Programmation Concurrente: Arguments et Fork

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Arguments

Exercice 1: Miroir

Variante 1

Écrivez un programme (miroir.c) qui prend en argument une chaîne de caractères et **l'affiche** à l'envers. Exemple:

```
user@machine:~$ ./miroir trace
ecart
```

Variante 2

Modifiez le programme précédent pour qu'il traite l'ensemble des arguments que l'utilisateur lui fourni. Exemple:

```
user@machine:~$ ./miroir trace soda saper
ecart ados repas
```

Exercice 2: Moyenne

Écrivez un programme (moyenne.c) qui calcule et affiche la moyenne d'un ensemble de notes (nombres entiers) passées en arguments.

Votre programme devra vérifier qu'une note fournie est valide. C'est-à-dire qu'il s'agit bien d'un entier dont la valeur est comprise entre 0 et 20.

Exemple d'usage:

```
user@machine:~$ ./moyenne 12 18 toto
Note(s) non valide(s)
user@machine:~$ ./moyenne 12 18 3.5
Note(s) non valide(s)
user@machine:~$ ./moyenne 12 18 22
Note(s) non valide(s)
user@machine:~$ ./moyenne
Aucune note fourni, moyenne impossible à calculer
user@machine:~$ ./moyenne 12 18 3
Moyenne: 11.00
```

Aide

Conversion chaîne de caractère vers entier

```
int nb_success = sscanf(str,"%d/%d/%d",&day,&month,&year);
int nb_success = sscanf(str,"%d",&nb);
```

sscanf renvoi le nombre de conversion réussi.

- 1er cas: si nb_success est différent de 3 alors la chaine (str) fournie ne contient pas trois entiers séparés par des slashs
- 2ème cas: si nb_success est différent de 1 alors la chaine (str) fournie ne contient pas un entier

Toutes les conversions sont enregistrées dans les variables qui suivent le format. C'est pour cela que l'on passe l'adresse de ces variables. La fonction pourra modifier leur valeur, puisqu'elle connaît leur adresse.

Formatage des affichages

```
printf("%4.2f",var);
```

- % indique qu'il s'agit d'un format
- f indique qu'il s'agit d'un float
- 4 indique que la taille minimale de la chaîne affiché sera de 4 caractères
- 2 indique qu'il y aura maximum deux décimales

Fork

Exercice 3: Fork dans une boucle

Écrivez un programme (boucle.c) qui fait appel à fork dans une boucle qui fera 3 itérations. À l'intérieur de chaque itération, le programme affichera en une seule fois:

- la valeur du compteur de boucle
- le PID du processus
- le PPID (PID du processus parent)
- la valeur retournée lors de l'appel à fork.

Dessinez l'arbre des processus correspondant à l'exécution de ce programme.

⚠ Il n'y a aucune condition (if/else) à mettre que ce soit sur l'affichage des informations ou sur fork.

Exemple d'affichage:

```
0: Je suis le processus 456, mon père est le processus 123 fork a retourné 0
```

Exercice 4: Wait

Écrivez un programme (wait.c) qui:

- lit sur la ligne de commande (argc/argv) le nombre **N** de processus qu'il doit créer.
- Une fois tous les processus créés, il se met en attente de ses fils
- Dès qu'un fils se termine, il affiche l'identité de ce fils et la valeur de retour de ce fils.

Les fils quant à eux:

- affiche leur pid, le pid de leur père et se mettent en attente (sleep) pendant 2*i secondes.
- Après l'attente, ils indiquent qu'ils reprennent leur exécution avant de faire un exit avec la valeur de i.

⚠ **i** étant le numéro de l'itération qui a servi à créer le processus.

Ce programme est-il déterministe ? Pourquoi ?

Exercice 5: Analyse de code

⚠ Rappel: break sort de la boucle dans laquelle il se trouve

Cas N°1

```
1  #include <sys/types.h>
2  #include <unistd.h>
3
4  int main(void) {
5      int i, n=0;
6      pid_t fils_pid;
7      for (i=1; i< 5; i++)
8      {
9          fils_pid = fork();
10         if (fils_pid > 0)
11         {
12             n = i*2;
13             break;
14         }
15     }
16     printf("%d\n", n);
17     return 0;
18 }
```

- Dans le block d'exécution du if (ligne 12 et 13). Dans quel processus se trouve-t-on ? (père ou fils)
- Ce programme est-il déterministe ? Justifiez
- Si le programme est déterministe:
  - Indiquez exactement ce qui sera affiché lors de son exécution.
- Si le programme n'est pas déterministe:
  - Indiquez un des affichages possible
  - Faîtes en sorte de rendre ce programme déterministe
  - Indiquez ce qui sera affiché alors.
- L'appel à fork() peut-il échouer ? Pourquoi ?

Cas N°2

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  int main(void) {
6      int i = 0;
7      for (i=0 ; i<4 ; i++)
8          if (fork())
9              break;
10     srand(getpid());
11     int delai = rand()%4;
12     sleep(delai);
13     printf("Je suis %c, j'ai dormi %d s\n"
14         ,'A'+i,delai);
15     return 0;
16 }
```

- Donnez l'arbre des processus générés par ce programme
- Ce programme est-il déterministe ?
- Sans modifier les lignes 1 à 12, modifiez ce programme pour qu'il produise un affichage dans l'ordre alphabétique inverse (EDCBA)

Cas N°3

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define PROCESS 4
#define TAB_SIZE 100

int main(void) {
    int i = 0;
    int pere = 1;
    int tab[100] = {0};
    for (i = 0; i < TAB_SIZE; i++)
        tab[i] = i % 10;
    for (i=0 ; i<PROCESS ; i++)
    {
        if (fork() == 0)
        {
            pere = 0;
            break;
        }
    }
    if (pere)
    {
        for (i = 0; i < PROCESS; i++)
            wait(NULL);
        puts("Calculs terminés");
    }
    else
    {
        int somme = 0;
        for(int j=i*25; j < (i+1)*25; j++)
            somme += tab[j];
        printf("%d %d\n",i,somme);
    }
    return 0;
}
```

- Donnez l'arbre des processus
- Ce programme est-il déterministe ? Justifiez
- Si le programme est déterministe:
  - Indiquez exactement ce qui sera affiché lors de son exécution.
- Si le programme n'est pas déterministe:
  - Indiquez un des affichages possibles

Quelles conclusions pouvez-vous tirer de ces trois cas ?

Exercice 6:

Variante 1

Écrivez un programme qui crée 4 fils. Le premier fils affiche les entiers de 1 à 50, le seconde de 51 à 100, le troisième de 101 à 150 et le dernier de 151 à 200

Variante 2

Faites en sorte que l'affichage soit garanti d'être systématiquement dans l'ordre numérique croissant 1,2,3,... , 200.

Programmation Concurrente: Fork et Exec

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Fork & Exec

Variante 1

Dans un shell & placer en fin de commande signifie qu'elle sera exécutée en arrière plan et qu'une ou plusieurs autres pourront être exécutées simultanément.

```
du -sh /usr/share & ls -l & ps
```

Écrivez le programme (simultane.c) qui correspond exactement à la suite de commande ci-dessus

Variante 2

Dans un shell ; placer en fin de commande signifie que les commandes sont exécutées successivement.

```
du -sh /usr/share ; ls -l ; ps
```

Écrivez le programme (successif.c) qui correspond exactement à la suite de commande ci-dessus

Exercice 2: Make

Écrivez un programme (make.c) qui prend en arguments une série de fichiers sources .c, les compile chacun séparément et simultanément puis fait l'édition de lien en utilisant le dernier argument pour nommer l'exécutable.

Si on détaille un peu.

Le programme:

- lance un fils pour chaque fichier passé en argument qui exécute *gcc -c*
  - les warnings ne sont pas des erreurs mais doivent être quand même utilisé et affiché
- attends et vérifie l'état de chacun des fils
  - si l'un des fils se termine en erreur (erreur de compilation), il indique une erreur et le fichier concerné et s'arrête
- Si tous les fils se sont terminé sans erreur, le père réalise l'édition de lien à partir de tout les fichiers objet .o pour générer l'exécutable avec comme nom, le dernier argument

Sur e-campus, vous trouverez une archive compressée tar.gz qui contient un certain nombre de fichier .c .h sur lequel tester votre programme.

Vous avez deux dossiers:

- **success** qui contient des exemples qui compilent avec et sans warning
- **failure** qui contient des exemples qui produisent des erreurs à différents niveaux

Exercice 3: Génération de miniature à partir d'un dossier

Écrivez un programme (miniature.c) qui prend le chemin d'un dossier en argument et qui redimensionne toutes les images au sein de ce dossier à 10% de leur taille originale. Le nombre de processus de conversion fonctionnant en même temps ne doit jamais excéder **huit**.

Vous avez sur e-campus une archive compresser (tar.gz) comprenant quarante photos et un fichier texte vous permettant de tester facilement votre travail.

Aide

Convert

Pour faire cela, vous avez comme utilitaire convert (ImageMagick)

```
user@machine:~$ convert image/img.jpg -resize 10% thumb/img.jpg
```

Parcourir un dossier

Un bref code qui illustre comment lire le contenu d'un répertoire nommé image.

```c
//open directory
DIR* directory = opendir(path);
//get one entry of the directory
struct dirent* entry = readdir(directory);
//close directory
closedir(directory);
```

entry est un pointeur vers une structure qui contient entre autre deux champs:

- d_name qui est le nom du fichier
- d_type qui est le type du fichier:
  - DT_DIR répertoire
  - DT_REG pour un fichier standard

⚠ d_name ne contient **que** le nom du fichier il faudra donc le concaténer avec le chemin du dossier pour avoir le chemin complet et valide du fichier.

Exercice 4: Téléchargements multiples

Écrivez un programme (telecharge.c) qui prend en arguments une liste de fichier et un répertoire de destination. Le programme récupère les fichiers en parallèle et les enregistrent dans le dossier de destination (dernier argument). Le nombre de processus de conversion fonctionnant en même temps ne doit jamais excéder **quatre**. Le programme indique le début du téléchargement du fichier ainsi que son pid et lorsque celui-ci est fini.

