

Informatique embarquée

Systemes d'exploitation temps réel (RTOS)

Philippe.Plasson@obspm.fr

Systèmes d'exploitation temps réel (RTOS)

1. Définition
2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
4. Temps d'exécution, temps de réponse, ordonnançabilité.
5. Aperçu des produits disponibles sur le marché

Systèmes d'exploitation temps réel (RTOS)

1. Définition

2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
4. Temps d'exécution, temps de réponse, ordonnançabilité.
5. Aperçu des produits disponibles sur le marché

RTOS

Définition

- Un OS temps réel (Real-Time OS = RTOS), ou encore noyau temps réel, apporte un certain nombre de services facilitant la conception et la mise au point des applications embarquées temps réel.
- En particulier, un OS temps réel apporte la notion de programmation multi-tâches et de pseudo-parallélisme.
- La programmation multi-tâches permet de concevoir une application sous la forme d'un ensemble de tâches indépendantes, c'est-à-dire ayant leur propre fil d'exécution
 - Chaque tâche a sa propre pile et son propre contexte d'exécution
 - Synonyme de tâche = thread (fil) ➔ on parle aussi de multi-threading

RTOS

Définition

- Un RTOS propose généralement différentes possibilités d'ordonnancement :
 - Time-slicing ou non
 - Ordonnancement selon les priorités
 - Ordonnancement préemptif ou non
 - Algorithme d'ordonnancement RMS pour les tâches périodiques
 - ...

RTOS

Définition

- Les tâches interagissent entre elles via
 - l'échange de messages asynchrones (c'est-à-dire non bloquants) transitant par des queues de messages
 - des mécanismes de synchronisation :
 - sémaphores,
 - mutex,
 - événements
- Le chef d'orchestre du RTOS s'appelle l'ordonnanceur.
 - L'ordonnanceur, en fonction d'une politique d'ordonnancement donnée, a pour rôle de gérer la machine à états de chaque tâche.
 - Lors de chaque appel à une fonction du RTOS (envoi de message, réception de message, obtention d'une sémaphore, libération d'une sémaphore, etc.), l'ordonnanceur est appelé et décide quelle tâche doit s'exécuter.
 - Si une préemption doit avoir lieu, c'est l'ordonnanceur qui va déclencher les mécanismes de sauvegarde / restauration des contextes d'exécution.

