

# Introduction à la programmation concurrente

## Verrous et sémaphores

Yann Thoma, Jonas Chapuis

Reconfigurable and Embedded Digital Systems Institute  
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

Février 2018

# Verrous

- L'attente active a un énorme désavantage:
  - Réquisition du processeur
- Solution: les verrous
- Le verrou
  - est une variable booléenne
  - possède une liste d'attente
  - est manipulé par deux opérations *atomiques*:
    - Verrouille(v)
    - Deverrouille(v)
- En anglais: *mutex*, pour mutual exclusion

# Verrous

- Pseudocode des deux opérations

```
void Verrouille(verrou v)
{
    if (v)
        v = false;
    else
        suspendre la tâche appelante dans la file associée à v
}

void Déverrouille(verrou v)
{
    if (la file associée à v != vide)
        débloquer une tâche en attente dans file
    else
        v = true;
}
```

# Verrous - Points importants

- Un mutex possède un "propriétaire" : le thread ayant obtenu le verrou est le propriétaire du mutex jusqu'à ce qu'il le déverrouille :
  - ⇒ le thread ayant verrouillé le mutex est responsable de le déverrouiller (relâcher); un thread ne peut pas déverrouiller un mutex déjà verrouillé par un autre thread.
- Les fonctions lock et unlock sont atomiques !
- Verrouiller un mutex déjà verrouillé bloque le thread appelant ⇒ deadlock assuré si un même thread verrouille un mutex de façon consécutive (sauf pour les mutex rékursifs).
- Déverrouiller un mutex plusieurs fois n'a pas d'effet ⇒ pas de "mémoire" du nombre de fois qu'un mutex a été déverrouillé (sauf pour les mutex rékursifs).

# Utilisation des verrous: Section critique

- Une section critique peut être protégée par un verrou
- Pour un nombre quelconque de tâches

```
⋮  
Verrouille(v);  
/* section critique */  
Déverrouille(v);  
⋮
```

# Mauvaise utilisation des verrous

- Un thread ne peut pas déverrouiller un verrou déjà verrouillé par un autre thread!

## Illégal

```
mutex m;  
  
void *thread_A(void *p) {  
    unlock(m);  
}  
  
int main() {  
    mutex_init(&m);  
    lock(&m);  
    create_thread(thread_A);  
}
```

# Verrous Qt

- Une classe verrou: **QMutex**
- Utilisation : **#include<QMutex>**
- Des fonctions
  - **lock()**
  - **unlock()**
  - **tryLock(int timeout = 0)**
- Initialisation:
  - Constructeur:  
**QMutex**(RecursionMode mode = NonRecursive)

# Pour la culture: Verrous POSIX

- Un type verrou: `pthread_mutex_t`
- Des fonctions
  - `pthread_mutex_lock(pthread_mutex_t *mutex)`
  - `pthread_mutex_unlock(pthread_mutex_t *mutex)`
  - `pthread_mutex_trylock(pthread_mutex_t *mutex)`
- Initialisation:
  - `pthread_mutex_init()`
  - `pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;`



# Verrous Qt: Initialisation

- Déclaration et initialisation faites de manière implicite via le constructeur:

```
QMutex mutex;  
QMutex mutex(QMutex::Recursive); ← Mutex récursif
```

- Après initialisation un mutex est toujours déverrouillé

# Verrous Qt: Verrouillage

```
void QMutex::lock();
```

- Si le mutex est déverrouillé, il devient verrouillé et son propriétaire est alors le thread appelant.
- Mutex récursif: peut être verrouillé plusieurs fois; un compteur mémorise le nombre de verrouillages effectués.
- Mutex non récursif: Si un thread tente de reverrouiller un mutex verrouillé par lui il y a deadlock.
- Si le mutex est déjà verrouillé par un autre thread, le thread appelant est suspendu jusqu'à ce que le mutex soit déverrouillé.

# Verrous Qt: Verrouillage

```
bool QMutex::tryLock(int timeout = 0);
```

- Même comportement que **lock**, sauf qu'il n'y a suspension du thread appelant seulement pendant `timeout` millisecondes
- La fonction retourne **true** si le verrou a été acquis et **false** s'il est possédé par un autre thread.
- Dans le cadre du cours, nous laisserons `timeout` à 0.

# Verrous Qt: Déverrouillage

```
void QMutex::unlock();
```

- Déverrouille (libère) le mutex; le mutex est supposé être verrouillé par le thread appelant avant l'appel à **unlock** () .

