

Gestion des Processus

Ryan Cassel

cassel@limsi.fr

Université Paris XI

Plan

- Concept de processus
- Ordonnancement
- Synchronisation

Exemple

Exemple de Tanenbaum

- Pour préparer un gâteau:
 - Recette <-> Programme
 - Ingrédients <-> Données
 - Ustensiles <-> Ressources
 - Pâtissier <-> Processeur
- Processus:activité qui consiste à préparer le gâteau:
 - Lire la recette
 - Mélanger les ingrédients
 - Mettre au four

Exemple

- Scénario 1: Le pâtissier se fait piquer par une guêpe
 - Suspendre la préparation du gâteau
 - Basculer vers le processus « Mettre une pommade pour calmer la douleur »
 - Reprendre la préparation du gâteau
- Scénario 2: Il faut préparer un 2^e gâteau
 - Pas besoin de prendre une autre recette
 - Seulement ajouter des ingrédients

Processus:définition

- Introduit dans les années 60 avec MULTICS
- Processus: un programme en cours d'exécution + son contexte d'exécution
- Un processus possède:
 - Un espace d'adressage qui contient :
 - Le programme
 - Ses données
 - Sa pile (stack)
 - PCB
 - Des registres :
 - Compteur ordinal
 - Pointeur de pile

Bloc de Contrôle de Processus

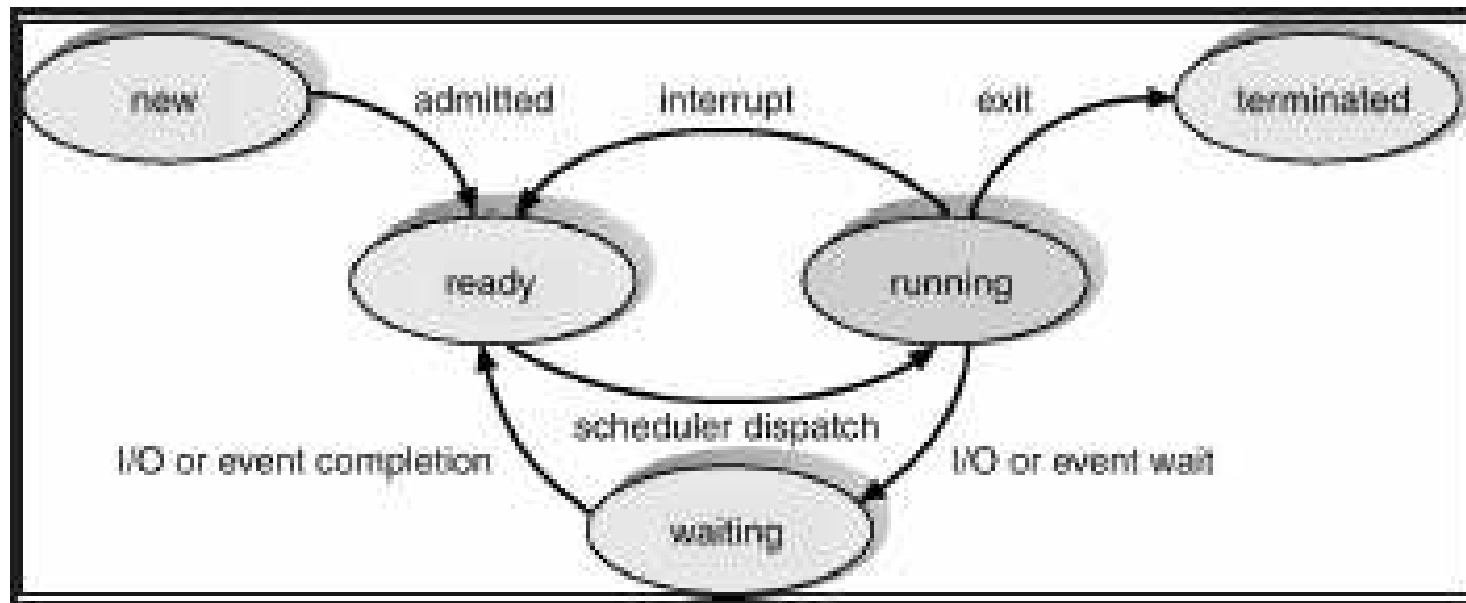
Représentation interne d'un processus : PCB

identificateur processus
état du processus
compteur instructions
contexte pour reprise (registres et pointeurs, piles,..)
pointeurs sur file d'attente et priorité(ordonnancement)
informations mémoire (limites et tables pages/segments
informations de comptabilisation et sur les E/S, périphériques alloués, fichiers ouverts,..

