

Programmation Système

1. Références

2. Introduction

La programmation système fait référence au développement de logiciels qui interagissent directement **avec le matériel informatique et le système d'exploitation**. Ces programmes sont conçus pour gérer les **ressources système**, tels que les **fichiers**, les **périphériques**, la **mémoire** et les **processus**.

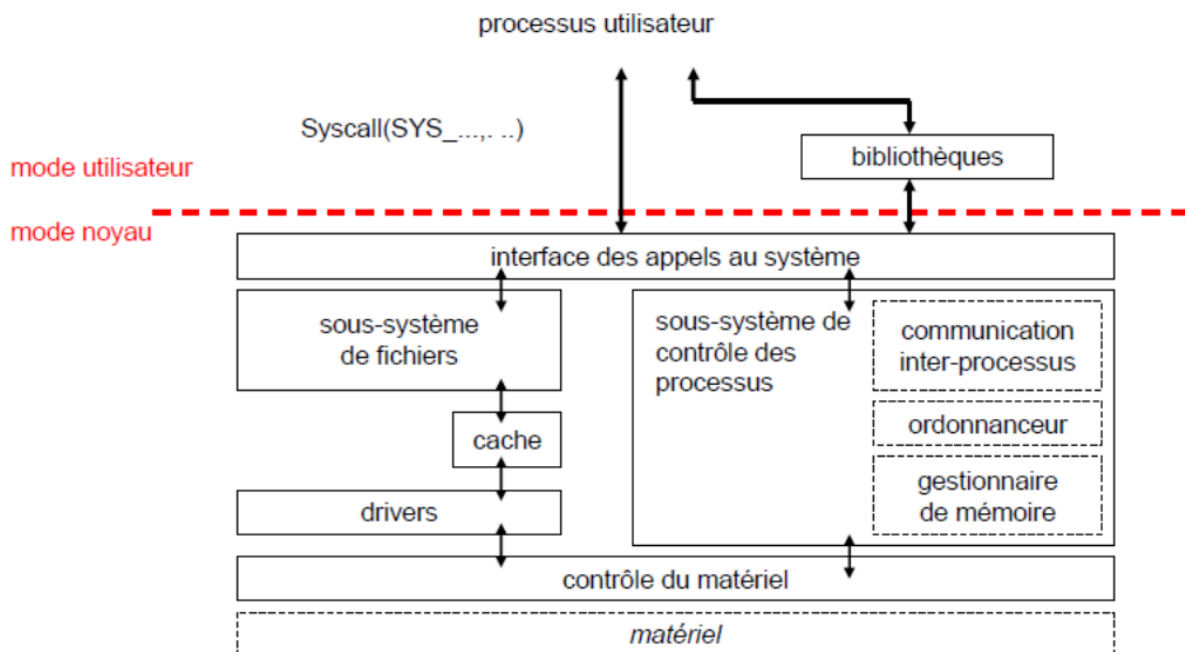
Voici quelques concepts et activités courants associés à la programmation système :

1. **Gestion des processus et de la mémoire** : La programmation système implique souvent la création, la manipulation et la gestion des processus et de la mémoire. Cela comprend la création de nouveaux processus, la gestion de leur cycle de vie, l'allocation et la libération de mémoire, etc.
2. **Gestion des fichiers et des entrées/sorties (E/S)** : Les programmes système doivent souvent interagir avec le système de fichiers et les périphériques d'E/S. Cela peut inclure la lecture et l'écriture de fichiers, la communication avec des périphériques comme des disques durs ou des imprimantes, etc.
3. **Communication inter-processus** : Les programmes système peuvent faciliter la communication entre différents processus, que ce soit via des mécanismes IPC (Inter-Process Communication) tels que les tubes, les files d'attente, les signaux, les sockets, etc.
4. **Gestion des threads** : En plus des processus, la programmation système peut également impliquer la création, la gestion et la synchronisation des threads, qui sont des unités d'exécution plus légères que les processus.
5. **Systèmes de fichiers et de périphériques** : La programmation système peut impliquer la création de pilotes de périphériques ou la manipulation directe des systèmes de fichiers pour effectuer des opérations telles que la lecture, l'écriture, la suppression, etc.
6. **Optimisation des performances** : Les programmes système sont souvent soumis à des contraintes de performance strictes, ce qui peut nécessiter une optimisation minutieuse des algorithmes et des structures de données.
7. **Sécurité** : En raison de l'accès direct au matériel et aux ressources système, la programmation système doit prendre en compte les considérations de sécurité pour éviter les vulnérabilités et les attaques potentielles.
8. **Portabilité** : Bien que de nombreux concepts de programmation système soient spécifiques à un système d'exploitation particulier, il est souvent souhaitable que le

code soit portable entre différentes plates-formes. Cela peut nécessiter l'utilisation de normes et de bibliothèques multiplateformes.

La programmation système est souvent réalisée en utilisant des langages de programmation bas-niveau tels que le **C** ou le **C++**, bien que d'autres langages comme **Rust** et **Python** gagnent également en popularité en raison de leurs fonctionnalités de sécurité et de performance.

NB : il faut différencier les programmes systèmes, des programmes d'application des utilisateurs. Ces programmes sont réalisés lors de la programmation dite « classique » de gestion.



Les 2 fonctions d'un système d'exploitation:

- Fournir au programmeur une abstraction (une interface de programmation commode) de la machine sur laquelle il travaille
- fournir les outils permettant de gérer les ressources de l'ordinateur et partager et protéger l'information qui y réside.

3. Initialiser les environnements du cours

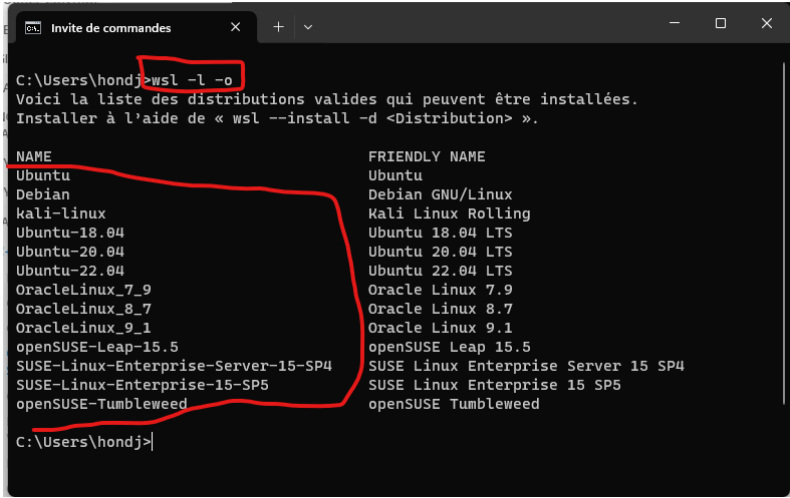
3.1. Visual Studio Code for C

1. Télécharger et installer <https://code.visualstudio.com/Download>
2. Installer les extensions C/C++ : <https://code.visualstudio.com/docs/languages/cpp>
3. **Ctrl+Shift+X** + rechercher "C++"

- 
- 4.
 5. Puis sur install
 6. pour prendre en main VScode suivre ce tutoriel :
 - a. <https://code.visualstudio.com/docs/cpp/config-wsl>

