Système d'exploitation et programmation système

Cours 7

EISTI

2008/2009

Département informatique

Communication Inter-processus

Sémaphores

Partage de ressources et interblocage

Principe

- Le fonctionnement en multi-taches implique que plusieurs processus peuvent nécessiter la même ressource au 'même moment'
- On doit pouvoir verrouiller l'accès à une ressource
- Si les accès sont verrouillés, on peut rencontrer des situations d'interblocages

Exemple :agenda électronique partagé

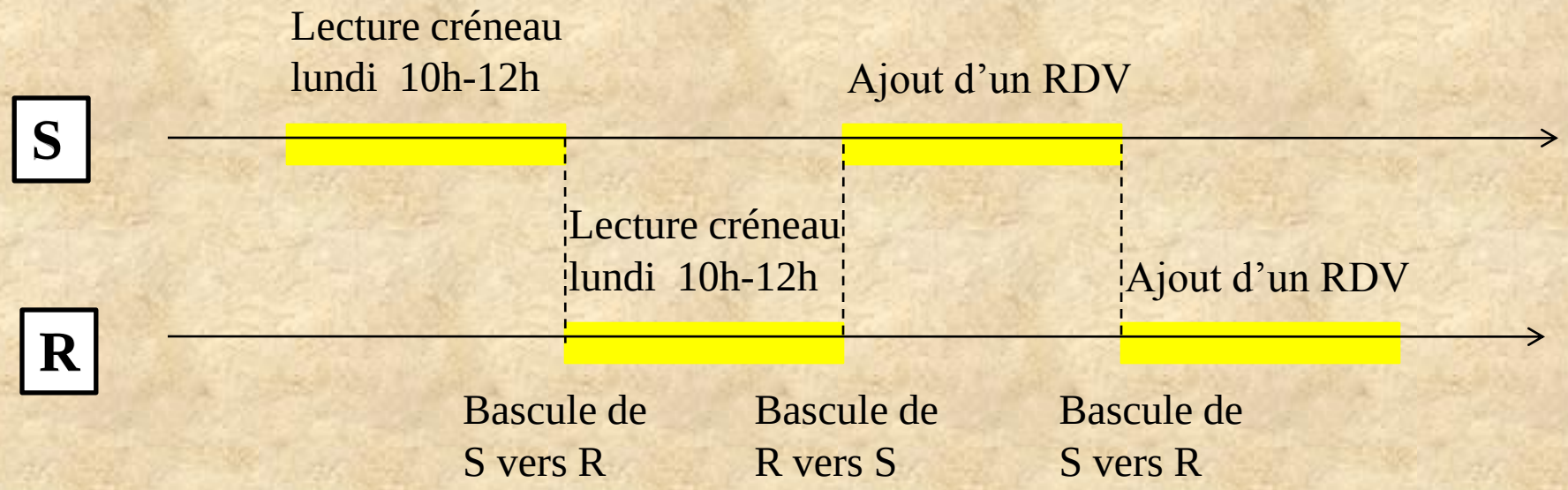
- Soit un agenda pour fixer les RDVs d'un responsable. Cette agenda est utilisé par le responsable lui même et par son ou sa secrétaire.
- LA procédure de prise d'un RDV est la suivante:
  - 1. Chercher un prochain créneau de libre
  - 2 . Fixer un RDV

	8h-10h	**10h-12h**		**13h-15h**	**15h-17h**
Lundi					
Mardi					
Mercredi					
Jeudi					
Vendredi					

  - Créneaux libres :

Scénario sans les sections critiques

- => Que se passe-t-il si deux utilisateurs **R**esponsable et **S**ecrétaire désirent ajouter simultanément un nouveau RDV?



