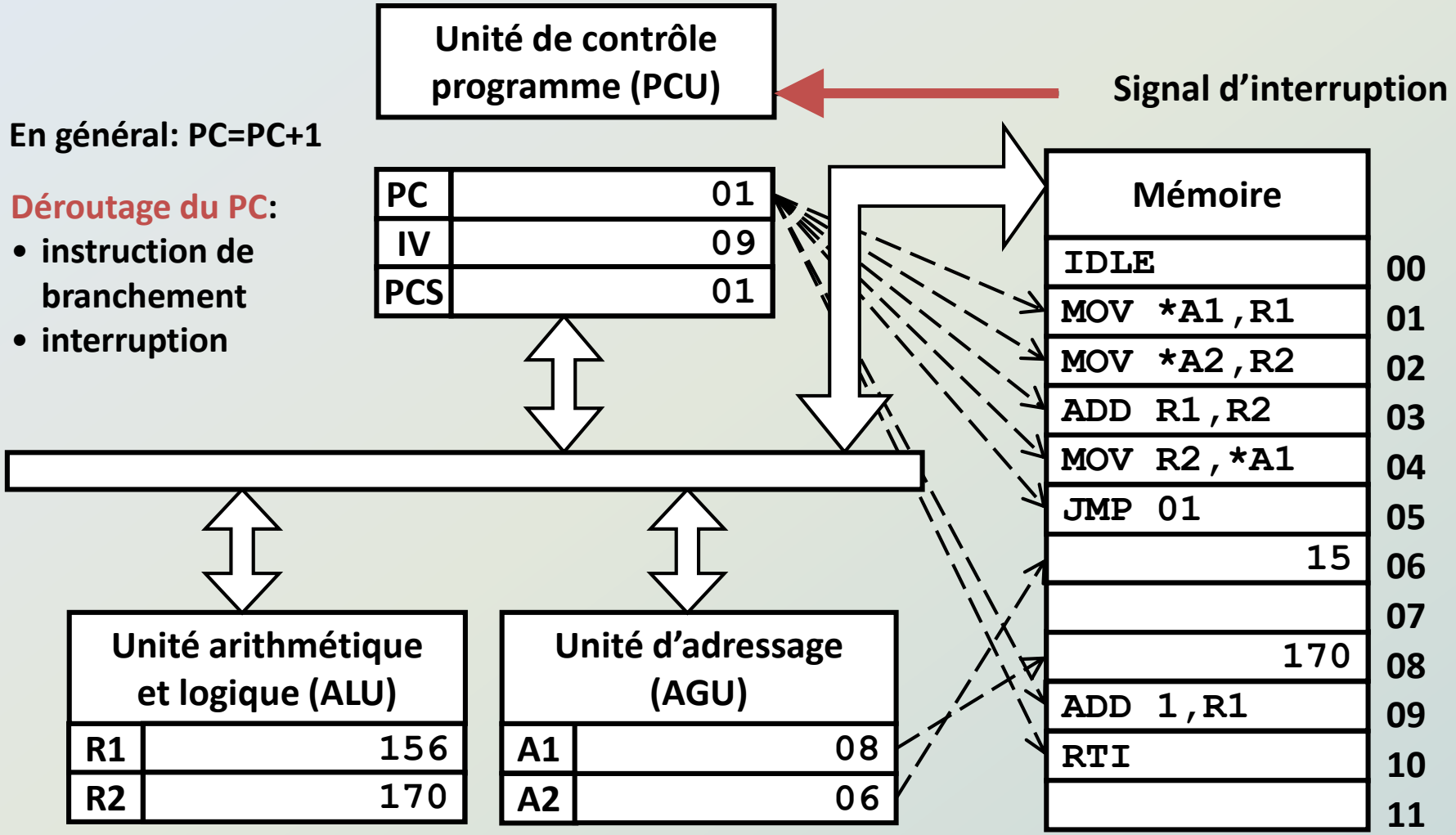
# Systèmes Multitâches

**Shebli Anvar**
**Irfu – CEA Saclay**
shebli.anvar@cea.fr

**Jean-Philippe Babau**
**UFR Sciences et Techniques – Univ. Brest**
jean-philippe.babau@univ-brest.fr

# Architecture électronique d'un processeur



# Comment implanter le multitâche préemptif ?

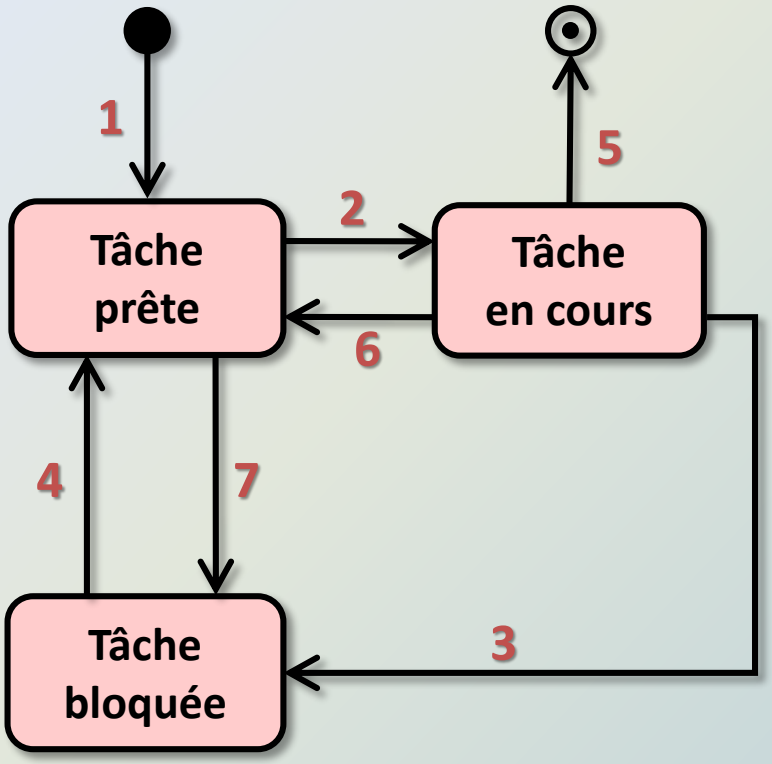
- **Appel régulier d'un ordonnanceur**
  - Régularité assurée par une alarme électronique (hardware timer)
  - TIC = période de cette alarme (typiquement quelques ms, en anglais: tick)
- **Préemption: appel de l'ordonnanceur (scheduler) par interruption**
  - L'alarme active un signal d'interruption
  - La routine associée à l'alarme d'ordonnancement est le code de l'ordonnanceur
- **Le code de l'ordonnanceur « active » une tâche (task, thread)**
  - Sauvegarde le contexte de la tâche en cours
  - Considère la liste des données liées à la gestion des tâches
  - Décide par un algorithme de la tâche à activer
  - Restaure le contexte de la tâche élue
  - Contexte $\supseteq$ valeurs des registres du processeur, notamment le PCS

# Ordonnanceur

**Rôle de l'ordonnanceur**

- Choix de la tâche à exécuter parmi les tâches prêtes
- Différents algorithmes d'ordonnancement → différentes politiques d'ordonnancement

**Machine d'états d'une tâche**




**Changements d'état d'une tâche**

| | |
