Processus

Thèmes abordés
- Notions de base sur la mise en œuvre et la gestion des processus
- Ordonnancement des processus
- Présentation de l'API processus Unix

1 Questions

1. Comment les interruptions rendent-elles possible la multiprogrammation ?
2. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ? Justifiez votre réponse en expliquant le mécanisme de commutation de processus.
3. Précisez grâce à quel mécanisme un processus peut appeler les primitives du noyau mais ne peut pas se brancher n'importe où dans le code du noyau ?

2 Ordonnancement des processus

2.1 Problématique (rappel)

Cette section est consacrée à la comparaison et à la réalisation de différentes politiques d'ordonnancement à **court terme**, c'est-à-dire aux politiques d'allocation du (ou des) processeur(s) entre processus prêts [1]. Les politiques d'ordonnancement se distinguent par les éléments qu'elles intègrent pour déterminer le (prochain) processus actif (*élu*) :
- elles peuvent ou non se baser sur une **priorité** associée à chacun des processus ;
- elles peuvent **préempter** (réquisitionner) le processeur pour attribuer à un nouveau processus élu, ou bien laisser au processus élu l'initiative de rendre le processeur ;
- elles peuvent ou non déterminer le processus élu en fonction de **contraintes de temps réel**.

Les politiques d'ordonnancement peuvent être évaluées selon des critères traduisant différents besoins des applications :
- temps de service : temps moyen d'attente ;
- temps de réponse : garantie d'un délai d'attente maximum.
- équité (absence de famine) : tout processus prêt finit-il par obtenir le processeur ?

2.2 Exemples

Indiquez les caractéristiques et le niveau de réalisation des différents critères pour les politiques suivantes :
- FIFO : le processeur est alloué aux différentes tâches suivant l'ordre chronologique des demandes d'allocation.
- SJF (*Shortest Job First*) : chaque processus annonce sa durée d'utilisation du processseur. Le processeur est alloué au processus ayant la plus courte durée d'utilisation.
- Tourniquet (*Round-Robin*) : le processeur est alloué par quantum, à tout de rôle à chacun des processus
- EDF : (*Earliest Deadline First*) : chaque processus annonce une date maximum de terminaison (échéance). Le processeur est alloué au processus ayant l'échéance la plus proche.

Question. *Les notions d'équité (absence de famine) et de priorité sont a priori antagonistes. Comment introduire cependant une forme d'équité dans un système avec priorités ?*

1. Un processus *prêt* est un processus qui n'est pas bloqué en attente d'une ressource ou d'un événement de synchronisation. Il peut donc progresser immédiatement, dès lors qu'il obtient le processeur.

2.3 Mise en œuvre du tourniquet

Programmer (en pseudo-code) l'algorithme de principe du tourniquet.

On supposera que les processus sont identifiés par un entier (qui correspond à l'indice de leur descripteur dans la table des processus). On supposera de plus que l'on dispose

- d'une implémentation du type file
- d'une opération `commuter(courant : id_proc, nouveau : id_proc)` qui sauvegarde le contexte d'exécution du processus actif en cours (d'identifiant `courant`) pour installer/restaurer le contexte du processus d'identifiant `nouveau`, qui devient le nouvel actif.
- d'une opération `cadencer_horloge(délai : entier)`, qui programme l'envoi d'un signal d'horloge (d'identifiant `IT_HORLOGE`) toutes les `délai` ms après l'exécution de cette opération.

Donner

- le code du traitant associé à `IT_HORLOGE`,
- ainsi que le code d'initialisation situé dans le programme principal de l'ordonnanceur. Pour ce dernier, et pour être complet, vous pouvez supposer que vous disposez d'une opération `associer(sgn : entier ; traitant : fonction())`, qui permet de demander l'exécution de la fonction d'identifiant `traitant`, chaque fois que le signal `sgn` est reçu.

Le code que vous venez d'écrire est un exemple caractéristique de *programmation événementielle* (ou *réactive*).

