La programmation multithreadée

Plan (1)

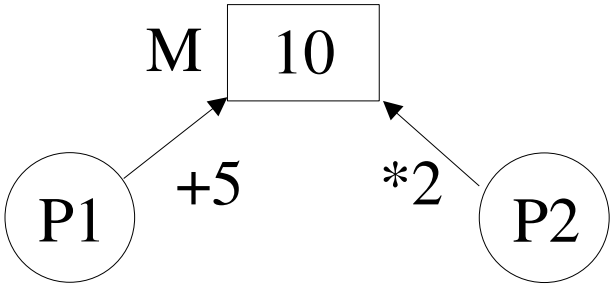
- Synchronisation de processus : rappels
  - synchronisation : problèmes
  - outils de synchronisation : les sémaphores
- La programmation multithreadée
  - définition, caractéristiques
  - structure d'un processus bi-threadé
  - utilisation du multithreading

Plan du cours (2)

- Les threads POSIX : la bibliothèque pthread
  - création et terminaison d'un thread
  - les pièges
  - synchronisation de threads
    - sémaphores
    - exemple : le modèle producteur/consommateur
    - variables conditionnelles
    - exemple : le modèle producteur/consommateur

Synchronisation de processus : problèmes (1)

- Pb1 : Non déterminisme
  - Ex : deux processus accèdent sans contrôle à une même cellule mémoire M



  - ✔ P1 puis P2 ⇒ (10 + 5) * 2 = 30
  - ✔ P2 puis P1 ⇒ (10 * 2) + 5 = 25

Synchronisation de processus : problèmes (2)

- Pb2 : les interblocages
  - un ensemble de processus est en interblocage si chaque processus de l'ensemble attend un événement que seul un autre processus de l'ensemble peut engendrer
  - exemple d'interblocage :
    - A demande une ressource R1 et l'obtient
    - B demande une ressource R2 et l'obtient
    - A demande la ressource R2
    - B demande la ressource R1
    - ⇒ A et B sont suspendus et en interblocage

Synchronisation de processus : problèmes (3)

- Pb3 : Cohérence des données
  - Ex : réservation de places de spectacles

```
#define N 25
void reserver (int dem) {
    if (res + dem <= MAX) res += dem;
    else ..... /* erreur */
}
```

  - Problème : préemption après le test
  - Solution : accès exclusif à la procédure de réservation

Synchronisation de processus : exclusion mutuelle

- Une section critique est une suite d'instructions dont l'exécution est gérée en exclusion mutuelle
  - Un processus peut entrer en section critique si et seulement si aucun processus ne s'y trouve
- Réalisation de l'exclusion mutuelle
  - Mauvaises solutions :

```
Si SC-libre alors                TQ non SC-libre Faire
        Reserver(10)             FTQ
Fsi                              Réserver(10)
```

Outils de synchronisation : sémaphores (1)

- Un sémaphore est une variable à valeur entière positive ou nulle, manipulable par l'intermédiaire de deux opérations P (Proberen) et V (verhogen)
- P(S) : Si S ≤ 0 alors

  mettre le processus en attente

  sinon

  S = S - 1

  Fsi
- V(S) : S = S + 1

  reveil d'un processus en attente

Outils de synchronisation : sémaphores (2)

- Implantation des primitives P et V :
  - Atomicité des primitives
  - Existence d'un mécanisme de file d'attente pour mémoriser les opérations P non satisfaites
- Exemple 1:

```
void reserver (int dem) {
    P(mutex);
    if (res + dem <= MAX) {
        res + = dem; V(mutex); }
    else { V(mutex); ... ; /* erreur */ }
}
```

Outils de synchronisation : sémaphores (3)

- Exemple 2 : k exemplaires d'une même ressource ($k \geq 2$)

```
Semaphore nb-ress = k;
P(nb-ress);
        /* section critique */
V(nb-ress);
```

- Exemple 3 : obtention de plusieurs ressources différentes (ex: un tampon de données et un segment de mémoire partagée)
  - $\Rightarrow$ 2 sémaphores et 2 opérations P

Outils de synchronisation : sémaphores (4)

- Mauvaise solution :

P1	P2
P(S1);	P(S2);
P(S2);	P(S1);
.....	.....
V(S2);	V(S1);
V(S1);	V(S2);

INTERBLOCAGE !!