|---|---|
| 2 | Élection |
| 5 | Fin de la tâche |
| 6 | Préemption |
| 3 | Demande de ressource occupée, attente événement non présent, auto-suspension |
| 1 | Création de la tâche |
| 7 | Suspension |
| 4 | Ressource libérée, événement arrivé, réactivation |

# Ordonnancement circulaire

**(round robin scheduling)**

Dans un 1er temps, on ne gère pas l'état « tâche bloquée » mais on en tient compte

- Soit **`T`** la structure de données décrivant une tâche.
- Soit **`idlet`** la variable globale de type **`T*`** pointant vers une tâche contenant l'unique instruction IDLE qui stoppe le CPU (le PC n'avance plus). Cette tâche existe toujours.
- Soit **`ct`** la variable globale contenant le pointeur de la tâche courante.
- On suppose données les macros **`ctxt_save(t)`** et **`ctxt_restore(t)`** de sauvegarde et restauration de contexte (**`t`** étant le pointeur sur une structure de tâche en mémoire).
- On suppose données les macros d'injection et de retrait des tâches dans une file d'attente:

```
void put_task(T* t);   /* Place tâche t en queue de file */
T* get_task( );  /* Retire de la file et renvoie la tâche en tête. NULL si file vide */
int get_num( );  /* Renvoie le nombre de tâches dans la file */
```

- On suppose également que la création / destruction de tâches est assurée par ailleurs.

