Les threads

Limitation du modèle classique des processus

Un processus fils est une copie indépendante de son père.

- 🙂 Synchronisation facile : chaque processus a une copie des resources, sauf ressources système, comme entrée TFO).
  - Pas de risque d'écrasement des données de l'autre processus.
- ☹ Partage les descripteurs mais pas de la mémoire
  - Communication réduite
    - communication par tubes, tubes nommés,... OK.
    - communication par mémoire partagée difficile.
  - Communication nécessite au moins une recopie,
  - Communication nécessite des protocoles parfois compliqués.

Les threads (coprocessus/activités)

- On peut voir un processus avec plusieurs threads comme exécutant plusieurs "procédures" en parallèle.
- Thread = un flux séquentiel d'instructions dans processus.
- Chaque thread d'un processus peut s'exécuter en parallèle.
- Différence 2 processus père-fils versus 2 threads d'un même processus :

  il y a plus de partage entre les threads.

Les threads (coprocessus/activités)

- Pour un processus, le noyau maintient : *id, environnement, cwd, code, données statiques, descripteurs, tables de signaux, pile, descripteurs...
- Idéalement, un thread partage avec les autres threads du processus tout sauf
  - son identifiant (unique au sein du processus),
  - ses registres, sa pile, son compteur de programme,
  - des informations concernant son ordonnancement,
  - sa propre valeur d'`errno`,
  - son ensemble de signaux pendants et bloqués,
  - $+\cdots$,
- En particulier, partage de la mémoire (zone statique).
- Processus = environnement d'exécution
- Thread = activité au sein d'un même environnement.

Les threads

- Lors d'un changement de thread actif, le noyau recharge les registres du processeur, la pile.

$\Rightarrow$ gain potentiel à la création de thread, versus `fork()`, l'appel système de création de processus.

- Le partage mémoire facilite la communication.

☺ Partage de données importantes sans recopie.

☹ Synchronisation difficile : à n'utiliser que si utile.

Exemples d'utilisation

- Partage de données volumineuses,
- Parallélisme.
- Exemples d'algorithme qui se parallélisent bien :
  - tri fusion,
  - traitement d'images (quad-trees),
  - analyse numérique matricielle.
- Gestion des entrées utilisateurs (interfaces graphiques/programme principal),
- Entrées multiples (multiplexage)
  - un thread peut bloquer sans bloquer tout le processus
- simplification des protocoles de communication.

API threads POSIX

- Standard IEEE POSIX 1003.1c (payant).
- En-tête `#include <pthread.h>`, éventuellement bibliothèque `pthread`.
- Fonctions retournent en général 0 si OK.
- Assez grosse API ($>$ 50 fonctions).

- Lorsqu'un programme commence, il n'a qu'un seul thread.
- Idem après un `fork()` pour le fils.
  - ⚠ Attention, certaines variables (mutex) utilisées pour la synchronisation des threads sont héritées par le fils.
- Idem après un `exec*`.

Création de thread

```
int
pthread_create (pthread_t *thread_id,
                const pthread_attr_t *attr,
                void * (*routine)(void *),
                void *arg);
```

- `thread_id` : identificateur (unique pour ce processus) rempli par l'appel.
- `attr` : permet de changer les attributs (politique d'ordonnancement, si le thread est ou non joignable...).

  `NULL` $\Rightarrow$ attributs par défaut.
- `routine` : fonction de démarrage du thread.
- `arg` : argument de cette fonction.
- Le masque des signaux hérité,
- L'ensemble de signaux pendants vide.

Identification des threads

```
pthread_t pthread_self (void);
```

- Renvoie l'identificateur du thread appelant.

```
int pthread_equal (pthread_t thr1, pthread_t thr2);
```

- Renvoie vrai si et seulement si les arguments désignent le même thread.

Terminaison d'un thread

```
void pthread_exit(void *status);
```

- Termine le thread appelant en permettant à un thread faisant un `pthread_join()` de connaître la valeur de sortie `status` (sauf si le thread est détaché).
- `pthread_join()` : permet à un processus d'attendre la terminaison d'un autre thread (équivalent du `wait()` qui sera vu plus tard).
- Pas de libération des ressources du processus, sauf si dernier thread.
- Un `return x` est un `pthread_exit(x)` implicite...
- ...sauf pour le 1er thread `main()` où ça équivaut à `exit(x)`.

