Récapitulatif :

1. **Quelques bibliothèques essentielles**

<stdlib.h> : Gestion de la mémoire, conversions et fonctions systèmes

<string.h> : Gestion de chaînes de caractères

<ctype.h> : Manipulation de caractères

<stdio.h> : Gestion des fichiers et des Entrées/Sorties en général

<unistd.h> : Accès à l'API du système d'exploitation POSIX

<threads.h> : Création/Gestion de threads

1. **Compilation/exécution**

Compilation

gcc fichier.c -o fichier\_sortie -Wall

ajouter -lpthread si le programme contient des threads

Exécution

./fichier\_sortie

1. **Utilisation de Read/Write**

**read(int fd, char\* buf size\_t count);**

fd : où on écrit : buf la chaîne dans laquelle on écrit ; count la longueur de buf ou longueur que l’on souhaite lire < longueur de buf

Note : recv peut être utilisé comme alternative. recv prend un 4ème paramètre nommé flag. Si ce paramètre est mis à 0, le comportement de recv est le même que celui de read : la lecture est bloquante, tant que l’on n’a pas lu les count caractères on attend. S’il est mis à une autre valeur le comportement est différent (non bloquant par exemple pour 1).

**write(int fd, char \* buf, size\_t count)**

Même chose que read. send également une alternative avec également des flags.

1. **Utilisation du fork**

Exemple basique :

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

int valeur, status;

valeur=fork();

switch (valeur) {

case 0 :

actions\_du\_fils;

default :

actions\_du\_pere;

wait(&status);

}

exit(EXIT\_SUCCESS);

}

Remarques :

1. Le fork crée un nouveau processus copie du processus appelant. Celui ci aura son propre PID.
2. wait permet d’attendre la fin d’exécution du processus fils
3. **Utilisation des threads**

Exemple basique

void fonction\_thread() {

code\_exécuté\_par\_le\_thread;

pthread\_exit(NULL);}

int main() {

int nb\_threads=x;

pthread\_attr\_t attr;

pthread\_t thread\_id[nb\_threads];

pthread\_attr\_init(&attr);

for (int i=0; i<nb\_threads;i++){

pthread\_create(&thread\_id[i], &attr, (void \*)fonction\_thread, (void \*)NULL);

pthread\_join(thread\_id[i],NULL);}

exit(0); }

Remarques :

1. Les étapes clés à retenir sont l’instantiation du thread (init + create) et le fait de s’assurer que le thread puisse se dérouler jusqu’à la fin. Pour ce faire, deux options : join (on l’attache au programme) ou detach (on l’en détache)
2. Chaque thread peut avoir sa propre fonction mais différents threads peuvent utiliser une même fonction.
3. Si de multiples threads doivent accéder à une même information, l’utilisation d’une variable globale est une solution
4. Si on a besoin qu’une partie de l’exécution d’un thread (exemple incrémentation d’une variable) ne puisse être ralisé à un instant donné que par un seul thread, utilisation de mutex est nécessaire. Pour ce faire, quatre étapes :
   1. En variable globale on définit un mutex (objet utilisé pour bloquer une exécution) : pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER;
   2. On locke l’exécution au moment où on commence à interagir avec la variable partagée par plusieurs threads (dans la fonction du thread) : pthread\_mutex\_lock(&mutex);
   3. On unlock dès qu’on a fini (toujours dans la fonction du thread) : pthread\_mutex\_unlock(&mutex);
   4. On supprime le mutex dans le main à la fin de l’exécution du programme principal : pthread\_mutex\_destroy(&mutex);

1. **Utilisation de pipe**

Un objectif : Etablier une communication entre processus père et un processus fils (fork)

Exemple basique :

int main () {

int tube[2];

char clavier, ecran;

if (pipe(tube)==-1) {

exit(EXIT\_FAILURE);

}

if (fork()==0) {

/\* processus fils \*/

close(tube[1]);

exécution du processus fils;

exit(EXIT\_SUCCESS);

}

else {

/\* processus père \*/

close(tube[0]);

exécution du processus père;

exit(EXIT\_SUCCESS);

}

}

Remarque : Le processus est simple : une fois que l’on a fait un fork, dans le fils on ferme une entrée du tube et dans le père on ferme l’autre. On lit/écrit ensuite dans l’entrée que l’on n’a pas fermé.