Utilisez wget comme programme pour la partie téléchargement de fichier, n'hésitez pas à regarder les options de celui-ci qui vous faciliteront la vie.

Une liste de fichier que vous pouvez utiliser en arguments et des explications se trouvent dans l'archive compressés.

Programmation Concurrente: Tubes Anonymes

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.
- Pensez à fermer les extrémités inutilisées de vos tubes

Exercice 1: Premiers Tubes

Écrivez un programme (guess) dont le processus père demande a l'utilisateur un nombre et le transmet à l'aide d'un pipe à son processus fils.

Le processus fils génère un nombre et compare celui-ci au nombre reçu de son père. Si le nombre est plus petit que celui généré, il répond au père dans un second pipe **+**, si le nombre est plus grand **-** et s'il est identique **=**

Les deux processus se terminent lorsque le bon nombre a été trouvé par l'utilisateur.

Exercice 2: Tubes et Redirection

Lors des TP d'administration système vous avez déjà utilisé les pipes et redirection. Dans cet exercice, vous allez à l'aide de pipe et de redirection l'équivalent d'une série de commandes. Le but n'est pas de réécrire ces commandes mais d'utiliser celle existante dans vos programmes.

Variante 1: Redirection Simple

Écrivez un programme dont le code correspond à la commande suivante:

```
wc < /etc/passwd
```

Variante 2: Pipe Simple

Écrivez un programme dont le code correspond à la commande suivante:

```
cut -f 1 -d : /etc/passwd | tr a-z A-Z
```

Variante 3: Pipeline Complet

Écrivez un programme dont le code correspond à la commande suivante:

```
cut -f 1,3 -d : < /etc/passwd  | sed 's+^\(.*\):\(.*\)+\2:\1+' | sort -n > users
```

⚠ cut n'a pas connaissance ici qu'il manipule le fichier /etc/passwd contrairement au cas précédent. Il en va de même pour sort qui ne sait pas qu'il écrit dans un fichier nommé users.

Exercice 3: Filtres

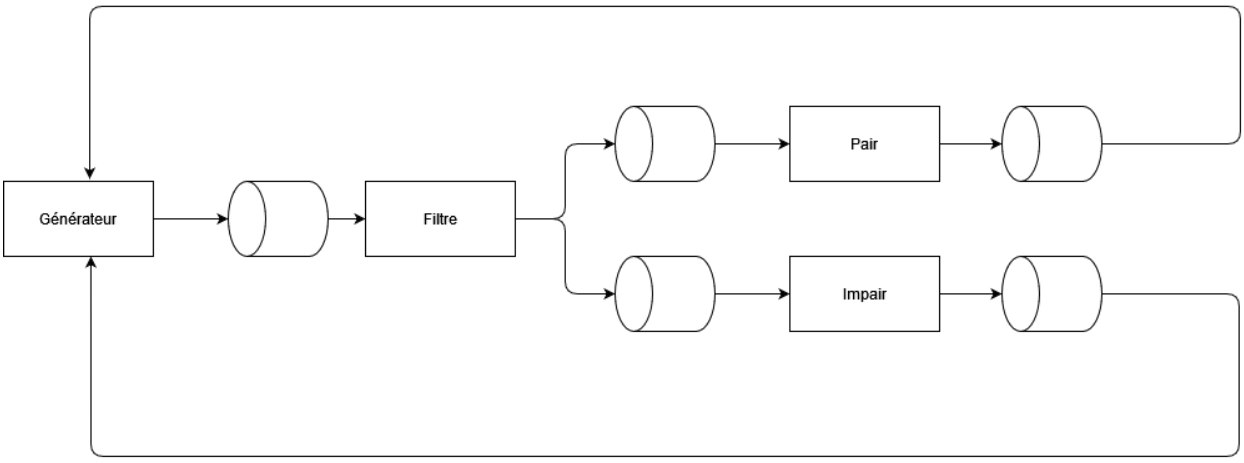
Écrivez un programme (somme) composé de 4 processus réalisant les traitements suivants (voir Schéma)

Un premier processus (générateur), génère N nombres aléatoires positifs ou nuls. Il dépose chacun de ses nombres dans un tube. Une fois la génération terminée, il envoie -1 dans le tube. Il se met alors en attente des sommes issus des processus pair et impair.

Un second processus (filtre) récupère les différents nombre depuis le tube et en s'ils sont pairs ou impairs les déposes respectivement dans un tube ou un autre. Il dépose -1 dans chacun des tubes.

Un troisième processus (pair) récupère les nombres pairs reçus depuis le tube et les somme. Une fois le dernier nombre reçu, il envoie la somme totale au processus générateur.

Un quatrième processus (impair) récupère les nombres impairs reçus depuis le tube et les somme. Une fois le dernier nombre reçu, il envoie la somme totale au processus générateur.



Programmation Concurrente: Tubes Anonymes Suite

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.
- Pensez à fermer les extrémités inutilisées de vos tubes

Exercice 1: Plus grand nombre

Écrivez un programme (greater) créant N processus. Chacun de ces N processus génère et affiche un nombre tiré aléatoirement. Le nombre de processus est spécifié en argument. Chacun des processus créé est identifié par un pid et un numéro d'ordre de création.

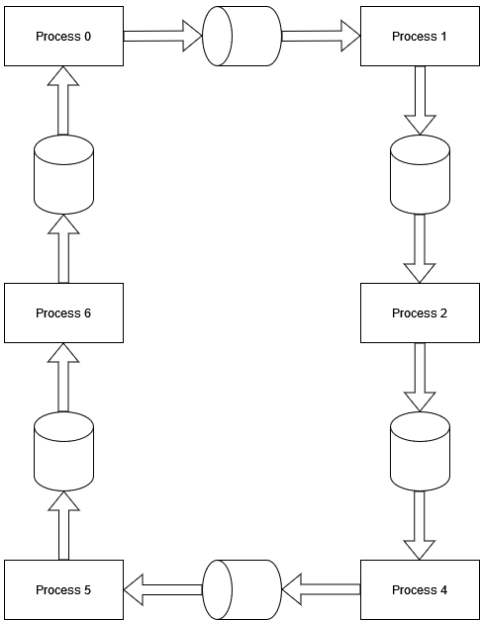
L'objectif est de déterminer le processus qui a généré le plus grand nombre. Pour cela, les N processus sont organisés en anneau au travers de leur entrée et de leur sortie standard en utilisant les tubes anonymes. Le processus 0 va être connecté au processus 1 par un premier tube, le processus 1 est connecté au processus 2 par un deuxième tube, le processus 2 connecté au processus 3 et ainsi de suite... Le dernier processus de l'anneau doit être connecté au processus 0.

Lorsque les processus et les tubes sont créés, le processus 0 envoie sa valeur générée, son identifiant et son numéro d'ordre vers le processus 1.

Le processus 1 compare la valeur reçue à sa valeur et envoie au processus 2 la valeur la plus grande.

- Si la valeur la plus grande était celle reçue de 0, alors il transmet l'identifiant et le numéro d'ordre reçu par le tube.
- Si c'était celle produite par le processus 1, alors il transmet sa valeur générée, son identifiant et son numéro d'ordre.

Chaque processus de l'anneau va appliquer le même traitement. À la fin, le processus 0 doit recevoir sur son tube la plus grande valeur, l'identifiant du processus l'ayant généré et son numéro d'ordre.



Exemple d'exécution:

```
user@machine:~$ ./greater 6
processus pid 4232 numéro 2 val = 1430
processus pid 4233 numéro 3 val = 485
processus pid 4234 numéro 4 val = 220
processus pid 4235 numéro 5 val = 2602
processus pid 4230 numéro 0 val = 2178
processus pid 4231 numéro 1 val = 1723
Le plus grand nombre = 2602 - pid = 4235 - Numéro 5
```

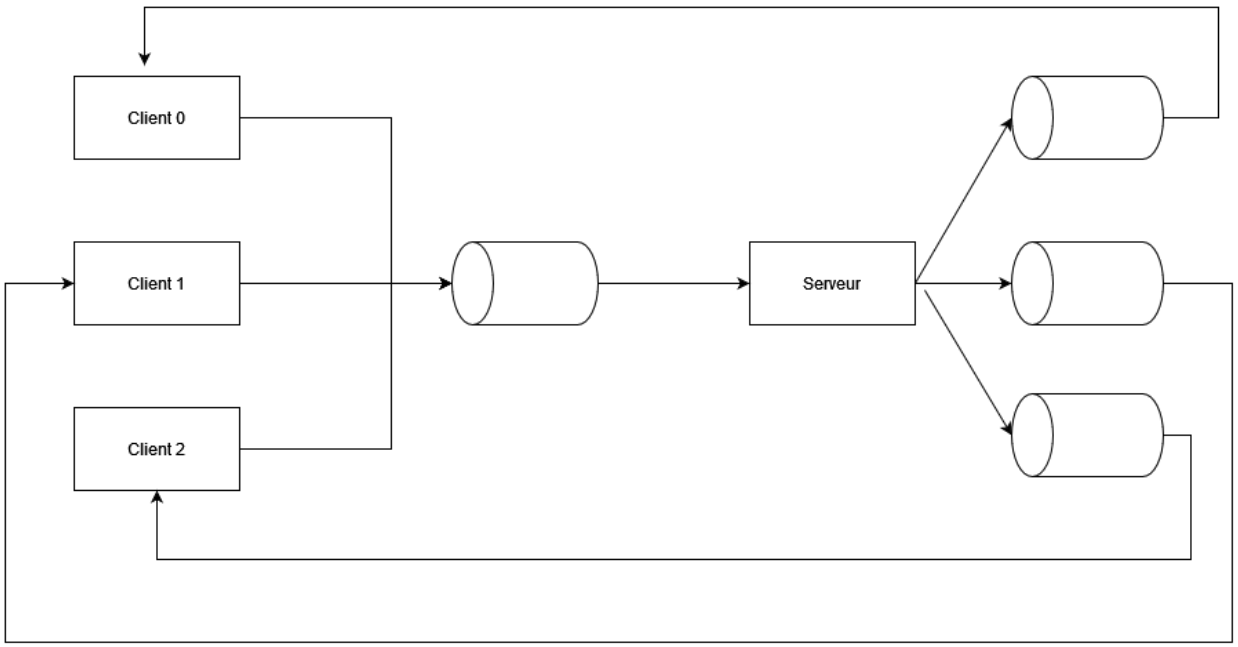
Exercice 2: Calculateur

Écrivez un programme (calculator) qui est composé d'un seul serveur qui reçoit et traite les requêtes envoyées par différents clients. Le nombre de client est fourni au programme en argument. Après traitement, le serveur transmet les résultats aux clients. Il faut permettre à plusieurs clients d'envoyer des requêtes au serveur. Le service réalisé par le serveur est d´évaluer des expressions arithmétiques simples de type a + b (ou a - b).