Outils de synchronisation : sémaphores (5)

- Solutions :
  - ✔ Opérations P dans le même ordre
  - ✔ Opérations P et V sur tableaux de sémaphores de façon atomique

La programmation multithreadée (1)

- Définition
  - Thread : flot d'exécution interne à un processus
  - processus monoprogrammé : un seul point d'exécution dans le programme (un compteur ordinal)
  - processus multiprogrammé (ou multithreadé) : un point d'exécution dans chaque thread. (un compteur ordinal par flot d'exécution)

La programmation multithreadée (2)

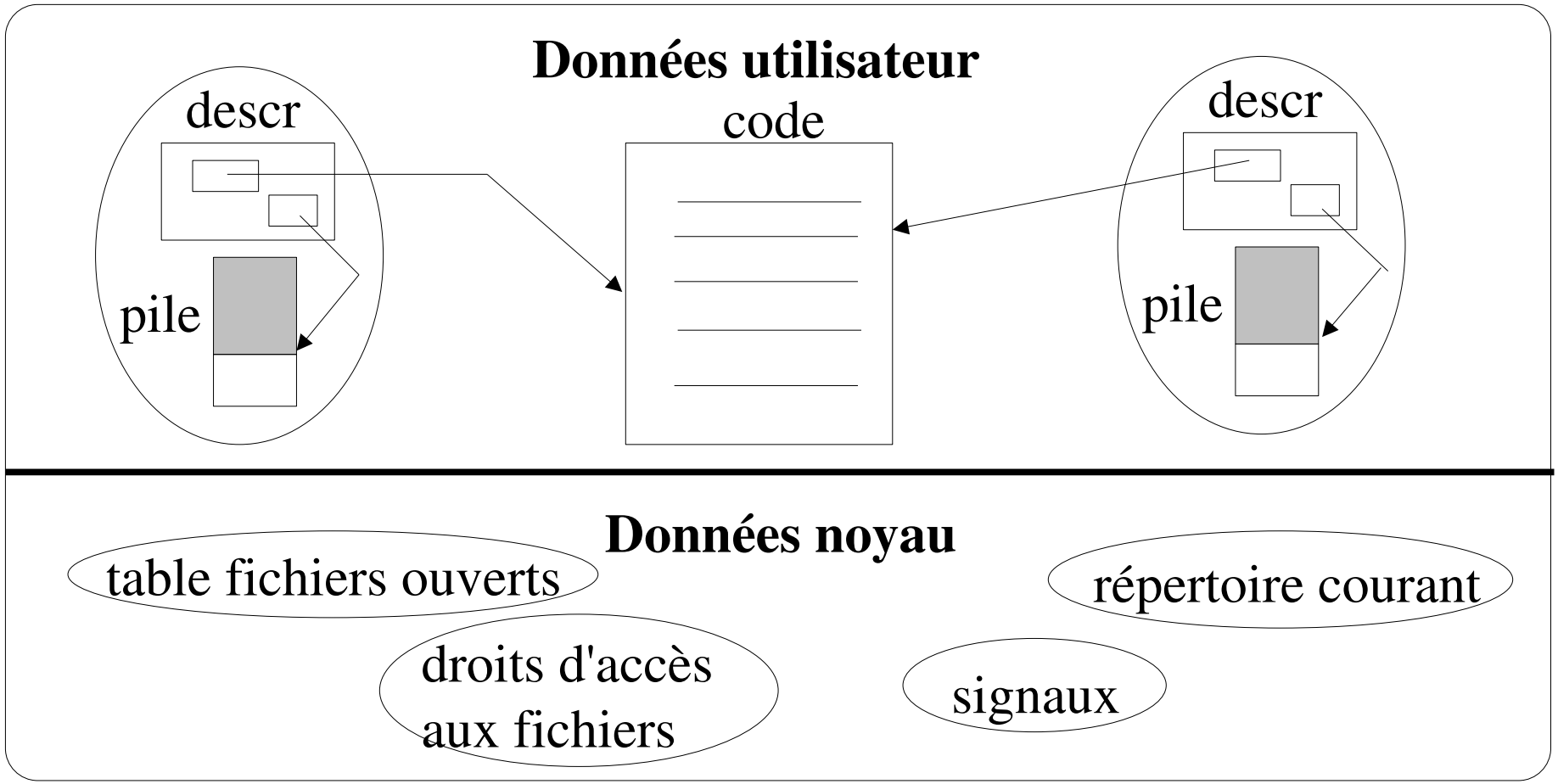
- caractéristiques
  - partage de ressources
    - ✔ le code (chaque thread possède un compteur ordinal et une pile d'exécution)
    - ✔ le tas
    - ✔ table des fichiers ouverts
    - ✔ table des traitements de signaux

