Programmation système

Steve Alabi - Taha Bendjeddou - Younes Benyamna - Malek Zemni

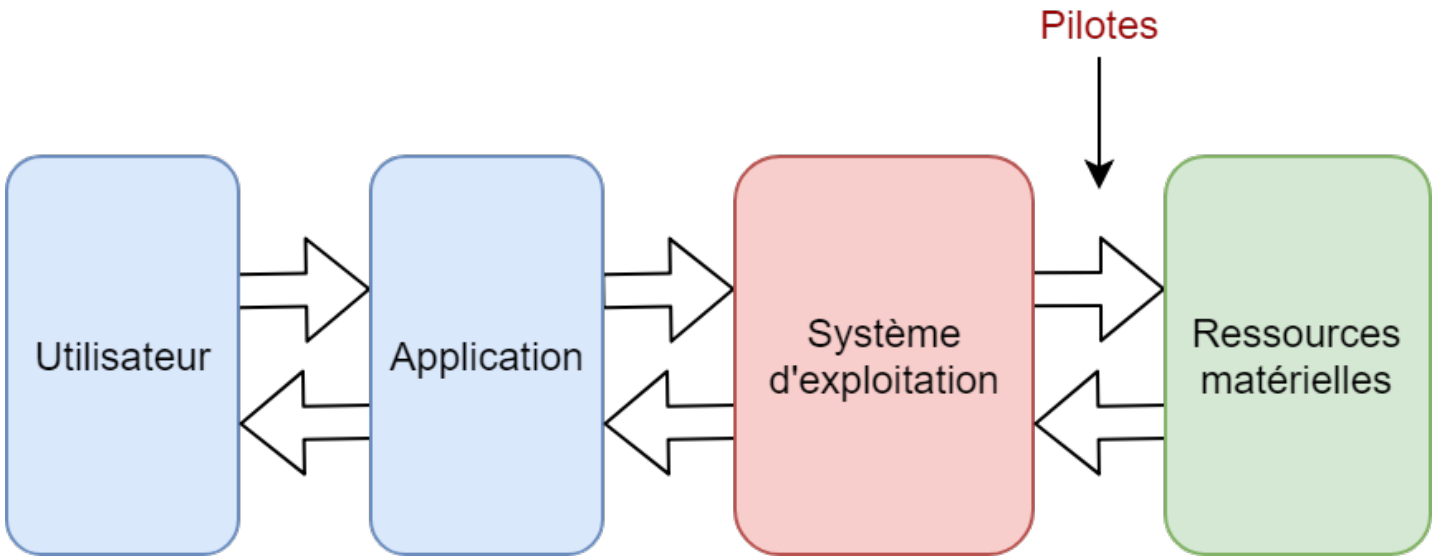
M2 SeCReTS - UVSQ

22/11/2018

Plan

Système d'exploitation

Ensemble de programmes qui dirigent l'utilisation des capacités d'un ordinateur par des logiciels applicatifs.
Il reçoit des demandes d'utilisation des capacités de l'ordinateur (stockage des mémoires et des disques durs, calcul du processeur, communication vers des périphériques ou via le réseau) de la part des logiciels applicatifs.
Le système d'exploitation accepte ou refuse ces demandes, puis réserve les ressources en question pour éviter que leur utilisation n'interfère avec d'autres demandes provenant d'autres logiciels.



-> dissocier les programmes et le matériel, afin notamment d'assurer la **gestion des ressources** et offrir à l'utilisateur une interface homme-machine simplifiée lui permettant de s'affranchir de la complexité de la machine physique.

Composantes d'un système d'exploitation :

- **Noyau (kernel)** : représente les fonctions fondamentales du système, gère les ressources de l'ordinateur et permet aux différents composants — matériels et logiciels — de communiquer entre eux.
- **L'interpréteur de commande (shell)** : permet la communication avec le système d'exploitation par l'intermédiaire d'un langage de commandes.
- **Le système de fichiers** : permet d'enregistrer les fichiers dans une arborescence.