Par exemple, si le client envoie la requête 4+5, le serveur évalue cette expression et retourne le résultat 9 au client concerné. Vous pouvez utiliser les pipes pour réaliser cette application :

- Un tube pour établir la communication avec le serveur (sens client vers le serveur).
- Un tube par client pour que le serveur puisse retourner le résultat de l'évaluation.

Le serveur lance les N clients et crée les pipes nécessaires pour communiquer avec les clients. La requête d'un client doit contenir un identifiant qui permet de retrouver facilement le tube qui lui est associé.



Vous pouvez utiliser les structures suivante pour représenter une requête et une réponse :

```
struct requete_client_serveur {
    int clientPid; //PID du Client
    int nombre1;
    int nombre2
    char operator;
};

struct resultat_client_serveur {
    int nombre1;
    int nombre2;
    int resultat;
    char operator;
};
```

Programmation Concurrente: Semaphore

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.
- Vous aurez besoin de 2 programmes:
  - Init qui se chargera de créer les sémaphores nécessaires à l'exercice avec le nombre de jeton utile
  - Clean qui se chargera de supprimer les sémaphores une fois l'exécution de vos programmes terminés
- ⚠ Pensez à utiliser Clean puis Init avant chacun de vos tests.
- En plus, d'Init et Clean, vous aurez besoin au minimum de deux programmes différents pour chaque exercice.
- Varier l'ordre de lancement de vos programmes pour vérifier qu'ils satisfont les consignes.

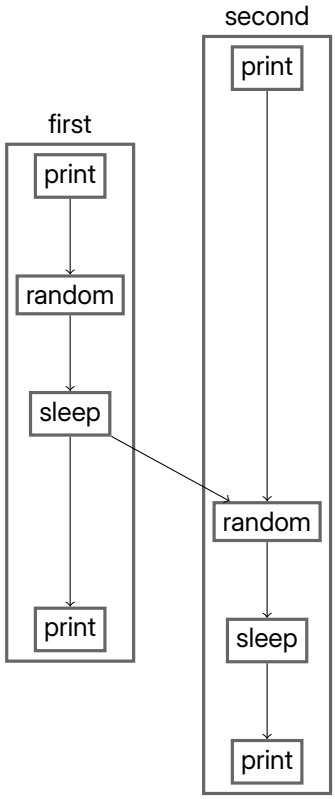
Exercice 1: Précédence

Écrivez deux programmes (first et second), chacun d'eux réalise un affichage au lancement.

first génère un nombre entre 3 et 5 et attends autant de seconde que le nombre généré. Il affiche ensuite un message au moment de se terminer.

second doit être en attente jusqu'à ce que first ait fini son attente. Il génère ensuite un nombre entre 1 et 4 et attends autant de seconde que le nombre généré. Il affiche ensuite un message au moment de se terminer.

Aucun lien de parenté n'existe entre first et second. Peu importe à quel moment et comment sont lancé les programmes, second doit être en attente que first ait fini son attente.



Exercice 2: Rendez-vous

Plusieurs processus souhaitent établir un rendez-vous. Chaque processus exécutent certain traitement mais ne peuvent continuer leur exécution que si l'ensemble des processus sont arrivés à la fin de leur traitement respectif. Une fois, cela fait, chacun reprend son exécution indépendamment des autres. L'ordre d'exécution de chaque process, le moment d'exécution des programmes ou le temps d'exécution de ceux-ci ne doit pas influencer le rendez-vous. Tant que tous les processus ne sont pas au rendez-vous, les autres doivent attendre.

Variante 1: Rendez-vous à deux

Écrivez deux programmes souhaitant établir un rendez-vous. Vous simulerez les traitements des différents programmes à l'aide de **sleep** et vous aiderez de **puts**/**printf** afin de suivre aisément les différentes étapes rencontrées par chacun des processus.

Variante 2: Rendez-vous à trois

Écrivez trois programmes souhaitant établir un rendez-vous. Vous simulerez les traitements des différents programmes à l'aide de **sleep** et vous aiderez de **puts**/**printf** afin de suivre aisément les différentes étapes rencontrées par chacun des processus.

Question

Existe-t-il un moyen de généraliser le rendez-vous quelque-soit le nombre de process attendu aux rendez-vous avec ce que l'on a vu jusqu'à présent dans le module ? Si je dois avoir N processus a un rendez-vous, N étant connu de tout les processus au moment de leur exécution. Si ce n'est pas le cas, que vous faudrait-il pour cela ?

Exercice 3: Rendez-vous 1 à 2

Le principe est similaire à l'exercice précédent. La différence est que nous avons deux types de processus différents:

- Un émetteur qui produit un message et attends que deux processus récepteurs soient présents avant de leur transmettre (affichage dans la console)
- Un récepteur attends qu'il y ait un émetteur et un autre récepteur avant de se terminer

Exemple d'exécution (après les # se trouve un suivi des processus au moment du démarrage du processus)

```
user@machine:~ $ ./recepteur& # 1 recepteur(23015) 0 emetteur
    Recepteur 23015 démarre
    Recepteur 23015 en attente
user@machine:~ $ ./emetteur& # 1 recepteur(23015) 1 emetteur(23021)
    Emmetteur 23021 démarre
    Emmetteur 23021 en attente premier recepteur
    Emmetteur 23021 en attente deuxième recepteur
user@machine:~ $ ./emetteur& # 1 recepteur(23015) 2 emetteur(23021, 23028)
    Emmetteur 23028 démarre
user@machine:~ $ ./recepteur& # 2 recepteur(23015, 23037) 2 emetteur(23021, 23028)
    Recepteur 23037 démarre
    Recepteur 23037 en attente
    Emmetteur 23021 libéré
    Recepteur 23015 libéré
    Recepteur 23037 libéré
    Emmetteur 23028 en attente premier recepteur
user@machine:~ $ ./recepteur& # 1 recepteur(23042) 1 emetteur(23028)
    Recepteur 23042 démarre
    Recepteur 23042 en attente
    Emmetteur 23028 en attente deuxième recepteur
user@machine:~ $ ./recepteur& # 2 recepteur(23042, 23047) 1 emetteur(23028)
    Recepteur 23047 démarre
    Recepteur 23047 en attente
    Emmetteur 23028 libéré
    Recepteur 23047 libéré
    Recepteur 23042 libéré
```

Un premier récepteur est lancé (23015) puis deux émetteurs (23021 et 23028), tous sont en attente, car la condition un émetteur et deux récepteurs n'est pas satisfaite. On lance ensuite un nouveau récepteur (23037) qui de fait libère 23015 et 23021, laissant seul l'émetteur (23028). En lançant successivement deux récepteurs ensuite (23042 et 23047) nous libérons l'émetteur qui restait en attente.

Même si votre affichage peut varier, faites attention à garantir que le lancement du second récepteur libère bien le premier émetteur et ne fasse pas avancer l'exécution du second qui doit attendre ses propres récepteurs.

Programmation Concurrente: Mémoire Partagée

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.
- Vous aurez besoin de 2 programmes:
  - Init qui se chargera de créer les sémaphores nécessaires à l'exercice avec le nombre de jeton utile
  - Clean qui se chargera de supprimer les sémaphores une fois l'exécution de vos programmes terminés
- ⚠ Pensez à utiliser Clean puis Init avant chacun de vos tests.
- En plus, d'Init et Clean, vous aurez besoin au minimum de deux programmes différents pour chaque exercice.
- Varier l'ordre de lancement de vos programmes pour vérifier qu'ils satisfont les consignes.

Exercice 1: Rendez-vous à N

Écrivez un programme souhaitant établir un rendez-vous avec N version de lui-même. N est un argument reçu sur la ligne de commande. Vous simulerez les traitements des différents processus à l'aide de **sleep** et vous aiderez de **puts**/**prints** afin de suivre aisément les différentes étapes rencontrées par chacun des processus. ⚠ Tous les processus reçoivent le même N.

Exercice 2: Producteur/Consommateur

Un processus prod produit des informations qui sont consommées par un autre processus conso. La communication entre les processus se fait par l'intermédiaire d'un segment de mémoire partagée nommé buffer de taille N fixée suivant les règles suivantes :

- prod dépose des informations dans le buffer. Ce processus doit attendre si buffer est plein.
- conso récupère les informations stockées dans buffer. Ce processus doit attendre si buffer est vide.
- Le segment mémoire buffer est une ressource critique (partagée par plusieurs processus).
- Les informations sont consommées dans l'ordre où elles sont produites [Premier Arrivé/Premier Consommé - FIFO].
- Il peut y avoir plusieurs prod et conso partageant la ressource buffer.

Exercice 3: Échange de valeur

Deux processus P1 et P2 disposant chacun d'un tableau stockant N entiers. P1 et P2 veulent pouvoir s'échanger des nombres de leur tableau de manière à ce que, à la fin de leur exécution réciproque, P1 dispose des N plus petites valeurs et P2 des N plus grandes.

À l'aide de sémaphore, d'une mémoire partager permettant de stocker 3 entiers, le fonctionnement sera le suivant:

- P1 cherche le maximum dans son tableau et l'écrit dans la 1ère case de la mémoire partagée.