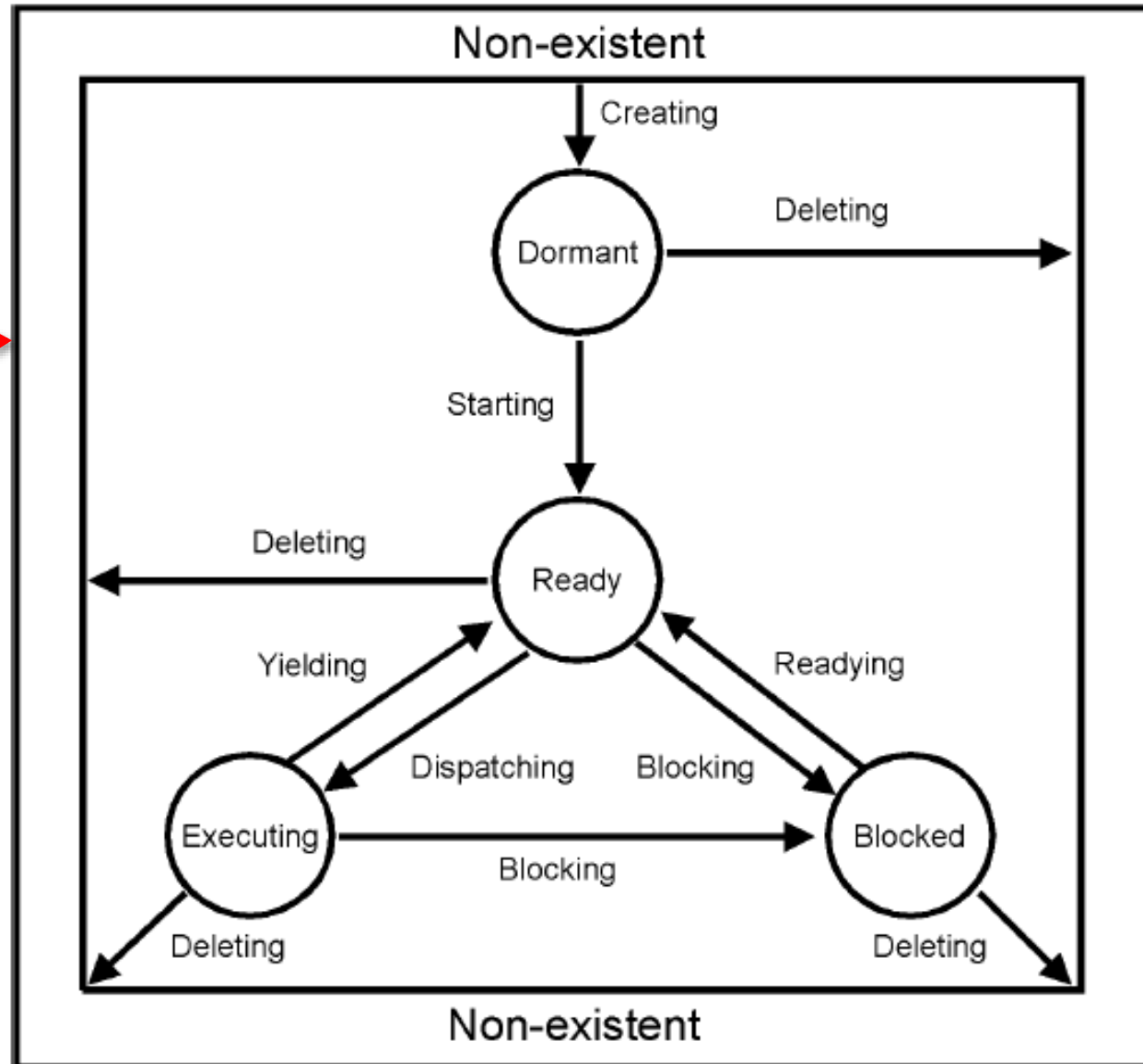
Systèmes d'exploitation temps réel (RTOS)

1. Définition
2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
4. Temps d'exécution, temps de réponse, ordonnançabilité.
5. Aperçu des produits disponibles sur le marché

Services et composants d'un RTOS

Les tâches - Machine à états d'une tâche

- C'est le rôle de l'ordonnanceur, en fonction d'une politique d'ordonnancement donnée, de gérer la machine à états de chaque tâche.
- A un instant donné (dans un système mono-core), une seule tâche peut être dans l'état Executing



Services et composants d'un RTOS

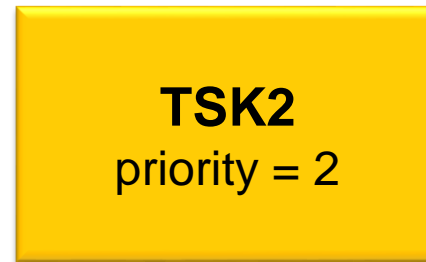
Les tâches - Initialisation

- Un RTOS offre des services pour initialiser, configurer et démarrer les tâches.
- Avec RTEMS, ce sont les fonctions `rtems_task_create()` et `rtems_task_start()` qui permettent de faire cela.

Services et composants d'un RTOS

Les tâches - Initialisation

- Soit 2 tâches TSK1 et TSK2
 - Les 2 tâches affichent leur compteur d'exécution en boucle
 - Les 2 tâches sont complètement indépendantes
 - La tâche TASK1 est plus prioritaire que la tâche TASK2
 - On choisit un mode d'ordonnancement préemptif selon les priorités



Services et composants d'un RTOS

Les tâches - Initialisation

```
#define TASK1_PRIORITY 1
#define TASK2_PRIORITY 2
#define TASK_STACK_SIZE 10240

rtems_id  task_id_1;
rtems_id  task_id_2;

rtems_task Init(rtems_task_argument argument)
{
    rtems_status_code status;

    status = rtems_task_create(rtems_build_name('T','S','K', '1'),
                               TASK1_PRIORITY, TASK_STACK_SIZE,
                               RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_INTERRUPT_LEVEL(0),
                               RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_1);

    status = rtems_task_create(rtems_build_name('T','S','K', '2'),
                               TASK2_PRIORITY, TASK_STACK_SIZE,
                               RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_INTERRUPT_LEVEL(0),
                               RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_2);

    status = rtems_task_start(task_id_1, task_1, 1);
    status = rtems_task_start(task_id_2, task_2, 1);

    status = rtems_task_delete(RTEMS_SELF);
}
```

- Le mode préemptif est activé.
- Le time-slicing est désactivé.
- Toutes les interruptions sont activées.

