

FONDAMENTAUX DES APPLICATIONS RÉPARTIES (FAR)

CHAP.1 : PROGRAMMATION CONCURRENTE

goo.gl/RkK8aW



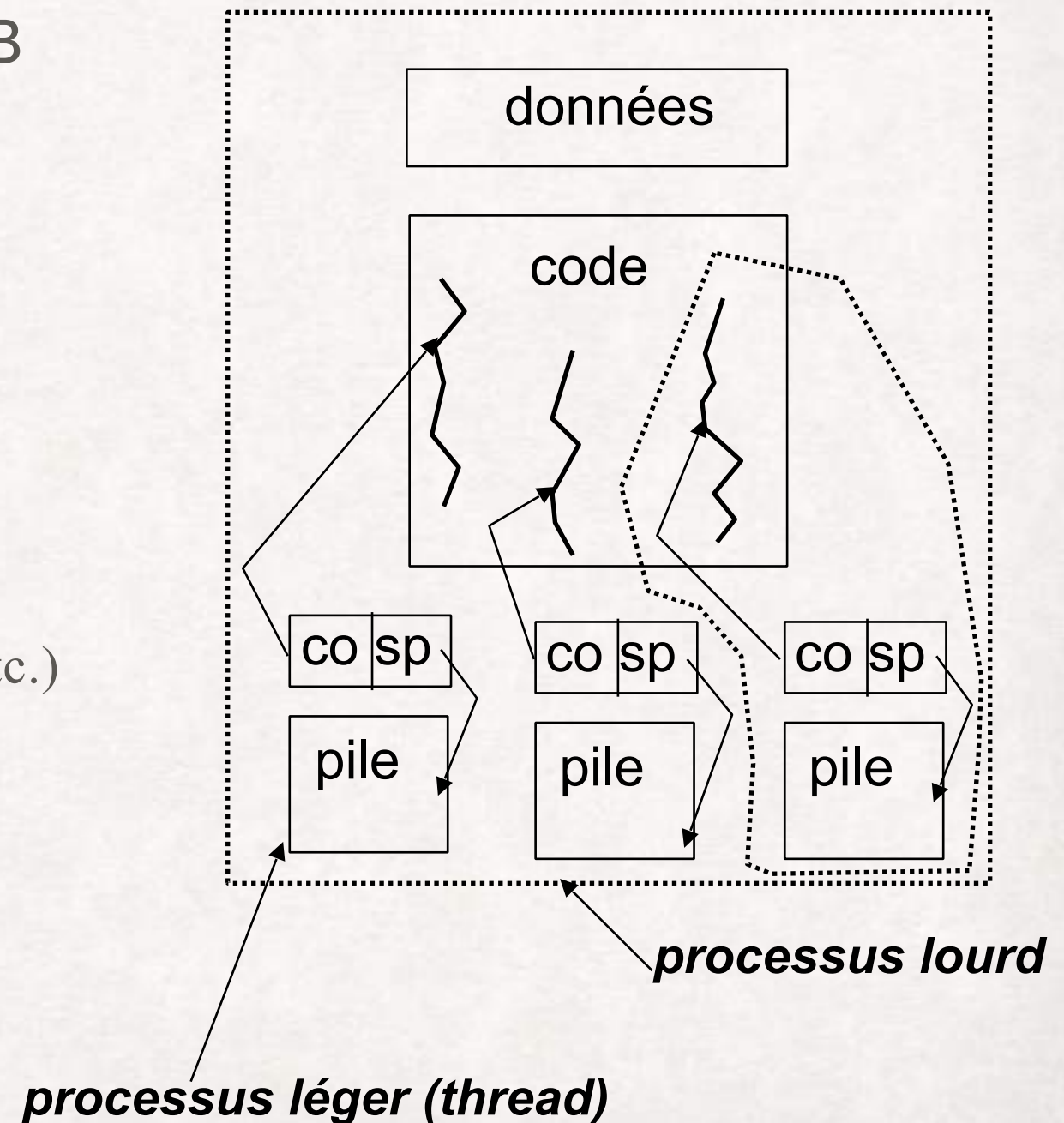
MOTIVATIONS

- S.E. = un **environnement concurrentiel** (accès à de nombreuses ressources partagées).
- Garantir qu'un processus a **accès aux ressources** dont ils a besoin
- Garantir que son **résultat est indépendant** du déroulement de l'exécution de certains processus et dépendants de ses **coopérateurs**
- Besoin de faire **communiquer** des processus threads, sur une même machine ou entre machines



PROCESSUS LÉGER OU "THREAD"

- Partage les zones de code, de données, de tas + des zones du PCB (Process Control Block) :
 - liste des fichiers ouverts, comptabilisation, répertoire de travail, userid et groupid, des handlers de signaux.
- Chaque thread possède :
 - un mini-PCB (son CO + quelques autres registres),
 - sa pile,
 - attributs d'ordonnancement (priorité, état, etc.)
 - structures pour le traitement des signaux (masque et signaux pendants).
- Un processus léger avec une seule activité
= un processus lourd.



CARACTÉRISTIQUES DES THREADS

- **Avantages**

- Création plus rapide
- Partage des ressources
- Communication entre les threads est plus simple que celle entre processus
 - communication via la mémoire : variables globales.
- Solution élégante pour les applications client/serveur :
 - une thread de connexion + une thread par requête

- **Inconvénients**

- Programmation plus difficile (mutex, interblocages)
- Fonctions de librairie non *multi-thread-safe*

Cooperating Processes

- **Advantages of process cooperation**
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience
- **Dangers of process cooperation**
 - Data corruption, deadlocks, increased complexity
 - Requires processes to synchronize their processing

Un exemple des dangers de la coopération naïve

- Deux threads exécutent cette même procédure et partagent la même base de données
- Ils peuvent être interrompus n'importe où
- Le résultat de l'exécution concurrente de P1 et P2 dépend de l'ordre de leur *entrelacement*

M. X demande une réservation d'avion

Base de données dit que fauteuil A est disponible

Fauteuil A est assigné à X et marqué occupé

Vue globale d'une exécution possible



P1

M. Leblanc demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Leblanc et marqué occupé

Interruption
ou retard

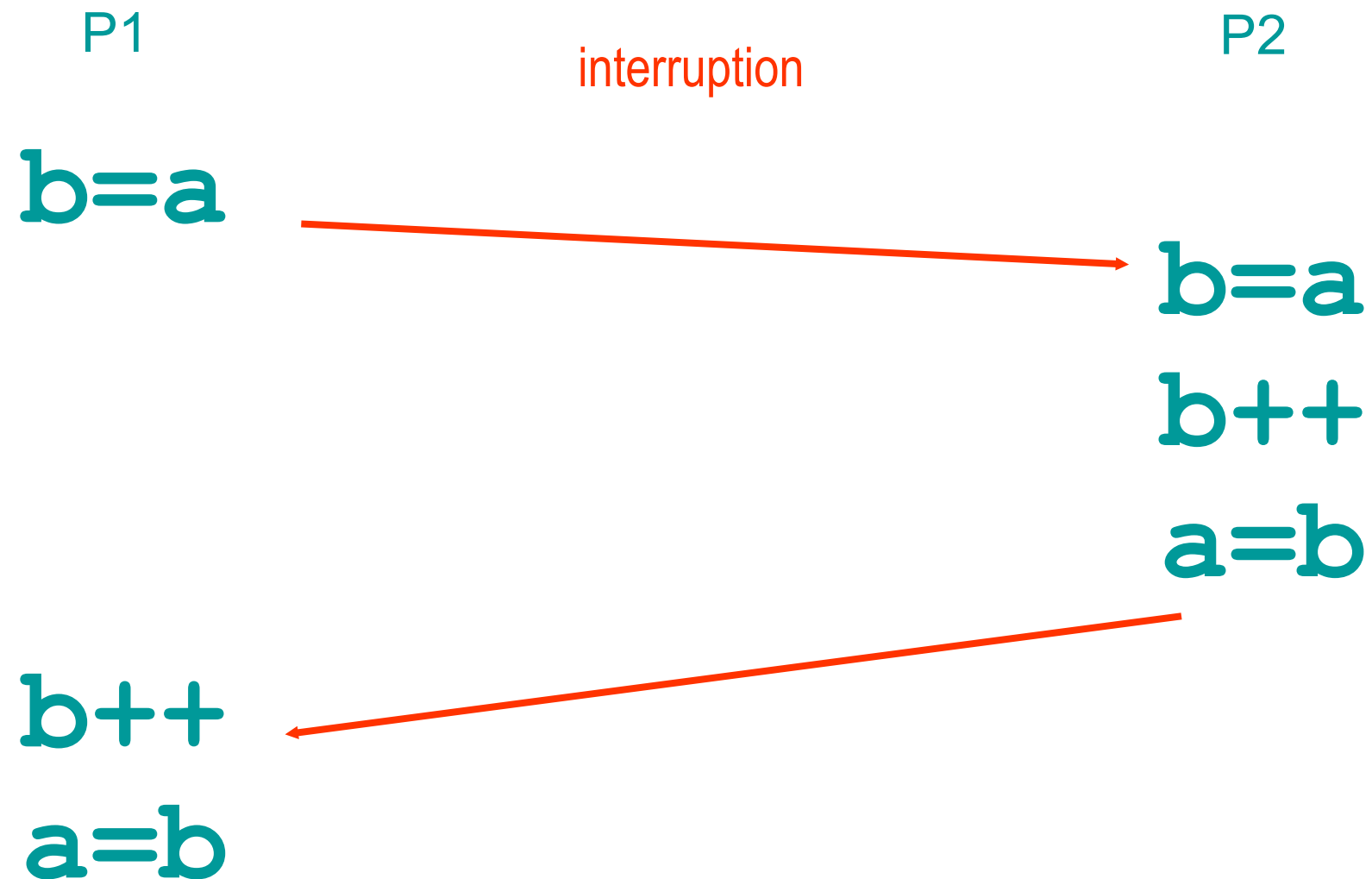
P2

M. Guy demande une réservation d'avion

Base de données dit que fauteuil 30A est disponible

Fauteuil 30A est assigné à Guy et marqué occupé

Deux opérations en parallèle sur une var a partagée (b est privé à chaque processus)



Supposons que `a` soit 0 au début

P1 travaille sur le vieux `a` donc le résultat final sera `a=1`.

