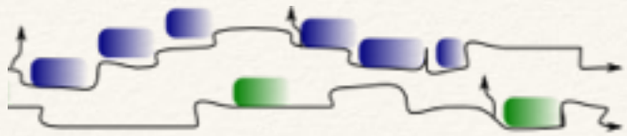
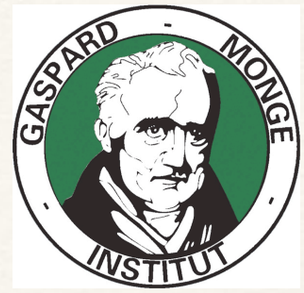





# *Temps Réel :*
# *De la théorie a l'application avec Java*

*Serge MIDONNET*

*Serge.Midonnet@univ-paris-est.fr*

*Janvier, 2017*

# Plan

- *Motivations : RT et Java*
- *Modèles de tâches.*
- *Ordonnancement.*
- *Faisabilité / Ordonnançabilité*
- *Tolérance aux fautes / Défaillances*
- *Partage de ressources: Synchronisations*
- *Gestion de la mémoire*
- *Conclusion: autres problématiques*

# Motivation

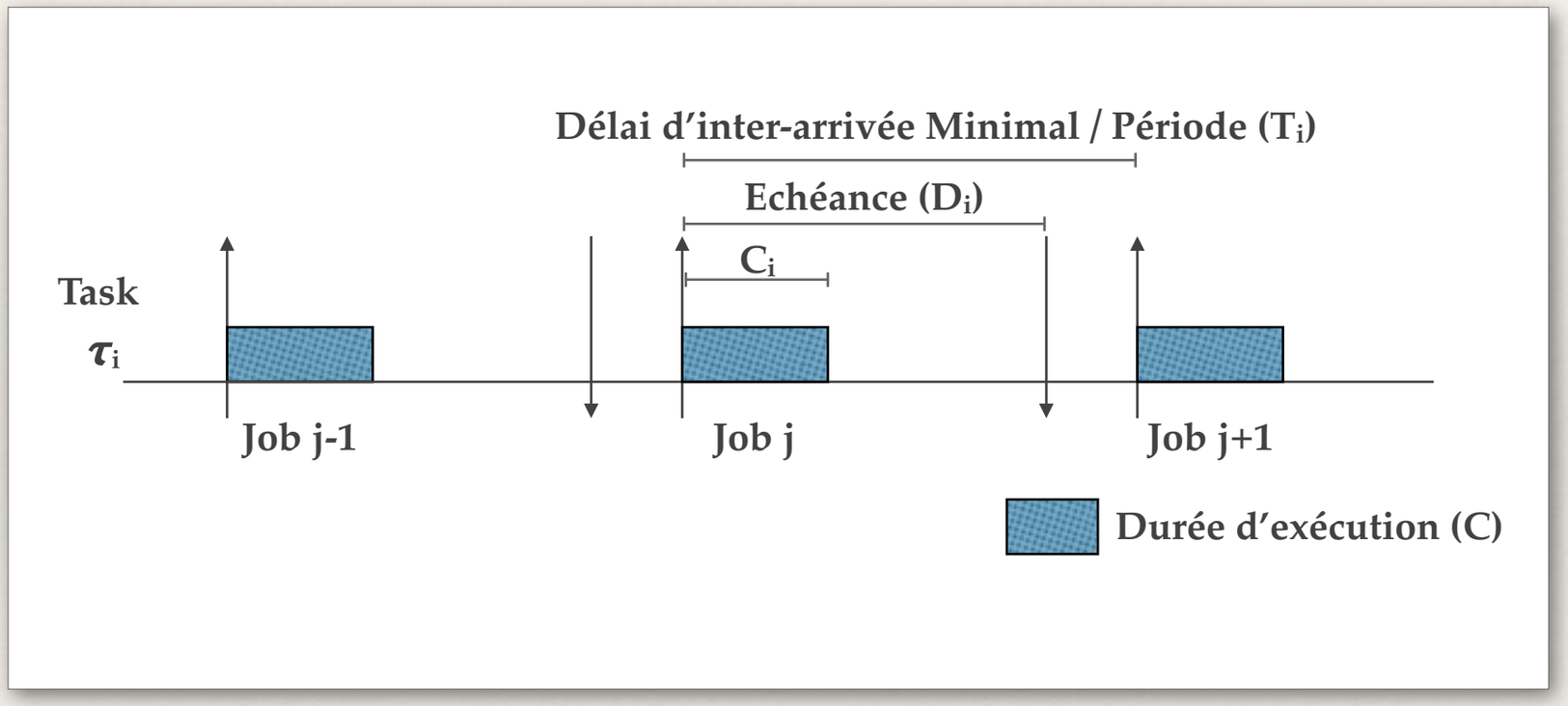
- *Un système temps réel est défini comme « un système dont la validité ne dépend pas que de la validité des résultats mais de la date à laquelle ils sont fournis ».*
- *Appliquer des contraintes temporelles vérifier/prédire qu'elles sont respectées*
- *On peut les diviser en systèmes « Durs » « Hard » et « Mou » « Soft »*
  - ***Hard**: Le non respect d'une échéance conduit à la défaillance du système (avionique, nucléaire, ferroviaire).*
  - ***Soft**: Le non respect d'un ou plusieurs échéances est tolérable (multimédia).*

# RTSJ

- *Développement avec Java (Real-time Specification For Java : RTSJ).*
- *Apports :*
  - *choix de l'ordonnançeur (priorité fixe / priorité dynamique)*
  - *définition des paramètres d'ordonnancement et du modèle d'activation des tâches.*
  - *prédiction possible du respect des contraintes temporelles (analyse de faisabilité).*
  - *préciser un comportement en cas de fautes temporelles / défaillances.*
  - *choix du protocole de synchronisation (tâches dépendantes)*
- *Contraintes :*
  - *développement plus complexe.*
  - *modèles de mémoire plus contraignants (si HRT).*
  - *nécessite une VM adaptée*