Etats d'un processus

- Un processus peut changer d'état:
 - Nouveau: le processus est en cours de création
 - En cours d'exécution: les instructions sont en train d'être exécutées.
 - En attente: le processus est en attente d'un événement
 - Prêt: le processus attend pour être exécuté
 - Terminé: le processus a fini son exécution

Etats d'un processus



Etats d'un processus

- Le SE détermine et modifie l'état d'un processus en fonction:

Od'événements internes au processus:

- Demande d'E/S (passage de « en exécution » à « en attente »)

Od'événements externes, provenant du SE:

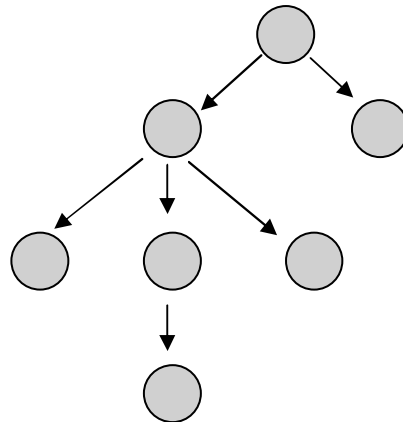
- Attribution de ressource (passage de « en attente » à « prêt »)

Opérations sur les processus

- Ensemble minimum d'opérations réalisées par le SE:
 - Création et destruction
 - Mise en attente et réveil
 - Suspension et reprise
 - Modification de la priorité

Création

- Un processus peut en créer un autre:
 - Processus père et fils
- Structure partiellement ordonnée:



Création

- Dans UNIX, primitive système `fork()`;
 - Duplique le processus en cours pour créer un processus identique au premier
 - Retourne 0 si on se trouve dans le fils
 - le PID du fils si nous nous trouvons dans le père
 - -1 en cas d'erreur
 - nécessité de modifier son code pour exécuter un autre programme avec les primitives `exec`:
 - `int execl(const char * app, const char * argv[]);`
 - `app` : chemin complet de l'application, le dernier élément est NULL
 - `argv[]` : paramètre à donner à l'application
 - `int execl(const char * app, const char * arg, ...);`
 - `app` : chemin complet de l'application
 - `arg` : paramètre sous forme d'une liste d'argument, termine par pointeur NULL.

Destruction

- Un processus fils peut terminer de différentes façons:
 - Normalement, après exécution de sa dernière instruction
 - Auto-destruction (exit dans UNIX)
 - Détruit par un autre processus (kill dans UNIX)
- Processus détruit: n'existe plus pour le SE
 - Ressources libérées
 - Bloc de contrôle effacé

Mise en attente et réveil

- Si un processus demande des ressources (autres que le CPU) non disponibles, il est mis dans l'état « en attente »
- Si elles deviennent disponibles: réveil et passage à l'état « prêt »: le processus dispose des ressources dont il a besoin sauf du CPU.

Suspension et reprise

- Passage à l'état « suspendu »:

- Progression figée

- Arrêt de l'exécution

- Entraîne la sauvegarde de son contexte

- Lors de la reprise:

- Passage à l'état « prêt » ou « en attente »

Modification de la priorité

- Très simple:
 - Modification d'une information dans le bloc de contrôle du processus
- Utile pour l'algorithme d'ordonnancement

L'ordonnancement: définition et objectifs

L'ordonnancement ?

- Multiprogrammation entraîne une concurrence des processus
- Sur un monoprocesseur, quel processus utilisera le CPU ?
- La réponse : c'est l'ordonnanceur (scheduler) et son algorithme d'ordonnancement (scheduling algorithm) qui le déterminent

Avant et maintenant

- Avant : un job à la fois. Un processus se termine et on en démarre un autre.
 - les processeurs : pas assez rapide pour traiter les informations de l'utilisateur
- Maintenant : processeurs beaucoup plus rapides que l'utilisateur.
 - Peut-on faire utiliser le processeur par un autre processus pendant que l'utilisateur réfléchit ?
 - Plusieurs utilisateurs, plusieurs processus et il faut essayer de satisfaire tout le monde, le plus vite possible.

Comment faire ?

- Exemple : un utilisateur rédige un mail, l'envoie et ferme l'application
 - Si la fermeture prend 2 secondes, l'ordinateur est considéré comme lent.
 - Si la fermeture se fait instantanément et le mail à envoyer retardé de 2 secondes, l'utilisateur est satisfait.
- L'ordonnanceur doit donc bien choisir

Quand ordonnancer ?

