



Programmation des systèmes

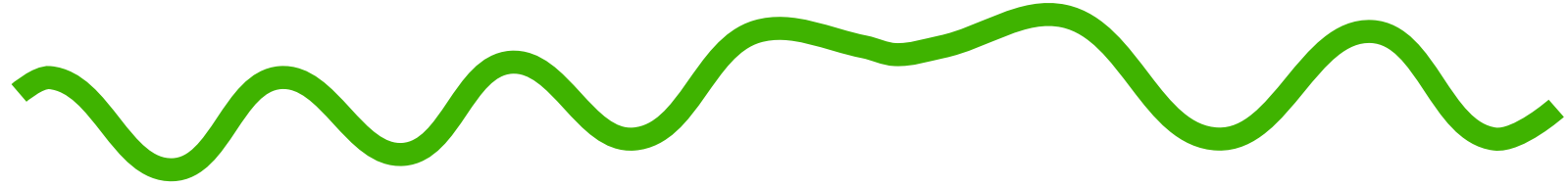
Synchronisation d'activités concurrentes

Philippe MARQUET

`Philippe.Marquet@lifl.fr`

Laboratoire d'informatique fondamentale de Lille
Université des sciences et technologies de Lille

Licence d'informatique de Lille
mars 2005



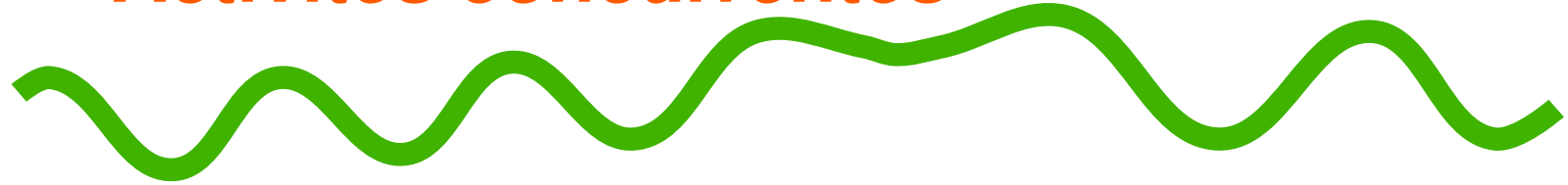
- ~ Ce cours est diffusé sous la GNU Free Documentation License,
`www.gnu.org/copyleft/fdl.html`
- ~ La dernière version de ce cours est accessible à
`www.lifl.fr/~marquet/cnl/pds/`
- ~ `$Id: sync.tex,v 1.7 2006/02/02 16:29:07 marquet Exp $`

Références & remerciements



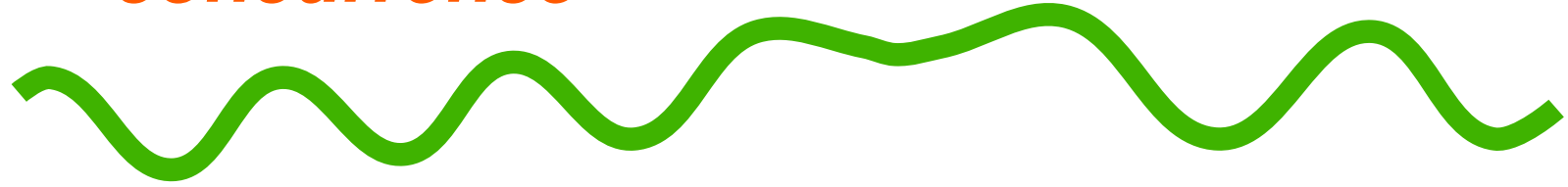
- ~ *Systemes d'exploitation*, 2e ed.
Andrew Tanenbaum
Prentice Hall, 2001, trad. française Pearson Education France, 2003
- ~ *Principes des Systemes d'Exploitation des Ordinateurs*
Sacha Krakowiak
Dunod Informatique, 1987

Activités concurrentes



- ~ Multiprogrammation : plusieurs activités *simultanées*
- ~ Activité d'exécution
 - ~ processus (*process*), tâche (*task*), processus léger (*thread*), etc
- ~ Activités sous le contrôle d'un ordonnanceur
 - ~ virtualise le processeur
 - ~ chaque activité détient le processeur tour à tour
 - ~ ordonnanceur décide du tour à tour
- ~ Raisonner sur des activités concurrentes
 - ~ aucune d'hypothèse sur l'ordre relatif des exécutions
 - ~ raisonnement vrai quelque soit l'ordonnancement
- ~ Raisonner sur des activités concurrentes
 - ~ prise en compte de l'exécution interne des activités
 - ~ prise en compte des dépendances entre les activités : synchronisation

Nécessité de contrôle de la concurrence



~ Toute action ne peut être faite concurremment

~ counter++;

```
LOAD @"COUNTER" Ri
INCR Ri
STOR Ri @"COUNTER"
```

~ deux activités, un processeur

counter++;

counter++;

LOAD @"COUNTER" R1

```
LOAD @"COUNTER" R2
INCR R2
STOR R2 @"COUNTER"
```

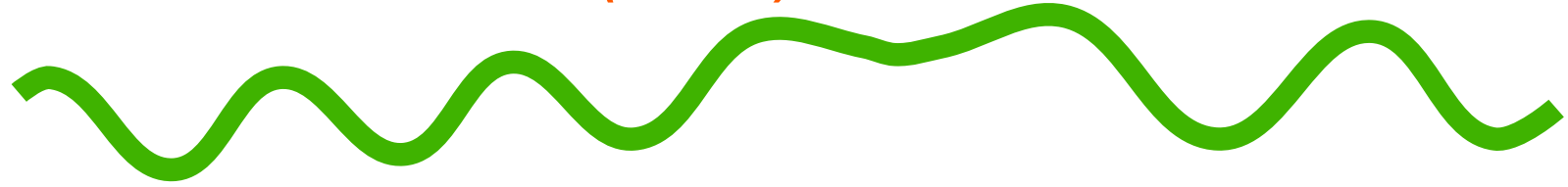
```
INCR R1
STOR R1 @"COUNTER"
```

on a perdu une incrémentation !