    ⇒ taille processus >> taille ressources d'un thread

La programmation multithreadée (3)

- efficacité (% processus)
  - ✔ ex : création de thread ⇒
    - allocation descripteur & pile
    - initialisation du contexte d'exécution (affectation de registres)
    - insertion dans la file des processus prêts
  - ✔ ex : changement de contexte de thread : repositionnement de registres du processeur

Structure d'un processus bi-threadé (1)



Structure d'un processus bi-threadé (2)

- les ressources propres à la gestion des threads sont dans l'espace utilisateur
- le noyau garde un contrôle total sur les ressources de la machine
- chaque thread peut utiliser directement les ressources du processus englobant
- attention à l'intégrité des données partagées

Utilisation du multithreading (1)

Dans le cadre des systèmes monoprocesseur

- **augmenter la réactivité** : traitement concurrent de requêtes
  - exemple1 : serveurs
    - ✔ serveurs monoprogrammé :
      - accepter requête
      - traiter requête
    - ✔ serveurs multiprogrammés :
      - accepter requête
      - déléguer le traitement à un thread
      - ⇒ requête prise en compte au plus tôt

Utilisation du multithreading (2)

- exemple 2 : dans les systèmes d'exploitation
  - ✔ prise en charge des appels systèmes par des threads

- exemple 3 : dans les applications
  - ✔ attacher un thread à chaque élément de l'interface graphique pour augmenter sa réactivité
  - ✔ simulation : observer le comportement collectif d'activités complexes souvent asynchrones
  - ✔ jeux d'arcade : faire évoluer plusieurs entités actives en parallèle

Utilisation du multithreading (3)

- **recouvrement des entrées sorties** (et des communications inter-processus)
  - rappel : ordonnancement de processus
    - ✔ système temps partagé ⇒ un processus élu obtient un quantum de temps (processus actif)
    - ✔ commutation de processus sur :
      - quantum épuisé (retour à l'état « prêt à s'exécuter »)
      - attente de ressource (passage dans l'état « endormi »)
  - processus monoprogrammé
    - ✔ attente de ressource ⇒ préemption ⇒ commutation de processus

Utilisation du multithreading (4)

- processus multiprogrammé
  - ✔ un thread en attente de ressource ⇒
    - commutation de thread s'il existe un thread prêt à s'exécuter dans le processus,
    - commutation de processus sinon

- intérêt : diminution du nombre de commutations de processus

(remarque : temps de commutation de thread << temps de commutation de processus)

Les threads POSIX : la bibliothèque pthread (1)