1) En reprenant le modèle électronique de processeur et la machine d'états d'une tâche, définir la structure **`T`** (en langage C).
2) Écrire l'algorithme de l'ordonnanceur circulaire **`scheduler()`**: chaque tâche, pourvu qu'elle soit prête, tourne à son tour le temps d'un tic (elle peut ne pas être prête du fait d'un appel système antérieur bloquant).

# Ordonnancement circulaire

**(round robin scheduling)**

```
enum
{
    READY,
    RUNNG,
    BLCKD
} State;

struct T
{
    void* reg_PCS;
    void* reg_A[2];
    long  reg_R[2];
    State state;
    void* stack;
};

T* ct = idlet;
```

```
void scheduler ( )
{
    ctxt_save (ct);
    if (ct->state != BLCKD) ct->state = READY;
    put_task(ct); /* Remise en queue de file */

    while(true)
    {
        ct = get_task();
        if (ct->state == READY)
        {
            ct->state = RUNNG;
            ctxt_restore (ct); /* PC JUMP */
        }
        else put_task(ct);
    }
}
```

NOTE: Le contexte d'une tâche doit inclure une pile en mémoire permettant à chaque tâche d'avoir ses variables automatiques. En conséquence, l'architecture électronique doit comporter un registre contenant le pointeur de pile et la structure **T** doit sauvegarder la valeur de ce pointeur.

- Que se passe-t-il si aucune tâche n'est éligible à l'état **RUNNG** ?

**idlet** *existe et est toujours éligible*

- Quel est le défaut de cette implémentation en termes d'utilisation du CPU ?

**idlet** *est régulièrement exécutée au moins sur 1 tic même s'il y a d'autres tâches éligibles.*

# Ordonnancement à priorité

**(priority scheduling)**

- On ne gère toujours que les états « tâche prête » et « tâche en cours » mais on tient compte de la possibilité qu'une tâche soit bloquée.
- Chaque tâche est affectée d'une priorité notée **`p`** $\in$ {0,…,16}
- L'ordonnanceur doit en priorité faire tourner la tâche de priorité la plus élevée.

1) Quelle doit être la priorité de la tâche **`idlet`** ?

   *La priorité de* **`idlet`** *doit être* **`0`**

2) Quelle doit être la priorité des tâches autres que **`idlet`** ?

   *La priorité tâches autres que* **`idlet`** *doit être* **`p∈{1,…,16}`**

3) Réécrire la structure **`T`**, ainsi que la fonction **`scheduler()`**.

# Ordonnancement à priorité

(priority scheduling)

```
enum
{
    READY,
    RUNNG,
    BLCKD
} State;

struct T
{
    void* reg_PCS;
    void *reg_A[2];
    int   reg_R[2];
    State state;
    void* stack;
    short prio;
};

T* ct = idlet;
```

```
void scheduler ( )
{
    ctxt_save (ct);
    if (ct->state == RUNNG) ct->state = READY;
    put_task(ct);

    ct = idlet;
    T* next;
    int n = get_num();
    while (n-- > 0)
    {
        next = get_task(); /* Lecture tête de file */
        if (next->state == READY and next->prio > ct->prio)
        {
            put_task(ct);
            ct = next;
        }
        else put_task(next); /* Remise en fin de file */
    }
    ct->state = RUNNG;
    ctxt_restore (ct);
}
```

# Ordonnancement par files de priorité

**(priority queue scheduling)**

- **Le plus souvent, les ordonnanceurs utilisent plusieurs files d'attente pour gérer les tâches**
- **Ordonnancement par priorité: . . . une file d'attente par priorité . . . .**

Soient les macros d'injection et de retrait des tâches dans une file d'attente associée à une priorité (on suppose ces macros données):

```
void put_task(int prio, T* t); /* Place t en fin de file de priorité prio */
T* get_task(int prio); /* Retire et renvoie la tâche en tête de file. NULL si file vide */
int get_num(int prio); /* renvoie nombre de tâches dans file de priorité prio */
```

☞ Réécrire la fonction **`scheduler()`** en utilisant ces macros ainsi que les macros de sauvegarde et restauration de contexte déjà définis. Attention : deux tâches de même priorité doivent être ordonnancées de manière circulaire.

