Temps Réel

Jérôme Pouiller <j.pouiller@sysmic.org>

Septembre 2011

Cinquième partie V

Partage de ressources

13 Ordonnancement avec contraintes de précédance

14 Problèmatique des accès concurents
- Protection des structures de données
- Protection des ressources matériel
- Réentrance
- Partage de ressource entre deux tâches : Fonctionnement d'un mutex

15 Problème liés aux partage de ressources
- Dead Lock
- Latence introduite
- Inversion de priorité

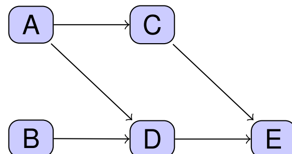
16 Solutions
- Priority inheritence
- Original ceiling priority
- Immediate Priority ceilling

17 Autres mécanismes de gestion d'accès concurrents
- Désactivation de l'ordonnanceur
- Sémaphore

Contraintes de précédence (1)

Problèmatique : Une tâche *j* doit s'éxecuter après *i*
Les contraintes de précédence sur l'ordre d'exécution des tâches les unes par rapport aux autres sont généralement décrites par un graphe orienté :

- $i < j$ indique que la tâche *i* est un prédécesseur de *j*
- $i \rightarrow j$ indique que la tâche *i* est un prédécesseur immédiat de *j*



Contraintes de précédence (2)

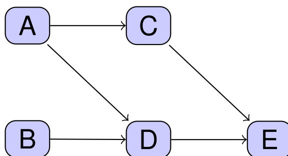
- Ici, seulement précédence simple : Les tâche dépendantes ont la même période
- Principe de l'établissement de l'ordonnancement :
  - Transformer l'ensemble des tâches dépendantes en un ensemble de tâches **indépendantes** que l'on ordonnancera par un algorithme classique
  - Par des modifications des paramètres des tâches : si $i \rightarrow j$ alors la règle de transformation doit respecter :
    - $r_j \leq r_i$
    - $P_i < P_j$
  - Validation de l'ordonnancabilité selon des critères utilisés pour des tâches indépendantes

Contraintes de précédence avec RM

Contraintes de précédence et Rate Monotonic : la transformation s'opère hors ligne sur la date de réveil et sur les délais critiques

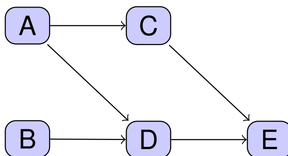
- $r_i^* = \max\{r_i, r_j^*\}$ pour tous les j tels que $j \rightarrow i$
- si $i \rightarrow j$ alors $P_i > P_j$ dans le respect de la règle d'affectation des priorités par Rate Monotonic

Exemple - RM



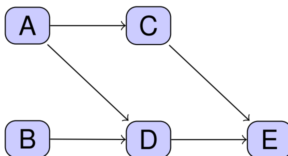
Tâche	Arrivée	r_i^*	P_i
A	0		
B	5		
C	0		
D	0		
E	0		

Exemple - RM



Tâche	Arrivée	r_i^*	P_i
A	0	0	
B	5	5	
C	0	0	
D	0	5	
E	0	5	

Exemple - RM



Tâche	Arrivée	r_i^*	P_i
A	0	0	1
B	5	5	1
C	0	0	2
D	0	5	2
E	0	5	3

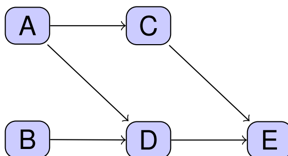
A renouveller pour chaque période

Contraintes de précédence avec DM

Contraintes de précédence et Deadline Monotonic : la transformation s'opère hors ligne sur la date de réveil et sur les délais critiques

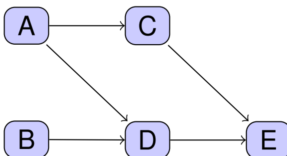
- $r_i^* = \max\left\{r_i, r_j^*\right\}$ pour tous les j tels que $j \rightarrow i$
- $D_i^* = \max\left\{D_i, D_j^*\right\}$ pour tous les j tels que $j \rightarrow i$ (ne corrspond pas au délais critique réel)
- si $i \rightarrow j$ alors $P_i > P_j$ dans le respect de la règle d'affectation des priorités par Deadline Monotonic

Exemple - DM



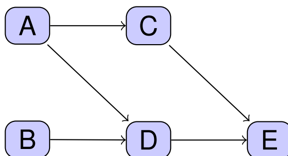
Tâche	Arrivée	Délai	r_i^*	D_i^*	P_i
A	0	5			
B	5	7			
C	0	5			
D	0	10			
E	0	12			

Exemple - DM



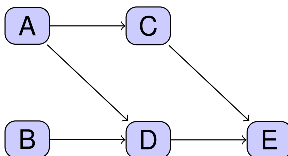
Tâche	Arrivée	Délai	r_i^*	D_i^*	P_i
A	0	5	0		
B	5	7	5		
C	0	5	0		
D	0	10	5		
E	0	12	5		

Exemple - DM



Tâche	Arrivée	Délai	r_i^*	D_i^*	P_i
A	0	5	0	5	
B	5	7	5	7	
C	0	5	0	5	
D	0	10	5	10	
E	0	12	5	12	

Exemple - DM



Tâche	Arrivée	Délai	r_i^*	D_i^*	P_i
A	0	5	0	5	1
B	5	7	5	7	3
C	0	5	0	5	2
D	0	10	5	10	4
E	0	12	5	12	5

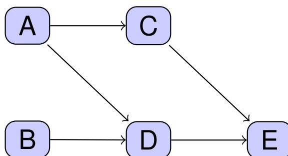
A renouveller pour chaque période

Contraintes de précédence avec EDF

Contraintes de précédence et EDF

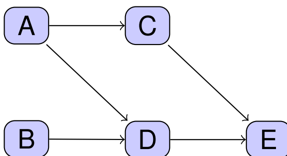
- modification des échéances de facon à ce qu'une tâche ait toujours un di inférieur à celui de ses successeurs (algorithme de Chetto et al.)
- une tâche ne doit être activable que si tous ses prédécesseurs ont terminé leur exécution
- modification de la date de réveil et de l'échéance
  - $r_i^* = \max\left\{r_i, \max\left\{r_j^* + C_j\right\}\right\}$ pour tous les j tels que $j \to i$
  - $d_i^* = \min\left\{d_i, \min\left\{d_j^* - C_j\right\}\right\}$ pour tous les j tels que $i \to j$
  - on itère sur les prédécesseurs et successeurs immédiats
  - on commence les calculs par les tâches qui n'ont pas de prédécesseurs pour le calcul des r et par les tâches qui n'ont pas de successeur pour le calcul des d

Exemple - EDF



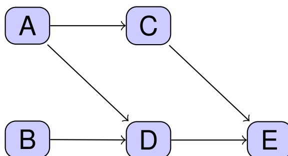
Tâche	Arrivée	Délai	r_i^*	d_i^*
A	0	5		
B	5	7		
C	0	5		
D	0	10		
E	0	12		

Exemple - EDF



Tâche	Arrivée	Délai	r_i^*	d_i^*
A	0	5	0	
B	5	7	5	
C	0	5	1	
D	0	10	7	
E	0	12	8	

Exemple - EDF



Tâche	Arrivée	Délai	r_i^*	d_i^*
A	0	5	0	3
B	5	7	5	7
C	0	5	1	5
D	0	10	7	9
E	0	12	8	12

A renouveller pour chaque période

Exemple de partage de données

Le partage de données entre les tâches posent les mêmes problème que le partage de données avec les interruptions

Prennons deux tâches `f1` et `f2` :

```
int a = 0;
char t[255];
void f1() {
  t[a] = data1;
  a++;
}
void f2() {
  t[a] = data2;
  a++;
}
```

Exemple de partage de données

Prennons le cas ou `f1` est préemptée par `f2` :

```
t[a] = data1; // t[0];



a++;
// a = 2 maintenant
```

```
t[a] = data2;
// t[0] est écrasé!
a++;
// a = 1 maintenant
```

3

Au lieu d'écrire les deux donnée l'une après l'autre, la valeur de `data1` est perdue alors que `t[1]` contiendra une valeur aléatoire.

Exemple de ressource partagée

Cas d'un périphérique réseau avec des registres mappés en mémoire. Le registre `0xABC0` permet de placer la donnée à envoyer. L'écriture d'un 1 sur le registre `0xABC4` permet d'effectuer l'envoi :

```
void send(int data) {
  *0xABC0 = data;
  *0xABC4 = 1;
}
```

Exemple de ressource partagée

Etudions le cas de l'éxecution simultanée de cette fonction par deux tâches. La première tâche appelle `send` avec `data = 42` :

```
*0xABC0 = 42
```

La tâche est préemptée. La seconde tâche appelle `send` avec `data = 10` :

```
*0xABC0 = 10
*0xABC4 = 1
```

`10` est envoyé. La tâche 1 reprend la main :

```
*0xABC4 = 1
```

`10` (au lieu de `42`) est de nouveau envoyé.

Ce cas est aussi valable dans le cas d'une interruption (bien que plus rare dans la pratique).

Comment éviter le problème ?

Nous devons élargir ce que nous avons précédement vu pour le partage d'informations avec les interruptions.

Les problèmes d'accès concurrents se traduisent très souvent par des *races conditions*. C'est à dire des problèmes aléatoires produit par une séquence particulière d'évènements

- Les *race conditions* sont souvent difficiles à reproduire et à identifier
- Les *race conditions* peuvent être latente dans le code et se déclarer suite à une modification de l'environnement externe
- Une race condition coûte chère (diffuculté de correction, peut potentiellement atterrir en production)

Comment s'en protèger ?

Comment s'en protèger ?

- Ne pas utiliser de variables globales ou de ressources partagées
- Utiliser des accès atomiques
- Placer des accès aux ressources partagée dans des *sections critiques*

Une fonction pouvant être appellée simultanénement depuis deux contextes de tâches différentes est dite *réentrante*

Partage de ressources critiques

Une ressource critique ne peut :

- être utilisée simultanément par plusieurs tâches
- être réquisitionnée par une autre tâche

Notion de section critique :

- séquence d'instructions pendant lesquelles on utilise une ressource critique
- sans problème dans le cas d'un ordonnancement non préemptif, mais c'est rarement le cas dans un environnement temps réel

$\Rightarrow$ évaluation du temps de réponse très difficile, sinon impossible (abondante littérature !)

Fonctionnement d’un mutex

Nécessite une instruction assembleur premettant un accès en lecture et une écriture en une instruction :
`test_and_set` affecte le registre d’état en fonction de la valeur du registre et affecte la valeur 1 au registre. On peut développer la fonction `lock` à partir de là :

```
void lock(mutex_t *m) {
  while (test_and_set(m))
    schedule();
}

void unlock(mutex_t *m) {
  m = 0;
  schedule();
}
```

Fonctionnement d'un mutex

Un peu mieux :

```
1  void lock(mutex_t *m) {
    while (test_and_set(m)) {
      this_task.reason = m;
      this_task.state = stop;
      schedule();
    }
   }

   void unlock(mutex_t *m) {
    m = 0;
11  foreach (i in tasks)
      if (i.state == stop && i.reason == m)
        i.state = run;
    schedule();
   }
```

Problèmes associés aux sections critiques

Voici les problèmes à prendre en considération lors de l'utilisation de sections critiques :

- Dead Lock
- Latence induite
- Inversion de priorité

Dead lock

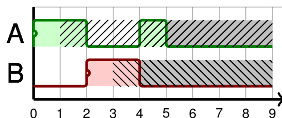
- Aussi appellé *étreinte fatale*
- Deux tâches utilisent deux ressources imbriquées dans l'ordre inverses

Tache 1 :

```
lock(m1);



5 lock(m2);
```

Tache 2 :

```

lock(m2);
// Deadlock ici:
lock(m1);
5
```



Dead lock

Remarque :
Le code suivant :

```
lock(m);
lock(m);
```

entraine un *double lock*, un type particulier de *Dead lock*

Mutex dans une interruption

Remarque :
Ne jamais utiliser de mutex dans une interruption.

- Si la ressource est occuppée par la tâche qui vient d'être préemptée, le `lock()` s'éxécutera dans le même contexte

→ Double lock

- De plus, le blocage d'un mutex peut entrainer peut entrainer une très important latence ce qui est en contradiction avec l'objectif de rester le minimum de temps dans une interruption

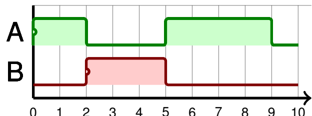
→ Règle générale : Il ne faut pas appeller `schedule` dans une interruption.

Latence

- Cas d'une tâche de faible priorité utilisant une ressource d'une tâche de haute priorité.
- Existe uniquement en environnement préemptif

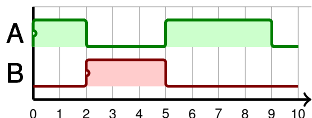
Exemple avec deux tâches $P_A < P_B$:

- Cas sans partage de ressource :



$TR_B = 3$

- Cas avec une ressource partagée :



$TR_B = 6$

Inversion de priorité

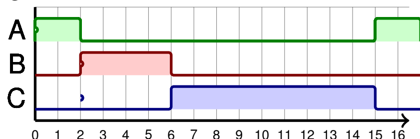
- Phénomène dû à la présence simultanée de priorités fixes et de ressources à accès exclusif dans un environnement préemptif
- Une tâche de priorité intermédiaire peut être élue par l'ordonnanceur alors que des tâches de plus haute priorité attentent.
- Il est diffile de calculer le temps de blocage des tâches par une inversion de priorité (il est généralement non-borné)

Cas notable de *Path Finder*

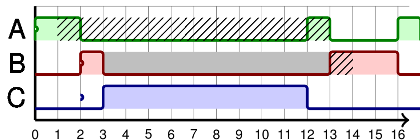
Inversion de priorité

Exemple :
On ajoute à l'exemple précédant une tâche C telle que :
$P_B > P_C > P_A$

- Cas sans partage de ressource :



- Cas avec un mutex :



La tâche C s'éxecute alors qu'elle a une priorité inféreure à B et ne partage aucune ressource avec B.

Temps de réponse

Avec les ressources partagées :

- Le temps de réponse avec un algorithme à priorité fixe devient :

$$TR_i = B_i + C_i + \sum_{P_j > P_i} \left\lceil \frac{TR_i}{T_j} \right\rceil C_j$$

- B_i est le temps de blocage de la tâche i par une ressource détenue par une tâche de priorité inférieure.
- B_i est difficile à calculer
- La condition nécessaire d'ordonnancabilité par un algorithme Rate Monotonic devient :

$$\forall i \in tasks, \left(\sum_{k \in tasks} \frac{C_k}{T_k} \right) + \frac{B_i}{T_i} \leq i \left(2^{\frac{1}{i}} - 1 \right)$$

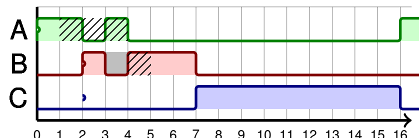
Conseils

- Essayer de limiter la taille des sections critique
- Essayer de découper les sections critiques de manière à favoriser la préemption des tâches de haute priorité
- Essayer de diminuer la granularité des mutex afin de limiter la taille des sections critiques.

Héritage de priorité

- Aussi appellé *Priority inheritence*
- Principe : Si une tâche bloque une ressource demandée par une tâche de plus haute priorité, elle acquiert temporairement la priorité de la tâche de haute priorité.
- Attention à la gestion de l'héritage en cascade.
- L'algorithme parait simple, mais c'est assez complexe. Il existe même des cas problèmatiques (dont le cout de résolution est polynomial) dans le cas d'architectures multiprocesseurs symétriques sur des `rw_lock`.

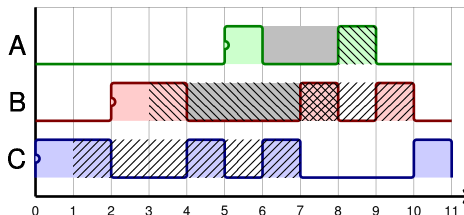
Exemple précédent :



Héritage de priorité

Autre exemple : Soit 3 tâches et 2 ressources telle que A et B partage la première ressource et B et C partage la seconde :

$$P_A > P_B > P_C$$



Héritage de priorité

Concernant le temps de réponse des tâches dans un système intégrant l'héritage de priorité :

- Le temps de blocage B_i d'une tâche de haute priorité par une tâche de plus basse priorité nominale est borné
- ... mais le calcul de ce temps de blocage maximum reste très complexe

Problème rémanent

- Blocages en chaîne (cf. notre exemple avec 3 tâches)
- Deadlock

Priorité plafonnée originale

- Aussi appellée *Original Ceiling Priority Protocol (OCPP)* ou tout simplement *Ceiling Priority*
- Introduit à la fin des années 80 pour résoudre le problème d'inversion de priorité tout en prévenant l'occurence de deadlocks et de blocages en chaîne
- Amélioration du protocole d'héritage de priorité : une tâche ne peut pas entrer dans une section critique s'il y a un sémaphore acquis qui pourrait la bloquer
- Principe : on attribue à chaque sémaphore une priorité plafond égale à la plus haute priorité des tâches qui pourraient l'acquérir. Une tâche ne pourra entrer dans la section critique que si elle possède une priorité supérieure à toutes celles des priorités plafond des sémaphores acquis par les autres tâches

Algorithme

- On attribue à chaque sémaphore S_k une priorité plafond $C(S_k)$ égale à la plus haute priorité des tâches susceptibles de l'acquérir
- Soit i la tâche prête de plus haute priorité : l'accès au processeur est donné à i
- Soit S_* le sémaphore dont la priorité plafond $C(S_*)$ est la plus grande parmi tous les sémaphores déjà acquis par des tâches autres que i
- Pour entrer dans une section critique gardée par un sémaphore S_k, i doit avoir une priorité supérieure à $C(S_*)$. Si $P_i \leq C(S_*)$, l'accès à S_k est interdit et on dit que i est bloquée sur S_* par la tâche qui possède S_*. La priorité de i est alors passée à la tâche proriétaire de S_*

Priorité plafonnée immédiate

- Aussi appellée *Immediate Ceiling Priority Protocol (ICPP)* ou *Hightest Lock*
- Similaire à OCPP mais plus simple.
- Dès qu'une tâche acquiert une sémaphore S_k, elle acquiert sa priorité $C(S_k)$ en même temps

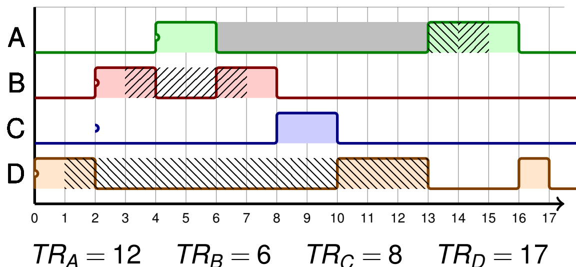
Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	__12_
B	2	2	4	_22_
C	2	3	2	__
D	0	4	6	_1111_

Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	__12_
B	2	2	4	_22_
C	2	3	2	__
D	0	4	6	_1111_

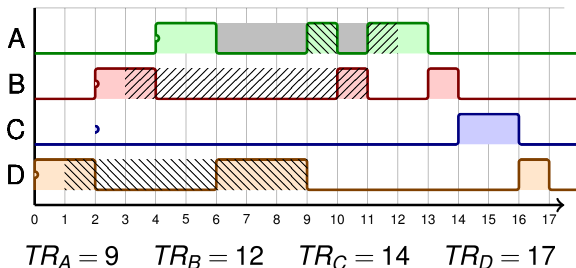
Sans mécanisme de protection



Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	__12_
B	2	2	4	_22_
C	2	3	2	__
D	0	4	6	_1111_

Par héritage de priorité



$TR_A = 9$ $TR_B = 12$ $TR_C = 14$ $TR_D = 17$

Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	___12_
B	2	2	4	_22_
C	2	3	2	___
D	0	4	6	_1111_

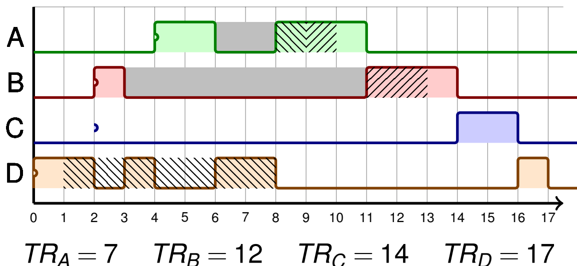
Par *Priority Ceiling*, les priorités plafond sont donc :

- $C(S_1) = P_A$
- $C(S_2) = P_A$

Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	__12_
B	2	2	4	_22_
C	2	3	2	__
D	0	4	6	_1111_

Par priorité plafonnée originale :

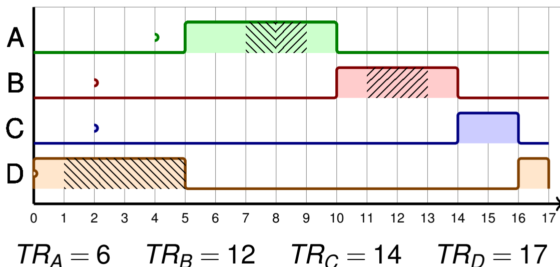


$TR_A = 7$ $TR_B = 12$ $TR_C = 14$ $TR_D = 17$

Exemple

Tâche	Arrivée	Priorité	Capacité	Ressources
A	4	1	5	__12_
B	2	2	4	_22_
C	2	3	2	__
D	0	4	6	_1111_

Par priorité plafonnée immédiate :



Priorité plafonnée

On a le même critère d'ordonnancabilité par RM que dans le cas du protocole d'héritage de priorité :

$$\forall i \in tasks, \left(\sum_{k \in tasks} \frac{C_k}{T_k} \right) + \frac{B_i}{T_i} \leq i \left(2^{\frac{1}{i}} - 1 \right)$$

et le même calcul du temps de réponse :

$$TR_i = B_i + C_i + \sum_{P_j > P_i} \left\lceil \frac{TR_i}{T_j} \right\rceil C_j$$

mais le calcul du temps de blocage maximum de chaque tâche est plus simple :

- on peut démontrer que le temps de bloquage maximum B_i d'une tâche i est la durée de la plus longue des sections critiques ($\max SC_{j,k}$) parmi celles appartenant à des tâches de priorité inférieure à P_i et gardées par des sémaphores dont la priorité plafond est supérieure ou égale à P_i

$$B_i = \max SC_{j,k} | P_j < P_i, C(S_k) \geq P_i$$

Sémaphore

- Différence entre un mutex et un semaphore binaire : presque aucune.
- Parfois le sémaphore binaire est utilisé pour implémentéer le mutex.
- Toutefois, d'un point de vue sémantique, on pourrait que le mutex permet d'avoir un morceau de code mutuelement exclusif tandis que le sémaphore est une section de code limité à uen 1 ressource.

Mutex Réentrant

- Idem mutex, mais si la même tâche tente de revérouiller le même mutex, le mutex est non-bloquant.
- Dans le cas d'un mutex non-réentrant, ceci entraine forcement un dead-lock.
- Un sémaphore est maintenu pour connaitre le nombre de passage.

Read/Write Lock

Permet de limiter le phénomène de latence en diminuant le nombre de sections critiques.

Solution 1 (*reader preference*) :

```
void read_lock() {
  // mutex protege
     read_count
  lock(mutex);
  readcount++;
  if (readcount == 1)
    lock(w);
  unlock(mutex);
}
```

```
void read_unlock() {
  lock(mutex);
  readcount--;
  if (readcount = 0)
    unlock(w);
  unlock(mutex);
}
void write_lock() {
   lock(w);
}
void write_unlock() {
  unlock(w);
}
```

