Fondamentaux des Applications Réparties (FAR)

Chouki Tibermacine
Chouki.Tibermacine@umontpellier.fr

POLYTECH®
MONTPELLIER

Plan de l'ECUE

- Partie système :
  1. Threads et problèmes de communication inter-processus ou -threads
  2. Communication par tubes et signaux
- Partie réseaux :
  1. Modèle OSI et couches réseaux basses
  2. Protocole de la couche réseau (IP)
  3. Protocoles de de la couche transport (TCP & UDP)
  4. Programmation réseau en langage C (sockets)
  5. Programmation par RPC (*Remote Procedure Call*)
  6. Programmation par Websockets (sockets sur HTTP)
  7. Programmation par objets distribués (Java RMI)

Plan de l'ECUE

- Partie système :
  1. Threads et problèmes de communication inter-processus ou -threads
  2. Communication par tubes et signaux
- Partie réseaux :
  1. Modèle OSI et couches réseaux basses
  2. Protocole de la couche réseau (IP)
  3. Protocoles de de la couche transport (TCP & UDP)
  4. Programmation réseau en langage C (sockets)
  5. Programmation par RPC (*Remote Procedure Call*)
  6. Programmation par Websockets (sockets sur HTTP)
  7. Programmation par objets distribués (Java RMI)

Intervenants et modalités d'évaluation

Intervenants

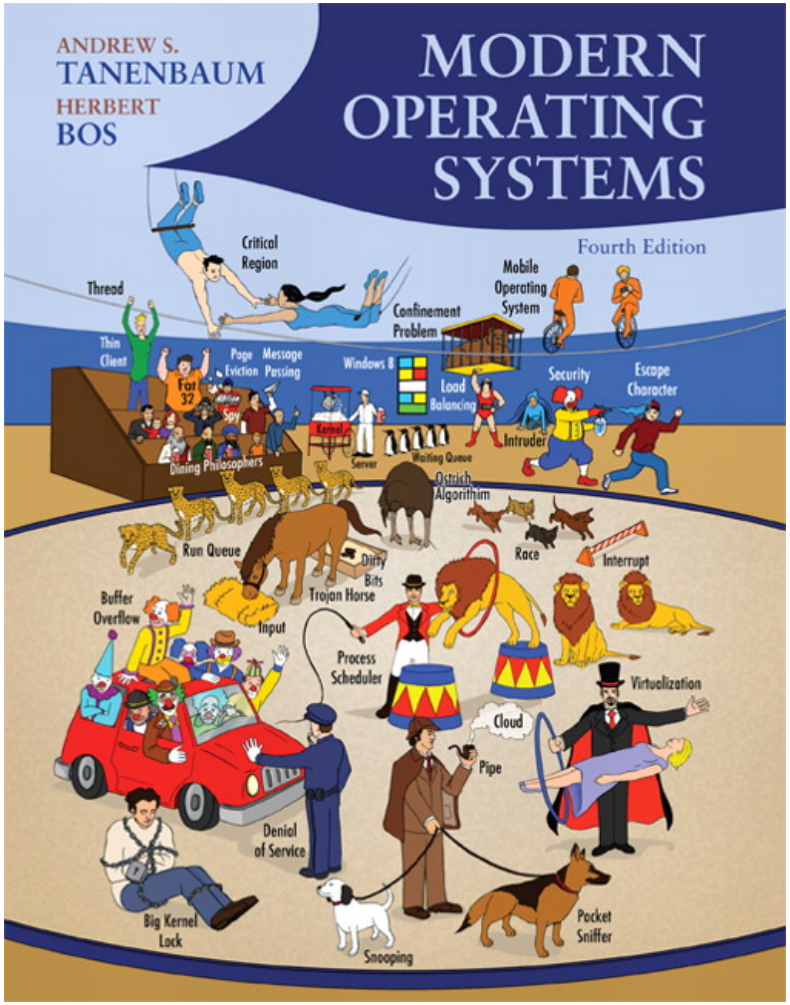
- Maxime MIRKA (doctorant/MCE) : TP de la partie système + partie réseaux + projets
- Hinde BOUZIANE (MCF FdS) : Cours et TP sur TCP/UDP et sockets

