Threads ou processus légers

Éric Renault et Frédérique Silber-Chaussumier

TELECOM SudParis

module CSC4508/M2

Avril 2018

Plan du document

1 Présentation

1.1 Bibliographie

- « Computer Systems: A Programmer's Perspective », *R. E. Bryant et D. R. O'Hallaron*, Prentice Hall, 2003.
- « UNIX SYSTEMS Programming: communication, Concurrency and Threads », *K. A. Robbins et S. Robbins*, Prentice Hall, 2003.
- « Programmation système en C sous Linux », *C. Blaess*, Eyrolles, 2000.
- « Understanding the Linux kernel, 2nd edition », *Daniel P. Bovet et M. Cesati*, O'Reilly, 2003.
- Threads spécifiquement
  - « Pthreads Programming: A POSIX Standard for Better Multiprocessing », *B. Nichols, D. Buttlar et J. P. Farrell*, O'Reilly and Associates, 1996.
  - « Threads Primer: A Guide to Multithreaded Programming », *B. Lewis, D. Berg et B. Lewis*, Prentice Hall, 1995.
  - « Programming With POSIX Threads », *D. Butenhof*, Addison Wesley, 1997.
  - « Techniques du multithread : du parallélisme dans les processus », *B. Zignin*, Hermès, 1996.

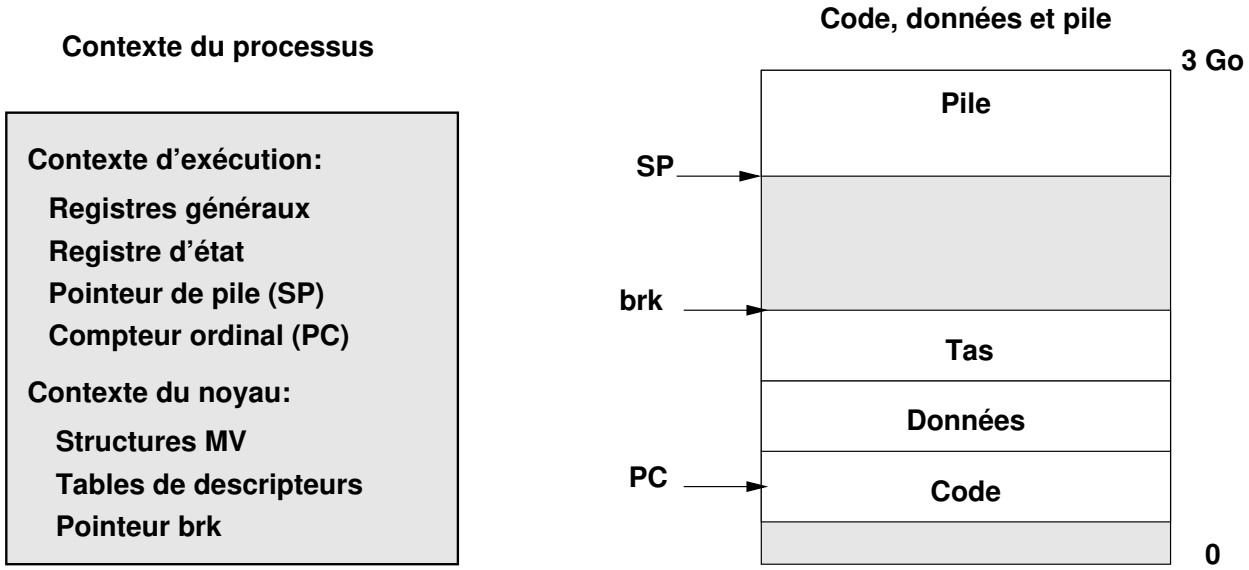
1.2 Threads

- Traduction : fil d'exécution, processus légers
- Principe : détacher le flot d'exécution des ressources
- Introduits dans divers langages et systèmes :
  - Programmation concurrente
  - Utilisation des ressources simultanément: recouvrement du calcul et des entrées/sorties
  - Exploitation des architectures multiprocesseurs
    - Architectures SMP ou CMP
    - Plus généralement des architectures à mémoire partagée
  - Exploitation de la technologie *hyper-threading*

