QUESTIONS sur le TP10 / Commutation des tâches

Q1) Pourquoi le lancement d"une tâche utilisateur sur un coeur nécessite-t-il de charger l'adresse de la première instruction de la tâche dans le registre EPC, et d'exécuter l'instruction assembleur *eret* ?

Une tâche utilisateur est lancée soit directement par le code de boot (quand elle est la première à s'exécuter sur le coeur), soit par le code système chargé d'effectuer le changement de contexte. Dans les deux cas, le lancement est réalisé par du code système qui s'exécute donc en mode kernel :

- le bit UM du registre SR est à 0 => exécution sans aucune limitation d'accès aux registres protégés ou à la mémoire protégée.
- le bit IE du registre SR est à 0 => toutes les interruptions sont masquées.

Mais le code applicatif doit toujours s'exécuter avec des droits d'accès limités et avec les interruptions non masquées. Le changement de mode (donc le changement de valeur du registre SR) doit se faire exactement au moment du branchement à la première instruction du code applicatif, ni avant, ni après. Le rôle de l'instruction *eret* est précisément d'exécuter un branchement (copier la valeur contenue dans le registre EPC dans le registre PC), ET de modifier la valeur du registre SR, en une seule instruction.

Q2) En quoi l'architecture d'un processeur peut affecter le code de certaines fonctions de l'OS comme _task_switch() ?

Le code du GIET, comme le code de beaucoup de systèmes d'exploitation est écrit en C (le langage C a été conçu par les développeurs du système UNIX). Mais, pour que le code écrit en C soit portable sur différentes architectures de processeurs, le langage C ne permet pas au programmeur de manipuler directement les registres internes du processeur, car ces registres sont différents pour chaque architecture. L'abstraction proposée par le langage C suppose que toutes les variables manipulées par un programme sont stockées en mémoire.

La fonction _task_switch() a précisément pour but de sauvegarder en mémoire les valeurs contenues dans les registres de la tâche sortante, et de restaurer depuis la mémoire les valeurs des registres pour la tâche entrantes. Cette fonction ne peut donc pas être écrite en C, puisque l'ensemble des registres définissant le contexte d'une tâche dépend de l'architecture. Elle doit être écrite en assembleur, et ce code assembleur est forcément différent d'un processeur à l'autre.

Q3) Deux tâches A et B s'exécutent sur un même processeur. La tâche A fait un appel système qui déclenche une opération d'entrée/sortie utilisant le DMA. Avant que la tâche A soit notifiée que l'opération est terminée (par IRQ), elle est chassée du proc et remplacée par la tâche B, qui reçoit cette interruption du DMA. Comment l'OS peut informer la tache A alors qu'elle n'est pas en cours d'exécution ?

L'interruption signalant la fin de l'opération d'entrée/sortie n'est pas destinée à la tâche en cours d'exécution. Elle n'est pas destinée non plus à la tâche qui a demandé l'opération d'entrée/sortie. Elle est destinée au système d'exploitation, qui a déclenché l'opération d'entrée/sortie. L'interruption va *voler* quelques centaines de cycles à la tâche en cours d'exécution, qui peut être A ou B, peut importe. Pendant ces quelques cycles volés, le processeur va exécuter l'ISR DMA, associée à l'interruption DMA, qui va notifier au système d'exploitation la fin du transfert DMA en écrivant la valeur 0 dans la variable _dma_busy, qui n'appartient ni à la tâche A, ni à la tâche B, mais appartient au système d'exploitation. Quand la tâche A exécutera l'appel système (bloquant) _ioc_completed(), elle sera débloquée et pourra continuer son exécution.

Remarque: puisque le DMA est une ressource partagée susceptible d'être utilisée par plusieurs tâches, il faut un verrou pour protéger l'accès exclusif au DMA, et cette prise de verrou est normalement effectuée par l'appel système qui lance l'opération d'entrée/sortie.

Q4) Dans le GIET, les différentes tâches des applications sont statiquement lancées par le code de boot. Comment les OS généralistes supportent-t-ils le lancement dynamique de nouvelles applications ?

Dans un OS généraliste de type UNIX, ce n'est évidemment pas le code de boot qui lance les applications. Le scénario général est approximativement le suivant:

- le code de boot (boot-loader) initialise au moins deux périphériques (TTY et IOC) et charge en mémoire le code du système d'exploitation. Le code binaire de l'OS est dans un fichier *kernel.elf* stocké sur disque. Puis le boot-loader se branche à la fonction kernel_init() de l'OS qui va initialiser les structures de données du kernel.
- la fonction kernel_init() de l'OS initialise tous les périphériques disponibles dans l'architecture, initialise le système de fichiers (cache des fichiers), et crée un premier processus applicatif appelé *init*, puis lance l'exécution de ce processus. Le code binaire de cette première "application" est un fichier *init.elf* stocké sur disque.
- Le process *init* utilise les appels système fork() et exec() pour créer un ou plusieurs processus *shell*, qui sont des processus inter-actifs permettant à l'utilisateur de demander à l'OS d'exécuter certaines commandes. Le code binaire de l'application *shell* est un fichier *init.elf* stocké sur disque.
- A travers le *shell*, l'utilisateur peut demander à l'OS de lancer de façon interactive une ou plusieurs applications. Le code de chacune de ces applications doit être stocké dans un fichier *appli.elf* stocké sur disque. le processus *shell* utilise les appels système *fork()* & *exec()* pour créer et lancer le processus applicatif *appli*.
- Finalement, si le processus applicatif est un processus multi-threads, c'est l'exécution du thread *main* associé à chaque processus applicatif qui va créer les autres threads en utilisant l'appel système *pthread_create()*.

Q5) Le GIET, comme la plupart des OS multi-tâches, permet qu'un changement de contexte se produise sur un coeur, alors que ce coeur est en train d'exécuter du code système (suite à un *syscall*). Pourquoi les appels systèmes sont-il interruptibles ?

Il existe moins deux raisons: La première est que certains appels système peuvent avoir des durées d'exécution très longues, alors que certaines requêtes d'interruption exigent des réactions rapides. La seconde raison est que dans un OS supportant des changements de contexte présomptifs, il est préférable que les interruptions TICK qui déclenchent les changements de contexte ne soient pas masquées, au risque de biaiser le partage de la ressource processeur entre les différentes tâches.

Dans la plupart des OS, l'OS définit, de façon spécifique pour chaque appel système, quand cet appel système est interruptible.

Last modified on 10 Jun 2020, 21:41:51