- Le point d'entrée de la tâche task_id_1 est la fonction task_1()

Services et composant d'un RTOS

Les tâches – Fonctions d'entrée

- Le code ci-dessous correspond aux fonctions d'entrée de la tâche 1 et de la tâche 2.
- L'exécution du programme montre que seule la tâche 1 est active. Elle est plus prioritaire que la tâche 2 et ne rend jamais la main.
- La fonction d'entrée d'une tâche a généralement la structure d'une boucle infinie qui attend des événements et les traite.

```
rtems_task task_1(rtems_task_argument unused) {  
    while (1) {  
        task_counter[T1]++;  
        print_task_counter();  
    }  
}
```

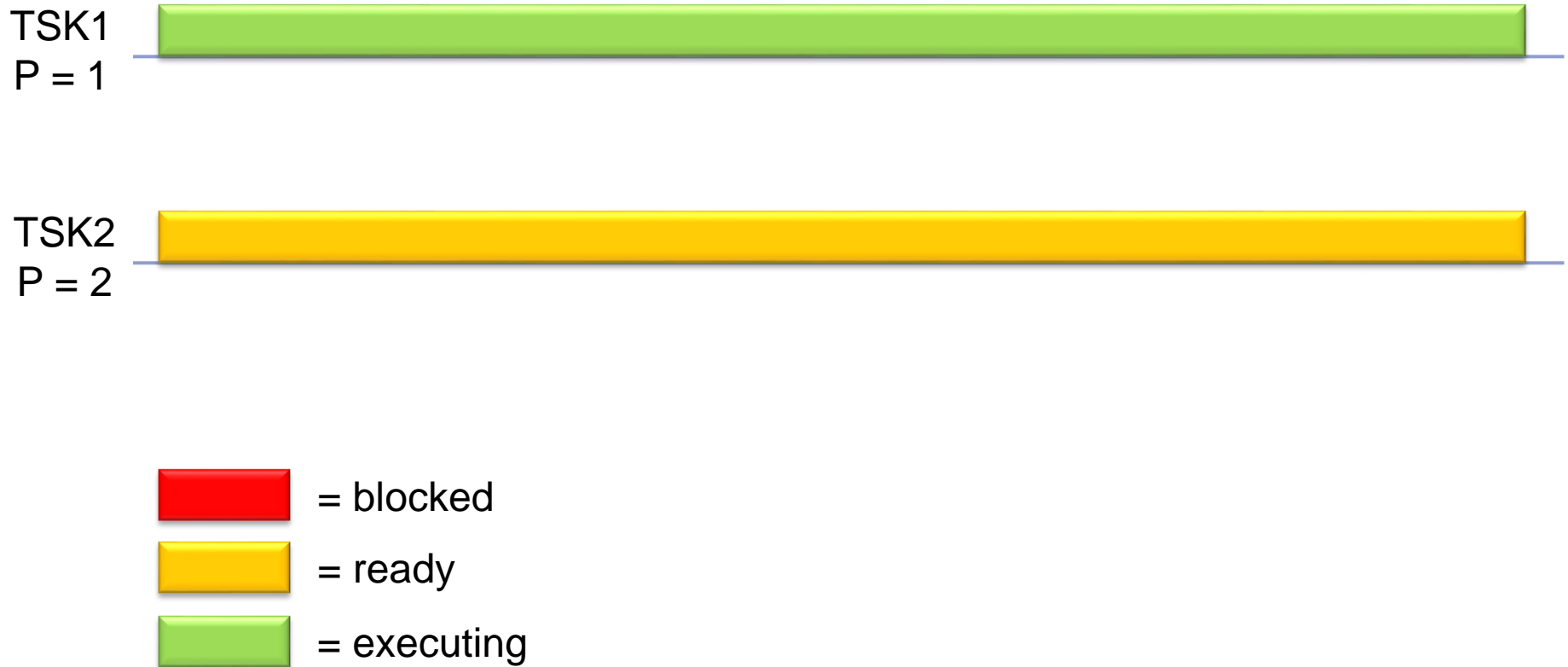
```
rtems_task task_2(rtems_task_argument unused) {  
    while (1) {  
        task_counter[T2]++;  
        print_task_counter();  
    }  
}
```