~ Il faut contrôler la concurrence : mécanismes ad hoc

~ (...à suivre)

Nécessité de contrôle de la concurrence (cont'd)



Exécutions possibles de nos incréments

LOAD @"COUNTER" R1
INCR R1
STOR R1 @"COUNTER"

LOAD @"COUNTER" R2
INCR R2
STOR R2 @"COUNTER"

ou

LOAD @"COUNTER" R2
INCR R2
STOR R2 @"COUNTER"

LOAD @"COUNTER" R1
INCR R1
STOR R1 @"COUNTER"

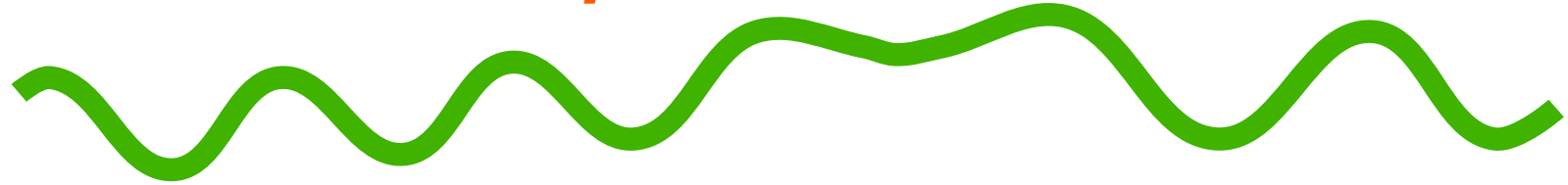
Garantir le résultat

assurer que l'ensemble des opérations

LOAD @"COUNTER" Ri
INCR Ri
STOR Ri @"COUNTER"

est exécuté de manière indivisible

Section critique



~ Section critique

- ~ section de code exécutée de manière indivisible
- ~ exécution *atomique*
- ~ exécutée en exclusion mutuelle

~ Règle de la section critique

- ~ à tout instant, une activité, au plus, peut être dans la section critique

~ Primitives d'identification de sections critiques

- ~ `enter_region` **et** `leave_region`
- ~ utilisation *systématique*

```
enter_region
    LOAD @"COUNTER" Ri
    INCR Ri
    STOR Ri @"COUNTER"
leave_region
```

- ~ garantissent l'exclusion mutuelle
- ~ doivent être exécutée de manière atomique !
- ~ plusieurs régions : `enter_region(r)` **et** `leave_region(r)`

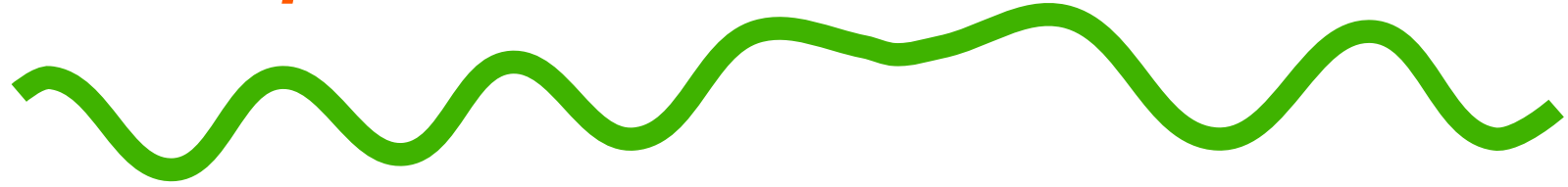
Réalisation des sections critiques



~ Attente active

- ~ l'activité boucle sur un test en entrée de section critique
- ~ inefficace !
 - ~ monopolise le processeur
 - ~ qu'une autre activité qui est en section critique ne peut utiliser...
- ~ parfois utilisée
 - ~ dans le noyau
 - ~ pour de très courtes sections critiques
 - ~ sur des systèmes multiprocesseurs
 - ~ évite le changement de contexte
 - ~ *spin-lock*

Réalisation des sections critiques (cont'd)



- ~ Fourniture par le système de primitives d'entrée et de sortie de section critique
 - ~ primitives elles-mêmes atomiques !
 - ~ exemples : verrous, sémaphores...
 - ~ implantation système repose sur des mécanismes matériels ad hoc
- ~ Réalisation bas niveau de sections critiques
 - ~ désactivation des interruptions

```
enter_region:
    cli
```

```
leave_region:
    sti
```

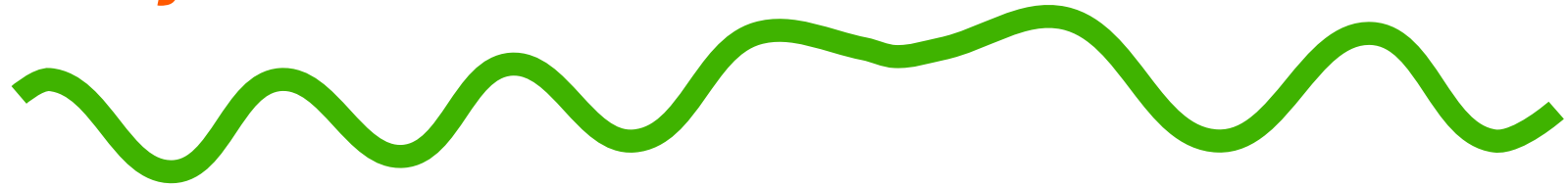
- ~ instruction *test and set*
 - ~ réalisation atomique de

```
boolean testset(int i) {
    if (i==0) {
        i = 1;
        return true;
    } else {
        return false;
    }
}
```

- ~ utilisation

```
init_region(b):
    b = 0;
enter_region(b):
    while(!testset(b))
        yield();
leave_region:
    b = 0;
```

Mécanismes classiques de synchronisation



~ Mécanismes fournis par les systèmes d'exploitation

- ~ relativement haut niveau !

~ Verrou

- ~ réalisation de l'exclusion mutuelle
- ~ aussi nommé *mutex*

~ Condition

- ~ gestion d'une file d'attente d'un état
- ~ exemple d'état : tampon non vide...