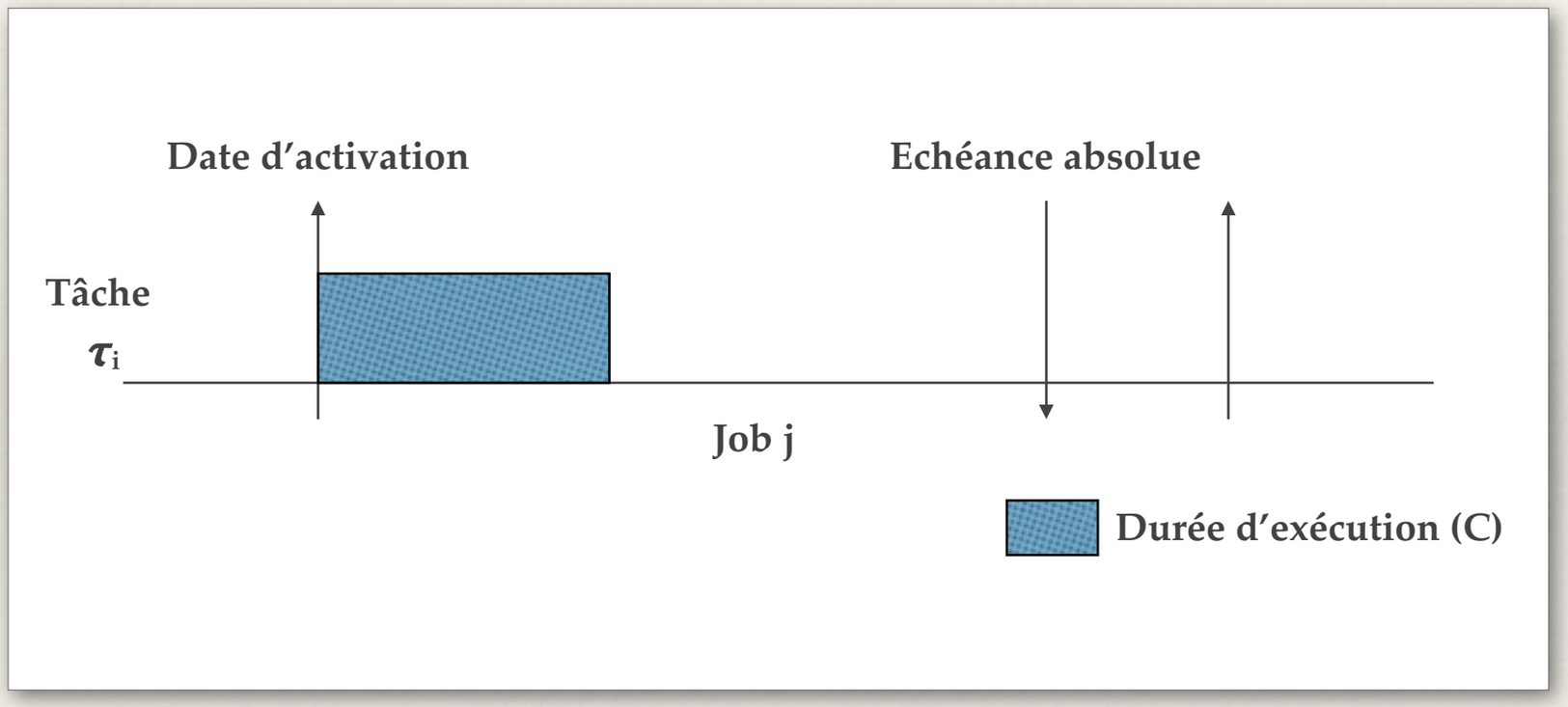
# Modèle de Tâche Périodique/Sporadique

- *Une tâche périodique génère une séquence illimitée de jobs.*



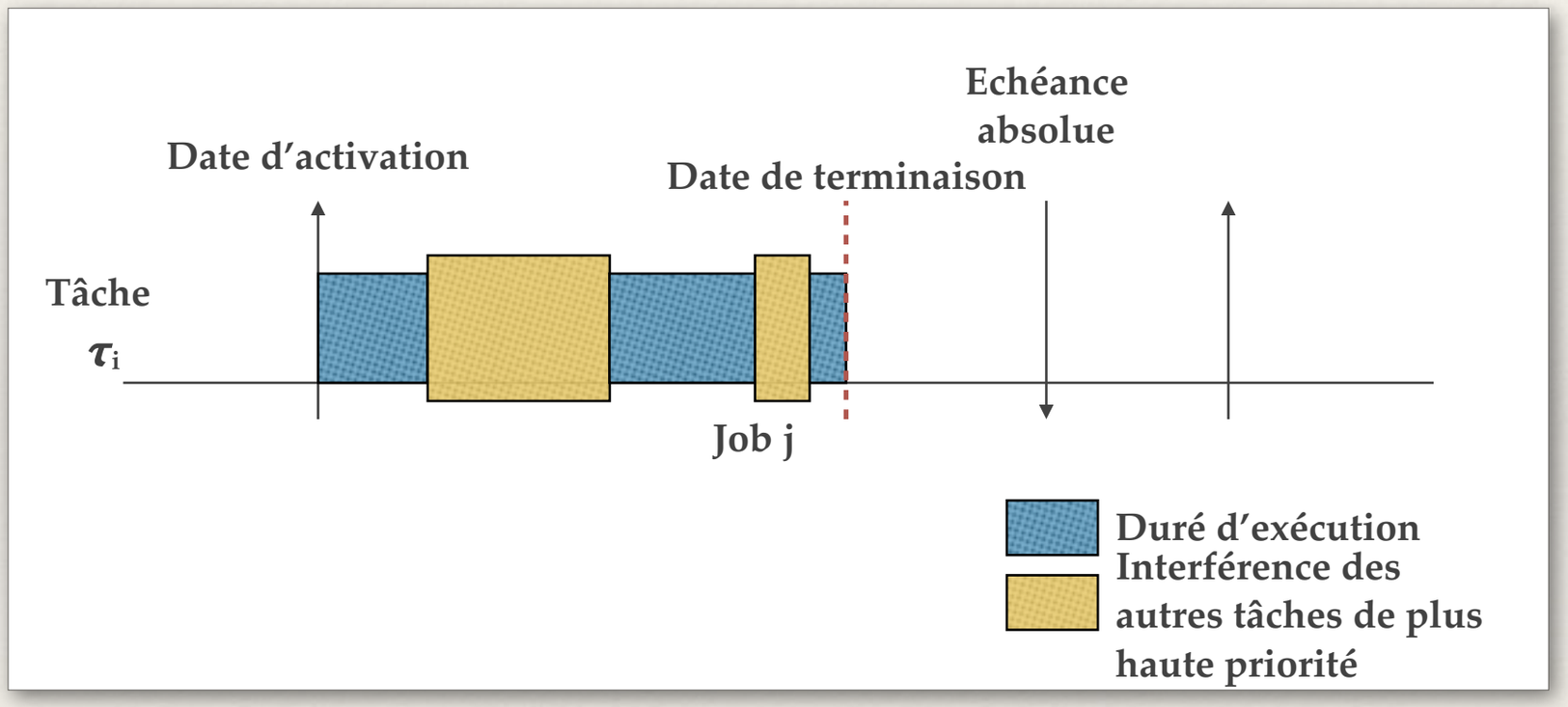
# Modèle de Tâche / Contrainte Temporelle (échéance)

- *Chaque job est caractérisé par une date d'activation, une durée d'exécution et une date d'échéance*



# Modèle de Tâche / Temps de réponse

- *L'exécution d'un Job peut être interrompue par un ou plusieurs jobs plus prioritaires (préemption)*



# définitions

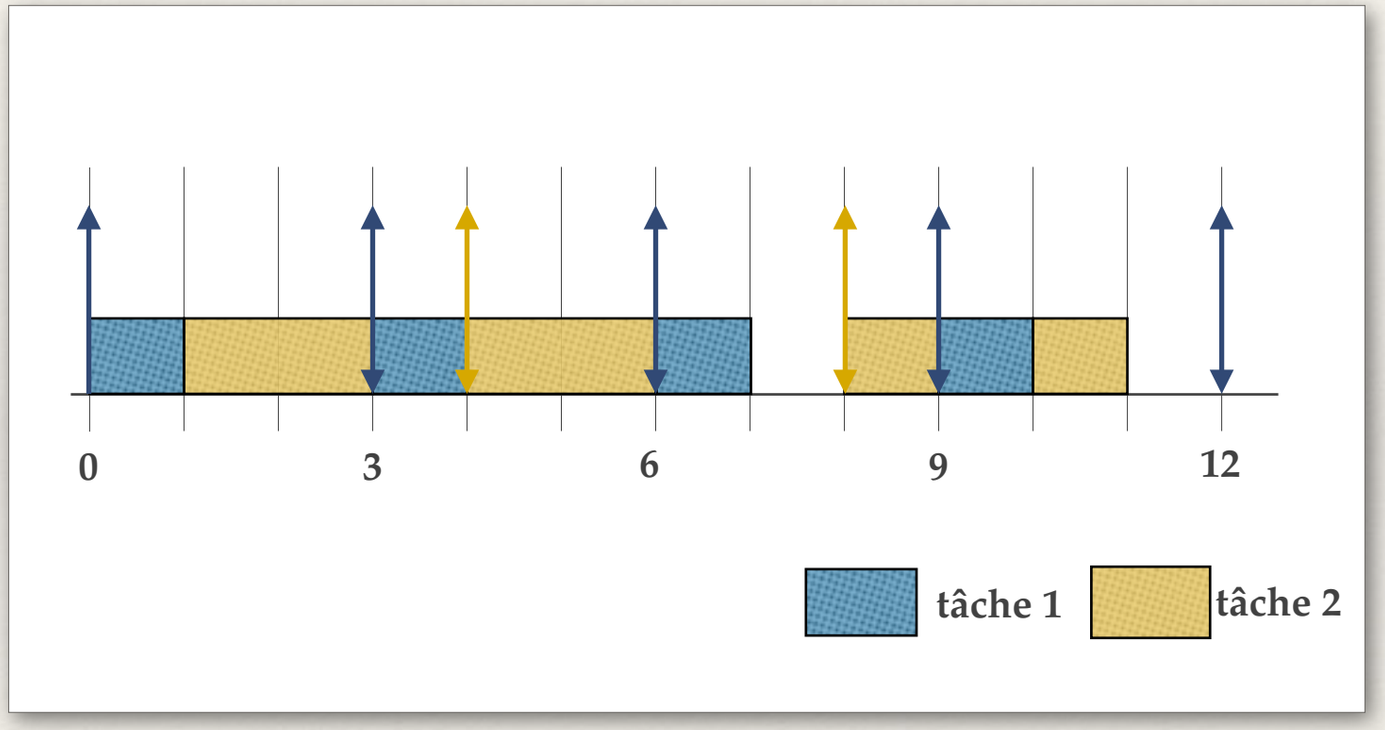
- *Tâches indépendantes: pas de relation de dépendance (ni synchronisation ni relation de précédence).*
  - *Tâche concrète: On connait son instant de première activation. Sinon la tâche est abstraite.*
- *Système préemptif: l'activation d'un job de priorité élevé interrompt l'exécution d'un job de priorité plus faible.*
- *Echéances*
  - *implicites (échéance sur requête): Di = Ti*
  - *contraintes: Di < Ti*
  - *arbitraires: Di > Ti*
- *Activation Synchrone: Toutes les tâches activées au même instant. Considérée comme pire cas d'activation si échéances implicites ou contraintes (tâches indépendantes).*
- *Modèle d'activation*
  - *Périodique: la période entre deux jobs est fixe.*
  - *Sporadique: la période entre deux jobs est variable mais bornée (borne inférieure)*

# Ordonnancement

- *une tâche s'exécutant sur une processeur unique se voit affecter une priorité en fonction de la règles d'affectation (RM, DM).*

**Système de Tâches:**
**$\tau_1$ = (1,3,3)**
**$\tau_2$ = (2,4,4)**

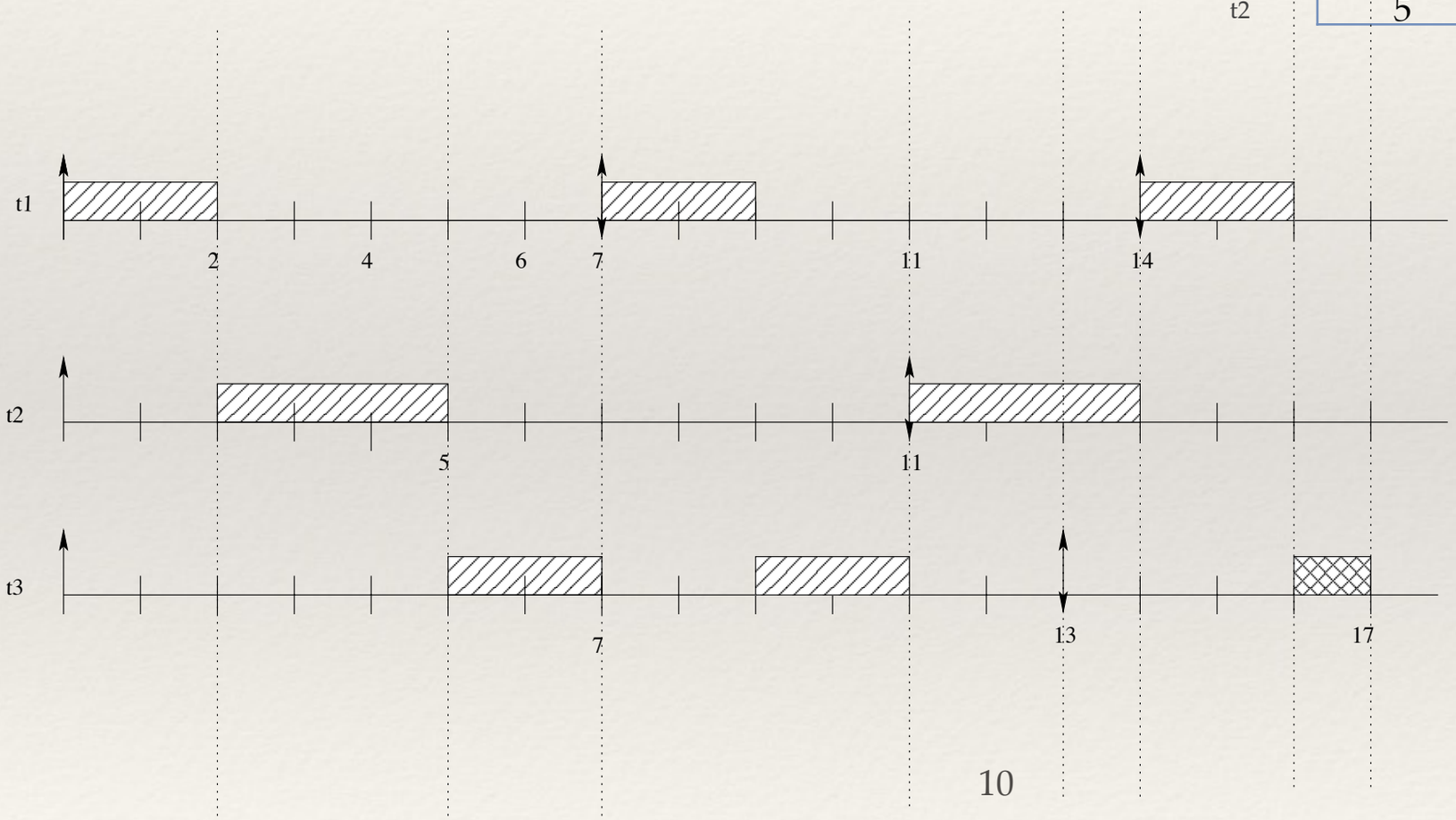
**$\tau_1$ a une priorité plus élevée que $\tau_2$ (DM/RM)**