- P2 cherche le minimum dans son tableau et l'écrit dans la 2ème case de la mémoire partagée.
- P0 vérifie que la valeur de la première case est supérieure à la valeur de la seconde case. Si c'est le cas, P0 effectue l'échange des valeurs, sinon il met la valeur -1 dans la troisième case de la mémoire partagée.
- Chaque processus (P1 et P2) récupère sa nouvelle valeur dans la case respective, l'écrit à la place de l'ancienne, puis recommence le même travail.

Si la valeur de la troisième case est égale à -1, les processus n'effectuent aucun échange. Ils affichent leur tableau et s'arrêtent.

Écrivez les programmes de P0, P1 et P2.

Programmation Concurrente: Thread

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Rendez-vous à N

Écrivez un programme créant N thread souhaitant établir un rendez-vous entre eux. N est un argument reçu sur la ligne de commande. Vous simulerez les traitements des différents thread à l'aide de **sleep** et vous aiderez de **puts**/**prints** afin de suivre aisément les différentes étapes rencontrées par chacun des threads.

Variante 1

Réaliser le programme sans utilisation de barrière

Varriante 2

Réaliser le programme avec barrière

Exercice 2: Producteur/Consommateur

Des threads prod produisent des informations qui sont consommées par d'autres threads conso. La communication entre les threads se fait par l'intermédiaire d'un buffer partagé de taille N fixée suivant les règles suivantes :

- prod dépose des informations dans le buffer. Ce thread doit attendre si buffer est plein.
- conso récupère les informations stockées dans buffer. Ce thread doit attendre si buffer est vide.
- Le buffer est une ressource critique (partagée par plusieurs thread).
- Les informations sont consommées dans l'ordre où elles sont produites [Premier Arrivé/Premier Consommé - FIFO].

Exercice 3: Échange de valeur

Deux threads T1 et T2 disposant chacun d'un tableau stockant N entiers. T1 et T2 veulent pouvoir s'échanger des nombres de leur tableau de manière à ce que, à la fin de leur exécution réciproque, T1 dispose des N plus petites valeurs et T2 des N plus grandes.

À l'aide de sémaphore, d'une mémoire partagée permettant de stocker 3 entiers, le fonctionnement sera le suivant:

- T1 cherche le maximum dans son tableau et l'écrit dans la 1ère case de la mémoire partagée.

- T2 cherche le minimum dans son tableau et l'écrit dans la 2ème case de la mémoire partagée.
- T0 vérifie que la valeur de la première case est supérieure à la valeur de la seconde case. Si c'est le cas, T0 effectue l'échange des valeurs, sinon il met la valeur -1 dans la troisième case de la mémoire partagée.
- Chaque thread (T1 et T2) récupère sa nouvelle valeur dans la case respective, l'écrit à la place de l'ancienne, puis recommence le même travail.

Si la valeur de la troisième case est égale à -1, les threads n'effectuent aucun échange. Ils affichent leur tableau et s'arrêtent.

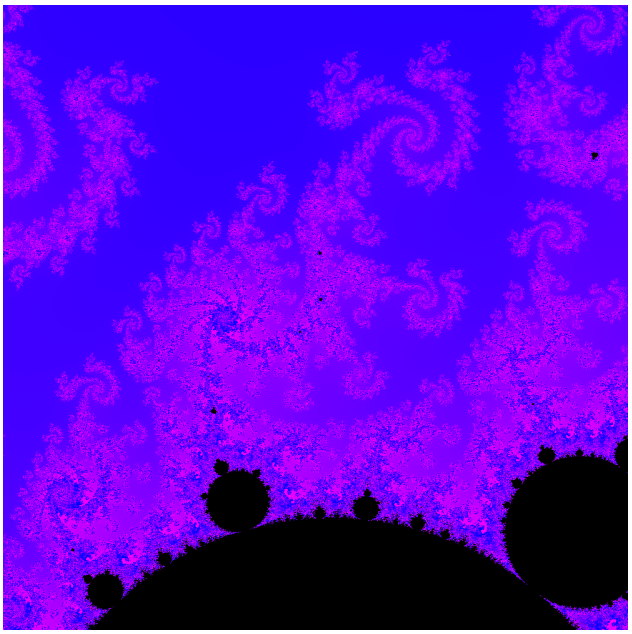
Écrivez le programme réalisant ce travail.

Exercice 4: Fractale

Modifier le programme mono-thread factal.c pour illustrer l'utilisation d'un ensemble de N threads pour la génération de l'image stockant la fractale de Mandelbrot. Chaque thread s'occupera du calcul de quelques lignes de l'image. Le nombre N est donné en ligne de commande.

La fonction calcul: `static void calcul(int x, int y, unsigned char *pixel)` ne doit pas être modifié. Elle calcule les informations du pixel de coordonnées **x**,**y** et stocke le résultat triplet (R,G,B) a l'adresse **pixel**

⚠ Attention : les parties noires [(R,G,B) proches de (0,0,0)] de la fractale correspondent à des calculs plus longs.



La commande de calcul du temps processeur : time

Les informations affichées par la commande time sont :

- real : temps réel d'exécution du programme
- user : temps processeur consommé par le programme
- sys : temps processeur passé dans les appels système.

Systèmes d'Exploitation && Programmation Concurrente
Pipeline graphique

Travaux Pratiques Threads & Barrières

1 Présentation

Les cartes graphiques modernes utilisées dans nos ordinateurs, tablettes ou smart-phones sont organisées en *pipeline* graphique.

Un pipeline est une suite d'étages permettant de réaliser une tâche complexe en la découpant en sous-tâches plus ou moins indépendantes.

Dans le cas d'un pipeline graphique, l'idée est de confier les différentes étapes de transformation des données graphiques à différentes unités de calcul.

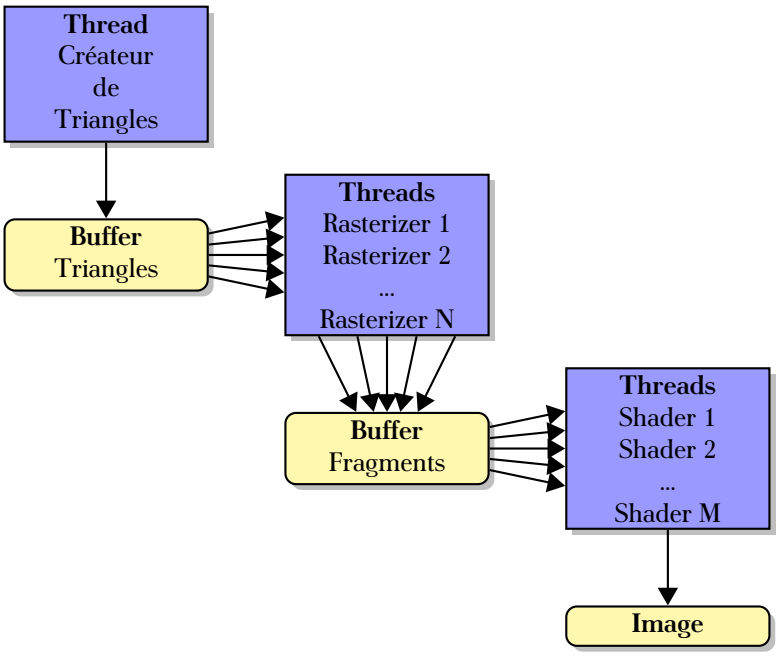


Figure 1 – Notre pipeline simplifié

Pour le TP d'aujourd'hui, nous allons simuler le pipeline simplifié présenté sur la figure 1.

Ce pipeline est composé de trois étages et de trois zones de mémoires. Chaque étage sera composée d'un ou plusieurs *threads*.

Le premier étage (`triangle_stage`), composé d'un seul thread, aura pour rôle d'analyser une scène 3D, de la découper en triangles, puis de stocker ces triangles dans un buffer de `Triangles`. Ce thread unique se comportera donc comme un *producteur*. Les triangles seront récupérés un à un à l'aide de la fonction `triangle_creator_get_next_triangle()` présentée ci-après.

Le deuxième étage (`raster_stage`) sera composé de plusieurs threads qui viendront *consommer* des triangles depuis le buffer de triangles précédents. Chacun de ces triangles sera ensuite découpé en fragments d'images comportant une position dans l'image ainsi que d'autres informations. Ces fragments seront obtenus un à un à l'aide de la fonction `raster_get_next_fragment()` présentée ci-après. Les fragments devront être déposés dans une zone de mémoire partagée (un buffer tournant). Le deuxième étage est donc également producteur de fragments.

Le troisième étage (`shader_stage`) sera composé de plusieurs threads qui viendront consommer des fragments pour les stocker dans l'image finale.

L'image est créée et sauvegardée par le premier étage.

Le programme complet comporte donc deux ensembles de producteurs / consommateurs : un pour les triangles, et un pour les fragments.

2 Les trois étages

Bien entendu, il ne vous est pas demandé de créer toutes les fonctions du pipeline, mais uniquement de les utiliser dans des threads qui devront implémenter des mécanismes de synchronisations via des mutex, des sémaphores ou des barrières.

Les fonctions manipulant les données graphiques sont déclarées dans le fichier `gpu.h` et implémentées dans le fichier `gpu.c`. **Ne modifiez pas ces fichiers !** Cependant, vous devrez utiliser ces fonctions.

Le but du programme est de créer une image (`image.ppm`) à partir d'une scène 3D. Cette scène est partiellement stockée dans le fichier `elephant.obj`.

2.1 Le créateur de triangles

Le rôle du thread (il n'y en a qu'un dans notre modèle) « créateur de triangles » est de récupérer des triangles et de les produire dans le tableau `triangles`. Il est implémenté dans la fonction `triangle_stage()` que vous devrez compléter.

La fonction suivante :

```
1 void *triangle_creator_init();
```

permet d'initialiser le créateur de triangles, de charger la scène et d'effacer l'image. La valeur renvoyée permet d'identifier le créateur et doit être passée comme premier paramètre aux deux fonctions suivantes.