1.3 Détacher le flot d'exécution des ressources

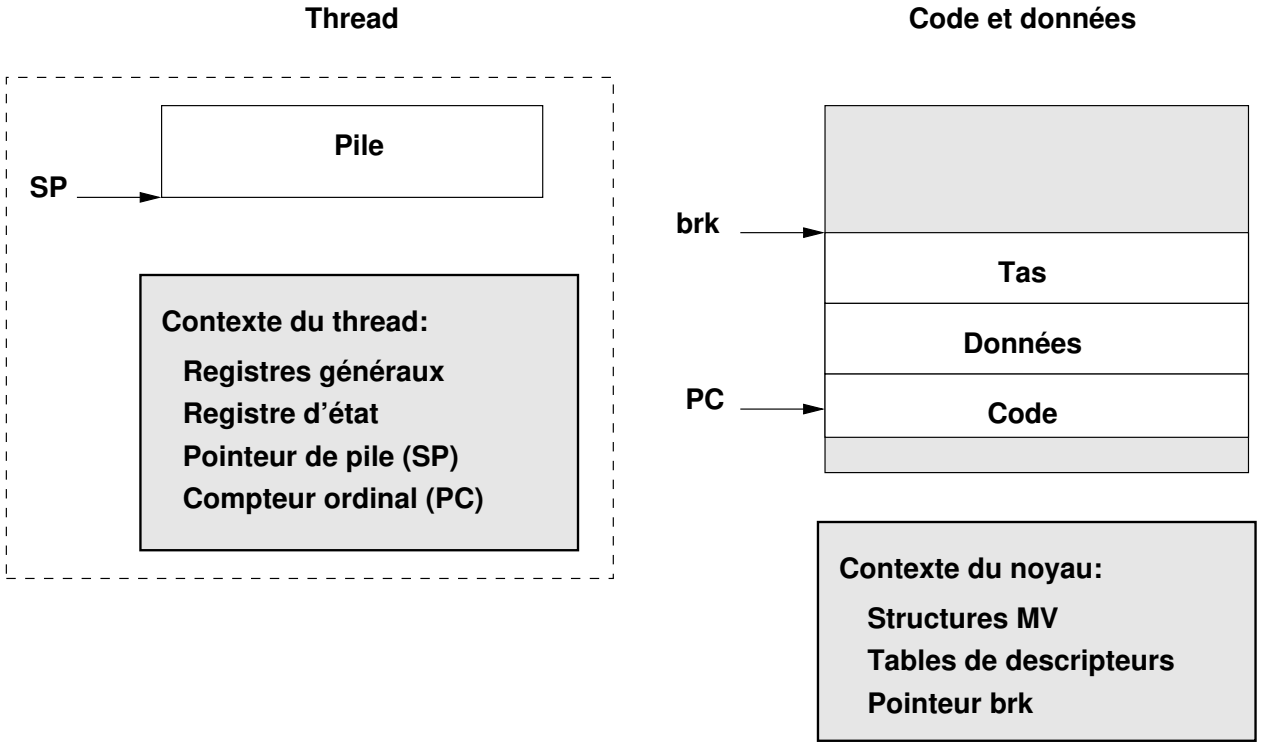
1.3.1 Vision traditionnelle d'un processus

- Processus mono-thread
  - Contexte: contexte d'exécution + contexte du noyau
  - Espace d'adressage: code, données et pile