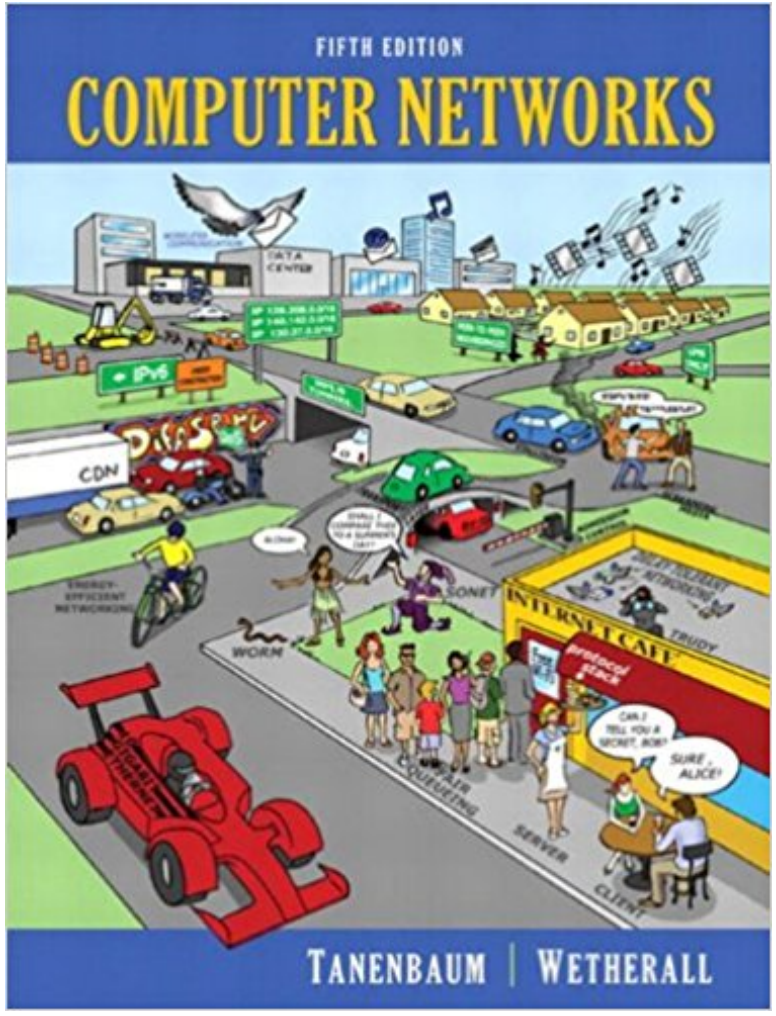
Évaluation

Moyenne avec le même coef :

- Contrôle continu (notes des TP rendus sur Moodle)
- Note d'examen final (séance programmée le 27.05 à 14h)
- Note de projets (une application répartie à développer du 03.04 au 28.05)

Références bibliographiques





Plan du cours

Plan du cours

Rappels sur les processus

- Un processus est une abstraction d'un programme en cours d'exécution
- Pour chaque processus, le système d'exploitation (l'OS) maintient les valeurs du compteur ordinal, des registres, des variables en mémoire centrale, la priorité, le PID, ...
- Le processeur bascule d'un processus à l'autre (pseudo-parallélisme ou multi-programmation) pour optimiser l'exécution des programmes (dans l'attente d'une E/S effectuée par un processus, il suspend l'exécution de ce processus et exécute un autre)