1. **Utilisation de signaux**

Exemple basique :

void traitement\_sig(int signo) {

/\* traitement associé au signal : qu’est ce qu’on fait \*/ }

int main () {

signal(sig,traitement\_sig);

/\* associer le traitement au signal \*/

code\_du\_programme;

}

Remarques :

1. Pour interagir avec un signal on utilise la fonction signal qui prend en paramètre le nom du signal et l’action à exécuter : signal(nom, action)
2. Il existe des actions de base : SIG\_IGN (on ignore le signal) et SIG\_DFL (après avoir modifié le comportement d’un signal on lui rend son comportement par défaut)
3. traitement\_sig est une fonction de type void que l’on appelle sans parenthèses
4. SIGINT correspond à CTRL + C

TP1 :

Pour compiler vos programmes utiliser gcc xxx.c –o xxx

N'hésitez pas à utiliser la commande **man** pour obtenir de l'aide sur les fonctions et commandes

**ATTENTION pour tous ces exercices n'oubliez pas de mettre un '\n' à la fin de vos printf pour être sûrs qu'ils soient faits immédiatement.**

1°) *Fonctions utiles : fork , getpid , getppid , exit, sleep, printf*

Ecrire un programme qui :

* affiche : "je suis le processus numéro x" où x est son numéro de processus
* puis récupère dans une variable la valeur retournée par la fonction fork
* puis affiche : "fork m'a renvoyé la valeur : x" où x est le contenu de cette variable
* puis affiche : "je suis le processus numéro x et mon père est le processus numéro y" où  x est son numéro de processus et y celui de son père

Combien d'affichages a-t-on ?

A quoi correspondent-ils ?

2°) Ecrire un processus qui crée un processus fils par fork.

Le processus père affiche les 10 premiers nombres impairs par une boucle for

Le processus fils affiche les 10 premiers nombres pairs par une boucle for

a) exécuter ce programme et voir comment se font les affichages.

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

c) 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 ?

Lancer à nouveau ce programme et pendant qu'il affiche des nombres pairs exécuter la commande ps -l. On voit que le processus fils continue à fonctionner mais qui est son père ? Quel est ce processus ?

*Fonctions utiles : fork , execlp , execl , exit, sleep, printf, wait*

3°) Ecrire un processus qui crée un processus fils par fork

Le processus fils exécute la commande unix 'ps' à l'aide de la fonction execlp

Le processus père attend la fin de son fils et affiche la valeur que lui a retournée son fils.

4°) Remplacer dans le programme du 3°) l'appel à execlp par un appel à execl que se passe-t-il pourquoi ?

5°) Modifier le programme du 3°) pour que la commande exécutée par le fils ainsi que les paramètres de cette commande soient passés en paramètre au programme.

Tester votre programme en tapant : ./votre\_programme ls –li

et vérifier que le processus fils tient bien compte de l'option –li de la commande ls qui affiche le répertoire courant en donnant, pour chaque fichier, son numéro d'i-node.

SOLUTION

1°)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main () {

int valeur;

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

valeur=fork();

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

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

exit(EXIT\_SUCCESS);

}

L'affichage obtenu est le suivant :

Je suis le processus numéro : 3012

fork m'a renvoyé la valeur : 0

Je suis le processus numéro : 3013 et mon père est le processus de numéro : 3012

fork m'a renvoyé la valeur : 3013

Je suis le processus numéro : 3012 et mon père est le processus de numéro : 2750

La 1ere ligne est affichée par le processus père

La 2éme ligne est affichée par le processus fils (fork lui retourne 0)

La 3ème ligne est affichée par le processus fils (son père est le 3012)

La 4éme ligne est affichée par le processus père (fork lui retourne le numéro du fils)

Le 5ème ligne est affichée par le processus père dont le père est le shell.