3.2. WSL2

1. Installer WSL :
 - a. suivre la procedure : <https://learn.microsoft.com/fr-fr/windows/wsl/install>
 - b. <https://code.visualstudio.com/docs/cpp/config-wsl>
2. Ouvrir une console DOS
 - a. lister les distributions linux disponible:
 - i. `wsl -l -o`



```

C:\Users\hondj>wsl -l -o
Voici la liste des distributions valides qui peuvent être installées.
Installer à l'aide de « wsl --install -d <Distribution> ».

NAME                                FRIENDLY NAME
Ubuntu                              Ubuntu
Debian                             Debian GNU/Linux
kali-linux                          Kali Linux Rolling
Ubuntu-18.04                        Ubuntu 18.04 LTS
Ubuntu-20.04                        Ubuntu 20.04 LTS
Ubuntu-22.04                        Ubuntu 22.04 LTS
OracleLinux_7_9                     Oracle Linux 7.9
OracleLinux_8_7                     Oracle Linux 8.7
OracleLinux_9_1                     Oracle Linux 9.1
openSUSE-Leap-15.5                  openSUSE Leap 15.5
SUSE-Linux-Enterprise-Server-15-SP4 SUSE Linux Enterprise Server 15 SP4
SUSE-Linux-Enterprise-15-SP5        SUSE Linux Enterprise 15 SP5
openSUSE-Tumbleweed                 openSUSE Tumbleweed

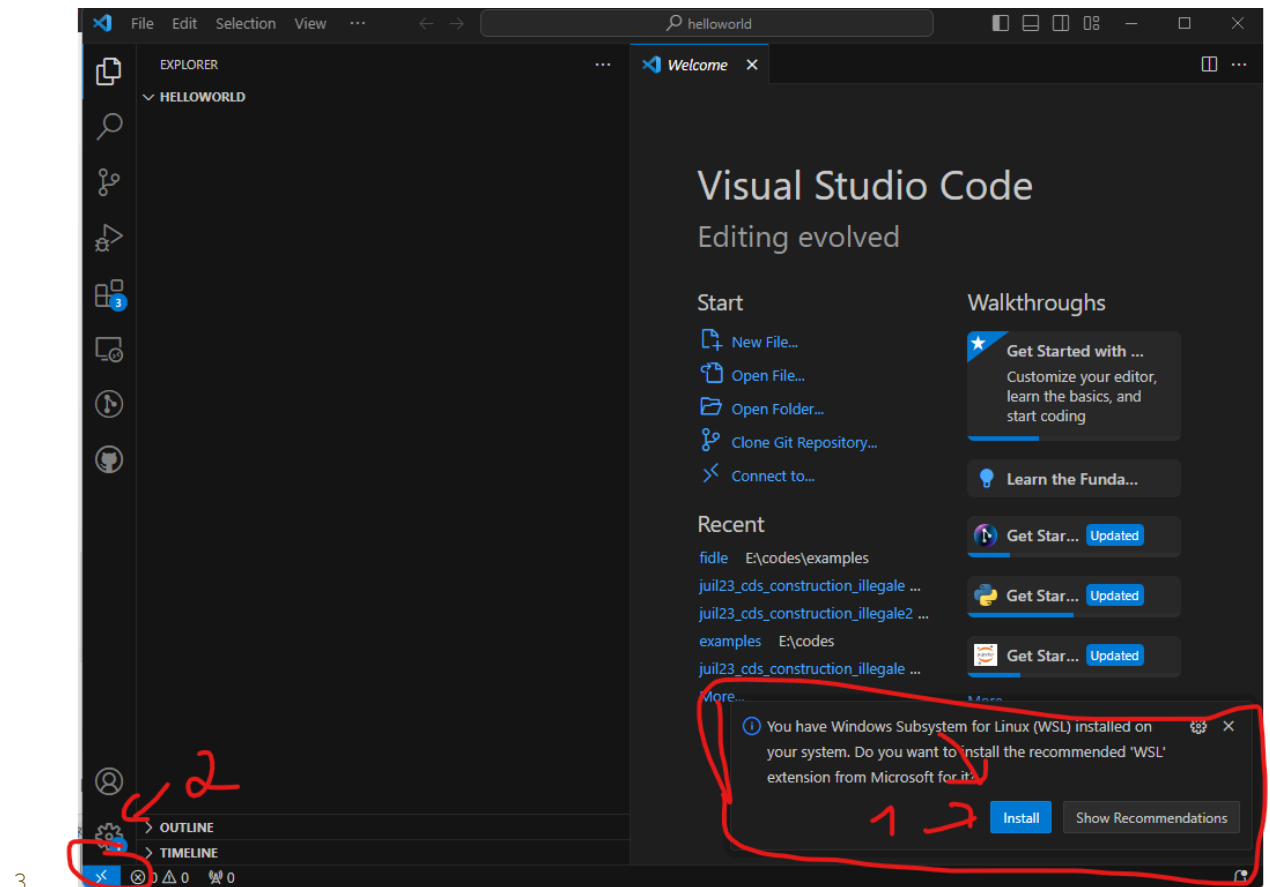
C:\Users\hondj>
  
```

- b.
- c. Choisir une version
 - i. Pour ce cours, nous choisirons : "Ubuntu"
- d. Installer la version de Linux
 - i. Exécuter la commande :
 1. `wsl --install -d ubuntu`
 - ii. Entrer un login et un mot de passe dans les prompts qui s'afficheront
 1. login : user
 2. mot de passe : ubuntu

La console suivante apparaîtra. Console ubuntu avec le user choisi

```
hondj@LAPTOP-OIKVA9GJ: ~  
Installing, this may take a few minutes...  
Please create a default UNIX user account. The username does not need to match your Windows username.  
For more information visit: https://aka.ms/wslusers  
Enter new UNIX username: hondj  
New password:  
Retype new password:  
passwd: password updated successfully  
Installation successful!  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
  
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.10.16.3-microsoft-standard-WSL2 x86_64)  
  
* Documentation: https://help.ubuntu.com  
* Management: https://landscape.canonical.com  
* Support: https://ubuntu.com/advantage  
  
This message is shown once a day. To disable it please create the  
/home/hondj/.hushlogin file.  
hondj@LAPTOP-OIKVA9GJ:~$ ls  
hondj@LAPTOP-OIKVA9GJ:~$ |
```

- 3.
- iii. Mettre à jour les packages ubuntu
 1. `sudo apt-get update`
- iv. Télécharger les packages indispensables pour les développements Linux
 1. `sudo apt-get install build-essential gdb`
- v. Vérifier l'installation
 1. `whereis g++`
 2. `whereis gdb`
 - 3.
- vi. Créer le dossier où sera stocké les code C/C++
 1. `mkdir projects`
 2. `cd projects`
 3. `mkdir helloworld`
 4. `cd helloworld`
- vii. lancer Vscode
 1. `code .`
 2. puis dans la fenêtre : installer le plugin WSL de VSCode (1)
 3. puis cliquer sur (2) pour se connecter à WSL



3.

4.

1.

2.

ii.