- création d'un processus fils
- processus terminé
- processus en sommeil
- quand une interruption d'E/S se produit

**Dans tous les cas :
A qui donne-t-on
le droit de s'exécuter ?**

Critères d'ordonnancement

- Efficacité : % d'utilisation du CPU
- Rendement : Nb de processus exécutés en un temps donné
- Temps d'exécution : le plus rapide possible
- Interactif : faible temps de réponse
- Équité : chaque processus a droit au même temps processeur

Types d'ordonnanceur

- Non préemptif

- sélectionne un processus qui s'exécute jusqu'à blocage ou libération volontaire du processeur

- Préemptif

- sélectionne un processus qui s'exécute pendant un délai déterminé

- si le processus toujours en cours après ce délai alors suspendu et un autre est choisi

Objectifs des ordonnanceurs

- Tous les systèmes

- Équité : attribuer un temps équitable à chaque processus et respecter la politique du système (sécurité, stockage, ...)
- Équilibré : toutes les parties du système doivent être occupées

Objectifs des ordonnanceurs

- **Systemes interactifs**
 - Temps de réponse : répondre rapidement aux requêtes
 - Répondre aux attentes des utilisateurs
- **Systemes temps réel**
 - respecter les délais
 - prévisibilité : éviter la dégradation de la qualité dans les systèmes multimédias

L'ordonnancement: les algorithmes

First In First Out (non préemptif)

- Processus Temps d'exécution
 - OP1 24 ms
 - OP2 3 ms
 - OP3 3 ms
- Temps d'attente moyenne (AWT):
 - OP1, P2 et P3 : $(24+27+30)/3 = 27$ ms
 - OP2, P3 et P1 : $(3+6+30)/3 = 13$ ms
- FIFO est correct mais très forte dépendance avec le temps d'exécution

Shortest job first (non préemptif)

- | ● Processus | Temps d'exécution |
|-------------|-------------------|
| OP1 | 8 ms |
| OP2 | 4 ms |
| OP3 | 4 ms |
| OP4 | 4 ms |
- $AWT = (4+8+12+20)/4 = 11$ ms
 - le S.E. doit connaître le temps d'exécution du processus
 - AWT optimal uniquement si tous les jobs sont soumis en même temps

Shortest remaining time next (préemptif)

- Le S.E choisit le processus dont le temps d'exécution restant est le plus court
- Il faut connaître le temps d'exécution à l'avance
- Favorise les jobs courts

Tourniquet (Round Robin)

- Le processeur fait tourner les processus dans une file
- A chaque processus est assigné un intervalle de temps (quantum) pendant lequel il s'exécute
- A la fin de ce quantum :
 - si le processus n'est pas terminé, il est placé en fin de file
 - Sinon un autre processus passe en exécution immédiatement sans attendre la fin du quantum, le processus terminé est supprimé
- Problème : comment fixer le quantum ?
 - si 1ms pour basculer et quantum de 3ms = 4ms alors 20% de temps de gaspillé
 - si quantum = 100ms alors 1% de perdu
 - mais si 10 utilisateurs qui appuient sur Entrée : 10 processus dans la file, le deuxième doit attendre 100ms et le dernier 1 seconde

Par priorité

- Existence d'une file de processus d'une priorité donnée
- Chaque file est gérée par l'algorithme du Round Robin
- La file de priorité maximale traitée en premier
- Quand une file est vide, on traite la file de priorité inférieure
- Problème : il faut revoir les priorités sinon famine dans les priorités basses

Files multiniveaux

- Plusieurs files qui possèdent chacune leur algorithme d'ordonnancement
- Chaque file a un but
 - Une file pour les jobs de 1er plan
 - Une file pour les jobs de 2ème plan...
- Chaque processus est assigné à une file lors de sa création
- Variante : migration d'un processus d'une file vers une autre

Synchronisation

Synchronisation

- Un processus peut être *coopératif* ou *indépendant*.
 - *indépendant* : s'il n'affecte pas les autres processus ou ne peut pas être affecté par eux.
 - Un processus qui ne partage pas de données avec d'autres processus est indépendant
 - Un processus est *coopératif* s'il peut affecter les autres processus en cours d'exécution ou être affecté par eux
 - Un processus qui partage des données avec d'autres processus est un processus coopératif
 - données partagées par les processus coopératifs : en mémoire principale ou en mémoire secondaire dans un fichier
 - Risque d'incohérence des données

Exemple

- Mise à jour d'un compte bancaire d'un client.
- Compte: fichier avec solde courant du client