~ Sémaphore

- ~ gestion de jetons (compteur)
- ~ associé à une file d'attente
- ~ exemple : accès multiples (mais bornés) à une ressource

~ Moniteur

- ~ ensemble de procédures d'accès à des ressources partagées
- ~ regroupe les conditions (files d'attentes) et les ressources protégées

Synchronisation par verrous



- ~ Entité protégée par un verrou
 - ~ une section critique, ou
 - ~ une donnée : exemple tous les accès à un tableau
- ~ Notion de propriétaire d'un verrou
 - ~ seul le propriétaire a accès à l'entité protégée par le verrou
- ~ Primitives pour l'utilisation de verrous
 - ~ création / initialisation / destruction
 - ~ verrouillage `mutex_lock()`
 - ~ devenir propriétaire
 - ~ bloquant
 - ~ déverrouillage `mutex_unlock()`
 - ~ verrou devient libre

Synchronisation par conditions



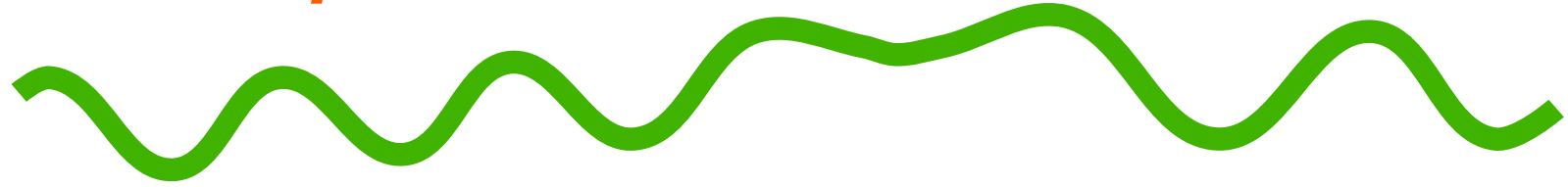
- ~ Condition est associée à un état
 - ~ file d'attente d'activités
 - ~ attendre que l'état soit réalisé : bloquer sur la condition
 - ~ changer la valeur de l'état : débloquent une activité de la file d'attente
- ~ Primitives pour l'utilisation de conditions
 - ~ création / initialisation / destruction
 - ~ attendre `cond_sleep()`
 - ~ se bloquer sur la condition
 - ~ signaler `cond_wakeup()`
 - ~ débloquent une activité en attente sur la condition
- ~ Pas de mémorisation des opérations « signaler »
 - ~ en l'absence d'activité bloquée un `cond_wakeup()` est sans effet
 - ~ les sémaphores pallient cet inconvénient

Synchronisation par sémaphores



- ~ Sémaphores = structure de données
 - ~ un compteur : « nombre de jetons libres »
 - ~ une file d'activités en attente d'un jeton
- ~ Primitives pour l'utilisation de sémaphores
 - ~ (*Speekt U nederlands?*)
 - ~ création / initialisation : nombre initial de jetons / destruction
 - ~ attendre `semaphore_down()`
 - ~ classiquement noté P (puis-je ?)
 - ~ attribue un jeton à l'activité demandeuse s'il en reste un
 - ~ la bloque sinon
 - ~ poster `semaphore_up()`
 - ~ classiquement noté V (vas-y !)
 - ~ rend un jeton
 - ~ débloque une activité s'il en existe une

Synchronisation par sémaphores (cont'd)



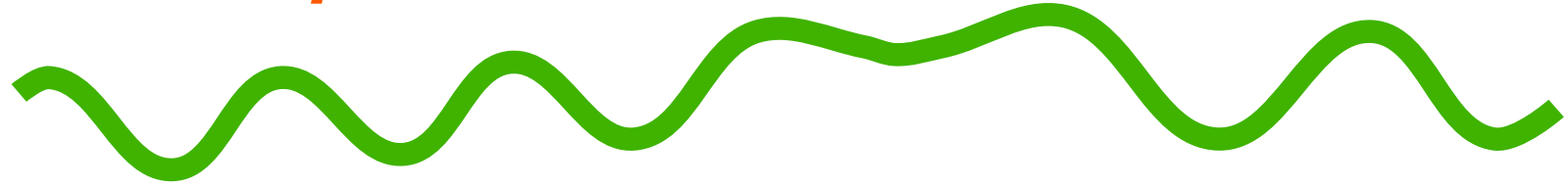
~ Compteur associé au sémaphore

- ~ nombre de « jetons »
- ~ valeur positive = nombre d'activités pouvant acquérir librement la ressource
- ~ valeur négative = nombre d'activités bloquées en attente de la ressource

~ Deux utilisations typiques des sémaphores

- ~ protection d'une ressource partagée
- ~ séquençement d'activités

Synchronisation par sémaphores (cont'd)



~ Protection d'une ressource partagée par sémaphore

- ~ ressource partagée ?
 - ~ une variable, une structure, une imprimante...
- ~ sémaphore initialisé au nombre d'activités pouvant concurremment accéder à la ressource

```
semaphore_init(s, <n>)
```

- ~ chaque accès à la ressource est encadré d'un couple

```
semaphore_down(s);
```

```
<accès à la ressource>
```

```
semaphore_up(s);
```

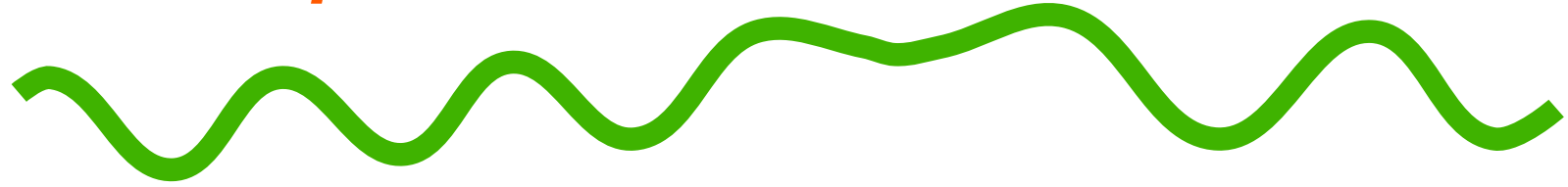
~ Exemple de l'exclusion mutuelle

- ~ une seule activité en section critique
- ~ donc un seul jeton

```
struct mutex {  
    semaphore_t mutex_sem;  
};  
  
mutex_init(struct mutex) {  
    init_semaphore(mutex.mutex_sem, 1);  
}
```

```
mutex_lock(struct mutex) {  
    semaphore_down(mutex.mutex_sem);  
}  
  
mutex_unlock(struct mutex) {  
    semaphore_up(mutex.mutex_sem);  
}
```

