INFO0604
PROGRAMMATION MULTI-THREADÉE

COURS 4

PROGRAMMATION MULTI-THREADÉE AVANCÉE

Pierre Delisle
Département de Mathématiques, Mécanique et Informatique
Janvier 2021

Plan de la séance

- Attributs des threads
- Attributs des mutex
- Attributs des variables de condition
- Annulation des threads

Objets « attributes »

- Jusqu'à maintenant, lors de la création de threads, mutex ou variables de condition
  - Nous avons toujours utilisé les attributs par défaut
  - Automatique pour l'initialisation statique
  - 2e paramètre à NULL pour l'initialisation dynamique
- En fait, ce 2e paramètre est un pointeur sur un objet « attributes »
  - Permet l'initialisation selon des caractéristiques spécifiques
- Liste d'arguments spécifiée lors de l'initialisation d'un objet
  - Structure « privée » : pour lire/écrire les membres de la structure, il faut utiliser des fonctions spéciales pthreads
- Les threads, mutex et variables de condition ont chacun leur propre type d'objet « attributes »
  - pthread_attr_t
  - pthread_mutexattr_t
  - pthread_condattr_t

ATTRIBUTS DES THREADS

Création d'un thread

- int pthread_create (pthread_t * thrd, pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);

- attr
  - Attributs du thread créé
  - Plutôt que passer NULL, on passe un pointeur sur une variable de type *pthread_attr_t*
- La norme POSIX définit les attributs suivants pour la création de threads
  - detachstate, stacksize, stackaddr, scope, inheritedsched, schedpolicy, schedparam
  - Pas nécessairement supportés par tous les systèmes

Thread « détaché »

- Par défaut, un thread créé est *joinable*
  - Son id peut être utilisé avec *pthread_join*
  - On peut récupérer sa valeur de retour
  - On peut le tuer (cancel)
- On peut aussi le créer détaché (detached)
  - On ne peut ni faire un join avec le thread, ni récupérer sa valeur de retour, ni le tuer
  - Lorsque le thread se termine, les ressources associées peuvent immmédiatement être récupérées par le système
- Si on est certain d'avance qu'on n'aura ni besoin de faire un join, ni de récupérer la valeur de retour, ni de le tuer
  - Il est plus efficace d'utiliser un thread détaché
- Attribut *detachstate*
  - PTHREAD_CREATE_JOINABLE
  - PTHREAD_CREATE_DETACHED
- Exemple : thread_attr.c

Thread « détaché » - Exemple

```
void *routineThread(void * arg) {
        sleep(1);
        printf("\nJe suis un thread\n");
        return NULL;
}
```

```
int main(int argc, char *argv[]) {
        pthread_t threadId;
        pthread_attr_t threadAttr;
        int statut;

        statut = pthread_attr_init(&threadAttr);

        statut = pthread_attr_setdetachstate(&threadAttr, PTHREAD_CREATE_DETACHED);

        statut = pthread_create(&threadId, &threadAttr, routineThread, NULL);

        pthread_exit(NULL);   /*Le main se terminera lorsque le thread sera termine*/

        return 0;
}
```

ATTRIBUTS DES MUTEX ET DES VARIABLES DE CONDITION

Attributs des mutex

- int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);

- Attr
  - Attributs du mutex
  - Plutôt que passer NULL, on passe un pointeur sur une variable de type pthread_mutexattr_t
- Pthreads définit les attributs suivants pour les mutex
  - pshared, protocol, prioceiling
  - Un système n'est pas obligé de les implémenter pour respecter la norme

Attributs des variables de condition

- int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *condattr);

- condattr
  - Attributs de la variable de condition
  - Plutôt que passer NULL, on passe un pointeur sur une variable de type pthread_condattr_t
- Pthreads définit l'attribut suivant pour les variables de condition
  - pshared
  - Un système n'est pas obligé de l'implémenter

ANNULATION DES THREADS

Annulation d'un thread

- La plupart du temps
  - Chaque thread s'exécute de façon indépendante
  - Effectue son travail
  - Se termine par lui-même
- Dans certains cas, un thread n'a pas besoin de terminer son travail
  - Exemple : bouton « annuler » pour une recherche
- L'annulation (cancellation) permet de « proposer » à un thread à se terminer
- Utiliser avec précaution
  - Annuler un thread entre le verrou et la libération d'un mutex ?
  - Annuler un thread alors qu'il est en train de briser un invariant du programme ?
- Pthreads permet à chaque thread de contrôler sa propre terminaison
  - Libération du mutex ou restauration de l'invariant avant de se terminer

Pthreads supporte 3 modes d'annulation

Mode	Etat de l'annulation	Type	Signification
Off	Non permis (disabled)	Peut être deferred ou asynchronous	L'annulation est ignorée jusqu'à ce qu'elle soit permise à nouveau
Retardé	Permis (enabled)	deferred	L'annulation se produira au prochain *point d'annulation*
Asynchrone	Permis (enabled)	asynchronous	L'annulation peut se produire n'importe quand

- Par défaut : Retardé

Annulation retardée

- Peut se produire seulement à des endroits spécifiques du programme
  - Points d'annulation (cancellation points)
- Certaines fonctions sont automatiquement des points d'annulation
  - condwait, lecture/écriture d'un fichier, etc...
- On peut spécifier un point d'annulation spécifique dans un programme
  - *pthread_testcancel*