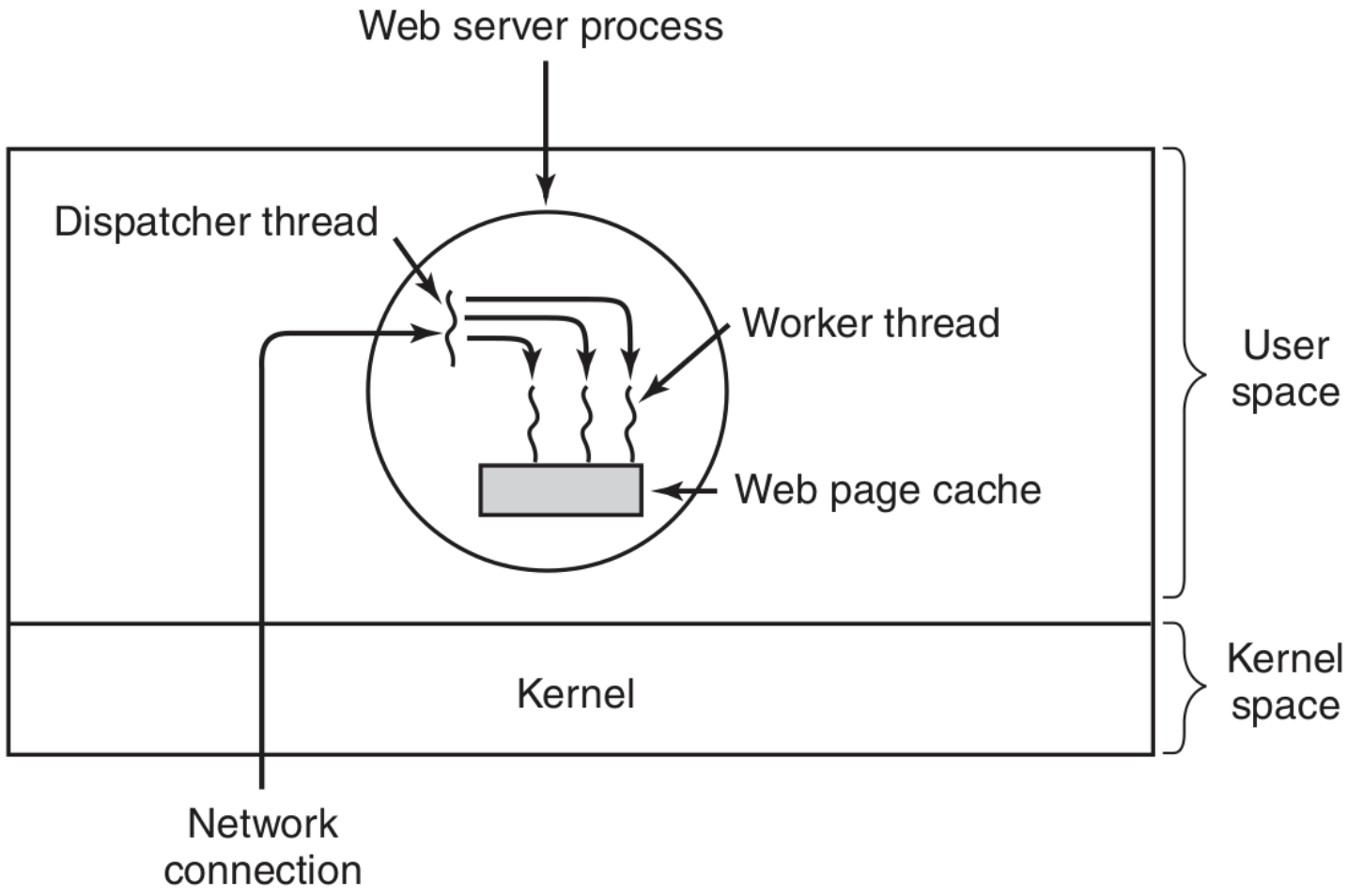
Besoins de threads

- Dans les OS traditionnels, chaque processus possède un espace d'adressage et un thread de contrôle unique
- Il arrive parfois qu'on ait besoin de plusieurs threads de contrôle, s'exécutant quasiment en parallèle, dans le même espace d'adressage
- Même chose que des processus parallèles (créés avec des `fork()`), mais qui **partagent le même espace d'adressage** (variables globales communes)

Pourquoi du “multi-threading” ?