Synchronisation par sémaphores (cont'd)



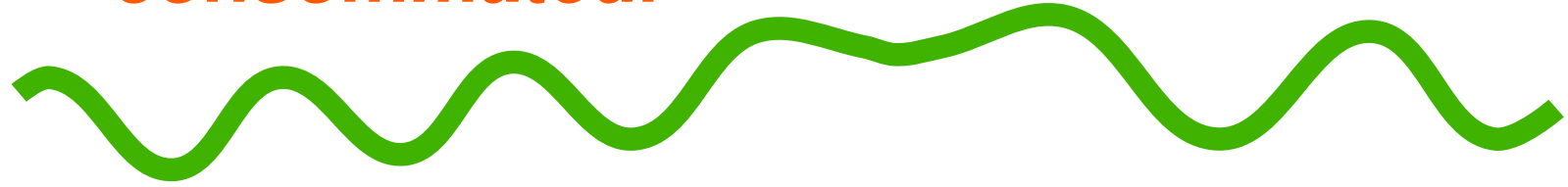
~ Séquencement d'activités par sémaphore

- ~ une activité doit en attendre une autre pour continuer ou commencer son exécution
- ~ on associe un sémaphore à l'événement
- ~ par exemple `findupremier`
- ~ initialisé à 0 ; l'événement n'a pas eu lieu

activité 1:
 <action 1>
 semaphore_up(findupremier);

activité 2:
 semaphore_down(findupremier);
 <action 2>

Problème du producteur consommateur



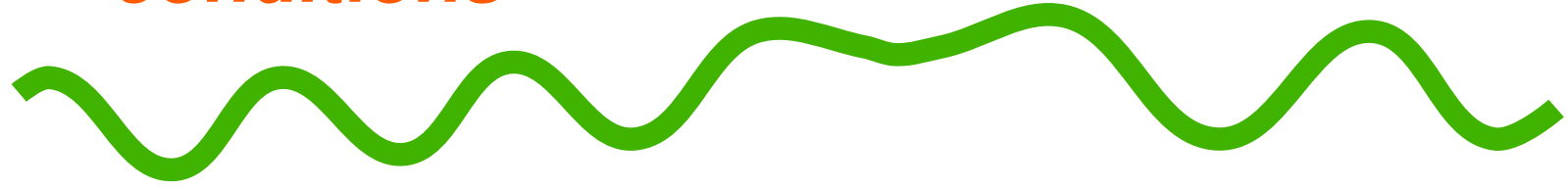
~ Deux activités

- ~ un producteur et un consommateur
- ~ relié par un tampon de taille bornée
- ~ (deux types d'activités : plusieurs producteurs / plusieurs consommateurs)
- ~ producteur ne peut pas produire si le tampon est plein
- ~ consommateur ne peut pas consommer si le tampon est vide
- ~ producteur et consommateur ne doivent pas travailler sur le même élément

~ Problème classique

- ~ mécanismes systèmes basés sur le producteur/consommateur
 - ~ tubes POSIX, *pipe* et *fifo*
 - ~ queues de messages POSIX

Producteur consommateur à base de conditions



Deux conditions

- une condition pour le producteur
- une condition pour le consommateur

```
#define N 100          /* nombre de places dans le tampon */
int count;             /* nombre d'objets dans le tampon */
cond_t c_cons, c_prod; /* les conditions */
```

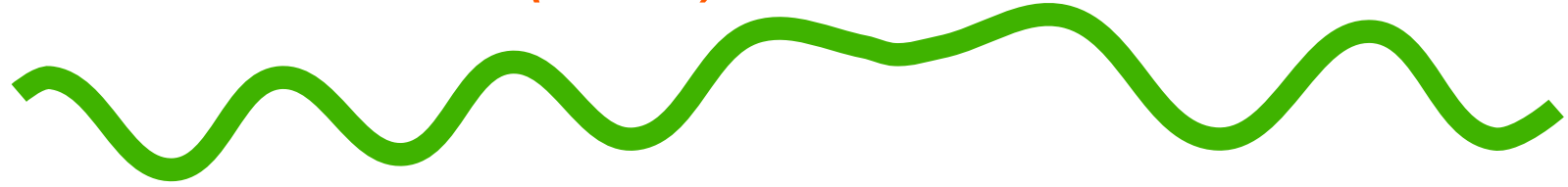
```
void producteur (void)
{
    objet_t objet;

    while (1) {
        produire_objet(&objet);
        /* tampon plein ? on attend */
        if (count==N)
            cond_sleep(c_prod);
        mettre_objet(objet);
        count++;
        /* le tampon était vide ? */
        if (count==1)
            cond_wakeup(c_cons);
    }
}
```

```
void consommateur (void)
{
    objet_t objet ;

    while (1) {
        /* tampon vide ? on attend */
        if (count==0)
            cond_sleep(c_cons);
        retirer_objet (&objet);
        count--;
        /* le tampon était plein ? */
        if (count==N-1)
            cond_wakeup(c_prod);
        utiliser_objet(objet);
    }
}
```

Producteur consommateur à base de conditions (cont'd)



~ Première erreur

- ~ pas de protection de l'accès à count
- ~ ajout d'un verrou

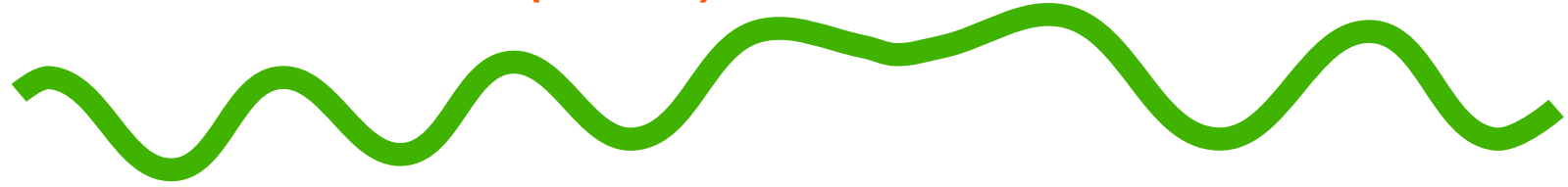
```
void producteur (void)
{
    objet_t objet;

    while (1) {
        produire_objet(&objet);
        mutex_lock();
        if (count==N)
            cond_sleep(c_prod);
        mettre_objet(objet);
        count++;
        if (count==1)
            cond_wakeup(c_cons);
        mutex_unlock();
    }
}
```

```
void consommateur (void)
{
    objet_t objet ;

    while (1) {
        mutex_lock();
        if (count==0)
            cond_sleep(c_cons);
        retirer_objet (&objet);
        count--;
        if (count==N-1){
            cond_wakeup(c_prod);
        }
        mutex_unlock();
        utiliser_objet(objet);
    }
}
```

Producteur consommateur à base de conditions (cont'd)



~ Seconde erreur

~ ne pas garder le verrou alors que l'on est bloqué !