0.000000	T1 = 1	T2 = 0
0.000000	T1 = 2	T2 = 0
0.000000	T1 = 3	T2 = 0
0.000000	T1 = 4	T2 = 0
0.000000	T1 = 5	T2 = 0
0.000000	T1 = 6	T2 = 0
0.010000	T1 = 7	T2 = 0
...		
0.220000	T1 = 155	T2 = 0
0.220000	T1 = 156	T2 = 0
0.220000	T1 = 157	T2 = 0
0.220000	T1 = 158	T2 = 0
0.230000	T1 = 159	T2 = 0

Services et composant d'un RTOS

Les tâches

- Le diagramme ci-dessous montre le chronogramme des tâches TSK1 et TSK2



Services et composants d'un RTOS

Gestion du temps

- Les RTOS offrent des services permettant de gérer le temps.
- Un concept important est le concept de « tick système » : c'est l'unité de temps du RTOS dans laquelle sont exprimés tous les délais et timeouts manipulés par les services du RTOS.
- La valeur du tick est généralement configurable en nombre de μ s.
 - Dans l'exemple ci-dessous, on configure le tick de RTEMS pour valoir 10000 μ s soit 10 ms.

```
#define CONFIGURE_MICROSECONDS_PER_TICK 10000
```

Services et composants d'un RTOS

Gestion du temps

- Les RTOS mette à disposition des fonctions permettant de mesurer le temps.
- L'exemple ci-dessous fait appel à la fonction RTEMS `rtems_clock_get()` qui retourne le nombre de ticks système écoulés depuis le démarrage de l'application :

```
void print_task_counter() {
    uint32_t i;
    uint32_t time;

    rtems_clock_get( RTEMS_CLOCK_GET_TICKS_SINCE_BOOT, &time );

    printf("%f\t", (float)(time) / 100.f );

    for (i=0; i<TASKS_TO_PRINT; i++) {
        printf("T%d = %d\t\t", i+1, task_counter[i]);
    }
    printf("\n");
}
```

Services et composants d'un RTOS

Les tâches – Suspension avec délai

- On fait évoluer le code des tâches en ajoutant un appel au RTOS permettant de suspendre les tâches pendant un certain nombre de ticks système (fonction `rtems_task_wake_after()`).
- Un tick système vaut ici 10 ms : i.e. le timer matériel associé au gestionnaire du temps du RTOS est réglé pour délivrer une interruption toutes les 10 ms.
- La tâche 1 se réveille toutes les 100 ms.
- La tâche 2 se réveille toutes les 50 ms.
- Quand la tâche 1 s'endort (est suspendue), la tâche 2, moins prioritaire peut prendre la main.

```
rtems_task task_1(rtems_task_argument unused) {  
    while (1) {  
        task_counter[T1]++;  
        print_task_counter();  
        rtems_task_wake_after(10); // 10 ticks = 100 ms  
    }  
}
```