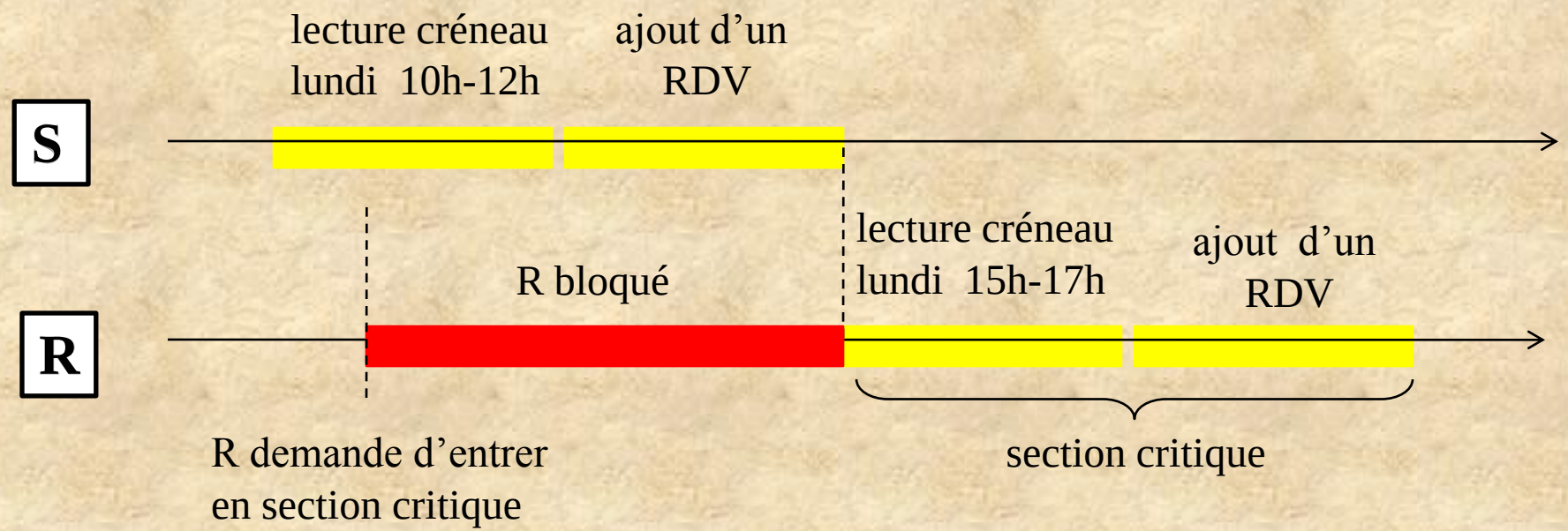
Définition : section critique

Une section critique est une portion de code qui doit être exécutée en exclusion mutuelle.

- Quatre conditions :
  - Deux processus ne peuvent pas être en même temps en section critique, **exclusion mutuelle**.
  - Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre des processeurs.
  - Aucun processus suspendu en dehors d'une section critique ne doit bloquer les autres processus.
  - Aucun processus ne doit attendre trop longtemps avant de rentrer en section critique, **famine**.

Scénario avec les sections critiques

- => Que se passe-t-il si deux utilisateurs **R**esponsable et **S**ecrétaire désirent ajouter simultanément un nouveau RDV?



Bascule de
S vers R

Approche simpliste : attente active

- Stocker dans un container partagé la disponibilité de la ressource et procéder par attente active. Problèmes :
  - C'est une ressource partagée, donc le problème reste le même (si deux accès sont simultanés, on ne peut rien garantir)
  - L'attente active est gourmande en temps CPU

Variable de verrou

- Solution logicielle au problème de l'exclusion mutuelle ;
- Une variable unique (verrou ou lock) est partagée entre tous les processus ;
- Lorsqu'un processus veut entrer en section critique :
  - Si le verrou est à 1, le processus attend qu'il passe à 0 ;
  - Si le verrou est à 0, le processus le place à 1 et entre en section critique ;
  - Lorsque la section critique est finie, le verrou est replacé à 0.

- Problème
  - Le test et la modification de la variable n'est pas atomique : deux processus peuvent être simultanément en section critique.