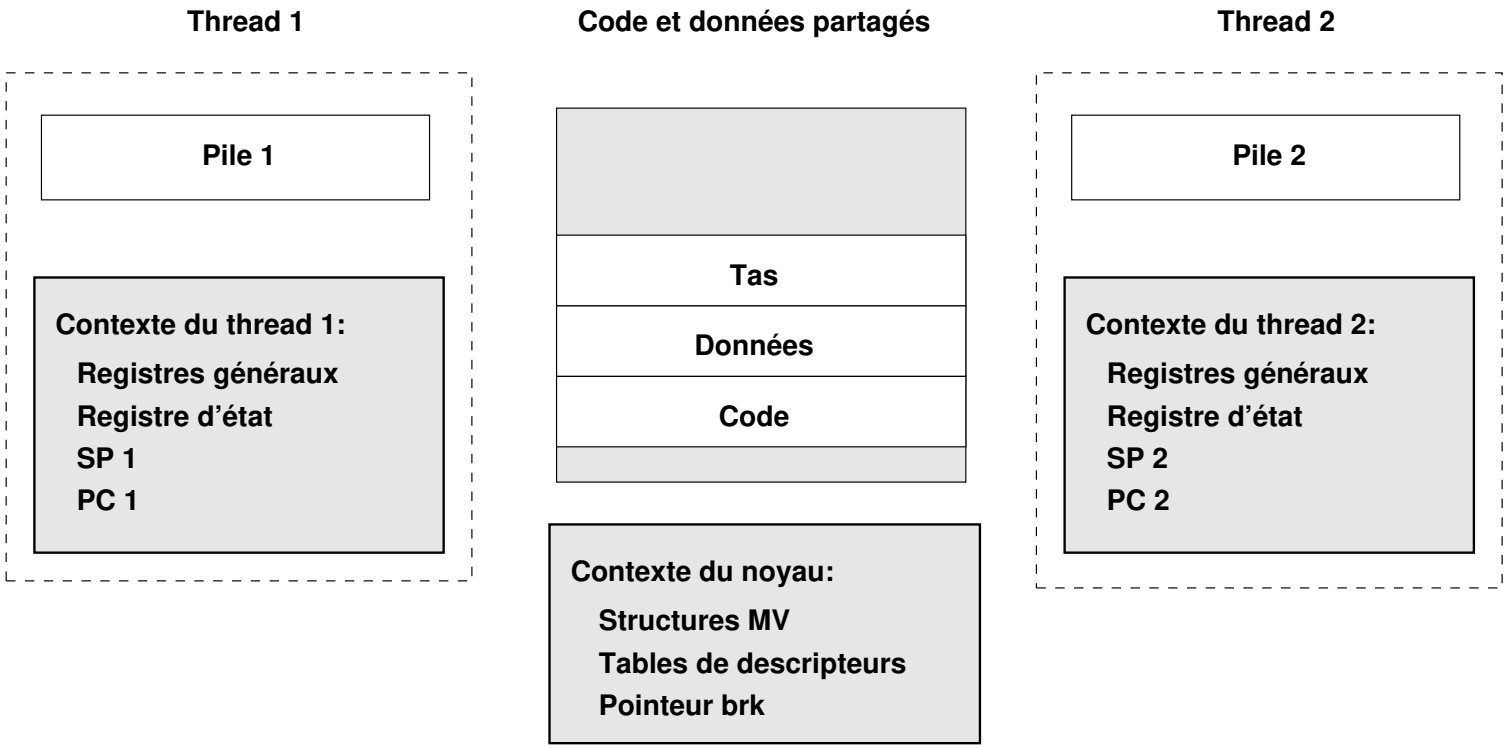
1.3.2 Autre vision d'un processus

- Détacher le flot d'exécution des ressources
- Processus mono-thread
  - Fil d'exécution ou *thread*: pile + contexte d'exécution
  - Code, données et le contexte du noyau