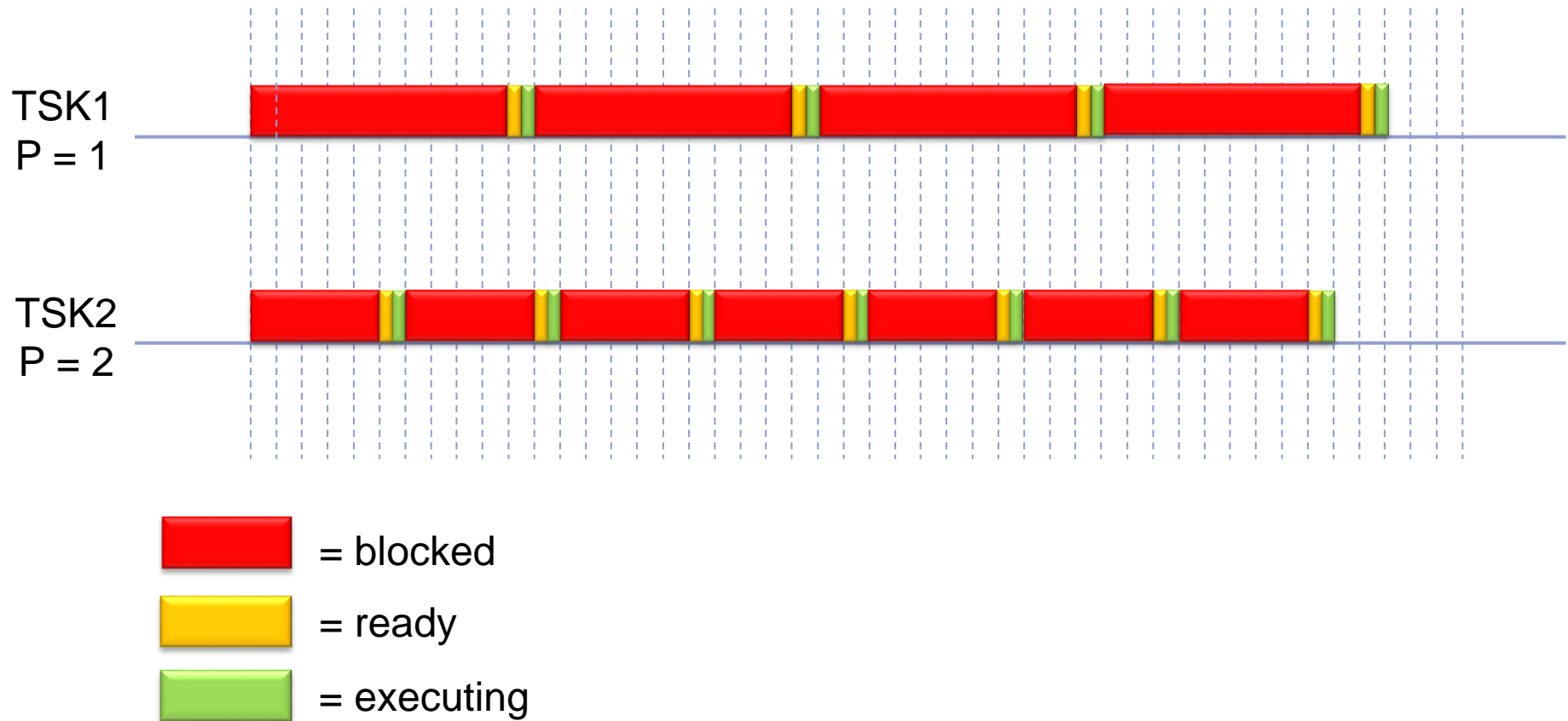
```
rtems_task task_2(rtems_task_argument unused) {  
    while (1) {  
        task_counter[T2]++;  
        print_task_counter();  
        rtems_task_wake_after(5); // 5 ticks = 50 ms  
    }  
}
```

0.000000	T1 = 1	T2 = 0
0.000000	T1 = 1	T2 = 1
0.050000	T1 = 1	T2 = 2
0.100000	T1 = 2	T2 = 2
0.100000	T1 = 2	T2 = 3
0.150000	T1 = 2	T2 = 4
0.200000	T1 = 3	T2 = 4
0.200000	T1 = 3	T2 = 5
0.250000	T1 = 3	T2 = 6
0.300000	T1 = 4	T2 = 6
0.300000	T1 = 4	T2 = 7
0.350000	T1 = 4	T2 = 8
0.400000	T1 = 5	T2 = 8
0.400000	T1 = 5	T2 = 9
0.450000	T1 = 5	T2 = 10
0.500000	T1 = 6	T2 = 10
0.500000	T1 = 6	T2 = 11
0.550000	T1 = 6	T2 = 12