# Verrous Qt: exemple

```
static int global=0;
static int iterations = 1000000;

class MyThread : public QThread {
    void run() {
        for(int i=0;i<iterations;i++) {

            global = global + 1;

        }
    }
};

int main(int /*argc*/, char **/*argv*/[]) {
    MyThread tache1, tache2;
    tache1.start();
    tache2.start();
    tache1.wait();
    tache2.wait();
    std::cout << "Fin des taches : global = " << global << " ("
               << 2*iterations << ") " << std::endl;
    return 0;
}
```



# Verrous Qt: exemple

```
static QMutex mutex;
static int global=0;

class MyThread : public QThread {
    void run() {
        for(int i=0; i<1000000; i++) {
            if (mutex.tryLock()) {
                global = global + 1;
                std::cout << "I got the mutex" << std::endl;
                mutex.unlock(); ← Ne pas oublier
            }
            else
                std::cout << "I cannot get the mutex" << std::endl;
        }
    }
};

int main(void) {
    MyThread tache1, tache2;
    tache1.start();
    tache2.start();
    tache1.wait();
    tache2.wait();
    std::cout << "Fin des taches : global = " << global << " ("
        << 2*iterations << ")" << std::endl;
    return 0;
}
```



# Récuratif vs Non Récuratif

## Non récuratif

```
void function() {  
    QMutex mutex;  
    mutex.lock();  
    mutex.lock(); ← Blocage  
  
    mutex.unlock();  
    mutex.unlock();  
}
```

## Récuratif

```
void function() {  
    QMutex mutex(QMutex::Recursive);  
    mutex.lock();  
    mutex.lock(); ← Pas de blocage  
  
    mutex.unlock();  
    mutex.unlock();  
}
```

# Attention

- Un verrou doit être libéré par le thread qui l'a acquis!!
- La solution pour la synchronisation est d'utiliser des sémaphores...



# Sémaphores

- Les sémaphores sont une généralisation des verrous
- Proposés par Dijkstra en 62 ou 63 (article en hollandais)
- Comprennent une variable entière plutôt qu'un booléen
- Opérations d'accès (atomiques):
  - $P(s)$  (pour tester: Prolaag (probeer te verlagen: essayer de réduire))
  - $V(s)$  (pour incrémenter: Verhoog)

# Sémaphores

- Pseudocode des deux opérations

```
void P(sémaphore s)
{
    s -= 1;
    if (s < 0)
        suspendre la tâche appelante dans la file associée à s
}

void V(sémaphore s)
{
    s += 1;
    if (s <= 0)
        débloquer une des tâches de la file associée à s
}
```

# Section critique

- Une section critique peut être protégée par un sémaphore
- Pour un nombre quelconque de tâches
- Un sémaphore *mutex* initialisé à 1

```
⋮  
P(mutex);  
/* section critique */  
V(mutex);  
⋮
```

- Sémaphore *mutex* initialisé à  $v > 0$
- jusqu'à  $v$  tâches peuvent être admises simultanément dans la section critique
- $\Rightarrow$  *section contrôlée*

# Sémaphores Qt

- Une classe sémaphore: **QSemaphore**
- Utilisation : **#include<QSemaphore>**
- Des fonctions
  - Constructeur: **QSemaphore (int n=0)**
  - **acquire (int n=1)**
  - **release (int n=1)**
- A éviter:
  - **tryAcquire (int n=1)**
  - **int available () const**

# Pour la culture: Sémaphores Posix

- Un type sémaphore: `sem_t`
- Des fonctions
  - `sem_init(sem_t *sem, int pshare, unsigned int value)`
  - `sem_destroy(sem_t *sem)`
  - `sem_wait(sem_t *sem)`
  - `sem_post(sem_t *sem)`
- A éviter:
  - `sem_trywait(sem_t *sem)`
  - `sem_getvalue(sem_t *sem, int *sval)`
- Fichier à inclure: **#include** `<semaphore.h>`

# Sémaphores: Création et destruction

## Constructeur

```
QSemaphore::QSemaphore(int n = 0);
```

- Crée un sémaphore et l'initialise à la valeur spécifiée;



Dans le cadre du cours, interdiction de l'initialiser à une valeur négative

# Sémaphores: Attente

```
void acquire(int n=1);
```

- Décrémente (bloque) le sémaphore spécifié. Si la valeur du sémaphore est  $> 0$ , sa valeur est décrémentée et la fonction retourne immédiatement;
- Si sa valeur est égale à 0, alors l'appel bloque jusqu'à ce que sa valeur devienne  $> 0$ ;



Dans le cadre du cours, interdiction d'utiliser  $n > 1$ .

# Sémaphores: Attente

```
bool tryAcquire(int n=1);
```

- Identique à **acquire**, sauf que si la décrémentation ne peut pas avoir lieu, la fonction renvoie **false** au lieu de bloquer
- Renvoie **true** en cas de succès et **false** si la valeur du sémaphore est plus petite ou égale à 0.