Destruction de thread par un autre

- Un thread peut accepter d'être terminé par un autre

```
int pthread_setcancelstate(int state, int *oldstate);
```

state soit PTHREAD_CANCEL_ENABLE, soit PTHREAD_CANCEL_DISABLE.

- Terminaison si un autre thread l'a demandé :

```
void pthread_testcancel(void);
```

- Terminaison sans test, mode asynchrone :

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,
                      NULL);
```

- Demande de terminaison du thread `thr` :

```
int pthread_cancel(pthread_t thr);
```

Détachement

- Par défaut, POSIX spécifie qu'un thread et joignable.
- Ceci signifie que les ressources allouées pour l'exécution d'un thread, comme sa pile, ne sont libérées que lorsque
  - le thread s'est terminé, et
  - un appel à `pthread_join()` pour ce thread a été effectué.
- L'appel à `pthread_join()` permet d'accéder au code retour effectué par `pthread_exit()`.

Détachement

- Par défaut, POSIX spécifie qu'un thread et joignable.
- Ceci signifie que les ressources allouées pour l'exécution d'un thread, comme sa pile, ne sont libérées que lorsque
  - le thread s'est terminé, et
  - un appel à `pthread_join()` pour ce thread a été effectué.
- L'appel à `pthread_join()` permet d'accéder au code retour effectué par `pthread_exit()`.
- Après sa création, un thread peut se détacher = devenir non joignable

```
int pthread_detach (pthread_t thread);
```

- Si on n'a pas besoin du code retour d'un thread, on peut le détacher.
- Dans ce cas, ressources libérées dès la terminaison de l'activité (et on ne peut pas attendre sa terminaison par `pthread_join()`).

Détacher dès la création : attributs

- Un threads a certains attributs, et l'un spécifie s'il est ou non joignable.
- Un thread peut décider de se détacher : il devient non joignable.
- Un thread détaché ne peut pas redevenir joignable.

Détacher dès la création : attributs

- Un threads a certains attributs, et l'un spécifie s'il est ou non joignable.
- Un thread peut décider de se détacher : il devient non joignable.
- Un thread détaché ne peut pas redevenir joignable.
- Pour créer un thread joignable ou détaché, on peut utiliser l'argument 2 de `pthread_create()`.
  - Déclarer une variable `attribut` de type `pthread_attr_t`,
  - L'initialiser : `pthread_attr_init(&attribut);`.
  - Pour créer un attribut détaché :

```
pthread_attr_setdetachstate(&attribut, PTHREAD_CREATE_DETACHED);
```

  - pour créer un attribut joignable :

```
pthread_attr_setdetachstate(&attribut, PTHREAD_CREATE_JOIGNABLE);
```

  - Libérer les ressources : `pthread_attr_destroy(&attribut);`.

Attente d'un thread

```
int pthread_join (pthread_t thr_id, void **get_status);
```

- La fonction suspend l'exeécution du thread appelant jusqu'à ce que le thread `thread_id` se termine
- Si `get_status` est non `NULL`, la valeur passée à `pthread_exit()` par le thread terminé est mémorisée à cette adresse.

Fonctions réentrantes

Fonction réentrante : fonction pouvant être utilisée par plusieurs tâches en concurrence, sans risque de corruption de données.

Exemple de fonction non réentrante.

```
int f(void)
{
  static int x = 0;
  x = 2*x+7;
  return x;
}
```

Problème :

- l'instruction `x = 2*x+7` se compile en plusieurs instructions machine.
- un thread peut être interrompu au milieu de ces instructions.
- Le problème peut se poser avec un seul thread interrompu par signaux.

Fonctions réentrantes

- Une fonction non réentrante ne peut être partagée par plusieurs processus ou threads que si on assure l'exclusion mutuelle : au plus un thread pourra exécuter son code à un moment donné.
- Une fonction réentrante peut être interrompue à tout instant et reprise plus tard sans corruption de données.

Exclusion mutuelle

- 1985–87 5 morts par irradiations massives dues à la machine Therac-25.
  Cause. Conflit d'accès aux ressources entre 2 parties logicielles.
- 2003 Panne d'électricité aux USA & Canada, General Electric.
  Cause. À nouveau : mauvaise gestion d'accès concurrents aux ressources dans un programme de surveillance.

- Les threads d'un même processus partagent le même espace d'adressage (variables globales, statiques).
- Il est nécessaire d'éviter l'accès simultané à une même variable.
- Section critique : portion de code où il est nécessaire que le thread qui l'exécute soit le seul dans cette section.

Fonctions réentrantes

Pour écrire des fonctions réentrantes

- Pas de données statiques utilisées d'un appel à l'autre.
- Pas de retour de pointeur statique : données fournies par l'appelant.
- Travail sur variables locales,
- Pas d'appel de fonctions non réentrantes
  - POSIX.1 donne une liste de fonctions réentrantes.
  - Typiquement `malloc()`, `free()`, les fonctions E/S de la `libc` ne sont pas garanties réentrantes.
- N'utiliser qu'en connaissance de cause des fonctions non réentrantes dans les programmes utilisant des signaux, ou multi-threadés.

Exclusion mutuelle

- Une fonction réentrante peut elle-même poser problème.