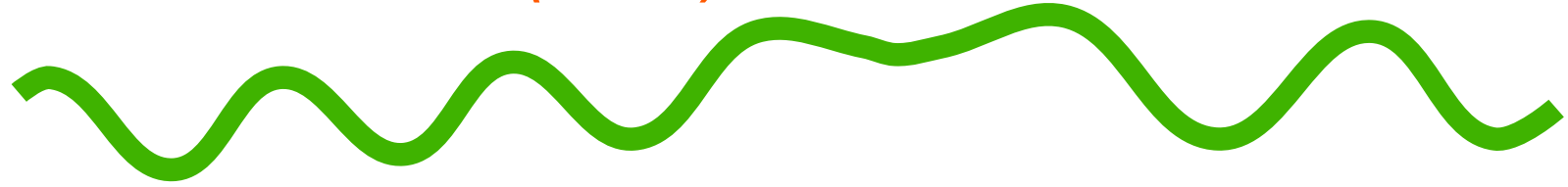
```
void producteur (void)
{
    objet_t objet;

    while (1) {
        produire_objet(&objet);
        mutex_lock();
        if (count==N) {
            mutex_unlock();
            cond_sleep(c_prod);
            mutex_lock();
        }
        mettre_objet(objet);
        count++;
        if (count==1)
            cond_wakeup(c_cons);
        mutex_unlock();
    }
}
```

```
void consommateur (void)
{
    objet_t objet ;

    while (1) {
        mutex_lock();
        if (count==0) {
            mutex_unlock();
            cond_sleep(c_cons);
            mutex_lock();
        }
        retirer_objet (&objet);
        count--;
        if (count==N-1)
            cond_wakeup(c_prod);
        mutex_unlock();
        utiliser_objet(objet);
    }
}
```

Producteur consommateur à base de conditions (cont'd)



~ Troisième erreur

- ~ condition de concurrence pouvant mener à une erreur
- ~ *race condition* en anglais !

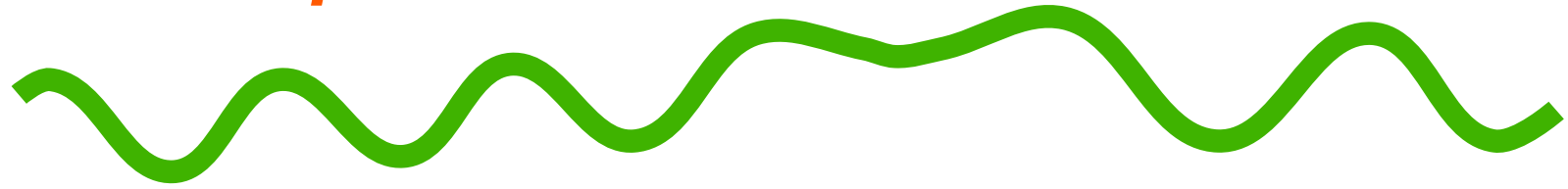
~ Scénario menant à l'erreur

- ~ le tampon est vide
- ~ le consommateur lit le compteur `count` et constate qu'il est nul
- ~ à cet instant précis le producteur insère un objet dans le tampon
- ~ il incrémente le compteur, constate qu'il est à 1
- ~ il appelle alors `cond_wakeup()` pour réveiller un éventuel consommateur...
- ~ mais le consommateur n'était pas en sommeil
- ~ le signal de réveil a été perdu
- ~ quand le consommateur va reprendre son exécution, il va se bloquer sur `cond_sleep()`

~ Solution ?

- ~ les sémaphores...

Producteur consommateur à base de sémaphores



~ Avantage sur les conditions

- ~ modification du compteur et blocage en une opération *atomique*

~ Plusieurs sémaphores

- ~ un sémaphore d'exclusion mutuelle d'accès au tampon
 - ~ initialisé à 1 : l'accès est libre
- ~ un sémaphore d'accès en écriture au tampon
 - ~ initialisé à N : le nombre d'emplacements libres
- ~ un sémaphore d'accès en lecture au tampon
 - ~ initialisé à 0 : le nombre d'emplacements occupés

```
#define N 100                                /* nombre de places dans le tampon */  
  
semaphore_t mutex, vide, plein;  
  
semaphore_init(&mutex, 1);                  /* contrôle d'accès au tampon */  
semaphore_init(&vide, N);                   /* nb de places libres */  
semaphore_init(&plein, 0);                  /* nb de places occupées */
```

Producteur consommateur à base de sémaphores (cont'd)