Programmation système :

Type de programmation qui vise au développement de programmes qui font partie du système d'exploitation d'un ordinateur ou qui en réalisent les fonctions.

Elle se distingue de la ***programmation des applications*** en ce qu'elle s'intéresse non pas au traitement des données, mais à la résolution des problèmes pour les humains, aux interfaces (API), aux protocoles (communication) et à la gestion des ressources.

En réalité, seuls les ***programmes d'application*** sont utilisés pas les utilisateurs. Les ***programmes système*** le sont implicitement.

La programmation système inclut, en outre, l'accès aux fichiers, la gestion de la mémoire vive et des processeurs et la programmation de tous les périphériques qui font entrer ou sortir de l'information d'un ordinateur (clavier, écran, modems...). Elle permet donc de communiquer avec ces périphériques, créer des pilotes, voire même créer un système d'exploitation.

Sous UNIX :

90% du code est écrit en ***langage C*** (qui a été spécialement créé pour le développement du système UNIX), le reste est écrit en assembleur.

- ***couche logicielle*** *:* regroupe tous les programmes qui fournissent des services similaires.
- ***couche d'abstraction matérielle*** *:* une couche logicielle qui a accès au matériel informatique.

-> Le ***noyau*** est une sorte de logiciel d'arrière-plan qui assure les communications entre ces programmes. C'est donc par lui qu'il va falloir passer pour avoir accès aux informations du système.

Appels-systèmes : fonctions qui permettent de communiquer avec le noyau depuis l'espace utilisateur, en demandant des services ou des ressources au système.

Plan

Les flux :

Les flux peuvent être représentés comme des canaux dans lesquels circulent des informations. Ces flux sont représentés par l'abstraction de base du disque dur : le fichier. Ainsi, tout ce qui sera écrit dans un des flux sera écrit dans le fichier associé.

- le flux d'entrée standard `stdin`
- le flux de sortie standard `stdout`
- le flux de sortie d'erreurs standard `stderr`

Le flux d'erreurs standard `stderr` se modélise par un fichier dans lequel on peut écrire nos erreurs. Grâce à cette écriture séparée, l'utilisateur pourra rediriger le flux standard afin d'isoler les erreurs et ainsi mieux les traiter, en nombre moins importants, devant la masse d'informations contenues en sortie.

La variable globale `errno` :

Pour signaler une erreur, les fonctions renvoient une valeur spéciale, indiquée dans leur documentation. Celle-ci est généralement **-1** (sauf pour quelques exceptions).
La valeur d'erreur alerte l'appelant de la survenance d'une erreur, mais elle ne fournit pas la description de ce qui s'est produit. La variable globale `errno` est alors utilisée pour en trouver la cause.

```
#include <errno.h>

extern int errno;
```

Sa valeur est valable uniquement juste après l'utilisation de la fonction que l'on veut tester. Elle contient donc une valeur correspondant au code de la dernière erreur s'étant produite.

À chaque valeur d'erreur possible de `errno` correspond une constante du préprocesseur, dont la description est disponible dans le manuel (`man errno`).

Il est difficile d'associer directement le contenu de `errno` à des codes d'erreur, car la portabilité risque de faire défaut. C'est pourquoi la bibliothèque standard de C met à disposition deux fonctions qui interprètent le contenu de `errno`.

La fonction `strerr` + la fonction `perror` plus utilisée et plus simple d'usage, associe à la valeur courante de `errno` sa description, l'affichant sur la sortie d'erreurs standard `stderr`. Il est également possible de placer un préfixe `s` devant cette description, que l'on pourra passer en paramètre.

```
if (fork() == -1) {
   perror("fork");
}
//Affiche "fork : Description de l'erreur"
```

Plan

Entrées / sorties standard où les données sont mises en mémoire temporaire (***buffer*** *ou mémoire tampon, zone de mémoire virtuelle utilisée pour stocker temporairement les données, notamment entre 2 processus*) (avec des fonctions standard).

Font eux-même appel à des fonctions système d'entrée sortie.

fopen, **fclose**, **fgetc**, **fgets**, **fputc**, **fputs**...