Annuler un thread

- Un thread peut annuler un autre thread
  - pthread_cancel (pthread_t thread);
- L'annulation est asynchrone
  - Le retour de *pthread_cancel* ne signifie pas que le thread cible a effectivement été annulé
  - Le thread cible peut avoir seulement été « averti » qu'une annulation sur celui-ci est en attente
    - Il peut se terminer ou prendre le temps de "nettoyer avant", mais pas éviter l'annulation (sauf mode OFF)
  - Pour savoir si le thread est annulé → *pthread_join*

Annuler un thread

```
int compteur ;
```

```
void *routineThread(void * arg) {
	/*tourne indéfiniment ou jusqu'a cancellation*/
	for (compteur = 0; ; compteur++)
		if ((compteur % 1000) == 0) {
			printf("annulation du thread\n");
			pthread_testcancel(); /*si en commentaire, ne sera jamais annule*/
			/*L'annulation effective est equivalente a pthread_exit(PTHREAD_CANCELED) */
		}
}
```

```
int main(int argc, char *argv[]) {
	pthread_t threadId;
	void *result;
	int statut;
	statut = pthread_create(&threadId, NULL, routineThread, NULL);
	sleep(1);
	statut = pthread_cancel(threadId);
	statut = pthread_join(threadId, &result);
	if (result == PTHREAD_CANCELED)
		printf("Le thread a ete cancelle a l'iteration %d\n", compteur);
	else
		printf("Le thread n'a pas ete cancelle\n");
	return 0;
}
```

Annuler un thread

Modifier le type d'annulation

- int pthread_setcanceltype (int type, int *ancienType);

- Annulation retardée
  - PTHREAD_CANCEL_DEFERRED
- Annulation asynchrone
  - PTHREAD_CANCEL_ASYNCHRONOUS
  - Très risqué et délicat, à éviter si possible

Modifier l'état de l'annulation

- int pthread_setcancelstate (int etat, int *ancienEtat);

- Permettre l'annulation
  - PTHREAD_CANCEL_ENABLE
  - Défaut
- Empêcher l'annulation (jusqu'à ce que l'état soit remis à PTHREAD_CANCEL_ENABLE)
  - PTHREAD_CANCEL_DISABLE

Empêcher l'annulation

```
int compteur ;
```

```
void *routineThread(void * arg) {
	int etat;
	int statut;

	/*tourne indéfiniment ou jusqu'a cancellation*/
	for (compteur = 0; ; compteur++) {

		if ((compteur % 755 == 0)) {
			/*Si on ne change pas l'etat, le cancel sera fait au sleep a l'iteration 755*/
			/*Si on change l'etat, le thread evitera l'annulation a l'iteration 755*/
			/*et sera cancelle a l'iteration 1000*/
			statut = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &etat);
			sleep(1);
			statut = pthread_setcancelstate(etat, &etat);
		}
		else if ((compteur % 1000) == 0) {
			printf("annulation du thread\n");
			pthread_testcancel(); /*si en commentaire, ne sera jamais annule*/
		}
	}
}
```

Nettoyage à l'annulation

- Si une section de code nécessite de ramener un état « correct » avant l'annulation du thread
  - Libération de mutex verrouillés
  - Restauration d'un invariant
  - Libération de mémoire allouée dynamiquement
- … il doit utiliser une procédure de nettoyage (*Cleanup handlers*)
  - cleanup_push et cleanup_pop
- Quand un thread est annulé, pthreads dépile et appelle chaque cleanup handler
- Empilage
  - void pthread_cleanup_push (void (*routine) (void *), void *arg);
- Dépilage
  - void pthread_cleanup_pop (int execute);

```
pthread_cleanup_push(pthread_mutex_unlock, (void *) &mutex);
pthread_mutex_lock(&mutex);
/* Traitements... */
pthread_mutex_unlock(&mutex);
pthread_cleanup_pop(0);
```

- (ici le thread doit être en mode d'annulation retardé, ex. : cancel entre push et lock)

PROGRAMMATION MULTI-THREAD ENCORE PLUS AVANCÉE

Programmation pthreads

- Ce que nous avons vu est suffisant pour beaucoup de programmes
- Pthread offre d'autres fonctionnalités répondant à des besoins plus spécifiques
  - Gestion de données statiques spécifiques aux threads
  - Gestion des priorités des threads
  - Ordonnancement des threads en temps réel
  - Gestion threads/processus

Pour finir : quelques conseils pour éviter le débogage

- Attention aux habitudes et idées reçues
  - Ce qui est acceptable et même requis en programmation séquentielle traditionnelle ne l'est pas nécessairement en multi-thread
- N'oubliez pas les règles de visibilité de la mémoire !
- Un problème peut ne pas survenir sur une machine possédant un seul processeur
  - … et apparaître sur un multiprocesseur
- Ne pariez pas sur une course de threads
  - À n'importe quel endroit du code, un thread peut s'endormir pour une période indéterminée
  - Il n'existe pas d'ordre entre les threads à moins que vous ne spécifiez un ordre
  - Les threads vont s'exécuter de la façon la plus diabolique possible
- Un programme « correct » n'est pas nécessairement un programme performant
  - Et vice-versa

Pour en savoir plus

- Référence
  - David R. Butenhof, « Programming with POSIX threads »

- Beaucoup de références sur internet (attention aux sources)
  - https://computing.llnl.gov/tutorials/pthreads/
  - http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html
  - ...

PROCHAIN COURS

INTRODUCTION À LA PROGRAMMATION MULTI-THREADÉE AVEC JAVA