Les « threads » de posix.1c

Michel RIVEILL

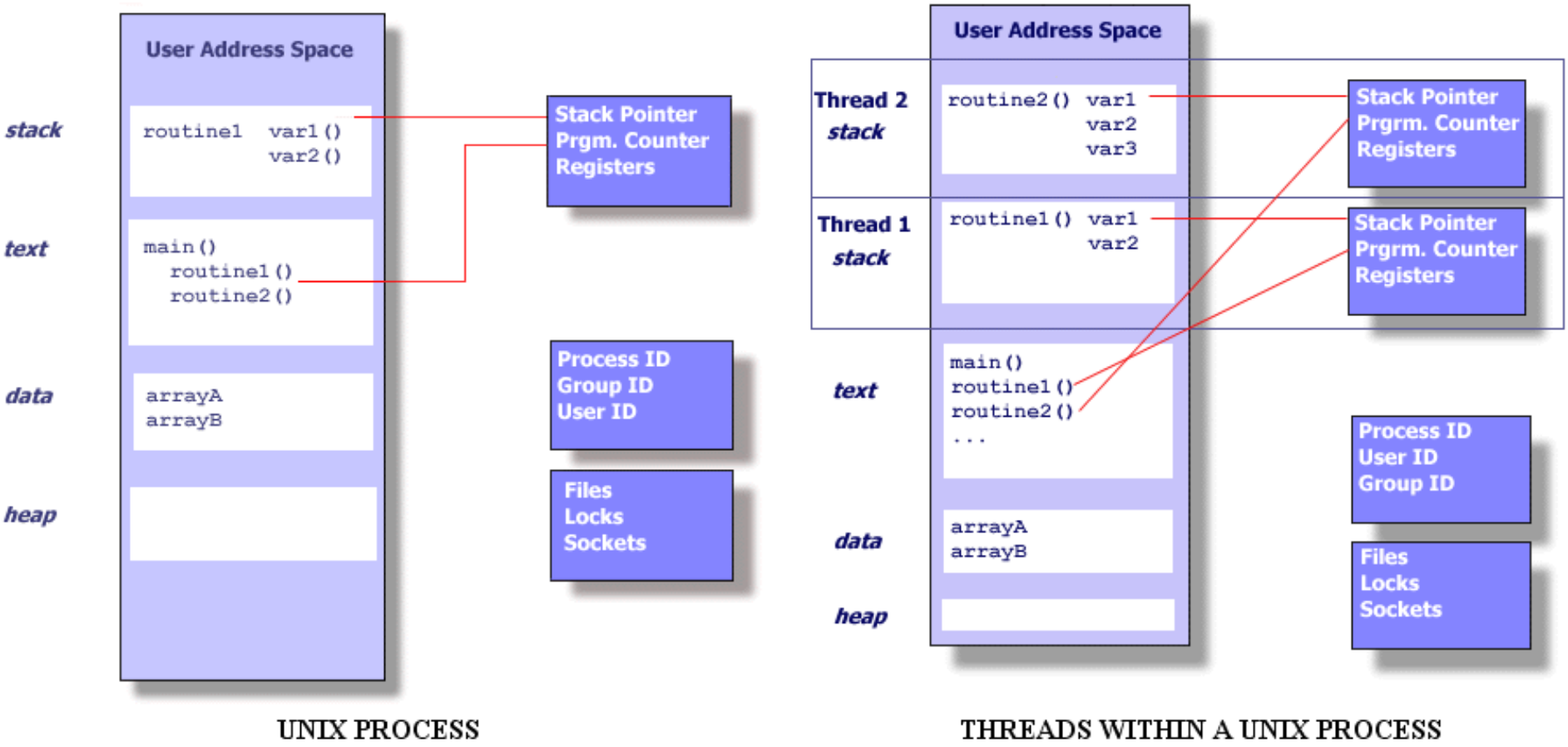
Les « threads » de posix.1c

- POSIX
  - POSIX = Portable Operating System Interface
- Pthread
  - Posix thread
- Pour le système UNIX
  - Interface adoptée en 1995 par l' IEEE (POSIX 1003.1c)

La norme POSIX

- Objectifs
  - Définir une interface standard des services offert par Unix afin de rendre portables les programmes
- Plusieurs chapitres, avec des parties obligatoires et d'autres optionnelles
  - POSIX 1003.1 — Services de base (fork, exec)
  - POSIX 1003.1b — Temps réel (sémaphores)
    - Presque tous les composants sont optionnels
  - **POSIX 1003.1c — Threads**
  - POSIX 1003.2 — Commandes shell (sh)
  - POSIX 1003.5 — POSIX en ADA
  - …

Processus versus thread (rappel)



Définition des threads

Processus traditionnel (processus lourd)

1. Un seul flot de contrôle séquentiel
2. Un espace d'adressage
3. Unité d'allocation des ressources système
4. Unité d'ordonnancement

Processus multi-threads

1. Plusieurs flots de contrôle séquentiels concurrents : threads
2. Un seul espace d'adressage
3. Unité d'allocation de la plupart des ressources système
4. Ordonnancement
   - Thread utilisateur : le processus lourd est plus l'unité d'ordonnancement
   - Thread noyau : les threads sont directement ordonnancées

Attributs d'un thread (1)

- Partage des attributs du processus lourd
  - Espace d'adressage : segments data et text
  - Variables globales
  - Fichiers ouverts, répertoire courant, masque de création…
  - Etat des signaux
  - Processus fils
  - Horloges, sémaphores…
  - Information de comptabilité, statistiques…

Attributs d'un thread (2)

- Attributs spécifiques à chaque thread
  - Compteur ordinal
  - Pile
  - Registres CPU
  - Thread fils
  - Etat : bloqué, prêt, actif...
  - Variables système propres (e.g. errno)
  - Données propres définies par l'utilisateur
  - Masque des signaux bloqués

Protection entre threads

- Chaque thread a des ressources propres
  - Pile
  - Données via son propre pointeur de données
- Mais aucune protection entre les threads d'un même processus
  - Il est possible de modifier des zones mémoires utilisées par les autres threads
  - Impossible, sans un surcoût élevé
  - En principe inutile (!)

