**Note R3.05**

**Processus**

**Prof :** On peut imaginer un processus comme un programme en cours d’exécution

**Identification par le PID**

Le premier processus du système, init, ainsi que quelques autres sont créés directement par le noyau au démarrage.

On peut les apercevoir via la commande linux ps

Une image contenant texte, table

Description générée automatiquement

Ainsi la seule manière La seule manière, ensuite, de créer un nouveau processus est d’appeler l’appel-système fork(),qui va dupliquer le processus appelant(cad le programme principal dans notre cadre). Au retour de cet appel-système, deux processus identiques continueront d’exécuter le code à la suite de fork()

La différence essentielle entre ces deux processus est le PID

On distingue ainsi le processus original, qu’on nomme traditionnellement le processus père, et la nouvelle copie, le processus fils

Pour connaître son propre identifiant PID, on utilise l’appel-système getpid() qui renvoie une valeur de type pid\_t.

Rmq : Ce numéro de PID est celui que nous avons vu affiché en première colonne de la commande ps

**1-Distinction entre processus père et fils**

Le même appel fork() retourne 0 dans le processus fils, retourne -1 en cas d’erreur et retourne le PID du processus fils dans le processus père (cad une fois dans le père fork() retourne le pid du fils)

**N.B :** Dans la plupart des applications courantes, la création d’un processus fils a pour but de faire dialoguer deux parties indépendantes du programme (à l’aide de signaux, de tubes…que nous verrons dans la suite).

C’est pourquoi, le processus fils peut aisément accéder au PID de son père (noté PPID pour Parent PID) grâce à l’appel-système getppid()

**Prof :** En cas d’erreur (par exemple due au probleme de mémoire disponible parle exemple ), fork() renvoie la valeur –1,

Prof : Voici un exemple de création d'un processus fils en utilisant l'appel système fork()

**Tp1-exercice2-b**

int main() {

pid\_t pid = fork();

if (pid == 0) {

//fils

for (int i = 0; i < 10; i++)

{

printf("Je suis le processus fils et j'affiche %d\n"2 \* i);

}

}else{

if (pid == -1) {

printf("Error");

exit(1);

}

else{//Pere

// Affiche les 10 premiers nombres impairs

for (int i = 0; i < 10; i++){

printf("Je suis le processus pere et j'affiche %d\n"2 \* i + 1);

sleep(1); // Endors-toi pendant 1 seconde stp ;)

}

}

}

return 0;

}

**N.B :** Ainsi nous devons faire la différence entre le PID d’un processus et la valeur retourner par fork() qui est aussi un entier de type pid\_t.

Notez que le processus fils est une copie exacte du processus père, y compris son PID, sa mémoire et ses variables.

**Rappel :** Lorsque un programme lancé se finit par exit(int code\_de\_sortie) alors le programme se termine et renvoie le code de sortie au système d’exploitation.

Si code\_de sortie = 0 alors le programmeur veut indiquer au système **que la fin normale** du programme sinon pour indiquer une erreur.

exit(0) // fin normale

**2-Lancement d’un nouveau programme**

Nous avons déjà vu que le seul moyen de créer un nouveau processus dans le système est d’invoquer fork(), qui duplique le processus appelant.

**De même,** la seule façon d’exécuter un nouveau programme est d’appeler l’une des fonctions de la famille exec()

L’appel de l’une des fonctions exec() permet de remplacer l’espace mémoire du processus appelant par le code et les données de la nouvelle application

il y a six variantes nommées **execl(), execle(), execlp(), execv(), execve() et execvp().** Ces fonctions permettent de lancer **une application.**

Les fonctions dont le suffixe commencent par un **"l"** utilisent une liste d’arguments à transmettre de nombre variable, tandis que celles qui débutent par un **"v"** emploient **un tableau** à la manière du **vecteur argv [].**

Les fonctions se terminant par un **"e"** transmettent l’environnement dans un tableau envp [] explicitement passé dans les arguments de la fonction, alors que les autres utilisent **la variable globale environ.**

Les fonctions se finissant par un **"p"** utilisent la variable d’environnement PATH pour rechercher le répertoire dans lequel se situe l’application à lancer, alors **que les autres nécessitent un chemin d’accès complet.**

**Récapitulons les caractéristiques des six fonctions de la famille exec().**

**• execv()**

– tableau argv[] pour les arguments

– variable externe globale pour l’environnement

– nom d’application avec chemin d’accès complet

**• execve()**

– tableau argv[] pour les arguments

– tableau envp[] pour l’environnement

– nom d’application avec chemin d’accès complet

• **execvp()**

– tableau argv[] pour les arguments

– variable externe globale pour l’environnement

– application recherchée suivant le contenu de la variable PATH

**• execl()**

– liste d’arguments arg0, arg1, ..., NULL

– variable externe globale pour l’environnement

– nom d’application avec chemin d’accès complet

**• execle()**

– liste d’arguments arg0, arg1, ..., NULL

– tableau envp[] pour l’environnement – nom d’application avec chemin d’accès complet

• execlp() /\*Ce que nous utiliserons en tp1 exercice3\*/

– liste d’arguments arg0, arg1, ..., NULL.

– variable externe globale pour l’environnement

– application recherchée suivant le contenu de la variable PATH

**Prof :** Pour voir les parametre que prennent ces fonctions, referez vous à la doc

Prof : Ainsi les mécanismes à base de **exec()** permettent de remplacer totalement le programme en cours par un autre qui est exécutable, tandis que les fonctions system() et popen()-pclose() servent plutôt à utiliser une autre application comme sous-programme de la première

**3-Terminaison d’un processus**

Attendre la fin d’un processus fils

Un processus qui se termine passe automatiquement par un état spécial, zombie, en attendant que le processus père ait lu son code de retour (par exemple exit(0) pour dire qu’il es’est bien terminé)

**N.B :** Si le processus père ne lit pas le code de retour de son fils, celui-ci peut rester indéfiniment à l’état zombie.

Un processus zombie est un processus qui a terminé son exécution, mais qui reste présent dans la table des processus du système, car le processus père n'a pas encore été informé de sa fin . Le processus père peut être informé de la fin du processus fils **en appelant la fonction wait(),** qui permet de récupérer le statut de fin du processus fils (par l’attente de la fin de son exécution )et de libérer les ressources associées au processus fils.

**Rmq :** On peut donc comprendre qu’un processus père doit se terminer après l’exécution du processus fils.

La fonction wait() est déclarée **dans <sys/wait.h>**

**pid\_t wait (int \* status);**

**N.B :** La fonction wait() peut échouer et renvoyer –1, en plaçant l’erreur ECHILD dans errno si le processus appelant n’a pas de fils. Dans notre premier exemple, le processus père va se dédoubler en une série de fils qui se termineront de manières variées. Le processus père restera en boucle sur wait() jusqu’à ce qu’il ne reste plus de fils.

**Pour récupérer le code de retour renvoyer par wait** on fait

WEXITSTATUS(status)) ou statuts est un int

**Exp :** programme qui illustre le fonctionnement de la fonction wait()

int main() {

int status;

pid\_t pid = fork();

if (pid == 0) {

//fils

printf("Le processus fils s'endort pendant 10s\n");

sleep(10);

printf("Le processus fils se termine.\n");

exit(0);

}else{

printf("Le processus pere attend la fin du processus fils.\n");

wait(&status);

printf("Le processus a terminer avec le code de retour de %d.\n", WEXITSTATUS(status));

}

return 0;

}

Dans ce programme, le processus fils s'endort pendant 10 secondes, puis se termine avec le code de retour 0. Le processus père attend la fin du processus fils en appelant la fonction wait() et récupère le statut de fin du processus fils dans la variable status. Le processus père peut alors utiliser la macro WEXITSTATUS() pour récupérer le code de retour du processus fils.

**4-Sommeil des processus et contrôle des ressources**

**Prof :** La fonction la plus simple pour endormir temporairement un processus est sleep(), qui est déclarée ainsi dans :

**unsigned int sleep (unsigned int nb\_secondes);**

Cette fonction endort le processus pendant la durée demandée et revient ensuite

**N.B :** lors d’une invocation de la fonction sleep() si un signal se produit au bout d’un dixième de seconde par exemple, la durée renvoyée sera quand même décrémentée d’une seconde complète.

**5-Entrées-sorties simplifiées**

Les flux sont du type opaque FILE.

Lorsqu’on désire accéder à un fichier par l’intermédiaire d’un flux, on invoque la fonction fopen()

**FILE \* fp;**

**fp = fopen ("mon\_fichier.txt", "r"); //ouverture du fichier mon\_fichier en mode lecture**

**Prof :** Tout programme s’exécutant sous Linux dispose de trois flux ouverts automatiquement lors de son démarrage :

• stdin : flux d’entrée standard.

• stdout : flux de sortie standard.

• stderr : flux d’erreur standard. Ce flux sert à afficher des informations concernant le comportement du processus ou ses éventuels problèmes.

Une image contenant table

Description générée automatiquement

**Prof :**

Pour les conversions **entières (%d, %i, %u, %o, %x, %X)** **ou réelles (%f, %e, %E, %g, %G)**, on peut utiliser – en premier caractère – les indicateurs de formatage suivants :

**Rmq :** La différence principale entre ces deux fonctions est que **printf** affiche le texte sur la sortie standard, tandis que f**printf** permet d'afficher le texte sur n'importe quel flux de sortie, pas seulement sur la sortie standard.

**Exp avec fprintf :**

int main() {

FILE\* f;

f = fopen("mon\_fichier.txt", "r");

if (f == NULL) {

return 1;

}

fprintf(f, "Bonjour jj");

fclose(f);

return 0;

}

Dans cet exemple, le texte "Bonjour, monde!" est affiché dans le fichier mon\_fichier.txt, au lieu d'être affiché sur la sortie standard.

**Prof :** Il est également possible d'utiliser fprintf pour afficher du texte sur la sortie standard en passant stdout comme premier argument :

int main() {

fprintf(stdout, "Bonjour jj");

return 0;

}

**int printf (const char \* format, ...);**

**fprintf(stdout, format, ...) ;**

**6-Les entrées scanf et sscanf**

int main() {

int age;

printf("Bonjour, votre age svp ?");

scanf("%d", age);

printf("Vous avez %d\n"age);

return 0;

}

La différence principale entre ces deux fonctions est que scanf lit les données depuis l'entrée standard (stdin), tandis que sscanf lit les données depuis une chaîne de caractères fournie en argument.