Les fonctions d'entrée / sortie système sont des appels-systèmes communiquant directement avec le noyau. Les fonctions d'entrée / sortie standard font eux-même appel à ces fonctions système. Ces fonctions système se trouvent dans les bibliothèques suivantes :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

Ouvrir et fermer un fichier :

```
int open(const char* pathname, int flags, mode_t mode);
```

- **retour** : un "file descriptor" `fd` >= 0, prend la plus petite valeur disponible (non ouvert) ou **-1** si l'ouverture échoue, auquel cas errno contient le code d'erreur
- **pathname** : chemin du fichier à ouvrir
- **flags** : mode d'accès, contient une ou plusieurs valeurs dont obligatoirement une parmi O_RDONLY, O_WRONLY ou O_RDWR
- **mode** : indique les permissions à utiliser si un nouveau fichier est créé, doit être fourni lorsque O_CREAT est spécifié dans `flags` sinon ignoré

```
int close(int fd);
```

- **retour** : 0 ou -1 en cas d'échec
- **fd** : descripteur du fichier à fermer

Lire et écrire dans un fichier :

```
ssize_t read(int fd, void* buf, size_t count);
```

- **retour** : -1 en cas d'échec, et la position de la tête de lecture est indéfinie, sinon, renvoie le nombre d'octets lus (0 en fin de fichier), et avance la tête de lecture de ce nombre
- **fd** : descripteur du fichier à lire
- **buf** : pointe sur un tampon (buffer) où sont stockées les données lues
- **count** : nombre d'octets à lire, résultat indéfini si `count` est supérieur à `SSIZE_MAX`

```
ssize_t write(int fd, const void* buf, size_t count);
```

- **retour** : -1 en cas d'échec, sinon, renvoie le nombre d'octets écrits et avance la tête de lecture de ce nombre
- **fd** : descripteur du fichier dans lequel on va écrire
- **buf** : pointe sur un tampon (buffer) où sont stockées les données à écrire
- **count** : nombre d'octets à écrire

```c
int copie (char* fic1, char* fic2)
{
   int fd1 = open(fic1, O_RDONLY);
   int fd2 = open(fic2, O_WRONLY);

   int nb;
   char buf[1024];

   // tq nb != 0 et != -1
   while ( (nb = read(fd1, buf, 1024)) > 0 )
      if ( write(fd2, buf, nb) != nb )
         return -1;

   return 1;
}
```

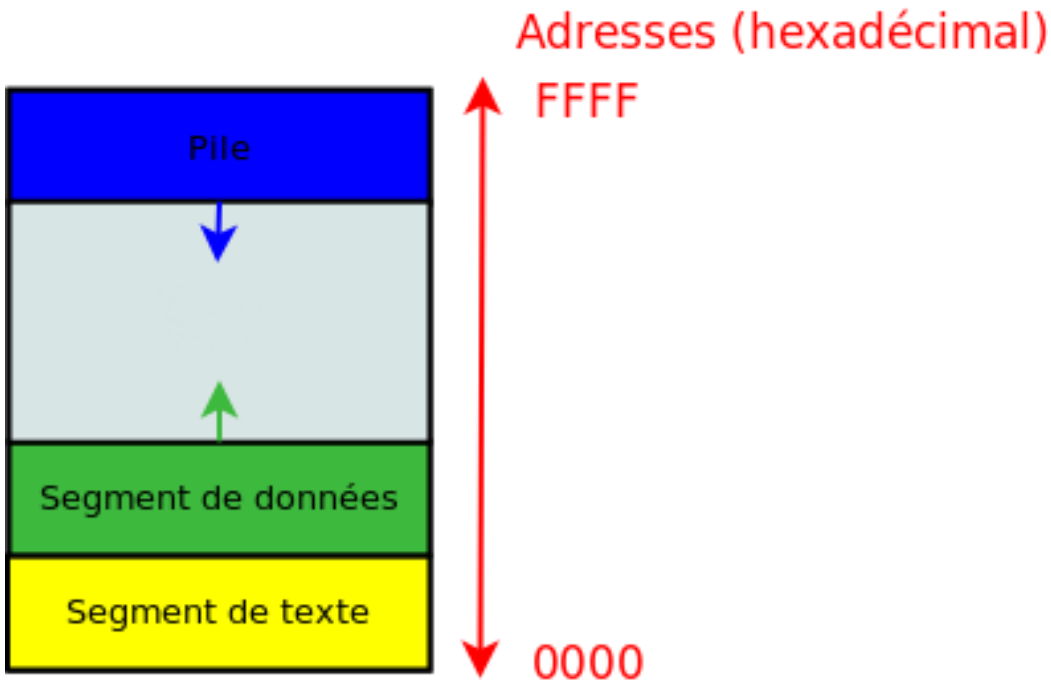
```c
int myGetChar (int fd)
{
   int retour;
   unsigned char c;

   if ( read(fd, &c, 1) == 1 )
      retour = c;
   else
      retour = EOF;

   return retour;
}
```

```c
int myGetCharBuffer (int fd)
{
   static unsigned char buf[MAX];
   static int nbCar = 0;    // nombre de caractères dans buf
   static char* p;    // tête de lecture

   int retour;

   // on fait un read si le buffer est vide
   if ( nbCarac == 0 ) {
      nbCarac = read(fd, buf, MAX);
      p = buf;
   }

   // si le buffer contient des caractères, on en lit le premier
   if ( nbCarac > 0)
      retour = *p;    // valeur à la tête de lecture
   else
      retour = EOF;  // plus rien à lire

   nbCarac--;
   p++;  // on avance la tête de lecture

   return retour;
}
```