1. Dans une même application, on peut disposer de plusieurs tâches qui peuvent s’exécuter en parallèle, et certaines peuvent se bloquer de temps en temps
2. Les threads n’ont pas de ressources propres (ont les mêmes que leur processus), ils sont donc plus faciles à créer et à détruire (que les processus)
3. Performance : accélérer le fonctionnement d’une application, si E/S et traitements lourds sont réalisés dans des threads, qui s’exécutent séparément
4. C’est adapté aux systèmes multi-processeurs (-coeurs), sur lesquels on peut avoir du vrai parallélisme (+ieurs threads s’exécutent en //)

Exemple de serveur Web multi-thread



- 1 thread `Dispatcher` (qui n'est pas bloqué par l'accès au contenu Web à retourner) et un vivier de threads *Workers*

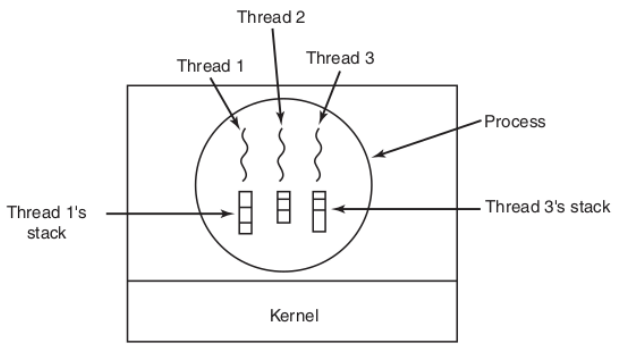
Le modèle de threads classique

- Les processus servent à regrouper les ressources
- Les threads sont les entités planifiées pour s'exécuter sur un processeur
- Les threads d'un même processus partagent un espace d'adressage, les fichiers ouverts et les autres ressources
- Les threads ont certaines propriétés comme les processus = processus légers (*lightweight processes*)
- Mêmes états que pour les processus : *running*, *blocked*, *ready*, ou *terminated*

Éléments spécifiques aux processus et aux threads

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Les éléments spécifiques aux processus sont partagés par tous les threads du même processus
- Chaque thread possède sa propre "*pile*"
  = { bloc d'activation (frame) par procédure appelée }



- Chaque frame contient les variables locales de la procédure et l'adresse de retour

Création et manipulation de threads

- Dans la plupart des cas, les processus démarrent avec un seul thread
- Ce thread est capable de créer de nouveaux thread (procédure de bibliothèque : `create_thread`)
- Généralement, l'un des paramètres de cette procédure est le nom d'une procédure à exécuter par le nouveau thread
- Cette procédure retourne un identifiant de thread
- A la fin de son exécution, il peut s'arrêter (`thread_exit`)
- Un thread peut attendre la fin de l'exécution d'un autre thread (`thread_wait`)
- Il est important que les threads se montrent "polis" en laissant du temps processeur aux autres threads (`thread_yield`)

Les threads de POSIX : norme IEEE

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

- `pthread_join` $\Leftrightarrow$ `thread_wait`

Les threads spontanés (*Popup Threads*)

- Très utilisés dans les applications qui gèrent des messages entrants
- A la réception d'un message, un nouveau thread (spontané/*popup*) est créé pour gérer le message
- Avantage : une latence très courte entre l'arrivée d'un message et le début du traitement (parce que création de threads rapide)

