Université Paul Sabatier
31062 Toulouse cedex 09

Master 1 Informatique - MCPR

Programmation Parallèle
Support de travaux pratiques

Rappels sur les processus Unix (voir cours de L2 et L3)

```
int fork (void)
```

Lorsqu'un processus est **créé** par appel à la primitive fork, il est créé comme une copie presque parfaite de son « père » : il hérite de ses segments de code, de données et de pile. Mais, il en diffère aussi ; notamment par son pid, son ppid et le fait que ses segments de données et de pile ne sont **pas partagés** avec son père ou avec ses frères. En d'autres termes, une **variable** déclarée localement ou globalement, n'est **jamais partagée** entre un processus père et un processus fils.

```
void exit (int status)
```

Outre le fait qu'un processus se termine normalement lorsque la dernière instruction de son main() est exécutée, un processus peut aussi choisir de **terminer** son exécution en appelant la primitive exit. Le paramètre status constitue un compte-rendu d'exécution adressé au processus père de l'appelant ; c'est une valeur entière dont l'octet de faible poids est récupéré dans l'octet de fort poids chez le père.

```
pid_t wait (int *status)
```

Un processus peut **attendre** la terminaison d'un (de ses) fils en appelant la primitive wait(). L'appelant n'est pas suspendu si un fils est déjà terminé mais cette terminaison n'a pas encore été prise en compte chez son père ou si plus aucun fils n'est en train de s'exécuter. Si le père choisit de récupérer un éventuel compte-rendu envoyé par le fils grâce à exit(), le paramètre status doit être différent de NULL. Le père pourra alors utiliser les macros (WIFEXITED, WEXITSTATUS…) pour analyser la valeur de status et connaître la valeur réellement renvoyée par le fils lors de sone exit().

Communication d'informations entre processus

Des processus peuvent se communiquer des informations sous la forme d'un flot d'octets via des tubes de communication (nommés ou non). Dans ces deux cas, les processus doivent avoir un **lien de parenté** afin de pouvoir communiquer. La primitive int pipe(int pipefd[2]) sert à créer un tube (non nommé) et les primitives ssize_t read(int fd, void *buf, size_t count) et ssize_t write(int fd, const void *buf, size_t count), de gestion de fichiers, servent à y écrire (sur pipefd[1]) et lire (sur pipefd[0]) des informations. Enfin, la primitive close(int fd) sert à fermer une extrémité d'un tube en libérant le descripteur associé à cette extrémité.

Lorsque les processus n'ont plus **aucun lien de parenté**, mais continuent à s'exécuter sur la même machine, ces mécanismes ne suffisent plus et il faut avoir recours à des outils appelés IPC (Inter Process Communication) ce qui rend plus lourd la mise en œuvre de cette communication.

Enfin, si les processus s'exécutent sur des machines différentes, des outils réseau (du type sockets) doivent être utilisés.

Rappels sur les threads Posix (voir cours de L3)

Devoir utiliser des outils pour permettre à des processus de communiquer est parfois coûteux en termes de ressources (pour simplifier, un utilisateur est limité sur le nombre de tubes de communication ou d'IPC qu'il peut utiliser au sein d'une application).

C'est pourquoi, pour réaliser des applications parallèles, on fait souvent appel à la notion de **processus dits « légers »** ou threads.

Un **thread** est un flux d'exécution au sein d'un processus. Les threads qui s'exécutent (en parallèle) au sein d'un même processus **partagent l'espace d'adressage de ce processus**. Ceci libère le programmeur du recours à des outils supplémentaires pour manipuler des données partagées, toute variable définie globalement dans le code d'un processus est connue et manipulable par tout thread s'exécutant au sein de ce processus. Une variable définie localement dans le code propre à un thread n'est pas partagée et n'est manipulable que par ce thread.

Le concept de thread est proposé dans différents langages de programmation (C, Java, Python…).

Nous utiliserons pour cela les **threads** de la bibliothèque **Posix** (Pthreads, norme POSIX1.b).

Pour générer une application « multi-threadée », il faut inclure la bibliothèque pthread.h et compiler avec l'option –lpthread.