Procédure `crediter_compte(entier numero_client,
entier somme)`

entier solde ;

debut

/ lecture du solde dans le fichier du client */*

`solde=lire_dans_fichier(numero_client);`
`solde + somme ;`

solde =

/ écrire dans le fichier le nouveau solde */*

`ecrire_dans_fichier(numero_client, solde) ;`

fin;

Exemple

- Si la procédure est exécutée simultanément par deux processus P0 et P1 dans un système d'exploitation à temps partagé. Sans faire d'hypothèse sur la durée du quantum, on peut faire les exécutions suivantes :

P0 : `crediter_compte(1233, 1000)`

`solde = lire_dans_fichier(1233) ;`

`/* P0 bloqué car P1 s'exécute */`

`solde=solde+1000 ;`

`Ecrire_dans_fichier(1233,solde) ;`

P1 : `crediter_compte(1233, 500)`

`solde = lire_dans_fichier(1233) ;`

`/* P1 bloqué car P0 s'exécute */`

`solde=solde+500 ;`

`Ecrire_dans_fichier(1233,solde) ;`

Comme le montre cet exemple, le résultat final n'est pas le résultat escompté (*le client a perdu 1000 euros dans cette affaire*).

Sections critiques (S.C)

Dans l'exemple: problème dû aux conflits d'accès au même fichier car accès en lecture et en écriture.

Si seulement accès en lecture → pas de problème

- On appelle **section critique** la partie d'un programme où se produit le conflit d'accès.

Sections critiques (S.C)

- Comment éviter ces conflits d'accès ?
 - Il faut trouver un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois.
 - Il faut une **exclusion mutuelle** qui empêche les autres processus d'accéder à des données partagées si celles-ci sont en train d'être utilisées par un processus.
 - Dans l'exemple précédent, il faut obliger le processus P1 à attendre la terminaison de P0 avant d'exécuter à son tour la même procédure.

Pour une bonne coopération entre processus

Pour que les processus qui partagent des objets (données en mémoire centrale ou dans fichiers) puissent coopérer correctement et efficacement, quatre conditions sont nécessaires :

1. **Exclusion mutuelle** : deux processus ne peuvent être en même temps en section critique
2. **Famine** : aucun processus ne doit attendre trop longtemps avant d'entrer en section critique
3. **Interblocage** (*deadlock*) : aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres pour y entrer .
4. **Aucune hypothèse** ne doit être faite sur les vitesses relatives des processus

Exclusion mutuelle

- Un processus désirant entrer dans une section critique doit être mis en attente si la section critique n'est pas libre
- Un processus quittant la section critique doit le signaler aux autres processus
- Protocole d'accès à une section critique :
 - <entrer_Section_Critique> /* attente si SC non libre */
 - <Section_Critique> /* Un seul processus en SC */
 - <Quitter_Section_Critique>

Exclusion mutuelle

- L'attente peut être :
 - **Active** : procédure *entrer_Section_Critique*=boucle dont la condition est un test qui porte sur des variables indiquant la présence ou non d'un processus en Section critique
 - **Non active** : le processus passe dans l'état endormi et ne sera réveillé que lorsqu'il sera autorisé à entrer en section critique

Techniques

- Désactivation des interruptions
- Variables de verrou
- Alternance stricte
- Algorithme de Peterson

1ère Solution : Masquage des interruptions

- Moyen le plus simple : chaque processus masque les interruptions avant d'entrer en section critique
 - l'interruption horloge qui permet d'interrompre un processus lorsqu'il a épuisé son quantum (temps CPU) sera ignorée
 - plus de commutation de processus
- Lorsqu'un processus quitte la section critique, doit restaurer les interruptions
- Solution dangereuse en mode utilisateur :
 - Si dans un processus, le programmeur a oublié de restaurer les interruptions, c'est la fin du système

2ème solution : les variables de verrouillage

- Un verrou est une variable binaire partagée qui indique la présence d'un processus en Section Critique :
 - si verrou = 0 alors la section critique est libre
 - si verrou = 1 alors la section critique est occupée
- Procédure *entrer_Section_Critique ()* :

```
void entrer_Section_Critique () {  
    if (verrou == 0) verrou=1 ;  
    else while (verrou == 1) ; /* attente active */  
    verrou=1 ;  
}
```
- Procédure *quitter_Section_Critique ()*

```
void quitter_Section_Critique () {  
    verrou=0 ;  
}
```
- L'exclusion mutuelle n'est assurée que si le test et le positionnement du verrou est ininterrompible (sinon le verrou constitue une section critique)