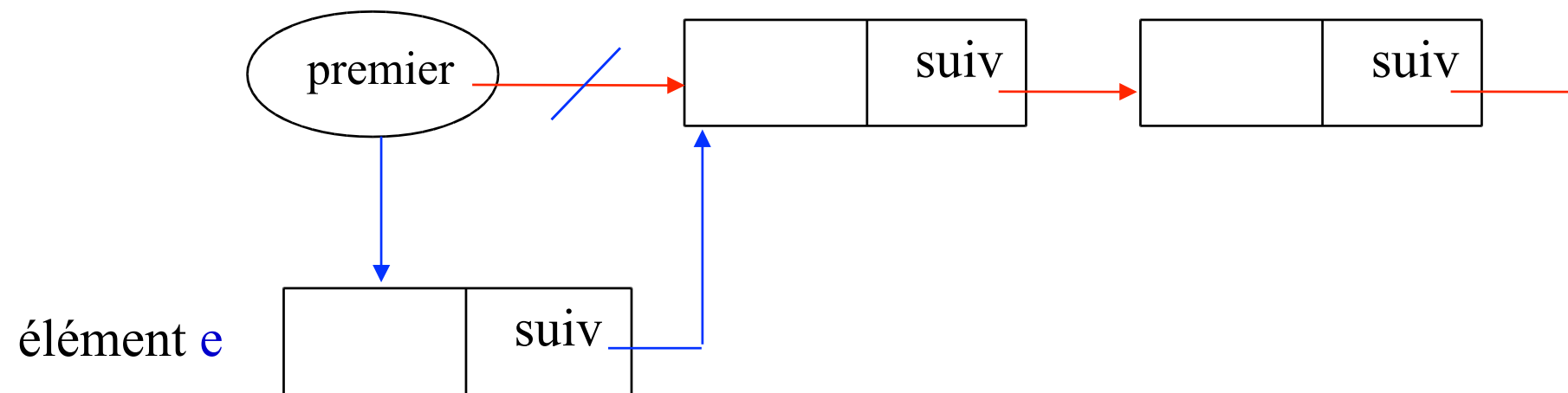
Sera `a=2` si les deux tâches sont exécutées l'une après l'autre

Si `a` était sauvegardé quand P1 est interrompu, il ne pourrait pas être partagé avec P2 (il y aurait deux `a` tandis que nous en voulons une seule)

EXERCICE

TROUVER UN ORDONNANCEMENT CATASTROPHE DES INSTRUCTIONS EXECUTÉES PAR 2 PROCESSUS EN PARALLÈLE

Exemple 2 : liste simplement chaînée avec **insertion en tête**

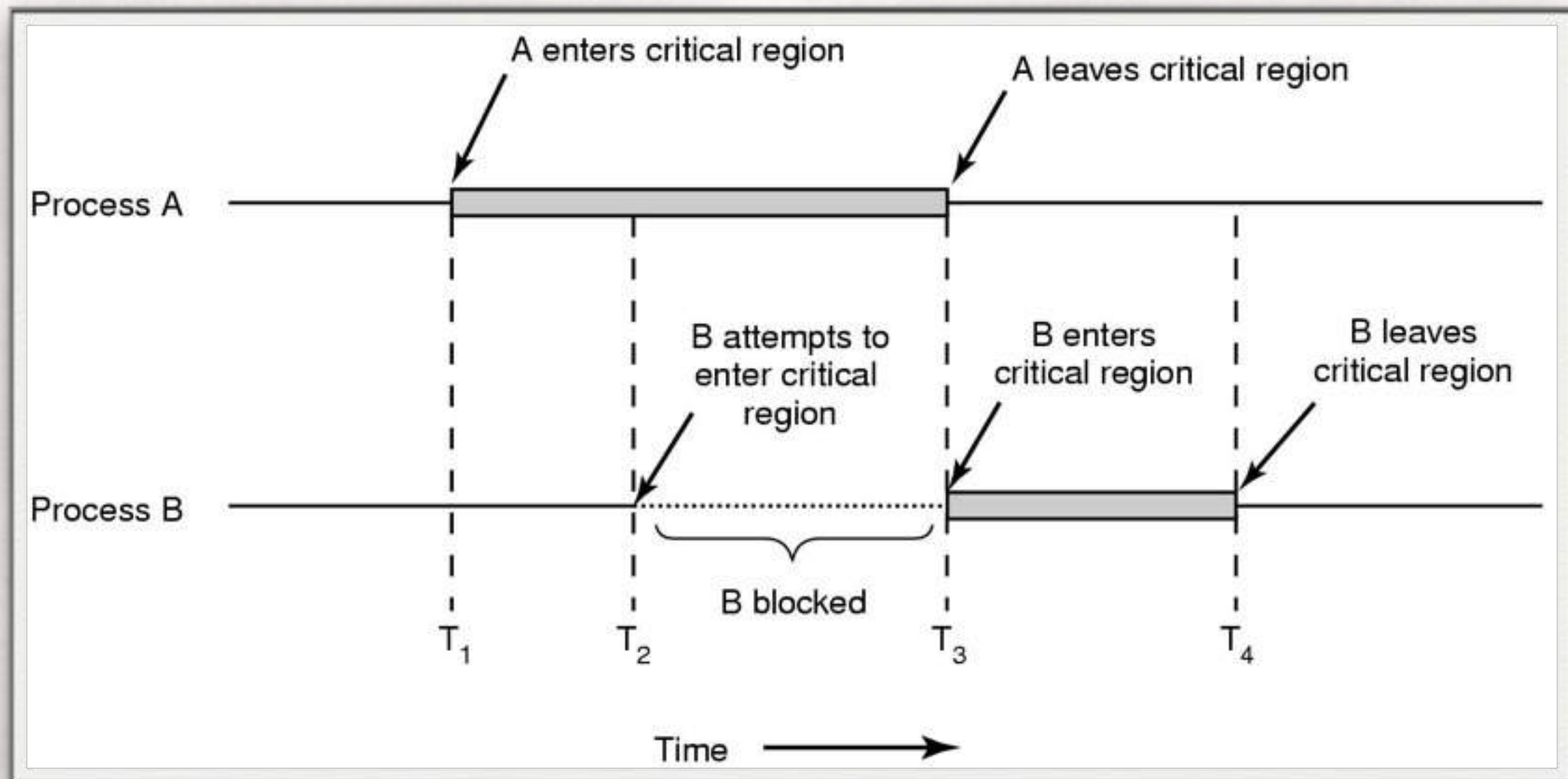


1. $e.suiv := premier$;
2. $premier := e$;

Supposons que 2 processus tentent simultanément d'introduire un élément dans la liste

SECTION CRITIQUE

- Partie d'un programme dont l'exécution ne doit pas être *entrelacer* avec autres programmes
- Une fois qu'une tâche y entre, il faut lui permettre de terminer cette section sans permettre à autres tâches de jouer sur les mêmes données



PLAN

1. Exclusion mutuelle
2. Synchronisation
3. Threads



RELATIONS
ENTRE
PROCESSUS

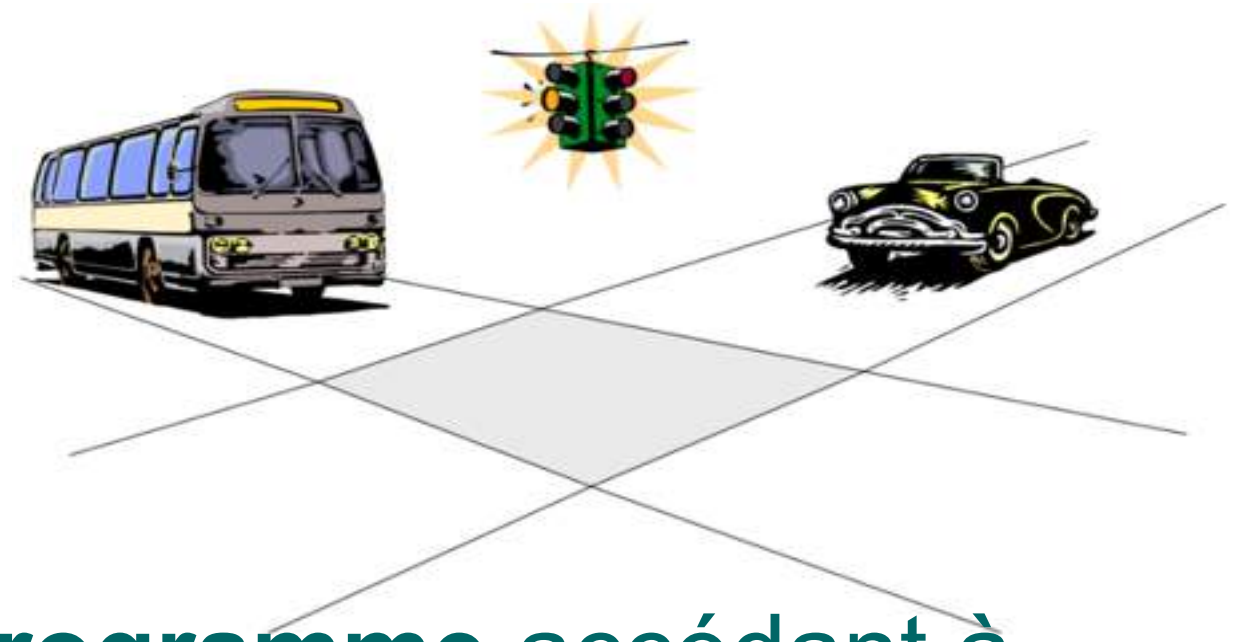
PARTIE 1

EXCLUSION MUTUELLE



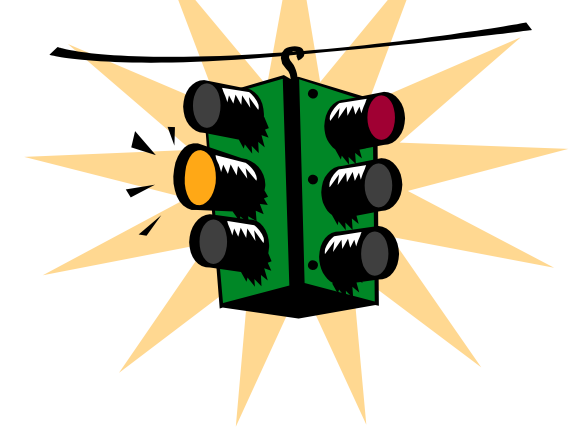
(c) Bill Bramhall

Section Critique (SC)



