Résumé du chapitre 2
Gestion des activités parallèles

Jacques Mossière

3 octobre 2006

1 Introduction

Ce chapitre est consacré à la gestion des activités parallèles et à la notion de processus. La partie 2 présente les notions fondamentales ; la partie 3 décrit les possibilités offertes par Unix ; enfin, la partie 4 traite de la spécification des problèmes de synchronisation et de leur programmation à l'aide de moniteurs.

La réalisation concrète d'un noyau permettant l'exécution de processus parallèles est abordée dans le prochain chapitre.

2 Définition des processus

2.1 Motivation

Dans les calculateurs modernes, plusieurs activités peuvent se dérouler en simultanéité réelle :

- plusieurs programmes dans le cas d'un ordinateur multiprocesseur,
- un programme et un certain nombre d'opérations d'entrée-sortie dans le cas d'un monoprocesseur.

D'autre part, tout se passe comme si les différentes applications lancées sur un ordinateur individuel, ou sur un serveur par différents clients, se déroulaient simultanément.

On parle dans le premier cas de parallélisme réel, dans le second de pseudo-parallélisme. L'objectif est de définir un modèle permettant d'une part de décomposer un « calcul » en plusieurs entités susceptibles de s'exécuter en parallélisme réel ou non et, d'autre part, de décrire les contraintes que doit respecter l'exécution de ces entités.

2.2 Processus : définitions

Etant donné un programme, nous appelons **processus séquentiel**, ou plus simplement **processus**, l'entité qui correspond à l'exécution de ce programme. Alors que le programme est un élément statique, le processus associé est une entité dynamique qui démarre lorsqu'on lance l'exécution, s'arrête lorsqu'on décide de lui retirer l'unité centrale et se termine lorsque l'exécution du programme est terminée.

Concrètement, un processus est défini par :
- un espace mémoire contenant le code, les données et la pile,
- un compteur ordinal,
- un ensemble de registres.

En admettant que l'espace mémoire d'un processus est alloué statiquement, lancer l'exécution d'un processus revient à initialiser les registres du processeur et à charger dans le compteur ordinal le point d'entrée du programme. Pour stopper temporairement l'exécution d'un processus, il suffit de sauvegarder les registres généraux et le compteur ordinal ; on le relance en chargeant registres et compteur à partir de la zone de sauvegarde.

Le système d'exploitation conserve les données des différents processus dans une table, la table des processus. Chaque processus est désigné par un identificateur unique, son pid (« process identifier »), qui sert d'entrée dans la table des processus.

2.3 Relations entre processus

Des processus peuvent être logiquement indépendants (par exemple, des processus créés sur un serveur par des usagers différents) ou coopérer pour la réalisation d'un même travail. Dans le premier cas, les seules relations entre processus proviennent de conflits pour l'utilisation des ressources, et en particulier de l'unité centrale. L'expression de la coopération entre processus sera étudiée au ğ 4.

2.4 Etats des processus

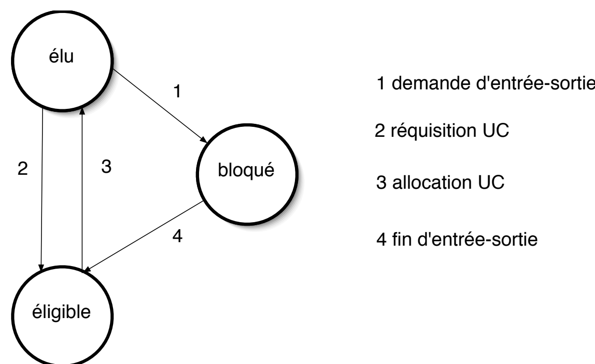

FIG. 1 – Etats des processus

Un processus peut se trouver dans l'un des états suivants (figure 1) :
- **élu** : un processus dans l'état élu s'exécute effectivement sur un processeur ; il existe donc au plus un processus élu par processeur ;
- **éligible** : un processus éligible est un processus qui pourrait s'exécuter si un processeur était disponible ;
- **bloqué** : un processus est dans l'état bloqué s'il ne peut pas s'exécuter, par exemple parce qu'il est en attente d'une fin d'entrée sortie

Les transitions peuvent s'interpréter de la façon suivante :

1. le processus élu fait une demande d'entrée sortie
2. l'unité centrale est retirée au processus élu qui devient éligible
3. choix pour éxécution d'un processus éligible qui devient élu
4. lors d'une fin d'entrée sortie, un processus bloqué passe dans l'état éligible.