1.3.3 Processus multi-thread

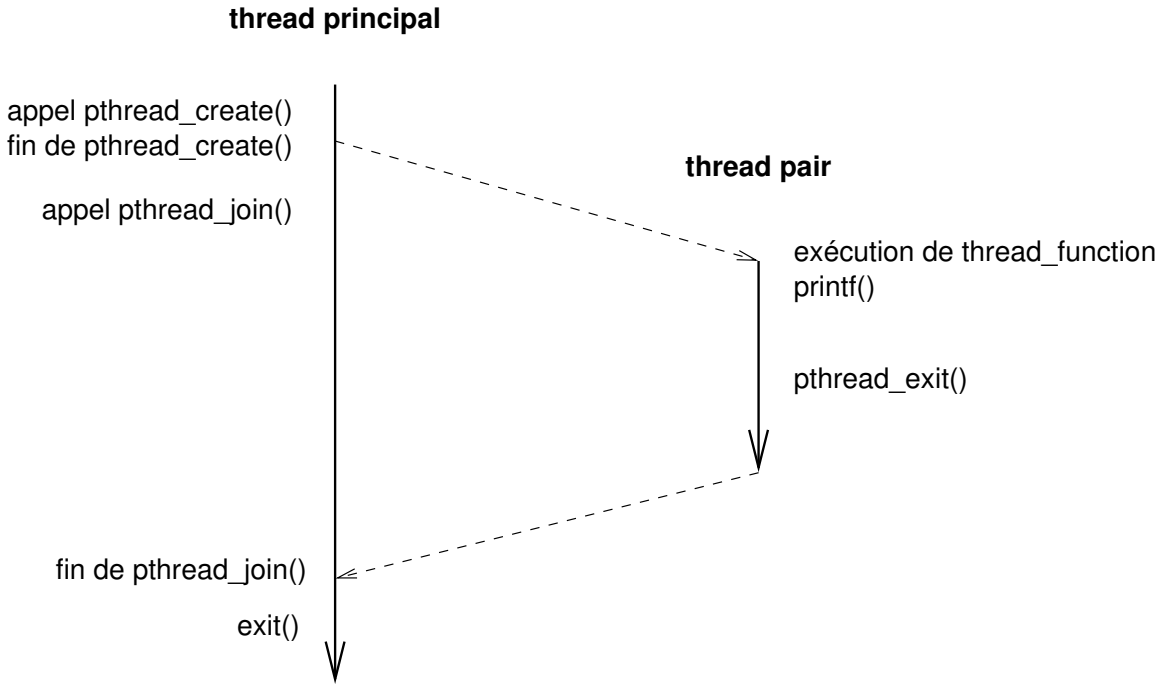
- Processus multi-thread
  - Plusieurs fils d'exécution
  - Code, données et contexte du noyau partagés: notamment partage des fichiers et des ports de communication



2 Création/destruction de threads

2.1 Pthread “Hello world”

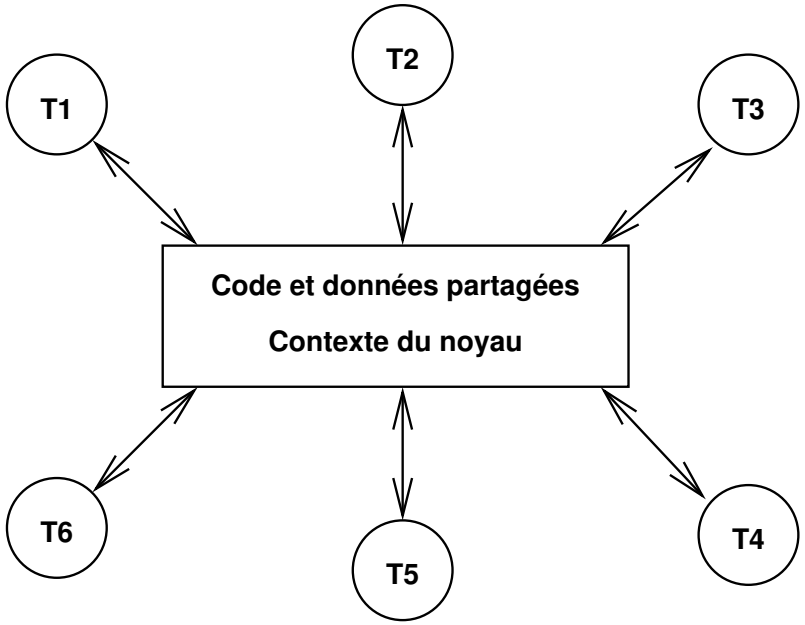
- Interface POSIX Pthread
- Fournir une fonction point d'entrée