Les « threads » de posix.1c
La norme posix.4

- Chapitre 1b : extensions « temps réel »
  - Fichiers consécutifs, E/S asynchrone
  - Gestion du temps
  - Ordonnancement préemptif à priorité (temps réel)
  - Mémoire partagée, sémaphores, files de messages
    - L'équivalent Posix de ce que l'on a vu la semaine dernière pour Unix
- Chapitre 1c : Threads
  - Flots de contrôle concurrents dans le même espace d'adressage
  - Mécanismes de synchronisation
    - Mutex et variables conditions
  - Contraintes sur posix.1 ➡ posix.1c

Création des threads

```
pthread_t                                   // type de la variable

int pthread_create(
        pthread_t *pth,                     // identificateur du thread
        const pthread_attr_t *pattr,        // attribut
        void (*start)(void *),              // fonction à exécuter
        void *arg                           // arguments de la fonction
);

thread_t pthread_self()                     // auto-identification
```

- Attributs : NULL = attributs par défaut
  - Etat : détachée ou joignable
  - Caractéristiques d'ordonnancement
  - Attributs de la pile...

Un exemple

```
int g;
void do_it_1(void *arg) {
    int n = *arg;
    for (i = 0; i < n; i++) g = i;
}
void do_it_2(void *arg) {
    int n = *(int *)arg;
    for (i = 0; i < n; i++) printf("%d\n", g);
}
main() {
    pthread_t th1, th2;
    int n = 10;
    pthread_create(&th1, NULL, do_it_1, &n);
    pthread_create(&th2, NULL, do_it_2, &n);
}
```

La thread du main()

- Une thread est crée automatiquement pour le programme exécutant le 'main'
- Si main se termine (exit())
  - Alors l'ensemble du processus lourd se termine
  - Ses threads aussi

Terminaison/attente d'une thread (1)

- Autoterminaison

```
int pthread_exit(void *value); // valeur de retour
```

- Attente d'une autre thread

```
int pthread_join(
   pthread_t th,          // thread attendue
   void **value);         // valeur de retour (ou NULL)
```

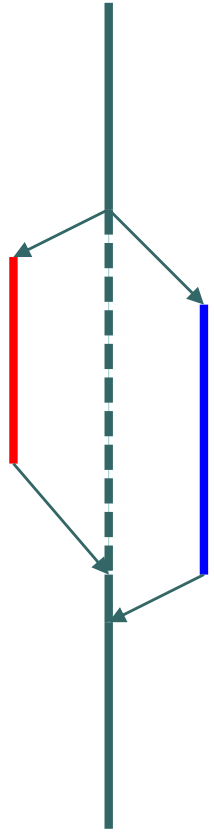
  - N'importe quelle thread peut en attendre une autre
  - Comportement indéfini quand la terminaison d'une thread est attendue simultanément par plusieurs autres threads

Terminaison/attente d'une thread (2)

```
// mes deux fonctions précédentes

main() {
    pthread_t th1, th2;
    int n = 10;

    pthread_create(&th1, NULL, do_it_1, &n);
    pthread_create(&th2, NULL, do_it_2, &n);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    ...
}
```



Données propres des threads (1)

- Les données locales 'automatiques' sont propres à chaque thread
  - puisqu'elles sont allouées dans la pile
- Les données statiques
  - sont globales à toutes les threads
- Possibilité de données propres
  - Mais gérées dans le tas du processus
  - Donc accessible à tous

Données propres des threads (2)

- Création de zones de données propres accessibles par des clés
  - pthread_key_t
  - pthread_key_create(...)
  - pthread_key_setspecific(...)
  - pthread_key_getspecific(...)
- La zone de donnée globale doit être allouée par le programmeur
- La zone propre n'est protégée en aucune manière
- Utile surtout si l'on ne connaît pas a priori le nombre de threads à créer

Un exemple

Destruction des threads (cancellation) (1)

Solution 1 : destruction directe d'une thread

- thread_cancel(pthread_t th); // la thread à tuer

- Mécanisme de synchronisation rustique et brutal
- Opération extraordinairement dangereuse
  - Risque d'incohérence sur les données partagées
  - Risque de confiscation de ressources système
  - Non libération de verrous…
- Ça existe mais on n'utilise pas

Destruction des threads (cancellation) (2)

Solution 2 (préférable) : en des points ou la thread peut être détruite

- Points de destruction (cancellation)
  - Tout point où la thread est ou pourrait être mise en attente (ce sont des points d'ordonnancement potentiels)
    - attente d'une autre thread, d'un signal, d'une condition...
    - certains appels-système
  - Des points supplémentaires peuvent être décidé par le programmeur grâce à la primitive pthread_testcancel()

Destruction des threads (cancellation) (3)

- Etats et types de 'cancellation' :
  - DISABLE : la thread est immunisée contre les tentatives de destruction i.e. elle ne peut pas être détruite
  - ENABLE : la thread peut être détruite
    - Quand ? en en fonction du type de 'cancellation' :
    - **ASYNCHRONOUS** : « dès que possible » (dangereux)
    - **DEFERRED** : au prochain « point de cancellation »
- Par défaut la thread est ENABLE/DEFERRED
- Peut être modifié par
  - pthread_setcancelstate(...)
  - pthread_setcanceltype(...)

Ordonnancement des threads (1)

- Attributs d'ordonnancement définis par threads
  - Priorité
    - Permet de choisir la/les threads à activer
    - Ordonnancement préemptif
  - Politique
    - Permet de gérer entre elles les threads de même priorité
    - Temps réel : SCHED_FIFO, SCHED_RR (Round-Robin)
    - SCHED_OTHER : dépend de l'implémentation (souvent type temps partagé) ; c'est la valeur par défaut
  - Portée
    - Portée de la compétition entre threads : système ou processus

Ordonnancement des threads (2)

- int pthread_setschedparam (
  pthread_t th,
  int policy,
  const struct sched_param *param);
- int pthread_getschedparam (…);
- pthread_attr_setschedXXX (…);
- pthread_attr_getschedXXX (…);

Threads et appels-système (de posix.1)

- Threads et signaux
- Threads et fork()
- Bibliothèques « thread-safe » (MT-safe)

Threads et signaux (1)