- généralités
  - Fichier d'include: <pthread.h>
  - nom de fonction: pthread[_*objet*]_operation[_*np*] où
    - *objet* : type de l'objet auquel la fonction s'applique (ex: mutex, cond, ...)
    - opération: opération à réaliser (ex: create, exit, ...)
    - _*np* : fonction non portable
  - nom de type: pthread[_*objet*]_t
  - identification d'un thread: TID de type pthread_t

Création, Terminaison (1)

- **création d'un thread**

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void*(*start_routine) (void *), void *arg)

  - thread : tid du thread créé
  - attr : attributs du thread à créer ou NULL (attributs par défaut)
  - start_routine : fonction, point d'entrée du thread : fonction ayant un paramètre de type (void *) et qui retourne un (void *)
  - arg : argument de la fonction start_routine

Création, Terminaison (2)

- attributs d'un thread :
  - PTHREAD_CREATE_JOINABLE : thread joignable (synchronisation possible sur sa terminaison) (attribut par défaut)
  - PTHREAD_CREATE_DETACHED : thread détaché : pas de synchronisation possible sur sa terminaison

Création, Terminaison (3)

- ***synchronisation sur terminaison :***

  *int pthread_join(pthread_t tid, void **status)*

  D : tid

  R : status : code retour du thread terminé

  - ✔ Suspend l'exécution du thread appelant jusqu'à la terminaison du thread tid.
  - ✔ tid doit être joignable (ie non détaché).
  - ✔ Au plus un thread peut attendre la terminaison d'un thread donné

  rem : pas de libération des ressources mémoire (descripteur et pile) sur terminaison d'un thread joignable tant qu'un autre thread n'a pas effectué *pthread_join* ; **il doit donc y avoir un *pthread_join* pour chaque thread joignable** de façon à éviter la saturation mémoire.

Création, Terminaison (4)

- ***détachement de thread joignable :***

  int pthread_detach(pthread_t tid)

- **terminaison d'un thread :**

  *void pthread_exit (void *status)* :

  - terminaison du thread courant avec code retour
  - Remarques:
    - ✔ un appel UNIX exit termine tous les threads du processus
    - ✔ *pthread_exit* provoque la terminaison du processus s'il s'agit du dernier thread

threads joignables : exemple (1)

```
#include <stdio.h>
#include <pthread.h>


void *hello(void * arg) {
    int status;
    int num = * (int *)arg;
    printf("hello de thread %d\n", num);
    pthread_exit((void *)&status);
}
```

threads joignable : exemple (2)

```
main() {
    pthread_t tid1, tid2;
    int arg1=1, arg2=2;
    pthread_create(&tid1, NULL, hello, (void *) &arg1);
    pthread_create(&tid2, NULL, hello, (void *) &arg2);
    printf("j'attends la fin de tid1 et tid2\n");
    pthread_join(tid1, NULL);
    printf("tid1 est terminé\n");
    pthread_join(tid2, NULL);
    printf("fin du programme principal");
}
```

Threads détachés : exemple (1)

```
#include <stdio.h>
#include <pthread.h>


int arg1=1, arg2=2;


void *hello(void * arg) {
    int i, num = * (int *)arg;
    printf("hello de thread %d\n", num);
    printf("entrer une valeur pour terminer le thread\n");
    scanf("%d", &i);
}
```

Threads détachés : exemple (2)

```
main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, hello, &arg1);
    pthread_detach(tid1);
    pthread_create(&tid2, NULL, hello, &arg2);
    pthread_detach(tid2);
    printf("fin du thread principal\n");
    pthread_exit(NULL);
}
```

Threads détachés : exemple (3)

résultat d'exécution :

hello de thread 1

entrer une valeur pour terminer le thread

hello de thread 2

entrer une valeur pour terminer le thread

fin du thread principal

1 // valeur entrée pour le scanf

1 // valeur entrée pour le scanf

commentaires :