- Visualisation des différents threads avec la commande ps

2.2 Ensemble de threads pairs

- Création d'un ensemble de threads pairs
- Pas d'arborescence parent/enfant comme pour les processus



2.3 Threads POSIX: création/destruction

2.3.1 Identification

- Un équivalent du `pid_t` : `pthread_t`
- Identifiant du thread courant :
  - `pthread_t pthread_self (void)`
- Comparaison de deux identifiants :
  - `int pthread_equal (pthread_t thread1, pthread_t thread2)`

2.3.2 Utilisation

- Création
  - `int pthread_create (pthread_t *thread, pthread_attr_t *attr, void *(* start_routine) (void *), void *arg)`
- Terminaison
  - `void pthread_exit (void *retval)`
  - `int pthread_join (pthread_t thread, void **thread_return)`
  - `int pthread_detach (pthread_t thread)`
- Attention: pour la plupart des fonctions de la bibliothèque Threads POSIX, un code d'erreur non nul est renvoyé en cas de problème mais `errno` n'est pas nécessairement positionné.

2.3.3 Attributs

- Création et destruction d'une structure d'attributs
  - `int pthread_attr_init (pthread_attr_t *attr)`
  - `int pthread_attr_destroy (pthread_attr_t *attr)`
- État joignable / détaché
  - `int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate)`
  - `int pthread_attr_setdetachstate(const pthread_attr_t *attr, int detachstate)`
  - → `detachstate = PTHREAD_CREATE_[JOINABLE|DETACHED]`

Attributs (2/2)

- Politique d'ordonnancement des threads

```
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy)
int pthread_attr_setschedpolicy(const pthread_attr_t *attr, int policy)
```

→ `policy = SCHED_[OTHER|RR|FIFO]`

- Paramètres d'ordonnancement
  - Priorité d'ordonnancement

```
int pthread_attr_getschedparam(const pthread_attr_t *attr, const struct
sched_param *param)
int pthread_attr_setschedparam(const pthread_attr_t *attr, struct
sched_param *param)
```

  - Héritage de l'ordonnancement

```
int pthread_attr_getinheritsched(const pthread_attr_t *attr, int
*inherit)
```

```
int pthread_attr_setinheritsched(const pthread_attr_t *attr, int inherit)
```

→ `inherit = PTHREAD_[EXPLICIT|INHERIT]_SCHED`

- Interprétation des valeurs d'ordonnancement

```
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope)

int pthread_attr_setscope(const pthread_attr_t *attr, int scope)
```

→ `scope = PTHREAD_SCOPE_[SYSTEM|PROCESS]`

- Adresse de la pile du thread

```
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void *addr)

int pthread_attr_setstackaddr(const pthread_attr_t *attr, void *addr)
```

- Taille de la pile du thread

```
int pthread_attr_getstacksize(const pthread_attr_t *attr, int size)

int pthread_attr_setstacksize(const pthread_attr_t *attr, int *size)
```

3 Partage des données

3.1 Notion de variable partagée

- Variable partagée: deux définitions possibles
  - Conceptuellement: variable utilisée par plusieurs threads
  - Techniquement: pendant l'exécution une seule instance de la variable pour tous les threads
    - Variable globale: une seule instance "partagée"
    - Variable statique locale: une seule instance "partagée"
    - Variable automatique locale: une instance dans chacune des piles des threads appelants
- Attention: les threads partagent l'intégralité de l'espace d'adressage du processus
  - Toutes les variables peuvent potentiellement être partagées.
  - Même les variables automatiques locales peuvent être partagées: à utiliser avec précaution.
- Partage du contexte du noyau
  - Gestion des flux
  - Gestion des signaux

3.2 Partage non-intentionnel des données

- Lorsqu'une même instance de variable est utilisée par plusieurs threads alors que conceptuellement ce n'est pas une variable partagée
  - Peut provoquer des situations de compétition (*race condition*)
    - Le résultat varie selon les conditions d'exécution
  - Erreurs difficiles à détecter!