# Ordonnancement Priorité Fixe au niveau des Tâches

- *La période la plus petite la priorité la plus haute = Rate Monotonic*

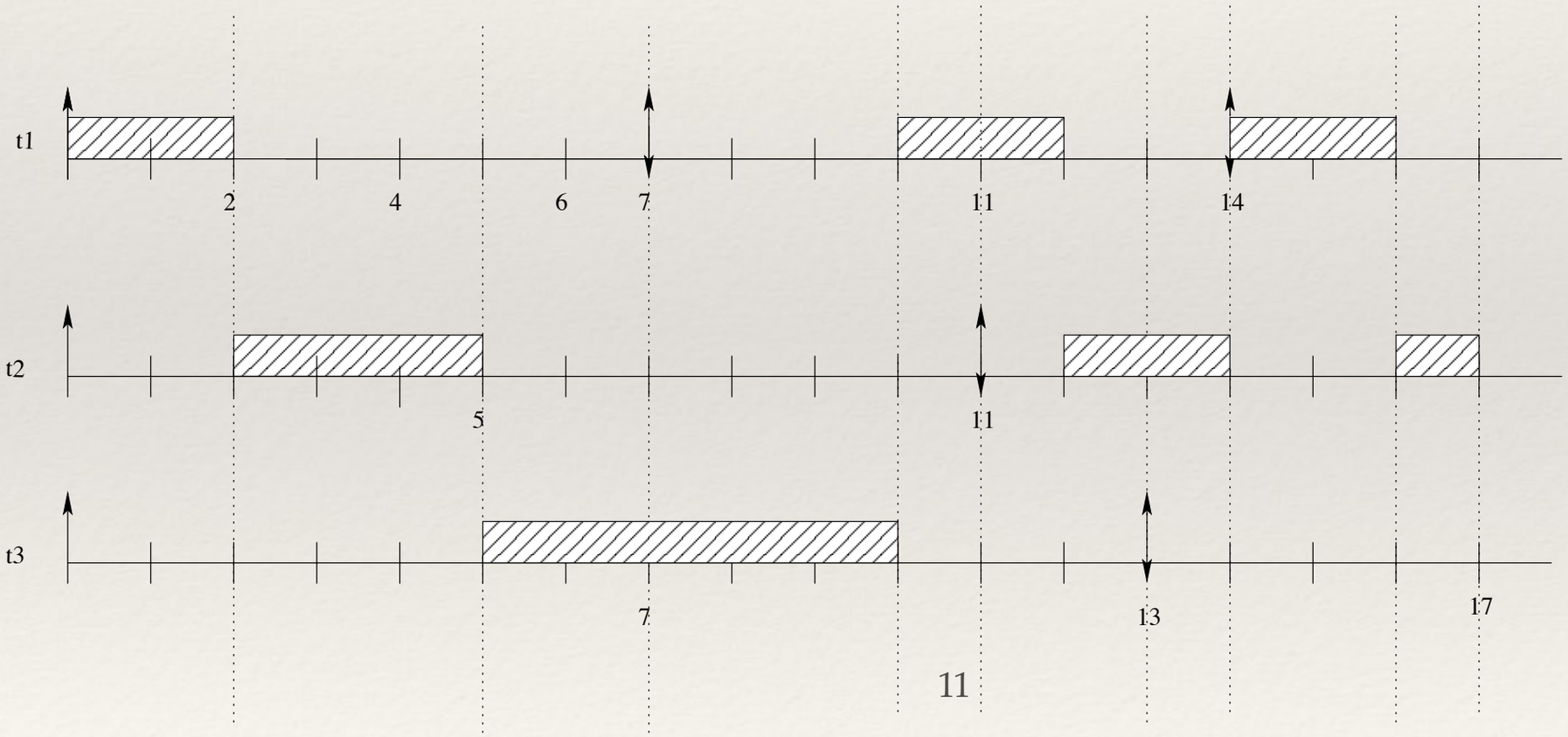
| | Ci | Ti | Di |
|---|---|---|---|
| t1 | 2 | 7 | 7 |
| t2 | 3 | 11 | 11 |
| t2 | 5 | 13 | 13 |



# Ordonnancement Priorité Fixe au niveau des Jobs

- *La période la plus petite la priorité pour l'échéance plus proche = EDF*

| | Ci | Ti | Di |
|---|---|---|---|
| t1 | 2 | 7 | 7 |
| t2 | 3 | 11 | 11 |
| t2 | 5 | 13 | 13 |



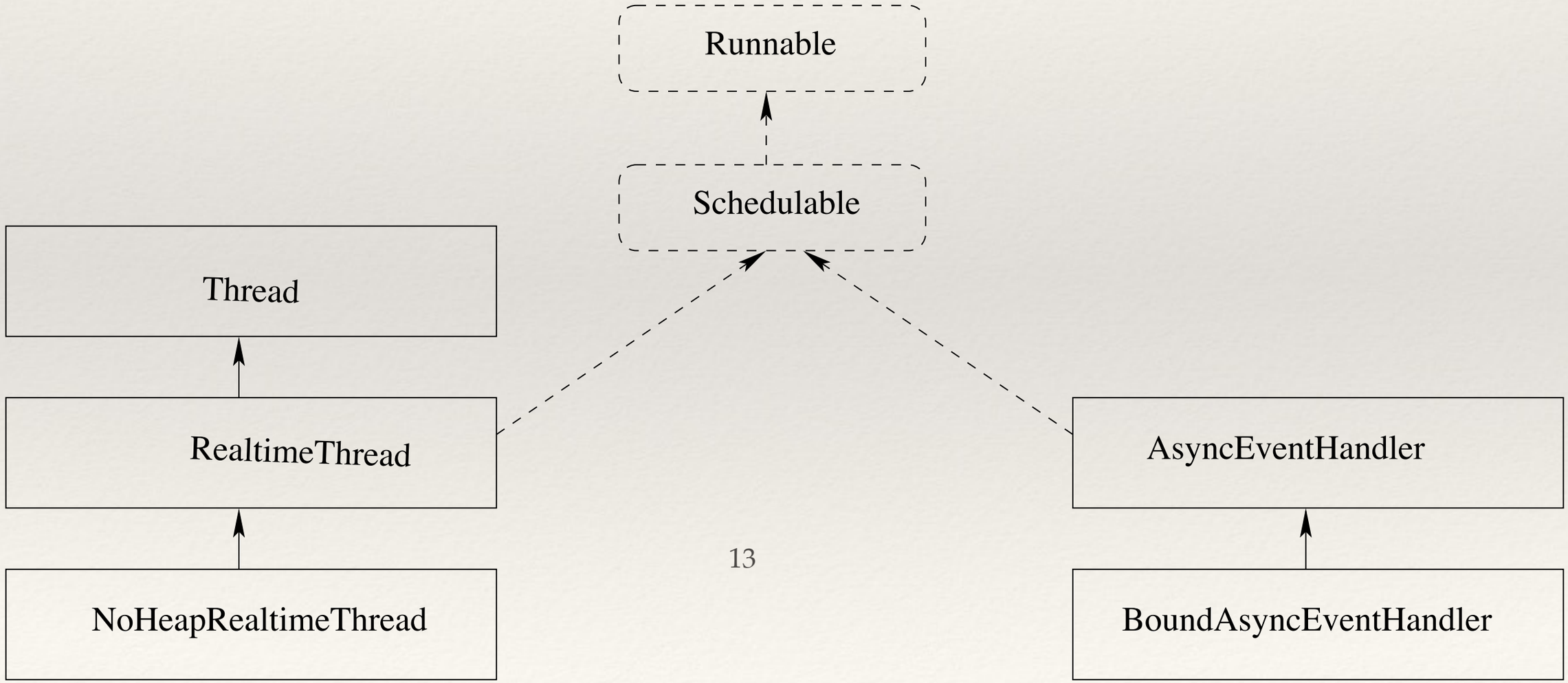
# Ordonnancement / RTSJ

- *Affectation des priorités (principaux algorithmes/règles): Rate Monotonic, Deadline Monotonic, Earliest Deadline First, FIFO.*
- *RTSJ: un ordonnanceur à priorités fixes est obligatoire: FP/HPF*
  - *classe PriorityScheduler (obligatoire)*
  - *autre classes (optionnelles) EDFScheduler*
    - *un minimum de 28 priorités*
    - *Prio > 10 threads temps réel / Prio < = 10 threads non temps réel*

```
Scheduler Scheduler.getDefaultScheduler();
void Scheduler.setDefaultScheduler(Scheduler s);
String Scheduler.getPolicyName();

public static void main(String [] args){
Scheduler.setDefaultScheduler(PriorityScheduler.instance());
```