3 Gestion des activités parallèles sous Unix

3.1 Introduction

On donne dans ce paragraphe une vue d'ensemble des facilités d'Unix en matière de programmation parallèle. Seules seront décrites les possibilités des principaux appels système. Pour une vision plus précise, on pourra se reporter au livre de Tanenbaum,« Systèmes d'exploitation », Pearson education, chapitre 10, ou, pour une étude systématique, à l'ouvrage de Rifflet et Yunès« Programmation et communication sous Unix », Dunod, 2003, ou tout simplement utiliser la commande man.

3.2 Les processus sous Unix

La notion de processus est une des notions de base de l'ensemble des systèmes Unix, comme de tous les systèmes modernes. Dès qu'un utilisateur se connecte à un système Unix, un premier processus est créé et exécute un interprète du langage de commande (un « shell »). L'exécution de la plupart des commandes entraîne la création d'un processus chargé d'exécuter la commande.

Au niveau du langage de commande, trois modes de création sont classiques.

Création simple

`#toto`
lance comme un processus autonome l'exécution du programme `toto`. Pendant l'exécution de `toto`, le processus qui interprète les commandes est bloqué ; il reprend la main lorsque `toto` se termine.

Création en arrière-plan

`#toto &`
lance de même comme un processus autonome l'exécution de `toto`, mais l'interprète du langage de commande reprend immédiatement la main et s'exécute en parallèle avec le processus qui exécute `toto`.

Création de plusieurs processus connectés par tuyau (« pipe »)

`#toto | lulu`
crée deux processus pour exécuter en parallèle les programmes `toto` et `lulu`. Tous les caractères produits sur la sortie standard de `toto` sont envoyés sur l'entrée standard de `lulu`. Le processus associé à `lulu` est mis en attente s'il a besoin de données d'entrée non encore fournies par `toto`.

Le processus associé à `toto` peut être mis en attente si `lulu` ne lit pas assez vite ce que `toto` envoie.

Un processus Unix possède un espace mémoire isolé (un espace virtuel), un compteur ordinal et des registres. Tout processus est identifié par un numéro appelé le pid du processus. La commande ps permet d'obtenir la liste de tous les processus existants.

3.3 Création de processus

L'appel système `fork` permet de créer un processus par clonage du processus père : l'espace du processus créé est une **copie** de l'espace du père (mémoire, registres et diverses informations propres au processus). Une conséquence de ce clonage, c'est que deux copies du même programme s'exécutent après le `fork`, respectivement dans le processus père et dans le fils. Pour permettre de faire la différence entre les deux processus, la valeur retournée par `fork` au processus fils est par convention 0 alors que le père reçoit le pid du processus créé ou un code d'erreur. L'appel système `waitpid` permet au processus père d'attendre la terminaison du processus fils.

Exemple.

```
#include <sys/types.h>
#include <sys/wait.h>

main(){
  int spid,status ;
  switch (spid = fork (){
        case -1 : perror(...) ; exit(-1) ;
        case 0  : // code du fils
                       break ;
        default : // le père attend la terminaison du fils
                  if (waitpid(spid,&status,0) == 1) {perror(...) ; exit(-1) ;}
  }
}
```

Un processus est détruit lorsque l'exécution du programme associé se termine. L'appel système `kill` permet la destruction explicite d'un processus, par exemple en cas d'erreur.

Un processus peut changer dynamiquement le programme qu'il est en train d'exécuter par l'intermédiaire de l'appel système `exec`. Cet appel système modifie l'espace mémoire affecté au processus et place dans le compteur ordinal l'adresse du point d'entrée du nouveau programme. La séparation de la création d'un processus et du changement du programme permet d'effectuer des traitements entre les deux phases, par exemple pour la création des tuyaux.

3.4 Processus légers

La création d'un processus par clonage du processus père est une opération coûteuse en temps d'exécution (création et initialisation d'un nouvel espace virtuel). Par ailleurs, l'isolation d'un processus dans un espace autonome, si elle fournit des garanties de sécurité, empêche toute communication simple entre deux processus qui désirent coopérer : ils doivent passer par l'intermédiaire de fichiers, de tuyaux ou de mécanismes généraux de communications interprocessus.

Pour pallier ces inconvénients, les systèmes Unix modernes fournissent un second niveau de gestion des activités parallèles : les processus légers (ou encore fils d'exécution ou « threads »).