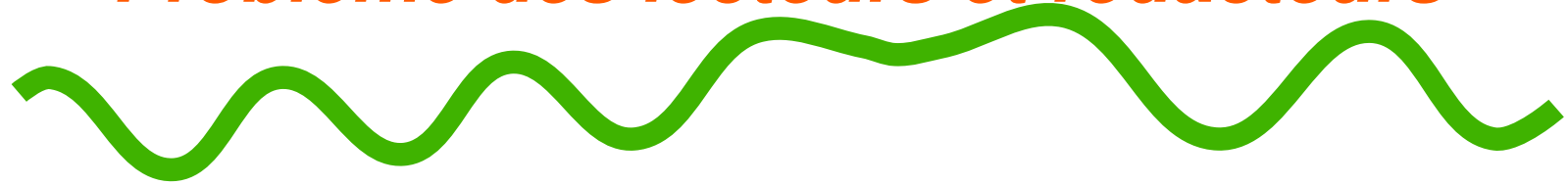
```
void producteur (void)
{
    objet_t objet ;

    while (1) {
        produire_objet(&objet);          /* produire l'objet suivant */
        semaphore_down(&vide);            /* dec. nb places libres */
        semaphore_down(&mutex);           /* entrée en section critique */
        mettre_objet(objet);              /* mettre l'objet dans le tampon */
        semaphore_up(&mutex);             /* sortie de section critique */
        semaphore_up(&plein);             /* inc. nb place occupées */
    }
}

void consommateur (void)
{
    objet_t objet ;

    while (1) {
        semaphore_down(&plein);           /* dec. nb emplacements occupés */
        semaphore_down(&mutex);           /* entrée section critique */
        retirer_objet (&objet);          /* retire un objet du tampon */
        semaphore_up(&mutex);             /* sortie de la section critique */
        semaphore_up(&vide);              /* inc. nb emplacements libres */
        utiliser_objet(objet);            /* utiliser l'objet */
    }
}
```

Problème des lecteurs et rédacteurs



~ Gestion des accès partagés à un fichier

- ~ par des lecteurs, et
- ~ par des rédacteurs
- ~ un autre problème classique !

```
void lecteur(void)
{
    while(1) {
        demande_de_lecture();
        lecture();
        fin_de_lecture();
    }
}
```

```
void redacteur(void)
{
    while(1) {
        demande_d_ecriture();
        ecriture();
        fin_d_ecriture();
    }
}
```

~ Règles d'accès assurant la cohérence du fichier

- ~ accès simultanés par plusieurs lecteurs possibles
- ~ accès exclusif au fichier pour les rédacteurs
 - ~ un seul rédacteur à la fois
 - ~ aucun lecteur pendant ce temps

Lecteurs rédacteurs à base de conditions



~ Mémoriser l'état de l'accès

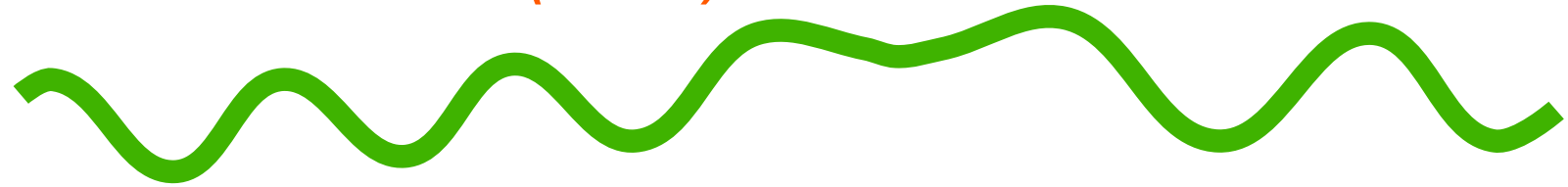
- ~ nombre de lecteurs dans le fichier
 - ~ initialisé à 0
- ~ y a-t-il un rédacteur dans le fichier
 - ~ booléen initialisé à faux

~ Files d'attente

- ~ une seule file d'attente ?
- ~ une file d'attente des lecteurs bloqués
- ~ une file d'attente des rédacteurs bloqués

```
~ int nb_lecteurs = 0;  
  bool redacteur = false;  
  cond_t c_lecture, c_ecriture;
```

Lecteurs rédacteurs à base de conditions (cont'd)



~ Première proposition

```
void demande_de_lecture(void)
{
    if (redacteur)
        cond_sleep(c_lecture);
    nb_lecteurs++;
    cond_wakeup(c_lecture);
}
```

```
void fin_de_lecture(void)
{
    nb_lecteurs--;
    if (nb_lecteurs==0)
        cond_wakeup(c_ecriture);
}
```

```
void demande_d_ecriture(void)
{
    if (nb_lecteurs!=0 || redacteur)
        cond_sleep(c_ecriture);
    redacteur = true;
}
```

```
void fin_d_ecriture(void)
{
    redacteur = false;
    if (!cond_empty(c_lecture))
        cond_wakeup(c_lecture);
    else
        cond_wakeup(c_ecriture);
}
```

- ~ un rédacteur quittant réveille un 1er lecteur qui va réveiller un 2nd lecteur...
- ~ on a implanté une priorité aux lecteurs !
- ~ priorité aux rédacteurs ? pas de priorité ?

~ Suspension d'une activité au milieu d'une des 4 fonctions ?

- ~ peut mener à une *race condition*
- ~ solution : les moniteurs de Hoare

Moniteurs de Hoare

Structure de programmation

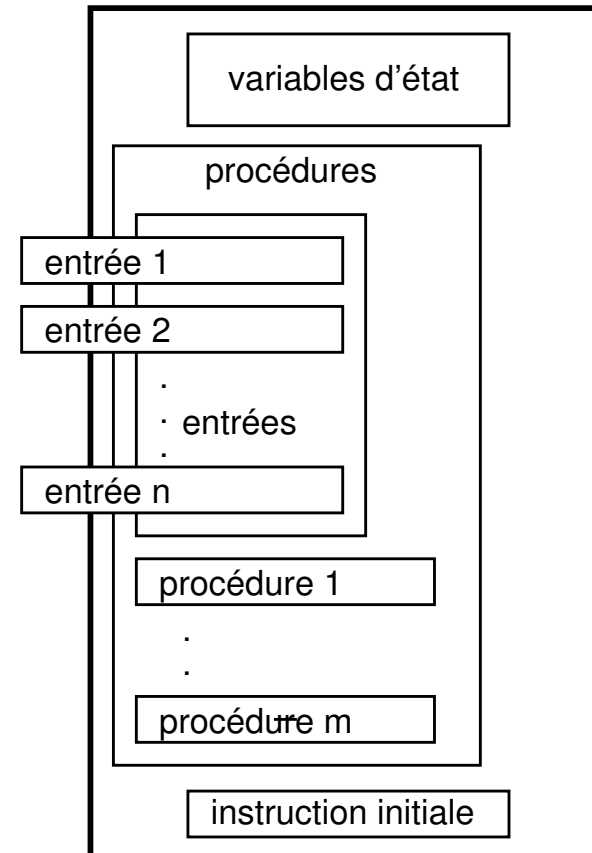
- un module de programme
- contrôle les accès à des données partagées