Approche simpliste : alternance

- Stocker dans un container partagé la disponibilité de la ressource et procéder par alternance.

Processus R

```
while (1) {
    while (lock != 0) ;
        // Section critiq ue
         lock = 1 ;
        // Section non critique
  }
```

Processus S

```
while (1) {
    while (lock != 1) ;
        // Section critiq ue
         lock = 0 ;
        // Section non critique
  }
```

  - La variable est partagée …
  - Cette solution impose une alternance
  - Les deux processus n'entrent pas en section critique simultanément ;
  - Ils y rentrent à tour de rôle.

- Problèmes : Un processus ne peut pas entrer deux fois d'affilé en section critique.

Solution de Peterson

- Chaque processus est identifié par un numéro (ici 0 ou 1).
- Données :

  Variable partagée tour ;

  Tableau partagé **interesse** : indique si le processus désire entrer en section critique ;

```
int tour;                  // a qui le tour?
int interesse[2]={false, false};  // false pour les 2 processus (generalisation n process)
P(numProcessus =1 ou numProcessus =0) :
void entrer_region ( int numProcessus ) {
   int autre = 1 − numProcessus ;
   interesse[numProcessus] = true;          //indique qu'on est interesse
   tour = numProcessus ;                    //positionne le drapeau
   while (tour ==  numProcessus  &&  interesse[autre] = = true) ; // attente active
   < section critique >
}

void sortie (int numProcessus ) {
    interesse [numProcessus] = false;
}
```

Preuve de sureté. (safety)

Si **R** et **S** sont tous les deux dans leur section critique.

Alors interesse[0] = interesse [1] = true.

Impossible car les deux tests auraient été franchis en même temps alors que tour favorise l'un deux. Donc un seul est entré. Disons **R**. Cela veut dire que **S** n'a pu trouver le tour à 1 et n'est pas entré en section critique.

Preuve de vivacité. (lifeness)

Supposons **R** bloqué dans le while.

Cas 1 : **S** non intéressé à rentrer dans la section critique. Alors interesse[1] = false. Et donc **R** ne peut être bloqué par le while.

Cas 2 : **S** est aussi bloqué dans le while. Impossible car selon la valeur de tour, l'un de **R** ou **S** ne peut rester dans le while.

Les sémaphores

- Les solutions précédentes font appel à l'attente active
  - Tant que les processus ne peuvent pas entrer en section critique, ils entrent dans une boucle qui consomme du CPU.
- Les solutions présentées ici ne sont valables que dans le cas d'une ressource (ou difficilement adaptables à n ressources)
- En 1965, Dijkstra propose de nouvelles variables nommées sémaphores ;
- Deux opérations atomiques possibles pour manipuler ces variables :
  - P(s) : test de la valeur de la sémaphore
    - Si la valeur est à 0, le processus est bloqué ;
    - Sinon, la valeur est décrémentée.
  - V(s) : incrémentation de la valeur de la sémaphore, réveiller un processus en attente (libération de la ressource).

Pré-requis

- Cette solution repose sur :
  - l'atomicité des opérations **P** et **V**, c'est-à-dire sur le fait que la suite d'opérations les réalisant (section critique) est non interruptible (afin d'éviter les conflits entre processus),
  - le mécanisme de **file d'attente** permettant de mémoriser les demandes d'opération **P** de processus non satisfaites, et de réveiller les processus en attente.
- L'atomicité des opérations **V** et **P** revient à bloquer l'accès à la variable **S** (en test et en modification) dès lors qu'un processus modifie la valeur de **S** soit par **P** ou par **V**.

Valeurs des sémaphores

- Pour un sémaphore initialisé à 1 :
  - Une seule ressource est disponible
  - L'acquisition de la ressource endort tout autre processus essayant de réaliser la même action
- Pour un sémaphore initialisé à n :
  - n ressources sont disponibles
  - Chaque processus peut acquérir une ressource tant qu'il y en a de disponibles

Exemple de synchronisation par rendez-vous de n processus

- Soit un sémaphore S initialisé à n.

