

## Systèmes d'exploitation, 2ème année

### Deadlock et famine

Yves STADLER

Université Paul Verlaine - Metz

3 novembre 2011

### Plan du chapitre

- Deadlocks
- Famine
- Stratégie de prévention

1/17

2/17

## Deadlock

### Illustration d'un deadlock

- On dispose de 200KB de mémoire
- Le processus A requiert 80KB de mémoire
- Le processus B requiert 70KB de mémoire
- Le processus A requiert 60KB de mémoire
- Le processus B requiert 80KB de mémoire

### Pourquoi arrive-t-on là ?

- Indépendamment chaque processus peut s'exécuter
- L'ordonnancement peut mener à un point où chaque processus réserve une partie de la mémoire sans pour autant pouvoir entrer en section critique.

3/17

## Exemple avec deux sémaphores

### Programme 1

```
P(semA)
P(semB)
Critical Section
V(semB)
V(semA)
```

### Programme 2

```
P(semA)
P(semB)
Critical Section
V(semB)
V(semA)
```

### Philosophes

```
think()
P(fork[i])
P(fork[i+1])
eat()
V(fork[i])
V(fork[i+1])
```

4/17

### Coffman conditions

- Exclusion mutuelle : un seul processus à la fois peut utiliser une ressource
- Rétention : un processus conserve une ressource tout en demandant une autre déjà allouée
- Pas de préemption (de ressource)
- Attente circulaire : il faut une chaîne de processus dans laquelle chaque processus détient la ressource nécessaire au processus suivant dans la chaîne.

5/17

### Plan du chapitre

- S'assurer que le système n'y entre jamais
- Permettre au système de se bloquer et l'en sortir
- Ne pas s'en occuper et espérer que ça n'arrive jamais.

6/17

## Prévention

### Objectif

- Éliminer l'une des conditions de Coffman

### Objectif

- Éviter de mettre des exclusions lorsqu'elle ne sont pas nécessaires (fichiers en lecture seule).
- Obliger les processus à réclamer toutes leurs ressources en une fois (risque de famine)
- Refuser la rétention (si une ressource est refusée, on relâche tout)
- Définir un ordre dans l'attribution de ressources (moins efficace)
- Ne démarrer un processus que si on peut réserver d'avance ses ressources
- Ne donner les ressources que si le système reste dans un état sain.
- Détection facile, prévention difficile.

7/17

## Algorithme du Banquier

### Conditions

- Un état est sûr lorsqu'il existe une séquence sûre pour tous les processus
- Une séquence de processus  $P_i$  est sûre si  $\forall P_i$  les ressources que ce processus n'a pas réclamé peuvent être satisfaites ou sont alloué aux processus le précédent dans la séquence.

8/17

## Algorithme du Banquier

### Réalisation

- Quand un processus arrive, il indique la quantité maximale de chaque ressource qu'il compte utilisée.
- Quand un processus se voit allouer des ressources, il doit les restituer dans un délai de temps fini.

### Implémentation

- Quantité de ressources potentielles à allouer (MAX)
- Quantité de ressources déjà alloués (CUR)
- Quantité de ressources dont le système dispose (AVL)
- Optionnellement la quantité de ressources potentiellement requise (NED).

9/17

## Algorithme du Banquier

### Allocations

- Requête  $\leq$  max ; Sinon le programme à mentit, exception ;
- Requête  $\leq$  disponible ; Sinon attente.
- $NED = MAX - CUR$ .

### Vérifications

- Est-il possible d'allouer NED pour au moins un processus
- Permet de faire terminer ce processus
- Libère les ressources
- Si impossible, état "unsafe".

10/17

## Algorithme du Banquier

### Exemple

MAX	A	B	C	AVL	A	B	C	CUR	A	B	C	NED	A	B	C
P1	5	7	2	ALL	6	9	2	P1	0	0	0	P1	5	7	2
P2	3	5	1					P2	0	0	0	P2	3	5	1

### Exemple

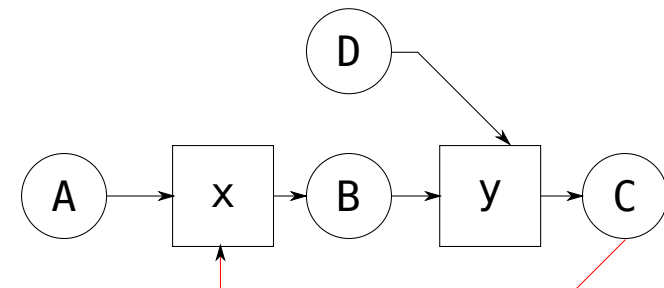
MAX	A	B	C	AVL	A	B	C	CUR	A	B	C	NED	A	B	C
P1	5	7	2	ALL	1	9	2	P1	5	0	0	P1	0	7	2
P2	3	5	1					P2	0	0	0	P2	3	5	1

11/17

## Représentation graphique

### Définitions

- Deux types de nœuds : processus ou ressources
- Une arrête d'un processus à une ressource est une demande
- Une arrête d'une ressource à un processus est une allocation
- L'objectif est de découvrir les cycles



12/17

## Plan du chapitre

- Soit  $D_T$  l'ensemble des processus sur lesquels T attend.
- $D_T = T$  si T ne veut rien.
- $D_T = D_{owner(R)}$  (Union de lui avec un autre  $D_i$ ).

## Plan du chapitre

- $D_C = C$
- $D_D = C, D$
- $D_B = C, B$
- $D_A = B, C, A$
- C voulant x détecte un cycle dans  $D_B$

13/17

## Structure pour opération sur sémaphores

## Fonction de contrôle

```
int semctl(int semid, int semnum, int cmd, ...);
union semun {
    int          val; /* Value for SETVAL */
    struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */
    unsigned short *array; /* Array for GETALL, SETALL */
    struct seminfo *__buf; /* Buffer for IPC_INFO
                           (Linux specific) */
};
```

- Opération cmd: IPC\_STAT | IPC\_SET | IPC\_RMID
- Le quatrième paramètre quand il existe est la structure semun

15/17

## Headers

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

## Fonctions de création

```
int semget(key_t key, int nsems, int semflg);
```

- Appel et renvoi nsems sémaphores key
- semflg : IPC\_PRIVATE | IPC\_CREAT
- Le flag permet aussi de choisir les permissions du sémaphore.

14/17

## Structure pour opération sur sémaphores

## Fonction d'usage

```
int semop(int semid, struct sembuf *sops, unsigned nsops);

unsigned short sem_num; /* semaphore number */
short          sem_op;  /* semaphore operation */
short          sem_flg; /* operation flags */
```

- Opérations sur sem\_num
- de type sem\_op (entier, incr/décromente le sémaphore de cette valeur
- avec un flag sem\_flg : IPC\_NOWAIT | SEM\_UNDO

16/17

# Les primitives exec

## Plan du chapitre

- On connaît le nombre d'arguments : famille `exec1`
- On ne connaît pas le nombre d'arguments : famille `execv`

## Plan du chapitre

- Remplace l'image du programme par un autre
- Toutes les instructions qui suivent `exec` ne seront jamais exécutées.