# RTSJ / Entités Ordonnançables

*Classes RealtimeThread,NoHeapRealtimeThread, AsyncEventHandler, BoundedAsyncEventHandler.*
*Méthodes: setSchedulingParameters(), getSchedulingParameters()*



# RTSJ - RealTime Thread (oneshot)

*Classe RealtimeThread*
*Méthode: start().*

```
import javax.realtime.*;
public class LaPlusSimple {
        public static void main (String [] args){
                RealtimeThread lps = new RealtimeThread(){
                    public void run () {
                        System.out.println ("hello from LaPlusSimple");
                    }
                };
                lps.start();
        }
}
```

*Question: RealTime Thread. Quelle priorité par défaut ?*

# RTS - AsyncEventHandler

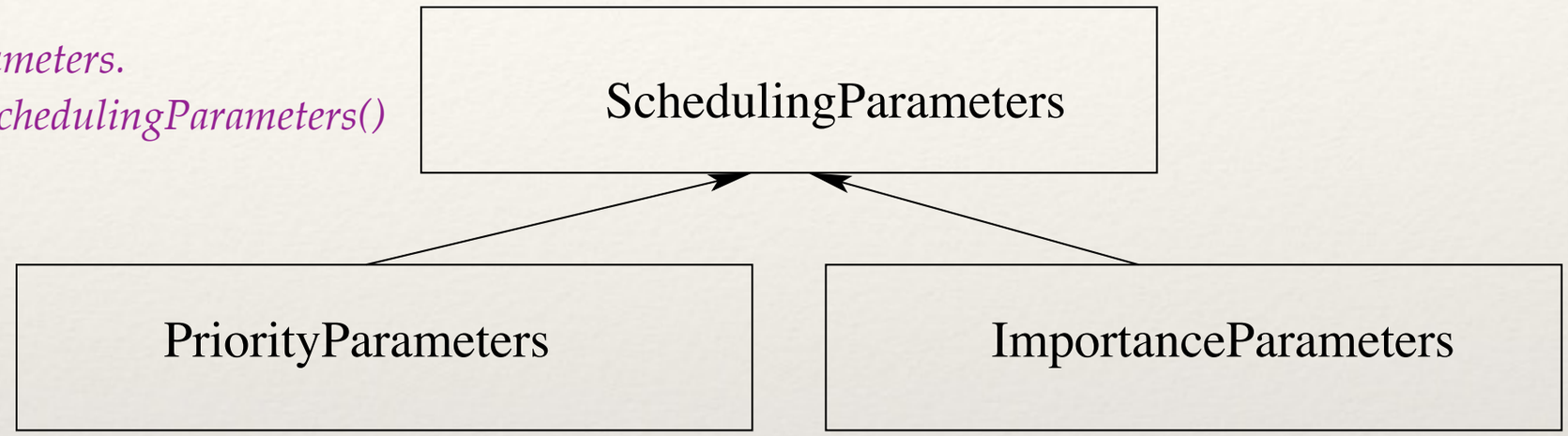
*Classe AsyncEventHandler, AsyncEvent.*
*Méthodes: addHandler(); fire()*

```
AsyncEventHandler aeh=new AsyncEventHandler(){
    public void handleAsyncEvent(){
        System.out.println("hello from laPlusSimpleAperiodic");
    }
};
AsyncEvent ae = new AsyncEvent();
ae.addHandler(aeh);
ae.fire();
```

*Note:*
*On peut associer plusieurs AEH à un même évènement;*
*On peut associer un même AEH à un plusieurs évènements ;*
*Dans ce cas on en pas capable d'indentifier la source de l'évènement*

# RTS - Scheduling Parameters

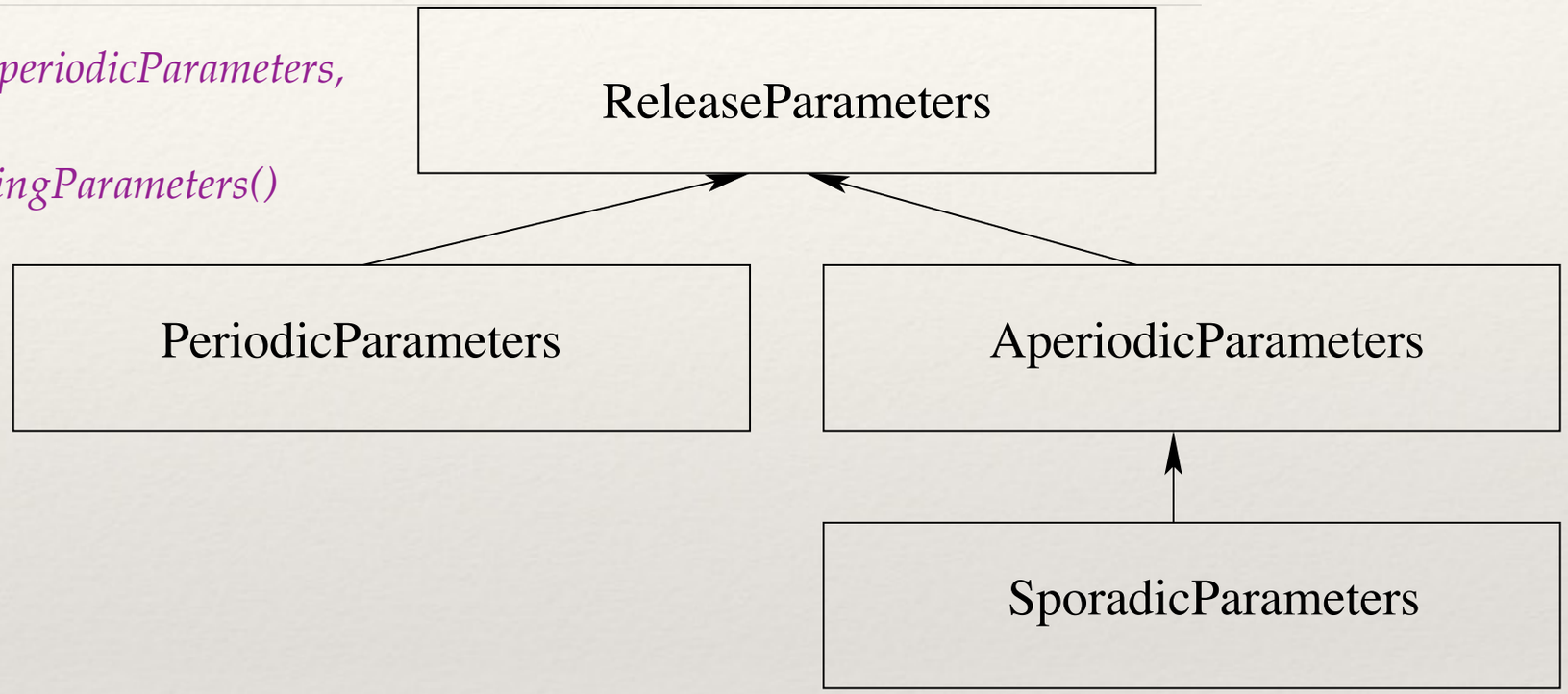
*Classes PriorityParameters,ImportanceParameters.*
*Méthodes: setSchedulingParameters(), getSchedulingParameters()*




```
import javax.realtime.*;
public class PasTropDure {
        public static void main (String [] args){
                RealtimeThread ptd = new RealtimeThread(new PriorityParameters(prio)){
                   public void run () {
                        System.out.println ("hello from PasTropDure");
                   }
                };
                lps.start();
        }
}
```

# RTS - Release Parameters

*Classes*: *ReleaseParameters,PeriodicParameters, AperiodicParameters, SporadicParameters.*
*Méthodes*: *setSchedulingParameters(), getSchedulingParameters()*




*PeriodicParameters(HighResolutionTime start,*
*RelativeTime period,*
*RelativeTime cost,*
*RelativeTime deadline,*
*AsyncEventHandler overrunHandler,*
*AsyncEventHandler missHandler)*

*SporadicParameters(RelativeTime minInterarrival,*
*RelativeTime cost,*
*RelativeTime deadline,*
*AsyncEventHandler overrunHandler,*
*AsyncEventHandler missHandler)*

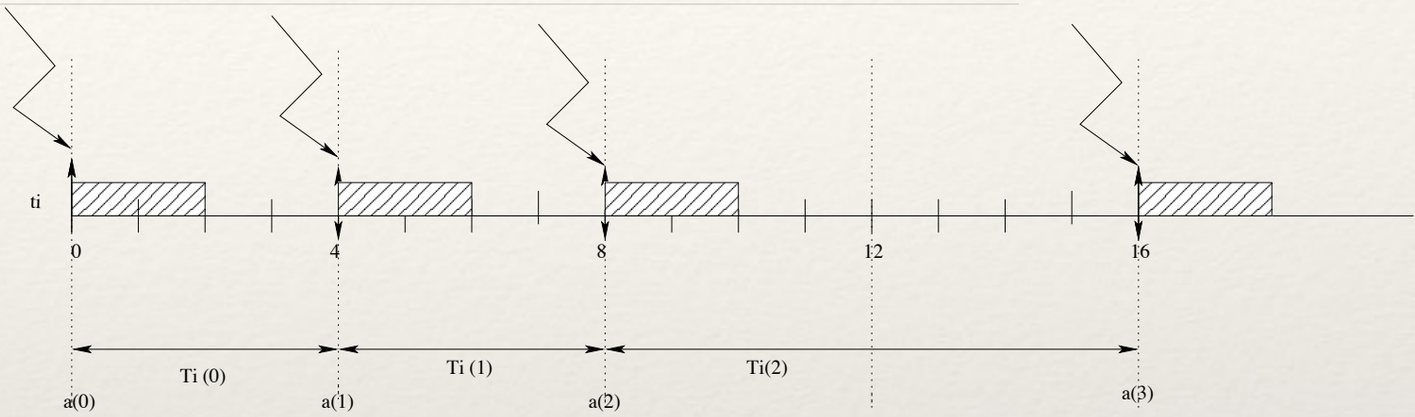
# RTS - Periodic Realtime Thread

*Méthodes: waitForNextPeriod();*

```
import javax.realtime.*;
public class SimplePeriodic {
    public static void main (String [] args){
        RealtimeThread spt = new RealtimeThread(
                new PriorityParameter(prio),
                new PeriodicParameters(
                    null, Offset "null=démarrage immédiat"
                    new RelativeTime(7000,0),  Période (Ti)
                    new RelativeTime(2000,0), Coût (Ci)
                    new RelativeTime(7000,0), Echéance (Di)
                    null, Handler de dépassement de cout "overrunHandler"
                    null) Handler de dépassement d'échéance "DeadlineMissHandler"
                ){
                public void run () {
                do  {
                    System.out.println ("hello from SimplePeriodic!");
                    } while (waitForNextPeriod()
                };
        spt.start();
    }
}
```