Plan du cours

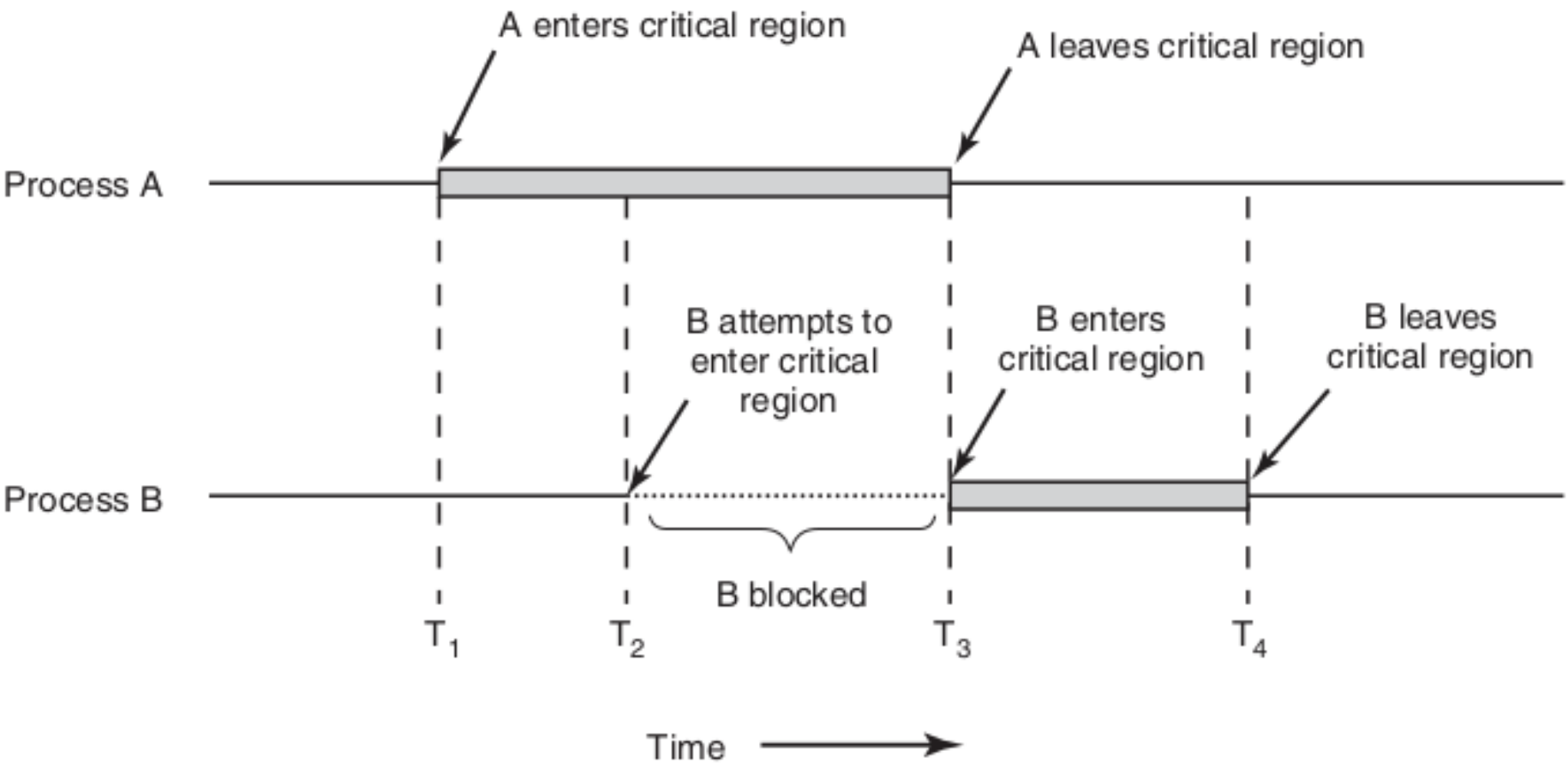
Problèmes liés à la communication

- Entre threads d'un même processus, la communication est facile à réaliser : ils partagent le même espace d'adressage
- Communication inter-threads de différents processus = communication inter-processus (tubes et signaux ⇒ prochain cours)
- On va s'intéresser aux problèmes liés à la communication :
  1. **Exclusion mutuelle** : des processus concurrents peuvent produire des conflits en s'engageant dans des activités critiques (deux processus de réservation aérienne tentant de récupérer pour deux clients différents le dernier siège disponible)
  2. **Synchronisation** : séquence d'exécution en présence de dépendances (un processus produit des données utilisées par un autre processus : gestion de l'attente)

Section critique (SC)

- Une partie d'un programme accédant à une ressource (ou une donnée en mémoire) partagée et dont l'exécution ne doit pas être gérée par plus d'un thread à la fois
- Une fois qu'un thread y entre, il faut lui permettre de terminer cette section sans permettre à d'autres threads de jouer sur les mêmes ressources/données
- Exemple : la procédure qui : 1) vérifie la disponibilité de sièges dans un vol et 2) réserve un siège si disponibilité

Illustration d'une section critique



Conditions de concurrence (*race conditions*)

1. **Exclusion Mutuelle :** à tout instant, au plus un thread peut être dans une SC pour une ressource/variable donnée
2. **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 est en dehors de sa section critique, il ne doit pas affecter/bloquer les autres threads
3. **Attente limitée (*bounded waiting*) :** aucun thread n'est éternellement empêché d'atteindre sa SC

