Multitâches

Processus

Une processus est la session d'exécution d'un programme.
Caractéristiques d'un processus :

- Le programme qu'il exécute.
- Un identifiant : PID
- Zone de données de variables : global, local (pile), malloc (tas)
  Fin du programme, fin du processus

Création d'un processus par programmation

La fonction `fork()` permet de créer un processus fils du processus du programme exécuté.
Le processus fils exécute le même programme que le processus père en démarrant à l'instruction suivant `fork()`.
La zone de données du fils est une copie de celle du père à l'instant du `fork()`.

La fonction retourne des valeurs selon le notre point de vu :

- Chez le fils, la valeur de retour vaut 0.
- Chez le père, la valeur est positif et équivaut au PID du fils ou négatif s'il y a une erreur.

```
int pid;
pid = fork()
if(p == 0) {
        // Les instructions du fils.
}
else {
        // Les instructions du père.
}
// Instruction exécuté par les deux.
```

Mécanisme execv

La fonction `execv(<file_path>,<args_tab>)` permet de changer le programme exécuter par un processus.
Les données du premier programme sont libérer et une nouvelle pile de données est allouer au nouveau programme.
Lorsque le deuxième programme est fini, alors le processus se termine.
Pour passer des arguments nous utiliserons un tableau `args[size]` dont la taille est le nombre d'arguments passés + 2. Si aucun arguments doit être passé il faut au moins deux

éléments dans le tableaux. Celui-ci commence toujours par le chemin du fichier et se termine par la valeur `NULL` pour déterminer la fin de la chaine d'argument.

```
int pid;
if(p == 0) {
        execv("progB",...); // Exécute un nouveau programme dans le fils.
}
else {
        // Instructions du père.
}
```

Fin d'un processus

Un processus se termine toujours par un exit.
La valeur de retour du main équivaut à la valeur de l'exit d'un processus.
La valeur de retour est codée sur 8bits donc vas de 0 à 255.
`exit(<value>)` OU *dans le main* `return <value>`

Mécanisme wait

> **Remarque**
>
> Le père et le fils ont des exécution indépendant.
> La fin de l'un nécessite par la fin de l'autre.

La fonction `wait()` est utiliser seulement lorsque le père doit se synchroniser sur la fin d'un fils. Le père est débloqué dès la fin d'un fils.
Si l'instruction `wait()` est exécuter mais que l'in des fils est déjà terminé alors passé à l'instruction suivant directement.
La fonction retour -1 si échec ou le PID du fils terminé.

Paramètre de `wait()` :

1. Le père n'est pas intéressé par le code d'exit du fils. Passage du pointeur NULL.
2. Le père souhaite le code d'exit. Passage d'une adresse de variable à la fonction wait() qui vas stocker la valeur d'exit.
   Puisque la variable est est initialiser en 32bit (int), et que la valeur d'exit n'est que sur 8bits les autre bits sont utilisés pour avoir d'autre information. (Voir documentation).
   Pour récupérer facilement récupère le code exit (de 8bits sur les 32bits) il faut utiliser la fonction `WEXITSTATUS(<variable>);`

Fonctions outils

La fonction `getpid();` permet de retourner le PID du processus appelant.
La fonction `getppid();` permet de retourner le PID du processus père de l'appelant.

Les commande Linux

La commande bash (Linux) : `ps` permet de visualiser l'ensemble des processus exécuter en indiquant le PID de chacun d'entre eux et leur liaison père-fils.

Threads

Un thread est un fils d'exécution à l'intérieur d'un processus qui s'exécute en parallèle du reste du processus. Il est matérialisé par l'exécution d'une fonction.

Diagramme Thread

Création d'un Thread

La fonction `pthread_create(<addr_thread_id>, NULL, <function>, <args>)` est défini par deux argument majeur :

- L'adresse d'une variable `pthread_t` permettant de manipuler le thread.
- La fonction appelée pour créer le thread secondaire.

Fonction d'exécution d'un thread

Le prototype est imposé : `void *<function>(void *arg)`
Le contenue est libre.
Fin de la fonction implique la fin du thread, qui se fait par li biais d'un `return ...` OU `pthread_exit(...)`

> **Remarque**
>
> Dans le cas de l'exercice nous n'utiliserons pas de retour donc le mots clé `return` prendra la valeur NULL puisque la fonction attend le retour d'un pointeur.
> Voir les slides pour plus d'information sur les valeur de retour d'un fonction de thread.

Argument de la fonction d'un thread

Les arguments de la fonction d'un thread permet de lui passé des informations à son démarrage, ces informations sont passé par le biais du dernier paramètre de la fonction `pthread_create`. *La variable passé en argument **NE DOIT PAS** être une variable **LOCAL** au thread principal* (voir diagramme). Donc elle est de type **global**.

```c
void *my_function(void *arg)
{
        float *p = (float *)arg;
        float value = *p;
        .
        .
        .
        return NULL; // OR pthread_exit(NULL);
}
```

Si aucun arguments veut être passé alors le dernier paramètre de la fonction `pthread_create` prend comme valeur le pointeur `NULL`. A contrario si l'on souhaite passé en argument plusieurs données, il est conseille de passé par une structure pour manipuler différents type de variable.

Synchronisation sur la fin du thread

La fonction `pthread_join(<thread_id>, NULL);` est bloquante jusqu'à que le thread portant l'ID en paramètre n'est pas encore terminé. La fonction retourne 0 si OK ou une valeur si une erreur.

Serveur

Un serveur basique peut dialogue qu'avec un seul client à la fois.
Pour palier à ce problème, le serveur peut créer un thread par client.
Donc le serveur attend une connexion, reçoit une connexion, le délègue à un thread et retour en écoute de connexion. Ainsi de suie.

Pour que le Thread puisse communiquer avec le client on lui envoie en argument le canal de dialogue retourner par la fonction d'acceptation.