Toute application comporte au moins un **thread initial**, celui créé par le main() de cette application.

Attention :

La documentation ci-dessous vous donne l'ensemble minimal de primitives utiles pour créer une application « multi-threadée ». À vous de consulter le manuel en ligne (man) ainsi que d'autres sources pour approfondir le sujet.

Créer un thread Posix

Un thread peut créer un autre thread en appelant:

```
int pthread_create (pthread_t             *thread,
                    const pthread_attr_t  *attr,
                    void                  *(*start_routine)(void*),
                    void                  *arg);
```

- thread = adresse de l'identificateur du thread créé (de type opaque pthread_t, affichable grâce à %lu).
- attr = attributs optionnels de création du thread (laisser à la valeur NULL pour créer un thread ayant des valeurs d'attributs par défaut).
- start_routine = fonction exécutée par le thread dès son lancement (= code du thread). Cette fonction renvoie obligatoirement un void * et prend un paramètre un void *.
- arg = argument (optionnel) de la fonction.

Retourne 0 en cas de succès, une valeur différente en cas d'échec (EINVAL : attributs invalides, EAGAIN : ressources insuffisantes, EPERM : permission non accordée pour créer…).

Remarque :

void * désigne un pointeur générique, i.e. que tout type de pointeur peut être converti vers ce type générique. Cela signifie que start_routine peut récupérer une adresse sur n'importe « quoi » (int, float, char, structure…) et retourner une adresse sur n'importe « quoi » à condition de faire les bons transtypages.

Si l'on choisit d'utiliser un attribut attr lors de la création du thread, il faut que attr ait été initialisé auparavant et il devra être détruit une fois devenu inutile après la création (voir le paragraphe suivant).

Attributs d'un thread

Les attributs d'un thread permettent de définir certaines propriétés attachés au thread.

Si l'on ne précise pas d'attributs lors de la création d'un thread (deuxième paramètre valant NULL), ce thread est créé avec des propriétés par défaut.

Pour initialiser ou détruire un attribut utilisé lors de la création, on dispose de :

```
int pthread_attr_init (pthread_attr_t *);
int pthread_attr_destroy (const pthread_attr_t *);
```

Pour changer la valeur d'une propriété, on modifie l'attribut relatif à cette propriété. Par exemple, sa politique d'ordonnancement (attribut sched_policy), le fait qu'il hérite de la politique d'ordonnancement de son créateur (attribut inherit_scheduler), le fait qu'il est en compétition pour accéder à l'UC avec les threads de son processus ou tous les threads existants (attribut scope), la taille maximale de sa pile (attribut stacksize), l'adresse du début de sa pile (attribut stackaddr), etc.

Les primitives permettant de positionner la valeur d'une propriété ou de la consulter sont de la forme pthread_attr_get* et pthread_attr_set*. Exemple d'accès à la propriété « scope » :

```
int pthread_attr_setscope (pthread_attr_t *, int );
int pthread_attr_getscope (const pthread_attr_t *, int *);
```

Voir le manuel en ligne pour plus de détails.

Terminer un thread

Un thread se termine normalement à la fin de la fonction décrivant son traitement (le paramètre start_routine précisé lors de sa création).

Mais, un thread peut aussi choisir de se terminer en appelant la primitive :

```
void pthread_exit (void *retval);
```

Le paramètre retval est un pointeur générique, le thread retourne l'adresse d'un compte-rendu qui pourra être récupérée grâce à la primitive pthread_join() (voir ci-après). pthread_exit() fait passer le thread dans un état « zombie » jusqu'à ce que le compte-rendu rendu par le thread soit effectivement récupéré.

Attention : Le pointeur retval doit repérer une zone qui sera encore accessible après la terminaison du thread. Cette zone ne peut donc être une variable locale au thread car elle serait allouée dans la pile du thread et donc détruite à la terminaison de ce thread.