4. Librairies standards C

voir ce PDF

- https://webusers.i3s.unice.fr/~tettaman/Classes/L2I/ProgSys/0_LibrairieC_ANSI.pdf

4.1. #include <stdio.h>

4.2. #include <stdarg.h>

4.3. #include <stdarg.h>

4.4. #include <stdarg.h>

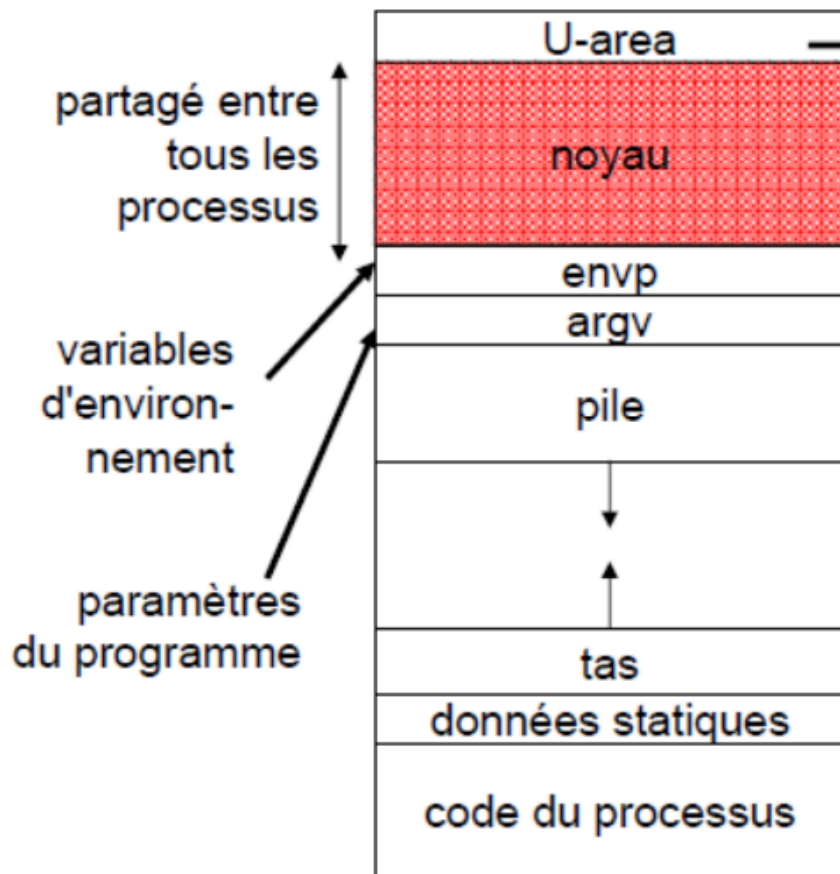
5. Processus Linux

Un processus sous linux est composé de:

- une espace d'adressage
- un (ou plusieurs) threads (flots de contrôle)
- un ensemble de ressources (fichiers ouverts, etc.)

2 structures fournissent l'information d'un processus :

- **U-area** (User area) : information nécessaire uniquement lorsque le processus est en exécution:
 - Sauvegarde du contexte hardware : lorsque le processus est en pause
 - pointeur de la structure Proc
 - UIS et GIS réels et effectifs
 - "handlers des signaux
 - descripteurs des fichiers ouverts
 - utilisateur CPU , quotas disques, etc.
 - pile "noyau" du processus
- **descripteur de processus**: information nécessaire même lorsque le processus ne s'exécute pas. Structure de données identifiée sous le nom proc(prédéfinie dans <sys/proc.h>). C'est une entrée de la **table des processus** du noyau.



gnor

Structure du Proc

- PID : identificateur du processus
- pointeur sur U-aera du processus
- liens avant et arrière de chaînage (pour inclure de la structure proc dans une des listes du noyau)
- priorité du processus
- information liées à la gestion des signaux (signaux bloqués, à ignorer, etc.)
- information de gestion de la mémoire
- liens pour inclure la structure dans une des listes de processus actifs , libres ou zombies
- informations indiquant la relation du processus par rapport à d'autres processus (pointeur sur proc du père, pointeur sur proc du 1er fils, etc.)

Identification d'un processus

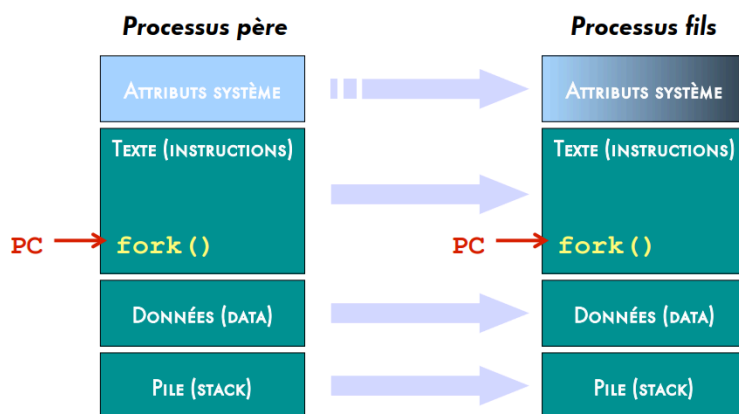
- PID
- P-PID
- PGRP : identification du groupe auquel appartient le processus (un entier)
 - par défaut , héritée du processus-père
 - permet de désigner un ensemble de processus
-

Fonctions sur les processus:

- fork
- exit
- wait
- exec

5.1. Création d'un processus - Fork()

- création d'un processus par clonage:
- Duplication des segments de texte, de données, de piles et de la plupart des attributs système
- Les deux processus exécutent le même programme, mais indépendamment



```
#include <sys/types.h>
#include <unistd.h>
pid_t fork();
```

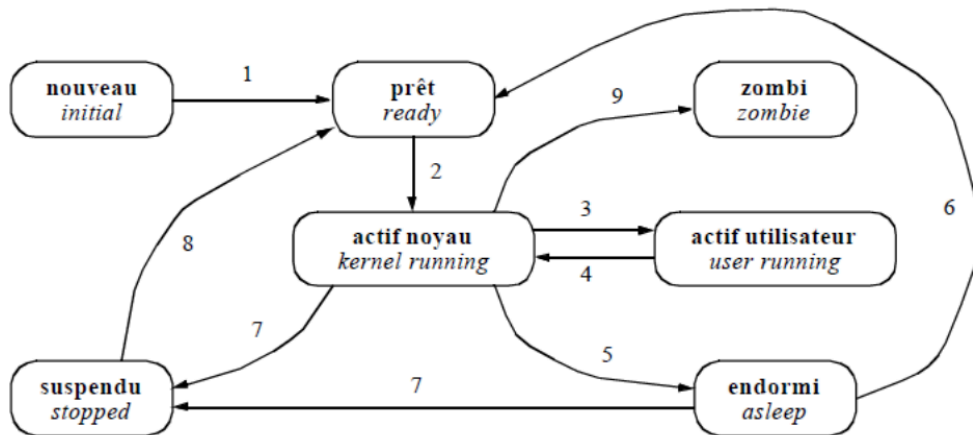
- `fork()` est donc appelé une fois mais a deux retours
 - un dans le fils, avec la valeur 0
 - un dans le père avec comme valeur le pid du fils
- Héritage des attributs système
 - (descripteurs de) fichiers ouverts
 - le pointeur d'E/S est partagé entre le père et le fils
 - uid, gid, répertoire courant, terminal de contrôle, masque de création, état des signaux, etc.
-

voir exemple - test-fork.c

et d'autres exemples ici

https://www.geeksforgeeks.org/fork-system-call/?ref=header_search

Etats d'un processus



1. le processus a acquis les ressources nécessaires à son execution
2. le processus vient d'être élu par l'ordonnanceur : il y a chargement de contexte
3. le processus revient d'un appel système ou d'une interruption (par exemple il vient d'être élu ou il a terminé l'execution du handler d'une interruption)
4. le processus a réalisé un appel système ou une interruption est survenue (peripherique ou horloge))
5. le processus se met en attente d'un evenement : appel system boqué par une evenement interne ou système (liberation de ressources ou terminaison d'un processus par exemple) ou exterieur (interruption)
6. l'evenement attendu par le processus s'est produit
7. delivrance d'un signal particulier (SIGSTOP, SIGTSTP)
8. reveil du processus par le signal SIGCONT
9. le processus se termine
- 10.