# Ordonnancement par files de priorité

**(priority queue scheduling)**

```
void scheduler ()
{
    ctxt_save(ct);
    if(ct->state == RUNNG) ct->state = READY;

    put_task(ct->prio, ct); /* Remise dans sa file */
    for (int prio = MAXPRIO ; prio >= 0 ; prio--)
    {
        int n = get_num(prio);
        while(n-- > 0)
        {
            ct = get_task(prio);
            if (ct->state == READY)
            {
                ct->state = RUNNG;
                ctxt_restore (ct); /* Election */
            }
            else put_task(prio, ct); /* Remise dans file */
        }
    }
}
```

**Quel est l'intérêt d'un tel ordonnancement par rapport à précédemment ?**

- *L'exécution de l'ordonnanceur est minimisée.*
- *Le déterminisme de l'exécution des tâches est amélioré.*

# Ordonnancement par files de priorité avec ressources

## implémentation du mutex

On gère cette fois-ci l'état « tâche bloquée ». On considère une structure **`M`** représentant un sémaphore d'exclusion mutuelle (mutex): lorsqu'une tâche **`t`** possède le jeton du mutex, toute autre tâche qui tente de le prendre est bloquée, et ce jusqu'à ce que **`t`** rende le jeton. La tentative par une tâche de réacquérir un mutex qu'elle possède déjà est une erreur. . La tentative par une tâche de rendre un mutex qu'elle ne possède pas est une erreur.

Pour gérer les conflits de concurrence et les mutex, les macros et variable suivantes sont disponibles:

- **`intLock()`** (resp. **`intUnlock()`**) interdisant (resp. autorisant) les interruptions en activant (resp. désactivant) un bit dans un registre matériel du processeur.
- la variable système **`preempt_lock`** doit être mise à **`1`** (resp. **`0`**) pour interdire (resp. rétablir) la préemption.
- L'appel de **`mput_task(m,t)`** injecte la tâche **`t`** dans la file associée au mutex **`m`**.
  L'appel de **`mget_task(m)`** renvoie *une* tâche de la file associée à **`m`** et la retire de la file (**`NULL`** s'il n'y a aucune tâche bloquée sur le mutex) .

Pour une variable **`M* m`**, **`m->owner`** est la tâche qui en possède le jeton (**`NULL`** si aucune tâche ne possède son jeton).
Une fonction peut déclencher à tout moment l'interruption d'ordonnancement par l'instruction **`TRAP(scheduler)`**.

- ☞ Écrire les fonctions **`mLock(M* m)`** (resp. **`mUnlock(M* m)`**) de prise (resp. rendu) de jeton, renvoyant **`0`** si le jeton a été effectivement pris (resp. rendu), **`-1`** s'il y a eu erreur.
- ☞ Réécrire une fonction **`scheduler()`** tenant compte de **`preempt_lock`**.

# Ordonnancement par files de priorité avec ressources

## implémentation du mutex

```
int mLock (M* m)
{
    preempt_lock=1;
    if (m->owner == ct) return -1;
    if (m->owner != NULL)
    {
        ct->state = BLCKD;
        mput_task(m, ct);
        preempt_lock=0;
        TRAP(scheduler);
        preempt_lock=1;
    }
    m->owner = ct;
    preempt_lock=0;
    return 0;
}
int mUnlock(M* m)
{
    preempt_lock=1;
    if (m->owner != ct) return -1;
    m->owner = mget_task(m);
    if (m->owner != NULL)
    {
      m->owner->state = READY;
      preempt_lock=0;
      TRAP(scheduler);
      return 0;
    }
    preempt_lock=0;
    return 0;
}
```