Procesus 1	Processus 2	Processus j	Processus n
A1	A2	Aj	An
P(S) Z(S)	P(S) Z(S)	P(S) Z(S)	P(S) Z(S)
B1	B2	Bj	Bn

← Point de RDV

Le bloc d'instructions B peut être exécuté une fois que les n processus auront terminé d'exécuter le bloc d'instructions A.

Utilisation d’un sémaphore

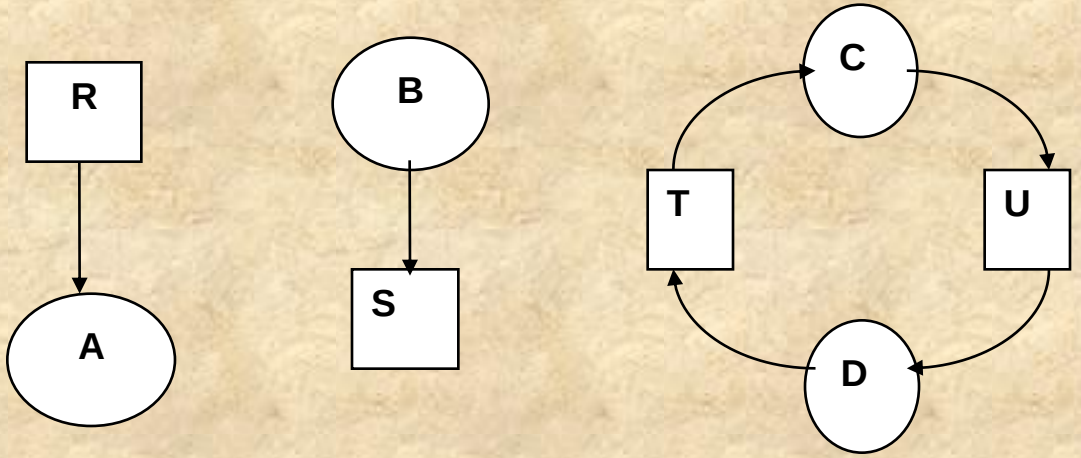
- Exclusion mutuelle
- Partage de n ressources pour p processus
- Mécanismes de rendez-vous (attente de nullité du sémaphore) : Z(s)

Interblocage

- Un ensemble de processus est en interblocage si chaque processus attend un événement que seul un autre processus de l'ensemble peut engendrer.
- Un interblocage doit remplir les 4 conditions suivantes :
  - 1 Condition d'exclusion mutuelle : chaque ressource est attribuée à un seul processus ou bien elle est disponible ;
  - 2 Condition de détention et d'attente : un processus ayant déjà obtenu une ressource peut en demander une nouvelle ;
  - 3 Pas de réquisition : si un processus possède une ressource, elle ne peut lui être retirée de force ;
  - 4 Condition d'attente circulaire : il doit y avoir un cycle d'au moins 2 processus, chacun attendant une ressource détenue par un autre processus du cycle.

Interblocage : modélisation

- Modélisation de Holt:



A détient R **B demande S** **Circuit** **Interblocage**

Exemple d'interblocage

- Tous les problèmes de synchronisation ne sont pas solubles avec les sémaphores simples de Djiskstra, tels qu'ils ont été décrits précédemment.

Par exemple : la demande de **m** ressources différentes, dans un ensemble de **n** ressources (**m <= n**).

Soit les **n** sémaphores **S1, S2, ..., Sn** permettant de bloquer les **n** ressources. Si **P1** demande **R1, R2, R3** et **R4,** et **P2** demande **R2, R3, R4** et **R5** alors on a :

P1 → **P(S1)**, **P(S2)**, **P(S3)**, P(S4)

P2 → **P(S3)**, **P(S4)**, **P(S5)**, **P(S2)**

On a un risque d'**interblocage**.

