Systèmes d'exploitation
Notes de cours

Loris Croce

Table des matières

1 Gestion des processus. à la Linux **1**
1.1 Politique d'ordonnancement 2
1.2 Automate des états du processus 2

2 Le dîner des philosophes **3**

3 Communication entre processus **5**

4 Sémaphores **5**

5 Sections critiques (Communication) **6**
5.1 Quelques exemples 6
5.2 Mécanismes 7
5.3 Rappel gestion des interruptions 7
5.4 Quelques conditions 7
5.5 Variables de verrouillage 7
5.5.1 Solution 1 Masquage des interruptions 8
5.5.2 Solution 2. variable de Petersen 8

6 Autre **9**
6.1 Mémoire 9

1 Gestion des processus *à la Linux*

Multi-processus, à chaque instant t, un seul processus sur le processeur → Le système fait l'entrelacement entre les processus, ce qui donne une illusion de parallélisme. → Gestion de priorité avec un ordonnanceur

Pour gérer ceci on utilise une table des processus (lorsque l'on fait un appel système à fork c'est l'index dans cette table qui est retourné).

Par exemple un case de cette table peut contenir 3 classes d'information :

— informations sur la gestion du processus : valeurs registres, compteur ordinal, pointeur sur la pile, numéro de processus parent, tableau d'interruptions, ...

— informations sur la mémoire : pointeur segment du code, pointeur segment données, pointeur segment pile
— informations sur les fichiers : description de fichiers, identifiant groupe utilisateur, ...

Pour ces informations une partie doit se trouver en mémoire (autant que possible) (autant que possible) ces informations sont regroupées en 2 catégories :

— Celle pour le système (à ne pas déplacer en disque).
— Celle utilisateur que l'on n'a pas besoin de garder en mémoire.

Lorsque le système décide changer de processus, il fait ce que l'on appelle un changement de contexte. Cela consiste à stocker les informations du processus courant et à charger les informations du processus suivant (choix de ce dernier délégué à l'ordonnanceur) et à demander au processus de commencer l'exécution du processus choisi.

Exemple d'opérations

— Au niveau matériel on stocke le compteur ordinal ; ...
— Au niveau matériel on charge le nouveau compteur ordinal (pour exécuter du code du Système d'exploitation)
— Par exemple à ce stade on peut continuer le stockage des informations du processus courant.

On peut commencer l'exécution du code du système d'exploitation.

— chargement pile (au niveau assembleur par exemple)
— typiquement l'ordonnanceur qui choisit le prochain processus)

On retourne au niveau assembleur pour charger les informations du processus courant et commencer son exécution.

1.1 Politique d'ordonnancement

L'ordonnanceur choisit le processus à exécuter en maintenant souvent les structures de données suivantes :

— le processus courant
— la liste des processus prêts à s'exécuter, en attente, zombies, ...(ceci peut-être codé avec un drapeau)
— la liste des interruptions. Pour chacune les processus en attente (et peut-être traitement à faire !)

1.2 Automate des états du processus

L'ordonnanceur choisit à chaque changement de contexte un paramètres les prêt à s'exécuter, il peut choisir 1 politique de :

— FIFO le processus prêts à s'exécuter sont dans une file. Inconvénient le processus avec beaucoup d'appels bloquants sont désavantagés
— On peut améliorer le FIFO en gérant des priorités. L est une file de priorités

— Anneaux : chacun a un temps qui lui est imparti et ce temps doit être utilisé avant 1 changement de contexte. On désavantage les autres qui ne sont pas en attente (ceux en attente occupant le processus sans rien faire)

Solution pour ne pas désavantager ceux qui ne son pas en attente :

— si 1 processus en attente et son temps pas terminé, donné la main à 1 autre.
— lorsque le processus en attente est réveillé :
  — soit lui donner la main automatiquement (avec le temps qui lui restait)
  — soit la prochaine fois lui ajouter ce temps à son temps imparti

Les priorités dépendant du nombre d'exécutions à exécuter (par exemple on choisit toujours le plus court on le 2 plus courts, ...)

Comment décider du nombre d'instructions à exécuter ? On fait des stats basées sur le passé.

L'ordonnanceur peut choisir 1 politique de la liste des pcs à s'exécuter = L.)

— Susteme interactif : on prend en compte les types d'instructions et on met à jour les priorités
  — On fait des statistiques basées sur le passé (et ou futur danger peut être instructions non autorisées)
— Mix : priorité + système interactif

2 Le dîner des philosophes

5 philosophes assis autour d'une table et on commandé un plat (supposé infini) de spaghettis. Cinq fourchettes sont disposées autour de la table, une entre chaque assiette. Chaque philosophe passe alternativement dans 3 états : *en train de penser, affamé, en train de manger*. Tout d'abord, il pense pendant un temps aléatoire. Puis, il est affamé et attend que la fourchette à sa gauche et que la fourchette) sa droite se libèrent. Ensuite, il prend les deux fourchettes et commence à manger pendant un temps aléatoire. Quand il a fini de manger, le philosophe pense (jusqu'à ce qu'il soit à nouveau affamé).