2°)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main () {

int i;

if (fork()==0) {

/\* processus fils \*/

for (i=0; i<20; i=i+2) {

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

/\* sleep(1); \*/

}

exit(EXIT\_SUCCESS);

}

else {

/\* processus père \*/

for (i=1; i<21; i=i+2) {

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

/\*sleep(1);\*/

}

exit(EXIT\_SUCCESS);

}

}

a) Les affichages ne sont pas mélangés (on voit s'exécuter le fils puis le père). Ceci est dû au fait que la durée d'exécution d'une boucle pour 20 valeurs ne suffit pas pour changer de processus. On peut voir le changement si on fait une boucle pour 10 000 valeurs mais c'est illisible.

b) Les nombres pairs sont affichés puis les impairs (1 par seconde) puis le programme rend la main.

c) Les nombres impairs sont affichés puis le programme rend la main puis les nombres pairs sont affichés (1 par seconde).

avec ps –l on peut voir que le processus fils a été hérité par init (processus 1)

3°)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

int main () {

int valeur,fils,etat;

valeur=fork();

switch (valeur) {

case 0 :

execlp("ps","ps",NULL);

exit(EXIT\_SUCCESS);

break;

default :

fils=wait(&etat);

printf("Mon fils a terminé avec l'état : %d\n",etat);

exit(EXIT\_SUCCESS);

}

exit(EXIT\_SUCCESS);

}

4°) Le fils n'exécute pas la commande ps car execl ne la trouve pas.

5°)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <errno.h>

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

int valeur,fils,etat,retour;

valeur=fork();

switch (valeur) {

case 0 :

**retour=execlp(argv[1],argv[1],argv[2],NULL);**

exit(retour);

break;

default :

fils=wait(&etat);

printf("Mon fils a terminé avec l'état : %d\n",etat);

exit(EXIT\_SUCCESS);

}

exit(EXIT\_SUCCESS); }

TP2

ATTENTION : pour compiler les programmes utilisant des threads vous devez ajouter l'option **-lpthread** à la commande gcc.

*Fonctions utiles: pthread\_create, pthread\_attr\_init, pthread\_attr\_setdetachstate, pthread\_exit, pthread\_join, sleep*

1°) Ecrire un programme qui :

* crée deux threads :
* affiche : "les 2 threads sont lancés"
* et se termine normalement

Le premier thread fait une boucle infinie qui affiche un '.' chaque seconde (avec printf avec retour à la ligne après le point).

Remarque : si vous voulez utiliser un printf sans retour à la ligne pour que ce soit plus lisible vous pouvez  mettre dans ce thread : **setbuf(stdout,NULL)** qui supprime la bufferisation de l'affichage.

Le second thread affiche "tapez un caractère" puis attend un caractère au clavier par getchar, dès qu'il l'a reçu il se termine.

Exécuter ce programme, Ça ne marche pas !  Pourquoi ?

2°) Modifier le programme du 1°) pour qu'après avoir affiché "les 2 threads sont lancés", il attende la terminaison du second thread (celui qui fait une saisie au clavier) puis affiche : "On va s'arrêter là" et se termine normalement.

Exécuter ce programme, qu'est ce que ça a changé ? Pourquoi ?

3°) Que se passe-t-il si on modifie le programme pour qu'il attende la fin du premier thread au lieu du second ?

4°) Modifier le programme du 2°) pour qu'il crée les deux threads dans l'état 'détaché'. Pour cela il faut utiliser la fonction pthread\_attr\_setdetachstate pour préparer convenablement les attributs des threads créés.

Exécuter ce programme. Ça ne marche plus !  Pourquoi ?

SOLUTION

1°)

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

void thread1() {

setbuf(stdout,NULL);

while(0==0) {

printf(".");

sleep(1);

}

pthread\_exit(NULL);

}

void thread2() {

char c;

printf("tapez un caractère\n");

c=getchar();

pthread\_exit(NULL);

}

int main() {

int i;

pthread\_attr\_t attr;

pthread\_t thread\_id[2];

pthread\_attr\_init(&attr);

pthread\_create(&thread\_id[0], &attr, (void \*)thread1, (void \*)NULL);

pthread\_create(&thread\_id[1], &attr, (void \*)thread2, (void \*)NULL);

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

exit(0);

}

Il ne se passe rien car le programme se termine de suite et donc les 2 threads sont tués avant d'avoir pu faire quoi que ce soit.