3.3 Code réentrant et thread-safe

- Code réentrant: pas de variables partagées lors d'exécutions simultanées par plusieurs threads
  - Ne pas maintenir d'état persistant entre les appels
    - Contre-exemple: `strtok`, `rand`
  - Ne pas retourner de pointeur sur une variable statique
    - Contre-exemple: `ctime`
- Code *thread-safe*: résultats corrects lors d'exécutions simultanées par plusieurs threads
  - Protéger les accès aux données partagées dans les fonctions
    - Contre-exemple: variable globale externe `errno`
    - Redéfinition de `errno`: chaque thread a son propre `errno`
  - Pas d'appel à du code non *thread-safe*

3.4 Thread-Local Storage

- Variables globales propre à chaque thread
- Chaque thread a une copie distincte de la variable
  - exemple: `errno`
- Création
  - `int pthread_key_create (pthread_key_t *cle, void (*destr_fonction ) ( void * )`
- Destruction
  - `int pthread_key_delete ( pthread_key_t cle)`
- Utilisation
  - `int pthread_setspecific (pthread_key_t cle, const void *pointeur)`
  - `void * pthread_getspecific( pthread_key_t cle)`
- Ceci peut utilement être utilisé avec `pthread_once`

4 Synchronisation

4.1 Synchronisation

- Garantir la consistance des données
  - Accès simultanés à une donnée partagée en lecture/écriture
    - Séquencement des instructions `load`, `update`, `store`
  - Exemple: exclusion mutuelle de type compte bancaire
- Implantation de P et V avec des threads: plusieurs outils de la bibliothèque Pthread
  - Sémaphores
  - Mutex
  - Conditions

4.2 Exclusions mutuelles

- Type : `pthread_mutex_t`
- Création
  - `int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)`
  - `pthread_mutex_init` retourne toujours 0.
- Destruction
  - `int pthread_mutex_destroy (pthread_mutex_t *mutex)`

Exclusions mutuelles (2/2)

- Utilisation
  - `int pthread_mutex_lock (pthread_mutex_t *mutex)`
  - `int pthread_mutex_unlock (pthread_mutex_t *mutex)`
  - `int pthread_mutex_trylock (pthread_mutex_t *mutex)`
- Attributs
  - Les attributs associés aux MUTEX ne sont pas portables
  - Il convient de ne pas trop les utiliser
  - Ou de se référer à la documentation en ligne

4.3 Sémaphores POSIX (rappel)

- Création et destruction
  - `int sem_init ( sem_t *sem, int pshared, u_int value)`
  - `int sem_destroy ( sem_t *sem)`
- Utilisation
  - `int sem_wait ( sem_t *sem)`
  - `int sem_post ( sem_t *sem)`
  - `int sem_trywait ( sem_t *sem)`
  - `int sem_getvalue ( sem_t *sem, int *sval)`
- Toutes ces fonctions renvoient -1 en cas de problème et positionnent `errno`.

4.4 Attente de conditions

- Principe
  - Un premier *thread* se met en attente d'un condition
  - Lorsqu'un second *thread* réalise la condition, il émet un signal à destination de la condition, réveillant un *thread* en attente
    - Si aucun *thread* n'est en attente, rien ne se passe
    - Si plusieurs *threads* sont en attente, un *thread* est réveillé
- Type : `pthread_cond_t`
- Création
  - `int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *cond_attr)`
  - `pthread_cond_init` retourne toujours 0.
- Destruction
  - `int pthread_cond_destroy ( pthread_cond_t *cond)`
- Notes : il n'existe pas d'attribut pour les conditions

Attente de conditions (2/2)

- Utilisation
  - `int pthread_cond_signal ( pthread_cond_t * cond)`
  - `int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex)`
  - `int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`
  - `int pthread_cond_broadcast ( pthread_cond_t * cond)`
  - Les fonctions `pthread_cond_signal`, `pthread_cond_wait` et `pthread_cond_broadcast` renvoient toujours 0.
- Notes
  - Toujours prendre le mutex associé à la condition avant et le relâcher ensuite.
    - Pas d'interblocage entre le *thread* se plaçant en attente de la condition et celui la réalisant.