Services et composant d'un RTOS

Les tâches

- Le diagramme ci-dessous montre le chronogramme des tâches TSK1 et TSK2



Services et composants d'un RTOS

Echange de messages

- On souhaite désormais faire évoluer l'application de telle sorte que les 2 tâches communiquent entre elles.
- Tous les RTOS offre un service d'échange de messages basé sur l'utilisation de composants « Queues de messages »
 - Chaque queue de messages peut contenir 1 ou plusieurs messages.
 - Service « send » permet de poster un message.
 - Service « receive » permet de lire un message.
- Les queues de messages fonctionnent généralement comme des FIFO.
- Les RTOS proposent cependant des fonctions permettant aussi de poster des messages selon une approche LIFO.

Services et composants d'un RTOS

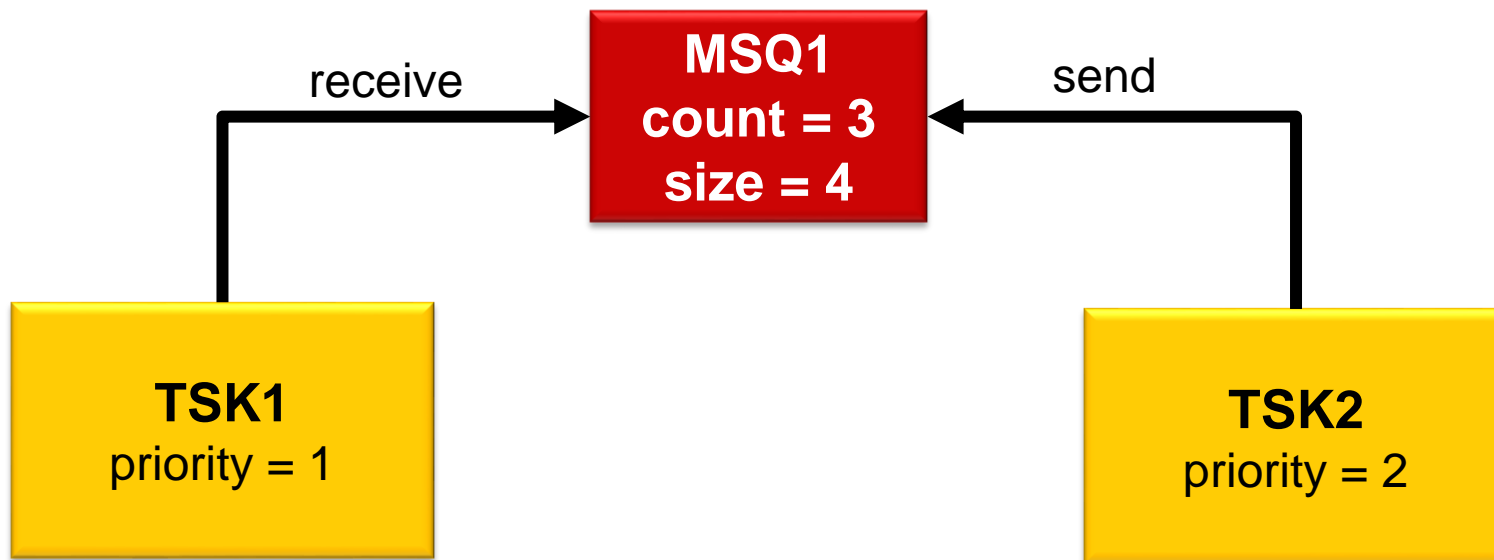
Echange de messages

- Trois modes de lecture des messages sont possibles :
 - Attente bloquante sans timeout : si une tâche appelle le service « receive » sur une queue de message vide, cela a pour effet de bloquer la tâche (état suspendu) jusqu'à ce qu'un message soit présent dans la queue de message.
 - Attente bloquante avec timeout : au bout d'un délai (timeout) exprimé en ticks système, la tâche va reprendre son exécution même si aucun message n'a été reçu.
 - Pas d'attente : si aucun message n'est présent dans la queue de messages, la tâche va continuer son exécution sans attendre.