- **Partie du code d'un programme** accédant à une ressource partagée qui ne doit pas être manipulée par plus d'un fil d'exécution (thread) à la fois.
- L'exécution de cette partie de code ne doit donc pas être *entrelacée* avec des parties de codes d'autres programmes manipulant la ressource en question
- Une fois qu'une tâche y entre, il faut lui permettre de terminer cette section sans permettre à autres tâches de jouer sur les mêmes données

Le problème de la section critique



- Le problème de la section critique est de trouver un **algorithme d'exclusion mutuelle** de threads dans l'exécution de leur SC afin que le résultat de leurs actions *ne dépendent pas de l'ordre d'entrelacement* de leur exécution (avec un ou plusieurs processeurs)
- L'exécution des sections critiques doit être **mutuellement exclusive**: à tout instant, **un seul thread** peut exécuter une SC pour une var donnée (même lorsqu'il y a plusieurs processeurs)
- Ceci peut être obtenu en plaçant des instructions spéciales dans les **sections d'entrée et sortie**

MUTUAL EXCLUSION

- Two events are **mutually exclusive** if they cannot occur at the same time.
- An example is tossing a coin once, which can result in either heads or tails, but not both.
- Failure to do so opens up the possibility of corrupting the shared state.



Structure du programme

- Chaque thread doit donc demander une permission avant d'entrer dans une section critique (SC)
- La section de code qui effectue cette requête est la **section d'entrée**
- La section critique est normalement suivie d'une **section de sortie**
- Le code qui reste est la **section restante** (SR): non-critique

repeat

section d'entrée

Section Critique (SC)

section de sortie

Section Restante (SR)

forever

Application

M. X demande une
réservation d'avion

Section d'entrée

Section
critique

Base de données dit que fauteuil
A est disponible

Fauteuil A est assigné à X et
marqué occupé

Section de sortie

Critères nécessaires pour solutions valides

- **Exclusion Mutuelle:**

- ◆ À tout instant, au plus un thread peut être dans une section critique (SC) pour une variable donnée

- **Progrès:**

- ◆ **absence d'interblocage** : un processus ne doit pas être empêché par un autre d'entrer en SC si aucun autre n'y est
- ◆ **absence de famine** : si un thread demande d'entrer dans une section critique à un moment où aucun autre thread en fait requête, il devrait être en mesure d'y entrer

- ◆ **Non interférence:**

- ✦ Si un thread termine **en dehors** sa **section critique**, ceci ne devrait pas affecter les autres threads

- **Attente limitée (bounded waiting):**

- ◆ aucun thread éternellement empêché d'atteindre sa SC (pas de famine)

TYPES DE SOLUTIONS

1. Solutions par logiciel

- des algorithmes dont la validité ne s'appuie pas sur l'existence d'instructions spéciales

2. Solutions fournies par le matériel

- s'appuient sur l'existence de certaines instructions (du processeur) spéciales

3. Solutions fournies par le S.E.

- appels systèmes dispos pour les programmeurs

→ Toutes les solutions se basent sur l'**atomicité de l'accès à la mémoire centrale**: une adresse de mémoire ne peut être affectée que par une instruction à la fois, donc par un thread à la fois.

→ Remarque : toutes les solutions se basent sur l'existence d'instructions atomiques, qui fonctionnent comme une SC de base

Atomicité = indivisibilité

I - SOLUTIONS PAR LOGICIEL

(INTÉRESSANTES POUR COMPRENDRE LE PB)

- ❖ Nous considérons d'abord 2 threads :
 - ❖ Algorithmes 1 et 2 ne sont pas valides mais montrent la difficulté du problème
 - ❖ Algorithme 3 est valide (algorithme de Peterson)
- ❖ Notations :
 - ❖ Débutons avec 2 threads: T0 et T1
 - ❖ Lorsque nous discutons de la tâche T_i , T_j dénotera toujours l'autre tâche ($i \neq j$)

Algorithme 1: threads se donnent mutuellement le tour

- ❏ La **variable partagée** **tour** est initialisée à 0 ou 1
- ❏ La SC de T_i est exécutée ssi $\text{tour} = i$
- ❏ T_i est **occupé à attendre** si T_j est dans SC.
- ❏ Fonctionne pour l'exclusion mutuelle!
- ❏ Pas de famine (seulement 1 thread à son tour selon **tour**).
- ❏ Mais critère du progrès n'est pas satisfait car l'exécution des SCs doit strictement alterner

```
Thread  $T_i$ :  
repeat  
    while ( $\text{tour} \neq i$ ) { } ;  
    SC  
     $\text{tour} = j$  ;  
    SR  
forever
```

↑
ne rien faire

Ex 1: T_0 possède une longue SR et T_1 possède une courte SR. Si $\text{tour} = 0$, T_0 entre dans sa SC et puis sa SR ($\text{tour} = 1$). T_1 entre dans sa SC et puis sa SR ($\text{tour} = 0$), et tente d'entrer dans sa SC: refusée! il doit attendre que T_0 lui donne le tour.

initialisation de tour à 0 ou 1

Thread T0:

repeat

while (tour != 0) {} ;

SC

tour = 1;

SR

forever

Thread T1:

repeat

while (tour != 1) {} ;

SC

tour = 0;

SR

forever

Algorithme 1 vue globale

Ex 2: Généralisation à n threads: chaque fois, avant qu'un thread puisse rentrer dans sa section critique, il lui faut attendre que tous les autres aient eu cette chance!

ALGORITHME 2 OU L'EXCÈS DE COURTOISIE...

- Une variable Booléenne par Thread: flag[0] et flag[1]
- Ti signale qu'il désire exécuter sa SC par: flag[i] =vrai
- Mais il n'entre pas si l'autre est aussi intéressé!
- Exclusion mutuelle ok
- Progrès ok
- Exempt de tout soucis alors ?

Thread Ti:

repeat

flag[i] = vrai;

while(flag[j]==vrai) {};

SC

flag[i] = faux;

SR

forever



rien faire

Algorithme 2 vue globale

Thread T0:

repeat

flag[0] = vrai;
while(flag[1]==vrai) {};

SC

flag[0] = faux;

SR

forever

APRÈS
VOUS !

Thread T1:

repeat

flag[1] = vrai;
while(flag[0]==vrai) {};

SC

flag[1] = faux;

SR

forever

APRÈS
VOUS !

QUEL PEUT ÊTRE LE
SOU CIS AVEC CET
ALGORITHME

?

T0: flag[0] = vrai
T1: flag[1] = vrai
interblocage!



ALGORITHME 3 (PETERSON)

COMBINE LES DEUX IDÉES: ON UTILISE 2 VARS PARTAGÉES :
FLAG[i]=INTENTION D'ENTRER; **TOUR**=À QUI LE TOUR



Initialisation:

- $\text{flag}[0] = \text{flag}[1] = \text{faux}$
- $\text{tour} = i \text{ ou } j$



Désire d'exécuter
SC est indiqué par
 $\text{flag}[i] = \text{vrai}$



$\text{flag}[i] = \text{faux}$ à la
section de sortie

Thread T_i :

repeat

$\text{flag}[i] = \text{vrai};$

// je veux entrer

$\text{tour} = j; // \text{donne une chance à}$

// l'autre process

do while

$(\text{flag}[j] == \text{vrai} \ \&\& \ \text{tour} == j) \{ \};$

SC

$\text{flag}[i] = \text{faux};$

SR

forever

ENTRER OU ATTENDRE ?

- ❖ Thread T_i attend si:
 - ❖ T_j veut entrer est c'est le tour de T_j
 - ❖ `flag[j]==vrai` et `tour==j`
- ❖ Un thread T_i entre si:
 - ❖ T_j ne veut pas entrer ou c'est le tour de T_i
 - ❖ `flag[j]==faux` ou `tour==i`
- ❖ Pour entrer, un thread dépend de la bonne volonté de l'autre qu'il lui donne la chance!

```
Thread  $T_i$ :
repeat
    flag[i] = vrai;
    // je veux entrer
    tour = j; // après vous
do while
    (flag[j]==vrai && tour==j) {};
SC
flag[i] = faux;
SR
forever
```

Thread T0:

repeat

flag[0] = vrai;

// T0 veut entrer

tour = 1;

// T0 donne une chance à T1

while

(flag[1]==vrai&&tour=1){};

SC

flag[0] = faux;

// T0 ne veut plus entrer

SR

forever

Thread T1:

repeat

flag[1] = vrai;

// T1 veut entrer

tour = 0;

// T1 donne une chance à 0

while

(flag[0]==vrai&&tour=0){};

SC

flag[1] = faux;