Question 1 *Pourquoi chaque philosophe est-il modélisé par un processus ? Est-ce qu'autre chose est modélisé par un processus ?*

Un philosophe est un processus car il effectue des calculs. Non les autres éléments sont des ressources.

Question 2 *Quelles sont les ressources critiques ?*

La fourchette.

Question 3 *Quand un philosophe doit-il demander la ressource ?*

Quand il est affamé.

Question 4 *Quand un philosophe doit-il libérer la ressource ?*

Quand il a fini de manger.

Question 5 *Quelle est la section critique ?*

C'est “Manger", c'est la seule fonction qui utilise des ressources critiques.

Question 6 *Pour protéger l'utilisation des fourchettes dans la section critique, nous allons implémenter une sémaphore par fourchette ? Décrivez le pseudo-code d'un programme qui implémente le dîner des philosophes de la manière suivante :*

— *Chaque philosophe pense pendant un temps aléatoire,*

— *Chaque philosophe mange pendant un temps aléatoire,*

— *L'accès aux ressources critiques est protégé.*

Exclusion mutuelle sur chaque fourchette variable de verrouillage

```
while(true) {
    penser();
    prendre_semaphore_gauche();
    prendre_fourchette_gauche();
    prendre_semaphore_droite();
    prendre_fourchette_droite();
    manger();
    liberer_fourchette_gauche();
    liberer_semaphore_droite();
    librerer_fourchette_gauche();
    liberer_semaphore_gauche();
}
```

Question 7 *Qu'est-ce qu'un interblocage ?*

Quand aucun processus ne peut entrer en section critique (ici manger). Il se produit lorque tout les philosophes ont faim et prennent tous la fourchette

Question 8 *Est-ce que votre solution est susceptible de faire apparaitre un interblocage ?*

Oui (exemple du carrefour priorité à droite).

Exercice 2 *Est-ce que l'interblocage est susceptible d'arriver en pratique ?*

Oui même si la probabilité est très faible.

Question 9 Oui

Question 10 *Comment faire en sorte que le problème soit résolu sans interblocage ?*

1. Un sémaphore pour les deux fourchettes
2. Un processus serveur que saura qui a la fourchette

3. Philosophe indice pair prenne la fourchette droite en premier, et indice impaire gauche

Ou

1. Ne donner que deux fourchettes à la fois, pas une.
2. Un processus “serveur" est chargé de distribuer les ressources. Accepter 2 au plus qui mangent

Question 11 *Chaque philosophe ne peut prendre que la fourchette à sa gauche et à sa droite. Combien de philosophes sont autorisés à manger simultanément . Écrivez le pseudo code correspondant.*

$\frac{n}{2}$

3 Communication entre processus

Question 13 *Il existe plusieurs mécanismes de communications entre processus : les fichiers, les signaux, les sockets, les tubes nommés, les tubes anonymes ; ou encore les segments de mémoire partagée. Pour chacun de ces mécanismes, indiquez :*

- *Si les processus communicants peuvent être sur des machines différentes*
- *Si les processus communicants doivent avoir une relation de parenté*
- *Si les opérations de lecture et d'écriture doivent être réalisées en exclusion mutuelle*

Table 1 – My caption

	Processus peuvent être dans des machines différentes	**Si processus communiquent doivent avoir une relation parenté**	**Operation de lecture/écriture en exclusion mutuelle**
fichiers	oui / non	non	oui
signaux	non	non	non
sockets	oui	non	non
tubes donnés	oui / non	non	oui
tubes anonymes	non	oui	oui
segments de mémoire partagée	non	non	oui (au moins pour l'écriture)

4 Sémaphores

Question 14 *exclusion mutuelle avec sémaphores*

```
Exclusion mutuelle
  mutex = 1
  p(mutex)
  // section critique
  v(mutex)
```

(...)

Question 16 *On considère une barrière de synchronisation particulière : les processus A et B s'exécutent indépendamment, mais une certaine partie du code de B doit obligatoirement s'exécuter après une certaine partie du code de A. Implémentez cette barrière particulière.*

```
mutex = 0;
Processus A;
// tache non importante pour B
// tache importante pour B (section critique)
v(mutex);
// continuer traitements

Processus B;
// taches non critiques
P(mutex);
// taches critiques
```

(...)

5 Sections critiques (Communication)

On dispose d'1 zone et on ne veut pas que 2 ou plus processus "l'utilisent" en même temps.

Section critique : partie du code exécuté exclusivement par les processus désignés à partager la ressource. Il y a une **"exclusion mutuelle"** lorsqu'un seul processus est habilité à accéder à la ressource.