Plan du cours

Types de solutions

1. Solutions par le 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 spéciales (du processeur)

- 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 (par un thread à la fois)
- Toutes les solutions se basent sur l'existence d'instructions atomiques, qui fonctionnent comme une SC de base

1. Solutions par le logiciel

- Nous considérons d'abord 2 threads et ensuite on généralisera le problème
- Algorithmes 1 et 2 ne sont pas valides mais montrent la difficulté du problème
- Algorithme 3 est valide (Algorithme de Peterson)

Variables de verrou (*lock*)

- Une variable unique partagée : un verrou avec une valeur initiale = 0
- Lorsqu'un processus tente d'entrer dans sa section critique, il teste le verrou
- Si verrou = 0, il est positionné à 1 et ensuite le processus entre dans sa SC
- Si le verrou = 1, le processus attend qu'il passe à 0
- Verrou = 0 $\Rightarrow$ aucun processus n'est en SC
- Inconvénient : exclusion mutuelle ne peut être assurée, car entre le test du verrou et sa màj, il se peut que l'OS bloque le processus qui a testé le verrou et exécute le second processus

Alternance stricte

- Une variable partagée `turn` joue le rôle de verrou
- Une boucle d'attente active pour tester la valeur d'un verrou (*spin lock*)
- Solution consommatrice en temps processeur (à n'utiliser que lorsque l'on prévoit une attente brève)

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

Alternance stricte -suite-

- Inconvénient : Si processus 0 sort de la section critique et rentre dans une **longue section non critique**, et si le processus 1 passe après et termine vite une itération de la boucle, il reste bloqué (violation de la condition de “Non-interférence”)
- Il est impossible qu’un processus entre deux fois de suite dans la section critique (d’où le nom “alternance stricte”)

Solution de Peterson (vérifie toutes les conditions)

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Critique des solutions par le logiciel

Difficiles à programmer et à comprendre !

- Les solutions que nous verrons désormais sont toutes basées sur l'existence d'instructions spécialisées, qui facilitent le travail

Solutions par le matériel

1. Suspension des interruptions
2. Instructions processeurs dédiés TSL (*Test and Set Lock*) et XCHG (*Exchange*)

Suspension des interruptions

- Idée : un processus désactive les interruptions dès qu'il rentre dans sa SC et les réactive après l'avoir quittée
- Pas d'interruptions, impossible pour le processeur de basculer d'un processus à un autre
- Problèmes :
  - Donner un pouvoir important aux processus utilisateurs (s'ils n'arrivent pas à réactiver les interruptions ? ! ! !)
  - Sur un mono-processeur, l'efficacité se détériore
  - Sur un multi-processeur, l'exclusion mutuelle n'est pas préservée

Instructions dédiées (composées mais atomiques)

- L'instruction TSL (*Test and Set Lock*) :

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was not zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Verrouillage matériel du bus mémoire pendant l'exec de TSL

- L'instruction de permutation XCHG (processeurs Intel x86) :

```
enter_region:
        MOVE REGISTER,#1           | put a 1 in the register
        XCHG REGISTER,LOCK         | swap the contents of the register and lock variable
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Critique des solutions par logiciel/matériel

Les threads qui requièrent l'entrée dans leur SC sont occupés à attendre (attente active) ; consommant ainsi du temps de processeur

- Pour de longues sections critiques, il serait préférable de bloquer les threads qui doivent attendre

$\Rightarrow$ Utiliser des mécanismes de synchronisation de processus

Plan du cours

Sommeil et activation

- Appel système `sleep` : provoque le blocage de l'appelant jusqu'à ce qu'un autre processus le réveille
- Appel système `wakeup` : provoque le réveil d'un processus indiqué en paramètre
- Procédures permettant de synchroniser des processus, même en dehors de la gestion des situations d'exclusion mutuelle

Problème du producteur-consommateur

- Problème connu également sous le nom de *bounded buffer*
- Deux processus partagent un tampon commun de taille fixe
- Processus 1 = producteur : place des données dans le tampon
- Processus 2 = consommateur : récupère les données du tampon
- Problème : producteur veut placer une donnée dans le tampon qui est plein ⟶ doit entrer en sommeil et attendre d'être réveillé par le consommateur
- Même problème pour le consommateur quand tampon vide
- Problème de concurrence : ressource critique (taille tampon)

Problème du producteur-consommateur -suite-

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                item = produce_item( );         /* generate next item */
                if (count == N) sleep( );       /* if buffer is full, go to sleep */
                insert_item(item);              /* put item in buffer */
                count = count + 1;              /* increment count of items in buffer */
                if (count == 1) wakeup(consumer);   /* was buffer empty? */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* repeat forever */
                if (count == 0) sleep( );       /* if buffer is empty, got to sleep */
                item = remove_item( );          /* take item out of buffer */
                count = count – 1;              /* decrement count of items in buffer */
                if (count == N – 1) wakeup(producer);   /* was buffer full? */
                consume_item(item);             /* print item */
        }
}
```