- Signaux internes : pthread_kill(...)
  - D'une thread à une autre du même processus lourd
  - Reçu par le thread destinatrice
- Déroutements (exceptions hardware)
  - Provoqués par une thread, reçus par elle
- Signaux asynchrones (issus de l'extérieur)
  - Reçus par le processus lourd... qui fait éventuellement suivre sur un thread acceptant le signal

Threads et signaux (2)

- L'état des signaux (défaut, ignoré, action de l'utilisateur, bloqué) est global pour tout le processus lourd
  - Il ne peut y avoir simultanément plusieurs « handlers » pour le même signal
- Il existe un masque des signaux bloqués par thread
  - On peut donc faire attendre à une thread un signal asynchrone spécifique
  - sig_procmask(…)
  - sigwait(…)

Threads et fork() (1)

- Que se passe-t-il lorsqu'une thread invoque fork() ?
  - Le processus lourd est dupliqué
  - Les threads le sont-elles ?
    - NON, pas dans posix.1c
    - Le processus fils est mono-thread ; seule la thread ayant exécuté fork() est dupliquée
  - Que se passe-t-il pour les ressources (mémoire, fichiers…) et verrous détenus par les (multiples) threads du père ?
    - La thread unique du fils ne sait même pas qu'ils existent !

Les « threads » de posix.1c
Threads et fork() (2)

- Enregistrement d'actions à effectuer avant le fork, puis après le fork dans le père et le fils

```
int pthread_atfork(
        void (*prepare)(void),      // dans le père, avant
        void (*parent)(void),       // dans le père, après
        void (*child)(void)         // dans le fils, après
        );
```

- On peut enregistrer plusieurs actions
  - À l'aide de plusieurs appels de pthread_atfork avant un fork
- Le mécanisme est horriblement complexe

Bibliothèques « thread-safe »

- Dans posix.1c, la grande majorité des fonctions de posix.1 est « thread-safe » (i.e. réentrantes)
  - Peuvent être suspendues et réexécutées pour le compte d'une autre thread
  - Elles ne renvoient de pointeurs sur des objets statiques
  - La variable errno est définie pour chaque thread

- Si une bibliothèque n'est pas « thread-safe », il faut exécuter les fonction en section critique
  - appel protégé par l'utilisation d'un verrou de type mutex

Panorama des mécanismes de synchronisation

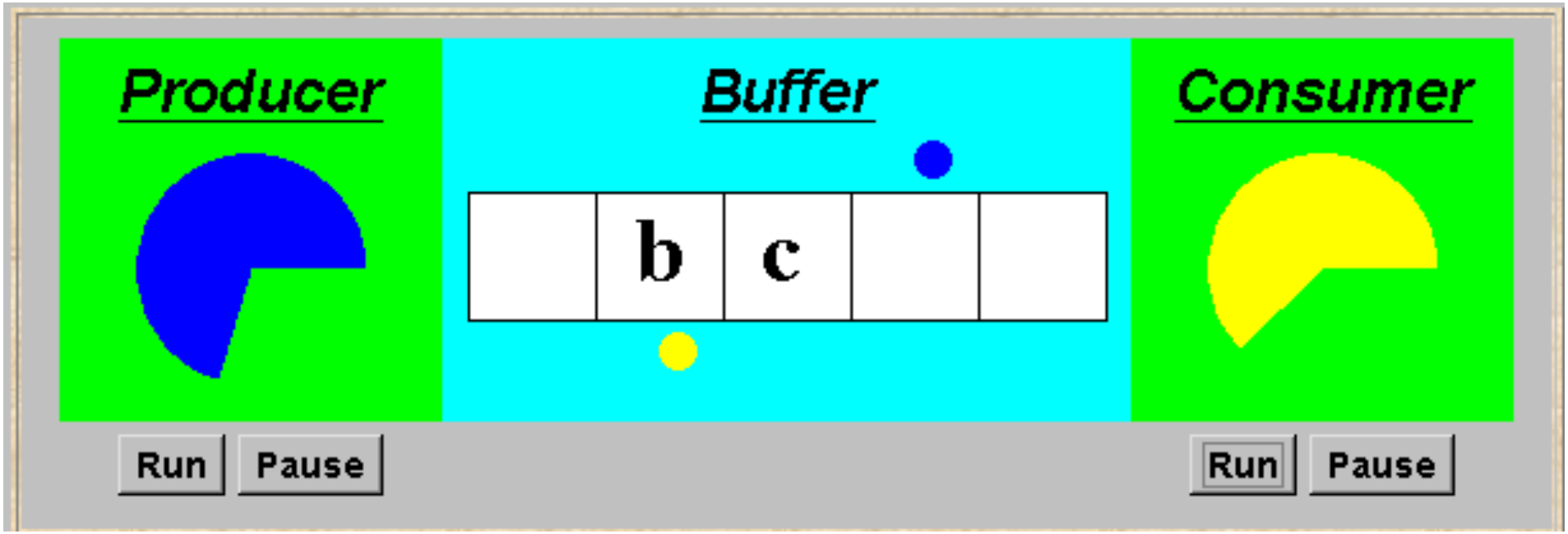
- Fonction pthread_join
  - Pour mémoire
- Verrou d'exclusion mutuelle (mutex)
  - Verrou garantissant l'exclusivité d'accès (à une donnée, à du code)
- Variable de condition
  - Nommage, attente et signalisation d'événements
- Fonction pthread_once
  - Unicité de l'initialisation

Le problème de l'exclusion mutuelle (exemple : producteur-consommateur)

```
main() {
  pthread_t th1, th2, th3;
  pthread_create(&th1, NULL, prod, 0);
  pthread_create(&th2, NULL, prod, 0);
  pthread_create(&th3, NULL, cons, 0);
}
```