Dans le cadre du cours, interdiction d'utiliser  $n > 1$ .



Nous n'utiliserons pas cette fonction dans ce cours.



# Sémaphores: Signalisation

```
void release(int n=1);
```

- Incrémente (débloque) le sémaphore spécifié;
- Si la valeur du sémaphore devient  $> 0$ , alors un autre thread bloqué dans un appel à **acquire** sera débloqué de la file d'attente et pourra effectuer l'opération de décrémentation (déblocage);



Dans le cadre du cours nous n'utiliserons pas de valeur de  $n > 1$ .

# Sémaphores: exemple



```
#define NUM_THREADS 4
QSemaphore *sem;

class MyThread : public QThread
{
    int tid;
public:
    MyThread(int id) : tid(id) {};
    void run() {
        sem->acquire();
        cout << "Tache " << tid << " est rentrée en SC" << endl;
        sleep((int)((float)3*rand()/(RAND_MAX+1.0)));
        cout << "Tache " << tid << " sort de la SC" << endl;
        sem->release();
    }
};

int main(void)
{
    int i;
    QThread *threads[NUM_THREADS];
    sem = new QSemaphore(2);
    for (i = 0; i < NUM_THREADS; i++) {
        threads[i] = new MyThread(i);
        threads[i]->start();
    }
    for (i = 0; i < NUM_THREADS; i++)
        threads[i]->wait();
    return EXIT_SUCCESS;
}
```


# Exemple: gestion des imprimantes

- Gestion de  $N$  imprimantes
- Un objet responsable d'allouer une imprimante à un job d'impression
- $N$  allocations possibles avant blocage;

```
void test() {  
    PrinterManager *manager = new PrinterManager(5); // 5 printers  
  
    int index1 = manager->allocate();  
    int index2 = manager->allocate();  
    // Do some printing  
    manager->giveBack(index2);  
    manager->giveBack(index1);  
}
```

# Exemple: gestion des imprimantes

## Gestion de $N$ imprimantes



```
class PrinterManager {
    QSemaphore *nbPrintersSem;
    bool *occupee;

public:

    PrinterManager(int nbPrinters) {
        nbPrintersSem = new QSemaphore(nbPrinters);
        occupee = new bool[nbPrinters];
        for(int i = 0; i < nbPrinters; i++)
            occupee[i] = false;
    }
    ~PrinterManager() {
        delete nbPrintersSem;
        delete[] occupee;
    }

    ...
}
```

```
unsigned giveIndex(void) {
    unsigned compteur = 0;
    // Recherche de la 1ère imprimante libre
    while (occupee[compteur])
        compteur += 1;
    occupee[compteur] = true;
    return compteur;
}


unsigned allocate(void) {
    nbPrintersSem->acquire();
    return giveIndex();
}

void giveBack(unsigned index) {
    occupee[index] = false;
    nbPrintersSem->release();
}
};
```

# Exemple: gestion des imprimantes

- Gestion de  $N$  imprimantes
- `nbPrintersSem` est initialisé à  $N$ ;
- $N$  allocations possibles avant blocage;
- exécution concurrente de la fonction `giveIndex` :
  - problèmes d'exclusion mutuelle sur le tableau `occupee`;
  - mettre `giveIndex` en section critique.

# Exemple: gestion des imprimantes (solution correcte)



```
class PrinterManager {
    QSemaphore *nbPrintersSem;
    bool *occupee;
    QSemaphore *mutex;

public:

    PrinterManager(int nbPrinters) {
        nbPrintersSem = new QSemaphore(nbPrinters);
        mutex = new QSemaphore(1);
        occupee = new bool[nbPrinters];
        for(int i = 0; i < nbPrinters; i++)
            occupee[i] = false;
    }
    ~PrinterManager() {
        delete nbPrintersSem;
        delete mutex;
        delete[] occupee;
    }

    ...
}
```

```
unsigned giveIndex(void) {
    unsigned compteur = 0;
    mutex->acquire();
    // Recherche de la 1ère imprimante libre
    while (occupee[compteur])
        compteur += 1;
    occupee[compteur] = true;
    mutex->release();
    return compteur;
}


unsigned allocate(void) {
    nbPrintersSem->acquire();
    return giveIndex();
}

void giveBack(unsigned index) {
    occupee[index] = false;
    nbPrintersSem->release();
}

};
```