Problème de concurrence (scénario catastrophe)

1. Le consommateur commence le premier
2. Il voit que le buffer est vide (le test de count = 0 renvoie vrai)
3. L'ordonnanceur choisit à ce moment d'exécuter le producteur
4. Le producteur insère un élément dans le buffer vide et incrémente count (count = 1 désormais)
5. Il essaie de réveiller le consommateur, qui n'est pas encore entré en sommeil (**signal wakeup perdu** : problème principal)
6. L'ordonnanceur reprend l'exécution du consommateur
7. Ce dernier entre en sommeil parce qu'il avait testé count et c'était égal à 0
8. Le producteur continue à produire des éléments dans le buffer jusqu'à son remplissage complet
9. Il entre ensuite en sommeil
10. Les deux processus dorment pour toujours

Les sémaphores – E. W. Dijkstra (1965)

- Idée de base : sauvegarder les signaux wakeups destinés aux processus qui ne dorment pas
- **Sémaphore :** une variable compteur de *wakeups* munie de deux opérations → *up* (ou V) et *down* (ou P), des généralisations de *wakeup* et *sleep*
- Opération **down** : teste la valeur du sémaphore si égal à 0, sinon, elle décrémente la valeur et ne fait rien, sinon, elle met le processus en sommeil (action atomique/indivisible)
- Opération **up** : incrémente la valeur et vérifie s'il y a des processus qui dorment, si oui elle en réveille un pour compléter son *down* (opération atomique)

Résolution du pbm prod-conso (3 sémaphores)

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
        int item;

        while (TRUE) {                          /* TRUE is the constant 1 */
                item = produce_item( );         /* generate something to put in buffer */
                down(&empty);                   /* decrement empty count */
                down(&mutex);                   /* enter critical region */
                insert_item(item);              /* put new item in buffer */
                up(&mutex);                     /* leave critical region */
                up(&full);                      /* increment count of full slots */
        }
}


void consumer(void)
{
        int item;