Variables d'état

- partagées entre les procédures du module
- encapsulées dans le module

Procédures

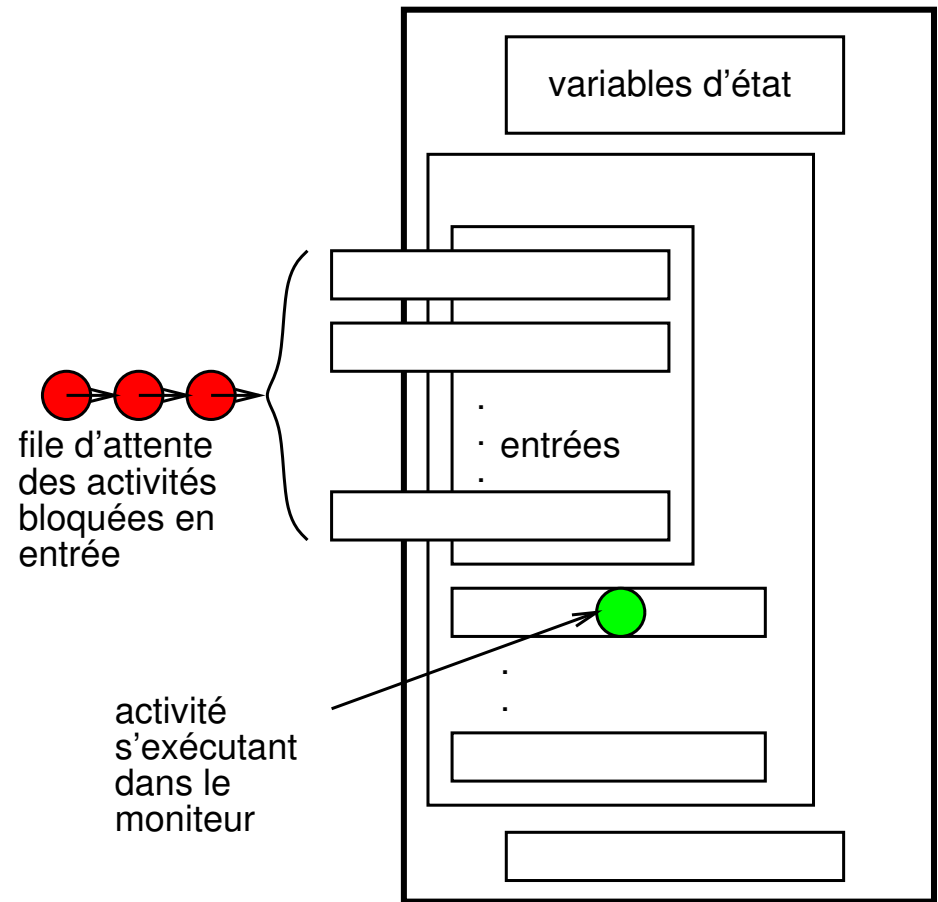
- locales
- entrées** : seules visibles depuis l'extérieur



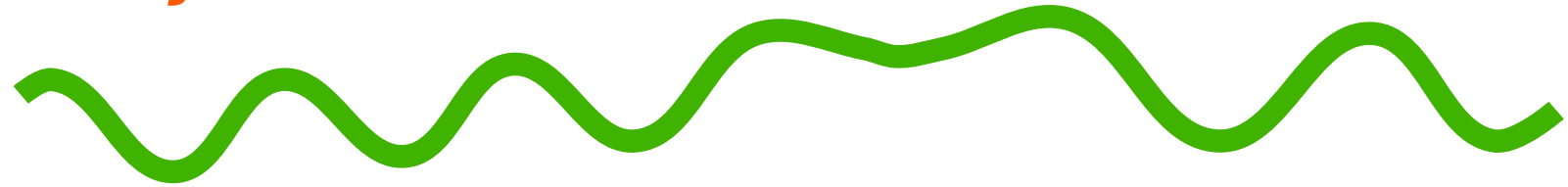
Synchronisation entre les activités concurrentes qui appellent les entrées

Mécanisme d'exécution du moniteur

- Exclusion mutuelle entre les entrées du moniteur
 - à un instant donné : une seule activité dans le moniteur
 - donc protection des variables d'état
- Si une seconde activité cherche à entrer dans le moniteur
 - bloquée jusqu'à libération du moniteur par l'activité précédente
 - file d'attente des activités bloquées en entrée du moniteur



Synchronisation dans le moniteur



~ L'activité s'exécutant dans le moniteur

- ~ peut s'apercevoir qu'elle ne peut pas continuer et vouloir s'endormir
- ~ peut aussi s'apercevoir qu'une activité en attente doit/peut reprendre son exécution

