Comprendre les

Newbie

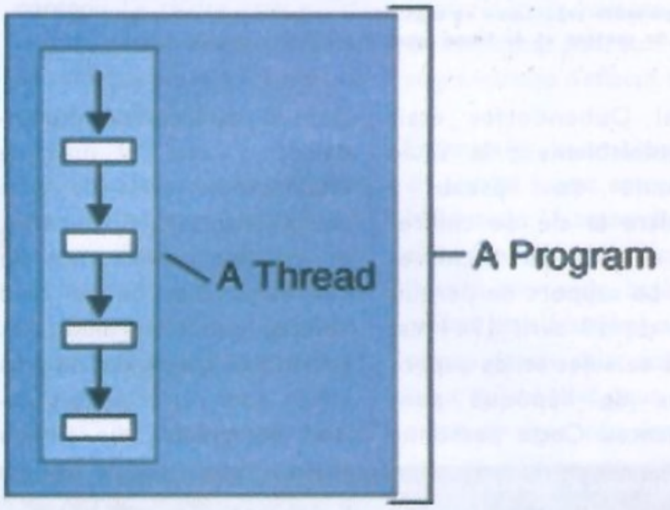
Voici l'essentiel de ce que vous devez de savoir sur les threads. Cet article montre pourquoi les développeurs de CPCNG veulent introduire cette notion dans le système d'exploitation qu'ils développent.

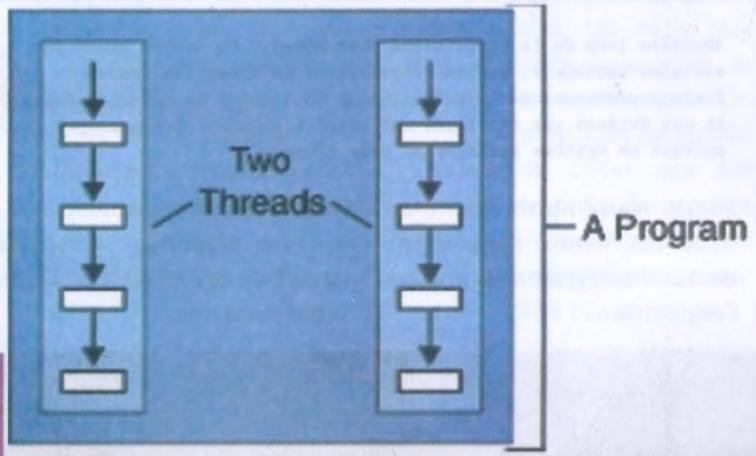
Présentation

Un coprocessus peut aussi se désigner par l'appellation processus léger ou thread (en Anglais). Il s'agit d'un fil d'exécution d'un programme qui est capable de s'exécuter en parallèle avec d'autres threads du même programme.

Sont notamment construites autour des threads les interfaces graphiques des programmes : en effet, souvent, les actions de l'utilisateur sur le logiciel par le biais de périphériques d'entrée comme le clavier ou la souris sont gérées par un coprocessus tandis que les tâches plus complexes (mise en page, transformation d'une image, génération d'un environnement 3D etc.) sont gérées par un autre thread. L'avantage ici est de permettre à l'utilisateur de continuer à interagir avec son environnement logiciel alors même que l'ordinateur est en train de travailler, et ce, de manière totalement transparente. On évite ainsi tout blocage. Il me serait en effet pénible de ne pas pouvoir continuer à écrire cet article sur mon traitement de texte sous prétexte que celui-ci, en même temps, a pour tâche de corriger les fautes d'orthographe !

Un programme peut donc se décomposer en plusieurs threads comme le montre les figures ci-dessous.





Les processus légers ont une partie commune; ils se partagent en effet des informations sur l'état du processus dont ils sont issus ainsi que sur diverses ressources.

Concernant les ressources justement, le fait que deux processus légers puissent être " nés " du même processus " père " peut faciliter le partage entre eux. Par contre, cela risque d'être davantage compliqué pour des threads issus de processus différents. Il est donc absolument nécessaire de mettre en place des systèmes de synchronisation, par le biais, par exemple, de sémaphores dont nous avons déjà parlé.

Fonctionnement

Créer un thread ne se fait pas de la même façon que la création d'un processus par le biais d'un fork comme nous avons pu le voir précédemment.

Ainsi, tous les coprocessus issus d'un même programme vont devoir être capables de partager le même espace d'adressage et les seules différences qui les caractériseront seront leurs identités, leurs piles d'exécution et certaines des informations systèmes qu'ils auront à leur disposition comme par exemple le masque des signaux, l'état des verrous ou encore des conditions.