// T1 ne veut plus entrer

SR

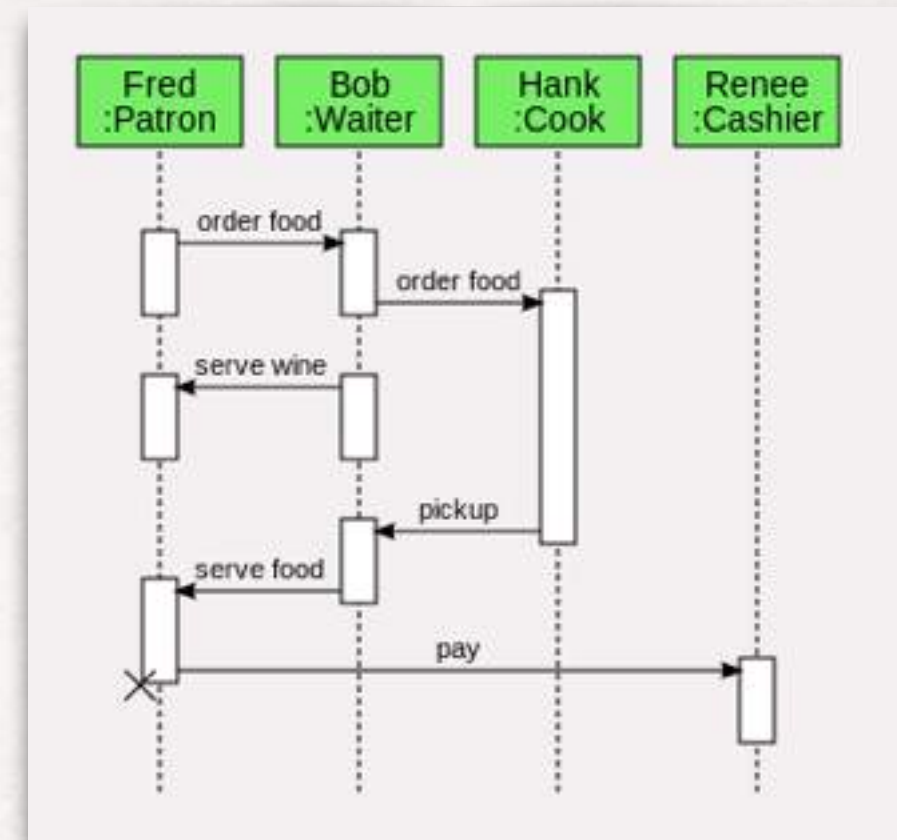
forever

Algorithme de Peterson vue globale

EXERCICE

DIAGRAMME DE SÉQUENCE (UML)

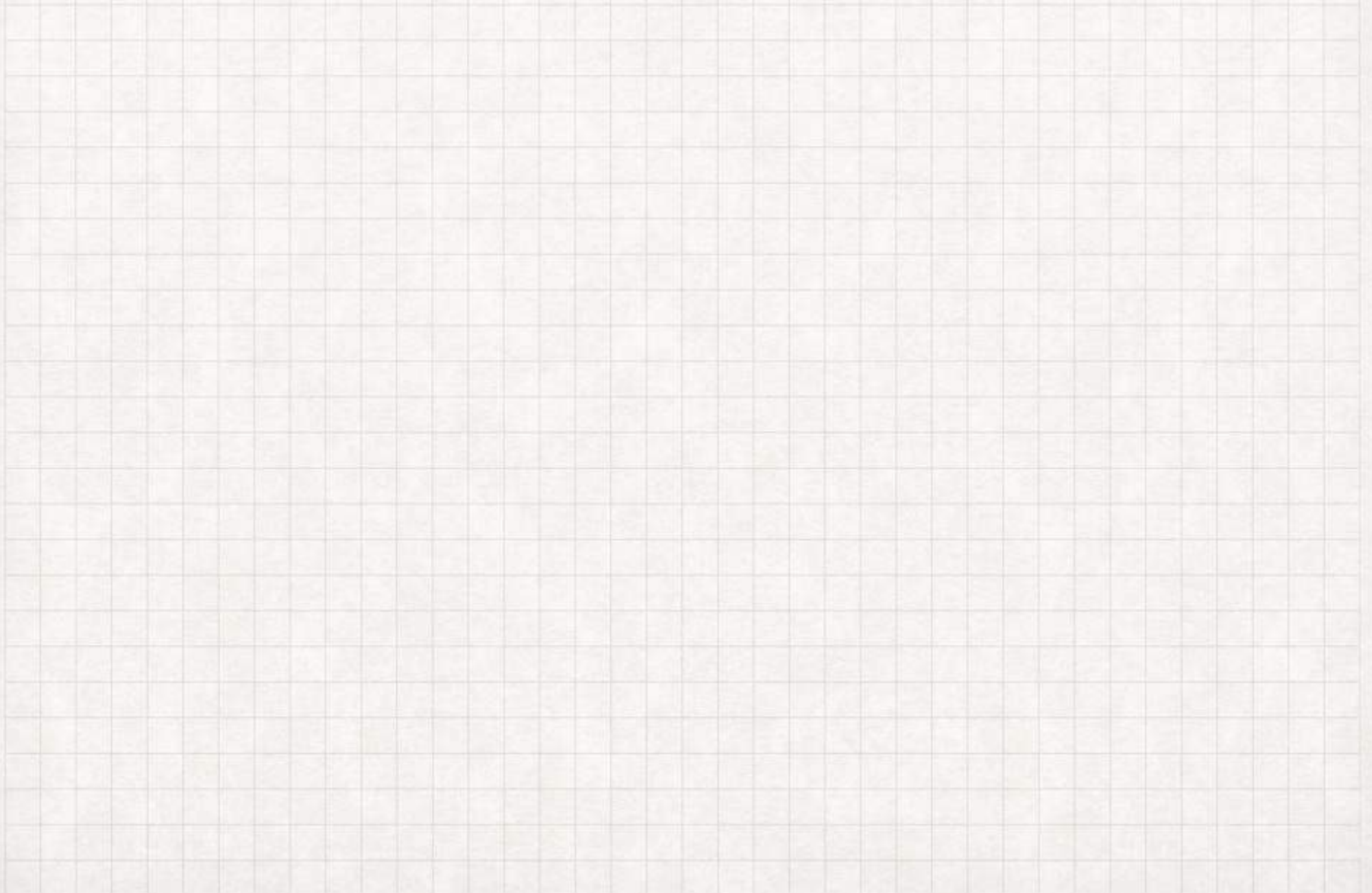
vous tracerez les valeurs des variables partagées et les actions des threads T0 et T1 au cours du temps



source figure : wikipedia

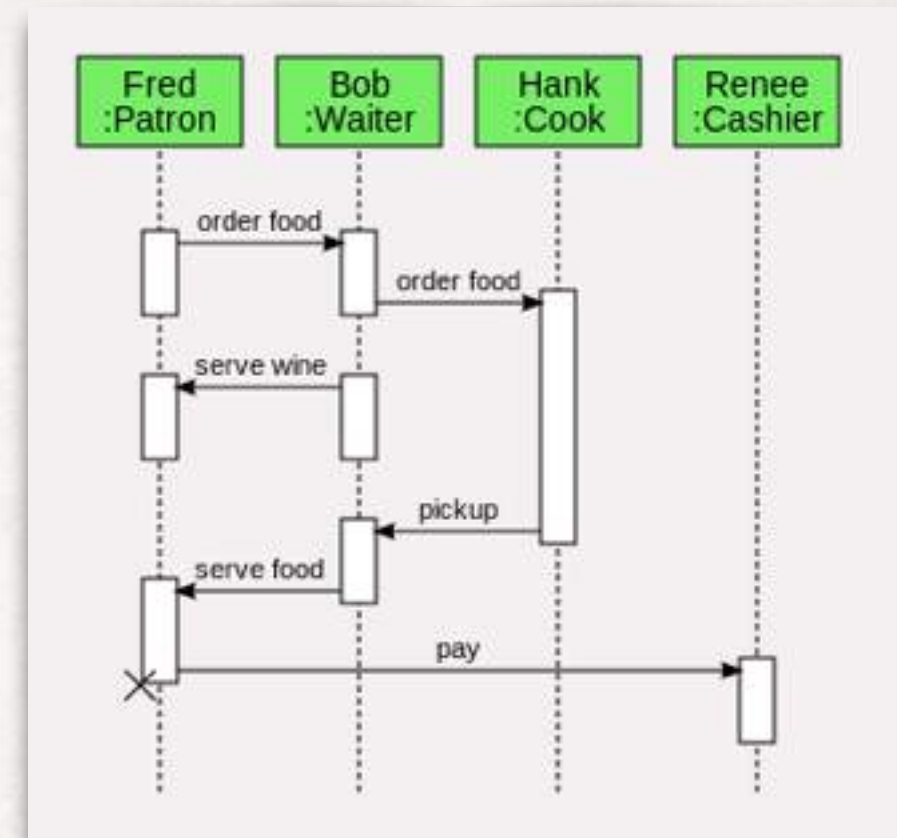
- Intéressez-vous au cas où T1 est le dernier qui est entré en SC, les deux processus ont tous deux ensuite exécuté des instructions et redemandent maintenant presque simultanément à entrer en SC. Tracez sur la page suivante un diagramme de séquence pour montrer qui y accède.
- Qu'en concluez vous ?