- le thread 1 est en attente sur le scanf
- le thread 2 prend la main et se met également en attente sur le scanf
- le thread principal prend la main et affiche son message
- l'entrée de 2 valeurs termine les 2 threads

Attributs d'un thread (1)

- **affectation des attributs d'un thread** : construire un objet *attr* de type *pthread_attr_t*, qui est passé en paramètre à la fonction *pthread_create*
  - *int pthread_attr_init (pthread_attr_t *attr);* initialisation par défaut d'un objet attribut.
  - fonctions *pthread_attr_set<attrname>*: affectation de l'attribut *attrname*
  - fonctions *pthread_attr_get<attrname>*: restauration de la valeur par défaut de *attrname*.

Thread détaché : autre exemple (1)

```
#include <stdio.h>
#include <pthread.h>


int arg1=1, arg2=2;


void *hello(void * arg) {
    int i,num = * (int *)arg;
    printf("hello de thread %d\n", num);
    printf("entrer une valeur pour terminer le thread\n");
    scanf("%d", &i);
}
```

Thread détaché : autre exemple (2)

```
main() {
    pthread_t tid1, tid2;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    pthread_create(&tid1, NULL, hello, &arg1);
    pthread_create(&tid2, NULL, hello, &arg2);
    printf("fin du thread principal\n");
    pthread_exit(NULL);
}
```

Pièges : exemple 1 (1)

```
void *hello(void * arg) {
    int i, num;
    scanf(« %d », &i);    // pour suspendre le thread
    num = * (int *)arg;
    printf("hello de thread %d\n", num);
}
main() {
    pthread_t tid1, tid2;
    int arg=1;
    pthread_create(&tid1, NULL, hello, (void *) &arg);
    arg = 2;
    pthread_create(&tid2, NULL, hello, (void *) &arg);
    pthread_join(tid1, NULL); pthread_join(tid2, NULL);
}
```

Pièges : exemple 1 (2)

- résultat d'exécution

```
1                          // entrée pour le scanf
hello de thread 2
1                          // entrée pour le scanf
hello de thread 2
fin du thread principal
```

- commentaires :
  - les threads sont en attente sur le scanf
  - le thread principal est en attente sur le join avec arg = 2
  - suite aux scanf, les threads reprennent la main avec num = *(int *)arg, c'est à dire avec num = 2

Pièges : exemple 2 (1)

```
void *hello(void * arg) {
    int i,num ;
    scanf("%d", &i);
    num = * (int *)arg;
    printf("hello de thread %d\n", num);
}
main() {
    pthread_t tid1, tid2;
    int arg1=1, arg2 = 2;
    pthread_create(&tid1, NULL, hello, &arg1);    pthread_detach(tid1);
    pthread_create(&tid2, NULL, hello, &arg2);    pthread_detach(tid2);
    printf("fin du thread principal\n"); pthread_exit(NULL);
}
```

Pièges : exemple 2 (2)

- résultat d'exécution

fin du thread principal

1

hello de thread 1075216656

1

hello de thread 2

- commentaires :
  - les threads sont en attente sur le scanf
  - le thread principal est affiche son message et se termine : les variables de bloc sont désallouées
  - suite aux scanf, les threads reprennent la main , l'adresse arg est une adresse illégale

solution à l'exemple 2 (1)

```
void *hello(void * arg) {
    int i,num ;
    scanf("%d", &i);
    num = * (int *)arg;  free(arg);
    printf("hello de thread %d\n", num);
}
main() {
    pthread_t tid1, tid2;
    int *arg1=(int *)malloc(sizeof(int));   int *arg2=(int *)malloc(sizeof(int));
    *arg1 = 1; *arg2 = 2;
    pthread_create(&tid1, NULL, hello, arg1);      pthread_detach(tid1);
    pthread_create(&tid2, NULL, hello, arg2);      pthread_detach(tid2);
    printf("fin du thread principal\n"); pthread_exit(NULL);
}
```

solution à l'exemple 2 (2)