Contrairement à un processus fils, **l'existence d'un thread fils est liée à celle du processus le contenant** (i.e. le main() de l'application). Il est donc nécessaire que le processus contenant les activités parallèles ne se termine pas avant ces threads. En d'autres termes, le processus – ou thread initial – doit donc attendre la fin de tous les threads composant l'application avant de se terminer.

Attendre la terminaison d'un thread

Un thread peut attendre la terminaison d'un autre thread en appelant :

```
int pthread_join (pthread_t thread, void **retval );
```

- thread est l'identificateur du thread attendu.
- retval est un pointeur void ** contenant l'adresse d'une variable void * devant recevoir le code retour du thread attendu (pour rappel, un pointeur générique).

Le thread attendu se termine réellement après réception de la valeur retournée (il quitte l'état « zombie »)

pthread_join() retourne 0 en cas de succès, un code d'erreur sinon (EDEADLCK : le thread s'attend soi-même, EINVAL : identificateur spécifié invalide, ESRCH : le thread attendu n'existe pas…).

Obtenir l'identité d'un thread

Un thread peut obtenir son identificateur (numéro unique) en appelant :

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

Retirer l'UC à un thread

Un thread peut choisir de relâcher l'UC en appelant :

```
int pthread_yield (void);
```

Toutefois, cette primitive n'est pas normalisée et il est conseillé d'utiliser :

```
#include <sched.h>
int sched_yield (void);
```

L'UC est allouée à un autre thread prêt à s'exécuter.

Retourne 0 en cas de succès, - 1 sinon (+ positionnement de errno).

Exemple de création d'un thread

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

/* Fonction exécutée par le thread */
void *f_thread (void *p) {
  int *adrParam = (int *)p ;       /* Récupérer l'adresse passée en paramètre */
  int *cr = malloc(sizeof(int));  /* Retourner une adresse qui ne disparaît pas */
  printf("\tDebut du thread fils\n");
  printf("\tIci le thread numero: %lu, %d recupere \n", pthread_self(), *adrParam);
  *cr = 100;
  printf("\tValeur retournee: %d, depuis l'adresse %p \n", *cr, cr);
  pthread_exit((void*)cr);
      /* On pourrait aussi écrire : return ((void *)cr);
      ou return ((void *)&cr); si cr était déclarée en global par : int cr
      (mais dans ce cas, cr est aussi partagé par le thread principal
      et retourner sa valeur ainsi a vraiment peu d'intérêt) */
}

int main(void){
  pthread_t ptid;
  int *res = NULL;
  int val = 20;  /* Juste un exemple de valeur */

  printf("Debut du thread principal\n");
  /* Créer le thread fils (avec des attributs par défaut)*/
  if ( pthread_create(&ptid, NULL, f_thread, (void *)&val) != 0){
    perror("Probleme lors de la creation du thread fils :");
    exit(99);  /* plutôt pthread_exit() si hors du thread principal */
  }
  /* Attendre la fin d'exécution du thread fils ! */
  pthread_join(ptid, (void**)&res);      /* Récupérer une adresse (res) */
  printf("Adresse recuperee %p, valeur %d \n", res, *res); /* Affiche compte-rendu */
  free(res);                  /* Libérer la mémoire allouée à l'adresse récupérée */
  printf("Fin du thread principal\n");
}
```

Exécution :

```
monPC %./exemple
Debut du thread principal
        Debut du thread fils
        Ici le thread numero: 3075885888, 20 recupere
        Valeur retournee: 100, depuis l'adresse 0xb6c00468
Adresse recuperee 0xb6c00468, valeur 100
Fin du thread principal
monPC %
```

Synchronisation de threads Posix par verrous (mutex)

Les verrous mutex font partie de la norme Posix Pthread (POSIX1.c). Un mutex possède un identificateur/descripteur unique de type pthread_mutex_t qui est un type « opaque ». un mutex possède deux états : libre ou verrouillé.

Avant de pouvoir faire des opérations sur un mutex, il est nécessaire de créer/initialiser ce mutex pour allouer les ressources nécessaires à cet « objet ». Lorsqu'il n'est plus utile, il faut le détruire pour libérer les ressources qui lui ont été associées.

Créer un mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init (pthread_mutex_t             *mutex,
                        const pthread_mutexattr_t   *attr);
```

En cas de succès, le mutex est initialisé et créé dans **l'état non verrouillé** (libre). Tenter d'initialiser un mutex déjà initialisé conduit à un comportement indéfini.