**Exp avec sscanf :**

int main() {

char str[] = "Mon nom est Jean et j'ai 35 ans";

char nom[50];

int age;

sscanf(str, "Mon nom est %s et j'ai %d ans", nom, &age);

print("Votre nom est %s et vous avez %d ans.\n");

return 0;

}

**Tp1- Processus**

**Exercice 1**

**1)**

int main(void)

{

// Affiche l'identifiant du processus en cours d'exécution

printf("Je suis le processus numéro %d\n", getpid());

// Crée un nouveau processus en copiant le processus actuel

pid\_t pid = fork();

// Affiche la valeur renvoyée par fork

printf("fork m'a renvoyé la valeur %d\n", pid);

// Affiche l'identifiant du processus en cours d'exécution et de son père

printf("Je suis le processus numéro %d et mon père est le processus numéro %d\n", getpid(), getppid());

return 0;

}

**Exemple d’affichage du programme :**

**Une image contenant texte

Description générée automatiquement**

**Rmq :** Par ce affichage, grâce a la valeur retourner par le pid, on sait que le processus en cours d’exécution ici est le fils jusqu’à la fin du programme. Mais toutefois l’odre peux varier car l’exécution du processus fils et père se font en parallèle. C’est pourquoi dans la suite nous utiliserons **la condition if**

**N.B** : on ne pourra jamais savoir si on est dans le pere ou dans le fils a moins dde connaitre la valeur retourner par le fork dans la variable pid

**2)Combien d'affichages a-t-on ?**

On a trois affichages (qu’on peut bien voir à l’écran)

**3)A quoi correspondent-ils ?**

La première correspond a l’affichage d’un processus père et les deux derniers correspondent à l’affichage de processus fils

**Exercice 2**

1. **exécuter ce programme et voir comment se font les affichages.**

**Prof :** Il est important de noter que le processus fils et le processus père exécutent leur code en parallèle mais cela ne veux pas dire que l'ordre d'affichage des messages pourra varier d'une exécution à l'autre ! puisqu’ici on est dans une boucle for

1. **Ajouter maintenant une attente (sleep(1)) dans la boucle d'affichage du processus père, que se passe-t-il ?**

**(voir programme exemple page 2 )**

Si nous rajoutant sleep(1) dans la boucle d’attente du père, le prere s’endormira une seconde après chaque iteration. Pendant ce temps, le processus fils continuera à s'exécuter normalement et à afficher les nombres pairs.

1. **Enlever l'attente dans la boucle du père et la mettre dans la boucle du fils. Lancer ce programme. Que se passe-t-il ? Pourquoi ?**

le processus fils affichera les 10 premiers nombres pairs, mais il y aura une pause d'une seconde entre chaque affichage. Le processus père, quant à lui, continuera d'afficher les 10 premiers nombres impairs sans pause.

Notez que le processus père et le processus fils exécutent leur propre code de manière indépendante l'un de l'autre. Cela signifie que la pause dans le processus fils n'affectera pas l'exécution du processus père.

Une image contenant texte

Description générée automatiquement

Prof : Il est important de noter que l'ordre dans lequel les affichages du processus père et du processus fils seront exécutés dépend de la façon dont le système d'exploitation gère les processus. Il est possible que les affichages du processus fils apparaissent avant ceux du processus père, ou vice versa.

**Exercice3**

#include <sys/types.h> /\*pour utiliser pid\_t et size\_t(pour stocker tailles

de mémoire)\*/

#include <sys/wait.h> /\*Pour utililser les macros wait, WEXITSTATUS..pour attendre

ou reccuperer les information sur le status des processus\*/

int main() {

pid\_t pid = fork();

int status;/\*déclare une variable pour stocker le statut de retour

du processus fils\*/

if (pid != 0) {

wait(&status);// attend la fin du processus fils

printf("La valeur retourner par mon fils est %d\n",WEXITSTATUS(status));

/\* Le fils ne peut retourner au pere que son etat ainsi Une fois que

le processus fils a terminé, le processus père affiche la valeur de

retour du processus fils en utilisant la fonction WEXITSTATUS

\*/

else {

if (pid == -1) {

printf("Error");

exit(1);

}

// Execution de la commande unix ps

execlp("ps", "ps", NULL);/\*Si l'exécution de la commande ps réussit,

le processus fils termine normalement et retourne un statut

de retour de 0\*/

/\*Si l'exécution de la commande ps échoue, le processus

fils quitte immédiatement en retournant un statut de retour non nul.\*/

exit(EXIT\_FAILURE);// quitte le processus en cas d'erreur

}

return 0;

}

**Rappel :** Si le processus fils a terminé normalement, la valeur de retour sera 0. Si le processus fils a quitté en retournant un statut de retour non nul, la valeur de retour sera non nulle.

Rmq  et rappel : La commande Unix ps (process status) est utilisée pour afficher des informations sur les processus en cours d'exécution sur le système.

**Une image contenant texte

Description générée automatiquement**

4)

int main() {

pid\_t pid = fork();

int status;/\*déclare une variable pour stocker le statut de retour

du processus fils\*/

if (pid != 0) {

wait(&status);// attend la fin du processus fils

printf("La valeur retourner par mon fils est %d\n",WEXITSTATUS(status));

else {

if (pid == -1) {

printf("Error");

exit(1);

}

// Execution de la commande unix ps

execl("/usr/bin/ps", "ps", NULL);/\* Ici on devrais ensuite preciser le chemin

sinon il y aurait une erreur en remplacant execlp par execl \*/

exit(EXIT\_FAILURE);// quitte le processus en cas d'erreur

}

return 0;

}

Quand nous remplaçons execlp par execl dans le programme précèdent, Il y a une erreur, car il n’y a pas precision du chemin. Le programme voit cela comme un chemin incorrect ou inexistan.

Il est donc important de vérifier que le chemin du programme à exécuter est correct avant de l'utiliser avec execl.

/\*Exemple de chemin correct\*/

Une image contenant texte, horloge, écran, sombre

Description générée automatiquement

Rappel : La fonction execl est similaire à la fonction execlp, mais elle prend en compte le chemin complet du programme à exécuter au lieu de chercher le programme dans le répertoire courant ou dans les répertoires du chemin $PATH.

Ainsi Si on veut remplacer l'appel à execlp par un appel à execl dans le programme précédent, on va devoir également fournir le chemin complet du programme ps à exécuter. Par exemple :

5) /\*Les paramètres du programme sont les paramètres du main\*/

int main(int argc, char\* argv[]){}

int main(int argc, char\* argv[]) { /\*Il nous revient a modifier les paramètres du

main\*/

// On doit donc vérifier qu'au moins un paramètre a été fourni

if (argc < 2) {

fprintf(stderr, "Erreur : veuillez fournir au moins un paramètre\n");

return 1;

}

pid\_t pid = fork();

int status;

if (pid != 0) {

wait(&status);// attend la fin du processus fils

printf("La valeur retourner par mon fils est %d\n",WEXITSTATUS(status));

else {

if (pid == -1) {

printf("Error");

exit(1);

}

// Execution de la commande unix ps

/\* le premier paramètre est la commande à exécuter, les autres sont ses

arguments\*/

execvp(argv[1], argv + 1);

exit(EXIT\_FAILURE);// quitte le processus en cas d'erreur

}

return 0;

}

**Prof :** Dans ce code, la fonction execvp est utilisée pour exécuter la commande et ses arguments spécifiés en paramètre. De plus elle cherche le programme dans le répertoire courant et dans les répertoires du chemin $PATH, comme la fonction execlp.

Il est important de noter que si vous utilisez execvp et que vous fournissez un nom de commande incorrect ou inexistant, l'appel à execvp échouera et le processus fils quittera immédiatement avec un statut de retour non nul. Le processus père ne sera pas en mesure d'attendre la fin du processus fils et ne pourra pas récupérer son statut de retour

**Prof :** En résumé, en modifiant le programme pour permettre de passer la commande à exécuter par le fils et ses paramètres en paramètre au programme, vous pouvez exécuter n'importe quelle commande Unix et ses arguments de manière flexible et dynamique. Il est important de vérifier que la commande et ses arguments fournis en paramètre sont corrects avant de les utiliser avec execvp.

**Fin tp1**

**Programmation multithread (Systèmes Multiprocesseurs)**

**Introduction**

Il s’agit d’une manière différente d’aborder la conception multitâche.

**Prof :** les différents threads d’une application partagent un même espace d’adressage en ce qui concerne leurs données.

**N.B :** C’est pourquoi si chacune des opérations effectuées par un logiciel doit attendre la fin d’une opération précédente avant de pouvoir démarrer, il est totalement inutile d’essayer une approche multithread.

Les threads sont généralement utilisés pour créer des applications multitâches ou pour effectuer des tâches en arrière-plan pendant que l'application principale continue de fonctionner. En contrepartie, l’accès concurrentiel aux mêmes données nécessite une synchronisation pour éviter les interférences.

**Rmq :** Les méthodes de communication entre les threads sont alors naturellement plus simples que les communications entre processus

**Threads**

De la même manière qu’il existe le PID, pid\_t pour distinguer les différents processus, on a aussi le type pthread\_t pour distinguer les différents d’une application.

Pour créer un thread, on utilise le prototype de **pthread\_create()** :

**int pthread\_create (pthread\_t \* thread, pthread\_attr\_t \* attributs, void \* (\* fonction) (void \* argument), void \* argument);**

Le troisième argument est un **pointeur** représentant la fonction principale du nouveau thread. Celle-ci est invoquée dès la création du thread et reçoit en argument le pointeur passé en dernière position dans **pthread\_create()**

Le quatrième paramètre est toujours le paramètre se trouvant dans la fonction void \* (\* fonction) (void \* argument).C’est un argument de type void car (c’est un pointeur générique en C). Il peut pointer vers n’importe quel type de donnée.

Par exemple il peut être utiliser pour passer des entiers (void \*)2 et dans la définition de la fonction void \* (\* fonction) ) , on peut le convertir en entier (int) argument

**Exp :**

void\* thread\_fuction(void\* arg) {

printf("Threaf %d: demarrage.\n", (long)arg);

sleep(2);

printf("Threaf %d: termine.\n", (long)arg);

return NULL; // ou pthread\_exit(NULL);

}

int main() {

phtread\_t thread1, thread2;

pthread\_create(&thread1, NULL, thread\_function, (void\*)1);

pthread\_create(&thread1, NULL, thread\_function, (void\*)2);

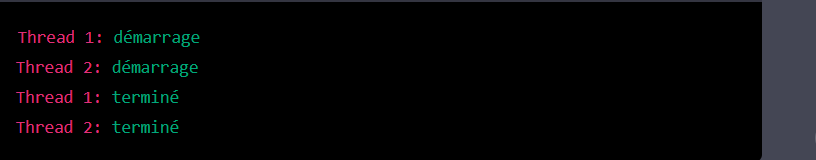
pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

Dans ce code, on crée deux threads en appelant pthread\_create() avec les arguments appropriés, puis on attend la fin de leur exécution en appelant pthread\_join(). Les threads s'exécutent de manière concurrente et leurs messages sont imprimés de manière intercalée.



N :B : l'ordre dans lequel les messages sont affichés peut varier en fonction de l'environnement d'exécution et de la façon dont les threads sont schedulés. Cet ordre peut donc être différent d'une exécution à l'autre.

**Rmq :** À la place de return NULL, on pouvait utiliser

pthread\_exit(NULL); // à la fin de cette fonction indique que le thread se termine normalement et renvoie un pointeur nul

**Prof :** En général, il est recommandé de renvoyer un pointeur nul lorsque la fonction ne doit pas renvoyer de **valeur significative.** Dans le cas contraire l’appelant récupère cette valeur en appelant pthread\_join() et en stockant le résultat dans une variable de type void \*\* retour.

**int pthread\_join (pthread\_t thread, void \*\* retour);**

**Prof:** D’une manière générale, la fonction pthread\_join() est utilisée pour attendre la fin de l'exécution d'un thread pthread\_t thread **et** récupérer une valeur de retour contenue dans void \*\* retour

**Rmq:** Nous pouvons donc remarquer que pthread\_join() et pthread\_exit() s’accompagnerons

void\* thread\_fuction(void\* arg) {

int result = 2;

pthread\_exit((void\*)resultat);

}

int main() {

phtread\_t thread;

int\* result;

pthread\_create(&thread, NULL, thread\_function,NULL);

pthread\_join(thread,(void \*\*)&result); /\* Attente de la fin d'execution du thread

et reccuperation de la valeur de result \*/

printf("Le resultat du thread est : %d\n");

return 0;

}

A l’exécution de ce programme, nous avons le message "Le résultat du thread est : 2", suivi d'un saut de ligne.