2°)

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

void thread1() {

setbuf(stdout,NULL);

while(0==0) {

printf(".");

sleep(1);

}

pthread\_exit(NULL);

}

void thread2() {

char c;

printf("tapez un caractère\n");

c=getchar();

pthread\_exit(NULL);

}

int main() {

int i;

pthread\_attr\_t attr;

pthread\_t thread\_id[2];

pthread\_attr\_init(&attr);

pthread\_create(&thread\_id[0], &attr, (void \*)thread1, (void \*)NULL);

pthread\_create(&thread\_id[1], &attr, (void \*)thread2, (void \*)NULL);

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

**pthread\_join(thread\_id[1],NULL);**

printf("On va s'arrêter là\n");

exit(0);

}

Le programme fonctionne : on voit s'afficher un point chaque seconde tant qu'on n'a pas saisi de caractère. Dès que l'on saisit un caractère le programme s'arrête ainsi que les threads.

3°)

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

void thread1() {

setbuf(stdout,NULL);

while(0==0) {

printf(".");

sleep(1);

}

pthread\_exit(NULL);

}

void thread2() {

char c;

printf("tapez un caractère\n");

c=getchar();

pthread\_exit(NULL);

}

int main() {

int i;

pthread\_attr\_t attr;

pthread\_t thread\_id[2];

pthread\_attr\_init(&attr);

pthread\_create(&thread\_id[0], &attr, (void \*)thread1, (void \*)NULL);

pthread\_create(&thread\_id[1], &attr, (void \*)thread2, (void \*)NULL);

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

**pthread\_join(thread\_id[0],NULL);**

printf("On va s'arrêter là\n");

exit(0);

}

TP3

Dans cet algorithme la variable 'tmp' est une variable locale à la fonction 'compteur'

Le programme attendra la fin des 2 threads puis affichera : "valeur finale du compteur = x" où x est le contenu de la variable cpt lorsque les deux threads sont terminés.

Remarque : pour afficher un unsigned long avec printf il faut utiliser le format %lu et non %d.

Exécuter plusieurs fois le programme et relever les valeurs affichées obtient-on toujours la même valeur finale ? Pourquoi ?

Pourquoi la valeur finale du compteur n'est-elle pas de 20 millions alors que chaque thread a ajouté 10 millions à ce compteur ?

2°) On va maintenant améliorer le programme du 1°) de façon à contrôler les accès concurrents à la variable partagée 'cpt'. Pour ce faire le programme va déclarer en variable globale un sémaphore de type MUTEX initialisé par la constante PTHREAD\_MUTEX\_INITIALIZER. Les deux threads utiliseront ce sémaphore pour protéger  les 3 instructions de modification de 'cpt' placées dans la boucle.

Exécuter plusieurs fois ce programme et relever les valeurs affichées. Vérifier que la valeur finale est bien maintenant de 20 millions à tous les coups.

3°) On peut mesurer le temps d'exécution d'un programme (prog) grâce à la commande UNIX 'time' pour cela il suffit de taper : **time prog**

Exécuter à nouveau le programme du 1°) avec mesure du temps puis faites la même chose avec le programme du 2°) et comparer les temps. Refaites cette comparaison plusieurs fois pour avoir des valeurs moyennes. Pouvez-vous expliquer cette différence ?

SOLUTION

1°)

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

unsigned long int cpt;

void calcul() {

int m;

unsigned long int tmp;

for (m=0; m<10000000; m++) {

tmp=cpt;

tmp++;

cpt=tmp;

}

printf("cpt=%lu\n",cpt);

pthread\_exit(NULL);

}

int main() {

int i;

pthread\_attr\_t attr;

pthread\_t thread\_id[2];

cpt=0;

pthread\_attr\_init(&attr);

pthread\_create(&thread\_id[0], &attr, (void \*)calcul, (void \*)NULL);

pthread\_create(&thread\_id[1], &attr, (void \*)calcul, (void \*)NULL);

for (i=0; i<2; i++) pthread\_join(thread\_id[i],NULL);

printf("Valeur finale de cpt : %lu\n",cpt);

exit(0);

}

Lorsque l'on exécute le programme plusieurs fois la valeur finale n'est jamais la même en raison de l'enchaînement aléatoire des threads.

La valeur finale de cpt est inférieure à 20 millions ceci est dû aux accès concurrents des processus à la variable cpt.