# RTS - Sporadic Realtime Thread

```
import javax.realtime.*;
public class SimpleSporadic {
    public static void main (String [] args){
        AsyncEventHandler aeh=new AsyncEventHandler(
            new PriorityParameters(20),
            new SporadicParameters(
                new RelativeTime(4000,0), “période min”
                new RelativeTime(1000,0), “coût”
                new RelativeTime(2000,0), “échéance”
    null, “overrunHandler”
    null); “deadlineMissHandler”
null, null, null, true,
null){
public void handleAsyncEvent(){
        System.out.println(”hello from SimpleSporadic”);
    }
};
```




```
        AsyncEvent ae = new AsyncEvent();
        ae.addHandler(aeh);
        ae.fire();
        sleep(800);
        ae.fire();
        sleep(800);
ae.fire();
}
}
```

# RTSJ

```
private final static ArrayList<String> releveValeur = new ArrayList<>();
```

```
PeriodicParameters MotorParams = new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(20000, 0),
                                                        new RelativeTime(5000, 0), new RelativeTime(20000, 0), null, null);

RealtimeThread controlMotor = new RealtimeThread(new PriorityParameters(14), MotorParams) {
public void run() {
    do {
    long start = System.currentTimeMillis();
    long end = start;

            MotorPort.A.controlMotor(POWER_MOTOR, 1);
            MotorPort.A.resetTachoCount();

            while (MotorPort.A.getTachoCount() <= ROTATE_2M) {
            end = System.currentTimeMillis();

            MotorPort.A.controlMotor(0, 3);

  } while (waitForNextPeriod());
```

```
PeriodicParameters ColorSensorParams = new PeriodicParameters(new RelativeTime(1_000, 0),
                                                              new RelativeTime(8_000, 0), new RelativeTime(4_000, 0), new RelativeTime(8_000, 0),
null, null);

RealtimeThread colorControl = new RealtimeThread(new PriorityParameters(16), ColorSensorParams) {
public void run() {

    ColorSensor colorsensor = new ColorSensor(SensorPort.S2);
    colorsensor.initWhiteBalance();
    do {
            for (int loop = 0; loop < 10; ++loop) {
            waitMillis(350);
            RConsole.println("Color sensor: capture nb " + loop);
            releveValeur.add("Color sensor: " + colorsensor.getColorNumber());
            }
} while (waitForNextPeriod());
```

# Ordonnançabilité

- *analyse basée sur des conditions de faisabilité*
  - *condition de charge (condition nécessaire):*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

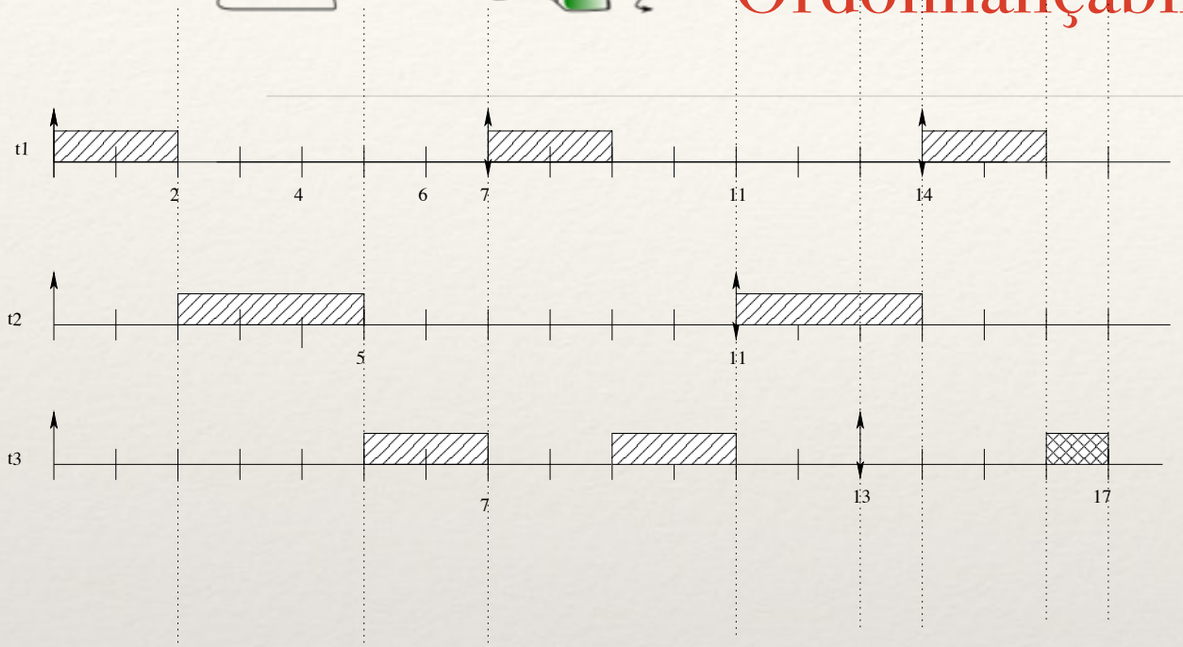
  - *condition de charge (condition suffisante):*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

- *analyse de temps de réponse (condition nécessaire et suffisante):*

$$w(t) = C_i + \sum_{k \in HP(j)} \left\lceil \frac{t}{T_k} \right\rceil \times C_k \quad (1)$$

# Ordonnançabilité / condition de charge



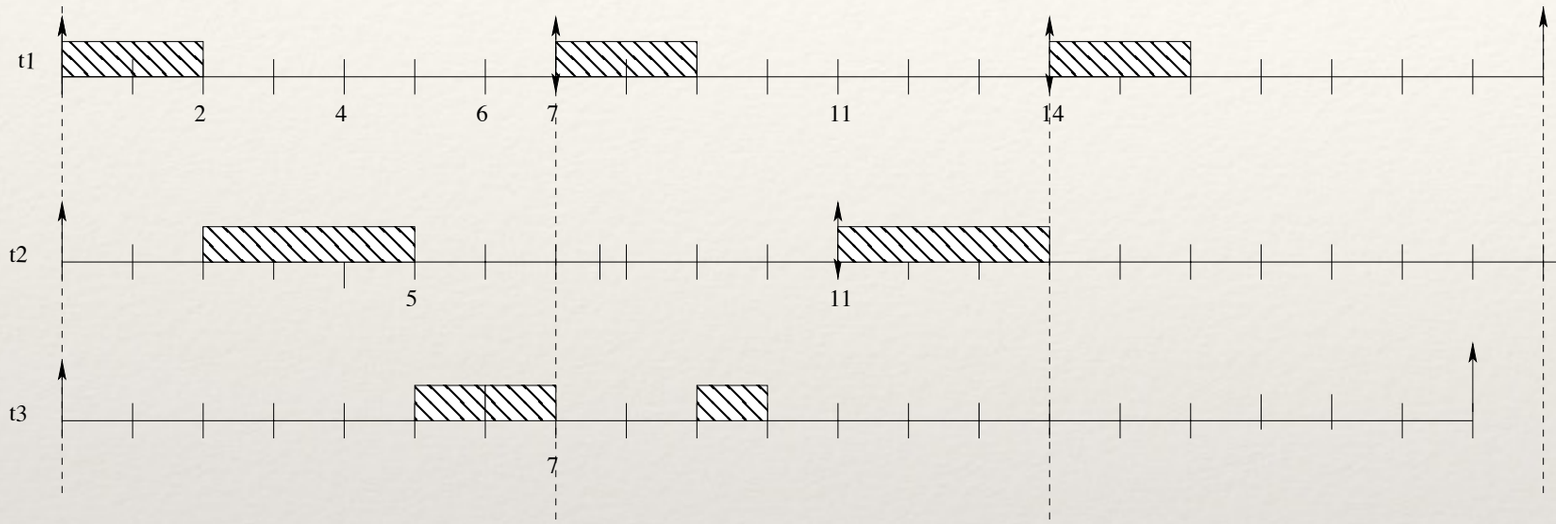

| | | |
|---|---|---|
| 2 | 7 | 7 |
| 3 | 11 | 11 |
| 5 | 13 | 13 |

- *U=2/7 + 3/11 + 5/13 = 0,943*
- *n(2**1/n-1)=0,80 -> U > 0,80 et U < 1*
- *La condition suffisante n'est pas respectée: IMPOSSIBLE de DECIDER*

| | | |
|---|---|---|
| 2 | 7 | 7 |
| 3 | 11 | 11 |
| 5 | 17 | 17 |

- *U=2/7+3/11+5/17=0,852*
- *La condition suffisante n'est pas respectée: IMPOSSIBLE de DECIDER*

# Faisabilité / Analyse des temps de réponse




| | | |
|---|---|---|
| 2 | 7 | 7 |
| 3 | 11 | 11 |
| 3 | 20 | 20 |

t1

| t | 1 | 2 |
|---|---|---|
| w(t) | 2 | 2 |

t2

| t | 1 | 5 |
|---|---|---|
| w(t) | 5 | 5 |

t3

| t | 1 | 8 | 10 |
|---|---|---|---|
| w(t) | 8 | 10 | 10 |



$$w(t) = C_i + \sum_{k \in HP(j)} \left\lceil \frac{t}{T_k} \right\rceil \times C_k \qquad (1)$$

$R_1(1)$

$w(1) = 2$

$w(2) = 2$

$R_2(1)$

$w(1) = 3 + \left\lceil \frac{1}{7} \right\rceil \times 2 = 5$

$w(5) = 3 + \left\lceil \frac{5}{7} \right\rceil \times 2 = 5$

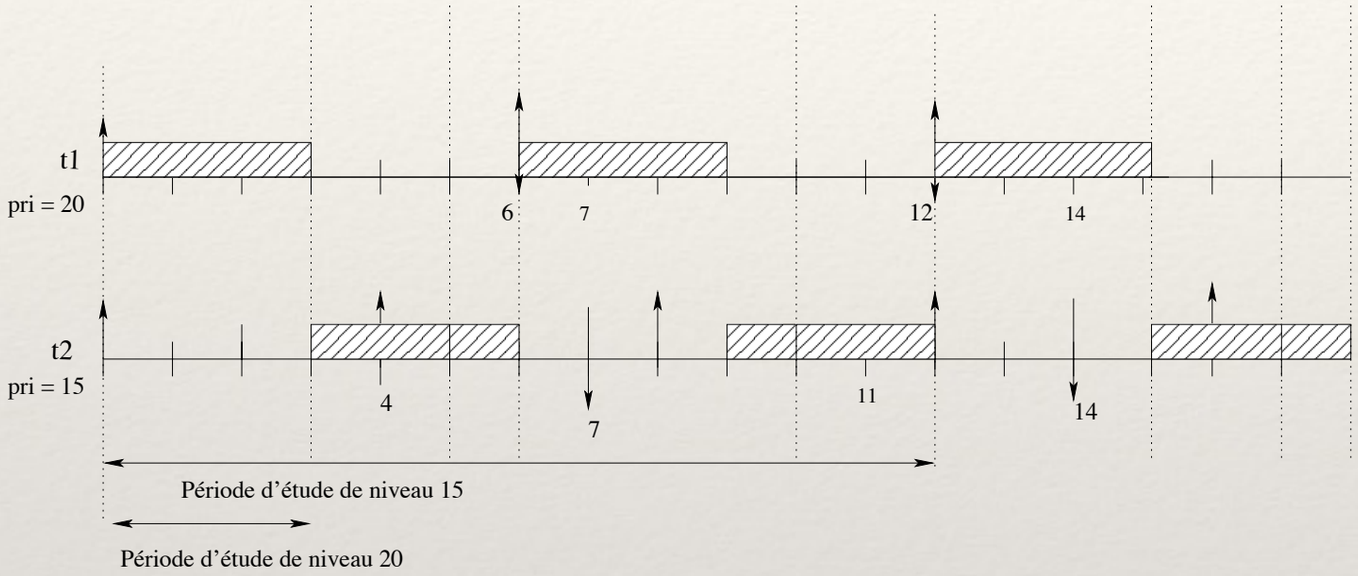
$R_3(1)$

$w(1) = 3 + \left\lceil \frac{1}{7} \right\rceil \times 2 + \left\lceil \frac{1}{11} \right\rceil \times 3 = 8$

$w(2) = 3 + \left\lceil \frac{8}{7} \right\rceil \times 2 + \left\lceil \frac{8}{11} \right\rceil \times 3 = 10$

$w(3) = 3 + \left\lceil \frac{10}{7} \right\rceil \times 2 + \left\lceil \frac{10}{11} \right\rceil \times 3 = 10$

# Faisabilité / Analyse des temps de réponse



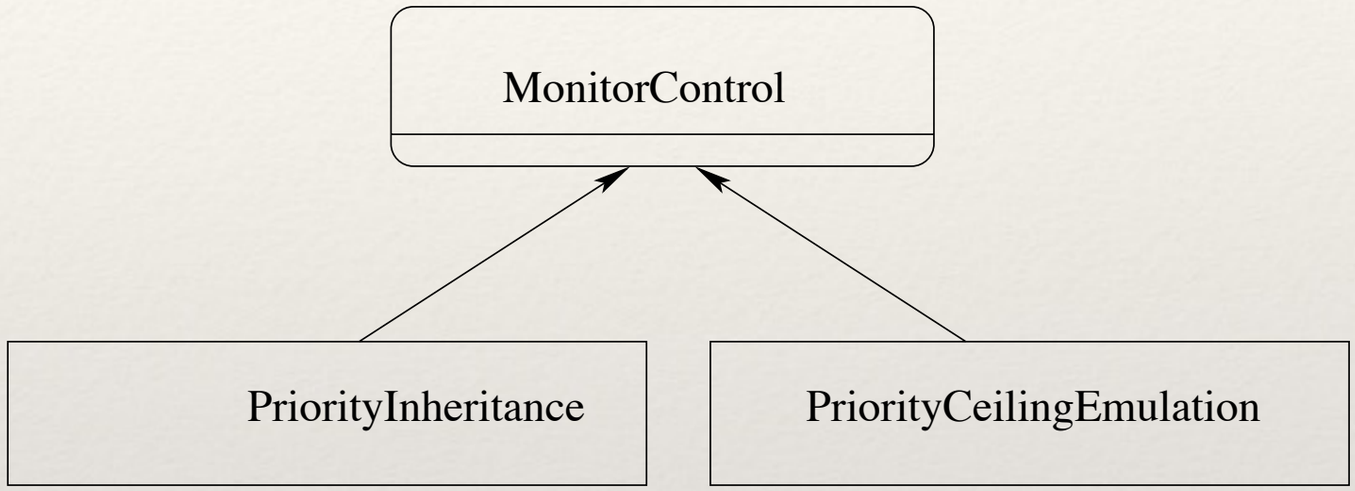

$$CT_i(m) = min\{t > 0 \mid w(t)\} \tag{1}$$

$$w(t) = m \times C_i + \sum_{k \in HP(j)} \left\lceil \frac{t}{T_k} \right\rceil \times C_k \tag{2}$$

$$R_i(m) = CT_i(m) - (m-1) \times T_i \tag{3}$$

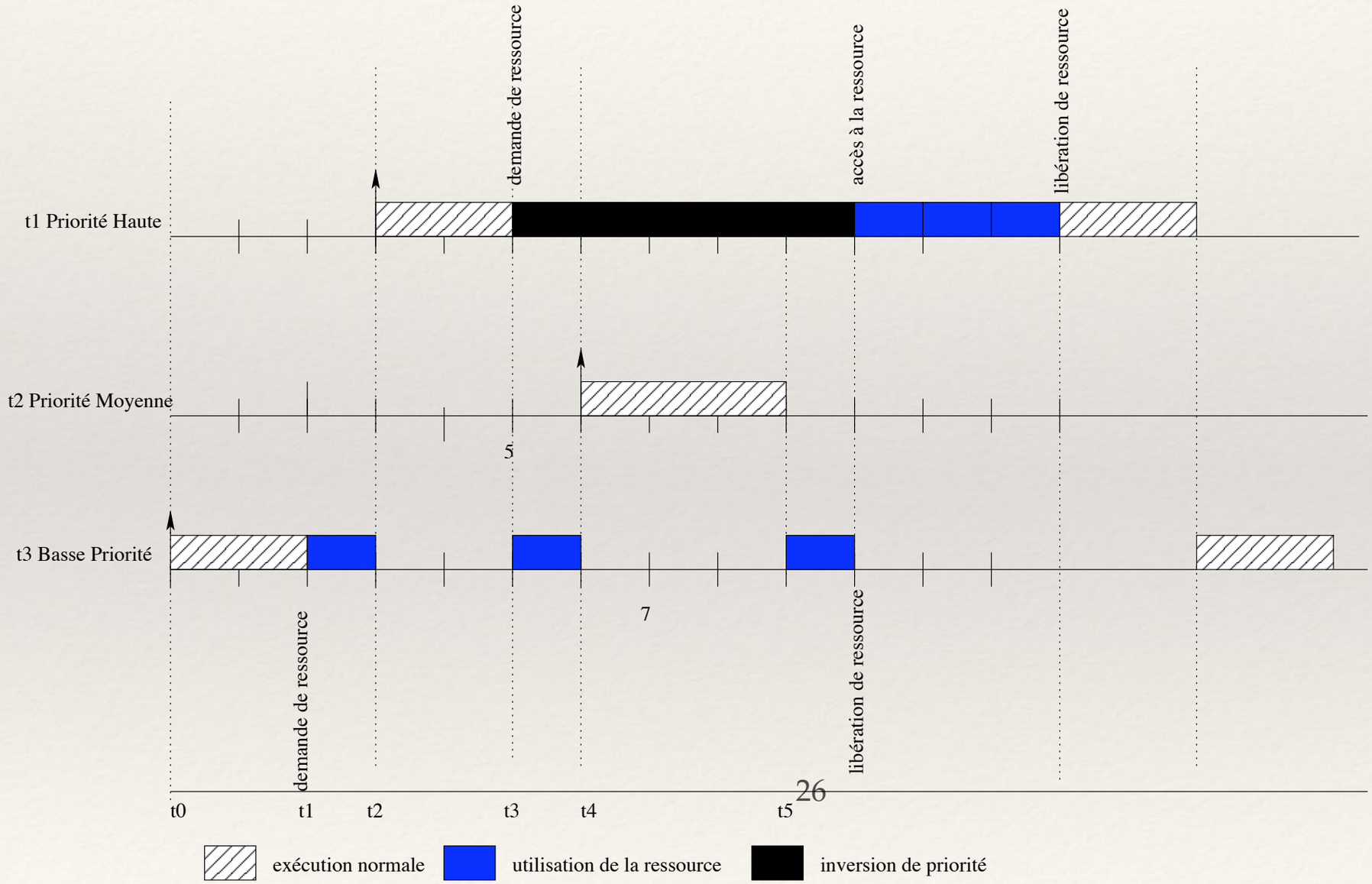
$$R_i = \max_{m=1 \ldots Q_i - 1} (R_i(m) \tag{4}$$

# Synchronisations / Moniteurs

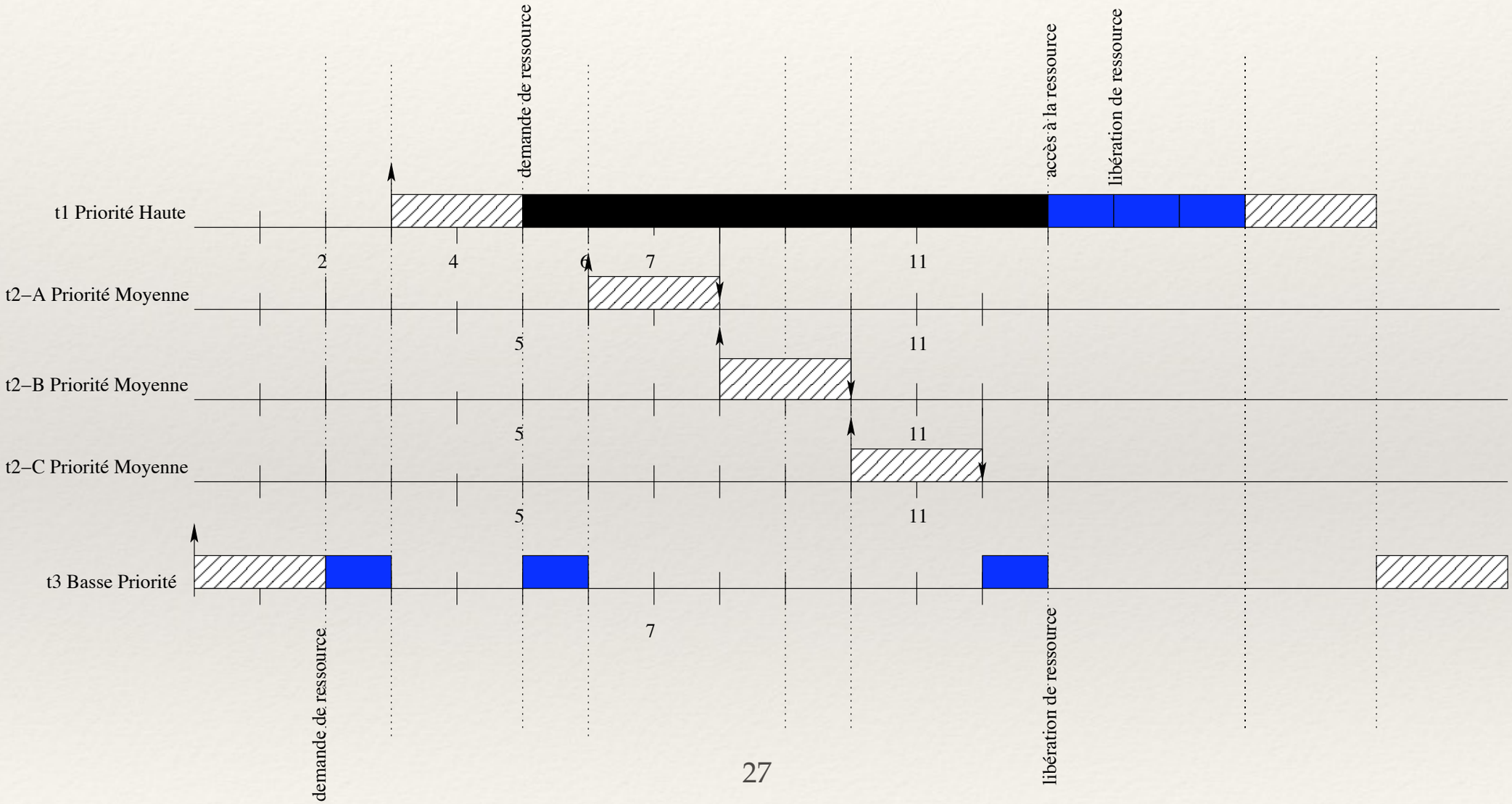



```
import javax.realtime.*;

public class SimpleMonitor{

public static void main(String[] args){

java.lang.Object res1 = new java.lang.Object();
PriorityInheritance pip = new PriorityInheritance.instance();
MonitorControl.setMonitorControl(res1,pip);
      MonitorControl policy = res1.getMonitorControlPolicy();
}
}
```

