

Introduction à la programmation concurrente

Tâches et QThread

Yann Thoma, Jonas Chapuis

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



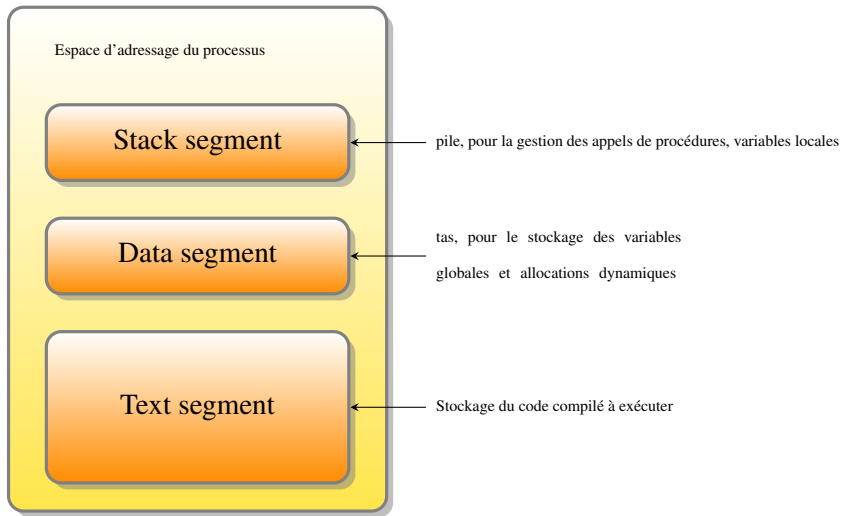
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License

Février 2018

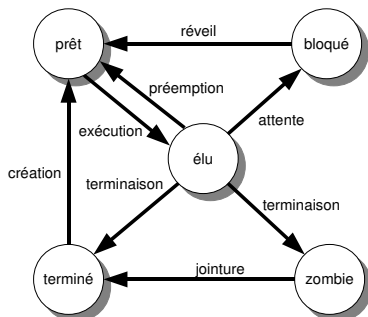
Anatomie d'un processus

- Un processus possède entre autre
 - Un code à exécuter
 - Un espace d'adressage
 - Une priorité
 - Un identifiant
 - Un contexte d'exécution (PC + registres)
- Les processus sont gérés par le système d'exploitation
- Plusieurs processus peuvent s'exécuter en parallèle

Espace d'adressage d'un processus



Etats et transitions d'un processus Unix



| Etat | Description |
|--------------|--|
| Prêt | Le processus est prêt à être exécuté. Cas d'un processus nouvellement créé, débloqué ou, d'un ou plusieurs processus occupant le ou les processeurs disponibles. |
| En exécution | Le processus est en cours d'exécution sur un processeur. Plusieurs processus peuvent être en exécution dans le cas d'une machine multiprocesseur. |
| Bloqué | Le processus est en attente sur une synchronisation ou sur la fin d'une opération d'entrée/sortie par exemple. |
| Zombie | Le processus a terminé son exécution, mais son processus parent doit encore récupérer sa valeur de terminaison. |
| Terminé | Le processus a terminé son exécution ou a été annulé (cancelled). Les ressources du processus seront libérées et le processus disparaîtra. |

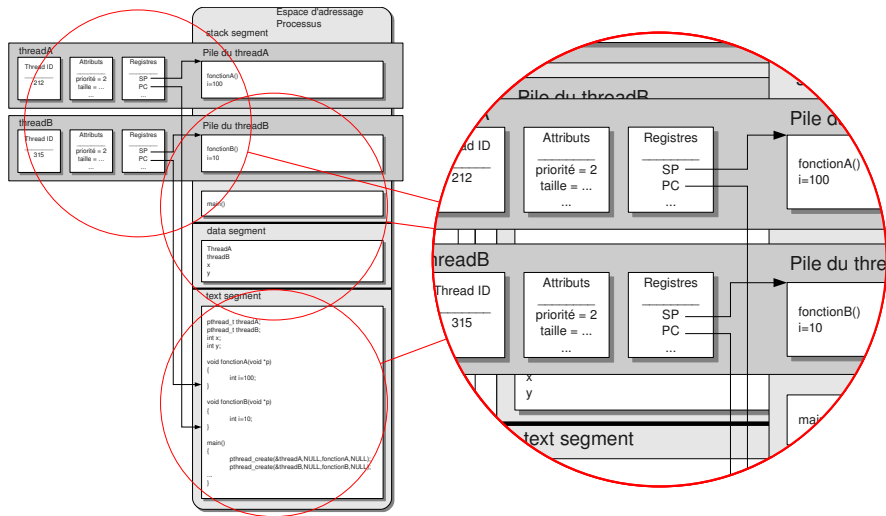
Anatomie d'un thread

Définition

Un thread est un fil d'exécution dans un processus

- Les threads d'un même processus se partagent l'espace d'adressage du processus
- Ils sont ordonnancés
- Ils possèdent
 - leur propre pile
 - leur propre contexte d'exécution (PC + registres)
- Ils ont un cycle de vie semblable à celui d'un processus

