Cours Système n°6 : les *threads*

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Les *threads*

1.1. Fils d'exécution

Lorsqu'un processus est lancé, il lui est associé un unique fil d'exécution, ou *thread* « main » , qui, après une premiére série d'initialisations, exécute la fonction main() du programme. L'instruction **fork** permet de créer un nouveau processus, qui est une copie autonome du processus courant. Chaque copie possède son propre fil d'exécution et continue l'instruction suivant le **fork**. Au contraire, la création d'un nouveau *thread* consiste à générer un nouveau fil d'exécution à l'intérieur du processus courant. Le *thread* appelant continue à l'instruction suivante, alors que le *thread* appelé exécute une fonction spéciale lors de son lancement et se termine automatiquement au retour de cette fonction. Les fils d'exécution d'un processus s'exécutent de façon parallèle et ordonnancée, et travaillent essentiellement sur le même espace mémoire. Le contexte système est lui aussi très largement partagé. Pour cette raison, on désigne parfois les *threads* sous le nom de processus légers. Dans la conception d'une application devant exécuter plusieurs tâches en parallèle, le choix d'une architecture multi-threads et/ou multi-processus dépend donc :

- du besoin d'indépendance entre tâches (notamment en prévision d'un appel à **exec**, voir plus loin) ;
- du besoin de partage ou d'échange d'informations entre les tâches (travail coopératif) ;
- de considérations liées aux performances et à l'économie en ressources système.

Les solutions base de *threads* seront plus légères et à fortes interactions entre tâches. Les solutions à base de processus seront plus lourdes mais permettront une meilleure isolation, don une certaine robustesse (pour la gestion des signaux, etc...).

Les *threads* permettent :

- de simplifier l'écriture du code (chaque *thread* exécute une fonction simple associée à son rôle) ;
- d'utiliser des appels bloquants sans paralyser l'ensemble du processus ;
- de bénéficier, dans certains cas, des capacités des architectures multi-CPUs (vrai parallélisme entre deux *threads*).

On peut distinguer deux types de stratégies d'utilisation des *threads* par une application :

- l'application peut lancer un nombre prédéfini de *threads* pour exécuter en parallèle plusieurs tâches bien identiques : par rapport une application mono-threadée ;
- l'application peut aussi créer des *threads* à la demande, le plus souvent en limitant le nombre de *threads* actifs un instant donné ; c'est souvent la stratégie adoptée par les serveurs réseau qui doivent gérer plusieurs clients en parallèle tout en partageant des données dynamiques.

1.1.1 Gestion des threads par le système

Du fait de leur intégration au sein d'un même processus, les *threads* partagent :

- en couche utilisateur :
  - le code exécutable ;
  - les variables de la section données et du tas : constantes, variables globales et/ou statiques, variables allouées dynamiquement (malloc) ;
  - les bibliothèques et autres objets projetés en mémoire ;
- en couche noyau (contexte d'exécution) :
  - le pid et le ppid ;
  - les privilèges [1] ;
  - la table des descripteurs de fichiers ouverts ;
  - les verrous de fichiers ;
  - les informations relatives à l'utilisation de VFS (umask, etc...) ;
  - les timers (pour des noyaux postérieurs au 2.6.12 2) et les fonctions de traitement des signaux ;
  - les éléments relatifs aux terminaux ;
  - les limitations sur l'utilisation des ressources système ;
  - les statistiques d'exécution.

Les piles (variables locales non statiques, etc...) sont en revanche distinctes [2] pour éviter toute confusion lors des changements de contexte entre *threads*. Chaque *thread* possède également un identifiant propre, le **tid**, servant à le distinguer au sein de son processus et peut disposer d'une zone mémoire virtuellement (voire réellement) privée : *Thread-Specific Data* et/ou *Thread Local Storage* (TLS), cf. cours sur l'assembleur.

Du fait de leur inclusion au sein d'un même processus, les *threads* seront généralement tous affectés par l'invocation d'une fonction de niveau processus , et ce même si l'appelant n'est pas le *thread* main. Ainsi :

- **fork** duplique le processus mais ne recrée que le *thread* courant ;
- **exec**écrase le processus et tous ses *threads* ;
- **exit** ou la sortie du *main*() termine tous les *threads*.