Une fois l'initialisation faite, la fonction suivante permet de récupérer chacun des triangles qui composent la scène.

```
1 int triangle_creator_get_next_triangle(void *creator, struct Triangle *triangle);
```

Le premier paramètre doit être ce qui est renvoyé par `triangle_creator_init()`, le second contient l'adresse d'une variable de type `struct Triangle` qui sera remplie par la fonction. La valeur renvoyée par cette fonction est 1 si le triangle a été correctement créé et 0 sinon. La valeur zéro signifie donc que la scène est terminée et que le thread créateur de triangles peut se terminer. Toutefois la fonction suivante doit être appelée pour libérer la mémoire et sauvegarder l'image sur le disque :

```
1 void triangle_creator_destroy(void *creator);
```

Enfin, la fonction suivante permet de savoir si un triangle est réellement un triangle ou une demande de synchronisation.

```
1 int triangle_is_sync(struct Triangle *triangle);
```

En effet, de temps en temps (lors d'un changement de couleur dans notre cas), il est nécessaire de synchroniser tous les étages, et donc de vider (consommer) les buffers. Cela permet d'éviter que les triangles et fragments qui restent lors d'un changement de couleurs ne soient dessinés avec la nouvelle couleur (voir figure 2).

2.2 Les rasteriseurs

L'étage des rasteriseurs a pour rôle de transformer les triangles (en les consommant) en fragments d'image (en les produisant). Un fragment contient les informations nécessaire à un *shader* (l'étage suivant) pour déterminer la couleur d'un pixel de l'image. Il est implémenté dans la fonction `raster_stage` que vous devrez compléter.

La fonction suivante permet d'initialiser la rasterisation d'un triangle (passé en paramètre) :

```
1 void *raster_init(const struct Triangle *tri);
```

La valeur renvoyée doit être utilisée comme premier paramètre de la fonction suivante (dont le fonctionnement rappelle celui de la fonction `triangle_creator_get_next_triangle()`)

```
1 int raster_get_next_fragment(void *raster_state, struct Fragment *fragment);
```

Une fois un triangle complétement traité, on doit libérer la mémoire qu'il a utilisé en appelant la fonction suivante :

```
1 void raster_destroy(void *raster_state);
```

Lors de la synchronisation, nous allons devoir créer des fragments de synchronisation (des faux fragments) à l'aide de la fonction suivante :

```
1 struct Fragment fragment_sync(void);
```

Enfin, la fonction suivante permet de savoir si un fragment est un réellement un fragment ou une demande de synchronisation.

```
1 int fragment_is_sync(struct Fragment *fragment);
```

2.3 Les shaders

Le dernier étage de notre pipeline est implémenté par la fonction `shader_stage`. Son rôle est de consommer des fragments pour les mettre dans l'image.

Pour cela, on appelera la fonction suivante qui placera le fragment au bon endroit, calculera sa couleur exacte, etc. :

```
1 void gpu_shader(struct Fragment fragment);
```

3 Travail demandé

Vous ne devrez modifier que le fichier `pipeline-graphic.c`. Le travail demandé est en trois temps :
- comprendre le programme (il n'est pas nécessaire de comprendre ce qu'il y a dans `gpu.c`)
- implémenter les producteurs/consommateurs de triangles et fragments dans les différents étages
- implémenter les synchronisations à l'aide de barrières

3.1 producteurs / consommateurs

Pour cette partie, il vous faudra compléter le code dans les trois fonctions des threads. Essayez de bien comprendre qui produit quoi et qui consomme quoi.

Une fois ceci fait, vous devez obtenir une image du genre de celle de la figure 2.

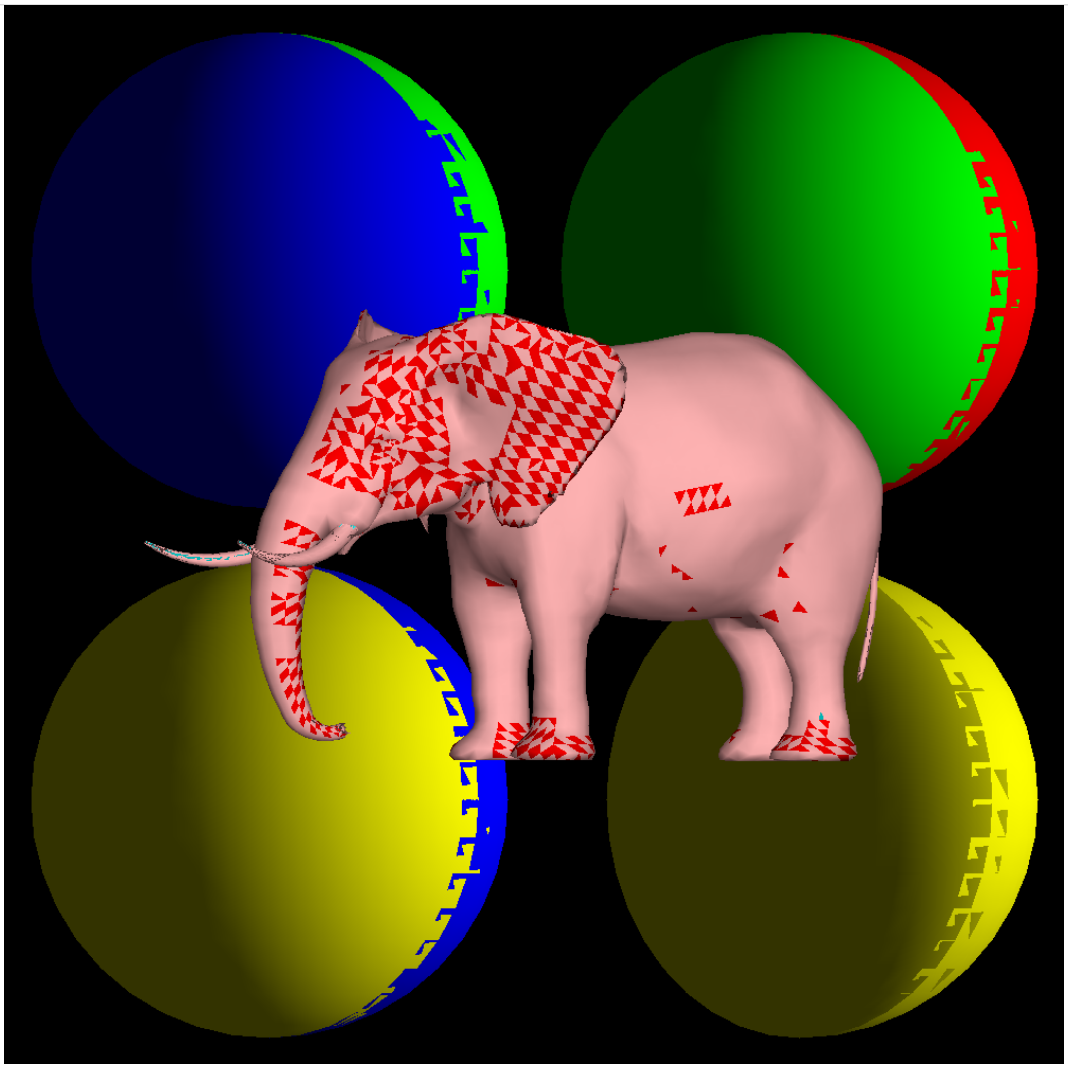


FIGURE 2 – Notre scène avec des problèmes de synchronisation des étages.

3.2 Barrières

Afin d'éviter les problèmes de la figure 2, il faut implémenter des synchronisations. En effet, lorsqu'un changement de couleur intervient, un triangle particulier, dit "de synchronisation" est créé. Cela permet au thread créateur de triangles de détecter qu'il doit attendre que les autres threads finissent leur travail.

Pour cela, vous utiliserez les fonctions suivantes :
— `pthread_barrier_init`
— `pthread_barrier_wait`
— `pthread_barrier_destroy`
Vous devriez alors obtenir une image identique à celle de la figure 3.

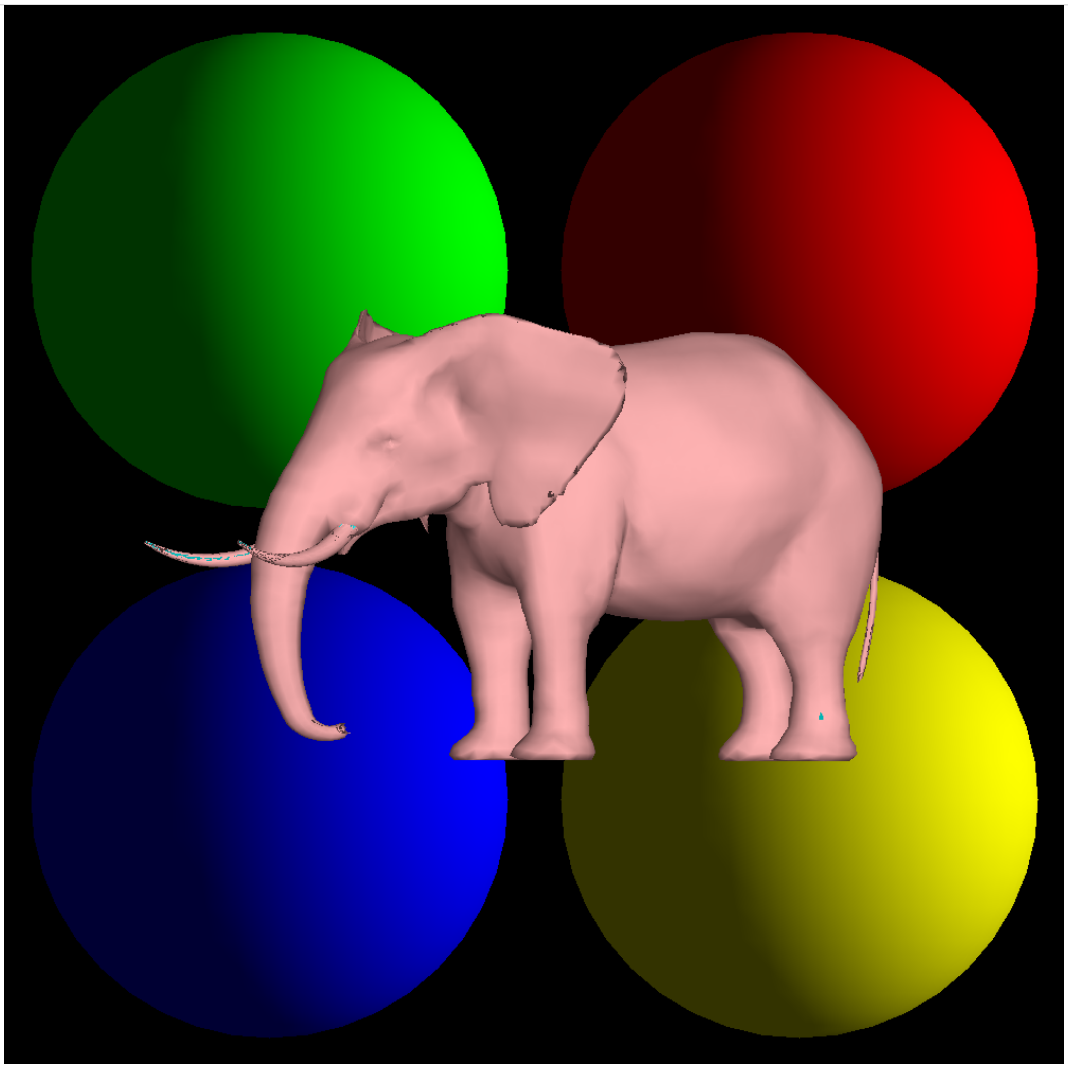


FIGURE 3 – Notre scène finale

Programmation Concurrente: Thread (Suite)