Espace d'adressage d'un processus multi-thread



Thread-processus: en commun

| Processus et Thread |
|--|
| Possèdent un ID, un ensemble de registres, un état, et une priorité |
| Possèdent un bloc d'information |
| Partagent des ressources avec les processus parents |
| Sont des entités indépendantes, une fois créés |
| Les créateurs de processus et thread ont contrôle sur eux |
| Peuvent changer leurs attributs après création, et créer de nouvelles ressources |
| Ne peuvent accéder aux ressources d'autres threads et processus non reliés |

Thread-processus: non commun

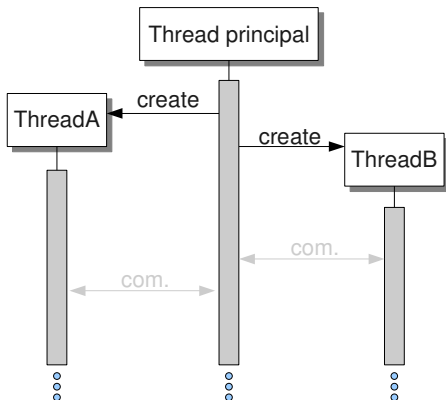
| Processus | Thread |
|---|--|
| Propre espace d'adressage | Pas d'espace d'adressage propre |
| Les processus parents et enfants doivent utiliser les mécanismes de communication inter-processus | Les threads d'un même processus communiquent en lisant et modifiant les variables de leur processus |
| Les processus enfants n'ont aucun contrôle sur les autres processus enfants | Les threads d'un processus sont considérés comme des pairs, et peuvent exercer un contrôle sur les autres threads du processus |
| Les processus enfants ne peuvent pas exercer de contrôle sur le processus parent | N'importe quel thread peut exercer un contrôle sur le thread principal, et donc sur le processus entier |

Changement de contexte d'exécution

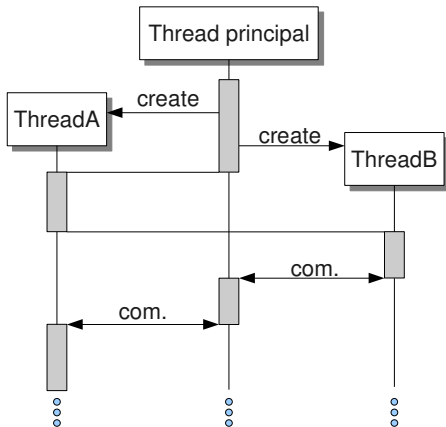
- L'opération de changement de contexte d'un processus (ou thread) comporte les séquences suivantes :
- Mise en attente du processus actif dans la liste des processus bloqués ou prêts (fin du quantum de temps)
- Sauvegarde de son contexte d'exécution
- Recherche du processus éligible ayant la plus haute priorité
- Restauration du contexte d'exécution du processus élu \Rightarrow restauration de la valeurs de ses registres lorsqu'il s'exécutait précédemment
- Activation du processus élu

Tout se passe comme si le processus préalablement interrompu n'avait pas cessé de s'exécuter

Flot d'exécution d'un processus multi-thread




Flot d'exécution d'un processus multi-thread sur un processeur simple coeur



Bibliothèque Concurrente Qt

- La concurrence sous Qt est incluse dans QtCore, rien de plus n'est nécessaire
- La classe représentant un thread:
QThread
- Deux possibilités d'exploiter cette classe:
 - En déplaçant un objet dans le thread et en exploitant les signaux et slots Qt (pas utilisé dans le cours)
 - En sous-classant **QThread** et en implémentant la méthode **void run()**

Déclaration d'un thread

```
class MyThread : public QThread
{
    private:
        virtual void run() Q_DECL_OVERRIDE  Vérifie l'override
            // Faire quelque chose
        }
};
```

- La méthode **run ()** sera appelée lors du lancement du thread
- L'utilisation de la macro **Q_DECL_OVERRIDE** permet au compilateur de vérifier que la méthode existe bien dans une des super classes (si le compilateur supporte ce test (C++11))

Passage d'arguments (1)

- Deux méthodes:
 - Via un constructeur
 - Via des accesseurs

Passage d'arguments via un constructeur

```
class MyThread : public QThread
{
    private:
        ArgType1 argument1;
        ArgType2 argument2;
        virtual void run() Q_DECL_OVERRIDE {
            // Faire quelque chose
        }
    public:
        MyThread(ArgType1 arg1, ArgType2 arg2)
            : QThread(), argument1(arg1), argument2(arg2)
        {
        };
};
```

