LINGI 1113: Systèmes informatiques 2

Organisation du cours

ÉCOLE
POLYTECHNIQUE
DE LOUVAIN

UCL

Organisation du cours

7 missions de deux semaines

- Lundi semaines impaires
  - 8h30: intro à la mission
  - 9h30: présentation tp de la mission
- Jeudi semaines impaires
  - 16h15: tp en salle
- Lundi semaines paires
  - 8h30: Q&A sur théorie mission
  - 9h30: feedback tp mission précédente
- Jeudi semaines paires
  - 16h16: tp en salle

Organisation du cours

Etude de la théorie

- Lundi semaines impaires
  - 8h30: intro à la mission
- Reste de la semaine:
  - étude personnelle dans livre de Stallings (100 pages mais bien expliqué)
- Lundi semaines paires
  - 8h30: Q&A sur théorie mission

Organisation du cours

Le bouquin:

- William Stallings « Operating systems
- 6me ou 7me édition
- Indispensable: les transparents des séances introductives ne suffisent P A S
- Clair et agréable à lire (explique bien)
- Vous devez lire et comprendre, pas retenir par coeur
- Vous lirez environ 100 pages par mission

Organisation du cours

Les TPs

- En général un travail par mission
- Présenté le lundi en début de mission, après intro à la théorie
- On travaille en binomes
- Les séances en salle du jeudi: occasion de discuter avec autres binomes et assistants
- Travaux à remettre avant la fin de la mission (lundi 8h30)
- Feedback 1 semaine plus tard, après la séance Q&A du lundi
- Certains travaux sont cotés et interviennent dans la note du cours

Organisation du cours

Les missions

- Rappels sur le parallélisme: chap. 5&6, p ~200 à ~300
- Rappel sur architectures et présentation générale des systèmes d'exploitation: chap 1&2, p 1 à ~100
- Les processus, threads, multiprocesseurs et micro-noyaux: chap 3&4, p ~100 à ~200
- Les mémoires réelles et virtuelles: chap 7&8, p ~300 à 400
- L'ordonnancement chap 9&10, p ~400 à 490
- Les entrées sorties et les fichiers, chap 11&12, p ~490 à ~600
- Introduction aux cours avancés: systèmes embarqués et temps réel, sécurité et systèmes répartis, chap 13 à 15 p ~600 à ~750

LINGI 1113: Systèmes informatiques 2

Mission 1: Concurrence

(rappels ;-)

UCL

Mission 1 : Concurrence

- **Concurrence:** plusieurs morceaux de programmes actifs en même temps peuvent interférer entre eux
- **Interférer:** partager des ressources (mémoire périphériques,...)
- **Programmes actifs en même temps:**
  - **Processus** (partage: périphériques, mémoire partagée mode utilisateur ou système)
  - **Threads** (dans un programme applicatif ou dans le noy
  - **Routines d'interruption** (signal ds proc. ou trap, interrupt dans le noyau)

Mission 1 : Concurrence

- Interférences : risques :
  - Données fausses :

    exemple : deux grues ajoutent le poids de leur container à la charge du bateau

    G1 lit poids initial

    G2 lit poids initial

    G1 ajoute son poids

    G1 écrit

    G2 ajoute son poids

    G2 écrit
  - Données incohérentes :

    exemple : Y lit le buffer mis à jour par X avant que X n'ait fini : Y aura le début du nouveau contenu et la fin de l'ancien

=> Nécessité d'exclusion mutuelle

ne fais pas ceci pendant que je fais ça

Mission 1 : Concurrence

- Interférences : parfois utiles : coopération
  - Synchronisations
    exemple : on attend que tout le monde soit prêt avant de commencer la partie
  - Communications
    exemple : demander un service

Mission 1 : Concurrence

Table 5.1 Some Key Terms Related to Concurrency

atomic operation A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.

critical section A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

deadlock A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

livelock A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.

Mission 1 : Concurrence

Table 5.1 More Key Terms Related to Concurrency

mutual exclusion The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

race condition A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution

starvation A situation in which a runnable process is overlooked indefinitely by the scheduler: although it is able to proceed, it is never chosen

Mission 1 : Concurrence

Table 5.1 More Key Terms Related to Concurrency

mutual exclusion The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

race condition A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution (the race looser wins!)

starvation A situation in which a runnable process is overlooked indefinitely by the scheduler: although it is able to proceed, it is never chosen

Mission 1 : Concurrence

Mutual exclusion: hardware solutions

- Masquer les interruptions:
  - Retarde leur prise en compte même si elles ont urgentes
  - Sans intérêt sur multi-coeurs etc
  - Sur un monoprocessur, permet de créer facilement une section critique
- Instructions read-modify-write atomiques hardwares
  - OK pour multi-coeurs si elles sont vraiment atomiques (nécessite de figer tout ce qui pourrait accéder à la cellule de mémoire)
  - Pas facile si copies multiples de la variable (cache(s))
  - Encourage l'attente active