**Prof :** Lorsqu’un thread ne renvoie pas de valeur intéressante a reccuper par pthread\_join() et qu’il n’a pas besoin d’être attendu par un autre thread, on peut employer la fonction pthread\_detach(), **qui lui permet de disparaître automatiquement du système quand il se termine.**

La fonction pthread\_detach() est donc utilisée pour détacher un thread de son processus parent. Lorsqu'un thread est détaché, il n'est plus nécessaire de l'attendre pour récupérer ses ressources système lorsqu'il se termine. Au lieu de cela, ces ressources sont automatiquement libérées par le système dès que le thread se termine.

**Rmq :** Contrairement aux processus, il n’y a pas de notion de hiérarchie chez les threads ni d’autorisations particulières pour modifier les paramètres d’un autre fil d’exécution.

**N.B :** pthread\_join() échoue si le thread n’existe pas ou s’il est déjà détaché car il n'est plus possible d'utiliser la fonction pthread\_join() pour attendre sa fin d'exécution

**int pthread\_detach(pthread\_t thread);**

La fonction **pthread\_detach()** prend en paramètre l'identifiant du thread à détacher, qui est un type pthread\_t. Elle retourne 0 en cas de succès et une valeur non nulle en cas d'erreur.

Pour connaître son propre identifiant, un thread invoque la fonction **pthread\_self(),** qui lui renvoie une valeur de type pthread\_t :

**Remarque :** Le programme précèdent manipulait 1 seul thread.En voici un programme qui créer et exécuter deux threads de manière concurrente.,

**Exemple :**

void\* thread\_non\_fin\_execution(void\* arg)

{

printf("Thread 1 exécutant la fonction thread\_non\_fin\_execution\n");

sleep(3); /\*Afin de ne pas voir l'execution de la suite a l'ecran

du coup l'execution de cette fonction se fera en arriere plan\*/

printf("Thread 1 terminé\n");

printf("Pourquoi as tu attendu la fin de mon execution\n");

return NULL;

}

void\* thread\_fin\_execution(void\* arg)

{

printf("Thread 2 exécutant la fonction thread\_fin\_execution\n");

sleep(1);

printf("Thread 2 terminé\n");

printf("Merci d'avoir attendu la fin de mon execution\n");

return NULL;

}

int main(int argc, char\* argv[])

{

pthread\_t thread1, thread2;

int status;

status = pthread\_create(&thread1, NULL, thread\_non\_fin\_execution, NULL);

if (status1 != 0) {

perror("Erreur lors de la création du thread 1");

exit(EXIT\_FAILURE);

}

status = pthread\_create(&thread2, NULL, thread\_fin\_execution, NULL);

if (status != 0) {

perror("Erreur lors de la création du thread 2");

exit(EXIT\_FAILURE);

}

status = pthread\_detach(thread1);

if (status != 0) {

perror("Erreur lors de la détachement du thread 1");

exit(EXIT\_FAILURE);

}

printf("Thread 1 détaché\n");

status = pthread\_join(thread2, NULL);

if (status != 0) {

perror("Erreur lors de l'attente de la fin du thread 2");

exit(EXIT\_FAILURE);

}

return 0;

}

**Exécution à l’écran :**

Une image contenant texte

Description générée automatiquementL’utilisation de la fonction pthread\_join() sur le deuxième thread oblige le programme principal à attendre la fin de l’exécution du deuxième thread (donc de la fonction thread\_func2 ) avant de continuer

Mais toutefois le premier thread continuera d'exécuter la fonction thread\_func1 en arrière-plan.

Si le premier thread met plus de temps à terminer son exécution (sleep(3))que le deuxième thread, On le fait pour pouvoir voir comment le programme gère l'exécution des threads en parallèle et comment il gère la fin d'exécution des threads.

**Prof :** Le premier thread continue de s'exécuter en arrière-plan après que le thread principal l'a détaché, de sorte que la sortie de ce thread peut apparaître après la fin de l'exécution du thread principal. Par exemple, la sortie peut également inclure les lignes suivantes :

Une image contenant texte

Description générée automatiquement

Prof : Gardez à l'esprit que la sortie exacte peut varier en fonction de la synchronisation exacte des threads.

**Tp2-Programmation multi Threads**

**Exercice1**

1)

#include <pthread.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

void\* function\_thread1() {

while(true){

printf(".\n");

sleep(1);

}

return NULL;

}

void\* function\_thread2() {

printf("tapez un caractere\n");

char c= getchar();

return NULL;

}

int main() {

pthread\_t thread1; // Declaration du thread1

pthread\_t thread2; // Declaration du thread2

pthread\_create(thread1, NULLL,function\_thread1 , NULL);

pthread\_create(thread2, NULLL, function\_thread2, NULL);

printf("Les deux threads sont lances");

return 0;

}

Ce programme ne fonctionne pas parce qu’il se termine dès la création des deux threads sans attendre la fin de l’exécution d’un des threads.

Le second thread fait une boucle infinie donc ne se termine pas et se plante

2)

void\* function\_thread1() {

while(true){

printf(".\n");

sleep(1);

}

return NULL;

}

void\* function\_thread2() {

printf("tapez un caractere\n");

char c= getchar();

return NULL;

}

int main() {

pthread\_t thread1; // Declaration du thread1

pthread\_t thread2; // Declaration du thread2

pthread\_create(thread1, NULLL,function\_thread1 , NULL);

pthread\_create(thread2, NULLL, function\_thread2, NULL);

printf("Les deux threads sont lances");

pthread\_join(thread2,NULL); // Attente de la fin de l'execution du second thread

printf("On va s'arreter la \n");

return 0;

}

Si on attend la terminaison du second thread par pthread\_join(), le programme, dès lors va attendre la fin de l’exécution du thread2 avant de se terminer pour afficher « On va s’arrêter là  »

C’est donc l'ordre d'exécution ou de fin d’exécution des threads qui a changé. Au lieu de s'exécuter en parallèle, les threads s'exécutent maintenant l'un après l'autre, avec le second thread qui s'exécute en premier. Cela signifie que le message "On va s'arrêter là" ne sera affiché qu'après que le second thread ait terminé.

3)

void\* function\_thread1() {

while(true){

printf(".\n");

sleep(1);

}

return NULL;

}

void\* function\_thread2() {

printf("tapez un caractere\n");

char c= getchar();

return NULL;

}

int main() {

pthread\_t thread1; // Declaration du thread1

pthread\_t thread2; // Declaration du thread2

pthread\_create(thread1, NULLL,function\_thread1 , NULL);

pthread\_create(thread2, NULLL, function\_thread2, NULL);

printf("Les deux threads sont lances");

pthread\_join(thread1,NULL);

printf("On va s'arreter la \n");

return 0;

}

Dans ce cas,le programme attendra la fin du premier thread avant de ce terminer. Cela signifie que le caractère c ne sera lu par le programme qu'après que le premier thread ait terminé. Ce qui est impossible à cause de la boucle infinie

4)

Pour utiliser pthread\_attr\_setdetachstate il faut au préalablement initialiser dynamiquement les threads

void\* thread\_function1(void\* arg)

{

while (1) // boucle infinie

{

printf(".\n");

sleep(1); // pause de 1 seconde

}

return NULL;

}

void\* thread\_function2(void\* arg)

{

printf("Tapez un caractère : ");

char c = getchar(); // attend un caractère au clavier

printf("Caractère reçu : %c\n", c);

return NULL;

}

int main(void)

{

pthread\_t thread1, thread2;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);// initialisation des attributs

// préparation des attributs pour créer les threads détachés

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

// création des deux threads

pthread\_create(&thread1, &attr, thread\_function1, NULL);

pthread\_create(&thread2, &attr, thread\_function2, NULL);

printf("Les 2 threads sont lancés\n");

pthread\_join(thread2, NULL);

printf("On va s'arrêter là\n");

return 0;

}

Prof : Ce programme ne marche plus parce qu’il n’attend plus la fin de l’exécution des deux threads à cause du fait que les attributs initialiser son dans l’etat détaché. Donc pthread\_join() ne marche plus

**Sémaphore de type MUTEX**

L’attribut pthread\_attr\_t est une structure utilisée pour définir les attributs d'un thread.

Lorsque les attributs par défaut sont suffisants, on passe généralement un pointeur NULL sur cet argument. Sinon, il faut passer un pointeur sur un objet pthread\_attr\_t qui aura été préalablement configuré correctement.

L’objet de type pthread\_attr\_t est alors prêt à être utilisé dans pthread\_create().

**Exp :**

int main(int argc, char\* argv[]) {

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

//Modifie les attributs du thread ici

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED);

pthread\_attr\_set(&thread, &attr, thread\_function, NULL);

// Creation du thread

pthread\_t thread;

pthread\_create(&thread,&attr, thread\_function, NULL);

// Netoyage

pthread\_attr\_destroy(&attr);

// Attendre la fin du thread

pthread\_join(thread, NULL);

return 0;

}

**L’attribut detachstate** permet de le modifier de manière dynamique l’attribut du thread.

**int pthread\_attr\_getdetachstate (const pthread\_attr\_t \* attributs, int \* valeur); int pthread\_attr\_setdetachstate (pthread\_attr\_t \* attributs, int valeur);**

La valeur de la constante entière peut-être :

**PTHREAD\_CREATE\_JOINABLE :** Configuration par défaut, la valeur de retour du thread sera conservée jusqu’à ce qu’elle soit consultée par un autre thread

**PTHREAD\_CREATE\_DETACHED** : Lors de la terminaison du thread, toutes ses ressources sont libérées immédiatement, il n’y a pas de valeur de retour valide.

**Zones d’exclusions mutuelles**

L’un des enjeux essentiels lors du développement d’applications multithreads est la synchronisation entre les différents fils d’exécution concurrents.

Il existe essentiellement deux cas où des données risquent d’être corrompues si l’accès aux ressources communes n’est pas synchronisé :

Il y a un risque de race condition lorsque deux threads concurrents veulent modifier une variable globale. Une race condition se produit lorsque deux threads accèdent à la même variable et que l'un d'entre eux modifie cette variable avant que l'autre n'ait eu le temps de la lire. Cela peut entraîner des comportements indésirables et des résultats inattendus dans le programme.

Pour éviter ce genre de problème, il est recommandé **de synchroniser** l'accès à la variable globale. Il existe plusieurs méthodes pour synchroniser l'accès à une variable partagée, comme **l'utilisation de verrous (mutex)** ou de sections critiques (critical sections).

**On s’intéresse ici au mécanisme du mutex**

Ce principe repose sur des données appelées mutex, de type pthread\_mutex\_t

Il existe deux états pour un mutex : **disponible ou verrouillé**

Un mutex ne peut être tenu que par un seul thread à la fois. En conséquence, il existe essentiellement deux fonctions de manipulation des mutex : une fonction de verrouillage et une fonction de libération

Lorsqu’un thread demande à verrouiller un mutex déjà maintenu par un autre thread, le premier est bloqué jusqu’à ce que le mutex soit libéré.

On peut initialiser un mutex de **manière statique ou dynamique**, en précisant certains attributs à l’aide d’un objet de type **phtread\_mutexattr\_t.** L’initialisation statique se fait à l’aide de la constante **PTHREAD\_MUTEX\_INITIALIZER :**

Pour l’initialisation dynamique, on emploie **pthread\_mutex\_init()** avec une variable regroupant les attributs du mutex.

