



Cours de système

Threads et démons

Sébastien Paumier



Les threads POSIX

- pour des applications multi-tâches, pas toujours simple d'avoir des processus séparés communiquant par IPC
 - coûte cher en commutations de contexte
 - coûte cher en organisation, car il faut pouvoir accéder à des données communes en utilisant de la mémoire partagée
- idée: avoir une unité moins lourde à gérer que le processus, le *thread*



Les threads POSIX

- un processus contient un ou plusieurs threads qui:
 - exécutent des portions du même programme
 - partagent la mémoire globale (données statiques, variables globales, tas)
 - ont chacun leur propre pile d'exécution
- cf. *man pthreads*



Threads utilisateur vs noyau

- les threads peuvent être gérés au niveau du noyau en affinant l'ordonnanceur pour qu'il travaille au niveau des threads
 - moins lourd à gérer que des processus
 - threads noyau = [binou] dans **ps lax**
- ils peuvent être aussi gérés au niveau utilisateur avec une bibliothèque dédiée
 - méthode utilisée pour les threads utilisateur, gérés avec la bibliothèque NPTL



Threads

- par défaut, un processus n'a qu'un seul thread
- on en crée avec:
- `int pthread_create(pthread_t *thread, pthread_attr_t* attr, void* (*start_routine)(void *), void* arg);`
- crée un thread qui va exécuter la fonction `start_routine`



Threads

- si non NULL, **attr** définit les attributs du thread (mode d'ordonnancement, état)
- deux états:
 - joignable (par défaut): on pourra se synchroniser sur la fin de ce thread depuis un autre grâce à **pthread_join**; les ressources du thread ne seront libérées qu'à ce moment-là
 - détaché: pas de synchronisation possible, mais libération immédiate des ressources



Threads

- on passe des arguments à **start_routine** sous la forme d'un pointeur, par exemple vers une structure
- un thread se termine soit quand **start_routine** se termine, soit quand il invoque **pthread_exit**

pthread.cpp

- note: compiler avec l'option **-lpthread**



Concurrence

- en programmation multi-threads, il faut éviter les problèmes de concurrence puisque tous les threads partagent la mémoire globale
- possibilité d'utiliser la propriété d'atomicité des E/S dans les tubes
- exemple: attendre la mort de threads sans devoir respecter un ordre précis avec `pthread_join`

pthread2.cpp



Variables privées

- que faire si chaque thread a besoin de sa propre valeur d'une variable ?
- on peut gérer à la main un tableau global de valeurs, mais:
 - fastidieux
 - pas de protection puisque tous les threads ont accès à la mémoire globale
- solution: avoir des instances privées pour chaque thread



Variables privées

- `int pthread_key_create(pthread_key_t *clé, void (*destr_function) (void *));`
- fabrique une "clé" à laquelle on pourra associer un pointeur propre à chaque thread, lisible avec `pthread_getspecific` et `pthread_setspecific`

`pthread3.cpp`



Les mutex

- pour protéger l'accès à une zone partagée et éviter ainsi les problèmes de concurrence, on utilise des verrous: les *mutex*
- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`



Les mutex

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- tentative bloquante de prendre le verrou
- tentative non bloquante avec `pthread_mutex_trylock`
- on relâche un verrou avec:
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

`pthread4.cpp`



Les conditions

- possibilité de bloquer un thread jusqu'à ce qu'une condition soit déclarée remplie par un autre thread
- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);`
- pour attendre une condition, on doit utiliser un mutex:
- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);`



Les conditions

- on réveille un thread en attente de la condition avec:
- `int pthread_cond_signal(pthread_cond_t *cond);`
- ou tous avec (mais un seul sera effectivement relancé):
- `int pthread_cond_broadcast(pthread_cond_t *cond);`

pthread5.cpp



Les sémaphores

- on peut aussi utiliser des sémaphores
 - verrou à compteur qui bloque si sa valeur est nulle
- `int sem_post(sem_t *sem);`
- `int sem_wait(sem_t *sem);`
- `int sem_trywait(sem_t *sem);`
- `int sem_getvalue(sem_t *sem, int *sval);`

`pthread6.cpp`



Les sémaphores

- utile quand un thread doit attendre plusieurs événements avant de poursuivre son exécution
- pour attendre n tâches, il faut que chaque tâche fasse un `sem_post` et que le thread en attente fasse n `sem_wait`

`pthread7.cpp`



Les sémaphores nommés

- permettent de communiquer entre processus
- `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`
- `int sem_close(sem_t *sem);`
- `int sem_unlink(const char *name);`
- exemple: tester qu'une instance tourne déjà

monoinstance.cpp



Mono-instance

- si une instance du programme tourne déjà, comment faire pour lui envoyer des données ?
 - exemple: gedit qui ouvre un nouvel onglet au lieu de se relancer complètement
- idée:
 - avoir un thread en tâche de fond qui vérifie si des données arrivent d'une nouvelle instance



Mono-instance

- quand le programme se lance, il teste s'il est ou non la première instance, avec un verrou sur un fichier
- s'il n'est pas la première instance, il envoie des données à celle-ci, via par exemple, un tube nommé
- sinon, il démarre la veille dans un thread et la tâche principale dans un autre

`monoinstance2.cpp`



Les démons

- démon=tâche de fond qui ne dépend pas d'un utilisateur
- exemples: cron, apache
- leur conception est soumise à des règles strictes pour:
 - ne pas avoir de dépendances inutiles
 - ne pas créer de dépendances inutiles
 - ne pas créer de failles de sécurité