Des processus légers cohabitent dans l'espace mémoire du processus qui les a créés, chacun d'eux disposant de sa pile et de son exemplaire des registres. Ils peuvent donc se transmettre des informations par l'intermédiaire de la mémoire qu'ils partagent. Différentes spécifications des processus légers ont été proposées ; la plus populaire est sans doute celle définie dans la norme Posix que nous allons utiliser en TP.

4 Synchronisation entre processus

Nous supposons dans ce paragraphe qu'un ensemble de processus ont accès à une même mémoire commune soit pour contrôler leurs exécutions, soit pour échanger des données. Nous commençons par montrer, sur deux exemples simples, comment spécifier un problème de synchronisation ; puis nous introduisons la notion de moniteur qui permet de traduire facilement une spécification en algorithme. Nous étudions ensuite un modèle classique de communication de données entre processus, le modèle du producteur et du consommateur, et nous le résolvons par moniteur.

4.1 Spécification, points de synchronisation

4.1.1 Exemples

Les deux exemples ci-dessous sont empruntés à S. Krakowiak, Principes des systèmes d'exploitation, Dunod, 1985.

Exemple 1 : lecture écriture.

Un premier processus, le processus de calcul, effectue un traitement, dépose le résultat du traitement dans un tampon, puis se termine. Un second processus, le processus d'impression, prélève le résultat dans le tampon, l'affiche sur l'écran puis se termine. Les deux processus sont lancés simultanément.

Dans cet exemple, le processus de calcul s'exécute sans contrainte. Au contraire, le processus d'impression doit, après une éventuelle phase d'initialisation, attendre que le processus de calcul ait déposé son résultat dans le tampon. Le point où le processus d'impression doit attendre est appelé un **point de synchronisation**.

Exemple 2 : rendez-vous de N processus.

N processus sont lancés simultanément. Dans le programme de chaque processus se trouve un point particulier, dit point de rendez-vous, que le processus ne peut franchir que si tous les processus sont arrivés à leur point de rendez-vous. Autrement dit, lorsqu'un processus arrive à son point de rendez-vous, il doit tester le nombre de processus arrivés ; s'il n'est pas le dernier, il doit se mettre en attente sinon il doit réveiller tous les autres qui sont arrivés avant lui.

Dans cet exemple, chaque processus contient un point de synchronisation, le point de rendez-vous.

4.1.2 Spécification d'un problème de synchronisation

Spécifier un problème de synchronisation, c'est tout d'abord identifier les points de synchronisation. C'est ensuite décrire l'état du système à l'aide d'un ensemble de variables d'état. C'est

enfin exprimer, en fonction des valeurs des variables d'état et pour chaque point de synchronisation, la condition qui doit être vérifiée pour qu'un processus soit autorisé à franchir le point de synchronisation.

Exemple 1. Un booléen, `fini`, reçoit initialement la valeur `faux` ; il est mis à vrai par le processus de calcul lorsqu'il a déposé le résultat dans le tampon. La condition pour que le processus d'impression puisse franchir le point de synchronisation est simplement
`fini = vrai`.

Exemple 2. Un entier `n` est initialement à `0`. Lorsqu'un processus arrive au point de rendez-vous, il incrémente `n` de `1`. La condition pour franchir le point de synchronisation est `n=N`.

4.1.3 Exclusion mutuelle aux variables d'état

On peut constater sur les exemples précédents que plusieurs processus peuvent avoir accès en lecture ou en écriture aux variables d'état. Ces accès peuvent être atomiques si une seule instruction permet de modifier ou de tester une seule variable ; ils peuvent aussi demander plusieurs instructions opérant sur une ou plusieurs variables. Il faut alors assurer que les transformations des variables d'état par plusieurs processus produisent un résultat cohérent.

Exemple. Reprenons l'exemple 2 ci dessus et examinons l'incrémentation de `n`. Cette incrémentation peut être compilée par un chargement de la valeur de `n` dans un registre, une incrémentation de ce registre, puis un rangement du registre dans `n)`.

En assembleur, on obtient le programme ci-dessous :

```
movl        n,%eax        (1)
inc         %eax          (2)
movl        %eax,n        (3)
```

Supposons que deux processus arrivent presque en même temps au point de rendez-vous et que chacun d'eux exécute l'instruction (1). Quelle sera alors la valeur finale de `n` ? Alors que nous aurions souhaité que `n` soit augmenté de `2`, le programme provoquera une augmentation de `1`.