- commentaires
  - les zones allouées dynamiquement sur le tas ne sont pas automatiquement désallouées au retour de fonction
  - libérer les zones allouées sur le tas après le déballage des arguments dans la fonction exécutée par le thread
  - autre solution : utilisation de variables globales (allouées sur la zone de données, désallouées en fin d'exécution du processus) (cf : page 34)

Pièges : exemple 3 (1)

```
void *hello(void * arg)  {
    int status;
    int num = * (int *)arg;
    printf("hello de thread %d\n", num);
}
pthread_t lanceThreadHello(int num) {
    pthread_t tid;
    pthread_create(&tid, NULL, hello, (void *) &num);
    return(tid);
}
```

Pièges : exemple 3 (2)

```
main() {
    pthread_t tid1, tid2;
    tid1=lanceThreadHello(1);
    tid2=lanceThreadHello(2);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("fin du programme principal");
}
```

☞ Commentaires :

- les paramètres de fonction sont alloués sur la pile à l'appel et désalloués au retour ; pthread_create prend en paramètre, une adresse devenue illégale suite à la terminaison de la fonction lanceThreadHello.

Solution à l'exemple précédent

```
void *hello(void * arg) {
    int status;
    int num = * (int *)arg;
    free(arg);                                  // libération explicite de la zone allouée
    printf("hello de thread %d\n", num);
}
pthread_t lanceThreadHello(int num){
    int * adnum = (int *) malloc(sizeof(int));
    pthread_t tid;
    *adnum = num;                               // num est sauvegardé sur le tas
    pthread_create(&tid, NULL, hello, (void *) adnum);
    return(tid);
}
```

Synchronisation de threads (1)

- trois mécanismes de synchronisation:
  - rendez-vous (pthread_join)
  - sémaphores d'exclusion mutuelle
  - les conditions (évènements)

Les sémaphores : bibliothèque pthread (1)

- type : *pthread_mutex_t*
- 2 états possibles : *unlocked* ou *locked* (ie : appartient a au plus un thread).
  - *int pthread_mutex_init(pthread_mutex_t *mutex,* const pthread_mutex_attr *attr)
    - Initialise *mutex* selon *attr* (si NULL : initialisation par défaut)
    - Dans Linux, un seul attribut possible : le type de mutex (fast, recursive ou error checking); determine si un mutex verrouillé peut être à nouveau verrouillé par son propriétaire

Les sémaphores : bibliothèque pthread (2)

- *int pthread_mutex_lock(pthread_mutex_t *mutex);*
  - ✔ verrouille mutex si possible sinon suspend le processus
  - ✔ cas particulier : mutex a été verrouillé par le thread appelant
    - fast : deadlock
    - error checking : retour immédiat de p*thread_mutex_lock* avec erreur
    - recursive : succès et retour de *pthread_mutex_lock* (il faudra autant de pthread_mutex_unlock)

Les sémaphores : bibliothèque pthread (3)

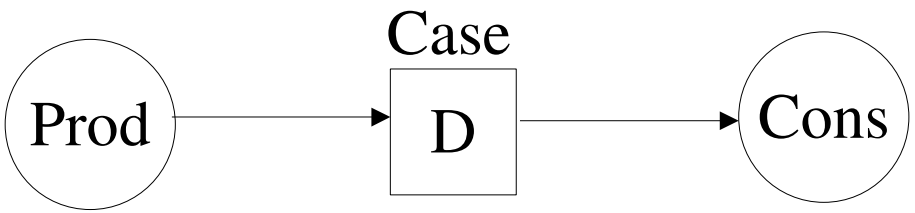
- *int pthread_mutex_unlock (pthread_mutex_t *mutex)*
  - ✔ déverrouille mutex (doit appartenir au thread appelant)
    - fast : retourne toujours; mutex déverrouillé
    - recursive : décrémentation du compteur de verrouillage et déverrouillage si compteur = 0
    - error checking : vérifie si le mutex a bien été préalablement verrouillé par le thread appelant
    - remarque : pour les mutex de type fast et recursive, pas de vérification sur le propriétaire de mutex; permet de déverrouiller un mutex bloqué par un autre thread mais attention : comportement non portable et donc, à éviter.)