5.2. Association programme processus exec

- Le pid du processus n'a pas changé
 - c'est le même processus
- Le code a changé
 - il exécute un autre programme
 - ce programme démarre au début (main())
- L'état de l'ancien programme est oublié
 - on ne peut revenir d'un exec réussi !
- Remplacement des segments d'un processus par ceux d'un programme pris dans leur état initial
- Conservation de la plupart des attributs système
 - (descripteurs) de fichiers ouverts avec la même valeur du pointeur d'E/S qu'avant exec()
 - uid, gid, répertoire courant, terminal de contrôle, masque de création, certains états des signaux, etc.

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

```
int execl(const char *path, const char *arg0, ..., NULL);
int execv(const char *path, char *const argv[]);
int execlp(const char *path, const char *arg0,..., NULL, char **envp);
int execve(const char *path, char *const argv[], char **envp);
int execlp(const char *path, const char *arg0,... , NULL);
int execvp(const char *path, char *const argv[]);
```

voir exemple test-exec.c

5.3. Terminaison volontaire d'un processus

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

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

- Toutes ces fonctions terminent le processus courant
 - `_exit()` et `exit()` transmettent le code de retour status au processus père
 - `abort()` produit un fichier core (signal SIGABRT)
- Terminaison normale : `exit()`
 - appelle les fonctions enregistrées par `atexit()`
 - « flushe » tous les fichiers de `stdio`
 - détruit les fichiers temporaire (`tempfile()`)
 - appelle `_exit()`
- Terminaison forcée : `_exit()`
 - ferme tous les fichiers et répertoires
 - réveille le processus père (si nécessaire)
 - provoque éventuellement l'adoption du processus, etc.

5.4. Attente d'un processus fils

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

pid_t wait(int *pstatus);
pid_t waitpid(pid_t pid, int *pstatus, int options);
```

- Attente de la terminaison d'un fils
 - wait() est réveillé par la fin d'un fils quelconque
 - waitpid() est réveillé par la fin du fils indiqué
- Retour immédiat si un/le fils déjà terminé

6. Manipulation de fichiers

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

- <https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16421-manipulez-des-fichiers-a-laide-de-fonctions>
- https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/4911_les-fichiers/

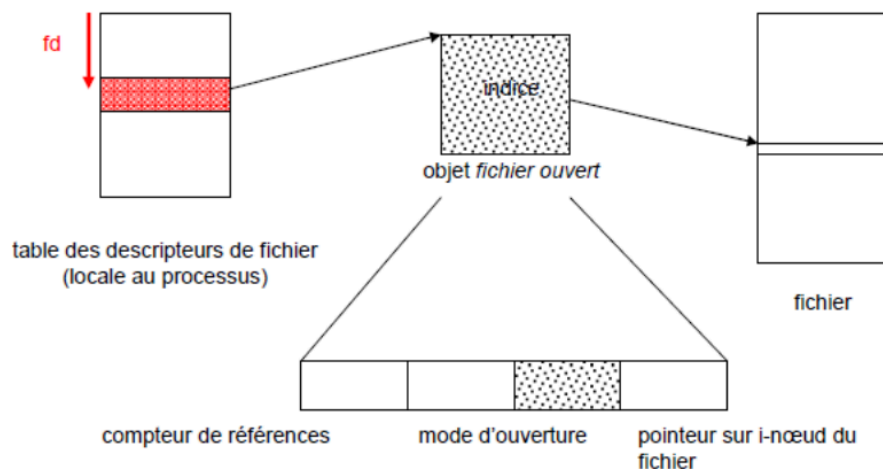
6.1. Ouvrir un fichier

```
FILE *fopen(char *chemin, char *mode);
```

Open crée un objet fichier ouvert, qui représente une instance de fichier ouvert.

open retourne un descripteur de fichier:

indice dans la table des descripteurs de fichiers du processus !



Les modes

e	Type(s) d'opération(s)	Effets
r	Lecture	Néant
r+	Lecture et écriture	Néant
w	Écriture	Si le fichier n'existe pas, il est créé. Si le fichier existe, son contenu est effacé.
w+	Lecture et écriture	<i>Idem</i>
a	Écriture	Si le fichier n'existe pas, il est créé. Place les données à la fin du fichier
a+	Lecture et écriture	<i>Idem</i>

voir exemple `open-file.c`

6.2. Fermer un fichier