5.1 Quelques exemples

1. pas 2 en même temps (en cas d'exclusion mutuelle) ou i (si i est la borne)
2. indépendant des vitesses des processus et du nombre de processus
3. seul 1 processus en section critique bloque les autres
4. On ne doit pas attendre indéfiniment l'accès à la section critique

5.2 Mécanismes

Masquage des interruptions, variables de verrouillage, sémaphores.

5.3 Rappel gestion des interruptions

1. Il existe plusieurs types d'interruption : 0 → horloge, 1 → disque, 4 → tmp (appels systèmes), ...
2. Une table d'interruptions qui appelle la routine associée à chaque interruption

5.4 Quelques conditions

Ces différentes étapes de la gestion des interruptions :

1. sauvegarde le compteur ordinal, détermine le type d'interruptions et on passe en mode noyau pour permettre l'exécution de la routine associée.
   (a) Dans cette routine on peut stocker le contexte du processus, (en ASM)
   (b) On appelle le vrai code la routine (en général mix C, assembleur)
   (c) Au retour de la procédure on retourne le contexte du processus appelant si peut continuer, sinon on passe la main à l'ordonnanceur.

Masquage des interruptions (bloque les interruptions) et permet l'exécution d'un code en mode atomique.

1. une instruction spéciale pour bloquer les interruptions
2. on l'invoque avant d'entrer en section critique
3. réactive les interruptions en fin de de section critique

Pas acceptable comme solution puisque qu'un processus peut s'accaparer toutes les ressources. Mais, reste une solution acceptable pour l'OS (de temps en temps) dans **un système mono-processeur**.

5.5 Variables de verrouillage

On définit une variable par **section critique** : c'est la variable qui conditionne à la section critique.

```
verrou = 1;
while (verrou == 1);
verrou = 1;
section_critique();
verrou = 0;
```

Cette solution n'est pas correcte car on n'a aucune garantie que la modification de verrou est atomique → il faut trouver un moyen de contourner la non-atomicité.

5.5.1 Solution 1 Masquage des interruptions

Par exemple le langage dispose d'une instruction qui permet de faire des affectations atomiques

Solution : Protéger nous-mêmes l'accès à la variable avec une seconde variable

1ère possibilité : une variable tour

```
while (tour != p)
    section_critique();
    tour = 1-p;
```

le processus le plus lent est attendu par les autres.
Vous comprendrez que cette solution ne marche que pour 2 processus.

5.5.2 Solution 2 *variable de Petersen*

```
int drapeau = {F, F};
int tour = 0;
enter_sec(int i) {
drapeau[i] = T;
tour = i;
while (drapeau[i+1]%2 == F || tour = (i+1) % 2)
    esc();
    sortie_esc(int i);
    drapeau[i] = F;
}
```

L'étendre à n processus ? Pour une extension à n processus, m processeurs (avec un mécanisme de variable d'alternance).

On dispose d'une instruction matérielle `test, set , lock` $\rightarrow$ changer le contenu d'une mémoire. On met une valeur non nulle à cette zone mémoire, l'atomicité garantit par un verrouillage du bus de données.

```
TSL(4 verrou) {old = verrou; verrou = 1; return old;}
enter_sl() {
    while (TSL(4 verrou))
        sortie_sc() {
            verrou = 0;
        }
}
```

6 Autre

Processus Programme qui tourne en machine
- — ensemble d'instruction et de données
- — structure alloué par le système pour le contrôleur

Un sémaphore est une **variable** partagée par différents **acteurs**, qui garantit que ceux-ci ne peuvent y accéder que de façon séquentielle à travers des opération atomique, et constitue la méthode utilisé couramment pour restreindre l'accès à des ressources partagées.

Un processus s'apparente à un programme, un processeur effectue des calculs. Les processus donnent des instructions au processeur.

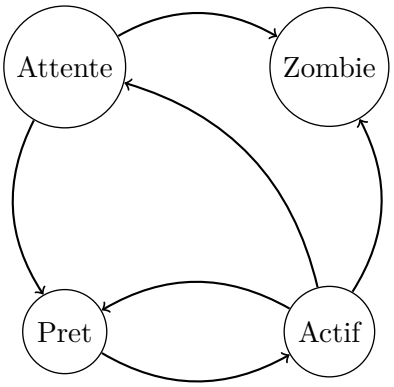

FIGURE 1 – États d'un processus

6.1 Mémoire

Algorithme de gestion de mémoire par liste chaînée : blocs de taille t, numérotés de 1 à b. Liste chaînée d'éléments : $\begin{cases} \text{bloc de début} \\ \text{nombre de blocs} \\ \text{drapeau libre} \end{cases}$

```
nbr_nece = entier_sup(n/t)
pour chaque element e de liste
 si e est libre :
    si e.nb_blocs >= nb_nece
      marquer occupé
    sinon
      divisiée e en blocs :
        - marquer le 1er occupé de taille nb_nece
        - le 2eme reste libre libre nouv;
```

```
        insererElement(liste, nouv)
    return e;
return null;
```