Université de Strasbourg

S31 Sujet N°3
Comprendre les Systèmes
gancarski@unistra.fr

IUT Robert Schuman Université de Strasbourg

Processus légers en C

Objectif : Comprendre les threads

Notions : Création de processus légers. Variables globales. Synchronisation par signaux

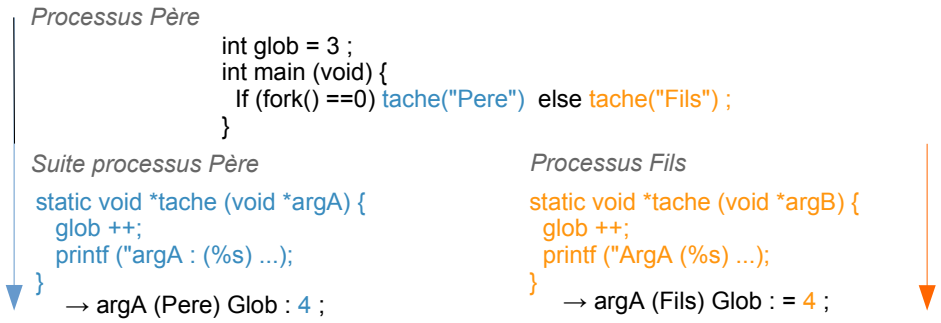
1 Processus légers et synchronisation

1.1 Présentation

Rappel : Un processus classique (dit lourd) dispose d'**un seul chemin d'exécution** : les instructions de son programme sont exécutées à la suite → Compteur ordinal.
A chaque duplication (`fork`) un nouvel espace d'adressage et un nouveau compteur ordinal sont créés.

```
int glob = 3;
static void *tache (void *argA) {
  glob ++;
  printf ("argA : (%s) Glob : %d\n",argA, glob);
}
main{
  if(fork() == 0)  { tache("Fils"); }
  else { tache("Pere"); }
}
```

Nous donne :

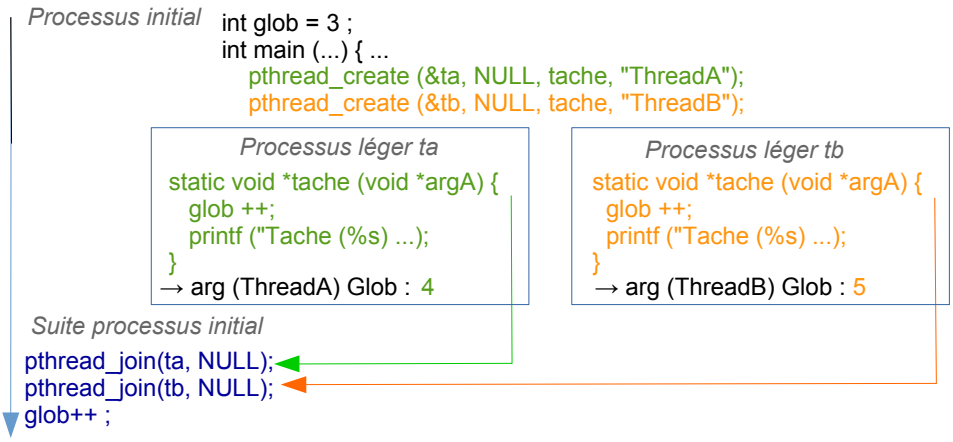


Les processus légers (threads) sont des "sous-processus" à l'intérieur d'un processus lourd dont chacun avec son propre chemin d'exécution : ces sous-processus s'exécutent en "parallèle"

Modifions le programme précédent pour avoir 2 processus légers "fils" :

```
int glob = 3;
int main(int argc, char *argv[]){
        pthread_t ta = 0, tb = 0;
  pthread_create (&ta, NULL, tache, "A");
  pthread_create (&ta, NULL, tache, "B");
  pthread_join(ta, NULL);
  pthread_join(tb, NULL);
  glob++;
}
```

Le processus lourd va créer (via `pthread_create`) deux processus légers, chacun exécutant une fonction qui peut être identique ou différente. Puis il attend leurs fins (via `pthread_join`) avant de continuer



1.2 Caractéristiques et avantages des processus légers

Un processus léger :

- Ne peut exister qu'au sein d'un processus lourd
- Partage les ressources du processus lourd : code, mémoire physique, fichiers, droits Unix, environnement de shell, etc.
- Exécute un sous-programme (procédure ou fonction).
- Dispose d'une pile privée pour ses données locales,
- Partage les données globales avec les autres processus légers,
- E/S indépendantes de celles des autres processus légers.