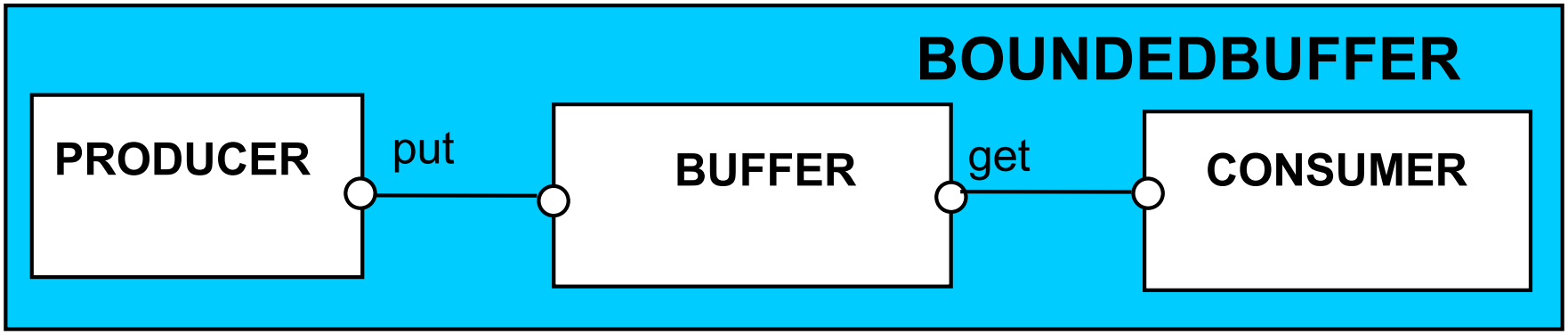
Rappel du problème

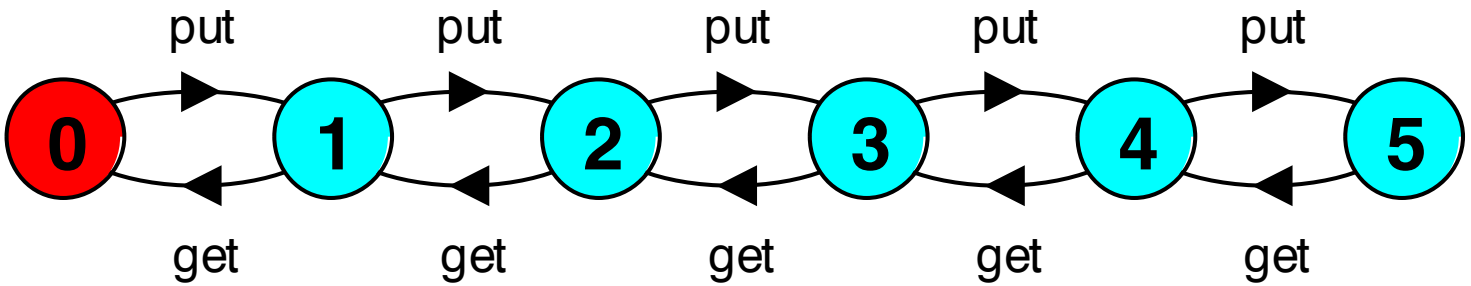
- Un buffer/tampon de taille fixe
- Des producteurs produisent des données et les déposent dans le buffer → 1 seul producteur par case, bloqués si buffer plein
- Des consommateurs, prennent les données qui sont dans le buffer → 1 seul consommateur par case, bloqués si buffer vide



Modélisation en FSP

- Le comportement du BUFFER est indépendant de la nature des données





Code FSP

```
const N = 5
BUFFER = COUNT[0],
COUNT[i:0..N]
       = (when (i<N) put->COUNT[i+1]
          |when (i>0) get->COUNT[i-1]
          ).

PRODUCER = (put->PRODUCER)+{get}.
CONSUMER = (get->CONSUMER)+{put}.

||BOUNDEDBUFFER = ({a, b}:PRODUCER
                  || {a, b, c, d}::BUFFER
                  || {c, d}:CONSUMER).
```

Exemple : producteur-consommateur

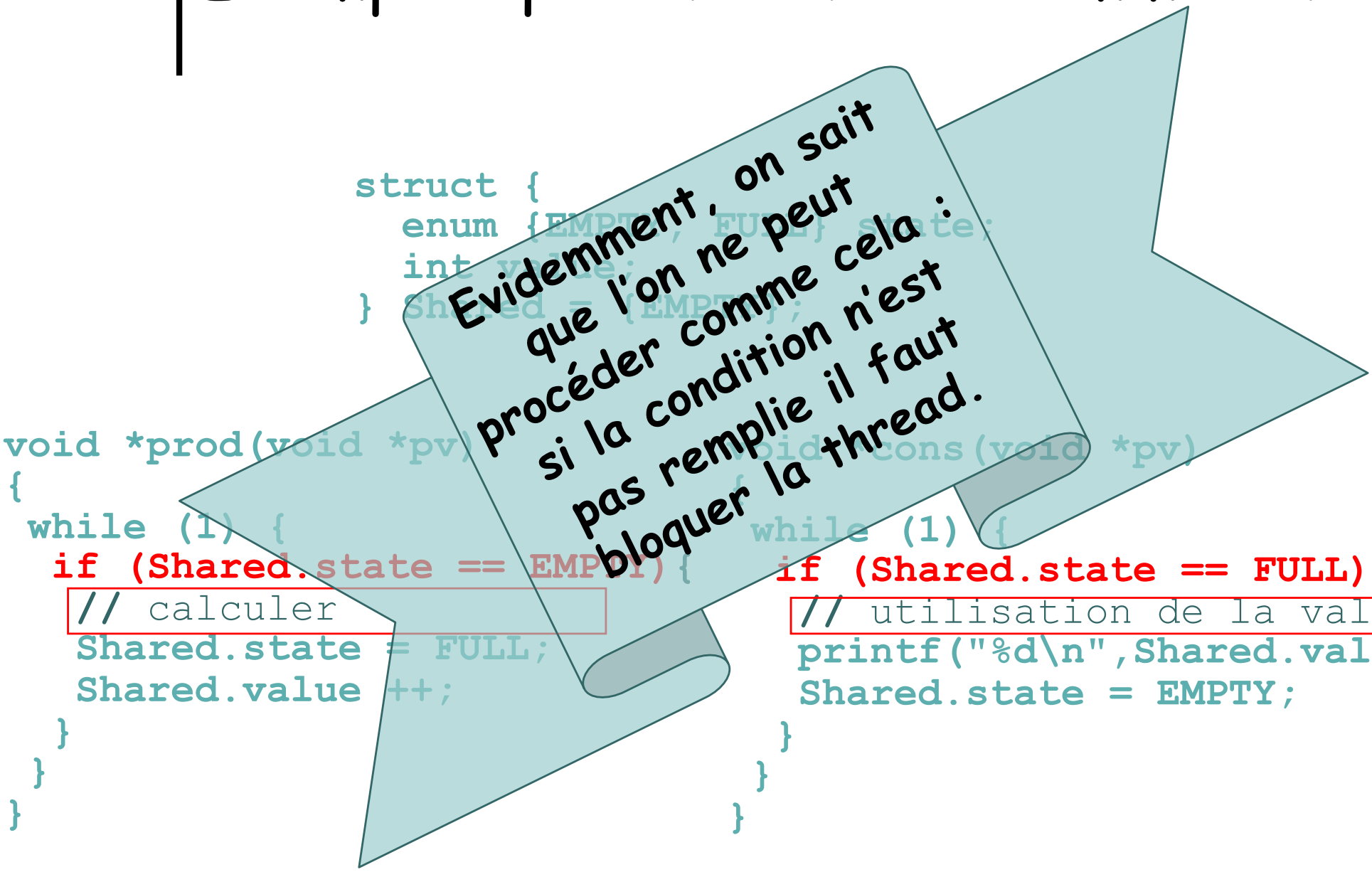
Pour simplifier on va prendre un tampon de longueur 1 qui est vide au démarrage

```
struct {
  enum {EMPTY, FULL} state;
  int value;
} Shared = {EMPTY};
```

```
void *prod(void *pv)
{
 while (1) {
  if (Shared.state == EMPTY){
   // calculer
   Shared.state = FULL;
   Shared.value ++;
  }
 }
}
```

```
void *cons(void *pv)
{
 while (1) {
  if (Shared.state == FULL) {
   // utilisation de la valeur
   printf("%d\n",Shared.value);
   Shared.state = EMPTY;
  }
 }
}
```