- `pthread_cond_wait` libére le mutex puis se met en attente de la condition. Une fois que la condition est réalisée, la fonction bloque à nouveau le mutex avant de sortir.

- `pthread_cond_wait` peut se terminer même si la condition n'a pas été réalisée.

5 Utilisation et limitations des threads

5.1 Utilisation des threads

- Pourquoi utiliser les threads?
  - Améliorer la réactivité des applications
  - Partager des ressources
  - Économiser du temps et de la place mémoire
  - Exploiter des architectures multi-processeurs
- Exemples d'architectures logicielles
  - Maître/esclave
  - Diviser pour régner
  - Producteur/consommateur
- Exemples d'applications
  - Traitement de texte
  - Navigateur web
  - Serveur web

- Pour mon application?
  - Identifier des opérations coûteuses
  - Déterminer la taille des données à partager (si possible petite)
  - Identifier les (nombreux?) problèmes de verrous
  - Répartir les tâches indépendantes

5.2 Limitations des threads

- Ressources que l'on ne souhaite pas partager
  - ID utilisateur, groupe
  - droits
  - quotas d'utilisation de ressources : nombre maximal de fichiers ouverts par un processus, ...
- Corruption des ressources partagées en mémoire
- Mort d'un *thread* → mort de l'application entière
- Problématiques
  - Exécution d'un `fork`
    - Duplication de tous les *threads* ou nouveau processus mono-*thread*
  - Gestion de signaux : à qui envoyer le signal ?
    - Au *thread* auquel il s'applique ?
    - À tous les *threads* du processus ?
    - À certains *threads* ?
    - À un *thread* spécifique qui se chargera de le gérer correctement ?
  - Éviter de communiquer par signaux dans une application multi-*thread*!

6 Autres fonctions de la bibliothèque POSIX threads

6.1 Annulation

- Envoi d'une requête d'annulation
  - `int pthread_cancel (pthread_t thread)`
- Définition du comportement
  - `int pthread_setcancelstate (int state, int * oldstate)`

    → state = `PTHREAD_CANCEL_[ENABLE|DISABLE]`
- Définition du type d'annulation (si `PTHREAD_CANCEL_ENABLE`)
  - `int pthread_setcanceltype (int type, int * oldtype)`

    → type = `PTHREAD_CANCEL_[DEFERRED|ASYNCHRONOUS]`

Annulation (2/2)

- Test d'annulation explicite
  - `void pthread_testcancel (void)`
- Test d'annulation implicite
  - `pthread_cond_wait, pthread_cond_timedwait`
  - `pthread_join, sem_wait, sigwait`

6.2 Nettoyage des ressources

- Deux routines
  - `void pthread_cleanup_push (void (* routine) (void *), void *arg)`
  - `void pthread_cleanup_pop (int execute)`
    - `execute` à 0: fonction supprimée de la pile mais non exécutée
    - `execute` à 1: fonction supprimée de la pile ET exécutée
- Attention
  - Les deux appels doivent appartenir au même bloc d'instructions

6.3 Initialisation

- À l'entrée d'une fonction
  - Type : `pthread_once_t`
  - Valeur : `PTHREAD_ONCE_INIT`
  - `int pthread_once ( pthread_once_t *once_control, void (*once_routine)())`
- Lors d'un `fork`
  - Duplication du fil d'exécution réalisant un appel à `fork`
  - `int pthread_atfork (void (*prepare) ( ), void (*parent) ( ), void (*child) ( ))`

Bibliographie du chapitre