```
int fclose(FILE *flux);
```

voir exemple `close-file.c`

6.3. Ecrire dans un fichier

<pre>int putc(int ch, FILE *flux); int fputc(int ch, FILE *flux); int putchar(int ch);</pre>	Les fonctions <code>putc()</code> et <code>fputc()</code> écrivent un caractère dans un flux. Il s'agit de l'opération d'écriture la plus basique sur laquelle reposent toutes les autres fonctions d'écriture. Ces deux fonctions retournent soit le caractère écrit, soit EOF si une erreur est rencontrée. La fonction <code>putchar()</code> , quant à elle, est identique aux fonctions <code>putc()</code> et <code>fputc()</code> si ce n'est qu'elle écrit dans le flux <code>stdout</code> .
<pre>int fputs(char *ligne, FILE *flux); int puts(char *ligne);</pre>	La fonction <code>fputs()</code> écrit une ligne dans le flux <code>flux</code> . La fonction retourne un nombre positif ou nul en cas de succès et EOF en cas d'erreurs. La fonction <code>puts()</code> est identique si ce n'est qu'elle ajoute automatiquement un caractère de fin de ligne et qu'elle écrit sur le flux <code>stdout</code> .
<pre>int fprintf(FILE *flux, char *format, ...);</pre>	La fonction <code>fprintf()</code> est la même que la fonction <code>printf()</code> si ce n'est qu'il est possible de lui spécifier sur quel flux écrire (au lieu de <code>stdout</code> pour <code>printf()</code>). Elle retourne le nombre de caractères écrits ou une valeur négative en cas d'échec.

6.4. Lire un fichier

<pre>int getc(FILE *flux); int fgetc(FILE *flux); int getchar(void);</pre>	Les fonctions <code>getc()</code> et <code>fgetc()</code> sont les exacts miroirs des fonctions <code>putc()</code> et <code>fputc()</code> : elles récupèrent un caractère depuis le flux fourni en argument. Il s'agit de l'opération de lecture la plus basique sur laquelle reposent toutes les autres fonctions de lecture. Ces deux fonctions retournent soit le caractère lu, soit EOF si la fin de fichier est rencontrée ou si une erreur est rencontrée. La fonction <code>getchar()</code> , quant à
--	---

	elle, est identique à ces deux fonctions si ce n'est qu'elle récupère un caractère depuis le flux stdin.
<pre>char *fgets(char *tampon, int taille, FILE *flux);</pre>	<p>La fonction fgets() lit une ligne depuis le flux flux et la stocke dans le tableau tampon. Cette dernière lit au plus un nombre de caractères égal à taille diminué de un afin de laisser la place pour le caractère nul, qui est automatiquement ajouté. Dans le cas où elle rencontre un caractère de fin de ligne : celui-ci est conservé au sein du tableau, un caractère nul est ajouté et la lecture s'arrête.</p> <p>La fonction retourne l'adresse du tableau tampon en cas de succès et un pointeur nul si la fin du fichier est atteinte ou si une erreur est survenue.</p>
<pre>int fscanf(FILE *flux, char *format, ...);</pre>	La fonction fscanf() est identique à la fonction scanf() si ce n'est qu'elle récupère les données depuis le flux fourni en argument (au lieu de stdin pour scanf()).

7. Communication inter processus (IPC)

La communication inter-processus (IPC) est essentielle dans la programmation système, car elle permet à des processus distincts de partager des informations, de coopérer et de synchroniser leurs activités. Voici quelques mécanismes courants de communication inter-processus :

- **Pipes** : Une pipe est un mécanisme de communication unidirectionnel entre des processus. Elle peut être utilisée pour relier la sortie d'un processus à l'entrée d'un autre.
- **Files d'attente (Message Queues)** :
 - Les files d'attente permettent à plusieurs processus de partager des messages à travers une file d'attente interne au noyau.
 - Chaque message a un identifiant de type qui peut être utilisé par le récepteur pour sélectionner les messages qu'il souhaite recevoir.
 - Les files d'attente sont souvent utilisées lorsque des données structurées doivent être échangées entre des processus.
- **Signaux** :
 - Les signaux sont des notifications asynchrones envoyées à un processus pour indiquer un événement ou une condition particulière.
 - Les signaux peuvent être utilisés pour des événements tels que la terminaison d'un processus, la division par zéro, la réception d'une interruption matérielle, etc.

- Les signaux sont souvent utilisés pour la communication simple et les notifications entre processus.
- **Sockets :**
 - Les sockets sont des interfaces de communication bidirectionnelle permettant la communication entre des processus s'exécutant sur des machines différentes ou sur la même machine.
 - Ils peuvent être utilisés pour communiquer entre des processus locaux (communication inter-processus) ou pour la communication sur un réseau (communication inter-machine).
 - Les sockets offrent une grande flexibilité et peuvent être utilisés pour différents types de communication, y compris TCP/IP, UDP, et plus encore.
- **Mémoire partagée :**
 - La mémoire partagée permet à plusieurs processus d'accéder à la même région de mémoire, ce qui facilite le partage de données entre eux.
 - C'est l'un des mécanismes les plus rapides pour la communication inter-processus car il évite la copie de données entre les processus.
 - Cependant, la mémoire partagée nécessite une synchronisation appropriée pour éviter les problèmes de concurrence et de cohérence des données.
- **Les sémaphores**
 - Les sémaphores sont des variables spéciales utilisées pour la synchronisation entre processus concurrents, leur permettant de coordonner leur accès à des ressources partagées.
 - Voici comment fonctionnent les sémaphores :

Sémaphore binaire :

 - Un sémaphore binaire, ou mutex (abréviation de "mutual exclusion"), ne peut avoir que deux valeurs : 0 ou 1.
 - Il est souvent utilisé pour assurer l'exclusion mutuelle entre les processus ou threads, garantissant qu'une seule entité à la fois peut accéder à une ressource partagée.

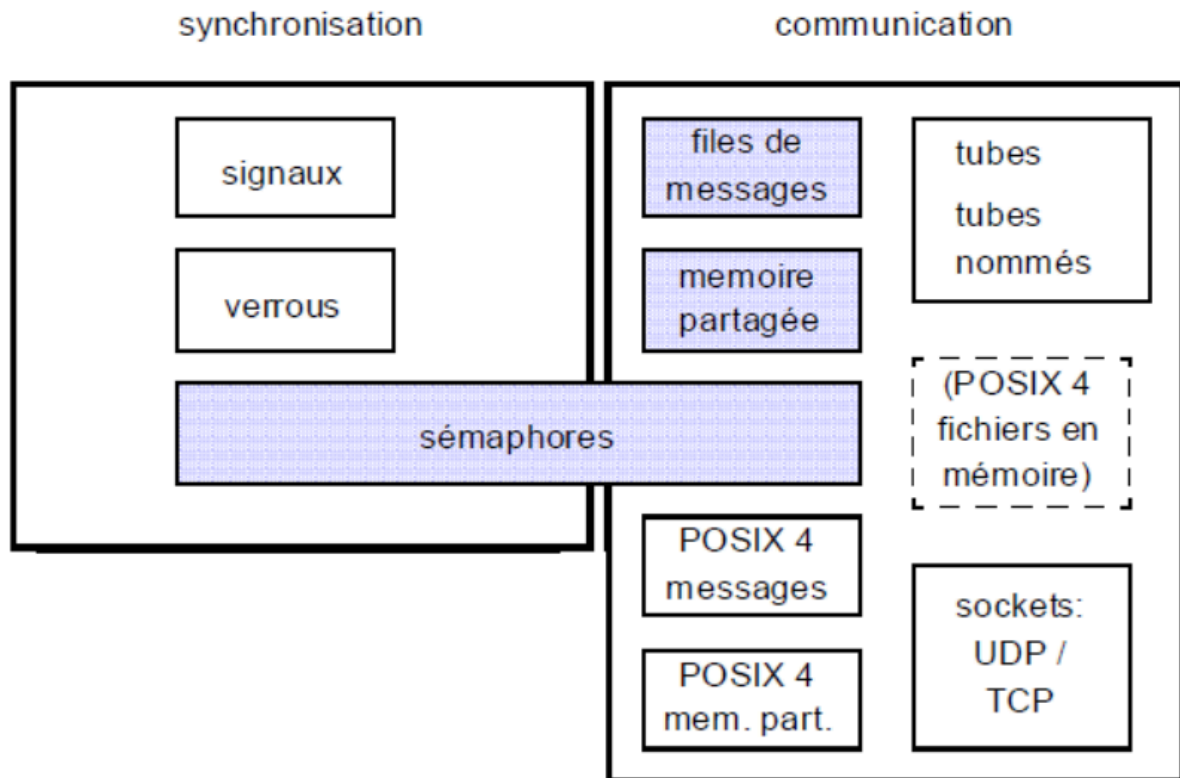
Sémaphore général :

 - Un sémaphore général peut avoir une plage de valeurs plus grande que simplement 0 ou 1. Il peut être utilisé pour contrôler l'accès à un certain nombre de ressources identiques ou limitées.
 - Les opérations courantes sur les sémaphores sont l'incrémementation (P ou wait) et la décrémentation (V ou signal).
 - L'opération P (pour prendre ou acquérir) décrémente la valeur du sémaphore et bloque le processus si la valeur devient négative, indiquant qu'aucune ressource n'est disponible.
 - L'opération V (pour libérer) incrémente la valeur du sémaphore, signalant qu'une ressource a été libérée et peut être utilisée par un autre processus.