Exemple : producteur-consommateur

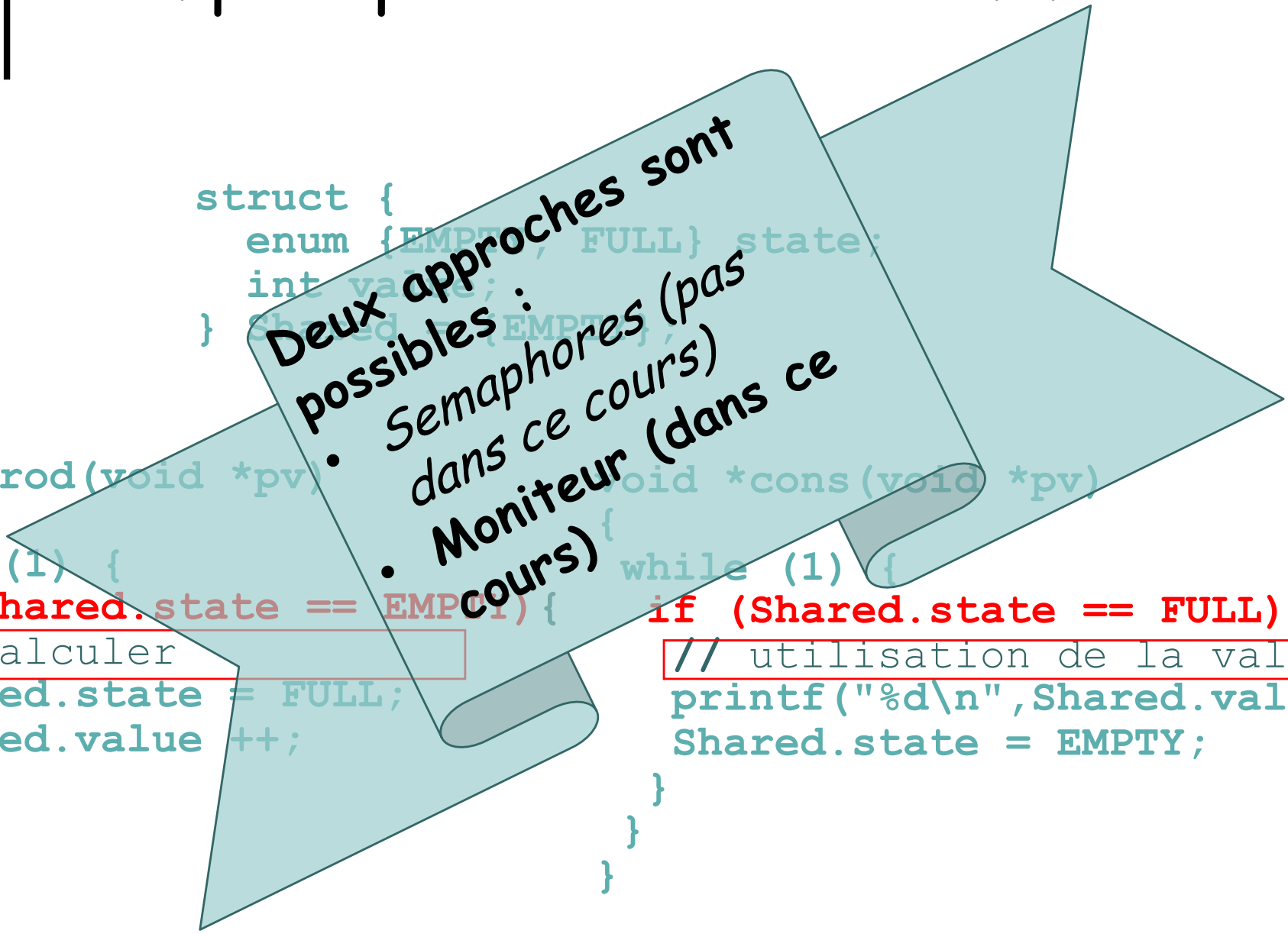



```
struct {
   enum {EMPTY, FULL} state;
   int value;
} Shared = {EMPTY};

void *prod(void *pv)
{
 while (1) {
  if (Shared.state == EMPTY){
   // calculer
   Shared.state = FULL;
   Shared.value ++;
  }
 }
}
```

```
void *cons(void *pv)
{
 while (1) {
  if (Shared.state == FULL) {
   // utilisation de la valeur
   printf("%d\n",Shared.value);
   Shared.state = EMPTY;
  }
 }
}
```