2.4 Ordonnancement par partage équitable

Les politiques précédentes ne conviennent pas nécessairement à toutes les situations :

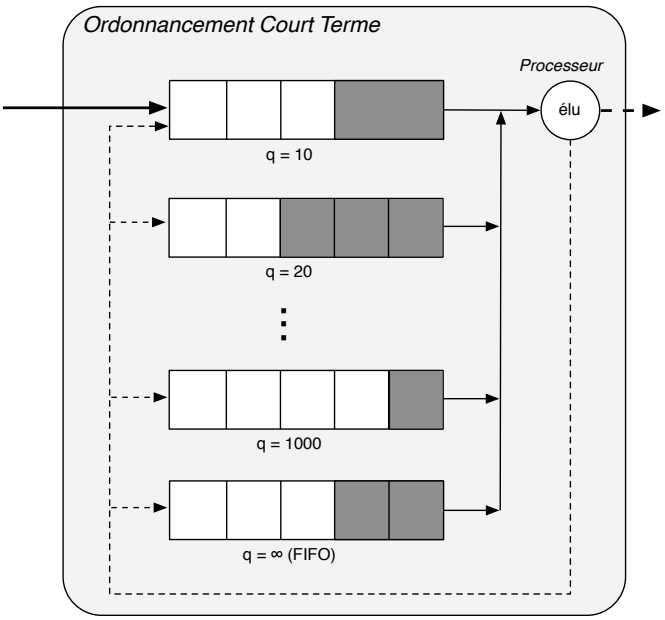
1. chaque politique d'ordonnancement est adaptée pour certains critères (équité, ou efficacité...) mais aucune ne l'est pour tous les critères. Il semble donc inadéquat d'appliquer un traitement indifférencié et critères identiques pour tous les processus, alors que les comportements et les besoins peuvent varier selon le profil des applications.
2. La disponibilité du processeur pour un processus donné peut être difficile à prédire, car elle peut dépendre du comportement et des caractéristiques des autres processus. Cet inconvénient peut être critique dès lors que les applications ont des contraintes de réactivité, de temps-réel.
3. La gestion des ensembles de processus en attente (listes...) peut s'avérer gourmande en temps de calcul (même si des optimisations sont possibles, et utilisées).

Les stratégies d'allocation multiniveaux (ou *hiérarchiques*) résolvent la première de ces restrictions. Leur principe est de regrouper les processus par catégories. Les processus d'une catégorie donnée ont un profil similaire vis à vis de l'utilisation du processeur : tâches de fond, tâches interactives, etc... Chaque catégorie a une politique d'allocation spécifique, adaptée au profil de ses processus. Par ailleurs, l'ordonnanceur alloue le processeur aux différentes catégories, en suivant une politique d'allocation globale, entre catégories.

La file multiniveaux est un exemple classique de cette classe de politiques d'ordonnancement. Cette politique considère plusieurs files, rangées par ordre de priorité. Chaque file est gérée selon une politique de tourniquet. Le quantum est lié à la priorité de la file : la file la plus prioritaire a le plus petit quantum, tandis que la file la moins prioritaire est un simple FIFO. L'ordonnancement global entre files suit une simple priorité : pour qu'un processus au niveau i soit élu, il faut qu'il n'y ait aucun processus prêt dans les files plus prioritaires que i. Un processus de niveau i actif est préempté en cas d'arrivée ou de reprise d'un processus plus prioritaire.

Une dernière caractéristique intéressante de la file multiniveaux est qu'il s'agit d'une politique **adaptative** : un processus de niveau i peut passer dans une file moins prioritaire dans le cas où il épuise son quantum, ou au contraire passer dans une file plus prioritaire s'il ne l'épuise pas. L'idée est qu'un processus finira par se ranger dans la file dont le quantum correspond le mieux à la durée de sa période de calcul moyenne.

Pour une stratégie d'allocation multiniveaux, si l'on considère l'allocation du processeur entre les différentes catégories, il est fréquent que certaines catégories correspondent à des profils d'applications demandant la satisfaction de prévisibilité, ou de contraintes temporelles (applications interactives, temps-réel...).



Les stratégies d'allocation par partage équitable visent à répondre à des besoins de prévisibilité en termes d'allocation du processeur. Nous allons étudier deux stratégies de base, appliquées aux processus, utilisées en particulier par Linux. Ces stratégies peuvent être étendues simplement pour la gestion de l'allocation entre catégories dans le cadre de l'allocation multiniveaux.

La question essentielle *(à laquelle vous pouvez prendre un peu de temps pour réfléchir...)* est donc :

Comment offrir à un processus donné une garantie d'accès au processeur, indépendamment de l'utilisation qu'en font les autres processus ?
Question subsidiaire : comment assurer cette garantie d'accès de manière efficace ?

L'algorithme de la loterie fournit une réponse originale et élégante à ces deux questions. Son principe est de mettre en circulation un nombre fixe de tickets, et de répartir ces tickets entre les différents processus. Le processeur est alloué par quantums de temps. A l'échéance du quantum, l'ordonnanceur tire un ticket aléatoirement. Le processus dont le ticket a été tiré est le nouveau processus élu.