Les coprocessus d'un même programme se regroupent pour former un groupe dans lequel ils seront tous traités de la même façon, à une exception près : le coprocessus principal qui a été créé au démarrage du processus principal (le programme) a davantage de valeur car sa terminaison provoque, à la fois, la clôture de tous les autres processus mais également la fer-

threads

meture du programme. La figure ci dessous nous montre un thread principal donnant naissance à d'autres coprocessus.

Il nous semble absolument nécessaire de bien préciser qu'un coprocessus ne doit pas être confondu avec une co-routine d'un programme. En effet, ces dernières (petites parties d'un programme exécutant une tâche précise) ne sont pas capables de s'exécuter en parallèle : tout à tour, une co-routine passe devant une autre et ainsi de suite. Les coprocessus, eux, sont aptes pour l'exécution en parallèle même s'ils peuvent être interrompus de façon préemptive par le système d'exploitation pour donner la main à d'autres coprocesus. Nous sommes donc ici dans une situation similaire aux processus dont nous avons parlé au cours d'un article précédent.

Partageant une mémoire commune, les coprocessus n'hésitent pas à s'en servir pour communiquer entre eux. La communication nécessite cependant que les deux coprocessus qui souhaitent communiquer le fassent lorsque la synchronisation entre eux est bonne. Cette dernière peut-être gérée efficacement par le biais de verrous et de conditions. Éventuellement, si nous mettons en place un système très évolué, nous pouvons opter pour une autre méthode plus efficace : celle des évènements.

Les différents appels

L'appel système create sert à créer un nouveau coprocessus tandis que la fonction exit sert à le terminer prématurément (suite à une décision de l'utilisateur). Le coprocessus principal est celui qui lance le premier d'autres coprocessus. S'il se termine, il exécute automatiquement une fonction dénommée pervasives.exit qui a pour but de clore à la fois le processus principal et tous ses coprocessus.

L'appel système join est utilisé pour permettre à un coprocessus d'en attendre un autre. Dès lors, le coprocessus appelant est suspendu jusqu'à ce que celui qui a été appelé se soit terminé. Notons que le coprocessus principal peut lui aussi lancer ce genre d'appel afin d'attendre que tous les autres coprocessus aient retourné une valeur avant de se terminer lui-même ou de terminer le programme. Ce genre d'appel bloquant peut-être interrompu par le biais d'un signal. S'ensuit alors une série d'opérations qui ne pouvaient se faire tant que le blocage existait. Puis, l'appel est relancé.

Notez cependant que le système d'exploitation peut suspendre un coprocessus afin de donner temporairement la main à un autre ou parce qu'il est en attente d'une ressource non disponible, soit utilisée par un de ses coprocessus, soit utilisée par un autre processus.

Utilisation des verrous pour la synchronisation entre coprocessus

Il existe un problème quand deux coprocessus veulent un accès concurrent à une même ressource. Prenons un compteur c et deux processus p et q qui incrémentent chacun en parallèle le même compteur c. Le coprocessus p lit la valeur du compter c puis donne la main à q qui, à son tour, lit la valeur de c. Puis il continue et écrit la valeur k+1 dans c. Le processus p reprend la main et écrit la valeur k+1 dans c. La valeur finale de c est donc k+1 au lieu de k+2.

La solution est d'utiliser des verrous qui empêchent l'entrelacement arbitraire de p et q. Par verrous, les programmeurs désignent des éléments partagés par le groupe de coprocessus issus d'un même programme ou processus. Il faut savoir qu'un seul coprocessus à la fois peut bloquer un verrou et qu'un verrou bloqué ne peut pas l'être une seconde fois (le coprocessus qui tenterait l'expérience serait gelé jusqu'au déblocage du verrou). C'est la fonction create qui créé le verrou en produisant un objet qui n'est pas bloqué. Ce fait est le résultat de la fonction lock. Pour libérer le verrou, il faudra utiliser la fonction unlock.

Les conditions

Cependant, les vérous ne peuvent pas nous satisfaire car, s'ils permettent d'attendre qu'une ressource ou donnée libre soit partagée, ils ne permettent pas d'attendre la forme précise d'une donnée.

Les conditions sont la solution à ce problème de forme. Ainsi, quand un coprocessus possède un verrou sur un objet, le coprocessus peut se mettre en attente jusqu'à ce que la situation souhaitée se réalise.

Les évènements

Pour faciliter l'exécution concurrente de coprocessus, il est possible d'établir une communication synchrone par le biais d'évènements. Ainsi, la communication se fait par le biais d'évènements au travers de canaux qui permettent de communiquer entre coprocessus d'un même processus.

Nous espérons que cette introduction aux threads, de notre point de vue, vous a permis de mieux saisir leur fonctionnement. Nous pourrons aller plus loin la prochaine fois. À bientôt !