Exemple : producteur-consommateur




```
struct {
   enum {EMPTY, FULL} state;
   int value;
} Shared = {EMPTY};

void *prod(void *pv)
{
 while (1) {
  if (Shared.state == EMPTY){
   // calculer
   Shared.state = FULL;
   Shared.value ++;
  }
 }
}
```

```
void *cons(void *pv)
{
 while (1) {
  if (Shared.state == FULL) {
   // utilisation de la valeur
   printf("%d\n",Shared.value);
   Shared.state = EMPTY;
  }
 }
}
```

Verrou d'exclusion mutuelle (1)

Déclaration et initialisation

- Définition simple

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Mise en place d'attributs

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, &attributs);
```

  - Pour les attributs, NULL correspond aux valeurs par défaut
  - Les attributs permettent le partage du mutex entre plusieurs processus lourds (via la mémoire partagée)

Verrou d'exclusion mutuelle (2)

Utilisation

- Demande du verrou
  - Demande suspensive

```
pthread_mutex_lock(&mutex);
```

  - Tentative sans blocage

```
pthread_mutex_trylock(&mutex);
```

- Libération du verrou

```
pthread_mutex_unlock(&mutex);
```

Verrou d'exclusion mutuelle (2)

Utilisation

- Demande du verrou
  - Demande suspensive

```
pthread_mutex_lock(&mutex);
```

  - Tentative sans blocage

```
pthread_mutex_trylock(&mutex);
```

- Libération du verrou

```
pthread_mutex_unlock(&mutex);
```

Equivalent aux actions lock, unlock de la classe Lock

Ou de

synchronized () {…}

Exemple : mise en œuvre de la section critique

```
struct {
  enum {EMPTY, FULL} state;
  int value;
} Shared = {EMPTY};
```

```
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```
void *prod(void *pv)
{
 while (1) {
  pthread_mutex_lock(&mutex);
  if (Shared.state == EMPTY){
   Shared.state = FULL;
   Shared.value++;
  }
  pthread_mutex_unlock(&mutex);
 }
}
```

```
void *cons(void *pv)
{
 while (1) {
  pthread_mutex_lock(&mutex);
  if (Shared.state == FULL) {
   printf("%d\n",Shared.value);
   Shared.state = EMPTY;
  }
  pthread_mutex_unlock(&mutex);
 }
}
```

Exemple : mise en œuvre de la section critique

```
struct {
  enum {EMPTY, FULL} state;
  int value;
} Shared = {EMPTY};
```

```
void *prod(void *pv)
{
 while (1) {
  pthread_mutex_lock(&mutex);
  if (Shared.state == EMPTY){
   Shared.state = FULL;
   Shared.value++;
  }
  pthread_mutex_unlock(&mutex);
 }
}
```

```
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
```

```
void *cons(void *pv)
{
 while (1) {
  pthread_mutex_lock(&mutex);
  if (Shared.state == FULL) {
   printf("%d\n",Shared.value);
   Shared.state = EMPTY;
  }
  pthread_mutex_unlock(&mutex);
 }
}
```

En C, nous n'avons pas de classe… la mise en œuvre d'un Moniteur est légèrement différente :

- Identification des fonctions du moniteur → code qui s'exécute en section critique

Variable de condition (1)

Déclaration et initialisation

- Définition simple

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Mise en place d'attributs

```
pthread_cond_t cond;
pthread_cond_init(&cond, &attributs);
```

  - Pour les attributs, NULL = valeur par défaut
  - Les attributs permettent le partage des variables de condition entre plusieurs processus lourds (via la mémoire partagée)

Variable de condition (2)

Utilisation

- Attente de la condition

  ```
  pthread_cond_wait(&cond, &mutex);
  ```

  - L'attente est toujours associée à un mutex
  - Le mutex est libéré au moment de la mise en attente
- Signalisation de la condition

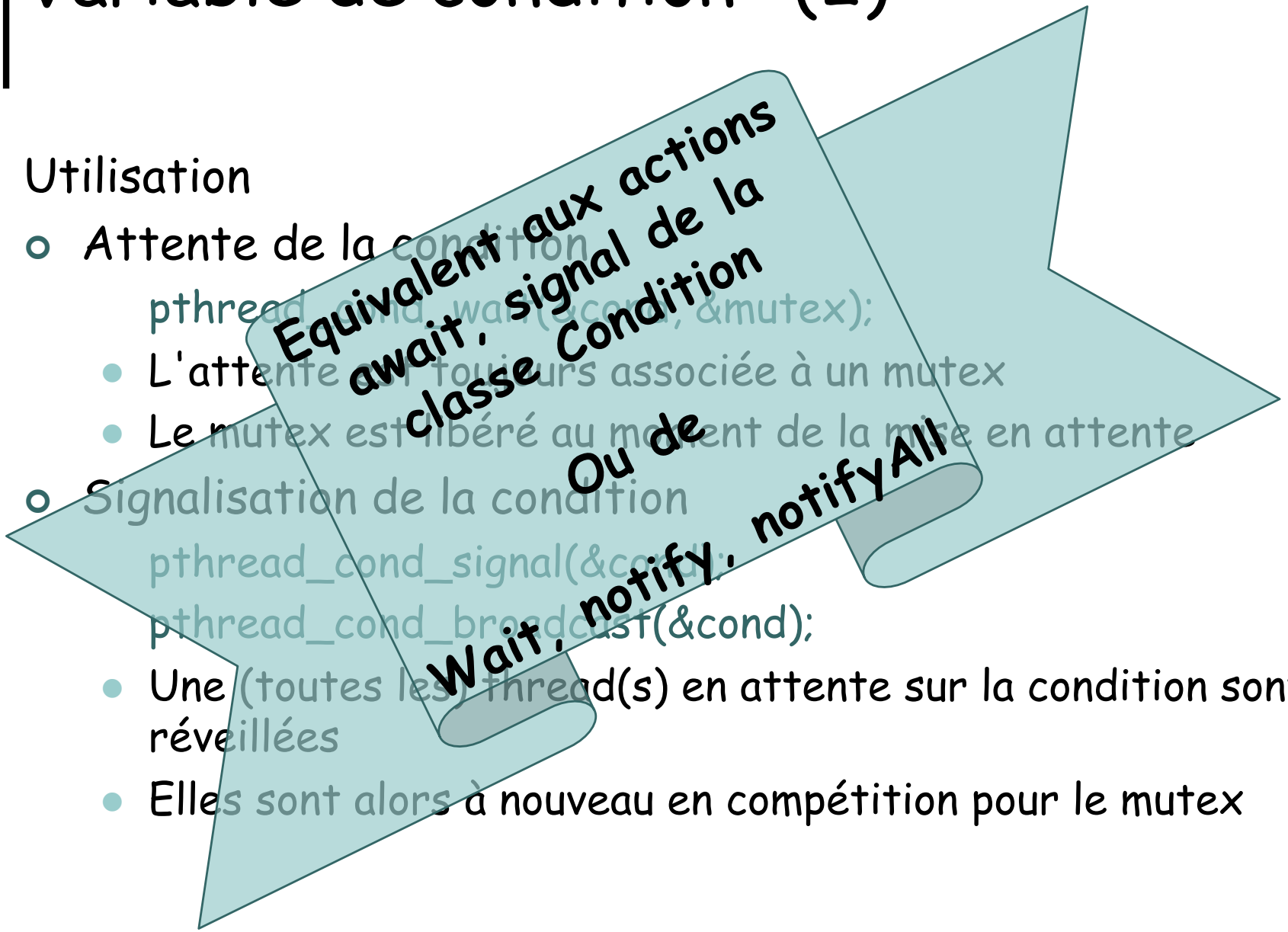
  ```
  pthread_cond_signal(&cond);
  pthread_cond_broadcast(&cond);
  ```

  - Une (toutes les) thread(s) en attente sur la condition sont réveillées
  - Elles sont alors à nouveau en compétition pour le mutex

Variable de condition (2)

Utilisation

- Attente de la condition
  ```
  pthread_cond_wait(&cond, &mutex);
  ```
  - L'attente est toujours associée à un mutex
  - Le mutex est libéré au moment de la mise en attente
- Signalisation de la condition
  ```
  pthread_cond_signal(&cond);
  pthread_cond_broadcast(&cond);
  ```
  - Une (toutes les) thread(s) en attente sur la condition sont réveillées
  - Elles sont alors à nouveau en compétition pour le mutex



Exemple : utilisation du wait/signal