Créer un démon

- règle n°1: ne dépendre d'aucun processus:
 - se faire adopter par *init* avec **fork+exit**
 - devenir leader de session avec **setsid** pour être détaché de tout terminal de contrôle
- règle n°2: ne rien hériter du processus qui a démarré le démon
 - fermer tous les descripteurs de fichiers (utiliser **getdtablesize**)



Créer un démon

- règle n°3: beaucoup de fonctions supposent qu'elles peuvent lire et écrire dans 0 1 et 2, qu'on vient de fermer
 - pour ne pas provoquer d'erreurs de descripteurs invalides ni retourner de mauvaises données, on doit proposer des descripteurs
 - on redirige 0 1 et 2 vers `/dev/null`
- note: la fonction `daemon` remplit ce rôle (entre autres)



Créer un démon

- un démon peut tourner avec les droits root !!!
- règle n°4: contrôler les droits des fichiers qu'il pourrait créer
 - on ne veut pas créer de fichiers appartenant à root avec des permissions libres en écriture!
 - on positionne un masque `rwxr-x--` avec `umask(027)`



Créer un démon

- règle n°5: on ne peut s'adresser à un démon que s'il a prévu et autorisé cette possibilité
 - lecture dans un tube nommé
 - écoute d'une socket (apache)
 - test périodique du contenu d'un fichier (comme cron)
- si c'est permis, **ATTENTION** aux attaques sur des données incorrectes!!! (rappel: un démon peut être root)



Créer un démon

- règle n°6: un démon ne doit pas gêner le système de fichiers
 - s'il a besoin d'accéder à des fichiers particuliers, se placer dans le répertoire nécessaire
 - sinon, se mettre à la racine avec `chdir("/")` pour ne pas gêner un éventuel unmount



Créer un démon

- règle n°7: un démon ne doit pas être gêné par des signaux non attendus
 - bloquer tous les signaux qui ne sont pas nécessaire, en particulier SIGCHLD
 - politique standard des démons:
 - utiliser SIGHUP comme ordre de redémarrage
 - utiliser SIGTERM comme ordre d'arrêt propre



Les logs

- pour parler, les démons doivent utiliser des logs gérés par le démon *syslogd* (qui peut éventuellement les stocker à distance)
- `void openlog(const char *ident, int option, int facility);`
- `void closelog(void);`
- `openlog` prépare le démon à ajouter ses messages à `/var/log/XXX.log` avec le préfixe désigné par `ident`



Les logs

- si **option** vaut **LOG_PID**, cela inclura le PID du démon dans chaque message
- **facility** indique la nature du log pour savoir où le ranger
 - **LOG_DAEMON**: daemon.log
 - **LOG_USER**: user.log
 - **LOG_KERN**: kern.log
 - etc.



Les logs

- `void syslog(int priority, const char *format, ...);` `/usr/include/sys/syslog.h`

- fonctionne comme `printf`

- `priority` définit le type de message:

| | |
|--------------------------|--|
| <code>LOG_EMERG</code> | Le système est inutilisable |
| <code>LOG_ALERT</code> | Des actions doivent être entreprises immédiatement |
| <code>LOG_CRIT</code> | Les conditions sont critiques |
| <code>LOG_ERR</code> | Des erreurs se produisent |
| <code>LOG_WARNING</code> | Des avertissement se présentent |
| <code>LOG_NOTICE</code> | Condition normale, mais message significatif |
| <code>LOG_INFO</code> | Message d'information simple |
| <code>LOG_DEBUG</code> | Message de débogage |

- filtrage possible avec `setlogmask`



Gestion des logs

- sans précaution, un démon pourrait finir par saturer le système en remplissant indéfiniment le fichier de logs
- on peut définir une politique de *rotation de logs* avec **logrotate** pour:
 - compresser les anciens logs
 - les archiver par mail
 - supprimer les logs trop vieux
- cf. contenu de **`/var/log/`**



Notre démon

- conception d'un démon qui va écouter les changements dans un répertoire donné avec `inotify_add_watch`
- la tâche principale sera de faire des lectures bloquantes sur un descripteur prévenant des changements et d'écrire dans le fichier de logs

`my_daemon.cpp`



Les run levels

- niveaux de fonctionnement des systèmes UNIX, qui peuvent être de significations variables (sauf 0, 1 et 6):
 - 0: arrêt de la machine
 - 1: mono-utilisateur
 - 2: multi-utilisateur sans NFS
 - 3: multi-utilisateur
 - 4: inutilisé
 - 5: session X
 - 6: reboot



Les run levels

- on consulte le niveau courant avec **runlevel** ou **who -r**
- on change de niveau avec **telinit** ou **shutdown**
- les tâches à lancer au démarrage de la machine sont contrôlées par les scripts de **/etc/init/**
- le lancement des démons se fait grâce aux scripts de **/etc/init.d/**



Lancement d'un démon

- chaque démon est paramétré par un script dans `/etc/init.d/` (prototype = `/etc/init.d/skeleton`) qui permet de définir:
 - les run levels de démarrage et d'arrêt du démon
 - les commandes start, stop, restart, etc
 - la résurrection (respawn)
 - le PIDFILE
 - etc

`my_daemon.config`



Lancement d'un démon

- une fois le script écrit, on doit invoquer la commande `update-rc.d` pour linker le script dans les répertoires correspondant aux run levels de la forme `/etc/rc?.d/`:
 - `sudo update-rc.d my_daemon start 99 2 3 4 5 .`
- `start` = on veut démarrer le démon
- `99` = priorité pour un run level donné
- `2 3 4 5 .` = niveaux de démarrage