Mission 1 : Concurrence

Mutual exclusion: hardware solutions

- Instructions read-modify-write atomiques hardwares

Comment ça marche

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
    void P(int i){
        while (true) {
            while (compare_and_swap(bolt, 0, 1) == 1)  /* do nothing */;
            /* critical section */;
            bolt = 0;
            /* remainder */;
        }
    }
    void main(){
        bolt = 0;
        parbegin (P(1), P(2), ... ,P(n));
    }
```

Mission 1 : Concurrence

Mutual exclusion: hardware solutions

- Instructions read-modify-write atomiques hardwares

Comment ça marche

- Il y a plusieurs instructions de ce genre, toutes les machines n'implémentent pas les mêmes:
  - Test & Set
  - Shift left with carry
  - Increment
- Comme il y a attente active, acceptable uniquement si la section critique est très courte ou si on rend le processeur en cas d'écher du test (yield): faut pas être pressé alors

Mission 1 : Concurrence

Mutual exclusion: sémaphores

- Semaphore An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment.
  - the decrement operation blocks the process if value becomes <0,
  - and the increment operation may unblocks a blocked process if value becomes >0.

Also known as a counting semaphore or a general semaphore : if initial value is n (positive), n processes may enter the critical section.

- Binary Semaphore A semaphore that takes only the values 0 and 1
- Mutex Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1)
- Condition Variable A data type that is used to block a process or thread until a particular condition is true.

Mission 1 : Concurrence

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)  /* «proberen » P*/
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s) /* «Verhogen» V */
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Mission 1 : Concurrence

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;

void P(int i)
{
    while (true) {
        SemWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
}

void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual exclusion using semaphores

Mission 1 : Concurrence

Mutual exclusion: moniteurs

- Des objets particuliers tel que :
  - Les variables ne peuvent être manipulées que par les méthodes de l'objet
  - Seule une des méthodes de l'objet peut être exécutée à la fois et par un seul processus à la fois
  - Dans une méthode de l'objet, un processus peut se mettre en attente d'un événement symbolisé par une variable **e** en exécutant cwait(**e**): il sort temporairement du moniteur et se met sur la file d'attente de **e**. Un autre proc peut alors entrer ou réentrer dans le moniteur
  - Le processus en tête de la file **e** peut rentrer dans le moniteur (continuer à exécuter la méthode en cours) si un autre processus exécute csignal(**e**), ce qui fait sortir celui-ci du moniteur

Mission 1 : Concurrence

Mutual exclusion: moniteurs

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N]; /* space for N items */
int nextin, nextout; /* buffer pointers */
int count; /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */
void append (char x)
{
    if (count == N) cwait(notfull); /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal (nonempty); /*resume any waiting consumer */
}
```

Mission 1 : Concurrence

Mutual exclusion: moniteurs

```
void take (char x)
{
    if (count == 0) cwait(notempty); /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N);
    count--; /* one fewer item in buffer */
    csignal (notfull); /* resume any waiting producer */
}
{ /* monitor initialization code */
    nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
```

Mission 1 : Concurrence

```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}

void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}

void main()
{
    parbegin (producer, consumer);
}
```

Mission 1 : Concurrence

Mutual exclusion: moniteurs

Les moniteurs qui précèdent ont été proposés par Hoare en 74.

Lampson et Redell ont proposés en 80 une variante plus pratique : **csignal,** tel que proposé par Hoare impose au processus qui l'utilise de quitter immédiatement le moniteur. Ils l'ont remplacé par **cnotify** dont l'effet n'est pas immédiat mais n'intervient que lorsque le moniteur devient libre

Notons que lors du **csignal(e)** ou **cnotify(e)**, c'est le processus en têtede la liste e qui doit être activé, si un autre peut l'être avant lui, rien ne garantit que la condition e soit toujours satisfaire et il doit la retester: par exemple, le

if (count == N) **cwait(notfull);**

devient

while (count == N) **cwait(notfull);**

Il faut donc bien vérifier la sémantique exacte des moniteurs utilisés
(cela dépend de l'implémentation de cwait et csignal ou notify)