Les sémaphores : bibliothèque pthread (4)

- *int pthread_mutex_destroy(pthread_mutex_t *mutex)*
  - ✔ détruit mutex qui doit être déverrouillé
  - ✔ rem : dans Linux, aucune ressource associée à un mutex et donc destroy n'a aucun effet si ce n'est de vérifier que le mutex est déverrouillé

Exemple : modèle producteur/consommateur (1)

- Solution à une case :



sémaphore plein = 0, vide = 1;

Producteur	Consommateur
produire(D);	P(Plein);
P(vide);	retirer(Case, D);
déposer(Case, D);	V(vide);
V(plein);	consommer(D);

Exemple : modèle producteur/consommateur (2)

```
#include <stdio.h>
#include <pthread.h>
#define N   5
int buf;      pthread_mutex_t plein, vide;
void * producteur(void *arg) {
    int i, data;        srand();
    for (i=0; i<N; i++){
        data = rand()%N;
        printf("production de %d\n", data);
        pthread_mutex_lock(&vide);
        buf = data;
        pthread_mutex_unlock(&plein);
        printf("%d déposée dans buf\n", data);
} }
```

```
void * consommateur(void *arg) {
    int i, data;
    for (i=0; i<N; i++){
        pthread_mutex_lock(&plein);
        data = buf;
        printf("%d prélevée dans buf\n", data);
        pthread_mutex_unlock(&vide);
        printf("consommation de %d\n", data);
    }
}
```

Exemple : modèle producteur/consommateur (3)

```
main() {
    pthread_t tid1, tid2;
    /* initialisation des sémaphores : plein verrouillé, vide non verrouillé */
    pthread_mutex_init(&plein, NULL);
    pthread_mutex_init(&vide, NULL);
    pthread_mutex_lock(&plein);

    pthread_create(&tid1, NULL, producteur, NULL);
    pthread_create(&tid2, NULL, consommateur, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("fin du thread principal\n");
}
```

Exemple : modèle producteur/consommateur (4)

- Résultat d'exécution

production de 3

3 déposée dans buf

production de 2

3 prélevée dans buf

consommation de 3

2 déposée dans buf

production de 0

2 prélevée dans buf

0 déposée dans buf

production de 3

consommation de 2

0 prélevée dans buf

3 déposée dans buf

production de 1

consommation de 0

3 prélevée dans buf

1 déposée dans buf

consommation de 3

1 prélevée dans buf

consommation de 1

fin du thread principal

Les variables conditionnelles (1)

- Définition : outil de synchronisation permettant à un thread de suspendre son exécution jusqu'à ce qu'un prédicat sur une variable partagée soit vérifié
- 2 opérations de base :
  - Cond-signal : signaler la condition c => réveil d'un thread en attente
  - Cond-wait : attendre la condition

Les variables conditionnelles (2)

- Une variable conditionnelle est associée à
  - Une ou plusieurs variables partagées **protégées par un mutex**
  - Un prédicat basé sur ces variables
- => Une variable conditionnelle est toujours associée à un sémaphore (gestion de l'accès exclusif à la variable partagée)
- Schéma de modification d'une condition :
  - Acquérir le mutex protégeant la condition
  - Positionner la condition en modifiant les variables partagées
  - Si le prédicat est satisfait, signaler la condition
  - Libérer le mutex

Les variables conditionnelles (3)

- Attendre une condition : primitive cond-wait(cond, mutex)
  - Implantation du cond-wait
    - ✔ V(mutex)
    - ✔ Suspension du processus
    - ✔ .....
    - ✔ P(mutex) au réveil
    - ✔

    Avec atomicité garantie
  - => verrouiller systématiquement le mutex avant le cond-wait

Les variables conditionnelles (4)

- Exemple : un ensemble de threads partagent deux variables x et y. Pour pouvoir effectuer un traitement, un thread doit vérifier la condition x>y.

