULL, *thread2, &resultat2);
```

```
  /* On attends que tous les tests se finissent */
  /* Nous n'avons pas besoin ici de récupérer la valeur de retour car on a toujours accès aux variables dont on a passé les pointeurs plus tôt, surtout que cela rends les choses très compliquées de manipuler des void** (pointeur de pointeur de valeur de type inconnue) */
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  /* Nous pouvons ensuite simplement récupérer les valeurs des résultats */
  printf("Résultat du thread 1 = %d\n", resultat1);
  printf("Résultat du thread 2 = %d\n", resultat2);

  return EXIT_SUCCESS;
}
```

Les interblocages

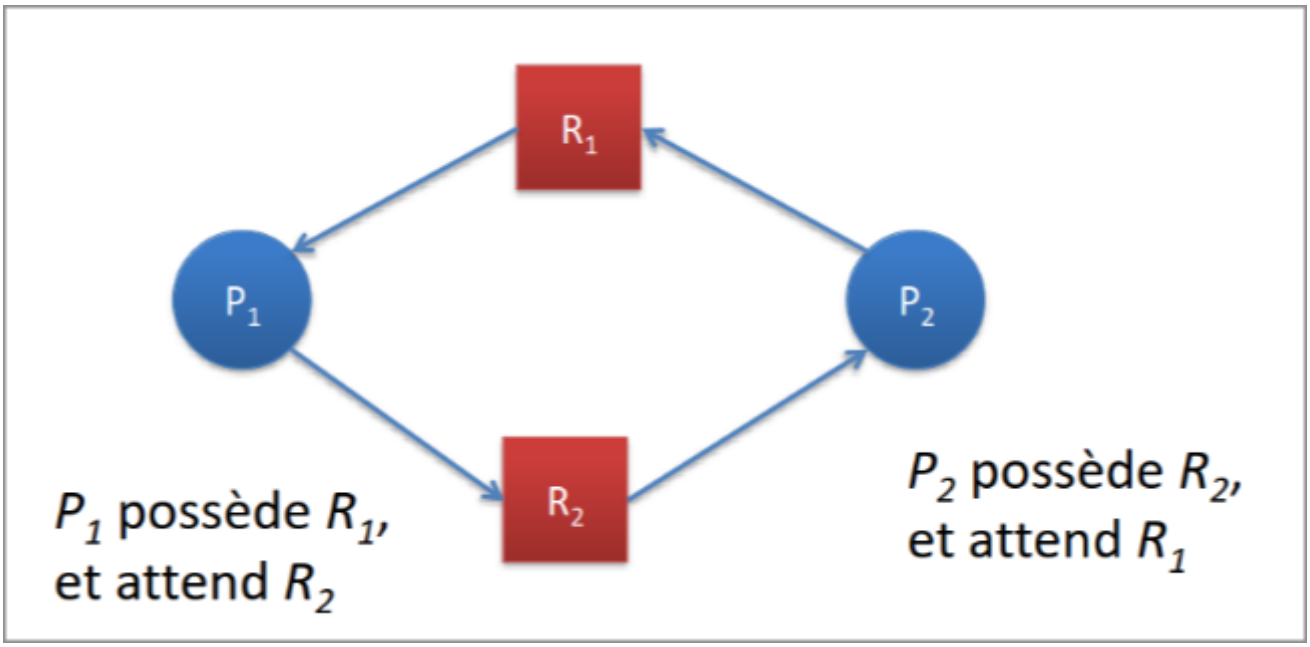
Les ressources (la mémoire, CPU, périphériques, etc) sont limitées, il faut donc gérer les ressources de manière efficace pour permettre au plus grand nombre de processus de s'exécuter.

Un interblocage peut survenir si un processus détient une ressource A qui est demandée par un autre processus détenant une ressource B qui est elle-même demandée par le premier processus.

Conditions d'un interblocage

Un interblocage survient lorsque ces 4 conditions sont réunies simultanément :

- L'exclusion mutuelle, c'est lorsque les processus utilisent des ressources qui ne peuvent pas être partagées.
- La détention et l'attente, les processus doivent à la fois détenir une ressource et attendre une autre ressource.
- L'impossibilité de réquisitionner une ressource, car c'est dégueulasse et que l'on ne peut pas savoir l'état de la ressource.
- L'attente circulaire, voir plus bas



Empécher un interblocage

Pour empécher un interblocage il faut empécher l'une des conditions d'arriver.