**int pthread\_mutex\_init (pthread\_mutex\_t \* mutex, const pthread\_mutexattr\_t \* attributs);**

**Recapitulation:**

**1-Déclarer et initialiser un mutex.**

**2-Lorsqu'un thread souhaite accéder à la ressource partagée, il doit acquérir le verrou en appelant la fonction de verrouillage du mutex.**

**3-Lorsque le thread a terminé d'utiliser la ressource, il doit libérer le verrou en appelant la fonction de déverrouillage du mutex.**

**Exp :** Exemple d**e** code qui utilise un mutex pour synchroniser l'accès à une variable partagée

**Initialisation dynamique du mutex**

pthread\_mutex\_t mutex;

int shared\_variable;

void\* thread\_function(void\* arg) {

pthread\_mutex\_lock(&mutex);

shared\_variable´++;

pthread\_mutex\_unlock(&mutex);

}

int main(int argc, char\* argv[]) {

pthread\_mutex\_init(&mutex, NULL);

pthread\_t thread1, thread2;

pthread\_create(&thread1,NULL, thread\_function, NULL);

pthread\_create(&thread1,NULL, thread\_function, NULL);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

pthread\_mutex\_destroy(&mutex);

return 0;

}

Dans cet exemple, le mutex est utilisé pour synchroniser l'accès à la variable partagée shared\_variable. Chaque thread appelle la fonction pthread\_mutex\_lock avant de modifier la variable et appelle la fonction pthread\_mutex\_unlock après avoir terminé. Ainsi, seul un thread à la fois peut accéder à la variable, ce qui empêche les race conditions.

**Initialisation statique du mutex**

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

int shared\_variable;

void\* thread\_function(void\* arg) {

pthread\_mutex\_lock(&mutex);

shared\_variable´++;

pthread\_mutex\_unlock(&mutex);

}

int main(int argc, char\* argv[]) {

pthread\_t thread1, thread2;

pthread\_create(&thread1, NULL, thread\_function, NULL);

pthread\_create(&thread1, NULL, thread\_function, NULL);

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

return 0;

}

**Prof :** La constante PTHREAD\_MUTEX\_INITIALIZER est utilisée pour initialiser un mutex de manière qu'il soit prêt à être utilisé dès le démarrage du programme.

**Prof :** A la différence de l’initialisation dynamique, lorsque nous utilisons l’initialisation statique alors la variable mutex **est automatiquement détruite** lorsque le programme se termine. Donc pas besoin d’appeler la fonction **pthread\_mutex\_destroy** pour détruire le mutex à la fin de son utilisation

**Prof :** Notez que lorsqu’un processus doit attendre le déblocage du premier mutex disponible dans un ensemble, ou s’il doit patienter jusqu’à ce qu’un événement survienne dans un autre thread, on emploie une autre technique de synchronisation. Il existe des variables « conditions » représentées par le type pthread\_cond\_t.

***Nous n’aborderons pas son ici***

**Tp3 – Utilisation de Sémaphore Mutex**

**Exercice1**

1)

unsigned long cpt = 0;

void\* compter\_function(void\* arg) {

for (int i = 0; i < 10000000; i++)

{

// mise à jour de la variable cpt

int tmp = cpt;

tmp++;

cpt = tmp;

}

printf("%lu", cpt);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

// création des threads

pthread\_create(&thread1, NULL, compter\_function, NULL);

pthread\_create(&thread2, NULL, compter\_function, NULL);

// attente de la fin des threads

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// affichage de la valeur finale du compteur

printf("Valeur finale du compteur %lu\n", cpt);

return 0;

}

Puisque les deux threads accèdent a la même variable compteur, la mise a jour de cette variable cpt se fait a la fois par les deux threads. Ce qui peut entrainer des modifications et les résultats aléatoires de la variables cpt.

Donc la variable cpt ne peut être 20 millions parce que les deux threads accèdent à la même variable cpt. Donc on peut avoir double incrémentation et une mise à jour au lieu d’une seule incrémentation

2)

**Etape1 :**

Initialisation statique

unsigned long cpt = 0;

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER ; // mutex pour protéger la variable cpt

void\* compter\_function(void\* arg) {

for (int i = 0; i < 10000000; i++)

{

// verrouillage du mutex pour proterger les 3 instructions

pthread\_mutex\_lock(&mutex);

int tmp = cpt;

tmp++;

cpt = tmp;

// déverrouillage du mutex

pthread\_mutex\_unlock(&mutex);

}

printf("%lu", cpt);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

// initialisation du mutex

pthread\_mutex\_init(&mutex, NULL);

// Création des threads

pthread\_create(&thread1, NULL, compter\_function, NULL);

pthread\_create(&thread2, NULL, compter\_function, NULL);

// Attente de la fin des threads

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// Affichage de la valeur finale du compteur

printf("Valeur finale du compteur %lu\n", cpt);

return 0;

}

Rappel : En utilisant un mutex, vous garantissez que seul un thread à la fois peut mettre à jour la variable cpt, ce qui évite les problèmes de perte de données et de valeur incorrecte de la variable.

**Initialisation dynamique**

unsigned long cpt = 0;

pthread\_mutex\_t mutex; // mutex pour protéger la variable cpt

void\* compter\_function(void\* arg) {

for (int i = 0; i < 10000000; i++)

{

// verrouillage du mutex

pthread\_mutex\_lock(&mutex);

// mise à jour de la variable cpt

int tmp = cpt;

tmp++;

cpt = tmp;

// déverrouillage du mutex

pthread\_mutex\_unlock(&mutex);

}

printf("%lu", cpt);

return NULL;

}

int main() {

pthread\_t thread1, thread2;

// initialisation du mutex

pthread\_mutex\_init(&mutex, NULL);

// création des threads

pthread\_create(&thread1, NULL, compter\_function, NULL);

pthread\_create(&thread2, NULL, compter\_function, NULL);

// attente de la fin des threads

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// affichage de la valeur finale du compteur

printf("Valeur finale du compteur %lu\n", cpt);

// destruction du mutex

pthread\_mutex\_destroy(&mutex);

return 0;

}

**Prof :** On vérifie que la valeur finale est bien 20.000.000

Pour vérifier on peut procéder utiliser une boucle do while dans le main qui englobe tout le programme du main

Do{// Code du main}while(cpt !=0) ;

**Communications classiques entre processus**

**Introduction**

Nous examinerons le moyen de communication le plus simple pour deux processus issus de la même application **(père et fils, ou frères)** : les tubes.

**Prof :** Nous nous limitons pour le moment aux communications entre deux processus résidant dans le **même système**

**Les tubes**

Un tube de communication est un mécanisme de communication utiliser **par les processus** pour échanger des données.

Ce sont des fichiers spéciaux qui peuvent être utilisés comme des descripteurs de fichier, ce qui signifie qu'ils peuvent **être lus et écrits** comme n'importe quel autre fichier. On peut imaginer par exemple un tuyau dans lequel un processus écrit des données qu’un autre processus peut lire.

Le tube est créé par l’appel-système pipe(), déclaré dans :

**int pipe (int descripteur [2]);**

**Descripteur de fichier**

Un descripteur de fichier est un entier positif qui représente un fichier ouvert sur un système d'exploitation comme Linux.

Chaque fois qu'un fichier est ouvert par un processus en utilisant la fonction d'ouverture de fichier, le système d'exploitation renvoie un descripteur de fichier qui peut être utilisé par le processus pour lire ou écrire dans le fichier

Les descripteurs de fichier sont généralement utilisés dans les appels de système qui permettent de lire et d'écrire des données dans les fichiers, **comme read() et write(),** ainsi que dans les appels de système qui permettent de déplacer le curseur de lecture/écriture dans un fichier, **comme lseek().**

Les descripteurs correspondent respectivement **à la sortie et à l’entrée du tube.**

Le descripteur d’indice 0 est la sortie du tube ouverte en lecture seule. Le descripteur 1 est l’entrée ouverte en écriture seule

**Tube Anonyme**

Les tubes anonymes : ce sont des tubes qui ne sont pas associés à un fichier sur le système de fichiers contrairement aux tubes nommés. Ils sont créés à l'aide de la **fonction pipe()** et sont généralement utilisés pour la communication **entre processus fils et processus père.**

Exemple d’utilisation de tube (Voir dossier tp4)

/\*exemple complet de code qui utilise un tube de processus pour lire

les résultats de la commande "ls -l" dans le processus père\*/

int main() {

//1-Creation d'un tube avec la fonction pipe()

int pipedf[2];//Creation du tableau de descripteur de fichier

int ret = pipe(pipedf); // Creation du tube anonyme

if (ret == -1) {

perror("pipe");

exit(EXIT\_FAILLURE);

}

//2- Creation d'un processus fils

pip\_t pid = fork();

if (pid ==-1) {

perror("fork");

exit(EXIT\_FAILLURE);

}

printf("Descriteur de fichier de lecture pipedf[0] ouvert (Sortie)--ok\n")

printf("Descriteur de fichier d'ecriture pipedf[1] ouvert(entree)--ok\n");

printf("Duplication du processus--ok\n")

/\*3-Pour assurer la communication, on ferme le descripteur de fichier de lecture

( de la sortie du tube pipedf[0] ) dans le processus fils\*/

if (pip == 0) {

close(pipe[0]);// fermeture du descripteur de fichier de lecture

dup2(pipe[1], STDOUT\_FILENO); // redirection de la sortie standard vers le descripteur de fichier d'écriture

execlp("ls", "ls", "-l", NULL); // exécution de la commande "ls -l"

}

/\*4- Dans le processus père, on utilise la fonction close() pour fermer le

descripteur de fichier d'écriture du tube (pipefd[1]) et utiliser la

fonction read() pour lire les données envoyées par le processus fils\*/

else {

close(pipedf[1]); // fermeture du descripteur de fichier d'écriture

char buffer[1024];

ssize\_t nbytes;

while ((nbytes = read(pipefd[0], buffer, sizeof(buffer))) > 0) {

write(STDOUT\_FILENO, buffer, nbytes);

}

}

/\*5-Utiliser la fonction close() pour fermer les descripteurs de fichier

du tube dans les deux processus :

close(pipefd[0]);close(pipefd[1]);\*/

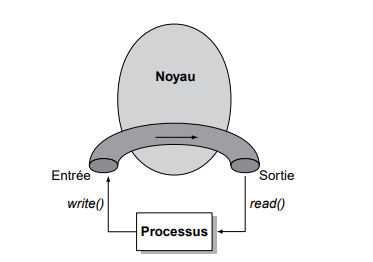
close(pipefd[0]);

close(pipefd[1]);

return 0;

}

**Rmq :** **Les tubes sont des systèmes de communication unidirectionnels. Si on désire obtenir une communication complète entre deux processus, il faut créer deux tubes et les employer dans des sens opposés.** (Cas du dernier exercice du td3)

****

**Prof :** On peut considérer un exemple de code où nous allons créer un tube, écrire des données dedans, lire son contenu et vérifier que les informations sont identiques

**Exp :** **On desire par exemple que le processus fils écrit le message "Bonjour, je suis le processus fils !" dans le tube et le processus père lit le contenu du tube et l'affiche à l'écran. Vous pouvez vérifier que les informations sont identiques en exécutant ce code.**

int main(void)

{

int pipefd[2]; // tableau de descripteurs de fichier pour le tube

int ret = pipe(pipefd); // création d'un tube anonyme

if (ret == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) { // processus fils

close(pipefd[0]); // fermeture du descripteur de fichier de lecture

const char\* message = "Bonjour, je suis le processus fils !";

write(pipefd[1], message, strlen(message)); // écriture du message dans le tube

close(pipefd[1]); // fermeture du descripteur de fichier d'écriture

}

else { // processus père

close(pipefd[1]); // fermeture du descripteur de fichier d'écriture

char buffer[1024];

ssize\_t nbytes = read(pipefd[0], buffer, sizeof(buffer)); // lecture du contenu du tube

buffer[nbytes] = '\0'; // ajout d'un caractère de fin de chaîne

printf("Message reçu dans le processus père : %s\n", buffer);

close(pipefd[0]); // fermeture du descripteur de fichier de lecture

}

return 0;

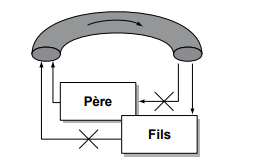
}

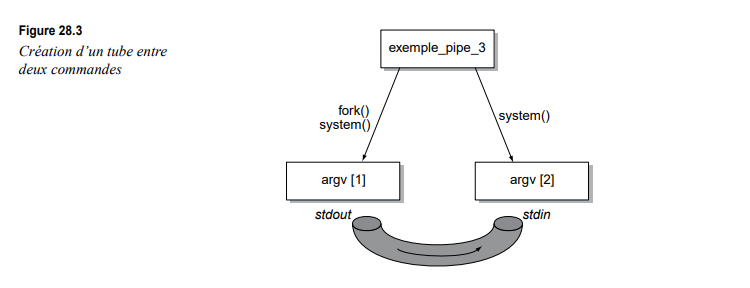
**Rmq :** Pour faire communiquer deux processus (ou plus), nous nous somme ramener à invoquer l’appel-système fork() après avoir créé le tube. Si celui-ci doit aller du processus père vers le fils, le père ferme son descripteur de sortie de tube, et le fils son descripteur d’entrée. Pourquoi ?