Passage d'arguments (2)

- Deux méthodes:
 - Via un constructeur
 - Via des accesseurs

Passage d'arguments via des accesseurs

```
class MyThread : public QThread
{
    private:
        ArgType1 argument1;
        ArgType2 argument2;
        virtual void run() Q_DECL_OVERRIDE {
            // Faire quelque chose
        }
    public:
        setArgument1(ArgType1 arg1) {
            argument1 = arg1;
        }
        setArgument2(ArgType1 arg2) {
            argument2 = arg2;
        }
};
```

Création et lancement d'un thread


Exemple

```
int main(int argc, char *argv[]) {  
    MyThread thread;  
    ArgType1 arg1 = ...;  
    ArgType2 arg2 = ...;  
    thread.setArgument1(arg1);  
    thread.setArgument2(arg2);  
    thread.start(); // Lance l'exécution du thread  
    thread.wait(); // Attend la terminaison du thread  
    return 0;  
}
```


Jointure

- La jointure permet à un thread d'attendre qu'un autre se termine
- La jointure se fait via une fonction bloquante
 - Attente jusqu'à ce que le thread "joint" se termine
 - Forme rudimentaire de communication inter-threads
 - Utilisation de la fonction **QThread::wait** ()
 - Attend que la tâche en paramètre se termine
 - Le thread appelant est bloqué jusqu'à la terminaison du thread spécifié

Jointure : Exemple (1)



```

typedef struct {
    int a;
    int b;
} struct_t;

class MyThread1 : public QThread {
private:
    struct_t *var;
    void run() Q_DECL_OVERRIDE {
        std::cout << "Task 1: a= " << var->a
                    << ", b= " << var->b << std::endl;
    }
public:
    MyThread1(struct_t *arg) : QThread(),
        var(arg) {};
};

class MyThread2 : public QThread {
private:
    int aValue;
    void run() Q_DECL_OVERRIDE {
        std::cout << "Task 2: value= "
                    << aValue << std::endl;
    }
public:
    MyThread2(int value) : QThread(),
        aValue(value) {};
};

```

```

int main(int argc, char *argv[])
{
    struct_t v;
    v.a = 1; v.b = 2;


    MyThread1 thread1(&v);
    thread1.start();
    thread1.wait();

    MyThread2 thread2(3);
    thread2.start();
    thread2.wait();

    return 0;
}

```

Jointure : Exemple (2)



```

typedef struct {
    int a;
    int b;
} struct_t;

class MyThread1 : public QThread {
private:
    struct_t *var;
    void run() Q_DECL_OVERRIDE {
        std::cout << "Task 1: a= " << var->a
                    << ", b= " << var->b << std::endl;
    }
public:
    MyThread1(struct_t *arg) : QThread(),
        var(arg) {};
};

class MyThread2 : public QThread {
private:
    int aValue;
    void run() Q_DECL_OVERRIDE {
        std::cout << "Task 2: value= "
                    << aValue << std::endl;
    }
public:
    MyThread2(int value) : QThread(),
        aValue(value) {};
};

```

```

int main(int argc, char *argv[])
{
    struct_t v;
    v.a = 1; v.b = 2;

    MyThread1 thread1(&v);
    thread1.start();


    MyThread2 thread2(3);
    thread2.start();

    thread1.wait();
    thread2.wait();

    return 0;
}


```

Terminaison: options

- La terminaison d'un thread peut être exécutée depuis:
 - Le thread lui-même:
 - **return**
 - Un autre thread:
 - **QThread::terminate()**
 - **QThread::requestInterruption()**
-  Mal terminer un thread peut laisser le système dans un état incohérent!!
 - Plus spécifiquement depuis un autre thread
- Pour terminer l'application (destruction de tous les threads):
 - **exit()**

Auto-terminaison

- **return;**

- La fonction met fin au thread immédiatement
- Réveille le thread qui serait en attente sur un **wait** ()
-  Attention avec le thread principal: Terminaison du programme!!

Auto-terminaison: exemple (1)



```
class MyThread: public QThread
{
    void run() {
        int i = 0;
        while (1) {
            std::cout << "Coucou" << std::endl;
            i ++;
            if (i > 100) {
                return; ← Dans quel état se trouve le thread ensuite?
            }
        }
    }
};

int main(int /*argc*/, char **/*argv*/) {
    MyThread thread1;
    thread1.start();

    return 0; ← Attention
}
```

Auto-terminaison: exemple (2)



```
class MyThread: public QThread
{
    void run() {
        int i = 0;
        while (1) {
            std::cout << "Coucou" << std::endl;
            i ++;
            if (i > 100) {
                return;
            }
        }
    }
};

int main(int /*argc*/,char **/*argv*/[]) {
    MyThread thread1;
    thread1.start();
    thread1.wait();

    return 0;
}
```

Terminaison par un autre thread

QThread::terminate() ;

- Cette fonction permet à un thread de se faire terminer par un autre



Le thread peut se terminer n'importe quand (lié à l'ordonnanceur)

⇒ Il peut se trouver dans un état incohérent, et laisser le système dans un état incohérent!

- A n'appeler que si l'on est sûrs que le thread laissera le système dans un état cohérent
- Donc pas...
- Et semble ne pas bien fonctionner (en tout cas sous Linux, si pas de point d'annulation)

Terminaison par un autre thread

```
QThread::setTerminationEnabled(bool enabled = true);
```

- Cette fonction permet au thread de contrôler sa réponse à un **terminate()** initié par un autre thread
- Enabled:
 - Le thread se termine immédiatement à l'appel de **terminate()**
- Disabled:
 - Le thread ne se termine pas, mais le sera s'il réactive l'autorisation de terminaison

Cette fonction permet donc d'utiliser **terminate()** de manière sûre, pour autant que les autorisations soient bien gérées

Annulation par un autre thread

```
QThread::requestInterruption() ;
```

- Cette fonction demande au thread de se terminer
- Le thread peut tester les demandes d'annulation avec:

```
bool QThread::isInterruptionRequested() const;
```

- Ceci permet un contrôle fin des moments où le thread est prêt à se terminer
- Evite de laisser le système dans un état incohérent

Exemple (1)



```
static int counter1 = 1;

class MyThread: public QThread
{
    void run() {
        while (true) {
            counter1++;
            if (counter1 % 100 == 0)
                if (isInterruptedRequested())
                    return;
        }
    }
};

int main(int /*argc*/, char **/*argv*/[]) {
    MyThread thread;
    thread.start();
    QThread::usleep(5000);
    thread.requestInterrupted();
    thread.wait();
    std::cout << "Counter1: " << counter1 << std::endl;
    return 0;
}
```

Exemple (2)



```
static int counter1 = 1;

class MyThread: public QThread
{
    void run() {
        setTerminationEnabled(true);
        while (true) {
            counter1++;
        }
    }
};

int main(int /*argc*/, char **/*argv*/[]) {
    MyThread thread;
    thread.start();
    QThread::usleep(5000);
    thread.terminate();
    thread.wait();
    std::cout << "Counter1: " << counter1 << std::endl;
    return 0;
}
```

Exemple (3)



```
static int counter1 = 1;


class MyThread: public QThread
{
    void run() {
        setTerminationEnabled(false);
        while (true) {
            counter1++;
        }
    }
};

int main(int /*argc*/, char **/*argv*/[]) {
    MyThread thread;
    thread.start();
    QThread::usleep(5000);
    thread.terminate();
    thread.wait();
    std::cout << "Counter1: " << counter1 << std::endl;
    return 0;
}
```

Auto-identification

- Un thread peut obtenir son identifiant avec la fonction:

```
Qt::HANDLE QThread::currentThreadId();
```

- Chaque thread dispose d'un numéro d'identifiant unique
- Sur linux x64, l'identifiant est un entier long non signé (unsigned long int)
-  il n'est pas garanti que l'identifiant d'un thread soit un entier ! Le type est propre à l'implémentation

Rendre le processeur

- Il est possible de forcer le thread appelant à relâcher le processeur avec la fonction:

```
QThread::yieldCurrentThread();
```

- Après l'appel à cette fonction, le thread est placé à la fin de la file d'attente des threads en attente du processeur
- Le thread suivant en attente du processeur est alors activé

Qt vs POSIX

| Action | Qt | POSIX |
|-----------------------|--------------------------------|-----------------------------------|
| Descripteur de thread | class QThread | pthread_t |
| Création | MyThread thread; | pthread_t thread; |
| Lancement | thread. start (); | pthread_create(...) |
| Passage d'arguments | Via constructeur ou accesseurs | Argument passé à pthread_create() |
| Jointure | thread. wait (); | pthread_join(&thread, &value); |
| Terminaison interne | return ; | pthread_exit(); ou return |
| Terminaison externe | terminate () | pthread_kill(); |
| Annulation externe | requestInterruption () | pthread_cancel(); |
| Comparaison | (&thread1) == (&thread2) | pthread_equal(); |

Code source

```
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_arguments.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_arguments2.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_nowait.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_wait.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_termination1.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_termination2.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/2-threads/thread_termination3.tar.gz
```