Mission 1 : Concurrence

Esquiver le problème de l'exclusion mutuelle

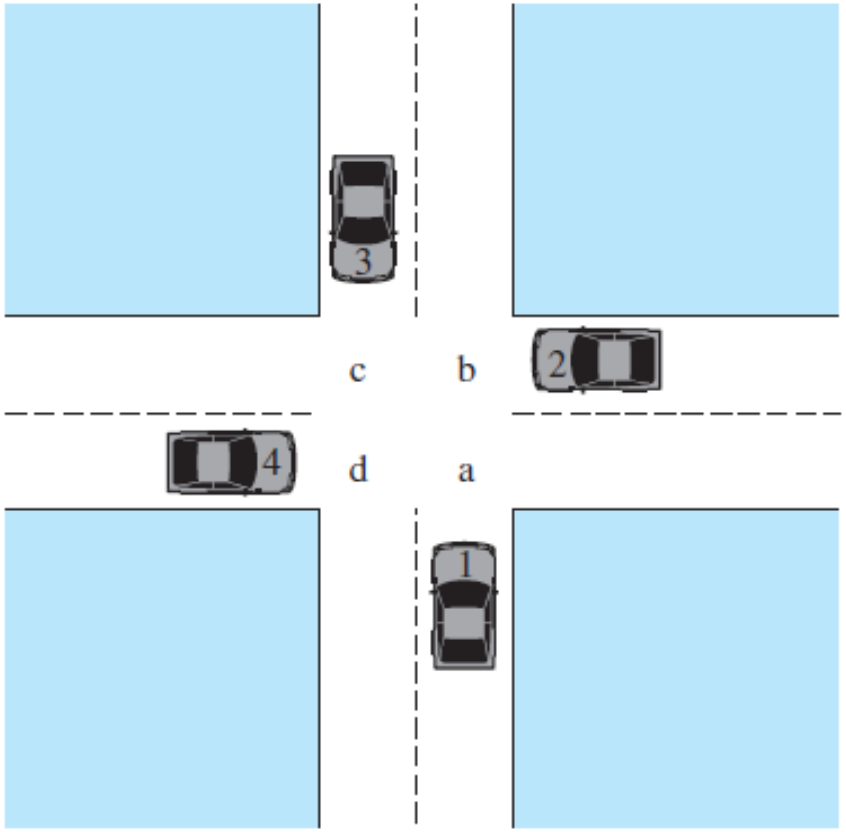
Lorsqu'on utilise l'exclusion mutuelle pour communiquer, on peut éviter les problèmes d'exclusion mutuelle en communiquant par messages et non par mémoire partagée: ce'la reporte le problème sur l'implémentation du système de messagerie

S'il s'agit de protéger des ressources non partageables (imprimante) on peut en réserver l'usage de faç!on permanente à un seul processus et envoyer des messages à celui-ci pour lui sous-traiter l'usage de la ressource (utilisé notamment dans les OS à micro-noyaux).

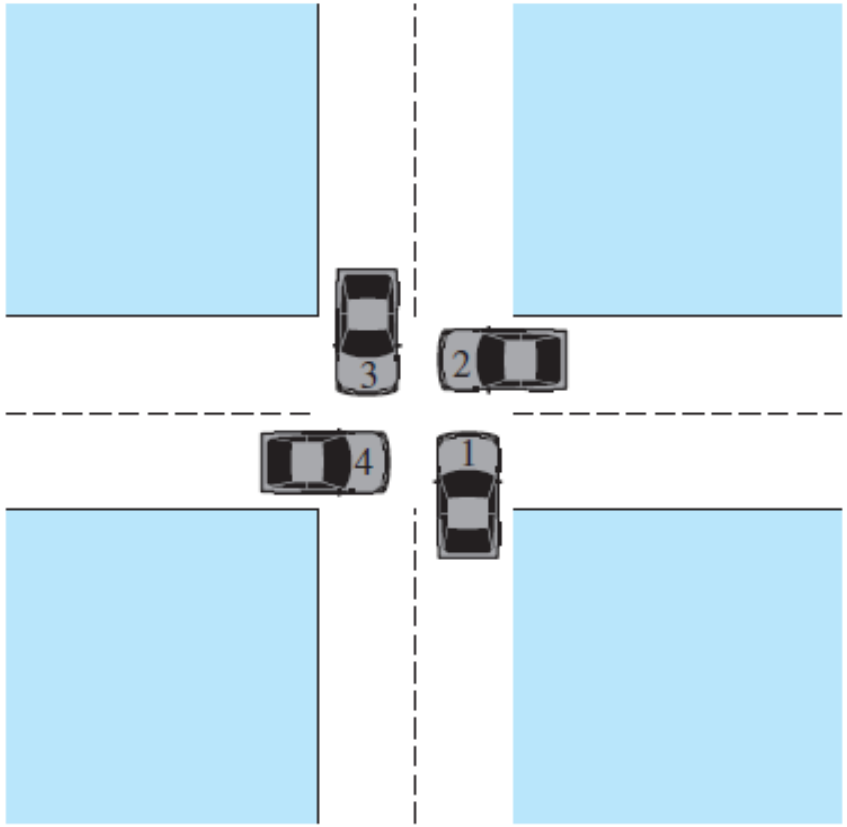
Mission 1 : Concurrence

Les deadlocks

Lorsque plusieurs processus utilisent plusieurs ressources; il y a risque de deadlocks



(a) Deadlock possible



(b) Deadlock

Figure 6.1 Illustration of Deadlock

Mission 1 : Concurrence

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses