

Introduction à la programmation concurrente

Exclusion mutuelle par attente active

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

Ressource critique

- Une ressource critique est une ressource non partageable et accédée par plusieurs tâches.
- Exemples:
 - une variable globale (ressource logique) accédée en lecture et en écriture par des tâches;
 - une ressource physique (périphérique) accédée par des tâches.

Ressource critique: exemple

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

class CounterThread: public QThread
{
public:
    void run() Q_DECL_OVERRIDE {
        for(int i=0;i<iterations;i++) {
            global = global + 1;
        }
    }
};

int main(int argc, char *argv[])
{
    CounterThread threads[2];
    for(int i=0;i<2;i++)
        threads[i].start();
    for(int i=0;i<2;i++)
        threads[i].wait();

    std::cout << "Fin des taches : global = " << global
               << " (" << 2*iterations << ")" << std::endl;

    return 0;
}
```



Exemple d'exécution

Fin de Tache 2
Fin de Tache 1
Fin des taches: global = 2000000 (2000000)

Fin de Tache 1
Fin de Tache 2
Fin des taches: global = 1405922 (2000000)

Fin de Tache 1
Fin de Tache 2
Fin des taches: global = 1394508 (2000000)

Fin de Tache 1
Fin de Tache 2
Fin des taches: global = 2000000 (2000000)

Explication

- Que signifie `global = global + 1;`?
- En langage machine:

`mov global,r1` % copie la valeur de la mémoire
 % commune désignée par `global`
 % dans un registre local `r1`;

`addi 1,r1` % ajoute 1 au registre;

`mov r1,global` % stocke le contenu du registre
 % local `r1` à l'adresse de
 % `global`

Explication

Exemple d'une exécution erronée

Tâche 1	$global = 10$	Tâche 2
$r1 \leftarrow 10$	changement de contexte	$r1 \leftarrow 10$
		$r1 \leftarrow r1 + 1$
		$global \leftarrow r1$ (vaut 11)
$r1 \leftarrow r1 + 1$	changement de contexte	
$global \leftarrow r1$		
	$global$ vaut ainsi 11 et non 12!	

Exemple 2

Valeur minimale de *global*

Section critique

- Une ressource critique doit être exécutée en exclusion mutuelle (c.-à-d. les accès à la ressource s'excluent mutuellement).
- La portion de code décrivant un accès à une ressource critique (RC) est appelée *section critique*
- L'exclusion mutuelle doit être assurée dans une section critique
- L'accès à une section critique est géré par un algorithme d'exclusion mutuelle en deux parties:
 - protocole d'entrée
 - protocole de sortie

Section critique

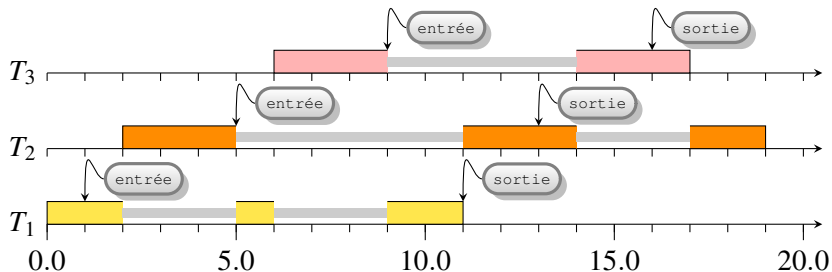
Modèle typique pour une tâche

```
void *T(void *arg) {  
    déclarations locales;  
    instructions;  
    while (true) {  
        instructions;  
        <prélude>           // protocole d'entrée  
        <section critique> // Accès à la RC  
        <postlude>          // protocole de sortie  
        instructions;  
    }  
}
```

Propriété des algorithmes

- Il ne doit pas avoir d'interblocage
 - La ressource critique doit être accessible

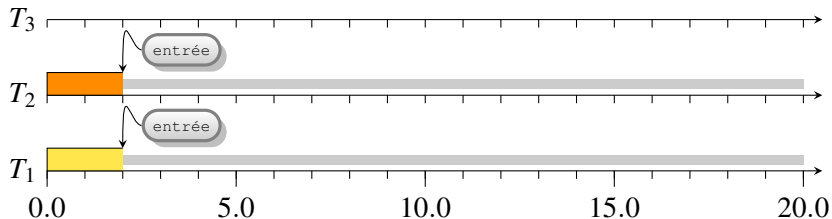
Exemple correct:



Propriété des algorithmes

- Il ne doit pas avoir d'interblocage
 - La ressource critique doit être accessible

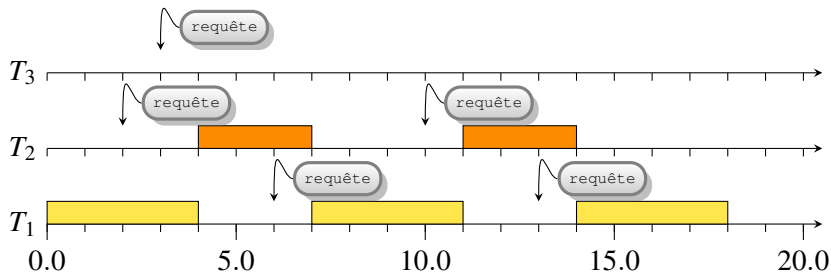
Exemple avec interblocage:



Propriété des algorithmes

- Il faut éviter la famine
 - Les autres tâches se "liguent" et empêche une tâche d'accéder à la ressource

Ici les slots alloués correspondent aux moments où une tâche est en section critique



Propriété des algorithmes

- Plusieurs tâches peuvent réclamer une ressource
- L'attente peut être:
 - *FIFO* (ou *PAPS*)
 - *Linéaire*: une tâche ne peut accéder deux fois la ressource si une autre est en attente
 - *bornée par une fonction $f(n)$* : pour n tâches, une tâche en attente ne peut se faire dépasser par $f(n)$ tâches
 - *finie*: l'attente n'est pas bornée mais pas infinie

Propriété des algorithmes

Règles pour la mise au point des algorithmes

- Règle 1 : à tout instant, une seule tâche peut se trouver en section critique (exclusion mutuelle);
- Règle 2 : si plusieurs tâches sont bloquées en attente d'entrer en section critique alors qu'aucune tâche ne s'y trouve, l'une d'entre elles doit pouvoir y accéder au bout d'un temps fini (pas d'interblocage);
- Règle 3 : le comportement d'une tâche en dehors de la section critique et des protocoles qui en gèrent l'accès n'a aucune influence sur l'algorithme d'exclusion mutuelle;
- Règle 4 : aucune tâche ne joue de rôle privilégié, la solution est la même pour toutes.

Propriété des algorithmes

- Nos premiers algorithmes d'exclusion mutuelle utilisent des instructions usuelles :
 - l'attente active par des boucles;
 - des variables partagées (globales);
- Ils possèdent des risques d'erreurs comme
 - exclusion mutuelle non satisfaite
 - interblocage
 - famine
- Ils tentent
 - d'éviter toute attente inutile
 - d'assurer l'équité si possible


Tentative 1

```

bool occupe = false;

void T0::run()
{
    while (true) {
        while (occupe)
            ;
        occupe = true;
        /* section critique */
        occupe = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        while (occupe)
            ;
        occupe = true;
        /* section critique */
        occupe = false;
        /* section non-critique */
    }
}

```

Exclusion	Interblocage	Famine	Couplage


Tentative 2

```

int tour = 0;  // ou 1

void T0::run()
{
    while (true) {
        while (tour != 0)
            ;
        /* section critique */
        tour = 1;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        while (tour != 1)
            ;
        /* section critique */
        tour = 0;
        /* section non-critique */
    }
}

```

Exclusion	Interblocage	Famine	Couplage


Tentative 3

```

bool etat[2] = {false, false};

void T0::run()
{
    while (true) {
        while (etat[1])
            ;
        etat[0] = true;
        /* section critique */
        etat[0] = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        while (etat[0])
            ;
        etat[1] = true;
        /* section critique */
        etat[1] = false;
        /* section non-critique */
    }
}

```

Exclusion	Interblocage	Famine	Couplage


Tentative 4

```

bool etat[2] = {false,false};

void T0::run()
{
    while (true) {
        etat[0] = true;
        while (etat[1])
            ;
        /* section critique */
        etat[0] = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        etat[1] = true;
        while (etat[0])
            ;
        /* section critique */
        etat[1] = false;
        /* section non-critique */
    }
}

```


Exclusion	Interblocage	Famine	Couplage

Tentative 5

```

bool etat[2] = {false,false};
void T0::run()
{
    while (true) {
        etat[0] = true;
        while (etat[1]) {
            etat[0] = false;
            while (etat[1])
                ;
            etat[0] = true;
        }
        /* section critique */
        etat[0] = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        etat[1] = true;
        while (etat[0]) {
            etat[1] = false;
            while (etat[0])
                ;
            etat[1] = true;
        }
        /* section critique */
        etat[1] = false;
        /* section non-critique */
    }
}

```

Exclusion	Interblocage	Famine	Couplage

Algorithme de Dekker

Concept

```
void T0::run()
{
    while (true)
    {
        setEnSectionCritique();
        while (autreEnSectionCritique()) {
            if (pasMonTour()) {
                clearEnSectionCritique();
                while (pasMonTour())
                    ;
                setEnSectionCritique();
            }
            /* section critique */
            passeMonTour();
            clearEnSectionCritique();
            /* section non-critique */
        }
    }
}
```


Implémentation

```
bool etat[2] = {false, false};
int tour = 0; // ou 1

void T0::run()
{
    while (true) {
        etat[0] = true;
        while (etat[1])
            if (tour == 1) {
                etat[0] = false;
                while (tour == 1)
                    ;
                etat[0] = true;
            }
        /* section critique */
        tour = 1;
        etat[0] = false;
        /* section non-critique */
    }
}
```

Algorithme de Dekker: implémentation complète

```
bool etat[2] = {false,false};
int tour = 0; // ou 1
```



```
void T0::run()
{
    while (true) {
        etat[0] = true;
        while (etat[1])
            if (tour == 1) {
                etat[0] = false;
                while (tour == 1)
                    ;
                etat[0] = true;
            }
        /* section critique */
        tour = 1;
        etat[0] = false;
        /* section non-critique */
    }
}
```

```
void T1::run()
{
    while (true) {
        etat[1] = true;
        while (etat[0])
            if (tour == 0) {
                etat[1] = false;
                while (tour == 0)
                    ;
                etat[1] = true;
            }
        /* section critique */
        tour = 0;
        etat[1] = false;
        /* section non-critique */
    }
}
```

Algorithme de Peterson

Concept

```
void T0::run()
{
    while (true) {
        setIntention();
        setTourAutre();
        while (intentionAutre() && pasMonTour())
            ;
        /* section critique */
        clearIntention();
        /* section non-critique */
    }
}
```

Implémentation

```
bool intention[2] = {false, false};
int tour = 0; // ou 1

void T0::run()
{
    while (true) {
        intention[0] = true;
        tour = 1;
        while (intention[1] && tour == 1)
            ;
        /* section critique */
        intention[0] = false;
        /* section non-critique */
    }
}
```

Algorithme de Peterson: implémentation complète

```

bool intention[2] = {false, false};
int tour = 0; // ou 1

void T0::run()
{
    while (true) {
        intention[0] = true;
        tour = 1;
        while (intention[1] && tour == 1)
            ;
        /* section critique */
        intention[0] = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        intention[1] = true;
        tour = 0;
        while (intention[0] && tour == 0)
            ;
        /* section critique */
        intention[1] = false;
        /* section non-critique */
    }
}

```


Analyse

- Mathématiquement parlant, les algorithmes de Peterson et Dekker sont corrects
 - Mais...
 - Les processeurs multi-cœur les mettent à mal
 - Problème de cohérence des mémoires caches
 - Et d'ordre des accès mémoires
- ⇒ Ils ne sont plus sûrs. Il faut placer des barrières de synchronisation mémoire pour qu'ils fonctionnent
- Sous Linux:

```
#define BARRIER __asm__ volatile ("mfence" ::: "memory")
```

Garantie que tous les load et store présents avant la barrière sont effectués avant les load et store présents après

Algorithme de Dekker: implémentation multicore

```
bool etat[2] = {false, false};
int tour = 0; // ou 1
```



```
void T0::run()
{
    while (true) {
        etat[0] = true;
        BARRIER;
        while (etat[1])
            if (tour == 1) {
                etat[0] = false;
                while (tour == 1)
                    ;
                etat[0] = true;
                BARRIER;
            }
        /* section critique */
        tour = 1;
        etat[0] = false;
        /* section non-critique */
    }
}
```

```
void T1::run()
{
    while (true) {
        etat[1] = true;
        BARRIER;
        while (etat[0])
            if (tour == 0) {
                etat[1] = false;
                while (tour == 0)
                    ;
                etat[1] = true;
                BARRIER;
            }
        /* section critique */
        tour = 0;
        etat[1] = false;
        /* section non-critique */
    }
}
```

Algorithme de Peterson: implémentation multicore

```

bool intention[2] = {false, false};
int tour = 0; // ou 1

void T0::run()
{
    while (true) {
        intention[0] = true;
        tour = 1;
        BARRIER;
        while (intention[1] && tour == 1)
            ;
        /* section critique */
        intention[0] = false;
        /* section non-critique */
    }
}

```



```

void T1::run()
{
    while (true) {
        intention[1] = true;
        tour = 0;
        BARRIER;
        while (intention[0] && tour == 0)
            ;
        /* section critique */
        intention[1] = false;
        /* section non-critique */
    }
}

```

Propriété des algorithmes

Définitions

- Attente active : le temps processeur est gaspillé au test d'une condition (boucle) pour bloquer un processus
 - Attente passive : un processus est bloqué par le noyau (processeur disponible)
-
- Les algorithmes proposés utilisent l'attente active
 - L'attente passive nécessite des constructions spéciales
 - Verrous
 - Sémaphores
 - Moniteurs
 - ...
 - Ces constructions facilitent la conception d'algorithmes, mais n'éliminent pas le risque d'erreurs!
 - Le noyau doit les supporter

Exercice

```
bool intention[2] = {false, false};
int tour = 0; // ou 1
```

```
void T0::run()
{
    while (true) {
        intention[0] = true;
        while (tour != 0) {
            while (intention[1])
                ;
            tour = 0;
        }
        /* section critique */
        intention[0] = false;
        /* section non-critique */
    }
}
```

```
void T1::run()
{
    while (true) {
        intention[1] = true;
        while (tour != 1) {
            while (intention[0])
                ;
            tour = 1;
        }
        /* section critique */
        intention[1] = false;
        /* section non-critique */
    }
}
```

- L'exclusion mutuelle est-elle garantie par les parties prélude et postlude des tâches? Justifiez votre réponse.

Code source

```
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/increment.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/exclusion1.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/exclusion2.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/exclusion3.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/exclusion4.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/exclusion5.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/dekker.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/peterson.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/dekker_multicore.tar.gz
http://reds.heig-vd.ch/share/cours/PCO/cours/code/3-exclusion/peterson_multicore.tar.gz
```