Services et composants d'un RTOS

Echange de messages

- Les 2 tâches TSK1 et TSK2 communiquent entre elles via une queue de message pouvant contenir 3 messages, chaque message ayant une taille de 4 octets.
- La tâche TSK1 lit les messages (lecture bloquante sans timeout) et la tâche TSK2 produit les messages.



Services et composants d'un RTOS

Echange de messages

```
#define TASK1_PRIORITY 1
#define TASK2_PRIORITY 2
#define TASK_STACK_SIZE 10240
#define MESSAGE_QUEUE_COUNT 3
#define MESSAGE_QUEUE_SIZE 4
```

```
rtems_id task_id_1;
rtems_id task_id_2;
rtems_id message_queue_id_1;
```

```
rtems_task Init(rtems_task_argument argument) {
    rtems_status_code status;

    status = rtems_task_create(rtems_build_name('T','S','K', '1'),
        TASK1_PRIORITY, TASK_STACK_SIZE,
        RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_ASR | RTEMS_INTERRUPT_LEVEL(0),
        RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_1);

    status = rtems_task_create(rtems_build_name('T','S','K', '2'),
        TASK2_PRIORITY, TASK_STACK_SIZE,
        RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_ASR | RTEMS_INTERRUPT_LEVEL(0),
        RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_2);

    status = rtems_message_queue_create(rtems_build_name('M','S','Q', '1'),
        MESSAGE_QUEUE_COUNT, MESSAGE_QUEUE_SIZE,
        RTEMS_LOCAL | RTEMS_PRIORITY, &message_queue_id_1);

    ...
}
```

Avec RTEMS, la création d'une queue de message se fait à l'aide de la fonction `rtems_message_queue_create()`

Services et composant d'un RTOS

Echange de messages

- La tâche TSK1 se met en attente de messages sur la queue de message MSQ1.
- Dès qu'elle reçoit un message, elle affiche son contenu.

```
#define BUFFER_SIZE 1

rtems_task task_1(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    size_t size;
    rtems_status_code status;

    while (1) {
        status = rtems_message_queue_receive(message_queue_id_1,
            buffer, &size, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

        if (status == RTEMS_SUCCESSFUL) {
            task_counter[T1]++;
            print_task_counter();
            printf("message from MSQ1 = %d\n", buffer[0]);
        }
    }
}
```

Services et composant d'un RTOS

Echange de messages

- La tâche TSK2 poste des messages (valeur de son compteur d'exécution) dans la queue de messages MSQ1.
- Si la queue de messages est pleine, elle affiche un message d'alerte.

```
#define BUFFER_SIZE 1

rtems_task task2(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    rtems_status_code status;

    while (1) {
        buffer[0] = task_counter[T2];
        status = rtems_message_queue_send(message_queue_id_1, buffer, sizeof(uint32_t));

        if (status == RTEMS_TOO_MANY) {
            printf("MSQ1 is full\n");
        }
        task_counter[T2]++;
        print_task_counter();
    }
}
```