Read/Write Lock

Problème : un accès en écriture doit attendre que toutes les lectures soient terminées. Solution 2 (*writer preference*) :

```
void read_lock() {
  lock(r);
  lock(mutex);
  readcount++;
  if (readcount == 1)
    lock(w);
  unlock(mutex);
  unlock(r);
  // r n'est pas bloque
     durant la lecture
}
```

```
void read_unlock() {
  lock(mutex);
  readcount--;
  if (readcount == 0)
    unlock(w);
  unlock(mutex);
}
void write_lock() {
   lock(r);
   lock(w);
}
void write_unlock() {
  unlock(w);
  unlock(r);
}
```

Rendez-vous

Permet de synchroniser deux tâches. La première tâche arrivée à la barrière attend la seconde.

```
void init() {
  lock(m1);
  lock(m2);
}
```

Tâche 1 :

```
  unlock(m1);
  lock(m2);
```

Tâche 2 :

```
  unlock(m2);
  lock(m1);
```

Conditions

Peuvent être considérés comme des *rendez-vous* à sens unique. Si une tâche attend, elle est débloquée, sinon, aucun effet. Très utilisée pour le pattern des `work-thread`

```
void init() {
  lock(m);
}

void wait() {
  lock(m);
}

void signal() {
  unlock(m);
  try_lock(m);
}
```

8

```
// broadcast debloque
// tous les waiters alors
// que signal en debloque
// uniquement un
void broadcast() {
  // Plus complexe, il
  // faut un mutex par
  // waiters.
}
```

8

Buffer circulaire et Queue

- Précédement décrit dans la section “Partage d’information avec les interruptions”
- Fonctionne aussi très bien entre les tâches
- Une des rare structure permettant d’être partagée à la fois avec une interruption et des tâches
- Faire attention à l’allocation dynamique des objets

Spin Lock, Mutex et désactivation des interruptions

Lorsque :

- Vos sections critique font intervenir des tâches et des interruptions
- Votre problème ne concerne pas un échange de données (et donc le buffer circulaire n'est pas une solution)
- Vous ne pouvez pas faire autrement

alors, vous devez combiner les trois mécanismes suivants : Désactivation des interruptions, Mutex et Spin Lock.

Le point sur ces trois mécanismes :

- Si une ressource est partagé entre une tâche et une interruption sur le même coeur, il est nécessaire de désactiver les interruptions
- Si une ressource est partagé entre deux tâche sur un même coeur, il est nécessaire d'utiliser un mutex
- Si la ressource est partagée avec un autre coeur et que le temps d'utilisation est court, utilisez un Spin Lock.

Spin Lock, Mutex et désactivation des interruptions

On pourrait imaginer un cas cummulant les trois contraintes :

```
1 disable_interrupts()
  mutex_lock(m)
  spin_lock(s)
  a++
  spin_unlock(s)
  mutex_unlock(m)
  enable_interrupts()
```

Globalement, évitez !

Algorithmes non-bloquants

- Algorithme thread-safe n'utilisant pas de sections ciritques.
- Ces algorithmes utilisent souvant des instructions atomiques proposés par certains processeurs
- Par conséquant, ils sont peu portables
- Souvent utilisé dans les base de données

Read-Copy-Update (RCU)

Type d'algorithme non bloquant :

- La lecture n'est pas bloquante
- On note le nombre de lecteurs
- Les modification s'effectuent sur une copie de l'objet
- Les lecture suivante se font sur la nouvelle version de l'objet
- Lorsque le dernier lecteur a terminé, l'objet d'origine est détruit. Seul subsiste la nouvelle version.

Exemple : Manipulation de listes

```c
typedef struct {
  struct a_t a;
  int count_usage = 0;
  bool obsolete = false;
} rcu_t;
rcu_t *a = malloc(sizeof(rcu_t));
```

```c
void read_a() {
 // lock:
 rcu_t *ptr = a;
 ptr->count_usage++;
 // do something with ptr;
 // unlock:
 ptr->count_usage--;
 if (ptr->obsolete && !ptr->
      count_usage)
   free(ptr);
}
```

```c
void write_a() {
  struct rcu_t *a3 = a;
  struct rcu_t *a2 = malloc(
       sizeof(rcu_t));
  memcpy(a2, a);
  // modify a2;
  a = a2;
  a3->obsolete = true;
  if (!a3->count_usage)
    free(ptr);
}
```