~ Une variable de type condition

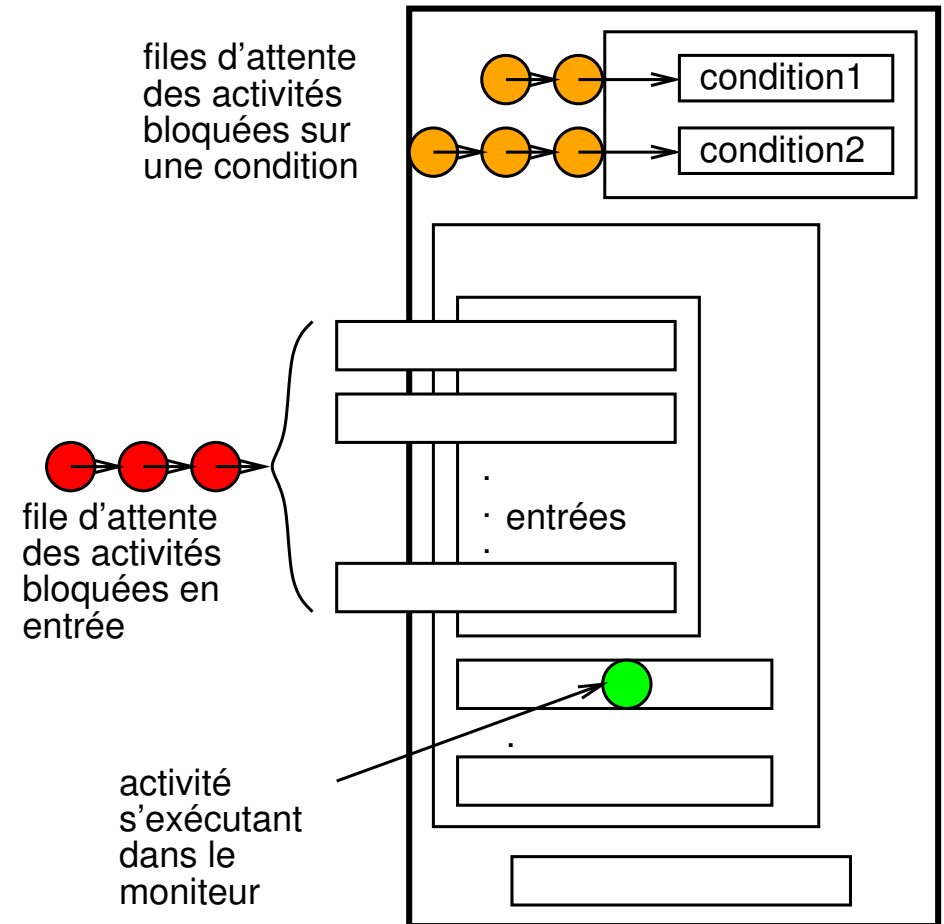
- ~ variable d'état du moniteur
 - ~ donc accessible uniquement de l'intérieur du moniteur
- ~ réalise une **synchronisation**
- ~ manipulée par deux opérations :
 - ~ attendre, `cond_sleep()`
 - ~ réveiller, `cond_wakeup()`

Synchronisation dans le moniteur (cont'd)

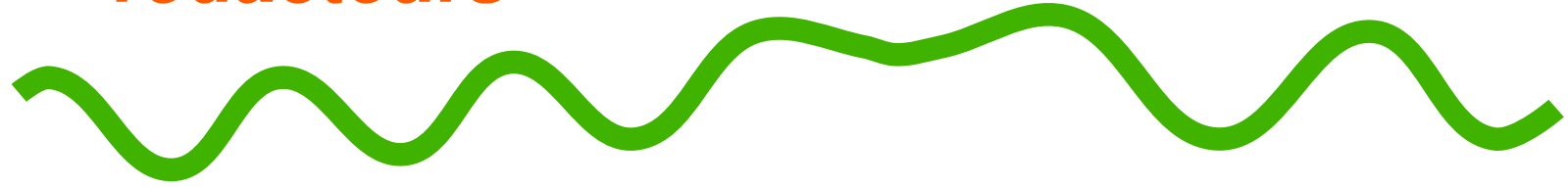
Pour une condition

- cond_sleep() bloque l'activité qui exécute cond_sleep() et la place en queue de file d'attente
- cond_wakeup() retire une activité en tête de la file d'attente (si $\neq \emptyset$) et l'active

Les conditions sont implantées comme des files d'attente d'activités bloquées



Moniteur de Hoare pour les lecteurs rédacteurs



~ Un moniteur

- ~ entrées
- ~ variables d'états

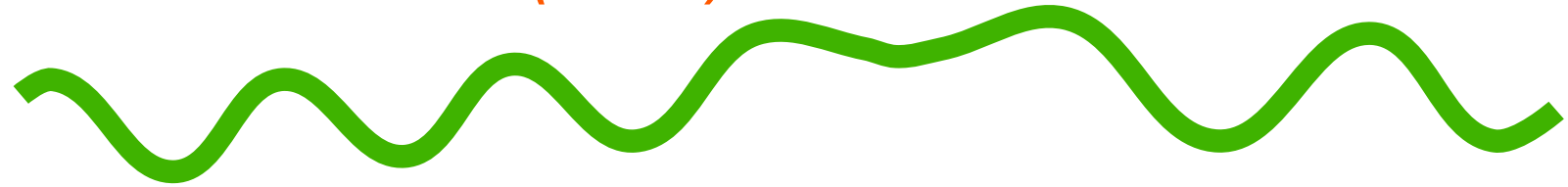
~ Entrées

- ~ `demande_de_lecture()` `fin_de_lecture()`
- ~ `demande_d_ecriture()` `fin_d_ecriture()`

~ Variables d'état

- ~ variable simples
 - ~ `nb_lecteurs`
 - ~ `redacteur`
- ~ conditions
 - ~ `lecture`
 - ~ `ecriture`

Moniteur de Hoare pour les lecteurs rédacteurs (cont'd)



~ Par construction du moniteur

- ~ exclusion mutuelle des accès aux variables d'état
- ~ non préemptibilité des activités au sein du moniteur

```
void demande_de_lecture(void)
{
    if (redacteur)
        cond_sleep(c_lecture);
    nb_lecteurs++;
    cond_wakeup(c_lecture);
}
```

```
void fin_de_lecture(void)
{
    nb_lecteurs--;
    if (nb_lecteurs==0)
        cond_wakeup(c_ecriture);
}
```

```
void demande_d_ecriture(void)
{
    if (nb_lecteurs!=0 || redacteur)
        cond_sleep(c_ecriture);
    redacteur = true;
}
```

```
void fin_d_ecriture(void)
{
    redacteur = false;
    if (!cond_empty(c_lecture))
        cond_wakeup(c_lecture);
    else
        cond_wakeup(c_ecriture);
}
```


Critique des moniteurs de Hoare



~ Problème du signaleur

- ~ suite à un `cond_wakeup()` deux activités sont potentiellement actives
- ~ le « signaleur » et le « signalé »
- ~ garantir l'exclusion mutuelle des accès au moniteur ?
- ~ proposition de multiples solutions
 - ~ Hoare : suspension des signalés bloqués
 - ~ Brinch Hansen : obliger `cond_wakeup()` à être terminal

~ Moniteur = structure de programmation

- ~ prise en charge nécessaire dans le langage
- ~ ne peut être fournit sous forme d'une bibliothèque

~ Utilisation des moniteurs

- ~ support à l'élaboration de raisonnements
- ~ mise au point des algorithmes
- ~ puis implantation à l'aide de verrous, conditions, sémaphores...