Les sémaphores sont largement utilisés pour la synchronisation entre processus, par exemple, pour éviter les conditions de concurrence dans les sections critiques de code où plusieurs processus tentent d'accéder

simultanément à une ressource partagée. Ils sont un outil puissant pour la programmation concurrente et parallèle, contribuant à éviter les problèmes tels que les interblocages et les conditions de course.

Chacun de ces mécanismes de communication inter-processus a ses avantages et ses inconvénients, et le choix dépend souvent des besoins spécifiques de l'application, tels que la complexité des données à échanger, la vitesse requise, la sécurité, etc.



7.1. Les tubes/pipes

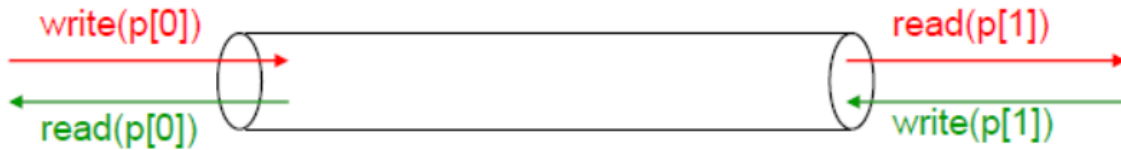
Une pipe est un mécanisme de communication unidirectionnel entre des processus.

Elle peut être utilisée pour relier la sortie d'un processus à l'entrée d'un autre.

Il existe deux types de pipes : les pipes anonymes et les pipes nommées.

- Les pipes anonymes sont créés avec la fonction `pipe()` et ne peuvent être utilisés que par des processus liés par une relation de parenté.
- Les pipes nommées, également appelées FIFO (First In, First Out), résident dans le système de fichiers et peuvent être utilisées entre des processus non apparentés.

Les pipes sont souvent utilisées pour implémenter le concept de redirection de flux dans les systèmes UNIX.



<pre># include <unistd.h> 2 i n t pipe (i n t f i l e d e s [2]) ;</pre>	creer une pipe. Retourne 0 en cas de succès, ou -1 en cas d'erreur
<pre>int read(P[x], buf, bufsize) int write(P[x], buf, bufsize)</pre>	$write(p[0]) \Leftarrow \Rightarrow read(p[1])$: l'information écrite dans p[0] est lue dans p[1] $write(p[1]) \Leftarrow \Rightarrow read(p[0])$: l'information écrite dans p[1] est lue dans p[0]

Synchronisation et cas particuliers:

- la lecture est bloquante
- le lecteur est bloqué si le tube est vide mais ouvert en écriture par une autre processus
- read retourne 0 si le tube est vide mais ouvert en écriture par aucun processus
- le signal SIGPIPE est envoyé au processus qui exécute write si le tube n'a pas été ouvert en lecture par aucun autre processus

7.2. Les signaux

7.2.1. Description

Les signaux sont des notifications asynchrones envoyées à un processus pour indiquer un événement ou une condition particulière.

Les signaux peuvent être utilisés pour des événements tels que la terminaison d'un processus, la division par zéro, la réception d'une interruption matérielle, etc.

Les signaux sont souvent utilisés pour la communication simple et les notifications entre processus.

Les signaux ne sont pas strictement considérés comme des mécanismes de communication inter-processus (IPC), bien qu'ils puissent être utilisés pour envoyer des notifications entre processus. Les signaux sont des mécanismes d'interruption asynchrones utilisés pour notifier un processus qu'un événement particulier s'est produit. Voici quelques caractéristiques des signaux :

Asynchrones : Les signaux sont asynchrones, ce qui signifie qu'ils peuvent être envoyés à tout moment par le système d'exploitation ou un autre processus, sans coordination explicite entre l'émetteur et le récepteur.

Notifications d'événements : Les signaux sont généralement utilisés pour signaler des événements tels que la terminaison d'un processus, la réception d'une interruption matérielle, la violation d'une instruction, etc.

Gestion par le système d'exploitation : La réception d'un signal déclenche généralement le système d'exploitation pour exécuter une action spécifique, comme

exécuter un gestionnaire de signal, ou même terminer le processus si aucun gestionnaire n'est défini pour le signal.

Peu fiables : Les signaux sont considérés comme peu fiables car ils peuvent être perdus s'ils sont envoyés trop fréquemment ou si le processus récepteur ne les traite pas rapidement.

Bien que les signaux ne soient pas strictement des IPC, ils peuvent être utilisés pour des opérations de communication simples et des notifications entre processus. Par exemple, un processus peut envoyer un signal à un autre processus pour l'informer de terminer son exécution, ou pour lui indiquer qu'une certaine action doit être effectuée. Cependant, leur utilisation est souvent limitée à des scénarios simples et des interactions basiques entre les processus. Pour des communications plus complexes, d'autres mécanismes IPC comme les pipes, les files d'attente ou les sockets sont généralement préférés.

Exemple 1 - envoie de signaux	
Exemple 2 - Envoie Signaux entre 2 processus	

Gestion des signaux

Exemple 3 - gestion de signaux	
--------------------------------	--

7.2.2. Les types de signaux

numéro de signal	Nom du signal	Description
1	SIGHUP	Hang-Up (fermeture du terminal ou du processus de contrôle)
2	SIGINT	Interruption (généralement généré par Ctrl+C)
3	SIGQUIT	Quitter (généralement généré par Ctrl+)
4	SIGILL	Instruction illégale

5	SIGTRAP	Trace trap (utilisé pour le débogage)
6	SIGABRT	Abandonnement (envoyé par la fonction abort())
7	SIGBUS	Erreur de bus
8	SIGFPE	Erreur en virgule flottante
9	SIGKILL	Tue (non bloquable)
10	SIGUSR1	Signal utilisateur 1
11	SIGSEGV	Violation de segmentation
12	SIGUSR2	Signal utilisateur 2
13	SIGPIPE	Pipe cassé
14	SIGALRM	Alarme
15	SIGTERM	Terminaison (signal de terminaison normal)
16	SIGSTKFLT	Stack Fault
17	SIGCHLD	Changement de statut des enfants
18	SIGCONT	Continuer (pour reprendre l'exécution d'un processus arrêté)
19	SIGSTOP	Arrêt (arrête l'exécution d'un processus)
20	SIGTSTP	Arrêt de terminal (généralement généré par Ctrl+Z)
21	SIGTTIN	Lecture en arrière-plan d'un terminal
22	SIGTTOU	Écriture en arrière-plan d'un terminal
23	SIGURG	Données urgentes disponibles sur une socket

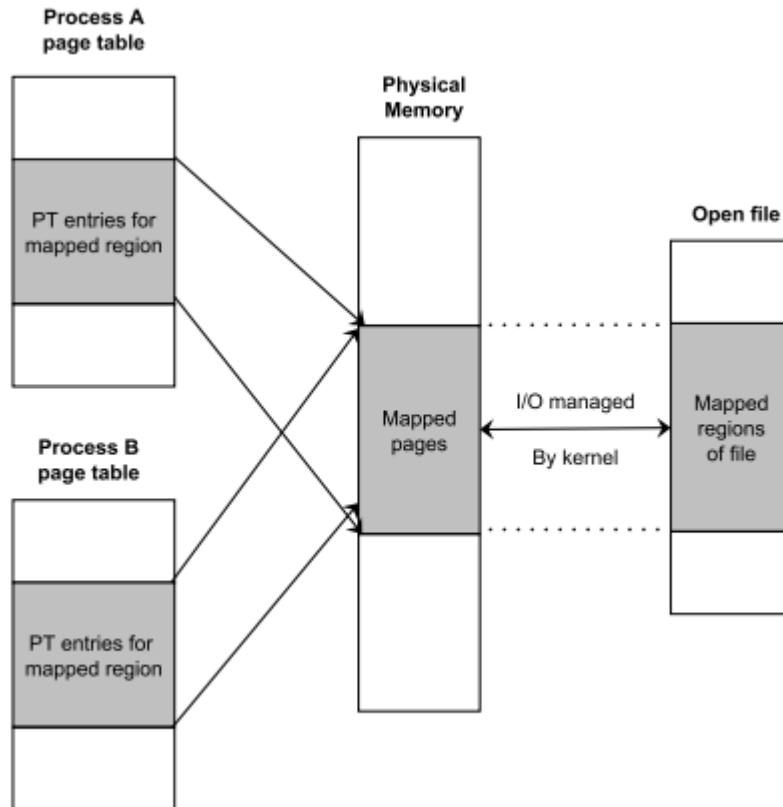
24	SIGXCPU	Temps CPU écoulé
25	SIGXFSZ	Taille maximale du fichier dépassée
26	SIGVTALRM	Alarme virtuelle
27	SIGPROF	Alarme profilée
28	SIGWINCH	Changement de taille de fenêtre
29	SIGIO	Entrée/sortie asynchrone disponible
30	SIGPWR	Événement de panne d'alimentation

7.3. Les mémoires partagées

La mémoire partagée permet à plusieurs processus d'accéder à la même région de mémoire, ce qui facilite le partage de données entre eux.

C'est l'un des mécanismes les plus rapides pour la communication inter-processus car il évite la copie de données entre les processus.

Cependant, la mémoire partagée nécessite une synchronisation appropriée pour éviter les problèmes de concurrence et de cohérence des données.



<pre>#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/ipc.h> #include <sys/shm.h></pre>	