Diagramme de séquence pour l'exercice :



EXERCICE SUITE

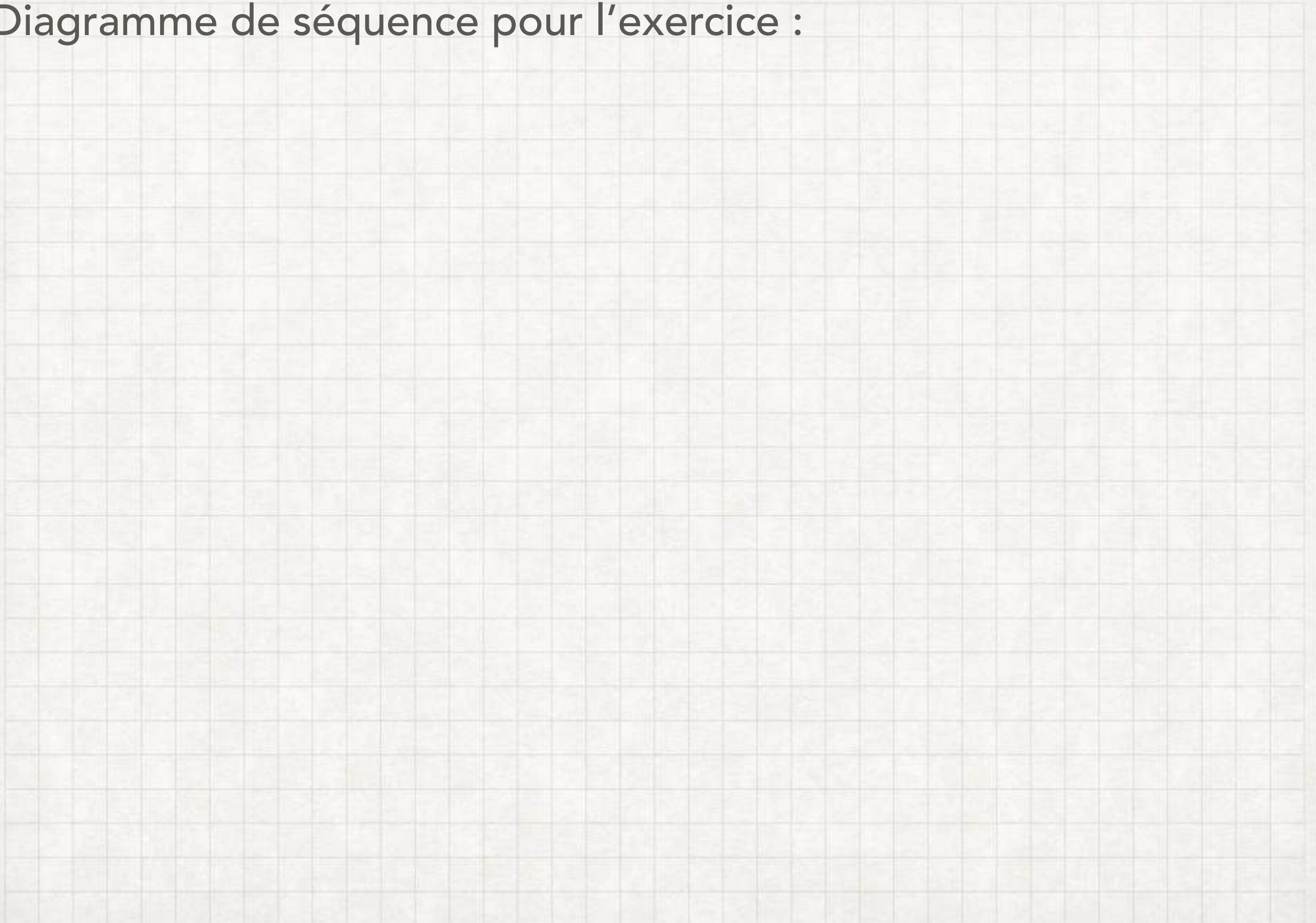
DIAGRAMME DE SÉQUENCE (UML)



source figure : wikipedia

- Intéressez-vous maintenant au cas où T1 vient d'accéder à la SC, et où il est dans une longue SR. Question : est-ce que T0 peut effectuer plusieurs fois la SC en attendant ? Expliquer pourquoi avec un diagramme de séquence

Diagramme de séquence pour l'exercice :



L'ALGO 3 N'OBLIGE PAS UNE TÂCHE À ATTENDRE POUR RIEN

SUPPOSONS QUE T0 SOIT LE SEUL À AVOIR BESOIN DE LA SC,
OU QUE T1 SOIT LENT À AGIR: T0 PEUT RENTRER DE SUITE
(FLAG[1]==FAUX LA DERNIÈRE FOIS QUE T1 EST SORTI)

```
flag[0] = vrai // prend l'initiative
```

```
tour = 1 // donne une chance à l'autre
```

```
while flag[1]==vrai && tour=1 {} //test faux, entre
```

SC

```
flag[0] = faux // donne une chance à l'autre
```

Cette propriété est désirable

ACHIEVEMENT OF PETERSON'S ALGORITHM

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting (Fixed – Waiting) requirement is met.

UNE LEÇON À RETENIR...

Afin que des threads avec des variables partagées puissent fonctionner correctement collectivement, ont fait exécuter les mêmes règles d'accès / de coordination à la SC à tous les threads impliqués

→ **Un protocole commun**

CRITIQUE DES SOLUTIONS PAR LOGICIEL

- ✱ Difficiles à programmer! Et à comprendre!
 - ✱ Les solutions que nous verrons dorénavant sont toutes basées sur l'existence d'instructions spécialisées, qui facilitent le travail.
- ✱ Les threads qui requièrent l'entrée dans leur SC sont **occupés à attendre (busy waiting)**; consommant ainsi du temps de processeur
 - ✱ Pour de longues sections critiques, il serait préférable de **bloquer** les threads qui doivent attendre...

II - SOLUTIONS PAR LE MATÉRIEL

A) SUSPENSION DES INTERRUPTIONS

- Sur un uniprocasseur: l'exclusion mutuelle est préservée mais l'efficacité se détériore: lorsque dans SC il est impossible d'entrelacer l'exécution avec d'autres threads dans une SR
- Perte d'interruptions
- Sur un multiprocasseur: exclusion mutuelle n'est pas préservée
- Une solution qui n'est généralement pas acceptable

Process P_i :

...

```
inhiber interrupt  
section critique  
rétablir interrupt  
section restante
```

...

II - SOLUTIONS MATÉRIELLES

B) INSTRUCTIONS MACHINES SPÉCIALISÉES

- ❖ Habituellement, pendant qu'un thread ou processus accède à une adresse de mémoire, aucun autre ne peut faire accès à la même adresse en même temps
- ❖ **Extension:** instructions machine exécutant **plusieurs** actions (ex: lecture **puis** écriture) sur la même case de mémoire de manière **atomique (indivisible)**
- ❖ Une instruction **atomique** ne peut être exécutée que **par un thread à la fois** (même en présence de plusieurs processeurs)

L'INSTRUCTION TEST-AND-SET

```
enter_region:  
    TSL REGISTER, LOCK  
    CMP REGISTER, #0  
    JNE enter_region  
    RET
```

- | copy lock to register and set lock to 1
- | was lock zero?
- | if it was nonzero, lock was set, so loop
- | return to caller; critical region entered

```
leave_region:  
    MOVE LOCK, #0  
    RET
```

- | store a 0 in lock
- | return to caller

TSL is **atomic**, guaranteed by hardware. Memory bus is locked until it is finished executing.

L'INSTRUCTION TEST-AND-SET

- Variable partagée LOCK est initialisée à 0
- C'est le 1er Pi qui met LOCK à 1 qui entre dans SC

Tâche Pi:

enter_region()


SC

leave_region()