Thread1 :

```
P(mutex)
    // modification de  x et y
Si x>y alors
    Cond-signal(cond)
Fsi
V(mutex)
.............
```

Thread2 :

```
P(mutex)
    Tant-Que x<=y Faire
            cond-wait(cond, mutex)
    Fin-Tant-Que
// utilisation de x et y
....
V(mutex)
```

- Remarque : la condition a pu être modifiée entre le réveil et le retour du wait => utilisation d'un *Tant-Que* et non d'un *Si*

Les variables conditionnelles : primitives (1)

- Type : pthread_cond_t
- Initialisation

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *cond_attr)

  - Rem : pas d'attribut de condition sous Linux => 2ème paramètre toujours NULL

- Signalement d'une condition

int pthread_cond_signal (pthread_cond_t *cond)

  - Réveil **d'un** thread en attente (sans effet sinon)

int pthread_cond_broadcast (pthread_cond_t *cond)

  - Réveil de **tous** les threads en attente sur cond

Les variables conditionnelles : primitives (2)

- Attente d'une condition

  int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t mutex)

  - Déverouille mutex, attend cond, et revérouille mutex

- Destruction

  int pthread_cond_destry (pthread_cond_t *cond)

  - Rem : destruction si et seulement si aucun thread n'est en attente sur cond
  - Aucune ressource associée sous Linux => destroy vérifie qu'il n'existe aucun thread en attente

Exemple : modèle producteur/consommateur (1)

```
#include <stdio.h>
#include <pthread.h>
#define N 5
    pthread_cond_t est_vide, est_plein;
    pthread_mutex_t mutex;
    int buffer, libre = 1;
void *produire(void * arg){
    int i, data;
    srand();
    for (i=0; i<N; i++) {
        data = rand()%N;
        printf("production de %d\n", data);
        pthread_mutex_lock(&mutex);
        while (!libre) {
            pthread_cond_wait(&est_vide, &mutex);
        }
        buffer = data;
        libre = 0;
        pthread_cond_signal(&est_plein);
        pthread_mutex_unlock(&mutex);
        printf("%d déposée dans buffer\n", data);
    }
}
```

Exemple : modèle producteur/consommateur (2)

```
void *consommer(void * arg){
    int i, data;
    for (i=0; i<N; i++) {
        pthread_mutex_lock(&mutex);
        while (libre)
        pthread_cond_wait(&est_plein, &mutex);
        data=buffer;
        printf("%d prélevée dans buf\n", data);
        libre = 1;
        pthread_cond_signal(&est_vide);
        pthread_mutex_unlock(&mutex);
        printf("consommation de %d\n", data);
    }
}
```

```
main() {
    pthread_t tid1, tid2;
    pthread_cond_init (&est_vide, NULL);
    pthread_cond_init(&est_plein, NULL);
    pthread_mutex_init(&mutex, NULL);

    pthread_create(&tid1, NULL, produire, NULL);
    pthread_create(&tid2, NULL, consommer,
        NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
}
```

Exemple : modèle producteur/consommateur (3)

- Résultat d'exécution

production de 3

3 déposée dans buffer

production de 2

3 prélevée dans buf

consommation de 3

2 déposée dans buffer

production de 0

2 prélevée dans buf

0 déposée dans buffer

production de 3

consommation de 2

0 prélevée dans buf

3 déposée dans buffer

production de 1

consommation de 0

3 prélevée dans buf

1 déposée dans buffer

consommation de 3

1 prélevée dans buf

consommation de 1