# Exemple: gestion des imprimantes

Cette solution est-elle correcte?



```
class PrinterManager {
    QSemaphore *nbPrintersSem;
    bool *occupee;
    QSemaphore *mutex;

public:

    PrinterManager(int nbPrinters) {
        nbPrintersSem = new QSemaphore(nbPrinters);
        mutex = new QSemaphore(1);
        occupee = new bool[nbPrinters];
    }

    ~PrinterManager() {
        delete nbPrintersSem;
        delete mutex;
        delete[] occupee;
    }

    ...
}
```

```
unsigned giveIndex(void) {
    unsigned compteur = 0;
    // Recherche de la 1ère imprimante libre
    while (occupee[compteur])
        compteur += 1;
    occupee[compteur] = true;
    mutex->release();
    return compteur;
}

unsigned allocate(void) {
    mutex->acquire();
    nbPrintersSem->acquire();
    return giveIndex();
}

void giveBack(unsigned index) {
    occupee[index] = false;
    nbPrintersSem->release();
}
};
```

# Coordination de tâches

- Les sémaphores peuvent servir à coordonner/synchroniser des tâches
- Exemple:
  - La tâche  $T1$  exécute les instructions  $I_1$
  - La tâche  $T2$  exécute les instructions  $I_2$
  - L'exécution de  $I_1$  doit se faire avant  $I_2$



# Coordination de tâches: exemple

```
void T1::run() {
    std::cout << "T1: I1" << std::endl;    // I_1
    return;
}

void T2::run() {
    std::cout << "T2: I2" << std::endl;    // I_2
    return;
}

int main(void) {
    T1 tache1; T2 tache2;
    tache1.start();
    tache2.start();
    tache1.wait();
    tache2.wait();
    return 0;
}
```



# Coordination de tâches: exemple

```
static QSemaphore sync(0);

void T1::run() {
    sleep(1); // I_1
    std::cout << "T1: fini sleep\n";
    sync.release();
}

void T2::run() {
    std::cout << "T2: avant acquire\n";
    sync.acquire();
    std::cout << "T2: apres acquire\n"; // I_2
}

int main(void) {
    T1 tache1; T2 tache2;
    tache1.start();
    tache2.start();
    tache1.wait();
    tache2.wait();
    return 0;
}
```



# Rendez-vous

- Deux tâches se donnent rendez-vous entre deux activités respectives

	<i>TâcheA</i>	<i>TâcheB</i>
Premières activités	a1	b1
Rendez-vous		×
Deuxièmes activités	a2	b2

# Première solution

```
static QSemaphore arriveA(0);  
static QSemaphore arriveB(0);
```

```
void MyThreadA::run() {  
    a1;  
    arriveB.acquire();  
    arriveA.release();  
    a2;  
}
```

```
void MyThreadB::run() {  
    b1;  
    arriveB.release();  
    arriveA.acquire();  
    b2;  
}
```

```
int main(void) {  
    MyThreadA tacheA;  
    MyThreadB tacheB;  
    tacheA.start();  
    tacheB.start();  
    tacheA.wait();  
    tacheB.wait();  
    return 0;  
}
```



# Deuxième solution

```
...  
void MyThreadA::run() {  
    a1();  
    arriveA.release();  
    arriveB.acquire();  
    a2();  
}  
  
void MyThreadB::run() {  
    b1();  
    arriveB.release();  
    arriveA.acquire();  
    b2();  
}  
...
```



- Comparaison des deux solutions...

# Exercices

- 1 Comment implémentez-vous un verrou si vous ne disposez que de sémaphores? La sémantique du verrou doit évidemment être préservée.
- 2 Commentez la différence entre les deux codes suivants:

```
void *tache(void *) {  
    QMutex mutex;  
    mutex.acquire();  
    std::cout << "Section critique"  
               << std::endl;  
    mutex.release();  
}
```

```
void *tache(void *) {  
    static QMutex mutex;  
    mutex.acquire();  
    std::cout << "Section critique"  
               << std::endl;  
    mutex.release();  
}
```

# Exercice

Nous désirons réaliser une application possédant 2 tâches. Le programme principal est en charge de lancer les deux tâches.

Etant donné que les tâches, une fois lancées, doivent attendre un signal du programme principal pour s'exécuter, comment résoudre le problème à l'aide de sémaphores?

# Code source

[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex\\_example.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex_example.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex\\_example2.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex_example2.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex\\_example3.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/mutex_example3.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/semaphore\\_exemple.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/semaphore_exemple.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer\\_manager.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer_manager.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer\\_manager2.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer_manager2.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer\\_manager3.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/printer_manager3.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task\\_coordination.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task_coordination.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task\\_coordination2.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task_coordination2.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task\\_coordination3.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task_coordination3.tar.gz)  
[http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task\\_coordination4.tar.gz](http://reds.heig-vd.ch/share/cours/PCO/cours/code/4-mutex/task_coordination4.tar.gz)