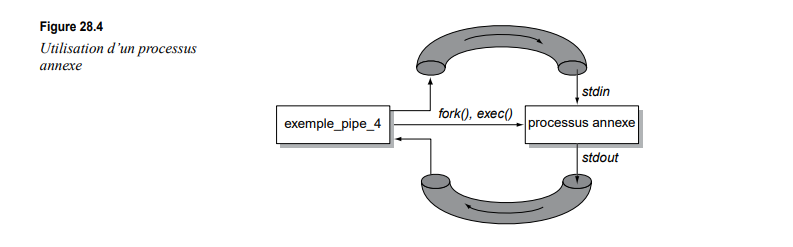
La fermeture du descripteur de fichier est nécessitée afin d’éviter un blocage de communication

Si le descripteur de fichier de lecture ou d'écriture du tube n'est pas fermé dans l'un des deux processus, la communication ne peut pas se terminer et les processus restent bloqués indéfiniment.

Dans notre code précèdent, le processus fils écrit un message dans le tube puis ferme le descripteur de fichier d'écriture **(pipefd[1]).** Le processus père lit le contenu du tube puis ferme le descripteur de fichier de lecture **(pipefd[0]).** **Cela permet à la communication entre les deux processus de se terminer et à chacun d'eux de continuer à s'exécuter.**







**Rmq :** En combinant **les tubes et les signaux**, on peut très bien parvenir à une bonne communication **entre les processus**

**Tp4-Processus et tube**

Exercice1

#define BUFFER\_SIZE 100

int main(int argc, char\* argv[]) {

int pipedf[2];

int ret = pipe(pipedf);

if (ret ==-1) {

//erreur

perror("Erreur lors de la creation du tube");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork();

if (pid == -1) {

//erreur

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

//fils

//Fermeture du descripteur d'ecriture 1

close(pipedf[1]);

// Lecture du tube jusqu'à la fin de la transmission

char buffer[BUFFER\_SIZE]; // tampon pour la lecture et l'écriture dans le tube

while (read(pipefd[0], buffer, BUFFER\_SIZE) > 0) {

// Mise en majuscule des caractères lus

for (int i = 0; i < BUFFER\_SIZE; i++) {

buffer[i] = toupper(buffer[i]);

}

// Affichage des caractères mis en majuscule

printf("%s", buffer);

}

//Fermeture du descripteur de lecture 0

close(pipedf[0]);

exit(EXIT\_SUCCESS);

}

else {

//pere

// Fermeture du descripteur de lecture 0

close(pipedf[0]);

// Ecriture dans le tube

char c;

// Tant qu'un caractere est recu du clavier et qu'il est different de $

while ((c = getchar() != '$'))

{

// Ecriture du caractere dans le tube

write(pipefd[1], &c, 1);

}

//Envoi du caractère '$' au fils pour lui signaler la fin de la transmission

write(pipefd[1], &c, 1);

//fermeture du descripteur d'ecriture 1

close(pipedf[1]);

//Attente du fils (important !)

wait(NULL);

exit(EXIT\_SUCCESS);

}

return 0;

**Exercice2**

#define BUFFER\_SIZE 100

int main(int argc, char\* argv[]) {

int pipedf[2];

int ret = pipe(pipedf);

if (ret ==-1) {

//erreur

perror("Erreur lors de la creation du tube");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork();

if (pid == -1) {

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

close(pipedf[1]);

char buffer[BUFFER\_SIZE];

/\*Si read retourne 0, cela signifie que le tube a été fermé par

l'autre extrémité, et donc que le processus père a terminé

l'envoi de données.\*/

while (read(pipefd[0], &c, 1) > 0) {

// Mise en majuscule des caractères lus

c = toupper(c);

printf("%s",c);

}

close(pipedf[0]);

exit(EXIT\_SUCCESS);

}

else {

close(pipedf[0]);

// Ecriture dans le tube

char c;

while ((c = getchar() != '$'))

{

// Ecriture du caractere dans le tube

write(pipefd[1], &c, 1);

}

write(pipefd[1], &c, 1);

close(pipedf[1]);

//Attente du fils (important !)

wait(NULL);

exit(EXIT\_SUCCESS);

}

return 0;

}

**Exercice3**

#define BUFFER\_SIZE 100

int main(int argc, char\* argv[]) {

int pipedf[2];

int ret = pipe(pipedf);

if (ret ==-1) {

//erreur

perror("Erreur lors de la creation du tube");

exit(EXIT\_FAILURE);

}

pid\_t pid = fork();

if (pid == -1) {

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Processus fils

// Fermeture de l'extrémité d'écriture du tube (inutile pour le fils)

close(pipefd[1]);

// Lecture du tube jusqu'à la fin de la transmission

while (read(pipefd[0], buffer, BUFFER\_SIZE) > 0) {

// Mise en majuscule des caractères lus

for (int i = 0; i < BUFFER\_SIZE; i++) {

buffer[i] = toupper(buffer[i]);

}

// Affichage des caractères mis en majuscule

printf("%s", buffer);

}

// Fermeture de l'extrémité de lecture du tube (important !)

close(pipefd[0]);

exit(EXIT\_SUCCESS);

}

else {

// Processus père

// Fermeture de l'extrémité de lecture du tube (inutile pour le père)

close(pipefd[0]);

// Lecture des caractères au clavier et écriture dans le tube

char c;

while ((c = getchar()) != EOF) {

write(pipefd[1], &c, 1);

}

// Envoi du signal EOF au fils pour lui signaler la fin de la transmission

write(pipefd[1], &c, 1);

// Fermeture de l'extrémité d'écriture du tube (important !)

close(pipefd[1]);

// Attente de la fin du fils

wait(NULL);

printf("fin de transfert\n");

exit(EXIT\_SUCCESS);

}

/\*N:B: Le processus père envoie le signal EOF au fils en écrivant

un octet nul dans le tube. Le fils doit s'arrêter lorsqu'il lit un octet nul

dans le tube, et fermer l'extrémité de lecture du tube.\*/

return 0;

}

**Signaux – TP5**

Le principe est a priori simple : un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même).

Le destinataire peut soit ignorer le signal, soit le capturer – c’est-à-dire dérouter provisoirement son exécution vers une routine particulière qu’on nomme gestionnaire de signaux

–, soit laisser le système traiter le signal avec un comportement par défaut.

**Chaque signal dispose d’un nom défini sous forme de constante symbolique commençant par SIG et d’un numéro associé**

en-tête <signal.h>

Prof : Pour assurer la portabilité d’un programme, on peut donc inclure en début de fichier des directives pour le préprocesseur,

**Gestion portable des signaux**

Un signal est un mécanisme utilisé pour envoyer une notification à un processus en cours d'exécution.

On utilise l’appel systeme kill pour envoyer des signaux aux processus.

Le processus peut alors traiter le signal en utilisant une fonction de rappel (également appelée "gestionnaire de signal") qui est exécutée lorsque le signal est reçu.

Prof : Le processus peut choisir d’ignorer le signal ou de réagir à celui-ci en exécutant une action spécifique.

Exp :

#include <signal.h>

#include <stdio.h>

void sigint\_handler(int sig) {

printf(" SIGINT reçu \n");

}

int main(void) {

// Installe le gestionnaire de signal pour SIGINT

signal(SIGINT, sigint\_handler);

// Boucle infinie

while (1) {

// Faites quelque chose d'autre ici...

}

return 0;

}

Dans cet exemple, le gestionnaire de signal sigint\_handler est défini pour traiter le signal SIGINT, qui est envoyé lorsque l'utilisateur appuie sur la touche "Ctrl + C" dans la console. Lorsque le signal est reçu, le gestionnaire de signal affiche un message à l'écran.

Rmq : Lorsqu’un signal est reçu par le processus, la fonction de rappel sera appelée à la place de la terminaison du processus.

**Signal SIGKILL**

Le signal SIGKILL est un signal spécial qui indique à un processus de terminer immédiatement

Lorsqu'il est envoyé à un processus, celui-ci est tué sans avoir la possibilité de s'exécuter ou de terminer proprement. Pour envoyer le signal SIGKILL à un processus :

kill -s SIGKILL <pid> Où <pid> est le numéro de processus du processus que vous souhaitez tuer.

**Signal SIGINT**

Le signal SIGINT (Interrupt) est envoyé au processus lorsque l'utilisateur appuie sur la touche Ctrl + C dans le terminal. Cela indique au processus qu'il doit arrêter son exécution immédiatement.

Par défaut, la plupart des processus ignorent le signal SIGINT et continuent de s'exécuter normalement lorsqu'il est envoyé

Rappel :

Les fonctions de rappels permettent à un processus de traiter un signal de manière spéciale et de définir son comportement lorsqu'il est reçu dans ce processus.

Exp:

#include <signal.h>

void sigint\_handler(int signum) {

// Code à exécuter lorsque le signal SIGINT est reçu

}

int main() {

// Définir la fonction de rappel pour le signal SIGINT

signal(SIGINT, sigint\_handler);

// Code du programme principal

while (1) {

// Faire quelque chose

}

return 0;

}

**Prof :** On peut utiliser la fonction sigint\_handler pour effectuer par exemple des tâches de nettoyage avant la terminaison du processus, ou pour ignorer complètement le signal et continuer à exécuter le programme normalement.

**NB :** Vous pouvez également définir vos propres signaux personnalisables et les traiter en utilisant des fonctions de rappel.

**N.B :** A la différence de ce signal, les signaux SIGTERM et SIGKILL ne peuvent être ignoré quoi qu’il arrive

**Tp5 exo2**

On désire créer un programme de façon a ce qu’il ignore le signal SIGINT

#include <signal.h>

int main() {

// Définir la fonction de rappel pour le signal SIGINT en utilisant SIG\_IGN

signal(SIGINT, SIG\_IGN);

// Code du programme principal

while (1) {

// Faire quelque chose

}

return 0;

}

**Prof:** Ce programme définit la fonction de rappel pour le signal SIGINT en utilisant la valeur **SIG\_IGN** ce qui indique au processus de l'ignorer. Le processus continuera à s'exécuter normalement même si le signal SIGINT est reçu.