<pre>int shmget (key_t clé, int taille, int option);</pre>	<p>Création d'un segment de mémoire partagée. crée le segment de mémoire de nom clé si celui-ci n'existe pas; s'il existe déjà, permet d'obtenir l'identificateur du segment de mémoire, que le processus utilisera par la suite pour manipuler le segment de mémoire</p>
<pre>void *shmat(int id, void *adr, int option)</pre>	<p>l'appel applique (ou attache) un segment de mémoire partagée dans l'espace d'adressage du processus • à partir de cette opération, toute lecture/écriture dans la zone dans laquelle le segment a été attaché est une lecture/écriture d'une zone de mémoire partagée</p>
<pre>int shmctl (int shmid , int cmd , struct shmid_ds * buf) ;</pre>	<p>Permet de consulter ou de modifier les caractéristiques d'un segment mémoire ainsi que de le supprimer.</p>

int shmdt (void *adr)	le détachement est l'opération inverse de l'attachement • adr: adresse d'attachement
shm_open	
mmap	
munmap	
shm_unlink	

7.4. Les files de messages

7.5. Les sémaphores

Voici le principe de fonctionnement des sémaphores :

7.5.1. Compteur :

- Un sémaphore est essentiellement un compteur entier non négatif, initialement initialisé à une valeur donnée. Ce compteur représente le nombre de ressources disponibles ou le nombre de places dans une file d'attente, selon le contexte.

7.5.2. Opérations atomiques :

- Les opérations sur les sémaphores sont atomiques, ce qui signifie qu'elles sont exécutées de manière indivisible. Cela garantit que les processus concurrents ne peuvent pas interagir de manière imprévisible avec le sémaphore.

7.5.3. Deux opérations fondamentales :

- Il existe généralement deux opérations fondamentales sur un sémaphore :
Opération P (ou down) : Cette opération décrémente le compteur du sémaphore. Si le compteur devient négatif, le processus qui a exécuté l'opération P est mis en attente.
Opération V (ou up) : Cette opération incrémente le compteur du sémaphore. Si le compteur devient positif ou nul après cette opération,

un ou plusieurs processus en attente peuvent être réveillés pour continuer leur exécution.

7.5.4. Synchronisation :

- Les sémaphores sont principalement utilisés pour synchroniser l'accès à une ressource partagée entre plusieurs processus ou threads.
- Avant d'accéder à la ressource, un processus exécute une opération P sur le sémaphore pour vérifier s'il y a des ressources disponibles. Si le compteur est supérieur à zéro, le processus peut accéder à la ressource en le décrémentant (s'il s'agit d'un sémaphore binaire) ou en effectuant d'autres actions nécessaires.
- Après avoir terminé d'utiliser la ressource, le processus exécute une opération V sur le sémaphore pour indiquer qu'il a libéré la ressource et pour permettre à d'autres processus d'y accéder.

<code>int semget (key_t clé, int nsems, int semflg</code>	Création
<code>int semop (int semid, struct sembuf *spos, int nsops</code>	Opérations
<code>: int semctl (int semid, int semno, int cmd, union semun ar</code>	Opérations de contrôle

voir l'exemple :

8. Threads

Les threads, également appelés processus légers, sont des unités d'exécution légers qui partagent le même espace d'adressage dans un processus. Ils sont gérés par le noyau du système d'exploitation et peuvent être exécutés en parallèle sur des processeurs multiples (s'ils sont disponibles). Voici le principe de fonctionnement des threads en Linux avec une illustration en langage C :

8.0.1. Principe de fonctionnement des threads en Linux :

Création de threads :

- Les threads peuvent être créés à l'intérieur d'un processus existant à l'aide de la fonction `pthread_create()`. Cette fonction prend plusieurs

arguments, y compris un pointeur vers une fonction qui sera exécutée par le thread nouvellement créé.

Partage de l'espace d'adressage :

- Les threads d'un même processus partagent le même espace d'adressage, y compris les variables globales, les variables statiques et la mémoire allouée dynamiquement. Cela signifie qu'ils peuvent accéder aux mêmes données sans avoir besoin de mécanismes de communication supplémentaires.

Planification des threads :

- Les threads sont planifiés par le noyau du système d'exploitation. La planification détermine quel thread s'exécute à quel moment et sur quel processeur. Les politiques de planification, telles que Round Robin, FIFO, et Priorités, sont utilisées pour décider de l'ordre d'exécution des threads.

Synchronisation entre threads :

- Comme les threads partagent la même mémoire, il est nécessaire de synchroniser leur accès aux données partagées pour éviter les conditions de concurrence. Pour cela, Linux fournit différents mécanismes de synchronisation tels que les verrous mutex (`pthread_mutex_t`), les variables condition (`pthread_cond_t`), les sémaphores (`sem_t`), etc.

<code>#include <pthread.h></code>	

voir les exemples

9. Gestions des périphériques

9.1. Présentation générale de Arduino et Raspberry Pi

9.1.1. Arduino :

Arduino est une plate-forme open-source de prototypage électronique qui permet aux amateurs, aux étudiants et aux professionnels de créer des projets interactifs et des dispositifs électroniques. L'idée principale derrière Arduino est de fournir un

moyen simple et abordable de contrôler des objets du monde réel à l'aide de microcontrôleurs programmables. La popularité d'Arduino repose sur sa simplicité d'utilisation, sa flexibilité et sa large communauté de développeurs.

9.1.1.1. Principales caractéristiques d'Arduino :

- Facilité de programmation : Arduino utilise un langage de programmation basé sur le langage Wiring, proche du C/C++, qui est facile à apprendre même pour les débutants.
- Environnement de développement intégré (IDE) : L'IDE Arduino fournit un éditeur de code, un compilateur et un chargeur de programme intégrés, simplifiant ainsi le processus de développement.
- Large gamme de cartes : Arduino propose une variété de cartes, des plus simples aux plus avancées, adaptées à différents types de projets et de budgets.
- Vaste communauté : La communauté Arduino est dynamique et collaborative, offrant un soutien technique, des tutoriels et des projets open-source.

9.1.2. Raspberry Pi :

Raspberry Pi est un ordinateur monocarte de la taille d'une carte de crédit, conçu pour encourager l'apprentissage de la programmation informatique et le bricolage électronique. Contrairement à Arduino, qui est principalement orienté vers le contrôle d'appareils électroniques, Raspberry Pi fonctionne comme un véritable ordinateur, capable d'exécuter un système d'exploitation complet comme Linux.