Pour éviter cette difficulté, les instructions (1), (2) et (3) doivent être exécutées toutes les trois par un processus avant qu'un autre puisse y avoir accès. Nous dirons qu'une telle séquence doit être exécutée en **exclusion mutuelle**, ou encore qu'une telle séquence constitue une **section critique**.

Passer de la spécification d'un problème de synchronisation à sa réalisation pratique implique d'une part de garantir que les sections critiques sont bien exécutées en exclusion mutuelle et, d'autre part, de fournir des instructions qui assurent le blocage ou le réveil des processus à leurs points de synchronisation. Nous allons présenter au paragraphe suivant une construction permettant de prendre en compte ces deux aspects. Cette construction a été introduite par Hoare (Comm. ACM, oct. 1974).

4.2 Moniteurs : définition

On appelle **moniteur** un ensemble de données et de procédures. Les données d'un moniteur ne sont accessibles que par les procédures du même moniteur. De plus, **les procédures d'un moniteur s'exécutent en exclusion mutuelle**. Autrement dit, un seul processus peut s'exécuter dans un moniteur à un instant donné.

Le blocage et le réveil des processus sont assurés par l'intermédiaire de variables spéciales appelées **conditions**. Des conditions ne peuvent être déclarées que dans un moniteur. Deux opérations sont définies sur les conditions, appelées respectivement `attendre` et `signaler`.

Soit `c` une variable condition.

Lorsqu'un processus exécute `c.attendre`, il se bloque « en attente de la condition `c` ». Lorsqu'un processus exécute `c.signaler`, un processus quelconque en attente de `c` est réveillé ; si aucun processus n'est en attente de `c`, `c.signaler` n'a aucun effet.

Ces deux opérations doivent être compatibles avec le respect de l'exclusion mutuelle aux procédures du moniteur :

- lorsqu'un processus se bloque par `c.attendre`, l'exclusion mutuelle au moniteur est relâchée pour permettre à un autre processus de venir le réveiller.
- lorsqu'un processus P réveille un processus Q par un `c.signaler`, le processus P se bloque et c'est Q qui poursuit l'exécution. P sera réveillé lorsque Q quittera le moniteur.

Nous pouvons maintenant définir des moniteurs résolvant les exemples précédents.

Exemple 1

Nous introduisons un moniteur sync contenant deux procédure, `fin_ecrire` et `debut_lire`. Lorsque le processus de calcul a déposé son résultat dans le tampon, il exécute `sync.fin_ecrire` ; avant de prélever un résultat, le processus d'impression exécute `sync.debut_lire`.

```
sync : moniteur ;
var fait : booleen ; fini : condition ;
procedure fin_ecrire ;
   debut
   fait := vrai ; fini.signaler
   fin ;
procedure debut_lire ;
   debut
   si non fait alors fini.attendre
   fin ;

fait := faux ;     /* initialisation */
fin sync ;
```

Si le processus de calcul entre le premier dans le moniteur, il positionne le booléen fait et l'exécution de `fini.signaler` est vide. Lorsque le processus d'impression exécute `début_lire`, la valeur du booléen `fait` lui évite de se bloquer. Inversement, si c'est le processus d'impression qui arrive en premier, il se bloque en attente de la condition `fini` et sera réveillé par le processus de calcul.

Exemple 2
Nous introduisons un moniteur `rendez_vous` comprenant une seule procédure, `arriver`, qui est exécutée par chaque processus lorsqu'il arrive au point de rendez-vous.

```
rendez_vous : moniteur ;
var n : entier ; tousla : condition ;
procedure arriver ;
debut
n := n+1 ; si n<N alors tousla.attendre ;
tousla.signaler
fin ;
n := 0 ;              /* initialisation */
fin rendez_vous ;
```

Lorsqu'un processus exécute `rendez_vous.arriver` et qu'il n'est pas le dernier, il incrémente `n` et se bloque en attente de `tousla`. Lorsque le dernier processus arrive au point de rendez-vous, il exécute `tousla.signaler` qui a pour effet de réveiller un des processus bloqués. Ce dernier poursuit son exécution dans le moniteur, exécute à son tour `tousla.signaler` qui réveille un second processus bloqué, et ainsi de suite jusqu'à ce que tous les processus bloqués soit réveillés. Le dernier processus réveillé exécutera un `tousla.signaler` alors qu'aucun processus n'est en attente, ce qui n'aura aucun effet.