Il est important de noter que certains signaux ne peuvent pas être ignorés et que le comportement par défaut sera appliqué lorsqu'ils sont reçus. Par exemple, le signal SIGKILL ne peut pas être ignoré et provoquera la terminaison immédiate du processus lorsqu'il est reçu.

**Signal SIGTSTP**

Le signal SIGTSTP (Terminal Stop) est un signal envoyé par le système lorsque l'utilisateur appuie sur la touche Ctrl + Z dans le terminal. Il indique au processus de mettre en pause son exécution et de passer en arrière-plan.

Lorsqu'un processus reçoit le signal SIGTSTP, il est mis en pause et ne s'exécute plus jusqu'à ce qu'il soit repris par l'utilisateur.

**Prof :** Cela permet à l'utilisateur de mettre en arrière-plan des processus longs ou gourmands en ressources, afin de pouvoir exécuter d'autres tâches dans le terminal sans attendre la fin de l'exécution du processus mis en arrière-plan.

1. Lancez le processus que vous souhaitez mettre en arrière-plan dans le terminal.
2. Appuyez sur la touche **Ctrl + Z** pour envoyer le signal **SIGTSTP** au processus.
3. Le processus sera mis en pause et un message de confirmation sera affiché dans le termina

**Tp5 exercice3**

**Une image contenant texte, capture d’écran, écran, fermer

Description générée automatiquement**

#include <signal.h>

#include <stdio.h>

void sigtstp\_handler(int signum) {

printf("Loupe !!\n");

}

int main() {

// Définir la fonction de rappel pour le signal SIGTSTP

signal(SIGTSTP, sigtstp\_handler);

// Boucle infinie sans affichage

while (1) {

// Ne rien faire

}

return 0;

}

**Prof :** Lorsque le signal SIGTSTP est reçu par le processus, la fonction sigtstp\_handler sera appelée et affichera le message "Loupe !!". Le processus continuera à s'exécuter en boucle infinie sans afficher quoi que ce soit d'autre.

**TP5 exo4**

On désire maintenant modifier le programme pour qu'il affiche le message "Loupe !!" lorsque le signal SIGTSTP est reçu pour la première fois, mais qu'il se termine normalement lorsqu'il est reçu une deuxième fois

#include <signal.h>

#include <stdio.h>

int stop\_received = 0; // Variable pour suivre le nombre de fois où le signal SIGTSTP a été reçu

void sigtstp\_handler(int signum) {

if (stop\_received == 0) {

// Premier signal SIGTSTP reçu

printf("Loupe !!\n");

stop\_received = 1; // Mettre à jour la variable pour indiquer qu'un signal a été reçu

}

else {

// Deuxième signal SIGTSTP reçu

printf("Arrêt du programme...\n");

exit(0); // Terminer le programme proprement

}

}

int main() {

// Définir la fonction de rappel pour le signal SIGTSTP

signal(SIGTSTP, sigtstp\_handler);

// Boucle infinie sans affichage

while (1) {

// Ne rien faire

}

return 0;

}

**Signal SIGUSR1**

Le signal SIGUSR1 (User Signal 1) est un signal personnalisable qui peut être envoyé à un processus par l'utilisateur ou par un autre processus. Il n'a pas de signification spécifique et peut être utilisé pour envoyer un message ou une instruction spéciale à un processus.

Pour l’envoyer a un processus par exemple :

kill -s SIGUSR1 <pid>

Prof: Comme pour tout autre signal, pour traiter le signal SIGUSR1 dans un programme, on va aussi définir une fonction de rappel qui sera appelée lorsque le signal est reçu par le processus

**Tp5 exercice 5**

Voici comment écrire un programme en C qui crée un processus fils en utilisant fork, envoie le signal SIGUSR1 au fils, attend sa terminaison et se termine lui-même.

int attente; // Variable globale pour le processus fils

function\_de\_rappel\_SIGUSR1(int signum) {

attente = 1; // Mettre à jour la variable pour sortir de la boucle

}

int main() {

pid\_t pid = fork();

if (pid == 0) {// Processus fils

printf("Processus fils lancé avec PID %d\n", getpid());

attente = 0;

// fils recoit la fonction le signal et execute la fonction de rappel associe

signal(SIGUSR1, function\_de\_rappel\_SIGUSR1)

while (ATTENTE == 0)

{

// Boucle vide

}

printf("Processus fils terminé\n");

exit(0);

}

else

{ // Processus pere

sleep(5);

// Envoyer le signal SIGUSR1 au processus fils

kill(pid, SIGUSR1); // Si on voulait envoyez le signal au pere dans le fils kill(getppid,SIGUSR1)

// Attendre la terminaison du processus fils

wait(NULL);

printf("Processus père terminé\n");

exit(0);

}

return 0;

}

**Signaux personnalisés**

Il est possible de définir ses propres signaux personnalisés en utilisant la fonction sigaction

Cette fonction permet de spécifier une fonction de rappel (ou "handler") pour être appelée lorsque le signal spécifié est reçu par le processus.

Un exemple de code qui montre comment définir un signal personnalisé et comment définir une fonction de rappel pour ce signal

// Fonction de rappel pour le signal personnalisé

void custom\_signal\_handler(int signum) {

printf("Le signal %d a été reçu !\n", signum);

}

int main() {

// Définition du signal personnalisé et de sa fonction de rappel

struct sigaction action;

action.sa\_handler = custom\_signal\_handler;

sigaction(SIGUSR1, & action, NULL);

// Code du programme...

return 0;

}

Pour envoyer le signal personnalisé au processus, vous pouvez utiliser la fonction kill

Exp :

#include <sys/types.h>

#include <signal.h>

int main() {

// Envoi du signal personnalisé au processus en cours d'exécution

kill(getpid(), SIGUSR1);

// Code du programme...sy

return 0;

}

**N.B :** Il est important de noter que vous ne pouvez pas utiliser n'importe quel numéro de signal comme signal personnalisé

**Contrôle parrain**

**Ancien contrôle 2015..**

Ce programme utilise les fonctions fork() et pipe() pour créer un processus fils et un tube nommé pour communiquer entre le père et le fils. Le fils utilise une boucle pour calculer des itérations en utilisant la fonction syracus() pour chaque indice dans une plage définie par les constantes symboliques DEBUT et FIN, et envoie ces itérations au père via le tube nommé. Le fils envoie également un signal SIGUSR1 au père pour lui indiquer qu'une itération a été envoyée. Le père utilise une structure de sigaction pour configurer la gestion des signaux SIGUSR1 pour exécuter la fonction rappelduFils() lorsqu'il reçoit ce signal. La fonction rappelduFils() lit l'itération envoyée par le fils via le tube nommé et l'affiche, et met à jour un compteur d'indice pour suivre les itérations traitées par le fils.

#include <stdio.h> // Pour printf et getchar

#include <stdlib.h> // Pour exit

#include <unistd.h> // Pour pipe et fork

#include <signal.h> // Pour sigaction et sigenphys

#define DEBUT 1000000 // Constante symbolique pour le début de la plage d'indices

#define FIN 1000000 // Constante symbolique pour la fin de la plage d'indices

// Fonction qui calcule une "iteration" pour un indice donné

long syracus(long indice) {

// Code qui effectue le calcul de l'iteration...

return iteration;

}

int main(){

int pipedef[2]; // Tableau de descripteurs de fichier pour le tube nommé

int ret = pipe(pipedef); // Création du tube

pid\_t pid = fork();

if (ret == -1) {

printf("pipe impossible\n");

exit(EXIT\_FAILLURE);

}

switch (pid)

{

case -1: printf("forck impossible\n");

exit(EXIT\_FAILLURE);

break;

case 0: fils();

break;

default:

pere();

break;

}

}

}

// Fonction exécutée par le processus fils

void fils() {

close(pipedef[0]); // Fermeture de l'extrémité de lecture du tube

long iteration;

for (long i = DEBUT + 1; i < FIN; i += 1) {

iteration = syracus(2\*i+1);

write(pipedef[1], &iteration, sizeof(iteration)); // Envoi de l'iteration au père via le tube nommé

kill(getppid(), SIGUSR1); // Envoi d'un signal au père pour lui indiquer qu'une iteration a été envoyée

}

close(pipedef[1]); // Fermeture de l'extrémité d'écriture du tube

exit(EXIT\_SUCCESS); // Fin du processus fils

}

// Fonction exécutée lorsque le processus père reçoit un signal SIGUSR1 envoyé par le processus fils

void rappelduFils() {

long iteration;

long indiceFilfs = DEBUT + 1; // Indice en cours de traitement par le processus fils

// Lecture de l'iteration envoyée par le fils via le tube nommé

read(pipedef[0], &iteration, sizeof(iteration));

printf("n=%d,iteration=%d", indiceFils, iteration); // Affichage de l'iteration

indiceFilfs += 2; // Mise à jour de l'indice en cours de traitement par le processus fils

}

// Fonction exécutée par le processus père

void pere() {

close(pipedef[1]); // Fermeture de l'extrémité d'écriture du tube

// Structure de sigaction pour configurer la gestion des signaux par le processus fils

struct sigaction sig\_fils;// Configuration de la gestion des signaux par le processus père

sig\_fils.saflags = SA\_RESTART; // SA\_RESTART pour indiquer que indique que les appels système bloquants doivent être automatiquement relancés lorsque le signal est reçu.

sigemptyset(&sig\_fils.sa\_mask); // Masquage de tous les signaux pendant l'exécution du handler

sig\_fils.sa\_handler = &rappelduFils; // Fonction à exécuter lorsque le process

if (sigaction(SIGUSR1, &sig\_fils, NULL) == -1) {

perror("Erreur lors de l'enregistrement du signal SIGUSR1");

}

sigaction(SIGUSR1, &sig\_fils, NULL);

// Pere continue son travail

long iteration;

for (long i = DEBUT; i <= FIN; i += 0) {

iteration = syracus(2\*i);

printf("m=%d,iteration=%d\n", i, iteration);

}

wait(0); // Attente de la fin du processus fils

close(pipedef[0]); // Fermeture de l'extrémité de lecture du tube

exit(EXIT\_SUCCESS); // Fin du processus père

}

**Contrôle 2015**

**Processus et pipes**

On veut écrire un programme permettant d’afficher simultanément a intervalles réguliers la charge de deux processeur d’une machine

On suppose que l’on dispose d’une fonction int chargeCPU() qui renvoie la charge du CPU sur lequel se deroule le processus qui appel cette fonction. Cette charge est mesuree en pourcentage (valeur entière entre 0 et 100)

Le principe retenu est le suivant :

Le programme lance un processus fils qui releve la charge de son CPU chaque10 seconde et l’envoie a son père par un pipe

Le programme reccupere dans le pipe la charge CPU relevee par son fils puis releve la charge de son propre CPU

Ainsi, chaque 10 seconde, le programme peut afficher la charge des deux processeurs sous la forme :

Charges des CPUs père/fils = xx/yy

**Code du programme**

#include <stdio.h> // Pour printf et getchar

#include <stdlib.h> // Pour exit

#include <unistd.h> // Pour pipe et fork

/\*On suppose qu'on dispose d'une fonction int chargeCPU() qui renvoie la charge d'un microprocesseur\*/

int main() {

int pipedef[2];

int ret = pipe(pipedef);

if (ret == -1) {

//erreur pipe

}

pid\_t pid = fork();

if (pid == -1) {

// erreur fork

}

if (pid == 0) {

//fils

close(pipedef[0]);

while (true) {

// Relevé de la charge CPU

int charge\_fils = chargeCPU();

// Envoi de la charge au père par le pipe

write(pipefd[1], &charge\_fils, sizeof(charge));

// Attente de 10 secondes

sleep(10);

close(pipedef[1]);

}

else

{

//pere

close(pipedef[1]);

while (true) {

// Relevé de la charge CPU du père

int charge\_pere = chargeCPU();

// Réception de la charge CPU du fils depuis le pipe

int charge\_fils;

read(pipefd[0], &charge\_fils, sizeof(charge\_fils));

// Affichage des charges

printf("Charges des CPUs père/fils = %d/%d\n", charge\_pere, charge\_fils);

}

}

return 0;

}

**Threads, sémaphores et signaux**

On veut écrire un programme qui ne s’arrêté qu’a la troisième fois où il a reçu CTRL C. Pour cela il faut utilisera un compteur incrémenté à chaque signal reçu

#include <stdio.h>

#include <signal.h>

int compteur = 0; // Compteur de signaux reçus

// Fonction appelée lorsque le signal SIGINT est reçu

void gestionnaire(int sig) {

compteur++;

printf("Signal reçu %d fois\n", compteur);

if (compteur == 3) {

printf("Arrêt du programme\n");

exit(0);

}

}

int main() {

// Enregistrement du gestionnaire pour le signal SIGINT

signal(SIGINT, gestionnaire);

// Boucle infinie

while (1) {

// Code du programme...