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Gestionnaire de Billes

On souhaite réaliser l'exemple suivant:

- **N** thread (par ex. N = 4) ont besoin chacun d'un nombre **k** d'une ressource (par ex. des Billes) pour avancer leur travail
- Cette ressource existe en un ***nombre limité*** : on ne peut satisfaire la demande de tout le monde en même temps. Par exemple, la demande des N (N=4) est de (4, 3, 5, 2) billes et on ne dispose que de **nb_max_billes** = 9 billes
- Chaque Processus répète la séquence (par ex. m fois) : "demander k ressources, utiliser ressources, rendre k ressources"
- Le "main" crée les N thread. Il crée également un thread contrôleur qui vérifie en permanence si le nombre de Billes disponible est dans l'intervalle [0..nb_max_billes] afin de détecter d'éventuelles erreurs.
- Pour chaque Thread T_i, l'accès à la ressource se fait par une fonction "demander(k)" qui doit bloquer le demandeur tant que le nombre de billes disponible est inférieur à k
- T_i, rend les k billes acquises après son travail et recommence sa séquence

Algorithme

Main:

```
Initialiser les ressources partagées
Créer Contrôleur
Lancer Contrôleur
Créer N thread travailleurs
Lancer ces N thread
Attendre la fin des N Processus
Terminer le Contrôleur
```

Travailleur:

```
itérer m fois
    Demander(k_billes)
    simuler un travaille avec une attente aléatoire (sleep)
    Rendre(k_billes)
```

Demander:

```
Tant que nbr_billes_disponible < k_billes:
    se bloquer sur un semaphore d'attente
nbr_billes_disponible = nbr_billes_disponible - k_billes
```

Rendre:

```
nbr_billes_disponible = nbr_billes_disponible - k_billes
réveiller ceux qui sont en attente (cf Demander)
```

Contrôleur:

```
Iterer toujours
    Verifer que 0 <= nbr_billes_disponible <= max_billes
    delai(1 sec)
```

Remarques

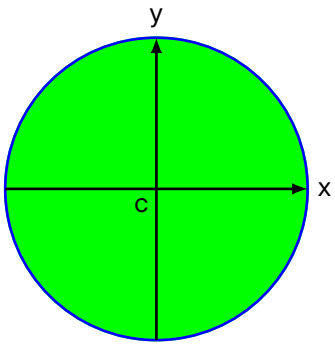
nbr_billes_disponible est une ressource partagée il faudra donc évidemment protéger les accès en lecture et en écriture.

Quand les thread en attente sont réveillés, ça ne veut pas dire que le nombre de billes dont ils ont besoin sont disponibles mais simplement qu'un thread a posé des billes et qu'il est possible qu'ils en aient suffisamment pour travailler. Si ce n'est pas le cas, ils vont se remettre en attente.

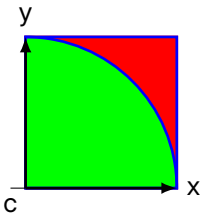
L'usage du semaphore pour l'attente est là pour garantir une attente passive. Il ne sert à rien de regarder le nombre de billes disponible tant qu'un thread n'en a pas rendu.

Exercice 2: Estimation de Pi

L'air d'un cercle de rayon R est $\pi * R^2$. Si on prend un Rayon de 1, l'air est donc π. En prenant un cercle de centre 0,0 et de Rayon 1, il est possible d'approximer π.

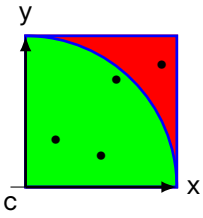


Si on s'intéresse à l'intervalle [0,1] on a donc un quart de cercle et donc une surface de $\frac{\pi}{4}$



La surface du carré (surface verte + surface rouge) est de 1. La surface verte est $\frac{\pi}{4}$.

Si on tire aléatoirement un couple (x,y) $\in [0.0, 1.0]$. Si $x^2 + y^2 \leq 1^2$ (équation de cercle) alors le point est dans le cercle (zone verte), sinon en dehors (zone rouge). La probabilité que l'on soit dans le quart de cercle est de $\frac{\pi}{4}$.



Si on répète l'opération un nombre N (grand) de fois, le rapport entre le nombre de point étant dans le quart de cercle (hit) et le total de points testés (N) approxime $\frac{1}{4}$ de la surface du cercle unitaire.

On a donc $\pi \approx 4 * \frac{hit}{N}$

Ce type d'approche est appelé méthode de Monte-Carlo, ici appliquée a π a l'aide du cercle unitaire.

Variante 1: Programmez cette méthode en mono-thread

Variante 2: Programmez cette méthode en multi-thread

Comparez les temps de calculs entre les deux méthodes. Prenez un N suffisamment grand par exemple N=1 000 000

Programmation Concurrente: Signaux

CPE Lyon

2024

Consignes générales:

- Vos programmes doivent être compilés avec les options: -Wall -Wextra -g
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Interception de signal

Écrire un programme qui réalise un affichage dans une boucle infinie, mais qui prévoit de s'arrêter à la réception du signal **SIGINT**. La fonction d'interception affichera un message signalant la réception du signal avant de terminer le programme par un appel **exit()**.

Après avoir vérifier que votre programme marchait correctement. Lancez-le en tâche de fond ./exo1 & et essayer de l'interrompre avec contrôle+C (^C). Que constatez-vous?

Exercice 2: Ignorer un signal

Modifier le programme précédent pour qu'il ignore le signal **SIGINT**.

Pour l'arrêter, il faut taper `ps -aux | grep exo2` pour obtenir son numéro de processus (pid) et lancer la commande : `kill -KILL pid` (pour envoyer le signal **SIGKILL**)

Exercice 3: Arrêt propre

Modifiez le programme précédent pour qu'il fasse son affichage dans une boucle conditionnée par une variable booléenne fin initialisée à faux et qui sera mise à vrai par la fonction d'interception du signal.

Exercice 4: Communication père/fils

Ecrire un programme composé de 2 processus : Le père fait des affichages toutes les secondes dans une boucle for et le fils fait des affichages toutes les secondes aussi mais dans une boucle infinie, quand le compteur de boucle du père arrive à 3, le père envoie un signal **SIGKILL** au fils.

Exercice 5: Contrôle+C et hierarchie

Recopier le programme précédent et le modifier pour que le père n'envoie plus de signal au fils mais que le fils intercepte tous les **SIGINT** en affichant un message d'interception.

Exécutez alors le programme et interrompez le par un ^C : Que constatez-vous? Expliquez pourquoi.

Exercice 6: SIGUSR1/SIGUSR2

Reprendre le programme de l'exercice précédent et le modifier pour que le fils ne fasse ses affichages qu'à la réception du signal **SIGUSR1**. Le père envoie ce signal dans son itération 3 et dans son itération 5. Il signale la fin du traitement au fils par envoi du signal **SIGUSR2**. Il n'y aura qu'une seule fonction d'interception dans le fils, elle recevra le numéro du signal déclencheur en paramètre.

Exercice 7: Timeout

Ecrire un programme qui demande à l'utilisateur de taper au clavier un entier en moins de 5 secondes. Pour cela, votre programme ne doit pas "planter" si ce qu'il lit n'est pas un entier. Il devra donc lire une chaîne et tenter de la convertir en un entier, si c'est bon il se terminera après avoir désarmé le **timeout**, sinon il recommencera éventuellement jusqu'aux 5 secondes où il affichera **"trop tard"** avant de s'arrêter.

Exemple de sortie écran possible:

```
$./exercice8
Entrez un entier en moins de 5 secondes
Svp un entier : a
Svp un entier : x
Svp un entier : 12
Ok merci !!
```

```
Entrez un entier en moins de 5 secondes
Svp un entier : E
Svp un entier : f
Svp un entier : D
Trop tard !!
```

Programmation Concurrente: Multiprocessing Python

CPE Lyon

2024

Consignes générales:

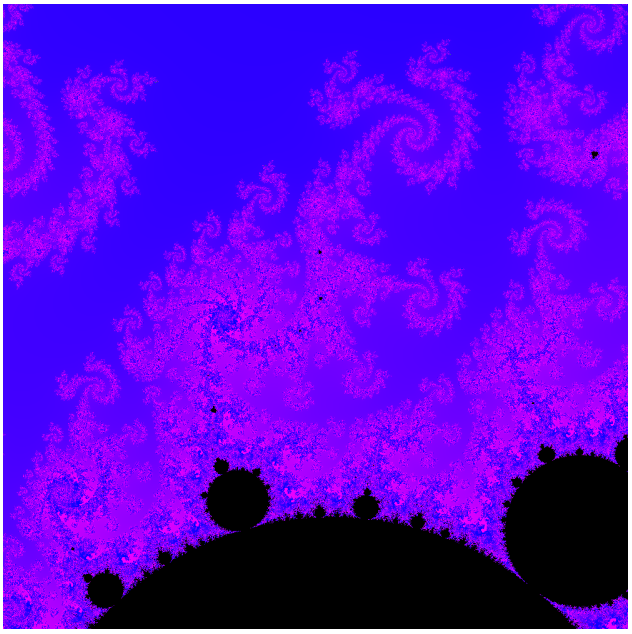
- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Fractale

Modifier le programme mono-thread factal.py pour illustrer l'utilisation d'un ensemble de N process pour la génération de l'image stockant la fractale de Mandelbrot. Chaque process s'occupera du calcul de quelques lignes de l'image. Le nombre N est donné en ligne de commande.