De façon générale, on cherchera à éviter d'utiliser ce type de fonctions lorsque plusieurs *threads* sont en cours d'exécution. Les fonctions de modification du contexte (setuid, chroot, close, ...) affectent également les *threads*.

Comme les différents *threads* d'un processus partagent quasiment le même espace mémoire, il est indispensable de synchroniser les accès aux variables globales et aux fonctions manipulant des variables locales statiques. De plus, il n'y a pas de nettoyage automatique à l'arrêt d'un *thread*, les ressources allouées étant susceptibles d'être exploitées par d'autres *threads*. Il faut donc s'assurer, avant d'arrêter un *thread*, qu'il a bien relaché tous les verrous pris et désalloué tout ce qui n'est pas exploité par les autres *threads* (qui seront responsables de désallouer le reste).

1. à partir de la glibc 2.3.4, les changements d'identité sont en effet propagés à l'ensemble des *threads* via une signalisation en couche applicative.
2. Cette séparation fonctionnelle n'implique pas que la pile d'un *thread* soit protégée en cas d'accès accidentel par un autre *thread*

La terminaison « anormale » (crash) d'un *thread* entraîne elle de tout le processus (cf. cours sur les signaux). Ce n'est don pas une source de fuite de ressources non désallouées, l'exception des ressources persistantes allouées l'échelle du système entier (ex : fichier temporaire).

Du point de vue de l'ordonnancement, la gestion des *threads* dépend des choix du programmeur et des possibilités offertes par l'OS et les bibliothèques utilisées. Il existe trois possibilités :

- traiter les *threads* comme des tâches système à part entière et gérer leurs priorités ainsi que leurs temps d'exécution à l'échelle du système entier (c'est le cas naturel lorsque les *threads* sont gérés en couche noyau, comme sous Linux). On parle de modèle « 1 :1 ») ;
- partager le temps alloué à chaque processus entre ses *threads* ac tifs en ne montrant qu'une seule tâche système à l'ordonnanceur principal et en gérant les priorités des *threads* à l'intérieur de ce temps alloué (c'est le fonctionnement naturel lorsque les *threads* sont gérés en couche applicative comme dans la plupart des systèmes BSD). Il s'agit du modèle « M :1 » ;
- combiner les deux techniques précédentes et distribuer les *threads* du processus sur plusieurs tâches système gérées par l'ordonnanceur principal (modèle « M :N » , hybride, à deux étages d'ordonnancement).

Dans tous les cas, le changement de contexte entre *threads* est particulièrement rapide du fait du partage de la mémoire et du contexte.

1.1.2. État des threads

Les *threads* POSIX possèdent plusieurs variables d'état permettant de spécifier leurs modalités de terminaison :

- **la joignabilité** : on dit qu'un *thread* est joignable s'il est possible, pour un autre *thread*, de récupérer son code de retour ; un *thread* non joignable est dit détaché et son espace m moire sera immédiatement libéré lorsqu'il se terminera (pas de passage par un état que l'on pourrait qualifier de *thread* zombie) ;
- **l'annulabilité** : on dit qu'un *thread* est annulable s'il accepte les requêtes de terminaison que les autres *threads* sont susceptibles de lui envoyer ;
- **le mode d'annulation** : un *thread* annulable peut être immédiatement annulable (terminaison dès réception de la requête) ou être en mode d'annulation différée (examen des requêtes d'annulation à certains moments seulement).

Le choix de la joignabilité permet d'optimiser l'utilisation des ressources si on détache tous les *threads* autonomes. L'annulabilité et le mode d'annulation fournissent des leviers commodes pour éviter la terminaison d'un *thread* au mauvais moment (pendant qu'il effectue une opération d'écriture, qu'il détient un verrou, etc...).

2. Synchronisation (*threads* et processus)

Les processus d'un système disposant par défaut de ressources mémoire propres, leurs interactions sont relativement faibles. De plus, l'accès aux ressources fournies par le système et partageables entre processus est généralement synchronisé de façon transparente par le noyau. Le besoin de synchronisation explicite est don limité quelques cas (cf. les cas d'utilisation des verrous de fichiers). En revanche, l'accès aux ressources mémoire partagées entre *threads* d'un même processus est par nature non synchronisé. Il y a donc un travail explicite à fournir pour éviter les incohérences liées aux accès concurrents à ces ressources (exception : ressources initialisées au lancement puis exploitées en lecture seule par tous les *threads*).