        while (TRUE) {                          /* infinite loop */
                down(&full);                    /* decrement full count */
                down(&mutex);                   /* enter critical region */
                item = remove_item( );          /* take item from buffer */
                up(&mutex);                     /* leave critical region */
                up(&empty);                     /* increment count of empty slots */
                consume_item(item);             /* do something with the item */
        }
}
```

Deux utilisations des sémaphores dans l'exemple

- Sémaphore binaire (`mutex`) pour gérer l'**exclusion mutuelle**
  Il suffit :
  1. d'initialiser le sémaphore à 1
  2. d'appeler `down` sur le sémaphore avant d'entrer dans la section critique
  3. et d'appeler `up` après l'avoir quittée
- Les 2 sémaphores (`full` et `empty`) pour gérer la **synchronisation**
  Ils garantissent que :
  1. le producteur est bloqué quand le tampon est plein
  2. et le consommateur est bloqué quand le tampon est vide

Les mutex

- Mutex : sémaphore utilisé pour l'exclusion mutuelle
- Utilisé quand il n'y a pas de besoin de comptage
- Facile à utiliser dans les threads et efficace
- Deux valeurs (int) : 0 pour déverrouillé et 1 pour verrouillé
- Deux opérations : `mutex_lock` et `mutex_unlock`
- Si plusieurs threads sont bloqués, le choix du thread à débloquer est aléatoire

Implémentation des opérations sur les mutex

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

- Opération `mutex_trylock` permet de ne pas être bloqué si le verrou ne peut pas être obtenu

Les mutex dans POSIX (bibliothèque Pthread)

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Les variables conditionnelles de Pthread

- Un autre moyen de synchronisation utilisé à la place des sémaphores
- On l'utilise souvent en combinaison avec les mutex (qui eux gèrent l'exclusion mutuelle) pour gérer le blocage (sommeil) et l'activation (par des signaux)
- Dans Pthread, on a le moyen de créer et détruire une variable conditionnelle (`pthread_cond_init` et `pthread_cond_destroy`)
- `pthread_cond_wait` bloque le thread appelant jusqu'à la réception d'un signal avec `pthread_cond_signal`
- `pthread_cond_broadcast` permet de débloquer plusieurs threads qui attendent le même signal

Problème du prod-conso avec var. cond. et mutex

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000          /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp; /* used for signaling */
int buffer = 0;                 /* buffer used between producer and consumer */

/* main function should be moved to the end of the program */
int main(int argc, char **argv) {
  pthread_t pro, con;
  pthread_mutex_init(&the_mutex, 0);
  pthread_cond_init(&condc, 0);
  pthread_cond_init(&condp, 0);
  pthread_create(&con, 0, consumer, 0);
  pthread_create(&pro, 0, producer, 0);
  pthread_join(pro, 0);
  pthread_join(con, 0);
  pthread_cond_destroy(&condc);
  pthread_cond_destroy(&condp);
  pthread_mutex_destroy(&the_mutex);
}
```

Problème du prod-conso avec var. cond. et mutex

```
void *producer(void *ptr) {                /* produce data */
  int i;
  for (i= 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
    while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
    buffer = i;                            /* put item in buffer */
    pthread_cond_signal(&condc);           /* wake up consumer */
    pthread_mutex_unlock(&the_mutex); /* release access to buffer */
  }
  pthread_exit(0);
}

void *consumer(void *ptr) {                /* consume data */
  int i;
  for (i = 1; i <= MAX; i++) {
    pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
    while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
    buffer = 0;                            /* take item out of buffer */
    pthread_cond_signal(&condp);           /* wake up producer */
    pthread_mutex_unlock(&the_mutex); /* release access to buffer */
  }
  pthread_exit(0);
}
```

Les moniteurs – Hansen (1973) et Hoare (1974)

- Sémaphores et mutex : communication inter-processus facile, mais risque d'interblocage si l'ordre des appels de procédures n'est pas respecté
- **Moniteur** : une collection de procédures, variables, et structures de données groupées ensemble dans un module ou package particulier (construction du langage de prog)
- Un seul processus peut appeler les procédures dans un moniteur (pas d'entrelacement possible)
- Exemple de procédure dans un moniteur : une méthode *synchronized* en Java
- Exclusion mutuelle garantie par le code généré par le compilateur du langage qui fournit les moniteurs
- Utilisation de variables conditionnelles pour gérer la synchronisation (en Java, méthodes `wait()` et `notify()`)

Défauts des sémaphores et moniteurs

- Utilisables avec des processus/threads s'exécutant sur un seul processeurs ou plusieurs processeurs partageant un espace d'adressage
- Dans un système réparti, où plusieurs processeurs ont chacun son propre espace d'adressage, ça ne marche pas
- Besoin d'un mécanisme d'échange d'information

Le *message passing*

- Appels système :
  - `send(destination, &message);`
  - `receive(source, &message);`
- L'appel à `receive` est bloquant jusqu'à l'arrivée d'un message (possibilité de synchroniser)
- Problème de perte de messages sur le réseau (nécessité d'envoyer des accusés de réception – **acknowledgments**) : Si pas d'accusé de réception, renvoyer le message
- Problème de perte de l'accusé de réception (réception double d'un message) : ajouter un numéro séquentiel au message
- Plus de détails, dans les cours de réseaux (problème déjà résolus et bien gérés par les OS)