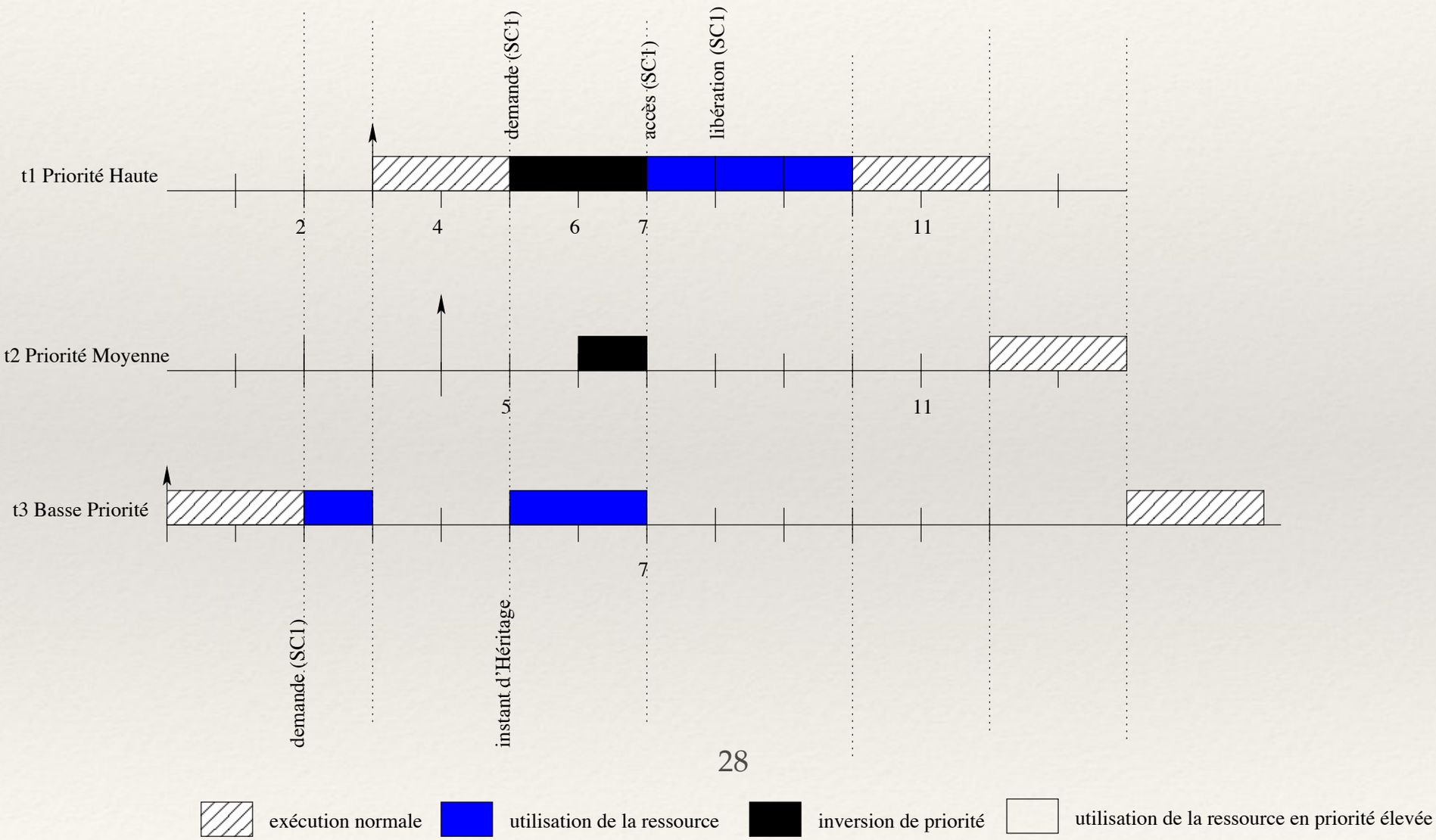
# Inversion de Priorité non bornée



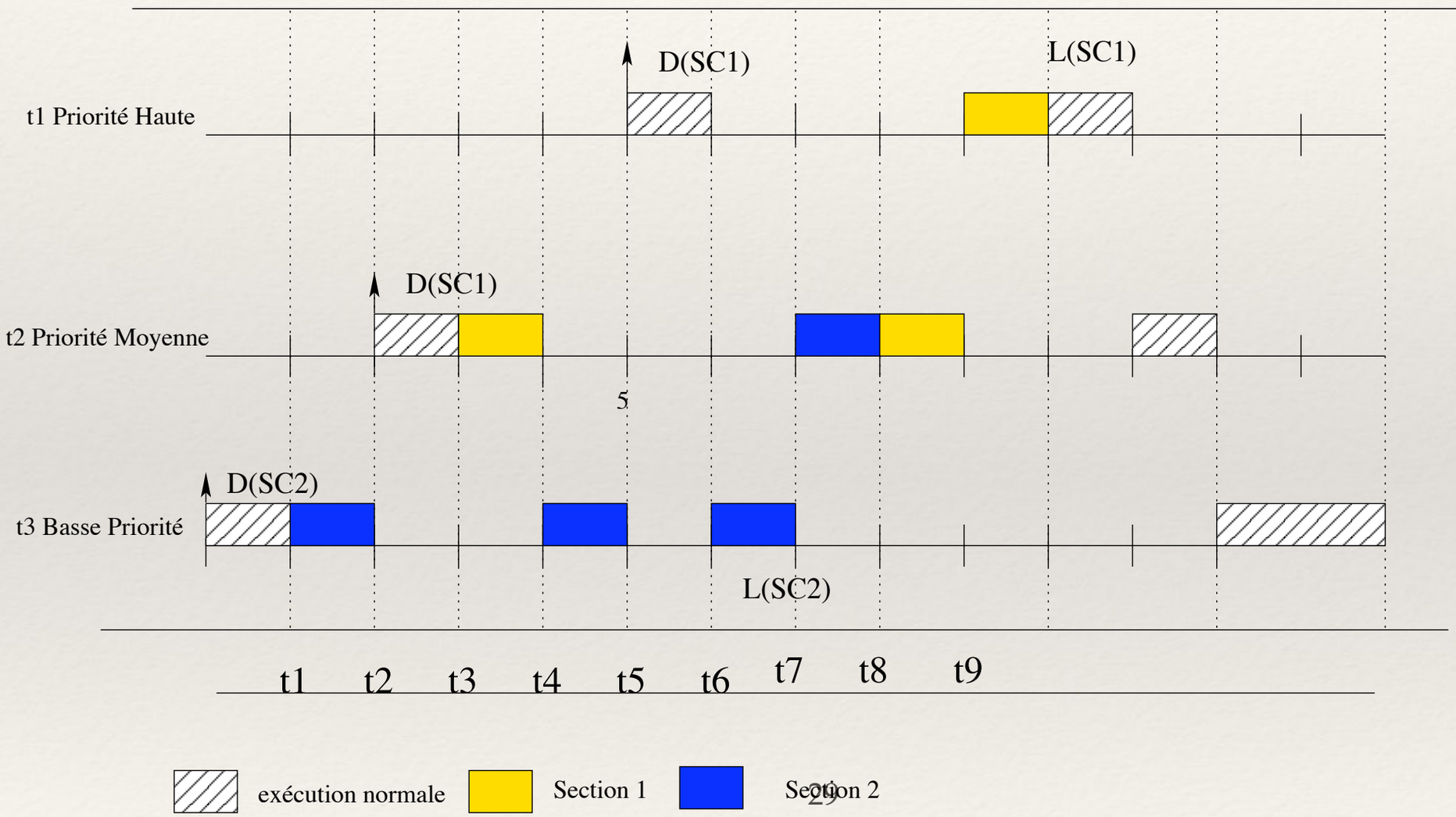
# Inversion de Priorité non bornée



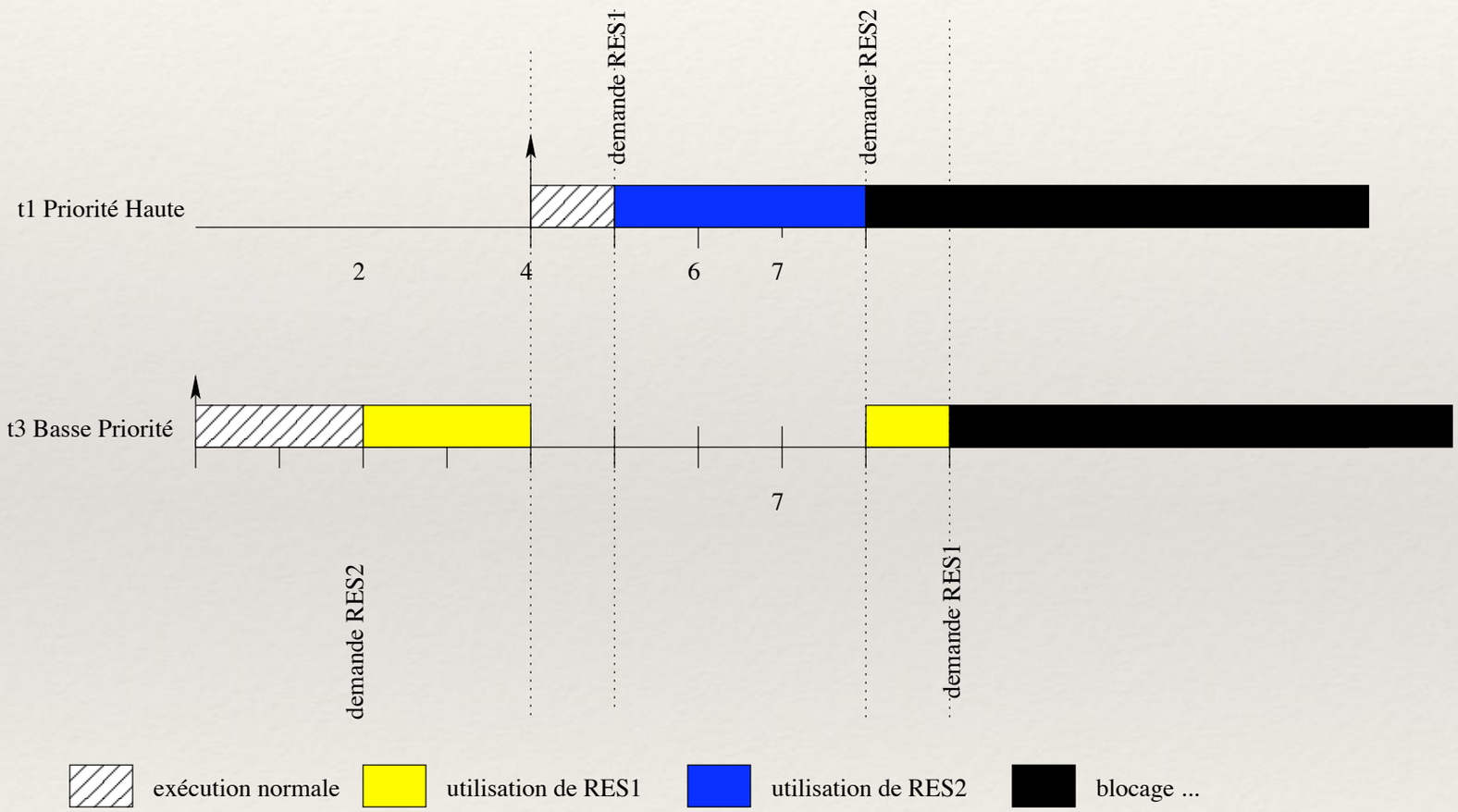
# Priority Inheritance Protocol (PIP)



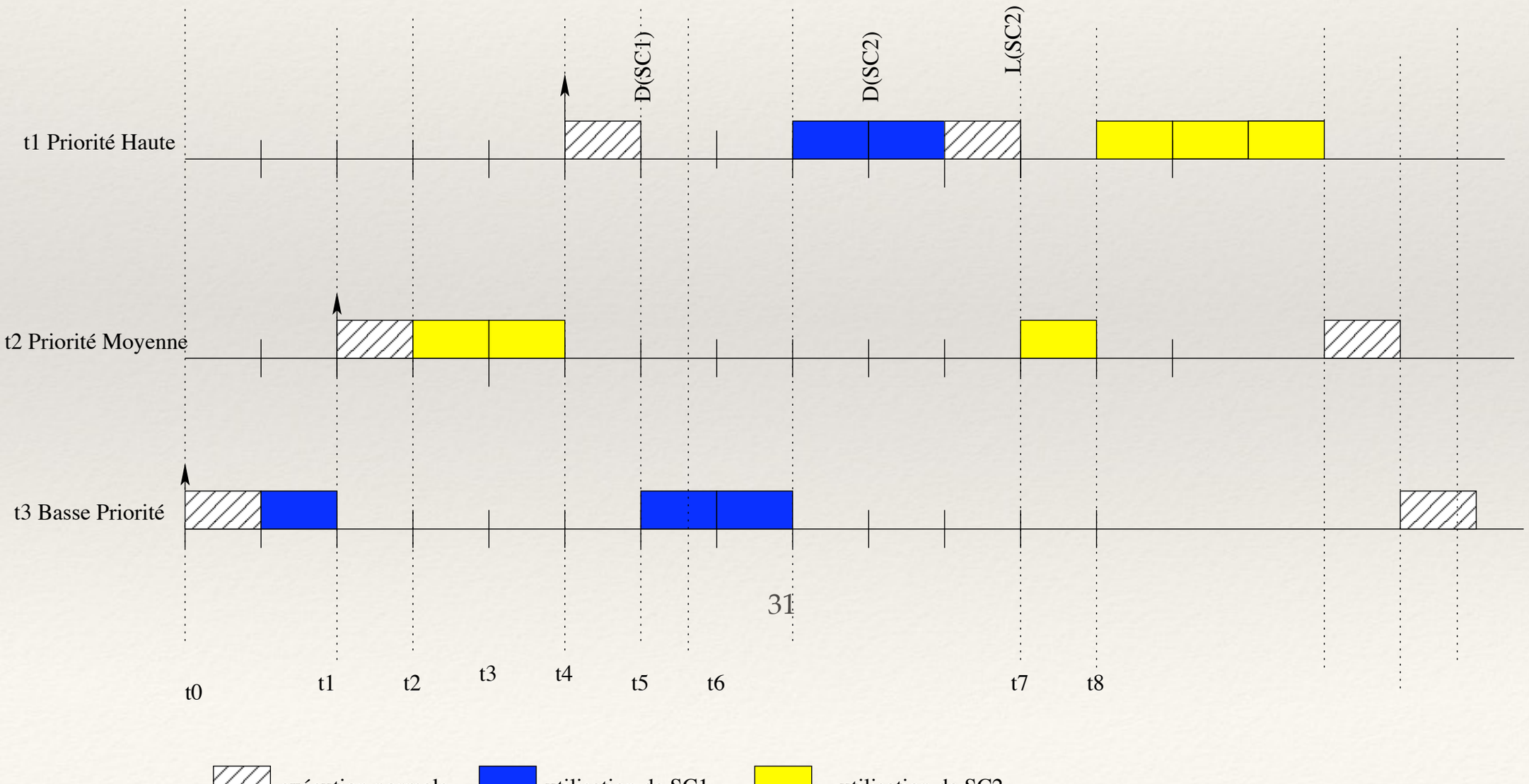
# Ressources Imbriquées - Héritage Transitif



# PIP - Ressources Imbriquées (Interblocage)



# PIP - chaines de blocage



# Faisabilité - Interférences

- *Le Facteur de blocage (Bi).*
  - *Avec PIP - Bi est égal à la somme des durées d'exécution des ressources partagées avec des tâches de priorité inférieure (2)*
  - *Avec PCE - Bi est égal au MAX des durées d'exécution des ressources partagées avec des tâches de priorité inférieure (1)*

$$B_{j,k} = \max_{m}(SC_{j,k}) \qquad (1)$$

$$B_j = \sum_{k \in LP(j)} \max_{m}(SC_{j,k}) \qquad (2)$$

# Conclusion

- *Reste à comprendre l'effet:*
  - *des dépendances des tâches sur le comportement de l'application (Synchronisations / Graphes)*
  - *du parallélisme (systèmes multiprocesseurs)*
  - *du traitement réparti (communication des tâches en réseau)*
  - *la gestion de l'énergie (ordonnancement sensible à l'énergie)*
- *et plus si affinité avec le sujet*