Plan

Programme vs Processus

Un processus est programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire. En effet, au cours de son exécution, les instructions d'un programme modifient les valeurs des registres (compteur ordinal, registre d'état...) ainsi que le contenu de la pile.
Un programme est une suite d'instructions (notion statique), tandis qu'un processus, c'est l'image du contenu des registres et de la mémoire centrale (notion dynamique).



Lorsqu'un processus est lancé, le système doit gérer la mémoire et l'allocation du processeur lui étant accordée. Il fait appel à l'***ordonnanceur***.

Sous les autres systèmes d'exploitation, un processus est l'équivalent d'un nouveau programme, alors que sous UNIX ce n'est pas forcément le cas.

La commande `ps` permet d'afficher ses propres processus en cours d'exécution. Pour afficher tous les processus en cours d'exécution, on utilise l'option `aux` (a : processus de tous les utilisateurs, u : affichage détaillé, x : démons).

PID :

Chaque processus peut être identifié par son numéro de processus, ou PID (***Process IDentifier***). Un numéro de PID est unique dans le système : il est impossible que deux processus aient un même PID au même moment.

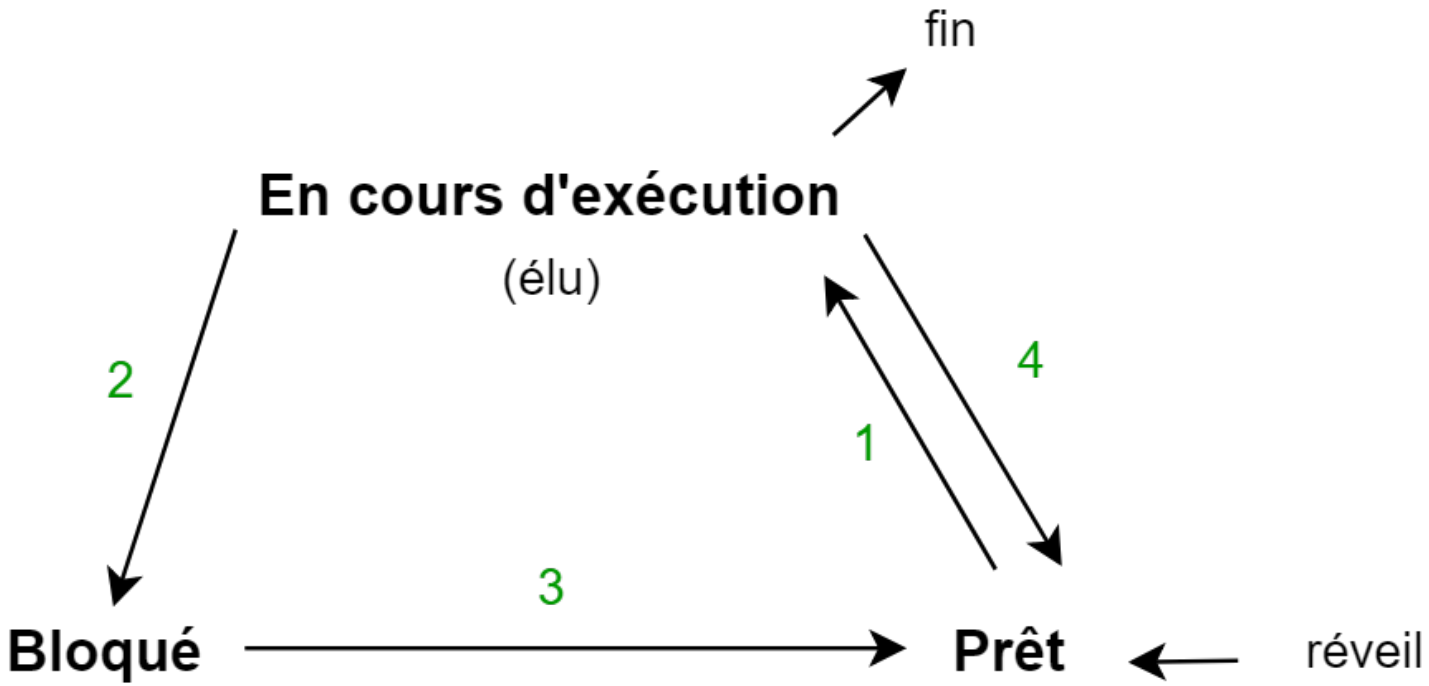
Organisation des processus :

Lorsque l'on crée un processus, on utilise une fonction qui permet de dupliquer le processus appelant. On distingue alors les deux processus par leur PID. Le processus appelant est alors nommé **processus père** et le nouveau processus **processus fils**. Quant on s'occupe du processus fils, le PID du processus père est noté **PPID** (***Parent PID***).

-> Les processus sont organisés en ***hiérarchie***.

États d'un processus :

- exécution (R pour running) : le processus est en cours d'exécution
- sommeil (S pour sleeping) : dans un multitâche coopératif, quand il rend la main, ou dans un multitâche préemptif, quand il est interrompu au bout d'un quantum de temps