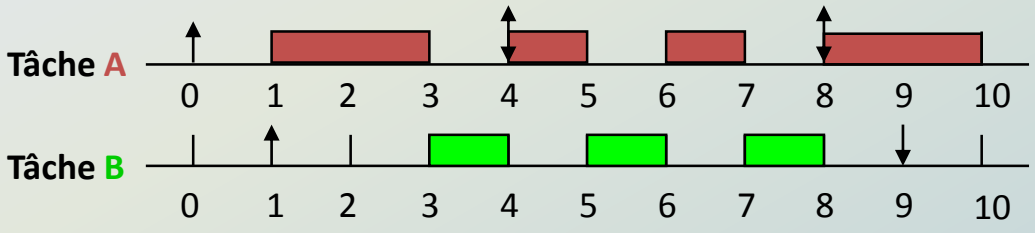
```
void scheduler ()
{
    intLock();
    ctxt_save (ct);
    if (preempt_lock)
    {
        intUnlock();
        ctxt_restore (ct); /* Préemption désactivée */
    }
    if (ct->state == RUNNG) ct->state = READY;
    int cprio = ct->prio;
    put_task(cprio, ct); /* Remise dans sa file */

    for (int prio = MAXPRIO ; prio >= 0 ; prio--)
    {
        int n = get_num(prio);
        while(n-- > 0)
        {
            ct = get_task(prio);
            if (ct->state == READY)
            {
                ct->state = RUNNG;
                intUnlock();
                ctxt_restore(ct); /* Élection */
            }
            else put_task(prio, ct);
        }
    }
}
```

# Séquence d'exécution

- **Séquence**
  - trace temporelle de l'exécution d'un ensemble de tâches (diagramme de Gant)
  - environnement de développement / débogage
- **Séquence valide**
  - une séquence est valide lorsque chaque tâche est exécutée dans le respect de ses contraintes de temps (activation, délai)

**Exemple**

| N° tâche | $r_i$ date départ | $c_i$ temps départ | $R_i$ délai | $T_i$ période | Échéance $r_i + R_i$ |
|---|---|---|---|---|---|
| A | 0 | 1 | 4 | 4 | |
| B | 1 | 3 | 8 | - | |



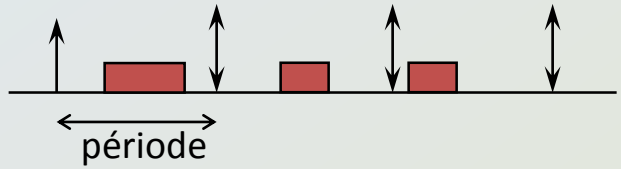

***QUESTIONS: d'après ce diagramme:***

1) Quel est le temps d'exécution de A? **2**
2) Quel est le temps d'exécution de B? **3**
3) Laquelle de ces assertions est vraie?

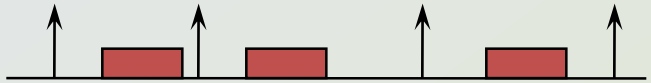
- ☐ A est plus prioritaire que B
- ☐ B est plus prioritaire que A
- ☑ A est probablement plus prioritaire que B
- ☐ B est probablement plus prioritaire que A