- mutex = adresse pour récupérer l'identificateur retourné après création.
- attr = attributs optionnels pour l'initialisation. Si attr vaut NULL, le mutex est créé avec les attributs par défaut.

En cas de succès retourne 0, sinon retourne une valeur différente de 0 (EAGAIN : ressources système (autre que mémoire) insuffisantes, ENOMEM : pas assez de mémoire pour initialiser le mutex, EPERM : droits insuffisants pour effectuer l'opération….)

Remarque : La macro PTHREAD_MUTEX_INITIALIZER permet une initialisation d'une variable mutex « static » pour laquelle une initialisation dynamique n'est pas utile. Cela revient à faire une initialisation dynamique par pthread_mutex_init() dans laquelle attr possède la valeur par défaut mais, dans ce cas, aucun contrôle des erreurs n'a lieu.

Détruire un mutex

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

- mutex = descripteur du mutex à détruire.

La destruction du mutex n'est possible que si le mutex référencé par mutex **n'est pas verrouillé** (sinon le comportement est indéfini). Cela provoque la libération des ressources monopolisées par le mutex. Une fois détruit, un mutex redevient « non initialisé » et peut être à nouveau initialisé. Toute autre opération ne peut plus être utilisée (comportement indéfini).

Une implantation peut faire en sorte que la valeur de l'objet référencé par mutex devienne invalide.

Retourne 0 en cas de succès, un code erreur sinon (EBUSY : tentative de détruire un mutex verrouillé, EINVAL : descripteur invalide…).

Verrouiller un mutex (Opération P)

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

Opération P **bloquante**. On demande l'accès au mutex et sa propriété. Si le mutex est libre, l'accès est autorisé. Si le mutex est déjà verrouillé, l'appelant est bloqué.

Retourne 0 en cas de succès, un code erreur sinon (EINVAL : valeur invalide de mutex, EDEADLK : l'appelant possède déjà le mutex dans le cas des verrous à vérification, du style des verrous récursifs – voir le manuel en ligne….).

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

Opération P **non bloquante**, l'appelant essaie d'obtenir l'accès au mutex et est averti du résultat (succès ou échec) grâce au retour de la fonction.

Retourne 0 en cas de succès, un code erreur sinon (EINVAL : valeur invalide de mutex, et selon le type de verrou : EDEADLK : l'appelant possède déjà le mutex, EBUSY : le mutex n'a pas pu être acquis car il est déjà verrouillé….).

Déverrouiller un mutex (Opération V)

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

Le mutex devient libre et un autre thread ayant demandé un accès est autorisé à l'obtenir grâce à la primitive pthread_mutex_lock() ou …_try_lock().

La manière dont le mutex est « libéré » dépend du type de mutex paramètre. Si des threads sont bloqués sur le mutex quand …_unlock() est appelée, la politique d'ordonnancement est utilisée pour déterminer quel thread devra obtenir le mutex.

Retourne 0 en cas de succès, un code erreur sinon (EINVAL : valeur invalide de mutex, et selon le type de verrous : EDEADLK : l'appelant possède déjà le mutex, EPERM : le thread appelant n'est pas propriétaire du mutex…).

Synchronisation de threads Posix par variables condition

Les variables condition permettent de reproduire le type de synchronisation mise en place par les moniteurs de Hoare à **deux différences** importantes près :

- **L'exclusion mutuelle** entre les opérations du « moniteur » n'est pas réalisée de base et il faut **utiliser des mutex pour la réaliser**. Une condition Posix est donc liée à un mutex (voir les spécificités des opérations pthread_cond_wait() et pthread_cond_signal()).
- L'activité parallèle qui est réveillée n'est pas sûre d'être prioritaire sur celle qui la réveille ; il lui est donc nécessaire de **tester à nouveau le prédicat booléen** qui l'a menée éventuellement à se bloquer (on utilise donc un while au lieu d'un if).