SR

```
enter_region:
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```

-  Exclusion mutuelle est assurée: si Ti entre dans SC, l'autre Tj est **occupé à attendre**

WHAT'S WRONG WITH PETERSON, TEST-AND-SET, ...?

- Both Peterson's solution and the solution using TSL are correct, but both have the defect of requiring **busy waiting**.
- In essence, what these solutions do is this: when a process wants to enter its critical region, it checks to see if the entry is allowed. If it is not, the process just sits in a tight **loop waiting** until it is.

PARTIE II - SÉMAPHORES ET SYNCHRONISATION



SÉMAPHORES

- ✦ Les sémaphores sont un mécanisme qui permet de réaliser l'exclusion mutuelle sur une SC et de façon plus générale de synchroniser des processus
- ✦ Il est partagé entre tous les threads qui s'intéressent à la même **SC**
- ✦ Un sémaphore S est une **file d'attente** + **un entier** qui, sauf pour l'Initialisation, est accessible seulement par **deux opérations atomiques et mutuellement exclusives** : **P(S)** et **V(S)** qui modifient l'entier et la file d'attente

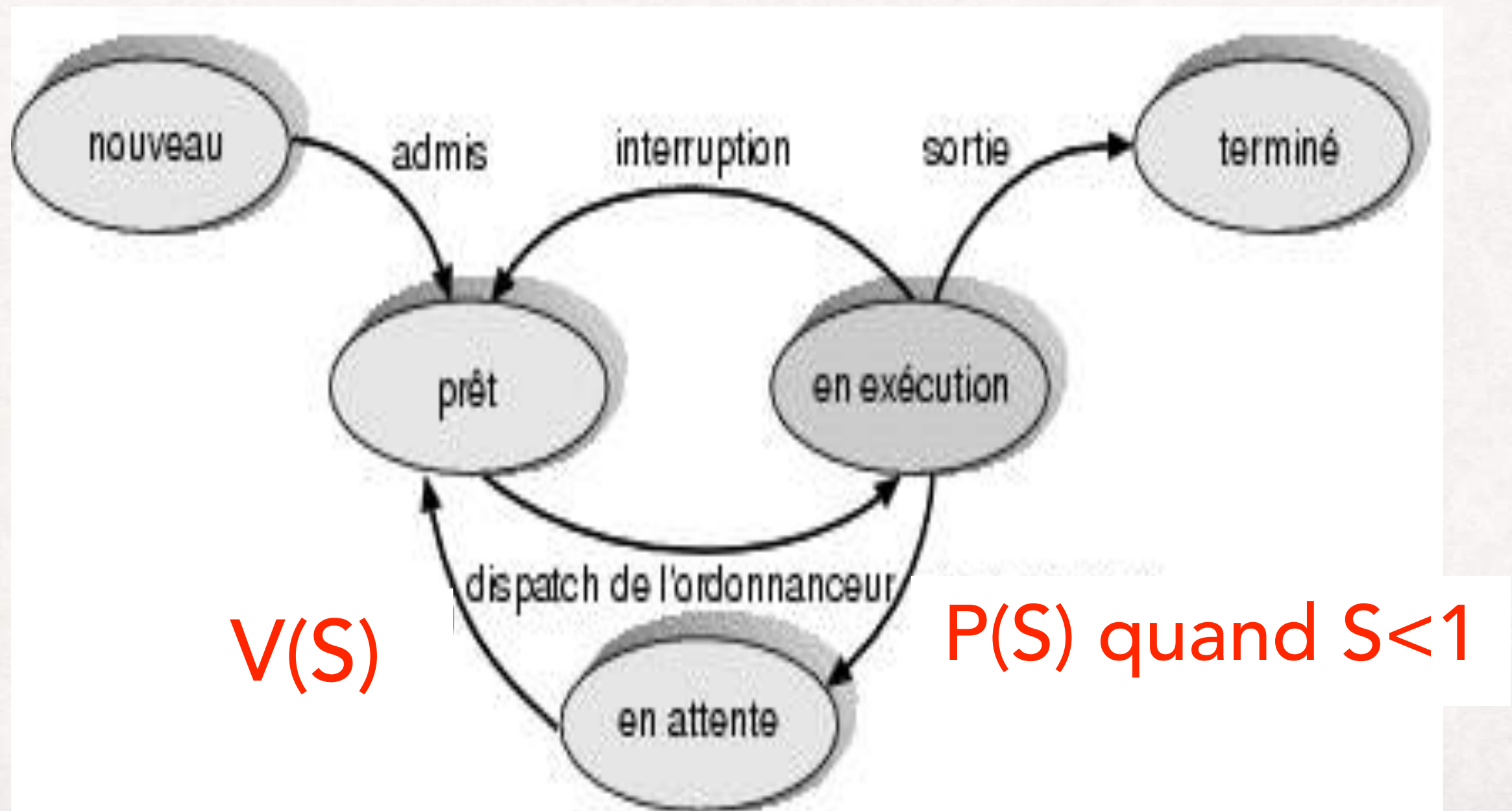
SÉMAPHORES

- **P(S) or wait(S):** Decrements the value of semaphore variable by 1. **If the value becomes negative**, the process executing wait() is blocked, i.e., added to the semaphore's queue.
- **V(S) or signal(S):** Increments the value of semaphore variable by 1. After the increment, if the pre-increment value was negative (meaning there are processes waiting for a resource), it transfers a blocked process from the semaphore's waiting queue to the ready queue. (like Wake up)
- ♦ La valeur de l'entier de S à un instant donné =
$$\text{init}(S) + \#V(S) - \#P(S)$$



SÉMAPHORES

- ♦ Les sémaphores sont un mécanisme proposé par le **S.E.** : quand un processus/thread est mis dans la file d'attente, il passe à l'état bloqué (sleep) : **pas d'attente active**



UTILISATION DES SÉMAPHORES POUR SECTIONS CRITIQUES

COMMENT L'UTILISER POUR RÉALISER UNE EXCLUSION MUTUELLE

- ✦ Pour n threads
- ✦ Initialiser S à 1
- ✦ Alors 1 seul thread peut être dans sa SC
- ✦ Il s'agit ici d'un sémaphore binaire : un seul processus peut entrer en SC.

Thread Ti :

...

wait(S) ;

SC

signal(S) ;

SR

...

SÉMAPHORES

GÉNÉRALISATION À PLUS D'UN THREAD EN SC

- Dans certaines applications on peut **autoriser plus d'un thread en SC**. Dans ce cas on initialise S à une valeur >1

Thread T1:
repeat
 wait(S)
 SC
 signal(S) ;
 SR
forever

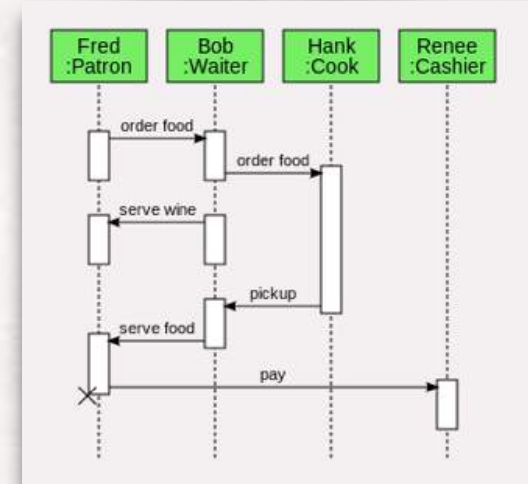
.....

Thread Tn:
repeat
 wait(S) ;
 SC
 signal(S) ;
 SR
forever

- Dans ce cas on pense à la variable S (comme au nombre de places disponibles pour la SC). Exemple : un sémaphore pour télésiège sur les remontées mécaniques serait initialisé à 3 ou 4 suivant le nombre de places

EXERCICE

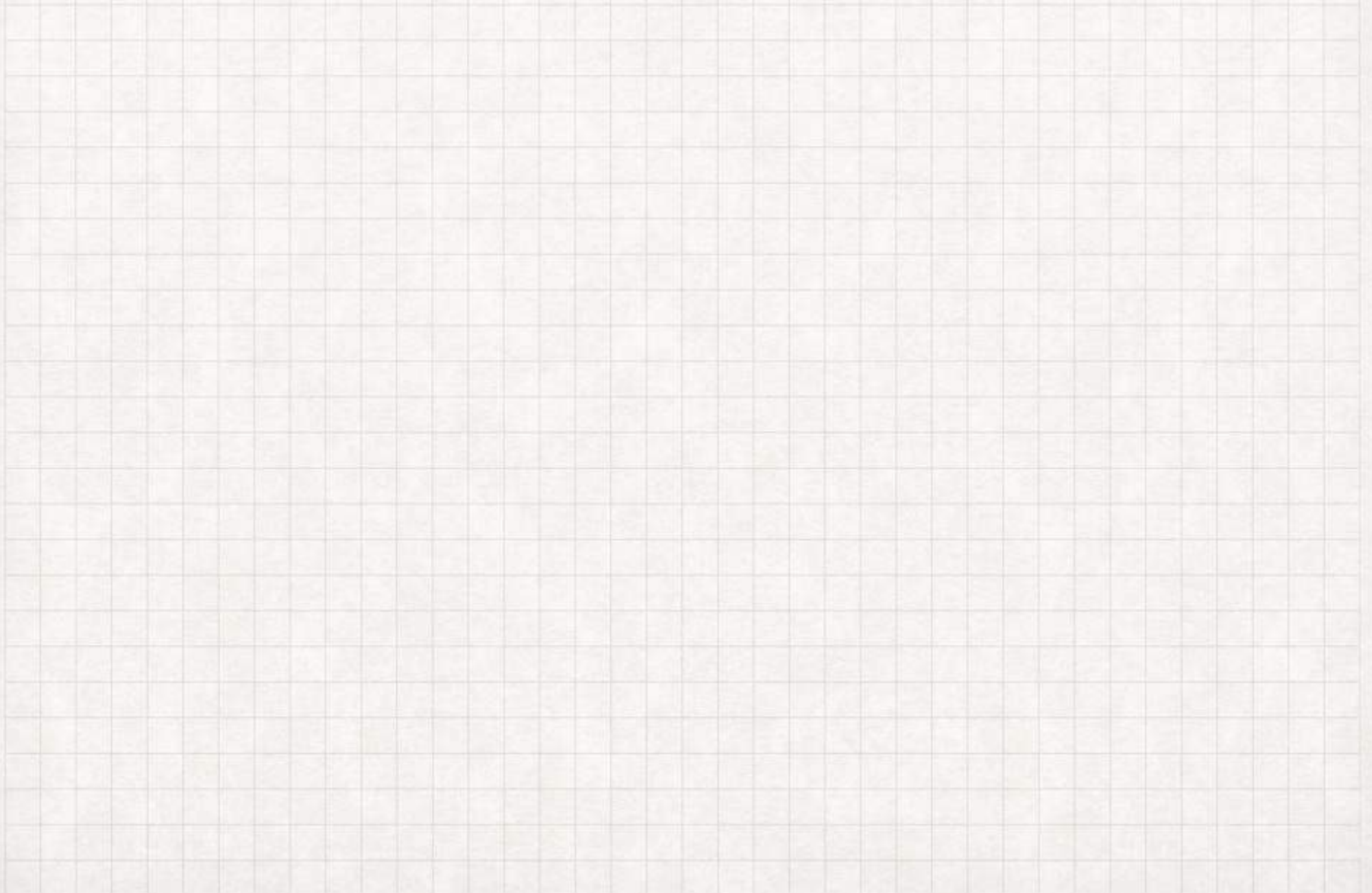
DIAGRAMME DE SÉQUENCE (UML)



source figure : wikipedia

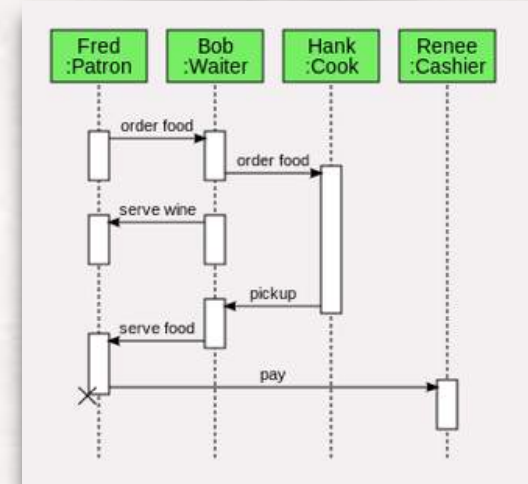
- Intéressez-vous au cas où **quatre** étudiants IG3 font la traditionnelle descente du concours de luge lors de la semaine blanche;
 - Le département ne dispose que de deux luges, qui sont remontées par les gens qui les ont empruntées dès qu'ils ont descendu la piste avec.
 - Chaque luge ne prend qu'une personne à la fois (petit modèle de luge)
1. Ecrivez sous forme de code le comportement effectué en parallèle par ces personnes.
 2. Tracez sur la page suivante un diagramme montrant qu'ils peuvent tous descendre la piste, chacun au bout d'un temps fini.
 3. Dans cet exemple est-il possible de se trouver dans une situation d'interblocage ?

Diagramme de séquence pour l'exercice :



EXERCICE

DIAGRAMME DE SÉQUENCE (UML)



source figure : wikipedia

- Intéressez-vous au cas où **quatre** étudiants IG3 veulent tous entrer dans la même agence bancaire locale, afin de faire un virement à leur enseignants bien aimés en prévision des futurs examens
- la banque dispose d'un sas d'entrée particulier : il ne contient que deux places mais ne marche que si deux personnes sont entrées : c'est un ascenseur manuel : il faut pousser très fort sur la porte d'entrée dans la banque, ce qui nécessite d'être deux
- Cet exemple est-il modélisable sous forme de sémaphores, si oui comment (écrivez le code), sinon pourquoi ?

UTILISATION DES SÉMAPHORES POUR SYNCHRONISATION

SYNCHRONISATION DE THREADS

- On a 2 threads T1 et T2
- On veut garantir que la ligne L1 dans T1 soit exécutée avant la ligne L2 dans T2
- On utilise un sémaphore S
- Initialiser S à 0

■ On écrit T1 ainsi :