- arrêt (T pour stopped) : le processus a été temporairement arrêté par un signal, il ne s'exécute plus et ne réagira qu'à un signal de redémarrage
- zombie (Z pour ... zombie) : le processus fils (zombie) s'est terminé avant le père



- 1 : élection, l'ordonnanceur choisit ce processus
- 2 : blocage, processus bloqué en attente d'une donnée
- 3 : déblocage, donnée devient disponible
- 4 : l'ordonnanceur choisit un autre processus

Création d'un processus :

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

pid_t fork(void);
```

Le processus d'origine est le processus père et le nouveau processus créé est le processus fils, qui possède un nouveau PID. Les processus deux ont le même code source, mais la valeur retournée par `fork` nous permet de savoir si l'on est dans le processus père ou dans le processus fils. Ceci permet de faire deux choses différentes dans le processus père et le processus fils.

La fonction `fork` retourne :

- -1 en cas d'erreur
- 0 si on est dans le processus fils
- le PID du fils si on est dans le processus père. Cela permet ainsi au père de connaître le PID de son fils

Autres fonctions :

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

pid_t getpid(void);
// retourne le PID du processus appelant

pid_t getppid(void);
// retourne le PPID du processus appelant
```

Terminaison d'un processus :

Un programme peut se terminer normalement de deux façons différentes. La plus simple consiste à laisser le processus finir le `main` avec l'instruction **`return`** suivie du code de retour du programme. Une autre façon est de terminer le programme grâce à la fonction `exit()` depuis n'importe quelle fonction.

```
#include <stdlib.h>

void exit(status);
```

Un programme peut aussi se terminer de façon anormale, en cas d'erreur par exemple. Pour cela, on peut utiliser les fonctions `abort(void)` ou `assert(int condition)` par exemple.