Créer une variable condition

```
int pthread_cond_init (pthread_cond_t               *cond,
                       const pthread_condattr_t     *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Initialise la variable conditionnelle référencée par cond avec les attributs référencés par attr. Si attr vaut NULL, les attributs par défaut sont utilisés. En cas de succès, l'état de la condition devient « initialisé ».

Le comportement résultant de la tentative de réinitialiser une condition déjà initialisée est indéfini.

Lorsque les attributs par défaut conviennent, la macro PTHREAD_COND_INITIALIZER peut être initialisée pour l'initialisation de conditions de manière statique. Ceci équivaut à l'initialisation dynamique avec attr valant NULL mais aucun contrôle d'erreur n'est effectué.

Retourne 0 en cas de succès, un code erreur sinon (EAGAIN : ressources insuffisantes (autre que la mémoire), ENOMEM : ressource mémoire insuffisante. Selon les implantations : EBUSY : tentative de réinitialiser l'objet référencé par cond, EINVAL : valeur de attr invalide).

Détruire une variable condition

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

Détruit la variable condition référencée par cond ; l'état de cet objet devient alors « non initialisé ».

Une condition détruite peut être à nouveau initialisée ; le résultat d'une autre opération référençant un objet condition détruit est indéfini.

On peut détruire une condition initialisée sur laquelle aucun thread n'est bloqué. Détruire une condition sur laquelle des threads sont bloqués provoque un comportement indéfini.

Retourne 0 en cas de succès, un code erreur sinon (Selon les implantations : EBUSY : tentative de détruire l'objet référencé par cond alors qu'il est référencé par un autre thread, EINVAL : valeur de cond invalide).

Attendre sur une variable condition

```
int pthread_cond_wait (pthread_cond_t      *cond,
                       pthread_mutex_t     *mutex);
```

pthread_cond_wait() déverrouille **atomiquement** le mutex et attend que la variable condition cond soit signalée. L'exécution du thread est **suspendue** et ne consomme pas de temps CPU jusqu'à ce que la variable condition soit signalée. **Le mutex doit être verrouillé par le thread appelant** à l'entrée de pthread_cond_wait(). Avant de rendre la main au thread appelant, pthread_cond_wait() **reverrouille** mutex.

Le déverrouillage du mutex et la suspension de l'exécution sur la variable condition sont liés de manière atomique. Donc, si tous les threads verrouillent le mutex avant de signaler la condition, il est garanti que la condition ne peut

être signalée (et donc ignorée) entre le moment où un thread verrouille le mutex et le moment où il attend sur la variable condition.

Retourne 0 en cas de succès, un code d'erreur sinon (selon l'implantation : EINVAL : valeur de cond ou mutex invalide, ou différents mutex ont été fournis pour des opérations concurrentes sur la même variable condition, ou le mutex n'appartient pas au thread appelant…).

Remarques :

Utiliser plus d'un mutex pour des opérations concurrentes d'attente sur la même variable condition a un effet indéterminé ; i.e. **une variable condition est liée à un unique mutex lorsqu'un thread attend sur cette condition** et cette liaison (dynamique) se termine quand l'attente se termine.

Si un signal est délivré à un thread en attente d'une condition, au retour du gestionnaire de signal, le thread termine l'attente comme s'il n'avait pas été interrompu ou il retourne zéro à cause d'un réveil non valable.

Il existe aussi une attente temporisée réalisable grâce à pthread_cond_timedwait(), voir la manuel en ligne.

Signaler une variable condition

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Relance l'un des threads attendant sur la variable condition cond. S'il n'existe aucun thread répondant à ce critère, rien ne se produit. Si plusieurs threads attendent sur cond, **seul l'un d'entre eux sera relancé**, mais il est impossible de savoir lequel.

Retourne 0 en cas de succès, un code d'erreur sinon.

Remarque : Un thread peut utiliser la fonction pthread_cond_broadcast() pour relancer **tous** les threads bloqués sur la variable condition cond (voir manuel en ligne).