}

return 0;

}

On veut maintenant faire évoluer ce programme de façon a ce qu’il ne s’arrête que si l’on tape trois fois CTRL C en moins de 10 secondes. Pour cela le programme lancera un thread qui, chaque 10 secondes, affichera « Vous avez 10s pour arrêter le programme » et remettra à zero la variable servant au programme à compter le nombre de CTRL C qu’il a reçu. Ainsi, pour arrêter le programme il faudra parvenir à taper 3 fois CTRL C en moins de 10s après avoir vu ce message s’afficher. Bien entendu, comme ce compteur est modifié à la fois par le programme et par le thread, il faudra en protéger l’accès par un sémaphore d’exclusion mutuelle

pthread\_create, pthread\_join, pthread\_mutex\_init, pthread\_mutex\_lock, pthread\_mutex\_unlock, signal et sleep:

#include <stdio.h>

#include <pthread.h>

#include <signal.h>

int compteur = 0; // Compteur de signaux reçus

pthread\_mutex\_t mutex; // Sémaphore d'exclusion mutuelle

// Fonction appelée lorsque le signal SIGINT est reçu

void gestionnaire(int sig) {

pthread\_mutex\_lock(&mutex); // Verrouillage du mutex

compteur++;

printf("Signal reçu %d fois\n", compteur);

pthread\_mutex\_unlock(&mutex); // Déverrouillage du mutex

}

// Fonction exécutée par le thread

void\* thread\_fonction(void\* arg) {

while (1) {

// Attente de 10 secondes

sleep(10);

pthread\_mutex\_lock(&mutex); // Verrouillage du mutex

compteur = 0;

printf("Vous avez 10s pour arrêter le programme\n");

pthread\_mutex\_unlock(&mutex); // Déverrouillage du mutex

}

return NULL;

}

int main() {

pthread\_t thread; // Identifiant du thread

// Initialisation du mutex

pthread\_mutex\_init(&mutex, NULL);

// Enregistrement du gestionnaire pour le signal SIGINT

signal(SIGINT, gestionnaire);

// Création du thread

pthread\_create(&thread, NULL, thread\_fonction, NULL);

// Boucle infinie

while (1) {

// Code du programme...

pthread\_mutex\_lock(&mutex); // Verrouillage du mutex

if (compteur == 3) {

printf("Arrêt du programme\n");

pthread\_mutex\_unlock(&mutex); // Déverrouillage du mutex

break;

}

pthread\_mutex\_unlock(&mutex); // Déverrouillage du mutex

}

// Attente de la fin du thread

pthread\_join(thread, NULL);

// Destruction du mutex

pthread\_mutex\_destroy(&mutex);

return 0;

}

Ce code crée un thread en utilisant la fonction pthread\_create, qui exécute la fonction thread\_fonction. Cette fonction attend 10 secondes à chaque itération de la boucle infinie en utilisant la fonction sleep, puis réinitialise le compteur à 0 et affiche un message.

On procède par une initialisation dynamique du mutex.

**Fin contrôles**

**TD 1-Processus et Threads**

**Exercice1**

Le message "A" sera affiché une seule fois, le message "B" sera affiché deux fois et le message "C" sera affiché quatre fois.

Le programme commence en imprimant le message "A" et en créant un processus fils.

Le processus fils imprime le message "B" et crée un processus fils.

Le processus fils imprime le message "C" et s'arrête.

Le processus parent (qui a imprimé le message "B") imprime le message "C" et s'arrête.

Le processus grand-parent (qui a imprimé le message "A") imprime le message "C" une fois et s'arrête.

**Exercice 2 aet b**

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

#define N 1000000 // Taille du tableau

int main()

{

int tab[N]; // Tableau d'entiers

int valeur; // Valeur à rechercher

// Initialisation du tableau (à remplacer par la lecture d'un fichier)

for (int i = 0; i < N; i++)

tab[i] = i;

printf("Entrez la valeur à rechercher : ");

scanf("%d", &valeur);

pid\_t pid = fork();

if (pid == -1)

{ // Erreur lors de la création du processus fils

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

else if (pid == 0)

{ // Processus fils

for (int i = N / 2; i < N; i++)

{

if (tab[i] == valeur)

{

printf("Le fils a trouvé en position : %d\n", i);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le fils\n");

exit(EXIT\_SUCCESS);

}

else

{// Processus père

for (int i = 0; i < N / 2; i++)

{

if (tab[i] == valeur)

{

printf("Le père a trouvé en position : %d\n", i);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le père\n");

// Attente de la terminaison du processus fils avec waitpid +

int status;

waitpid(pid, &status, 0);

}

return 0;

}

**Rmq :** Il est question dans cet exercice de comprendre que Le processus père et le processus fils partagent le même espace mémoire, donc ils peuvent accéder au même tableau et vérifier s'il contient la valeur recherchée

**Exercice 3**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#define N 1000000 // Taille du tableau

int tab[N]; // Tableau d'entiers

int valeur; // Valeur à rechercher

int position = -1; // Position où la valeur a été trouvée (initialisée à -1 pour indiquer qu'elle n'a pas encore été trouvée)

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; // Mutex pour protéger la variable "position"

// Fonction exécutée par le premier thread

void\* recherche1(void\* arg)

{ // Recherche de la valeur dans la 1ère moitié du tableau

for (int i = 0; i < N / 2; i++)

{

if (tab[i] == valeur)

{

// Mise à jour de la variable "position"

pthread\_mutex\_lock(&mutex);

position = i;

pthread\_mutex\_unlock(&mutex);

break;

}

}

pthread\_exit(NULL);

}

// Fonction exécutée par le second thread

void\* recherche2(void\* arg)

{ // Recherche de la valeur dans la 2ème moitié du tableau

for (int i = N / 2; i < N; i++)

{

if (tab[i] == valeur)

{

// Mise à jour de la variable "position"

pthread\_mutex\_lock(&mutex);

position = i;

pthread\_mutex\_unlock(&mutex);

break;

}

}

pthread\_exit(NULL);

}

int main()

{ // Initialisation du tableau (à remplacer par la lecture d'un fichier)

for (int i = 0; i < N; i++)

tab[i] = i;

printf("Entrez la valeur à rechercher : ");

scanf("%d", &valeur);

pthread\_t thread1, thread2; // Identifiants des threads

// Création des threads

int ret1 = pthread\_create(&thread1, NULL, recherche1, NULL);

int ret2 = pthread\_create(&thread2, NULL, recherche2, NULL);

if (ret1 != 0 || ret2 != 0)

{

fprintf(stderr, "Erreur lors de la création des threads\n");

exit(EXIT\_FAILURE);

}

// Attente de la terminaison des threads

if (pthread\_join(thread1, NULL) != 0 || pthread\_join(thread2, NULL) != 0)

{

fprintf(stderr, "Erreur lors de l'attente de la terminaison des threads\n");

exit(EXIT\_FAILURE);

}

// Affichage du résultat

if (position == -1)

printf("La valeur n'a pas été trouvée\n");

else

printf("La valeur a été trouvée en position : %d\n", position);

return 0;

}

**TD2**

**Exercice1 - Utilisation de Sémaphore**

**1** Et 3 Ok

3) Code en C qui traite le problème de 1er et dernier lecteur

(Récupérer le code du td )

4) idem

**Exercice 2 – Thread et mutex**

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#define N 1000000 // Taille du tableau

int max = 0; // Variable globale pour stocker le maximum

int array[N]; // Tableau d'entiers

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; // Sémaphore de type mutex

void\* find\_max(void\* arg) {

int start = \*((int\*)arg); // Début de la partie du tableau à traiter

int end = start + N / 2; // Fin de la partie du tableau à traiter

// Recherche du maximum dans la partie du tableau assignée au thread

for (int i = start; i < end; i++) {

pthread\_mutex\_lock(&mutex); // Verrouillage du mutex

if (array[i] > max) {

max = array[i];

}

pthread\_mutex\_unlock(&mutex); // Déverrouillage du mutex

}

return NULL;

}

int main(void) {

// Initialisation du tableau avec des valeurs aléatoires

for (int i = 0; i < N; i++) {

array[i] = rand(); // Initialisation du tableau des valeurs aleatoires

}

// Création des threads

pthread\_t thread1, thread2;

int start1 = 0;

int start2 = N / 2;

pthread\_create(&thread1, NULL, find\_max, &start1);

pthread\_create(&thread2, NULL, find\_max, &start2);

// Attente de la fin des threads

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

// Affichage du résultat

printf("Max: %d\n", max);

return 0;

}

**TD3 -Processus et tubes**

**Exercice**

Ce programme crée un tube avant de créer un processus fils avec la fonction fork(). Le fils calcule la somme des éléments de la première moitié du tableau et l'envoie au père via le tube en utilisant la fonction write(). Le père lit cette valeur calcul la somme des éléments attend la terminaison du fils au cas où il est lancé le premier

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#define N 1000000

int main() {

int i;

int sum = 0;

int array[N];

// Initialisez le tableau avec des valeurs

for (i = 0; i < N; i++) {

array[i] = i;

}

// Créez un tube

int pipefd[2]; // tableau de descripteur de fichier

int ret = pipe(pipedf);

if (ret == -1) {

perror("pipe");

exit(EXIT\_FAILURE);

}

// Créez un processus fils

pid\_t pid = fork();

if (pid == -1) {

perror("fork");

exit(EXIT\_FAILURE);

}

if (pid == 0) {

// Fermez la lecture du tube dans le fils

close(pipefd[0]);

// Calculez la somme des éléments de la première moitié du tableau

int sum = 0;

for (i = 0; i < N / 2; i++) {

sum += array[i];

}

// Écrivez la somme dans le tube

write(pipefd[1], &sum, sizeof(sum));

// Fermez l'écriture du tube dans le fils

close(pipefd[1]);

// Terminez le fils

exit(EXIT\_SUCCESS);

}

else {

// Fermez l'écriture du tube dans le père

close(pipefd[1]);

// Calculez la somme des éléments de la seconde moitié du tableau

for (i = N / 2; i < N; i++) {

sum += array[i];

}

// Lisez la somme envoyée par le fils dans le tube

int child\_sum;

read(pipefd[0], &child\_sum, sizeof(child\_sum));

// Fermez la lecture du tube dans le père

close(pipefd[0]);

// Ajoutez la somme du fils à la somme du père pour obtenir la somme finale

sum += child\_sum;

// Attendre la terminaison du fils

wait(NULL); // ou waitpid(pid,&status,0)

// Affichez le résultat final

printf("Sum: %d\n", sum);

exit(EXIT\_SUCCESS);

}

return 0;

}

**Prof :** Pour initialiser un tableau à partir de données lues dans un fichier, vous pouvez utiliser la fonction fscanf() dans une boucle.