```
...  
L1;  
signal(S);
```

■ et T2 ainsi :

```
...  
wait(S);  
L2;
```

INTERBLOCAGE ET FAMINE AVEC LES SÉMAPHORES

ATTENTION À L'ORDRE D'UTILISATION !

- ♦ Dans certains cas on utilise plusieurs sémaphores (chacun ayant son entier, et sa file d'attente)
- ♦ Attention à l'**interblocage**: supposons deux sémaphores S et Q initialisés à 1

T0

wait(S)

wait(Q)

T1

wait(Q)

wait(S)



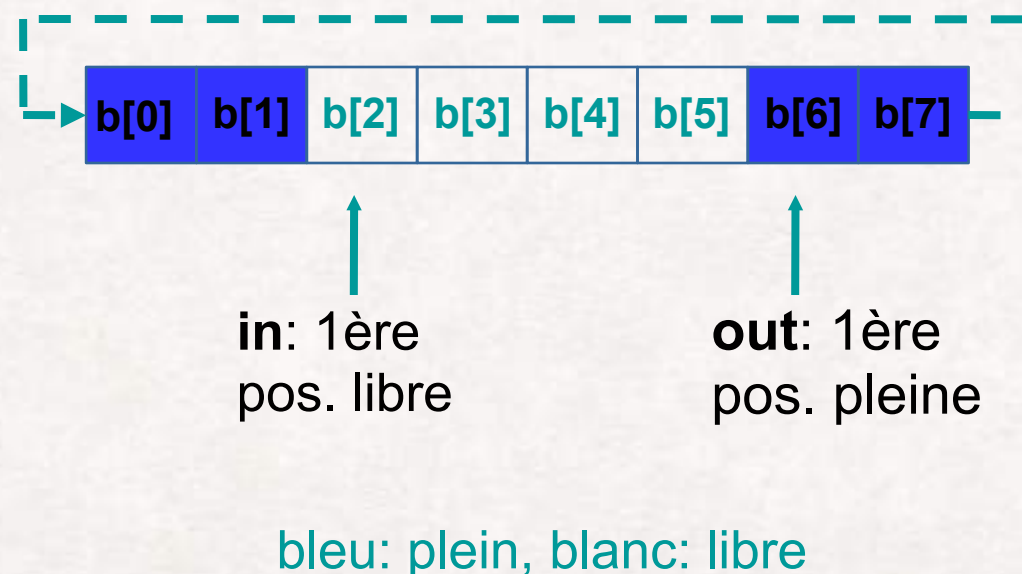
- ♦ Une solution = faire un graphe indiquant la précedence entre sémaphores dans le code : éviter les cycles

CAS CLASSIQUES D'APPLICATION DES SÉMAPHORES

- ♦ un thread producteur produit des données (p.ex. des enregistrements d'un fichier) pour un thread consommateur. Exemple : un capteur GPS et un thread écoutant les infos GPS.
- ♦ Cette variante existe aussi avec plusieurs producteurs et/ou plusieurs consommateurs
- ♦ Ecrivains / Lecteurs
- ♦ Gestion d'un carrefour entre routes

PRODUCER / CONSUMER WITH SEMAPHORES

- Un ou plusieurs producteurs génèrent un même type de ressources qui sont stockées dans un buffer borné (géré par un tableau rempli de façon cyclique)
- Un ou plusieurs consommateurs consomment les ressources du buffer
- Le tampon borné se trouve dans la mémoire partagée entre consommateur et usager



PRODUCER / CONSUMER WITH SEMAPHORES

- Synchronisation à l'aide de 3 semaphores: « **full** », « **empty** » and « **mutex** »
- **Full** counts the nb of full slots in the buffer (initially 0)
- **Empty** counts the nb of empty slots (initially N)
- **Mutex** protects the buffer which contains the produced and consumed items (binary semaphore)

Rules to be applied:

1. A single consumer enters its critical section. Since fullCount is 0, the consumer blocks.
2. The producers, one at a time, gain access to the queue through mutex and deposit items in the queue.
3. Once the first producer exits its critical section, fullCount is incremented, allowing one consumer to enter its critical section.

PRODUCER / CONSUMER WITH SEMAPHORES

- Exercice : écrivez le code des producteurs et celui des consommateurs. Indiquez la section critique de chacun

WHEN TO USE WHAT

- **Semaphore:** when you (thread) want to sleep till some other thread tells you to wake up. Semaphore 'down' happens in one thread (producer) and semaphore 'up' (for same semaphore) happens in another thread (consumer) e.g.: In producer-consumer problem, producer wants to sleep till at least one buffer slot is empty - only the consumer thread can tell when a buffer slot is empty.
- **Mutex:** when you (thread) want to execute code that should not be executed by any other thread at the same time. Mutex 'down' happens in one thread and mutex 'up' *must* happen in the same thread later.

CONCEPTS IMPORTANTS DE CE CHAPITRE

- ♦ Le problème de la section critique
- ♦ L'entrelacement et l'atomicité
- ♦ Problèmes de famine et interblocage
- ♦ Solutions logiciel
- ♦ Instructions matériel
- ♦ Sémaphores
- ♦ Fonctionnement des différentes solutions
- ♦ L'exemple du tampon borné

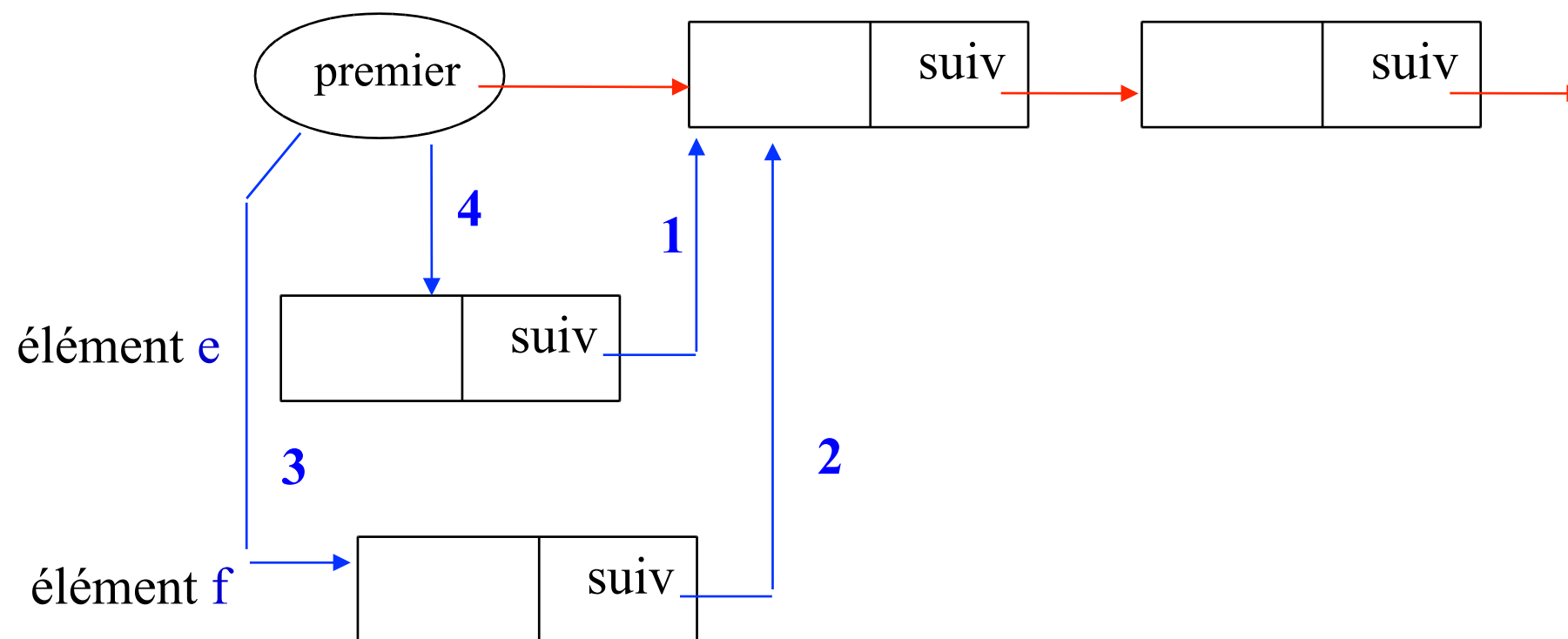
ANNEXES

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 <u>process(es)</u> without doing any useful work.
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.

SOLUTION DE L'EXERCICE : ENTRELACEMENT CATASTROPHE

Exemple 2 : liste simplement chaînée avec insertion en tête



1. e.suiv:= premier ;
2. f.suiv:=premier ;
4. premier:=e ;
3. premier:=f ;

L'élément **f** est perdu !

Comment éviter cette situation ?

Algorithme 3 (Peterson) : preuve de validité

- **Exclusion mutuelle est assurée car:**
 - ◆ T0 et T1 sont tous deux dans SC seulement si `tour` est simultanément égal à 0 et 1 (impossible)
- **Démontrons que progrès et attente limitée sont satisfaits:**
 - ◆ Ti ne peut pas entrer dans SC seulement si en attente dans la boucle `while()` avec condition: `flag[j] == vrai et tour = j.`
 - ◆ Si Tj ne veut pas entrer dans SC alors `flag[j] = faux` et Ti peut alors entrer dans SC

Algorithme 3: preuve de validité (cont.)

- ◆ Si T_j a effectué $\text{flag}[j]=\text{vrai}$ et se trouve dans le `while()`, alors $\text{tour}==i$ ou $\text{tour}==j$
- ◆ Si
 - ✦ $\text{tour}==i$, alors T_i entre dans SC.
 - ✦ $\text{tour}==j$ alors T_j entre dans SC mais il fera $\text{flag}[j]=\text{false}$ à la sortie: permettant à T_i d'entrer CS
- ◆ mais si T_j a le temps de faire $\text{flag}[j]=\text{true}$, il devra aussi faire $\text{tour}=i$
- ◆ Puisque T_i ne peut modifier tour lorsque dans le `while()`, T_i entrera SC après au plus une entrée dans SC par T_j (attente limitée)

CODAGE EN C

Threads Noyau / Threads Utilisateur

- **Bibliothèque Pthreads:**

- les threads définies par la norme **POSIX 1.c** sont indépendantes de leur implémentation.

- **Deux types d'implémentation :**

- **Thread usager (pas connue du noyau):**

- L'état est maintenu en espace utilisateur. Aucune ressource du noyau n'est allouée à une thread.
- Des opérations peuvent être réalisées indépendamment du système.
- Le noyau ne voit qu'une seule thread
 - Tout appel système bloquant une thread aura pour effet de bloquer son processus et par conséquent toutes les autres threads du même processus.

- **Thread Noyau (connue du noyau):**

- Les threads sont des entités du système (threads natives).
- Le système possède un descripteur pour chaque thread.
- Permet l'utilisation des différents processeurs dans le cas des machines multiprocesseurs.

Threads Noyau x Threads Utilisateur

Approche	Thread noyau	Thread utilisateur
Implémentation des fonctionnalités POSIX	Nécessite des appels systèmes spécifiques.	Portable sans modification du noyau.
Création d'une thread	Nécessite un appel système (ex. <i>clone</i>).	Pas d'appel système. Moins coûteuse en ressources.
Commutation entre deux threads	Faite par le noyau avec changement de contexte.	Assurée par la bibliothèque; plus légère.
Ordonnancement des threads	Une thread dispose de la CPU comme les autres processus.	CPU limitée au processus qui contient les threads.
Priorités des tâches	Chaque thread peut s'exécuter avec une prio. indépendante.	Priorité égale à celle du processus.
Parallélisme	Répartition des threads entre différents processeurs.	Threads doivent s'exécuter sur le même processeur.

Pthreads utilisant des threads Noyau

- **Trois différentes approches:**
 - **M-1 (many to one)**
 - Une même thread système est associée à toutes les *Pthreads* d'un processus.
 - Ordonnancement des threads est fait par le processus
 - Approche thread utilisateur.
 - **1-1 (one to one)**
 - A chaque *Pthread* correspond une thread noyau.
 - Les *Pthreads* sont traitées individuellement par le système.
 - **M-M (many to many)**
 - différentes *Pthreads* sont multiplexées sur un nombre inférieur ou égal de threads noyau.

Réentrance

- **Exécution de plusieurs activités concurrentes**
 - Une même fonction peut être appelée simultanément par plusieurs threads.
- **Fonction réentrante:**
 - fonction qui accepte un tel comportement.
 - pas de manipulation de variable globale
 - utilisation de mécanismes de synchronisation permettant de régler les conflits provoqués par des accès concurrents.
- **Terminologie**
 - Fonction **multithread-safe (MT-safe)** :
 - réentrant vis-à-vis du parallélisme
 - Fonction **async-safe** :
 - réentrant vis-à-vis des signaux

POSIX thread API

- **Orienté objet:**

- *pthread_t* : identifiant d'une *thread*
- *pthread_attr_t* : attribut d'une *thread*
- *pthread_mutex_t* : *mutex* (exclusion mutuelle)
- *pthread_mutexattr_t* : attribut d'un *mutex*
- *pthread_cond_t* : variable de condition
- *pthread_condattr_t* : attribut d'une variable de condition
- *pthread_key_t* : clé pour accès à une donnée globale réservée
- *pthread_once_t* : initialisation unique

POSIX thread API

- Une Pthread est identifiée par un *ID* unique
- En général, en cas de succès une fonction renvoie 0 et une valeur différente de NULL en cas d'échec.
- Pthreads n'indiquent pas l'erreur dans *errno*.
 - Possibilité d'utiliser *strerror*.
- Fichier *<pthread.h>*
 - Constantes et prototypes des fonctions.
- Faire le lien avec la bibliothèque *libpthread.a*
 - gcc -l pthread
- Directive
 - #define _REENTRANT
 - gcc ... -D _REENTRANT

Fonctions Pthreads

- **Préfixe**

- Enlever le *_t* du type de l'objet auquel la fonction s'applique.

- **Suffixe (exemples)**

- *_init* : initialiser un objet.
- *_destroy* : détruire un objet.
- *_create* : créer un objet.
- *_getattr* : obtenir l'attribut *attr* des attributs d'un objet.
- *_setattr* : modifier l'attribut *attr* des attributs d'un objet.

- **Exemples :**

- *pthread_create* : crée une thread (objet *pthread_t*).
- *pthread_mutex_init* : initialise un objet du type *pthread_mutex_t*.

Gestion des Threads

- **Une *Pthread* :**
 - est identifiée par un *ID* unique.
 - exécute une fonction passée en paramètre lors de sa création.
 - possède des attributs.
 - peut se terminer (*pthread_exit*) ou être annulée par une autre thread (*pthread_cancel*).
 - peut attendre la fin d'une autre thread (*pthread_join*).
- **Une *Pthread* possède son propre masque de signaux et signaux pendants.**
- **La création d'un processus donne lieu à la création de la thread main.**
 - Retour de la fonction *main* entraîne la terminaison du processus et par conséquent de toutes les threads de celui-ci.

Using the pthread library for Mutex

Posix thread, available in C and other languages.

Implemented code for mutex lock and unlock:

mutex_lock:

TSL REGISTER,MUTEX

CMP REGISTER,#0

JZE ok

CALL thread_yield

JMP mutex_lock

ok: RET

| copy mutex to register and set mutex to 1

| was mutex zero?

| if it was zero, mutex was unlocked, so return

| mutex is busy; schedule another thread

| try again

| return to caller; critical region entered

mutex_unlock:

MOVE MUTEX,#0

RET

| store a 0 in mutex

| return to caller

Pthread calls for *mutex*

Thread call	Description
<code>pthread_mutex_init</code>	Create a mutex
<code>pthread_mutex_destroy</code>	Destroy an existing mutex
<code>pthread_mutex_lock</code>	Acquire a lock or block
<code>pthread_mutex_trylock</code>	Acquire a lock or fail
<code>pthread_mutex_unlock</code>	Release a lock

- `pthread_mutex_trylock` tries to lock mutex. If it fails it returns an error code, and can do something else.

Pthread calls for Condition Variable

Thread call	Description
Pthread_cond_init	Create a condition variable
Pthread_cond_destroy	Destroy a condition variable
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake it up
Pthread_cond_broadcast	Signal multiple threads and wake all of them

EXERCICE : PRODUCER CONSUMER WITH CONDITION VARIABLES AND MUTEXES

- Producer produces one item and blocks waiting for consumer to consume the item.
- Producer signals consumer that the item has been produced.
- Consumer has been blocked and waiting for signal from producer that item is in buffer
- Consumer consumes item, signals producer to produce new item.

LIVRES DE RÉFÉRENCE

UNE PETITE LISTE

- Un grand classique : *Applied Operating System Concepts*, Silberchatz et al

REMERCIEMENTS

MERCI POUR LEURS PRÉSENTATIONS À

- M. Cart (Polytech Montpellier)
- L. Logrippo (Univ. Québec, Ottawa)
- P. Roy (Venice, Floride, USA)
- H. Kothari (Inde)
- Wikipedia



AUTRES

SCÉNARIO POUR LE CHANGEMENT DE CONTRÔLE

Thread T0:

```
...  
SC  
flag[0] = faux;  
// T0 ne veut  
// plus entrer  
SR  
...
```

Thread T1:

```
...  
flag[1] = vrai;  
// T1 veut entrer  
tour = 0;  
// T1 donne une chance à T0  
while  
  (flag[0]==vrai&&tour=0) {};  
  //test faux, entre  
...
```

T1 prend la relève, donne une chance à T0 mais T0 a dit qu'il ne veut pas entrer.

T1 entre donc dans la SC

AUTRE SCÉNARIO DE CHANGT. DE CONTRÔLE

Thread T0:

```
    SC
flag[0] = faux;
// T0 ne veut plus entrer
    SR
flag[0] = vrai;
// T0 veut entrer
tour = 1;
// T0 donne une chance à T1
while
( flag[1]==vrai&&tour=1) {} ;
// test vrai, n'entre pas
```

Thread T1:

```
flag[1] = vrai;
// T1 veut entrer
tour = 0;
// T1 donne une chance à T0
// mais T0 annule cette action
while
(flag[0]==vrai&&tour=0) {} ;
//test faux, entre
```

T0 veut rentrer mais est obligé de donner une chance à T1,
qui entre

MAIS AVEC UN PETIT DÉCALAGE, C'EST ENCORE T0!

Thread T0:

SC

```
flag[0] = faux;
```

```
// 0 ne veut plus entrer
```

RS

```
flag[0] = vrai;
```

```
// 0 veut entrer
```

```
tour = 1;
```

```
// 0 donne une chance à 1
```

```
// mais T1 annule cette action
```

while

```
(flag[1]==vrai&&tour=1) {};
```

```
// test faux, entre
```

Thread T1:

```
flag[1] = vrai;
```

```
// 1 veut entrer
```

```
tour = 0;
```

```
// 1 donne une chance à 0
```

while

```
(flag[0]==vrai&&tour=0) {};
```

```
// test vrai, n'entre pas
```

Si T0 et T1 tentent simultanément d'entrer dans SC, seule une valeur pour `tour` survivra : non-déterminisme (on ne sait pas qui gagnera), mais l'exclusion fonctionne