9.1.2.1. Principales caractéristiques de Raspberry Pi :

- Puissance de calcul : Raspberry Pi est équipé d'un processeur ARM, de mémoire RAM et de ports d'entrée/sortie, offrant des performances informatiques suffisantes pour diverses applications.
- Polyvalence : Raspberry Pi peut être utilisé pour une large gamme de projets, tels que les serveurs web, les médias centers, les systèmes d'automatisation domestique, les consoles de jeu rétro, etc.
- Connectivité : Raspberry Pi est doté de ports USB, HDMI, Ethernet, GPIO et de connexions sans fil (WiFi, Bluetooth), offrant ainsi de nombreuses options de connectivité.
- Prix abordable : Raspberry Pi est disponible à un prix abordable, le rendant accessible à un large public, y compris les écoles et les projets à but non lucratif.

9.1.3. Conclusion :

En résumé, Arduino et Raspberry Pi sont deux plates-formes puissantes et polyvalentes pour l'apprentissage de l'électronique et de la programmation. Alors que Arduino est idéal pour les projets nécessitant un contrôle en temps réel de dispositifs électroniques, Raspberry Pi convient mieux aux applications nécessitant un traitement informatique plus avancé et une connectivité réseau. Ensemble, ces deux plates-formes offrent un potentiel créatif immense pour les amateurs et les professionnels de l'informatique et de l'électronique.

9.2. Différences entre les deux plates-formes

Architecture matérielle :

- **Arduino** : Les cartes Arduino sont basées sur des microcontrôleurs, ce qui les rend idéales pour les applications qui nécessitent un contrôle en temps réel de dispositifs électroniques. Elles sont conçues pour être utilisées dans des systèmes embarqués et des projets interactifs.
- **Raspberry Pi** : Raspberry Pi est un ordinateur monocarte complet, équipé d'un processeur, de mémoire RAM, de ports d'entrée/sortie et de connectivité réseau. Il est conçu pour exécuter un système d'exploitation complet comme Linux, offrant ainsi plus de puissance de calcul et de capacités informatiques.

Langage de programmation :

- **Arduino** : Arduino utilise un langage de programmation basé sur le langage Wiring, proche du C/C++. Ce langage est simple à apprendre et est bien adapté pour la programmation des microcontrôleurs.
- **Raspberry Pi** : Raspberry Pi prend en charge une variété de langages de programmation, dont Python, C/C++, Java, etc. Étant un ordinateur complet, il offre une plus grande flexibilité en termes de langage de programmation.

Fonctionnalités et capacités :

- **Arduino** : Les capacités d'Arduino sont généralement limitées par la puissance de traitement du microcontrôleur, ce qui les rend plus adaptées aux projets nécessitant un contrôle en temps réel de dispositifs électroniques simples à moyennement complexes.
- **Raspberry Pi** : Raspberry Pi offre des capacités plus étendues en termes de puissance de calcul, de stockage et de connectivité. Il convient mieux aux applications nécessitant un traitement informatique plus avancé, comme la gestion de serveurs, le traitement d'images, le développement web, etc.

Connectivité et interfaces :

- **Arduino** : Les cartes Arduino sont équipées de diverses interfaces d'entrée/sortie (GPIO, analogiques, UART, I2C, SPI) pour interagir avec des capteurs, des actionneurs et d'autres périphériques électroniques.
- **Raspberry Pi** : En plus des interfaces d'entrée/sortie similaires à Arduino, Raspberry Pi dispose de ports USB, HDMI, Ethernet, audio, WiFi, Bluetooth, etc., offrant une connectivité étendue pour une utilisation avec divers périphériques et réseaux.

Coût :

- **Arduino** : Les cartes Arduino sont généralement moins chères que Raspberry Pi, ce qui les rend idéales pour les projets avec un budget limité.
- **Raspberry Pi** : Bien que plus coûteux que les cartes Arduino, Raspberry Pi offre une valeur ajoutée en tant qu'ordinateur complet avec des capacités plus avancées.
-

En résumé, Arduino est idéal pour les projets nécessitant un contrôle en temps réel de dispositifs électroniques simples à moyennement complexes, tandis que Raspberry Pi convient mieux aux applications nécessitant un traitement informatique plus avancé et une connectivité réseau.

9.3. Applications et domaines d'utilisation

9.3.1. Applications d'Arduino :

1. **Projets DIY (Do It Yourself)** : Arduino est largement utilisé par les amateurs et les bricoleurs pour créer une multitude de projets électroniques, tels que des thermostats intelligents, des robots, des stations météo, des systèmes de surveillance, etc.
2. **Domotique** : Arduino est utilisé pour automatiser et contrôler les appareils électriques et électroniques à domicile, y compris l'éclairage, le chauffage, les serrures de porte, les systèmes d'irrigation, etc.
3. **Éducation** : Arduino est largement utilisé dans les programmes éducatifs pour enseigner les concepts d'électronique, de programmation et de robotique de manière pratique et interactive.
4. **Contrôle de capteurs et d'actionneurs** : Arduino est utilisé dans de nombreux systèmes de contrôle qui impliquent l'acquisition de données à partir de capteurs (température, humidité, mouvement, etc.) et le contrôle d'actionneurs (moteurs, servomoteurs, relais, etc.).
5. **Prototypage rapide** : Grâce à sa simplicité d'utilisation et à sa flexibilité, Arduino est largement utilisé pour le prototypage rapide de produits électroniques et de systèmes embarqués.

9.3.2. Domaines d'utilisation de Raspberry Pi :

1. **Informatique embarquée** : Raspberry Pi est utilisé dans divers systèmes embarqués, tels que les appareils de surveillance, les kiosques interactifs, les systèmes de contrôle de la maison intelligente, etc.
2. **Serveurs et réseaux** : Raspberry Pi peut être utilisé comme serveur web, serveur de fichiers, serveur de médias, routeur, pare-feu, etc.
3. **Éducation** : Raspberry Pi est largement utilisé dans l'enseignement de l'informatique et de la programmation, offrant aux étudiants un accès abordable à un ordinateur complet pour l'apprentissage pratique.

4. **IoT (Internet des objets)** : Raspberry Pi est utilisé dans des projets IoT pour collecter, traiter et transmettre des données à partir de capteurs et d'appareils connectés, contribuant ainsi au développement de solutions intelligentes pour divers domaines tels que la santé, l'agriculture, l'industrie, etc.
5. **Divertissement et médias** : Raspberry Pi peut être utilisé comme centre multimédia pour diffuser des vidéos, de la musique et des jeux, grâce à des logiciels tels que Kodi, Plex, RetroPie, etc.

9.4. Arduino

voir

9.5. Raspberry Pi

voir