Les avantages :

- Permet d'avoir plusieurs chemins d'exécution
- Autorise des calculs différents
- Une opération bloquante ne bloque que le processus appelant
- PL restent "liés" : simplifie la synchronisation et l'ordonnancement → signaux interne : (voir section suivante)
- Nombre important de PL : décomposition du problème à grain très fin grain
  → permet l'utilisation plus optimale des architectures multi-coeurs (voir Cours "Conclusion")
- Partage d'information simplifiée par mémoire globale

Ordonnancement : Les processus légers sont vus comme des processus. Donc sont ordonnancés comme tels. Mais 1) le processus lourd peut définir des priorités "internes" et 2) la priorité dynamique du processus dépend aussi des temps d'exécution de ses processus légers.

1.1 - Combien y'a-t-il de chemins d'exécution ?

1.2 - Quelle est la valeur de `Glob` à la fin du processus loud

1.2 - Que se passerait-il si le père ne faisait pas de `pthread_join`

1.3 - Dans quel ordre s'exécutent les threads ?

1.4 - Comment pourrait-on faire pour "garantir" que `ta` incrémente `glob` en premier. Il existe au moins 3 solutions potentielles différentes ... Essayez de les trouver et comparez les .

2 Mise en œuvre (TP)

Implanter les trois solutions proposées à l'exercice précédent.

3 Serveur de calcul

Encore plus de savoir accumulé pour produire une première version V1b.1 du serveur de calcul à base de variables globales.

4 Processus légers et arguments

La fonction de création `pthread_create` demande une variable de type `void *` pour l'argument passé à la fonction principale (le "main") du processus léger. Par exemple, dans le code initial : `pthread_create (&ta, NULL, tache, (void *)"A");` on est obligé de transtyper la chaîne de caractères. Pourquoi ?

Supposons que nous décidions de passer un deuxième paramètre à la fonction appelée à la création du thread. Par exemple, on désire que les processus créés connaissent leur ordre de création. Deux solutions :

1. Utiliser une variable globale incrémentée avant chaque `pthread_create`. Le processus léger créé va simplement recopier sa valeur dans une variable locale. Pourquoi cela n'est-il pas souhaitable ?
2. Regrouper les paramètres dans une seule variable : une structure, un tableau ... Et passer l'adresse de cette variable comme paramètre de la fonction principale (main) du processus léger.

Un exemple concret

— On définit une structure qui contiendra tous les arguments nécessaires aux threads

```
struct ArgThread {
  char * name;
  int num;
};
```

— On modifie la fonction appelée à la création des threads

```
static void *tache (void *ArgTh) {
   char * argA = ((struct ArgThread *)ArgTh)->name;
   int num =( (struct ArgThread *)ArgTh)->num;

   glob ++;
   printf ("Tache (%s) Num %d Glob %d\n",argA, num, glob);
   return "FIN";
}
```

4.1 - Pourquoi a-t-on besoin de ce transtypage `((struct ArgThread *)ArgTh)` ?

— Enfin, on modifie le main

```
int main(int argc, char *argv[]){
  pthread_t ta = 0, tb = 0;
  ...ARG1, ARG2;
```

```
  ARG1.name = "A";
  ARG1.num = 1;
  pthread_create (&ta, NULL, tache, ... ARG1);

  ARG2.name = "B";
  ARG2.num = 2;
  pthread_create (&tb, NULL, tache, ... ARG2);

  pthread_join(ta, NULL);
  pthread_join(tb, NULL);
  glob++;
  printf ("Pere (%d) %d \n", getpid(), glob );
}
```

4.2 - Pourquoi a-t-on besoin de deux variables du type `struct ArgThread` ?

4.3 - Complétez le main

5 Serveur de calcul

Et toujours plus de savoir accumulé pour produire une deuxième version V1b.2 du serveur de calcul à base de passage de paramètres

6 Processus légers et signaux

Pour synchroniser des processus légers on peut utiliser des signaux. Il existe deux mécanismes d'envoi/reception dans Linux :

- inter-processus (entre des processus lourd) : `kill(int pid, int numero_signal)`
- Intra-processus (entre des processus légers d'un processus) : `int pthread_kill (pthread_t pthread, int numero_signal);`
  Attention, lorsque l'on associe une fonction à à signal par `sigaction(SIGUSR1, &new_action, NULL);` ceci est appliqué globalement à TOUS les processus : un processus léger réagira donc exactement de la même façon au signal même si ce n'est pas lui qui a fait l'association.

6.1 Modifier le code initial pour que :
- `ta` et `tb` attend le signal SIGUSR1 pour commencer
- le processus principal envoie les deux signaux dans le bon ordre

6.2 Modifier ce nouveau code pour que le processus principal attende le signal SIGUSR1 (venant du shell) avant d'envoyer les signaux aux deux threads

6.3 Modifier ce code pour que ce soit le thread `tb` qui envoie le signal SIGUSR1 à `ta` :
- dans une première version `tb` et `tb` seront déclarés en variables globales
- *(Facultatif)* dans une deuxième version, ces deux variables restent locales au main

7 Serveur de calcul

Et maintenant, avec tout ce beau savoir accumulé vous pouvez produire une troisième version V1b.3 du serveur de calcul dans laquelle les processus légers ne sont pas tués à la fin de l'exécution de l'instruction.