2.1. Sections critiques

La protection des sections critiques est envisager l'échelle de tout un projet informatique, et pas simplement fonction par fonction. Si deux fonctions accèdent à une même variable globale autrement qu'en lecture seule, il faut identifer et protéger les sections critiques correspondantes par un même verrou. Ce verrou est nécessaire, même si chaque fonction n'est utilisée que par un seul *thread* (sauf si ce *thread* est le même pour les deux fonctions). Pour cette raison, on associe le verrou la ressource qu'il protège plutôt qu'à la section critique qu'il isole.
Les risques liés à l'utilisation de mécanismes de synchronisation sont principalement :

- **l'interblocage** (*deadlock*), qui correspond une situation dans laquelle deux acteurs (processus ou threads) sont chacun bloqués dans l'attente d'un événement que seul l'autre pourrait délencher, ce qui se produit le plus souvent lorsqu'ils prennent plusieurs verrous dans un ordre différent : si le déroulement du parallélisme leur est défavorable, ils peuvent chacun attendre indéfiniment un verrou possédé par l'autre (l'attente active ne résout pas le problême) ;
- **l'autoblocage** est une variante dans laquelle un unique acteur se bloque dans l'attente d'un événement que lui seul pourrait produire (ex : tentative de reprise d'un verrou qu'il possède déjà) ;
- la famine, qui peut se produire si les opérations de synchronisation protégeant les sections critiques favorisent systématiquement certaines sections ou acteurs : certaines tâches peuvent alors ne jamais accéder aux ressources qu'elles attendent.

2.2. Concepts de base

Cette section décrit quelques objets permettant de réaliser des fonctions de synchronisation (verrous ou autres). Les définitions données en pseudo-code sont des modèles, qui ne correspondent pas des implémentations réelles mais qui permettent de comprendre leurs principes (voir aussi les imperfections de ces modèles, mentionnées dans le texte). Noter également que, dans la pratique, les cles d'attentes sur ces différents objets sont rarement gérées comme des FIFO dans la mesure où le choix de l'acteur à débloquer peut dépendre de considérations liées la priorité et l'ordonnancement. Il faut donc considérer que l'opération de relâche et de reprise quasi-immédiate d'un verrou :

- ne garantit pas l'absence de famine vis-à-vis des acteurs qui attendent (le mode de gestion du verrou et le jeu de l'ordonnancement peuvent conduire à réattribuer le verrou au même acteur ; le modèle ne fonctionne de façon fluide que si les sections de code sans possession du verrou sont les plus longues) ;
- ouvre la possibilité que les ressources protégées par le verrou aient changé d'état entre la sortie de la première section et l'entrée dans la seconde (si l'ordonnancement a fait qu'un autre acteur a bénéficié d'un accès en ériture aux ressources entre temps).

2.2.1. Mutex

Le mutex (*MUTually EXclusive*) est un verrou qu'au plus un acteur peut posséder à un instant donné (verrou exclusif). Il permet donc de délimiter des sections critiques en assurant qu'au plus un acteur exécute l'une de ces sections à un instant donné. Cette exclusion de tout parallélisme dans les sections protégées est une restriction forte voire excessive dans de nombreux cas. Il est don souhaitable de la réserver des sections particulièrement courtes et d'utiliser des verrous mieux adaptés dans les autres situations (lecteurs/ écrivain, etc...). Seul le possesseur d'un mutex p eut le relâcher.

2.2. Sémaphore

Le sémaphore est un verrou particulier permettant d'indiquer la disponibilité d'un certain nombre de ressources. Les deux opérations de base se notent habituellement P (prise d'une ressource) et V (ajout d'une ressource). Il existe une troisième opération, I (initialisation du sémaphore) qui permet de spécifier le nombre de ressources initialement disponibles.

Ce document est inspiré du « Cours Système n°6 : les threads » délivré par monsieur Olivier Levillain dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Décembre 2012).