Synchronisation entre processus :

Un processus **zombie** est un processus qui s'est achevé, mais qui dispose toujours d'un identifiant de processus (PID) et reste donc encore visible dans la table des processus. Lorsque le processus fils se termine avant le processus père (c'est à dire que le père n'a pas encore lu le code de retour du fils), le processus fils devient un zombie.

Pour permettre à un processus fils zombie de disparaître complètement, on utilise la fonction `wait()`.

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

pid_t wait(int* status);
```

Lorsque l'on appelle cette fonction, celle-ci bloque le processus à partir duquel elle a été appelée (père) jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID de ce dernier. En cas d'erreur, la fonction renvoie la valeur -1.

Le paramètre `status` correspond au code de retour du processus (ce code de retour est généralement indiqué avec la fonction exit).
Il faut veiller à mettre autant de `wait` dans le père qu'il a de processus fils.

La fonction `waitpid` :

Permet de suspendre l'exécution d'un processus père jusqu'à ce qu'un de ses fils, dont on doit passer le PID en paramètre, se termine.

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

pid_t waitpid(pid_t pid, int* status, int options);
\\waitpid(-1, status, 0) correspond à wait
```

- si pid $>$ 0, le processus père est suspendu jusqu'à la fin d'un processus fils dont le PID est égal à la valeur pid
- si pid $=$ 0, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils appartenant à son groupe
- si pid $=$ -1, le processus père est suspendu jusqu'à la fin de n'importe lequel de ses fils
- si pid $<$ -1, le processus père est suspendu jusqu'à la mort de n'importe lequel de ses fils dont le GID est égal
- `status` a le même rôle qu'avec `wait`
- `options` permet de préciser le comportement de `waitpid`

Le Papa

Descripteurs de fichiers

1: (Stdout) Position = 0

Variables globales

const char* quisuisje = "Le pere";

```
int main()
{
   pid_t pid;
   quisuisje = "Le pere";
   pid = fork(); //pid = 1000
   if(pid == 0){
      quisuisje = "Le fils";
      printf("Je suis %s", quisuisje);
   }
   else{
   ➡ printf("Je suis %s", quisuisje);
      wait(NULL);
   }
   return 0;
}
```

Le Fiston

Descripteurs de fichiers

1: (Stdout) Position = 0

Variables globales

const char* quisuisje ="Le pere";

```
int main()
{
   pid_t pid;
   quisuisje = "Le pere";
   pid = fork(); //pid=0
   if(pid == 0){
   ➡ quisuisje = "Le fils";
      printf("Je suis %s", quisuisje);
   }
   else{
      printf("Je suis %s", quisuisje);
      wait(NULL);
   }
   return 0;
}
```

Le Papa

Descripteurs de fichiers

1: (Stdout) Position = 30

Variables globales

const char* quisuisje = "Le pere";

```
int main()
{
    pid_t pid;
    quisuisje = "Le pere";
    pid = fork(); //pid = 1000
    if(pid == 0){
        quisuisje = "Le fils";
        printf("Je suis %s", quisuisje);
    }
    else{
        printf("Je suis %s", quisuisje);
        wait(NULL);
    }
    return 0;
}
```

Le Fiston

Descripteurs de fichiers

1: (Stdout) Position = 30

Variables globales

const char* quisuisje ="Le fils";

```
int main()
{
    pid_t pid;
    quisuisje = "Le pere";
    pid = fork(); //pid=0
    if(pid == 0){
        quisuisje = "Le fils";
        printf("Je suis %s", quisuisje);
    }
    else{
        printf("Je suis %s", quisuisje);
        wait(NULL);
    }
    return 0;
}
```

La variable `quiSuisJe` va être modifiée pour le processus fils seulement. La position de `stdout` va avancer de 15 caractères pour le `printf` du père et 15 autres caractères pour le `printf` du fils pour valoir 30 au final pour les deux processus.