```
struct {
  enum {EMPTY, FULL} state;
  int value;
} Shared = {EMPTY};
```

```
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t isfull =
    PTHREAD_COND_INITIALIZER;
```

```
void *prod(void *pv)
{
 while (1) {
   pthread_mutex_lock(&mutex);
   if (Shared.state == EMPTY){
    Shared.state = FULL;
    Shared.value++;
    pthread_cond_signal(&isfull);
   }
   pthread_mutex_unlock(&mutex);
 }
}
```

```
void *cons(void *pv)
{
  while (1) {
    pthread_mutex_lock(&mutex);
    if (Shared.state == EMPTY)
      pthread_cond_wait(&isfull,
                        &mutex);
    printf("%d\n",Shared.value);
    Shared.state = EMPTY;
    pthread_mutex_unlock(&mutex);
  }
}
```

Exemple : utilisation du wait/signal

```
struct {
  enum {EMPTY, FULL} state;
  int value;
} Shared = {EMPTY};
```

```
pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t isfull =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t isempty =
    PTHREAD_COND_INITIALIZER;
```

```
void *prod(void *pv)
{
 while (1) {
   pthread_mutex_lock(&mutex);
   if (Shared.state == FULL)
     pthread_cond_wait(&isempty,
                       &mutex);
   Shared.state = FULL;
   Shared.value++;
   pthread_cond_signal(&isfull);
   pthread_mutex_unlock(&mutex);
 }
}
```

```
void *cons(void *pv)
{
  while (1) {
    pthread_mutex_lock(&mutex);
    if (Shared.state == EMPTY)
      pthread_cond_wait(&isfull,
                        &mutex);
      printf("%d\n",Shared.value);
      Shared.state = EMPTY;
    pthread_cond_signal(&isempty);
    pthread_mutex_unlock(&mutex);
  }
}
```

Java vs Posix

```
synchronized prod (Data d) {
  while (Shared.state == FULL)
    wait();
  //…
  notifyAll();
}
```

```
void *prod(void *pv)
{
   pthread_mutex_lock(&mutex);
   if (Shared.state == FULL)
     pthread_cond_wait(&isempty,
                       &mutex);
   // …
   pthread_cond_signal(&isfull);
   pthread_mutex_unlock(&mutex);
}
```

```
synchronized Data cons {
  while (Shared.state == EMPTY))
    wait();
  //…
  notyfyAll();
}
```

```
void *cons(void *pv)
{
    pthread_mutex_lock(&mutex);
    if (Shared.state == EMPTY)
      pthread_cond_wait(&isfull,
                        &mutex);
    // …
    pthread_cond_signal(&isempty);
    pthread_mutex_unlock(&mutex);
}
```

Synchronisation de l'initialisation (1)

- Certains ensemble de threads ont besoin d'initialiser des objets communs avant de pouvoir fonctionner
- Cette initialisation doit avoir lieu une seule fois
- Il n'est pas toujours possible d'identifier une thread « maitresse » qui effectuerait l'initialisation

- D'où le mécanisme de pthread_once

Synchronisation de l'initialisation (2)

- Initialisation statique d'un « once block »