La fonction calcul: `def calcul(x, y, image, pixel_index)` ne doit pas être modifié. Elle calcule les informations du pixel de coordonnées **x**,**y** et stocke le résultat triplet (R,G,B) a l'index **pixel_index** de l'image

⚠ Attention : les parties noires [(R,G,B) proches de (0,0,0)] de la fractale correspondent à des calculs plus longs.



La commande de calcul du temps processeur : time

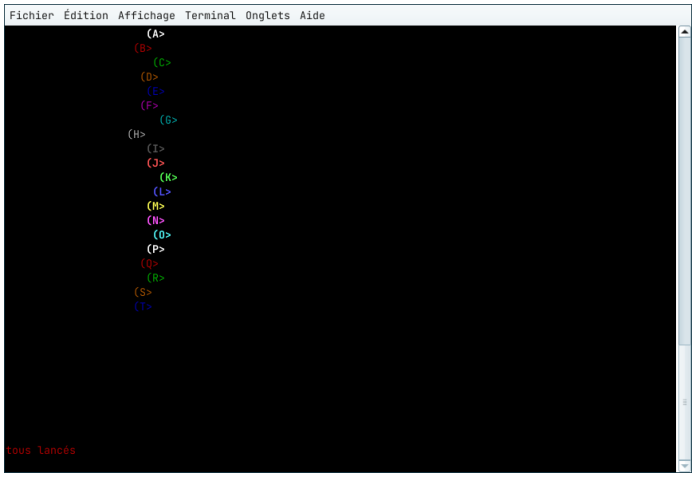
Les informations affichées par la commande time sont :

- real : temps réel d'exécution du programme
- user : temps processeur consommé par le programme
- sys : temps processeur passé dans les appels système.

Exercice 2: Course Hippique

On souhaite réaliser, sur les machine Linux, une course hippique. Sur e-campus, vous avez course-hippique.py, gérant une course hippique en multiprocessing. Néanmoins, aucun mécanisme de synchronisation n'existe actuellement.

Chaque cheval est représenté par une lettre 'A' à 'T' ⇒ (A> est le premier cheval. Chaque cheval occupe une ligne de l'écran et la progression de chacun est affichée. L'avancement de chacun est indépendante et aléatoire.



Travail à réaliser

Après avoir pris connaissance de la structure du code et de son fonctionnement vous réaliserez ce qui suit.

Resource partagée

L'écran est une ressource critique où chaque processus peut apporter des modifications en même temps qu'un autre. Les affichages dans le code actuel ne sont pas en exclusion mutuelle. Vous devrez donc protéger ces affichages pour garantir que chaque process n'est pas interrompu lors d'une opération.

Arbitre

Ajouter un processus Arbitre qui aura la charge d'afficher en permanence le cheval qui est en tête, ainsi que celui qui est dernier.

Pari

Avant le début de la course, ajoutez la possibilité de parier sur le cheval gagnant.

Amélioration esthetique

Un cheval est représenté par (A>, en utilisant des caractères affichable, changer l'affichage des chevaux. Par exemple:

```
 _______\/
/|__A__/\/
 / \ / \
```

⚠ Le dessin dans ce cas est sur plusieurs lignes. Donc chaque cheval occupera plusieurs lignes de l'écran.

Programmation Concurrente: Multiprocessing Python

CPE Lyon

2024

Consignes générales:

- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Robot

Réaliser le système suivant dans une version concurrente avec les mécanismes de base graphique (screen comme dans la course Hippique).

Un robot a les caractéristiques suivants * Pas de but particulier : avancer et éviter les obstacles * Plusieurs capteurs : infra rouge (IR) sur les 2 côtés, sonar (US) frontal, de contact (Bumper) frontal * Les actions sur les servo moteurs : avancer, reculer, tourner à gauche/droite * Le comportement par défaut est : avancer * Un écran d'affichage de l'état

Principe de chaque tâche

Controlleur:

```
Répéter toutes les X secondes
    Commande = "avancer"
    Drapeau = faux
    Si (Drapeau_IR) Alors
        Commande = Cmd_IR
        Drapeau = Drapeau_IR
    Si (Drapeau_US) Alors
        Commande = Cmd_US
        Drapeau = Drapeau_US
    Si (Drapeau_BU) Alors
        Commande = Cmd_BU
        Drapeau = Drapeau_BU
    Transmettre Commande aux servos
    verrouiller(Ver) ;
    mem_Cmd = Commande
    mem_Flag = Drapeau
    libérer(Ver) ;
Fin Répéter
```

Ecran:

```
Répéter toutes les A secondes
    verrouiller(Ver) ;
    C = mem_Cmd
    F = mem_F lag
```

```
    libérer(Ver) ;
    écrire C et F
Fin Répéter
```

Infrarouge:

```
Répéter toutes les S secondes
    lire la valeur Vg sur le capteur gauche
    lire la valeur Vd sur le capteur droit
    Convertir_AN(Vg,Dg)
    Convertir_AN(Vd,Dd)
    Si Dg < d OU Dd < d Alors
        Drapeau_IR = vrai
        Si Dg < d ET Dd < d Alors
            Cmd_IR = "reculer"
        Sinon Si Dg < d Alors
            Cmd_IR = "à gauche"
        Sinon
            Cmd_IR = "à droite"
    Sinon
        Drapeau_IR = faux
Fin Répéte
```

Ultrason:

```
Répéter toutes les K secondes
    lire la valeur V sur le capteur
    Convertir_AN(V ,D)
    Si D < d Alors
        Drapeau_US = vrai
        Cmd_US = "reculer"
    Sinon
        Drapeau_US = faux
Fin Répéter
```

Bumper:

```
Répéter toutes les Z secondes (Z petit)
    Si contact=1 Alors
        Drapeau_BU = vrai
        Cmd_BU = "reculer"
    Sinon
        Drapeau_BU = faux
Fin Répéte
```

Remarques

- lire la valeur V sur le capteur X et Convertir_AN sont des cas pratique lecture de la tension du capteur et conversion de la valeur analogique en numérique ⇒ Ici on simule ça en calculant la valeur a partir de nos données (mur présent ou non a une distance x)
- La tâche écran a pour but de dessiner le parcours du robot et d'afficher l'état des commandes et flags pour aider au debug
- Vous mettrez en place tout mécanisme de synchronisation entre les tâches nécessaire

Exercice 2: Jeu de la vie

Réaliser le jeu suivant dans une version concurrente avec les mécanismes de base graphique (screen comme dans la course Hippique).

Il s'agit d'une grille (matrice de taille d'au moins 15×15) dont les cases représentent soit un "être" vivant soit rien.

L'état d'une case peut être modifié en fonction de son voisinage selon les règles décrites ci-dessous.

Extrait de l'énoncé d'origine :

- The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead.
- Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur :
  - Any live cell with fewer than two live neighbours dies, as if caused by under-population.
  - Any live cell with two or three live neighbours lives on to the next generation.
  - Any live cell with more than three live neighbours dies, as if by overcrowding.
  - Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.
- The initial pattern constitutes the seed of the system.
- The first generation is created by applying the above rules simultaneously to every cell in the seed-births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one).
- The rules continue to be applied repeatedly to create further generations.

Programmation Concurrente: Multiprocessing Python

CPE Lyon

2024

Consignes générales:

- Vos programmes ne doivent pas avoir de warnings
- Testez plusieurs fois de suite vos programmes et vérifiez que les résultats obtenus sont ceux attendus.

Exercice 1: Gestionnaire de Billes

On souhaite réaliser l'exemple suivant:

- **N** processus (par ex. N = 4) ont besoin chacun d'un nombre **k** d'une ressource (par ex. des Billes) pour avancer leur travail
- Cette ressource existe en un ***nombre limité*** : on ne peut satisfaire la demande de tout le monde en même temps. Par exemple, la demande des N (N=4) est de (4, 3, 5, 2) billes et on ne dispose que de **nb_max_billes** = 9 billes
- Chaque Processus répète la séquence (par ex. m fois) : "demander k ressources, utiliser ressources, rendre k ressources"
- Le "main" crée les N processus. Il crée également un processus contrôleur qui vérifie en permanence si le nombre de Billes disponible est dans l'intervalle [0..nb_max_billes]
- Pour chaque Processus P_i, l'accès à la ressource se fait par une fonction "demander(k)" qui doit bloquer le demandeur tant que le nombre de billes disponible est inférieur à k
- P_i, rend les k billes acquises après son travail et recommence sa séquence

Algorithme

Main:

```
Créer Contrôleur
Lancer Contrôleur
Créer N processus travailleurs
Lancer ces N processus
Attendre la fin des N Processus
Terminer le Contrôleur
```

Travailleur:

```
itérer m fois
    Demander(k_billes)
    simuler un travaille avec une attente aléatoire (sleep)
    Rendre(k_billes)
```

Demander:

```
Tant que nbr_billes_disponible < k_billes:
    se bloquer sur un semaphore d'attente
nbr_billes_disponible = nbr_billes_disponible - k_billes
```

Rendre:

```
nbr_billes_disponible = nbr_billes_disponible - k_billes
réveiller ceux qui sont en attente (cf Demander)
```

Contrôleur:

```
Iterer toujours
    Verifer que 0 <= nbr_billes_disponible <= max_billes
    delai(1 sec)
```

Remarques

nbr_billes_disponible est une ressource partagée il faudra donc évidemment protéger les accès en lecture et en écriture.