Les variables du processus père et celles du fils sont totalement distinctes. Par contre, leur descripteurs de fichiers sont les mêmes. Donc, si l'un des deux processus modifie son pointeur de position dans un fichier, ça se répercutera également chez l'autre. Cela ne vaut que pour les descripteurs de fichiers hérités durant le `fork`, c'est-à-dire, si le père ou le fils ouvre d'autres fichiers après le `fork`, ces descripteurs ne seront pas partagés entre eux deux. De même, si le fils ferme un descripteur de fichier hérité du père, le descripteur de fichier du père ne sera par contre pas fermé (même chose dans le sens inverse).

On ne peut pas savoir quel processus va s'exécuter en premier. Pour la fonction `wait`, il est préférable l'utiliser ainsi :

```
if ( wait(NULL) == -1 )
   perror("wait");
```

Plan

Un **thread** ou **fil (d'exécution)** ou **tâche** (aussi processus léger) représente, comme un processus, l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Dans la plupart des systèmes d'exploitation, chaque processus possède un espace d'adressage et un thread de contrôle unique, le thread principal. Du point de vue programmation, ce dernier exécute le `main`.

En général, le système réserve un processus à chaque application, sauf quelques exceptions. Beaucoup de programmes exécutent plusieurs activités en parallèle, du moins en pseudo-parallélisme. Comme à l'échelle des processus, certaines de ces activités peuvent se bloquer, et ainsi réserver ce blocage à un seul thread séquentiel, permettant par conséquent de ne pas stopper toute l'application.

Créer un thread :

Pour créer un thread, il faut déjà déclarer une variable le représentant. Celle-ci sera de type `pthread_t`. Ensuite, pour créer la tâche elle-même, il suffit d'utiliser la fonction :

```
#include <pthread.h>

int pthread_create(pthread_t* thread, pthread_attr_t * attr, void*
    (*start_routine) (void*), void* arg);
```

- retour : 0 si la création a été réussie ou une autre valeur si il y a eu une erreur
- `thread` : pointeur vers l'identifiant du thread à créer
- `attr` : attributs du thread, possible de mettre le thread en état joignable (NULL par défaut) ou détaché, et choisir sa politique d'ordonnancement (usuelle, temps-réel...)
- `(*start_routine) (void*)` : pointeur vers la fonction à exécuter dans le thread, de la forme `void* fonction(void* arg)` et contiendra le code à exécuter par le thread
- `arg` : l'argument à passer au thread

Supprimer un thread :

```c
#include <pthread.h>

void pthread_exit(void* ret);
```

Prend en argument la valeur qui doit être retournée par le thread, et doit être placée en dernière position dans la fonction concernée.

Attendre la fin d'un thread :

Le thread principal n'attendra pas que le thread créé termine de s'exécuter. La fonction `pthread_join` pour cela.

```c
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

Elle prend en premier paramètre l'identifiant du thread et son second paramètre, un pointeur, permet de récupérer la valeur retournée par la fonction dans laquelle s'exécute le thread (c'est-à-dire l'argument de `pthread_exit`).

```c
void* thread_1(void *arg) {
   printf("Nous sommes dans le thread.\n");
   pthread_exit(NULL);
}

int main(void) {
   pthread_t thread1;

   printf("Avant la création du thread.\n");

   if (pthread_create(&thread1, NULL, thread_1, NULL))
   {
      perror("pthread_create");
      return EXIT_FAILURE;
   }