```
pthread_once_t once_block = PTHREAD_ONCE_INIT;
```

- Chaque thread appelle la fonction d'initialisation

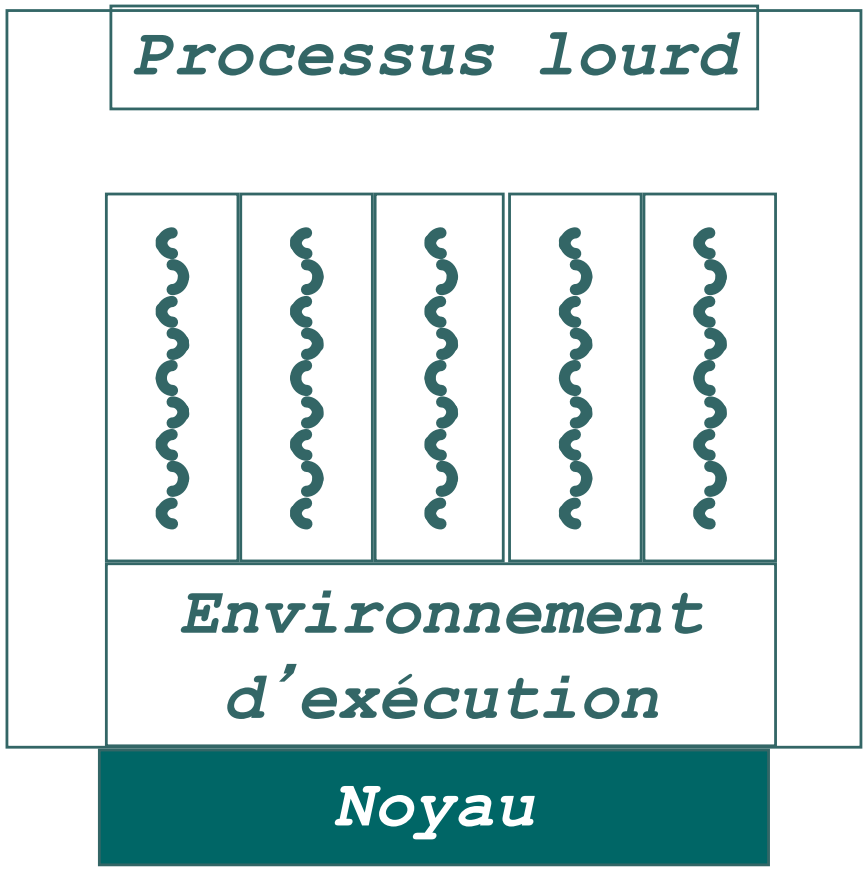
```
pthread_once(&once_block, &init_func);
```

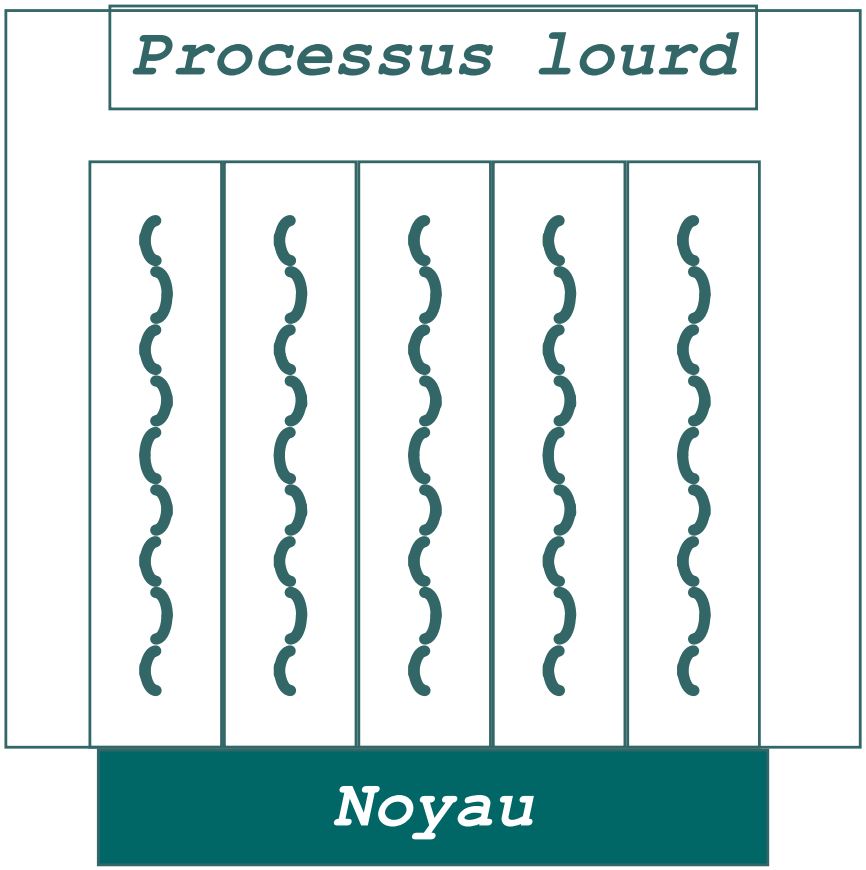
- La fonction d'initialisation est garantie n'être appelée qu'une seule fois
  - Elle doit être invoquée uniquement par pthread_once
  - Il peut y avoir plusieurs « once block » avec chacun une fonction d'initialisation

Plan général

- Définition et motivation
- Les threads de posix.1c (Pthreads)
- Implémentation des threads
  - Threads en mode utilisateur
  - Threads en mode noyau
  - Problèmes généraux

Mode utilisateur ou mode noyau ?





En mode utilisateur : avantages

- Le noyau ne sait rien des threads
  - Pas de ressources à gérer
- Commutation rapide
- Possibilité pour l'utilisateur d'adapter ou de redéfinir l'ordonnancement

En mode utilisateur : inconvénients

- L'ordonnancement des threads nécessite l'ordonnancement du processus lourd
- Un seul thread actif par processus lourd
- Difficile d'utiliser un ordonnancement préemptif par tranche de temps
  - manque de finesse de l'horloge
- Appels système bloquants
  - Certains peuvent être redéfinis comme non-bloquants (E/S)
  - Mais pas tous !

En mode noyau : avantages

- Ordonnancement par thread
- Possibilité de réponse « temps réel »
- Plus de problème avec les appels-système bloquants
- Si les threads sont souvent bloquées sur des appels-système, le surcoût de commutation est faible pour le noyau

Threads en mode noyau : inconvénients

- Coût plus élevé de commutation
- Ordonnancement difficilement modifiable

Résumé des types et des fonctions utilisés

Types et fonctions	Description
pthread_t	Contexte de thread
pthread_create()	Création d'une thread
pthread_join()	Attente terminaison
pthread_mutex_t	Sémaphore d'exclusion mutuelle
pthread_mutex_lock()	Début de zone critique
pthread_mutex_unlock()	Fin de zone critique
pthread_cond_t	Condition
pthread_cond_wait()	Mise en attente
pthread_cond_signal()	Déclenchement

TD : 'autour du parking'

Utilisation :

- Moniteurs à la posix