# Contraintes sur les tâches

- **Tâche périodique**

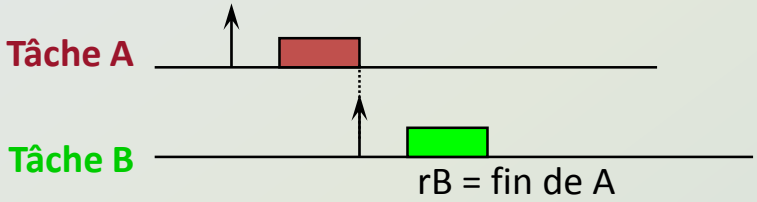



- **Tâche apériodique**

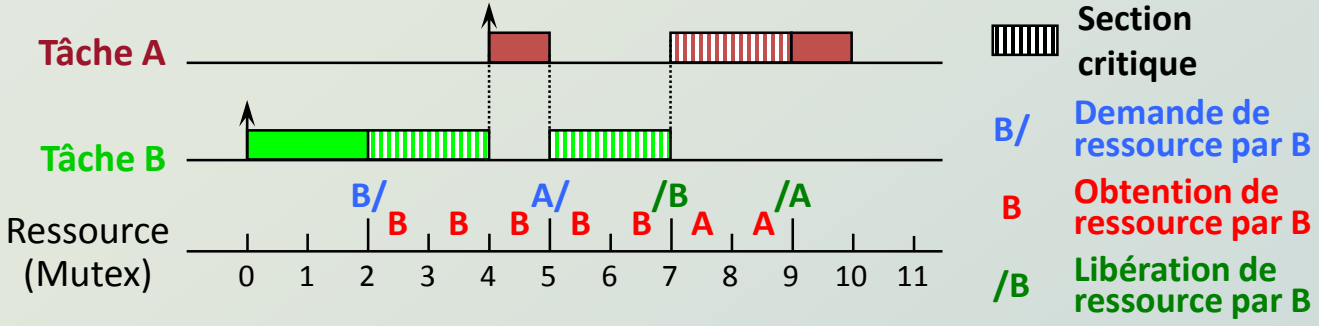


$d_{min}$ : intervalle miminum entre deux activations

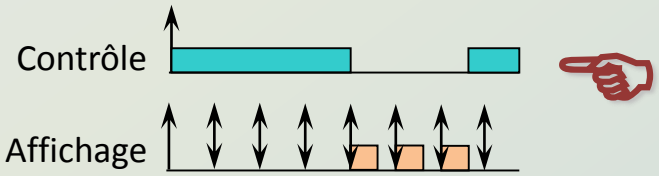
- **Précédence**

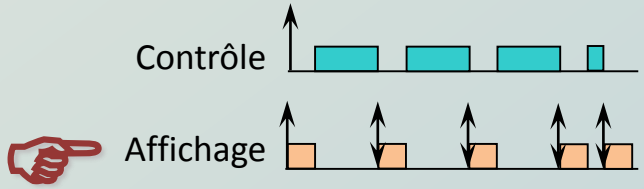



- **Exclusion mutuelle**




**Quelle tâche est prioritaire ?**

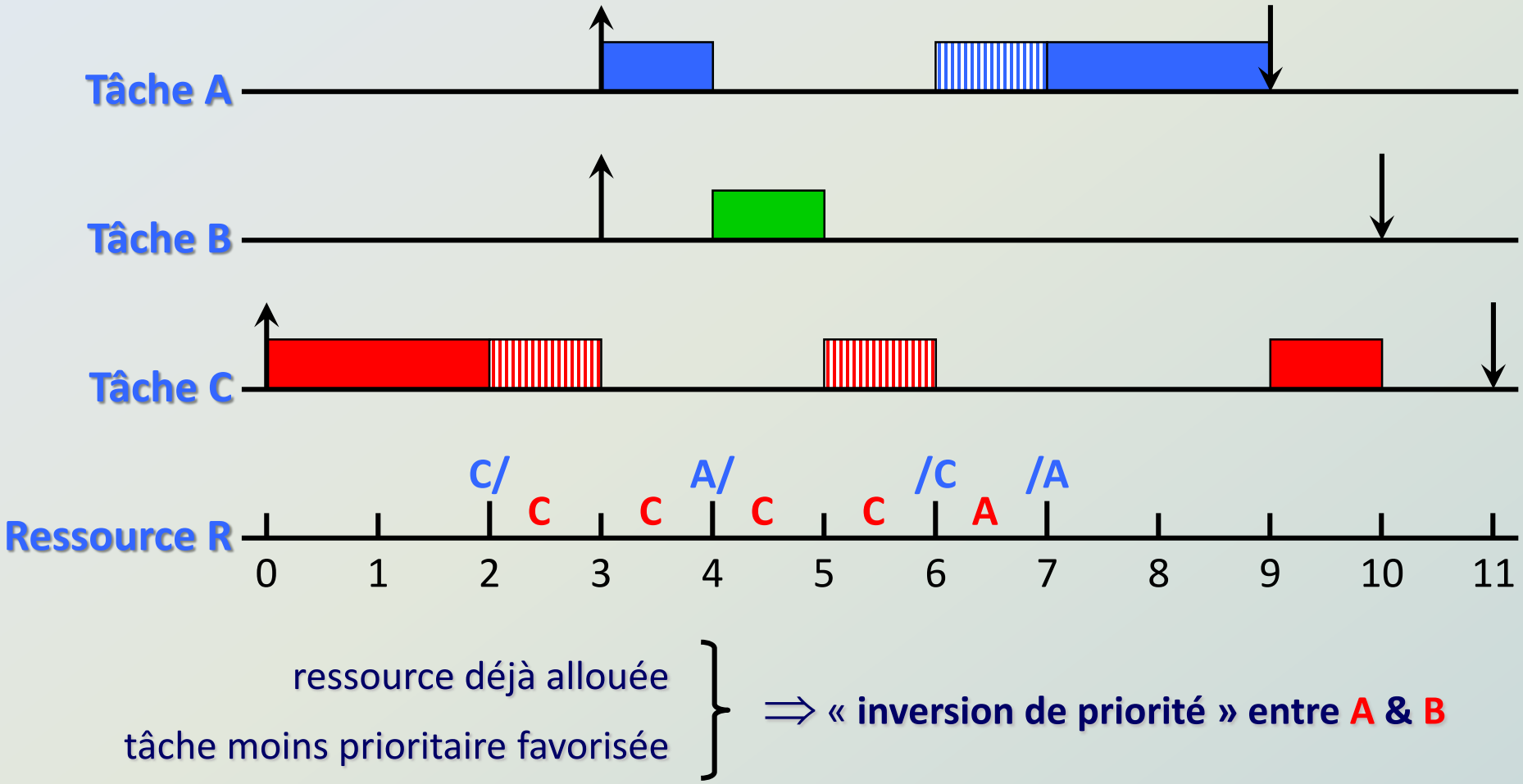


# Diagramme de Gant : exercice

- **Établissez les diagrammes de Gant correspondant à la description textuelle suivante:**
  - Soient A, B et C trois tâches telles que:
    priorité(A) > priorité(B) > priorité(C)
    Soit R une ressource accessible par A, B et C
    Soit t le temps en tics système ($t \in N$)
  - A est activée à t=30, B est activée à t=30 et C est activée à t=0
  - Les temps d'exécution de A, B et C sont respectivement de 40, 10 et 50
  - Les délais de A, B et C sont respectivement de 60, 70 et 110
  - C demande R après 20 tics d'exécution et la libère après 20 tics d'exécution alors que A demande R après 10 tics d'exécution et la libère après 10 tics d'exécution.

# Prise en compte des ressources critiques




ressource déjà allouée

tâche moins prioritaire favorisée

⇒ « **inversion de priorité** » entre **A** & **B**

# Protocole à héritage de priorité

**(priority inheritance protocol)**

- **Principe**
  - **La tâche en section critique hérite de la plus haute priorité parmi les tâches bloquées**
- **Héritage simple**
  - **L'héritage se fait par file d'attente d'accès aux ressources critiques**
- **Résultat**
  - **réduction du temps d'attente en section critique**

**Posix**

```
/* creating a mutex semaphore with priority inheritance */
pthread_mutexattr_t mutexAttr; pthread_mutex_t mutex;
pthread_mutexattr_init(&mutexAttr);
pthread_mutexattr_setprotocol(&mutexAttr, PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&mutex, &mutexAttr);
```

☞ Réécrire **`mLock(M*)`** et **`mUnlock(M*)`** pour implémenter ce protocole.
**`mget_task`** renvoie la tâche de plus haute priorité.
macro **`mpeek_task(m)`** renvoie la tâche de plus haute priorité sans la retirer de la file associée à **`m`**.