Si les 4 opérations **P** étaient **atomiques**, on éviterait les interblocages :

P (S1, S2, S3, S4)

P (S2, S3, S4, S5)

Interblocage : que faire ?

- Les ignorer : il faut pouvoir quantifier la fréquence et les conséquences
- Reprendre les ressources créant l'interblocage
- Casser le cycle
- Annuler l'état bloquant (rollback) : Les solutions adoptées sur "SYSTEM V" (et LINUX) consistent à réaliser des ensembles de sémaphores avec la propriété qu'une opération de l'ensemble ne pourra être réalisée que si toutes les autres peuvent l'être également.

Primitives des sémaphores sous Linux/Unix

- Les primitives IPC pour les sémaphores sont une généralisation des sémaphores de Dijkstra pour la gestion des accès concurrents à une ressource. Ces primitives permettent de réaliser de manière atomique un ensemble d'opérations sur un ensemble de sémaphores. Cette extension permet d'éviter les verrous mortels : tous les sémaphores désignés sont modifiés ou aucun.
- Les sémaphores sont implantés par System V sous forme de tableaux de sémaphores repérés par son identificateur qui est retourné par la primitive **semget**.
- L'initialisation des compteurs associés à chacun des sémaphores de l'ensemble est faite par **semctl** qui permet aussi d'obtenir des informations sur l'état des sémaphores (processus en attente, valeur courante, ...).

Structures : __sem

Structure d'un sémaphore en système V, est définie dans <sys/sem.h>

Une structure par sémaphore

```
struct  __sem {
  unsigned short int semval;  // valeur du semaphore
  unsigned short int sempid;  // pid du dernier processus l'ayant manipulé
  unsigned short int semcnt;  //nombre de processus attendant l'augmentation du sem
  unsigned short int semzcnt; // nombre de processus attendand la nullité de semval
};
```

Structures : semid_ds

A chaque ensemble de sémaphores est associé la structure semid_ds définie dans <sys/sem.h>

```
struct semid_ds{
   struct ipc_perm  sem_perm;   // permissions d'accès
   struct __sem     *sem_base;  // pointeur sur le premier semphore de l'ensemble
   time_t           sem_otime;  // date de derniere operation par semop
   time_t           sem_ctime;  // date de dernier changement par semctl
   unsigned short   sem_nsems;  // Nombre de semaphores de l'ensemble
};
```

Structure : sembuf

Une structure par sémaphore, la structure **sembuf** correspond à une opération sur un sémaphore (**incrémenter**, **décrémenter**, **attendre une valeur nulle**), définie dans <sys/sem.h>.

```
struct sembuf {
  unsigned short int sem_num ; // n° du sémaphore dans le groupe : 0,..
  short sem_op ; // opération sur le sémaphore
  short sem_flg ; // options pour l'operation
} ;
```

- Les valeur de *sem_op* sont :
  - Négative, l'opération à réaliser est une opération P (décrémenter)
  - Positive, l'opération à réaliser est une opération V (incrémenter)
  - Nulle, l'opération à réaliser est une opération Z (attendre)
- Les valeur de *sem_flg* sont IPC_NOWAIT et SEM_UNDO.
  - Si une opération indique SEM_UNDO, elle sera annulée lorsque le processus se terminera.

Opérations P, V ou Z sur les sémaphores

int **semop** (int *semid*, struct sembuf **tab_ops*, unsigned *nb_ops*);

- Cette primitive permet de réaliser atomiquement les *nb_ops* opérations spécifiées dans le tableau *tab_op*, sur l'ensemble de sémaphores identifié par *semid*.
- Ces opérations sont réalisées séquentiellement.
- Par défaut, le système réalise toutes les opérations ou aucune : si l'une d'entre elles ne peut être effectuée, les autres sont annulées. L'ordre de ces opérations dans le tableau n'est pas indifférent.