3ème Solution : l'alternance

- On utilise une variable partagée (*tour*) qui mémorise le numéro du processus autorisé à entrer en section critique
- Exemple d'utilisation pour N processus :

```
void entrer_Section_critique (int MonNumero) {  
    while (tour != monNumero) ; /* attente active */  
}
```

```
void quitter_Section_critique () {  
    tour=(monNumero +1) % N ; /* au suivant ! */  
}
```

- Avantage : simple et facile à utiliser
- Inconvénient : problème de **famine**, un processus possédant *tour*, peut ne pas être intéressé immédiatement par la section critique

Solution de *Peterson* (1981)

- Pour le cas de deux processus P0 et P1 :

```
#define FAUX 0
```

```
#define VRAI 1
```

```
#define N 2
```

```
int tour ; /* à qui le tour */
```

```
int interesse[N] ; /* initialisé à FAUX */
```

```
void entrer_Section_Critique (int process) /* n° de processus : 0 ou 1*/
```

```
{
```

```
    int autre ;
```

```
    autre = 1-process ;
```

```
    interesse[process]=VRAI ; /* indiquer qu'on est intéressé */
```

```
    tour = process ; /* lever le drapeau */
```

```
    while (tour == process && interesse[autre] == VRAI) ; /* attente active */
```

```
}
```

```
void quitter_Section_Critique(int process)
```

```
{
```

```
    interesse[process]=FAUX ;
```

```
}
```

Solution de *Peterson* (1981)

- Pourquoi l'exclusion mutuelle est assurée par cette solution?
 - Réponse : Considérons la situation où les deux processus appellent *entrer_Section_Critique* simultanément. Les deux processus sauvegarderont leur numéro dans la variable *tour*. La valeur sauvegardée en dernier efface la première. Le processus qui entrera en SC est celui qui a positionné la valeur *tour* en premier.
- Gros inconvénient : elle est basée sur l'attente active ; un processus ne pouvant entrer en SC utiliserait l'UC inutilement.

Solution évitant l'attente active

- **Idée** : un processus ne pouvant entrer en section critique, passe dans un état endormi, et sera réveillé lorsqu'il pourra y entrer.

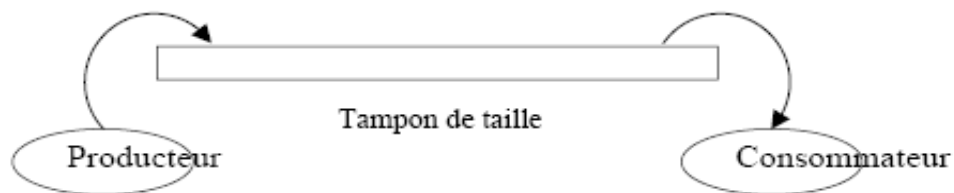
○ Nécessite un mécanisme de réveil

○ Le SE fournit deux appels système :

- Sleep (dormir) qui suspend le processus appelant
- Wakeup (réveiller) qui réveille le processus donné en argument

Solution évitant l'attente active

- Application au modèle Producteur/Consommateur



- Les deux processus coopèrent en partageant un même tampon (buffer)
 - Le producteur produit des objets qu'il dépose dans le tampon
 - Le consommateur retire des objets du tampon pour les consommer
- **Conflits**
 - Le producteur veut déposer un objet alors que le tampon est déjà plein
 - Le consommateur veut retirer un objet du tampon alors que celui-ci est vide
 - Le producteur et le consommateur ne doivent pas accéder simultanément au tampon

Suivre le nombre d'informations ?

- Suppose que le nombre d'informations soit stocké dans un variable Count
- Soit N la taille du buffer
 - Le Producteur :
 - Si $\text{Count} == N$ alors Producteur en sommeil
 - Il ajoute l'information; $\text{Count}++$;
 - Si $\text{Count} == 1$ alors Réveille le consommateur
- Le Consommateur :
 - Si $\text{Count} == 0$ alors Consommateur en somme
 - Il retire l'information; $\text{Count}--$;
 - Si $\text{Count} == N-1$ alors Réveille le producteur

Analyse de cette solution

- L'accès à la variable *compteur* n'est pas protégé, ce qui peut entraîner des incohérences dans les valeurs prises par cette variable
- Réveils perdus : c'est le principal défaut de ce mécanisme. Un signal wakeup envoyé à un processus qui ne dort pas (encore) est perdu.

Réveils perdus

- Consommateur lit Count à 0 et le S.E. bascule sur le producteur
- Producteur insère une information. Lit Count à 1 donc réveille le consommateur. Le signal est perdu car le consommateur ne dort pas
- S.E. bascule sur le consommateur
- Le consommateur a lu 0 donc s'endort
- Le producteur finit par remplir le buffer et s'endort

Solution

- Utilisation de sémaphore