# Protocole à héritage de priorité

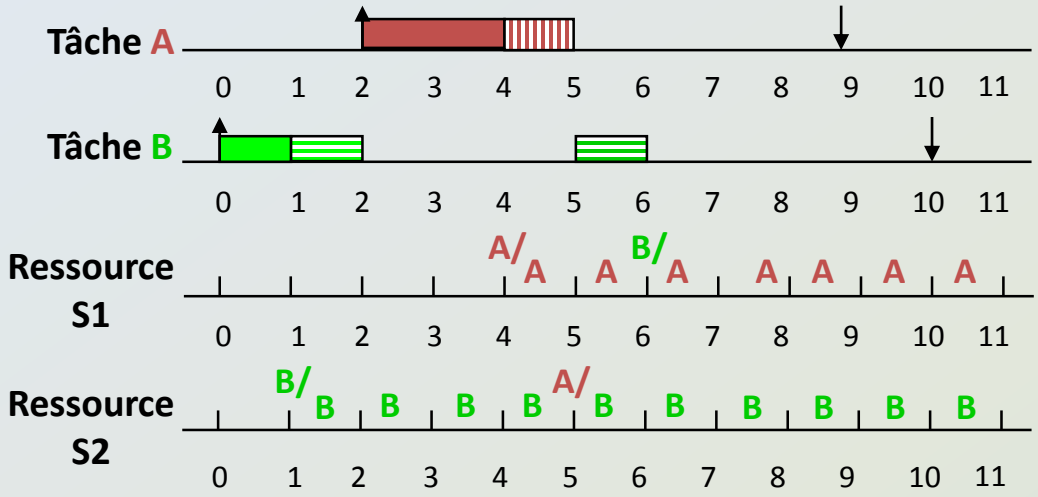
## (priority inheritance protocol)

```
struct T
{
    void* reg_PCS;
    void* reg_A[2];
    int   reg_R[2];
    State state;
    void* stack;
    short prio; /*init. à prio_org*/
    short org_prio;
};

void mUnlock(M* m)
{
    preempt_lock=1;
    if (m->owner != ct) return -1;
    m->owner = mget_task(m);
    if (m->owner != NULL)
    {
      m->owner->state = READY;
ct->prio = ct->org_prio;
      preempt_lock=0;
      TRAP(scheduler);
    }
    preempt_lock=0;
    return 0;
}
```

```
int mLock (M* m)
{
    preempt_lock=1;
    if (m->owner == ct) return -1;
    if (m->owner != NULL)
    {
        ct->state = BLCKD;
        mput_task(m, ct);
        if (ct->prio > m->owner->prio)
        {
            m->owner->prio = ct->prio;
        }

        preempt_lock=0;
        TRAP(scheduler);
        preempt_lock=1;
    }

    m->owner = ct;
    preempt_lock=0;
    return 1;
}
```

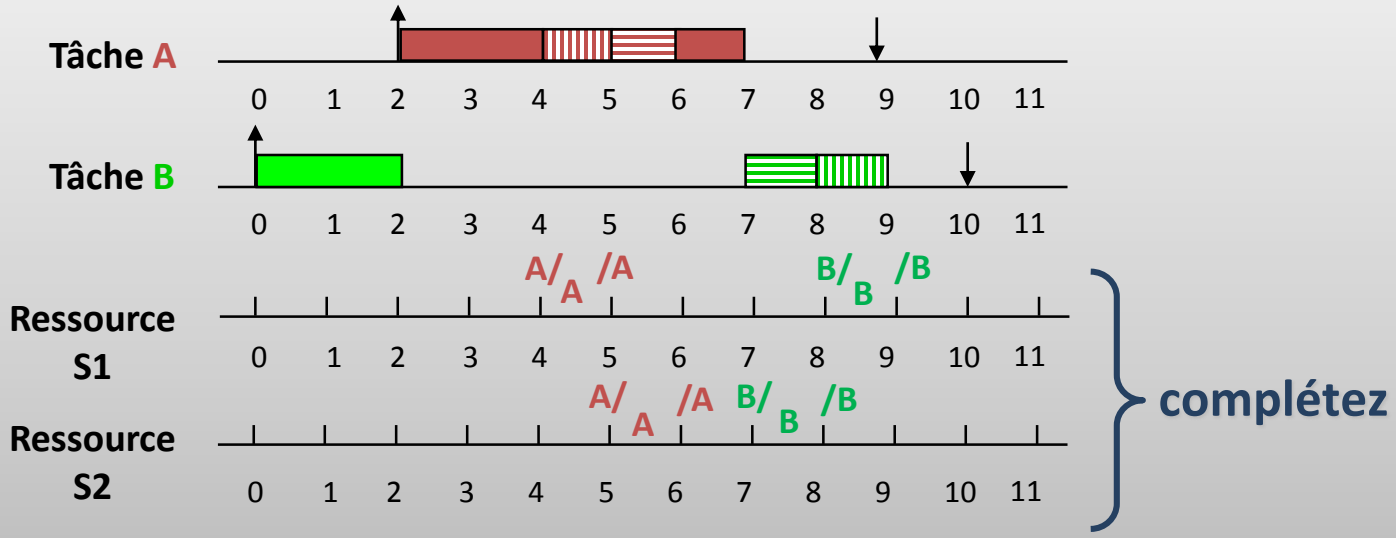
# Interblocage

**(deadlock)**




Priorité(A) > priorité(B)

Sections critiques




**complétez**

# Prévention de l'interblocage

- Analyse de code
- Contraintes de programmation
  - Prise / libération atomique des ressources
  - Prise / libération dans l'ordre des ressources
  - Timeout