- L'exclusion mutuelle ? on ne peux pas empécher un processus de détenir des ressources non partageable
- La détetention et l'attente ? Il y a deux solutions pour faire en sorte que la détention et l'attente n'arrive pas en même temps :
  - Un processus pourrait demander toutes les ressources dont il pourrait avoir besoin dès le départ de son exécution
  - Lorsqu'un processus demande une nouvelle ressource, il doit libérer toutes les autres puis récupérer toutes ces ressources, plus la ressource demandée. Ce qui signifie que le processus accumule toujours plus de ressources ce qui peut créer une famine parmis les autres.
- Impossibiilité de réquisitionner une ressource ? Il n'est pas possible de s'assurer que les ressources seront dans un bon état lorsqu'elle sont réquisitionnées
- L'attente circulaire ? On peut essayer de détecter un cycle et si un cycle arrive, on peut par exemple numéroté chaque ressource et imposer aux ressources de demander les ressrouces dans l'ordre croissant de leur numéro.

Eviter l'attente circulaire

Pour éviter l'attente circulaire il faut donc savoir la quantité de ressources disponibles et occupées ainsi que les besoins de chaque processus.

Le système est dit en **état sûr** s'il est capable de satisfaire tous les processus. Et tant que le système évolue d'état sûr en état sûr, aucun interblocage ne peut survenir. Ce pendant un état non sûr ne conduit pas nécessairement à un interblocage.

Algorithme du banquier

> Vidéo d'explication de l'algorithme du banquier

Compléter les informations que l'on a

Au total pour pouvoir appliquer l'algorithme du banquier il nous faut :

- La matrice des ressources existantes (E)
- La matrice des besoins des processus (B)
- La matrice des allocations courrantes (C)
- La matrice des ressources disponibles (A), qui correspond aux ressources existantes - les allocations courrantes (E-C)
- La matrice des demandes des processus (R), qui correspond aux besoins des processus - les allocations courrantes (B-C)

Les matrices que l'on va vraiment utiliser pour l'algorithmes sont celles des allocations courrantes (C), des demandes (R) et des ressources disponibles (A).

Vérfier si le système est dans un état sûr

- Pour chaque processus on va regarder si on peut remplir sa demande (R) à partir des ressources disponibles (A).
  - Si c'est possible, alors on marque le processus comme terminé et on ajoute aux resources disponibles (A) toutes les allocations du processus terminé (C).
- On fait cela en boucle jusqu'a arriver à un résultat où tous les processus (C) sont terminés. Si à la fin tous les processus ne sont pas terminé, alors l'état n'est pas sûr.

Pour allouer depuis un état sûr

- Pour un processus qui demande une ressource, on va *hypothétiquement* diminuer les ressources disponible de la demande, on va augmenter ses ressources allouées et diminuer ses besoins. C'est à dire que l'on va faire :
  - ressources disponibles -= demande
  - besoins -= demande
  - ressources allouées += demande
- On va ensuite effectuer l'algorithme précédent pour vérifier si ce système hypothétique est en état sûr, si c'est le cas, alors on peut allouer, sinon il faut attendre

Détecter un interblocage

Le problème avec la première solution est que l'on ne sait pas en avance ce dont les processus ont besoins. Et il est plus efficace de simplement détecter et corriger un interblocage que d'empécher un interblocage car les interblocages restent peu fréquent.

Cet algorithme de détection et de correction va se lancer lorsque le CPU n'est plus utilisé, ce qui signifie que les processus sont en état "waiting".

Détection d'un cycle d'attente dans l'allocation

Pour détecter un interblocage il suffit de simplement connaitre les ressources disponibles, les allocations courrantes et les demandes actuelles.

Pour chaque processus en cours on va vérifer si ses demandes actuelles peuvent être satisfaite avec les ressources disponibles. Pour chaque processus trouvé, on va incrémenter les ressources disponibles des allocations courrantes et on va définir le processus comme terminé.

Si à la fin il reste des demandes non satisfaite, il y a un interblocage.

Correction d'un interblocage

Pour corriger un interblocage on va tuer un processus qui pose problème et tenter de maintenir les ressources dans un état cohérent.

On peut donc faire un rollback vers le contexte où le système était avant pour s'assurer que les ressources ne sont pas dans un état dégueulasse, du moins si on sauvegarde le contexte du processus régulièrement.

La politique de l'autruche

Sur certains systèmes (tel que les systèmes UNIX), c'est à l'administrateur·ice de s'occuper de gérer un interblocage et le système d'exploitation s'en fout.