Remarques :
- la répartition n'a pas à être nécessairement équilibrée, ce qui permet de réaliser une forme de priorité
- le nombre fixe de tickets permet de réaliser une forme de *contrôle d'admission*, ce qui est un outil de régulation important en cas de forte charge du système.

Le tirage aléatoire est généralement implanté de manière efficace. Le point important du point de vue des performances est donc d'assurer une distribution et une recherche efficace des tickets. Le principe est simple :
- les tickets sont numérotés de 0 à Max ;
- le descripteur de processus comporte le nombre de tickets alloués au processus ;
- les tickets attribués aux processus sont assimilés à des plages de valeurs successives : l'espace des valeurs [0 .. Max] est ainsi virtuellement partitionné en intervalles successifs, et un processus est (virtuellement) associé à chaque intervalle ;
  Exemple : 100 tickets sont émis. La liste des processus comporte (dans l'ordre) 3 processus : A avec 20 tickets, B avec 50 tickets, et C avec 30 tickets. On peut alors considérer que A détient les tickets 0 à 19, B détient les tickets 20 à 69, et C détient les tickets 70 à 99.
- lorsque le nombre N est tiré, la liste des processus est parcourue, en cumulant le nombre de tickets attribués. Le premier processus pour lequel le cumul est supérieur à N a le ticket gagnant.

Question 1. *Adapter l'algorithme pour améliorer l'efficacité de la boucle de recherche (autrement dit :*

minimiser le nombre d'itérations). Montrer que cette adaptation n'altère pas les chances d'accès au processeur de chacun des processus.

Le recours aux choix aléatoires permet d'allouer une fraction de temps processeur à chaque processus, indépendamment de l'utilisation qu'en font les autres processus, et ce de manière effective et efficace. Cependant, dans certaines situations, comme dans le cas de systèmes critiques, le caractère aléatoire de l'allocation peut s'avérer problématique. Des adaptations déterministes de l'algorithme de la loterie ont ainsi été élaborées, comme l'ordonnancement par pas (*stride scheduling*). L'ordonnanceur des versions récentes du système Linux fonctionne sur cette base.

L'ordonnancement par pas conserve le principe d'émission et d'attribution de tickets aux processus. Chaque processus est alors caractérisé par un **pas**, inversement proportionnel à son nombre de tickets. Par ailleurs, pour chaque processus, les pas sont cumulés chaque fois que le processus est élu. Le cumul des pas d'un processus représente donc la comptabilisation du temps processeur consommé par ce processus. Plus un processus a de tickets, plus la « facturation » d'un pas sera favorable.

L'algorithme d'ordonnancement proprement dit consiste simplement, à l'échéance du quantum, à parcourir la liste des processus pour trouver (et élire) le processus ayant le cumul de pas minimum (en cas d'égalité, l'identifiant de processus est utilisé pour départager et rester déterministe)

Question 2. *Est ce vraiment mieux que la loterie ? Autrement dit : quel est le coût de l'introduction du déterminisme ?*

3 API processus Unix

Résumé et objectifs de la présentation L'interface programmatique (API, Application Programming Interface) de gestion des processus proposée par UNIX repose sur un modèle et un patron de conception élégants et efficaces, bien que surprenants au premier abord. Les opérations de base de l'interface programmatique sont tout d'abord présentées de manière simplifiée. Elles sont ensuite justifiées et illustrées à travers la présentation du protocole d'usage qui leur est associé. Enfin, quelques éléments (rassurants) sont donnés quant à l'efficacité de la mise en œuvre de l'API.

Déroulement

- Introduction : descripteur de processus UNIX (planche 3)
- Gestion des processus : `fork`, `exec`
  - fork : principe = clonage ; (planches 5,4)
  - exercice 2.2.5 (3 forks)
  - discrimination par la valeur de retour
  - premier schéma d'usage :
    `id_fils = fork() ; si (id_fils=0) alors code_fils() sinon code_père() ;`
    (planche 6)
  - recouvrement (exec...) : motivation = modularité + programmation du contexte d'exécution d'un programme avant de lancer ce programme.
- Synchronisation père/fils : `wait`, `exit`
  - planches 10-13
  - schéma du shell.
- Autres : `getpid`...
- *Elégance* ***et*** *efficacité* : copie sur écriture (*c.o.w : copy on write*). Le clonage de l'image mémoire réalisé par le fork() est en réalité virtuel : le père et le fils partagent en lecture la même image mémoire, et la duplication effective n'a lieu qu'en cas de modification (accès en écriture), et ne porte dans ce cas que sur le fragment de mémoire (la page) modifié. Ce mécanisme est une bonne illustration du principe d'évaluation paresseuse.