3°) Le temps d'exécution de la version avec mutex est bien plus long (le double environ) en raison des blocages / déblocages des threads et de la manipulation du sémaphore.

2°)

#include <stdio.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

unsigned long int cpt;

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

void calcul() {

int m;

unsigned long int tmp;

for (m=0; m<10000000; m++) {

**pthread\_mutex\_lock(&mutex);**

tmp=cpt;

tmp++;

cpt=tmp;

**pthread\_mutex\_unlock(&mutex);**

}

printf("cpt=%lu\n",cpt);

pthread\_exit(NULL);

}

int main() {

int i;

pthread\_attr\_t attr;

pthread\_t thread\_id[2];

cpt=0;

pthread\_attr\_init(&attr);

pthread\_create(&thread\_id[0], &attr, (void \*)calcul, (void \*)NULL);

pthread\_create(&thread\_id[1], &attr, (void \*)calcul, (void \*)NULL);

for (i=0; i<2; i++) pthread\_join(thread\_id[i],NULL);

printf("Valeur finale de cpt : %lu\n",cpt);

**pthread\_mutex\_destroy(&mutex);**

exit(0);

}

La valeur finale de cpt est maintenant bien de 20 millions car il n'y a plus d'accès concurrents.

TP4 :

1°) Ecrire un programme qui ouvre un tube (pipe) puis génère un processus fils (fork) avec lequel il va communiquer par ce tube.

Le père lit des caractères au clavier (getchar) et les écrit dans le tube, le fils récupère ces caractères et les affiche après les avoir mis en majuscule (fonction toupper). La saisie du caractère '$' termine le programme. Attention pour éviter que le processus fils ne continue à attendre que des caractères arrivent il faut que le père envoie le caractère $ dans le tube et que le fils s'en serve pour se terminer.

Exécuter ce programme et vérifier qu'il fonctionne correctement.

Remarque : Il faut taper 'entrée' après chaque caractère saisi pour que getchar prenne en compte la saisie.

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <ctype.h>

int main () {

int tube[2];

char clavier, ecran;

if (pipe(tube)==-1) {

printf("création du tube impossible\n");

exit(EXIT\_FAILURE);

}

setbuf(stdout,NULL);

if (fork()==0) {

/\* processus fils \*/

close(tube[1]);

do {

read(tube[0],&ecran,sizeof(char));

printf("%c", toupper(ecran));

} while(ecran != '$');

exit(EXIT\_SUCCESS);

}

else {

/\* processus père \*/

close(tube[0]);

while ((clavier=getchar())!='$') {

write(tube[1],&clavier,sizeof(char));

}

write(tube[1],&clavier,sizeof(char)); /\* pour envoyer le $ \*/

exit(EXIT\_SUCCESS);

}

}

2°) La méthode utilisée ici pour arrêter le processus fils (envoi du caractère '$') n'est pas toujours possible. En effet cela suppose de pouvoir disposer d'un caractère spécial non utilisé par ailleurs. Si, par exemple, le programme utilisait le tube pour passer le contenu de fichiers à son fils il faudrait être sûr qu'aucun fichier ne contient de caractère '$'.  Pour éviter  ce problème il vaut mieux que le fils détecte la terminaison du transfert par la fermeture du tube par son père. Ceci est possible car la fonction de lecture (read) retourne 0 lorsqu'elle ne trouve plus rien à lire parce que le tube est fermé en écriture. Modifier le programme précédent pour que le processus fils s'arrête selon ce procédé.

2°)

 #include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <ctype.h>

int main () {

int tube[2];

char clavier, ecran;

if (pipe(tube)==-1) {

printf("création du tube impossible\n");

exit(EXIT\_FAILURE);

}

setbuf(stdout,NULL);

if (fork()==0) {

/\* processus fils \*/

close(tube[1]);

**while (read(tube[0],&ecran,sizeof(char))!=0) {**

pfrintf("%c", toupper(ecran));

}

**close(tube[0]);**

exit(EXIT\_SUCCESS);

}

else {

/\* processus pÃ¨re \*/

close(tube[0]);

while ((clavier=getchar())!='$') {

write(tube[1],&clavier,sizeof(char));

}

**close(tube[1]); /\* on n'envoie plus le $ mais on ferme le tube \*/**

exit(EXIT\_SUCCESS);

}

}