Quand les processus en attente sont réveillés, ça ne veut pas dire que le nombre de billes dont ils ont besoin sont disponibles mais simplement qu'un processus a posé des billes et qu'il est possible qu'ils en aient suffisamment pour travailler. Si ce n'est pas le cas, ils vont se remettre en attente.

L'usage du semaphore pour l'attente est là pour garantir une attente passive. Il ne sert à rien de regarder le nombre de billes disponible tant qu'un processus n'en a pas rendu.

Exercice 2: Serveur de calculs

On souhaite réaliser des calculs en parallèle. Le programme est découpé de sorte d'avoir des processus demandeurs et des processus calculateurs. Les processus demandeurs déposent dans une file de message (multiprocessing.Pipe, opération send) une demande de calcul: par exemple le demandeur n°1 veut faire l'opération 3 + 5 ⇒ `pipe.send((1,'+',3,5))`. Un processus calculateur extrait le calcul du pipe (operation recv), effectue le calcul et écrit la réponse dans une deuxième file de message.

Variante 1: 1 demandeur et n calculateurs

Dans cette variante, deux pipes suffisent. Un pour les échanges demandeur ⇒ calculateurs et un pour les échanges calculateurs ⇒ demandeur.

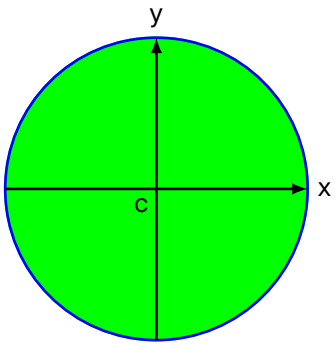
Variante 2: m demandeurs et n calculateurs

Dans cette variante, il va falloir réfléchir au moyen de communication nécessaire entre les différents processus.

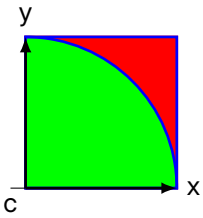
- Combien de pipe ai-je besoin pour la communication demandeurs ⇒ calculateurs?
- Comment identifier un demandeur?
- Combien de pipe ai-je besoin pour la communication calculateurs ⇒ demandeurs?
- Comment être sûr que le demandeur à reçu le résultat qu'il demande?
- Ai-je besoin de mécanisme de synchronisation entre les calculateurs ou les demandeurs?
- ...

Exercice 3: Estimation de Pi

L'air d'un cercle de rayon R est $\pi * R^2$. Si on prend un Rayon de 1, l'air est donc π. En prenant un cercle de centre 0,0 et de Rayon 1, il est possible d'approximer π.

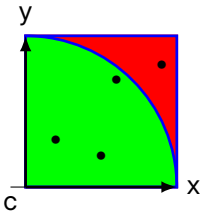


Si on s'intéresse à l'intervalle [0,1] on a donc un quart de cercle et donc une surface de $\frac{\pi}{4}$



La surface du carré (surface verte + surface rouge) est de 1. La surface verte est $\frac{\pi}{4}$.

Si on tire aléatoirement un couple (x,y) $\in [0.0, 1.0]$. Si $x^2 + y^2 \leq 1^2$ (équation de cercle) alors le point est dans le cercle (zone verte), sinon en dehors (zone rouge). La probabilité que l'on soit dans le quart de cercle est de $\frac{\pi}{4}$.



Si on répète l'opération un nombre N (grand) de fois, le rapport entre le nombre de point étant dans le quart de cercle (hit) et le total de points testés (N) approxime $\frac{1}{4}$ de la surface du cercle unitaire.

On a donc $\pi \approx 4 * \frac{hit}{N}$

Ce type d'approche est appelé méthode de Monte-Carlo, ici appliquée a π a l'aide du cercle unitaire.

Variante 1: Programmez cette méthode en monoprocess

Variante 2: Programmez cette méthode en multiprocess

Comparez les temps de calculs entre les deux méthodes. Prenez un N suffisamment grand par exemple N=1 000 000

Programmation Concurrente: Contrôle TP

CPE Lyon

2023

Exercice 1

Le programme sepia.c présent dans l'archive sepia.tar.gz est actuellement un programme mono-thread qui transforme le fichier image.ppm et génère le fichier sepia.ppm. Vous n'avez besoin de comprendre que le fonctionnement de la fonction main, les autres fonctions s'occupent de divers traitements mais ne sont pas votre préoccupation.

La fonction **calcul** s'occupe de calculer un pixel indépendant de notre image.

Elle reçoit: image: l'image d'origine out_image: l'image de sortie width: largeur des deux images height: hauteur des deux images x: coordonnée x du pixel y: coordonnée y du pixel

Consignes:

Vous devez transformer le programme sepia.c pour qu'il utilise **4 thread** afin d'accélérer la vitesse de traitement. Vous mettrez en place tous les mécanismes d'échanges et de synchronisations que vous jugerez nécessaires à la réalisation de cet objectif.

Vous ne devez écrire ou modifier du code que dans les sections marquées a cet effet.

Les deux zones sont délimitées par: ***/—— Zone à Modifier ——/*** et ***/—— Fin de Zone à Modifier ——/***

Attention, il ne doit pas contenir de warning ou d'erreur de compilation.

Info utiles:

```
convert sepia.ppm sepia.jpg
```

Cette commande vous permet de convertir l'image du format ppm au format jpg afin de la visualiser plus facilement

Rappel:

```
time ./programme
```

vous donne 3 temps: * **real**: temps d'exécution réel * **user**: temps d'exécution en espace utilisateur * **sys**: temps d'exécution en espace noyau

Le temps qui doit changer **drastiquement** est le temps réel. user et sys seront similaires que l'on soit en mono-thread ou en multi-threads. (le temps de calcul ne change pas, ce qui change c'est le temps réel puisque l'on découpe les calculs en plusieurs threads)

Programmation Concurrente: Contrôle Final

CPE Lyon

2023

Exercice 1

Le code présent dans l'archive **prod_consor.tar.gz** présente un souci. Il devrait afficher abdefghijklmnopqrstuvwxyz mais il affiche autre chose.

Le code est découpé en 3 exécutables:

- **prod** (prod.c) qui produit des données dans une mémoire partagée.
- **conso** (conso.c) qui consomme des données depuis une mémoire partagée et les affiche.
- **launcher** (launcher.c) s'occupe de lancer prod et conso en créant les ressources utilisées par prod et conso.

Certaines variables communes sont définies dans commun.h afin de simplifier l'écriture et l'usage des différentes informations partagées entre les programmes.

Après avoir compilé, il vous suffit d'exécuter launcher pour vous rendre compte du problème.

Consignes:

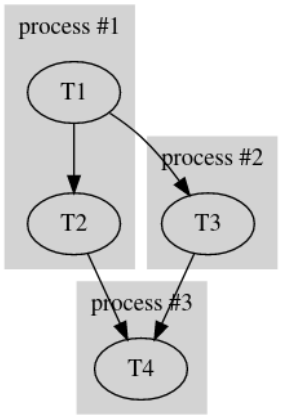
Corrigez le ou les bugs qui empêche-nt d'avoir le résultat escompté.

Exercice 2

A partir de l'archive **precede.tar.gz**

Consignes:

Sans toucher aux fonctions t1,t2,t3 et t4. Faîtes en sorte que l'enchaînement des fonctions correspondent au schéma suivant:



- La tâche T2 (p1) attend que T1 soit finie.
- La tâche T3 (p2) attend que T1 soit finie.
- Le tâche T4 (p3) attend que T2 et T3 soient finies.
- Le programme launcher s'occupera d'initialiser les ressources nécessaires et de les libérer. Il s'occupe de lancer les 3 processus et de les attendre.

Certaines variables communes peuvent être définies dans commun.h afin de simplifier l'écriture et l'usage des différentes informations partagées entre les programmes.

Après avoir compilé, il vous suffit d'exécuter launcher.

Exercice 3

Le programme flou.c présent dans l'archive **flou.tar.gz** est actuellement un programme mono-thread qui transforme les fichiers au format ppm passés en arguments et génère le fichier avec le même nom dans le dossier sortie. Vous n'avez besoin de comprendre que le fonctionnement de la fonction main, les autres fonctions s'occupent de divers traitements mais ne sont pas votre préoccupation.

La fonction **flou** s'occupe de traiter une image.

Elle reçoit:

- filepath : le chemin de l'image à traiter
- sortie : le chemin du dossier de destination

Pour l'exécuter:

```
./flou images/*
```

Consignes:

Vous devez transformer le programme flou.c pour qu'il utilise 6 threads afin d'accélérer la vitesse de traitement. Vous mettrez en place tous les mécanismes d'échanges et de synchronisations que vous jugerez nécessaires pour la réalisation de cet objectif.

Vous ne devez écrire ou modifier du code que dans les sections marquées à cet effet.

Les deux zones sont délimitées par: **/*—— Zone à Modifier ——*/** et **/*—— Fin de Zone à Modifier ——*/**

Rappel:

```
time ./programme
```

vous donne 3 temps:

- **real**: temps d'exécution réel
- **user**: temps d'exécution en espace utilisateur
- **sys**: temps d'exécution en espace noyau

Le temps qui doit changer **drastiquement** est le temps réel. user et sys seront similaires que l'on soit en mono-thread ou en multi-threads. (le temps de calcul ne change pas, ce qui change c'est le temps réel puisque l'on découpe les calculs en plusieurs threads)

Exercice 4

Écrivez un programme alarme.c dont le fonctionnement est le suivant:

- Le processus crée un fils
- Le processus fils affiche toutes les secondes un message jusqu'à ce qu'il reçoive le signal **SIGUSR1**. Après réception du signal, il se termine normalement.
- Le processus père lorsqu'il reçoit le signal **SIGALRM** envoie un signal **SIGUSR1** à son fils.
- Le processus père attend 10 secondes avant de recevoir le signal **SIGALRM**. Pour cette attente, vous utiliserez la fonction **pause()** (man 2 pause).

Exercice 5

Écrivez un programme pipeline.c qui correspond à l'enchaînement des commandes suivantes:

```
grep "invalid credentials" < server.log | cut -d, -f3 | sort > sortie
```

Vous avez à votre disposition le fichier **server.log** pour réaliser cela.