// Ouverture du fichier en lecture

FILE\* fp = fopen("data.txt", "r");

if (fp == NULL)

{

perror("fopen");

exit(EXIT\_FAILURE);

}

// Lecture des données et initialisation du tableau

for (i = 0; i < N; i++)

{

if (fscanf(fp, "%d", &tab[i]) != 1)

break;

}

// Fermeture du fichier

fclose(fp);

**Exercice2**

On veut écrire un programme d'analyse de son stéréo

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#define READ 0

#define WRITE 1

int lireWav(int\* right, int\* left) {

// Implémentation de la fonction de lecture du fichier wav

// Place la valeur de l'échantillon pour la voie droite dans right et celle de la voie gauche dans left

// Retourne 1 si le fichier est fini, 0 sinon

return 0;

}

int main(int argc, char\* argv[]) {

int right, left;

int pipe\_left[2], pipe\_right[2];

// Création des tubes

if (pipe(pipe\_left) == -1 || pipe(pipe\_right) == -1) {

perror("Erreur lors de la création des tubes");

exit(EXIT\_FAILURE);

}

pid\_t pid;

// Création du processus fils

pid = fork();

if (pid < 0) {

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

if (pid > 0) {

// Processus père

close(pipe\_left[READ]);

close(pipe\_right[WRITE]);

while (lireWav(&right, &left) == 0) {

// Envoi de la valeur de la voie gauche au processus fils

write(pipe\_left[WRITE], &left, sizeof(int));

// Traitement de la valeur de la voie droite

}

// Envoi de -1 au processus fils pour indiquer la fin du fichier

left = -1;

write(pipe\_left[WRITE], &left, sizeof(int));

close(pipe\_left[WRITE]);

// Réception du max et de la moyenne calculés par le processus fils

int max, average;

read(pipe\_right[READ], &max, sizeof(int));

read(pipe\_right[READ], &average, sizeof(int));

close(pipe\_right[READ]);

// Affichage des valeurs obtenues

printf("Max voie gauche : %d\n", max);

printf("Moyenne voie gauche : %d\n", average);

// Attente de la terminaison du processus fils

wait(NULL);

}

else {

// Processus fils

close(pipe\_left[WRITE]);

close(pipe\_right[READ]);

int value, sum = 0, count = 0;

int max = 0;

while (true) {

// Réception des valeurs envoyées par le processus père

read(pipe\_left[READ], &value, sizeof(int));

if (value == -1) {

// Fin du fichier

break;

}

// Mise à jour du max et du total des échantillons reçus

if (value > max) {

max = value;

}

sum += value;

count++;

}

close(pipe\_left[READ]);

// Calcul de la moyenne

int average = sum / count;

// Envoi du max et de la moyenne au processus père

write(pipe\_right[WRITE], &max, sizeof(int));

write(pipe\_right[WRITE], &average, sizeof(int));

close(pipe\_right[WRITE]);

}

return 0;

}

**TD4 – Signaux**

La fonction alarm() est utilisée pour programmer un signal d'interruption (SIGALRM) pour se déclencher après un certain délai (exprimé en secondes). Lorsque ce signal est reçu, le comportement par défaut est de terminer le processus, mais il peut être intercepté en utilisant un gestionnaire de signaux (signal handler) pour effectuer une action spécifique.

Lorsque le signal SIGALRM est envoyé, le processus est interrompu et le contrôle est transmis à la fonction de gestionnaire de signal, s'il en existe une.

**Exercice1**

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

/\*A la fin de la terminaison des 15 s par alarm, on appel SIGALRM\*/

int input\_done = 0;

void input\_handler(int sig) {

// Lorsque ce signal est reçu, on indique que la saisie est faite

input\_done = 1;

}

void alarm\_handler(int sig) {

printf("alors ça vient\n");

// Réarmer l'alarme pour qu'elle se déclenche de nouveau dans 15 secondes

alarm(15);

}

int main() {

/\* Configuration: Lorsque SIGALRM est declenchee par alarm(15), il

appel le gestionnaire alarm\_handler\*/

signal(SIGALRM, alarm\_handler);

// Configuration: Lorsque la saisie est faite il appel input\_handler

signal(SIGINT, input\_handler);

// Attend une saisie

printf("Entrez une entrée: ");

fflush(stdout);

/\* Déclencher l'alarme pour qu'elle se déclenche dans 15 secondes

apres l'attente de la saisie\*/

alarm(15);

// Attendre une entrée jusqu'à ce que input\_done soit mis à 1

char input[100];

while (!input\_done) {

fgets(input, sizeof(input), stdin);

}

// Désactiver l'alarme

alarm(0);

printf("Entrée reçue: %s", input);

printf("En attente de 50 secondes...\n");

sleep(50);

return 0;

}

**Exercice2**

#include <stdio.h>

#include <unistd.h>

#include <signal.h>

#include <string.h>

int input\_done = 0;

void input\_handler(int sig) {

// Lorsque ce signal est reçu, on indique que la saisie est faite

input\_done = 1;

}

void alarm\_handler(int sig) {

printf("alors ça vient\n");

// Réarmer l'alarme pour qu'elle se déclenche de nouveau dans 15 secondes

alarm(30); // alarm after 30 seconds instead of 15

}

int main() {

// Configurer le handler pour le signal SIGALRM

signal(SIGALRM, alarm\_handler);

// Configurer le handler pour le signal SIGINT

signal(SIGINT, input\_handler);

printf("Entrez une entrée: ");

fflush(stdout);

// Déclencher l'alarme pour qu'elle se déclenche dans 15 secondes

alarm(15);

// Attendre une entrée jusqu'à ce que input\_done soit mis à 1

char input[100];

while (!input\_done) {

/\*Si la valeur de retour de alarm(0) est 30, cela signifie

que l'alarme a été définie pour 30 secondes et qu'elle n'a

pas encore été déclenchée\*/

if (alarm(0) == 30) {

printf("Bon j'abandonne ! \n");

return 0;

}

fgets(input, sizeof(input), stdin);

}

// Désactiver l'alarme

alarm(0);

printf("Entrée reçue: %s", input);

return 0;

}

**Exercice 3**

#define N 1000000 // Taille du tableau

//Fonction pour traiter le signal USR1

void signal\_handler(int sig)

{

if (sig == SIGUSR1)

{

printf("La valeur a été trouvée par l'autre processus. Arrêt de la recherche...\n");

exit(0);

}

}

int main()

{

int tab[N]; // Tableau d'entiers

int valeur; // Valeur à rechercher

// Initialisation du tableau (à remplacer par la lecture d'un fichier)

for (int i = 0; i < N; i++)

tab[i] = i;

printf("Entrez la valeur à rechercher : ");

scanf("%d", &valeur);

pid\_t pid = fork();

if (pid == -1)

{ // Erreur lors de la création du processus fils

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

else if (pid == 0)

{ // Processus fils

signal(SIGUSR1, signal\_handler);

for (int i = N / 2; i < N; i++)

{

if (tab[i] == valeur)

{

printf("Le fils a trouvé en position : %d\n", i);

kill(getppid(), SIGUSR1);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le fils\n");

exit(EXIT\_SUCCESS);

}

else

{// Processus père

signal(SIGUSR1, signal\_handler);

for (int i = 0; i < N / 2; i++)

{

if (tab[i] == valeur)

{

printf("Le père a trouvé en position : %d\n", i);

kill(pid, SIGUSR1);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le père\n");

}

return 0;

}

**Exercice4**

#define N 1000000 // Taille du tableau

void signal\_handler(int sig) {

if (sig == SIGCHLD)

printf("Fils terminé, fin du père aussi.\n");

exit(0);

}

int main()

{

int tab[N]; // Tableau d'entiers

int valeur; // Valeur à rechercher

// Initialisation du tableau (à remplacer par la lecture d'un fichier)

for (int i = 0; i < N; i++)

tab[i] = i;

printf("Entrez la valeur à rechercher : ");

scanf("%d", &valeur);

signal(SIGCHLD, signal\_handler);

pid\_t pid = fork();

if (pid == -1)

{ // Erreur lors de la création du processus fils

perror("Erreur lors de la création du processus fils");

exit(EXIT\_FAILURE);

}

else if (pid == 0)

{ // Processus fils

for (int i = N / 2; i < N; i++)

{

if (tab[i] == valeur)

{

printf("Le fils a trouvé en position : %d\n", i);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le fils\n");

exit(EXIT\_SUCCESS);

}

else

{// Processus père

for (int i = 0; i < N / 2; i++)

{

if (tab[i] == valeur)

{

printf("Le père a trouvé en position : %d\n", i);

exit(EXIT\_SUCCESS);

}

}

printf("La valeur n'a pas été trouvée par le père\n");

// Attente de la terminaison du processus fils

pause();/\*On utilise la fonction pause() pour faire attendre le pere

indéfiniment,jusqu'à ce qu'il reçoive un signal.

Dès que le fils termine son execution, le signal\*/

}

return 0;

}

**Fin des TD 😉**

**Gestion classique des signaux (Hors programme)**

**Signal SIGPIPE**

Ce signal est émis par le noyau lorsqu’un processus tente d’écrire dans un tube qui n’a pas de lecteur. Ce cas peut aussi se produire lorsqu’on tente d’envoyer des données dans une socket TCP/IP dont le correspondant s’est déconnecté.

Le signal SIGPIPE (défini dans SUSv3) arrête par défaut le processus qui le reçoit. Toute application qui établit une connexion réseau doit donc soit intercepter le signal, soit (ce qui est préférable) l’ignorer. Dans ce dernier cas en effet, les appels système comme write() renverront une erreur EPIPE dans errno. On peut donc gérer les erreurs au coup par coup dès le retour de la fonction

EPIPE : Tentative d’écriture avec write() dans un tube dont l’autre extrémité a été fermée par le processus lecteur. Cette erreur n’est envoyée que si le signal SIGPIPE est bloqué ou ignoré

U2n signal interne – émis par pthread\_kill() – ou un signal externe synchrone – comme SIGBUS, SIGFPE, SIGPIPE qui viennent en réponse à une action particulière d’un thread – sont naturellement reçus par le thread visé.

**2Différences entre flux et descripteur**

Les descripteurs de fichiers sont des valeurs de type int, que le noyau associe à un fichier à la demande d’un processus. Ces entiers sont en réalité des indices dans des tables propres à chaque processus, que le noyau est le seul à pouvoir modifier. Les descripteurs fournis par le noyau peuvent bien entendu être associés à des fichiers réguliers, mais aussi à d’autres éléments du système, comme des répertoires, des périphériques accessibles par un fichier spécial, des moyens de communication comme les tubes (pipe) ou les files (FIFO)

**Ouvertures particulières de flux**

La fonction fopen() dispose de deux variantes permettant d’ouvrir un flux de deux manières légèrement différentes. La première fonction est fdopen(), dont le prototype est : FILE \* fdopen (int descripteur, const char \* mode); Cette fonction permet de disposer d’un flux construit autour d’un descripteur de fichier déjà obtenu auparavant. Ce descripteur doit avoir été fourni précédemment par l’un des appels système suivants :

open(), creat(), ouvrant un fichier disque. • pipe(), qui crée un tube de communication entre processus. • socket(), permettant d’établir une liaison réseau. • dup(), dup2(), qui servent à dupliquer un descripteur existant