   // on attend que le thread s'exécute
   if (pthread_join(thread1, NULL))
   {
      perror("pthread_join");
      return EXIT_FAILURE;
   }

   printf("Après la création du thread.\n");

   return EXIT_SUCCESS;
}
```

Avec les threads, toutes les variables sont partagées (mémoire partagée). Cela pose des problèmes quand deux threads cherchent à modifier deux variables en même temps. Les mutex est mécanisme de synchronisation, un des outils permettant l'exclusion mutuelle.

Un **mutex** en C est une variable de type `pthread_mutex_t`. Elle va nous servir de verrou, pour nous permettre de protéger des données. Ce verrou peut donc prendre deux états : disponible et verrouillé.

Quand un thread a accès à une variable protégée par un mutex, on dit qu'il tient le mutex. Il ne peut y avoir qu'un seul thread qui tient le mutex en même temps.

Le mutex doit être accessible en même temps que la variable et dans tout le fichier (vu que différents threads s'exécutent dans différentes fonctions). La solution la plus simple consiste à déclarer les mutex en variable globale.

Initialiser un mutex :

On initialise un mutex avec la valeur de la constante `PTHREAD_MUTEX_INITIALIZER`, déclarée dans `pthread.h`.

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Verrouiller un mutex :

L'étape suivante consiste à établir une **zone critique**, c'est-à-dire la zone où plusieurs threads risquent de modifier ou de lire une même variable en même temps. On verrouille donc le mutex pour éviter ce risque.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mut);
```

Déverrouiller un mutex :

À la fin de la zone critique, il suffit de déverrouiller le mutex.

```
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mut);
```

Détruire un mutex :

Une fois le travail du mutex terminé, on peut le détruire.

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mut);
```

Les conditions :

Lorsqu'un thread doit patienter jusqu'à ce qu'un événement survienne dans un autre thread, on utilise les conditions.
Quand un thread est en attente d'une condition, il reste bloqué tant que celle-ci n'est pas réalisée par un autre thread.

Comme avec les mutex, on déclare la condition en variable globale :

```
pthread_cond_t nomCondition = PTHREAD_COND_INITIALIZER;
```

Pour attendre une condition, il faut utiliser un mutex :

```
int pthread_cond_wait(pthread_cond_t *nomCondition, pthread_mutex_t
    *nomMutex);
```

Pour réveiller un thread en attente d'une condition, on utilise la fonction :

```
int pthread_cond_signal(pthread_cond_t *nomCondition);
```

Créez un code qui crée deux threads : un qui incrémente une variable compteur par un nombre tiré au hasard entre 0 et 10, et l'autre qui affiche un message lorsque la variable compteur dépasse 20.

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

pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void* threadAlarme (void* arg);
void* threadCompteur (void* arg);

int main (void)
{
   pthread_t monThreadCompteur;
   pthread_t monThreadAlarme;

   pthread_create (&monThreadCompteur, NULL, threadCompteur,
     (void*)NULL);
   pthread_create (&monThreadAlarme, NULL, threadAlarme, (void*)NULL);

   pthread_join (monThreadCompteur, NULL);
   pthread_join (monThreadAlarme, NULL);

   return 0;
}
```

```c
void* threadCompteur (void* arg)
{
   int compteur = 0, nombre = 0;
   srand(time(NULL));

   while(1) {
      nombre = rand()%10; /* On tire un nombre entre 0 et 10 */
      compteur += nombre; /* On ajoute ce nombre à la variable compteur
    */

      printf("\n%d", compteur);

      if(compteur >= 20) /* Si compteur est plus grand ou égal à 20 */
      {
         pthread_mutex_lock (&mutex); /* On verrouille le mutex */
         pthread_cond_signal (&condition); /* On délivre le signal :
    condition remplie */
         pthread_mutex_unlock (&mutex); /* On déverrouille le mutex */

         compteur = 0; /* On remet la variable compteur à 0 */
      }
      sleep (1); /* On laisse 1 seconde de repos */
   }
   pthread_exit(NULL); /* Fin du thread */
}
```

```
void* threadAlarme (void* arg)
{
   while(1) /* Boucle infinie */
   {
      pthread_mutex_lock(&mutex); /* On verrouille le mutex */
      pthread_cond_wait (&condition, &mutex); /* On attend que la
    condition soit remplie */
      printf("\nLE COMPTEUR A DÉPASSÉ 20.");
      pthread_mutex_unlock(&mutex); /* On déverrouille le mutex */
   }

   pthread_exit(NULL); /* Fin du thread */
}
```