```
typedef struct {
  int montant_actuel;
  int interets;
} compte;

void crediter(void *argument)
{
  compte *x = (compte *)argument;

  x->montant_actuel += x->interets;
}
```

- Fonction est appelée par 2 threads indépendemment, scénario possible
  - 1er thread lit le montant actuel, disons 100, sans pouvoir compléter +=
  - 2ème thread idem.
  - 1er thread incrémente : le montant actuel vaut 110.
  - 2ème thread incrémente : le montant actuel vaut 110 au lieu de 120.

Exclusion mutuelle

- On doit interdire l'accès au corps de `crediter()` par 2 threads ou plus.
- Une section critique est une section qu'on veut être exécutable par au plus un thread à un instant donné.
- La propriété d'exclusion mutuelle pour une section critique dit qu'au plus un thread est dans cette section à un instant donné.
- Typiquement, un accès à des ressources partagées pose la question de l'exclusion mutuelle.

Exclusion mutuelle

- Il existe des protocoles
  - garantisssant l'exclusion mutuelle à une section critique,
  - en assurant que si un processus ne veut pas entrer en section critique, il ne bloque pas les autres,
  - ... mais faisant une attente active.
- Exemple : algorithme de Peterson.

```
int req[2], turn;

while(true)
{
  nc();          // section non critique
  req[i] = true;
  turn = 1-i;
  while (req[1-i] && turn != i)
      ;
  sc();      // section critique
  req[i] = 0;
}
```

Synchronisation par mutex

- La bibliothèque fournit des moyens d'assurer l'exclusion mutuelle : mutex (verrous), variables conditions, sémaphores.
- Création/destruction d'un verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
pthread_mutex_t m;
pthread_mutex_attr a;
// ici initialiser a
pthread_mutex_init(&m, const pthread_mutex_attr &a);
```

Synchronisation par mutex

- Destruction d'un verrou :

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Verrouillage (blocage si déjà verrouillé).

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- Verrouillage (échec si déjà verrouillé).

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- Déverrouillage
  - seul, le thread qui a verrouillé le mutex a le droit de le déverrouiller.
  - Si des threads sont bloqués en attente du verrou, l'un d'entre eux obtient le verrou et est débloqué (lequel ? dépend de l'ordonnancement → attributs).

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Sémaphore

IF 219
Programmation Système

Producteur / Consommateur

- Les processus qui veulent une ressource s'adressent au « guichet » d'un « entrepôt »
- S'il y a une ressource disponible, elle disparait des comptes de l'entrepôt
  - Le processus peut la prendre dans l'entrepôt
- S'il n'y a pas de ressource disponible, les processus patientent au guichet
  - Attente passive, pour économiser des ressources

Sémaphore

- Structure de données articulée autour d'un compteur
  - Modélise le nombre de ressources disponibles
  - Initialisé à une valeur positive ou nulle

- Interface POSIX
  - `sem_init()` : création et initialisation d'un sémaphore
  - `sem_destroy()` : détruit un sémaphore
  - `sem_wait()` : demande une ressource
  - `sem_post()` : rajoute une ressource

Sémaphore (2)

- Lorsqu'un thread demande une ressource :
  - Si le compteur n'est pas à zéro, il est décrémenté
  - S'il est a zéro, le thread demandeur est endormi

- Lorsqu'un thread apporte une ressource :
  - Le compteur est incrémenté
  - S'il était à zéro, les threads endormis sont réveillés
    - L'un d'eux pourra la consommer