Ce style de programmation, dans lequel chaque processus bloqué réveille le processus bloqué suivant, est classique dans la programmation à l'aide de moniteurs. On l'appelle le réveil en cascade.

4.3 Le modèle du producteur et du consommateur

4.3.1 Le problème

De nombreuses applications sont construites sur le modèle client serveur. Dans ces applications, un ou plusieurs processus, les clients, envoient des requêtes à un ou plusieurs processus serveurs. Les serveurs doivent récupérer les requêtes stockées dans un tampon intermédiaire, les traiter et envoyer une réponse au client. Nous présentons ici le modèle d'interaction sous-jacent à toutes les relations client serveur, le modèle du producteur et du consommateur.

Dans ce modèle, un processus, le producteur, envoie des messages à l'intention d'un autre processus, le consommateur. Les cadences de production et de consommation peuvent varier au cours du temps. Pour augmenter la probabilité d'exécution parallèle du producteur et du consommateur, ceux-ci communiquent par l'intermédiaire d'un tampon de `N` cases, chaque case pouvant contenir un message. Dans ces conditions, le producteur peut déposer un message s'il existe au moins une case libre dans le tampon ; si ce n'est pas le cas, il se bloque pour être réveillé par le consommateur lorsque ce dernier a vidé une case. De façon symétrique, le consommateur peut retirer un message si au moins une case est pleine ; si ce n'est pas le cas, il se bloque et sera réveillé par le producteur après un dépôt de message.

4.3.2 Spécification du problème de synchronisation

Les points de synchronisation sont, pour le producteur, le début du dépôt de message et, pour le consommateur, le début du retrait de message. On peut décrire l'état du système en introduisant

une variable entière n comptant le nombre de cases occupées du tampon.

- condition pour déposer : n<N
- condition pour retirer : n>0

4.3.3 Réalisation avec un moniteur

Les procédures deposer et retirer vont appartenir à un même moniteur prod_cons. Dans ce moniteur seront définies la variable n, initialement à 0 et deux conditions, cprod et ccons destinées à bloquer le producteur et le consommateur. Il vient :

```
prod_cons : moniteur ;
var n : entier ; cprod, ccons : condition ;
procedure deposer (m) ;
   debut
   si n=N alors cprod.attendre ;
   dépôt du message ;
   n := n+1 ;
   ccons.signaler
   fin ;
procedure retirer (m) ;
   debut
   si n=0 alors ccons.attendre ;
   retrait du message ;
   n := n−1 ;
   cprod.signaler
   fin ;
n := 0 ;            /* initialisation */
fin prod_cons ;
```

Les programme des processus producteur et consommateur peuvent être les suivants :

```
Producteur :                      Consommateur :

tantque vrai faire                tantque vrai faire
   debut                             debut
   phase de calcul                   prod_cons.retirer (mess) ;
   prod_cons.deposer (msg)           phase de traitement
   fin ;                             fin ;
```

Remarque. Dans le moniteur prod_cons, on effectue systématiquement un signaler à l'intention de l'autre processus : comme signaler ne fait rien si aucun processus n'est en attente, il est superflu de tester l'état de l'autre processus (ce qui demanderait des variables d'état supplémentaires).

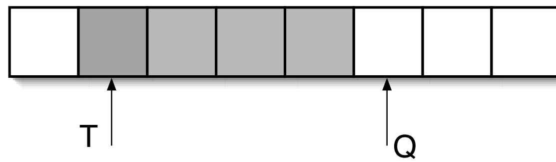

FIG. 2 – Tampon circulaire

4.3.4 Gestion du tampon

Nous n'avons pas détaillé ci-dessus la gestion du tampon. Si le tampon est un tableau `BUF` de `N` cases numérotées de `0` à `N-1`, on peut le gérer de façon circulaire comme indiqué sur la figure 2.

Soit `T` et `Q` deux variables entières initialisées toutes deux à `0` (ou à n'importe quelle valeur comprise entre `0` et `N-1`). Le dépôt du message `m` s'écrit simplement

```
BUF[Q] := m; Q := Q+1 mod N;
```

Et le retrait d'un message dans une variable `msg` s'écrit de façon symétrique

```
msg := BUF[T]; T := T+1 mod N;
```

Remarque. On appelle souvent boîte aux lettres un tampon utilisé par des processus selon le schéma producteur consommateur.