Dans le cas de succès, la primitive renvoie 0; -1 sinon.

semid : descripteur du groupe de sémaphores visé

Tab_ops : tableau de structures qui contient la liste des opérations

Nb_ops : nombre des opérations à réaliser lors de l'appel

Création et recherche d'un groupe de sémaphores

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

int **semget**(key_t *key*, int *nbsems*, int *flags*);

■ Cette fonction permet d'allouer une entrée pour un tableau de n sémaphores ou d'accéder à une entrée existante. Elle renvoie un descripteur sur le groupe de sémaphores créé, ou récupère le descripteur d'un sémaphore existant.

■ Elle retourne un identificateur, ou -1 en cas d'erreur.

key : n° de clé d'un groupe de sémaphores (existant ou non) ou **IPC_PRIVATE** (groupe non nommée)

nsems : nombre de sémaphores à créer dans le groupe

flags : donne les droits sur le sémaphore. S'il comporte la valeur **IPC_CREAT | IPC_EXCL** (création exclusive), **0666** (droits d'accès)

Contrôle d'un groupe (tableau) de sémaphores

#include<sys/types.h>

#include<sys/ipc.h>

#include<sys/sem.h>

int semctl (**int** *semid*, **int** *semnum*, **int** *cmd*, **union** semun *arg*);

Cette primitive permet d'effectuer l'opération indiquée par *cmd* (consulter, initialiser ou supprimer) sur l'ensemble de sémaphores (ou sur le *semnum-ième* sémaphore) de l'ensemble identifié par *semid*.

Elle est utilisée pour donner la valeur initiale du sémaphore après sa création.

semid : descripteur du groupe(tableau) de sémaphores visé

semnum : numéro du sémaphore choisi dans le groupe (0 pour le premier)

cmd : type d'opération à effectuer sur le sémaphore

arg : sert à passer des arguments aux commandes exécutées par "*cmd* "

Contrôle d'un groupe (tableau) de sémaphores

■ La structure semnum est définie par :

```
union semun {
        int val; // utilisé uniquement pour SETVAL
        struct semid_ds *buffer; // utilisé pour IPC_STAT et IPC_SET
        unsigned short int * array; // utilisé pour GET_ALL, SET_ALL
                                    /* Spécificité Linux mais pas Unix: */
        struct seminfo * __buf;     /* buffer pour IPC_INFO */
    }
```

■ Initialiser un sémaphore

semctl (semid, 0, SETVAL, 3) initialisation à la valeur 3 du sémaphore 0 dans l'ensemble désigné par l'identifiant semid.

■ Détruire un ensemble de sémaphores désigné par l'identifiant semid

int semctl (int semid, 0, IPC_RMID, 0);

Commandes

Les différentes valeurs que peut prendre cmd sont :

Valeur de *op*	Interprétation *semnum*	Interprétation *arg*	Valeur de retour et effet
GETNCNT	Numéro d'un sémaphore		Nbre de processus en attente d'augmentation du sémaphore.
GETZCNT	Numéro d'un sémaphore		Nbre de processus en attente de nullité du sémaphore
GETVAL	Numéro d'un sémaphore		Valeur du sémaphore
GETALL	Nombre de sémaphore	tableau d'entiers courts non signés	*arg* contient les valeurs des semnum premiers sémaphores
GETPID	Numéro d'un sémaphore		PID du dernier processus réalisant une opération sur le sémaphore
SETVAL	Numéro d'un sémaphore	entier	Initialisation du sémaphore à arg
SETALL	Nombre de sémaphore	tableau d'entiers courts non signés	Initialisation des semun-ième premiers sémaphores à *arg*
IPC_RMID			Supprimer tous le groupe de sémaphores en réveillant tous les processus en attente
IPC_STAT		pointeur sur **struct** *semid_ds*	Extraction de l'entrée de la table des sémaphores
IPC_SET		pointeur sur **struct** *semid_ds*	Modification de l'entrée de la table des sémaphores

FIN