Services et composant d'un RTOS

Echange de messages

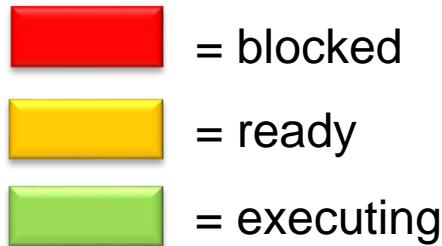
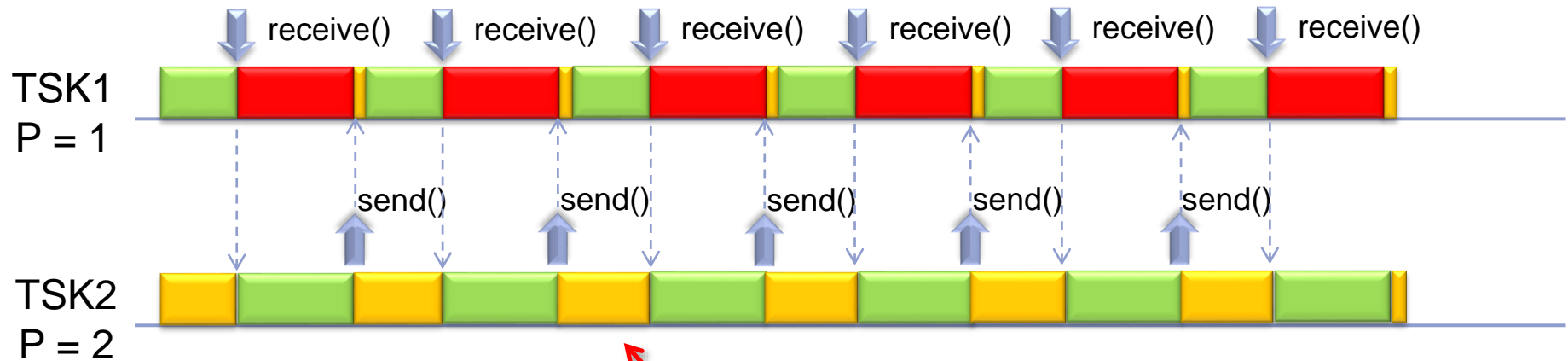
- Le résultat de l'exécution du programme est donné ci-dessous : les tâches TSK1 et TSK2 s'exécutent de façon alternative.

```
0.000000      T1 = 1      T2 = 0
MSQ1 = 0
0.000000      T1 = 1      T2 = 1
0.000000      T1 = 2      T2 = 1
MSQ1 = 1
0.000000      T1 = 2      T2 = 2
0.010000      T1 = 3      T2 = 2
MSQ1 = 2
0.010000      T1 = 3      T2 = 3
0.010000      T1 = 4      T2 = 3
MSQ1 = 3
0.010000      T1 = 4      T2 = 4
0.010000      T1 = 5      T2 = 4
MSQ1 = 4
0.020000      T1 = 5      T2 = 5
0.020000      T1 = 6      T2 = 5
MSQ1 = 5
```


Services et composant d'un RTOS

Ré-ordonnancement suite à un envoi de message

- Le diagramme ci-dessous montre le chronogramme des tâches TSK1 et TSK2




- Quand TSK1 appelle la fonction `receive()`, elle est suspendue par l'ordonnanceur. L'ordonnanceur passe alors TSK2 dans l'état « executing ».
- Quand TSK2 appelle la fonction `send()`, l'ordonnanceur passe TSK1 dans l'état « ready » puis dans l'état « executing »; TSK2 passe quant à elle dans l'état « ready »

Services et composant d'un RTOS

Echange de messages

- Que se passerait-il si l'on inversait les priorités de TSK1 et TSK2 ?
 - Priorité de TSK1 = 3
 - Priorité de TSK2 = 2

```
0.000000      T1 = 0      T2 = 1
0.000000      T1 = 0      T2 = 2
0.000000      T1 = 0      T2 = 3
MSQ1 is full
0.010000      T1 = 0      T2 = 4
MSQ1 is full
0.020000      T1 = 0      T2 = 5
MSQ1 is full
0.020000      T1 = 0      T2 = 6
MSQ1 is full
0.020000      T1 = 0      T2 = 7
MSQ1 is full
0.020000      T1 = 0      T2 = 8
MSQ1 is full
0.030000      T1 = 0      T2 = 9
MSQ1 is full
0.030000      T1 = 0      T2 = 10
```

- 
- TSK2 devenant plus prioritaire reste continuellement active.
 - Aucun appel bloquant ne la fait passer dans l'état « blocked »
→ TSK1 n'est jamais activée

Services et composant d'un RTOS

Synchronisation des tâches / évènements

- On souhaite faire évoluer l'application de telle sorte que, quand la queue de message MSQ1 est pleine, la tâche TSK2 productrice des messages suspende son exécution en attente d'un signal de la tâche TSK1 consommatrice des messages.
- La tâche TSK1, quand elle s'exécute, doit vider la queue de message MSQ2, et une fois la queue de message vide, envoyer un signal à la tâche TSK2 qui va pouvoir à nouveau remplir la queue de messages jusqu'à ce qu'elle devienne pleine.
- Pour atteindre ce but, on a besoin d'un service permettant de « synchroniser » les 2 tâches.

Services et composant d'un RTOS

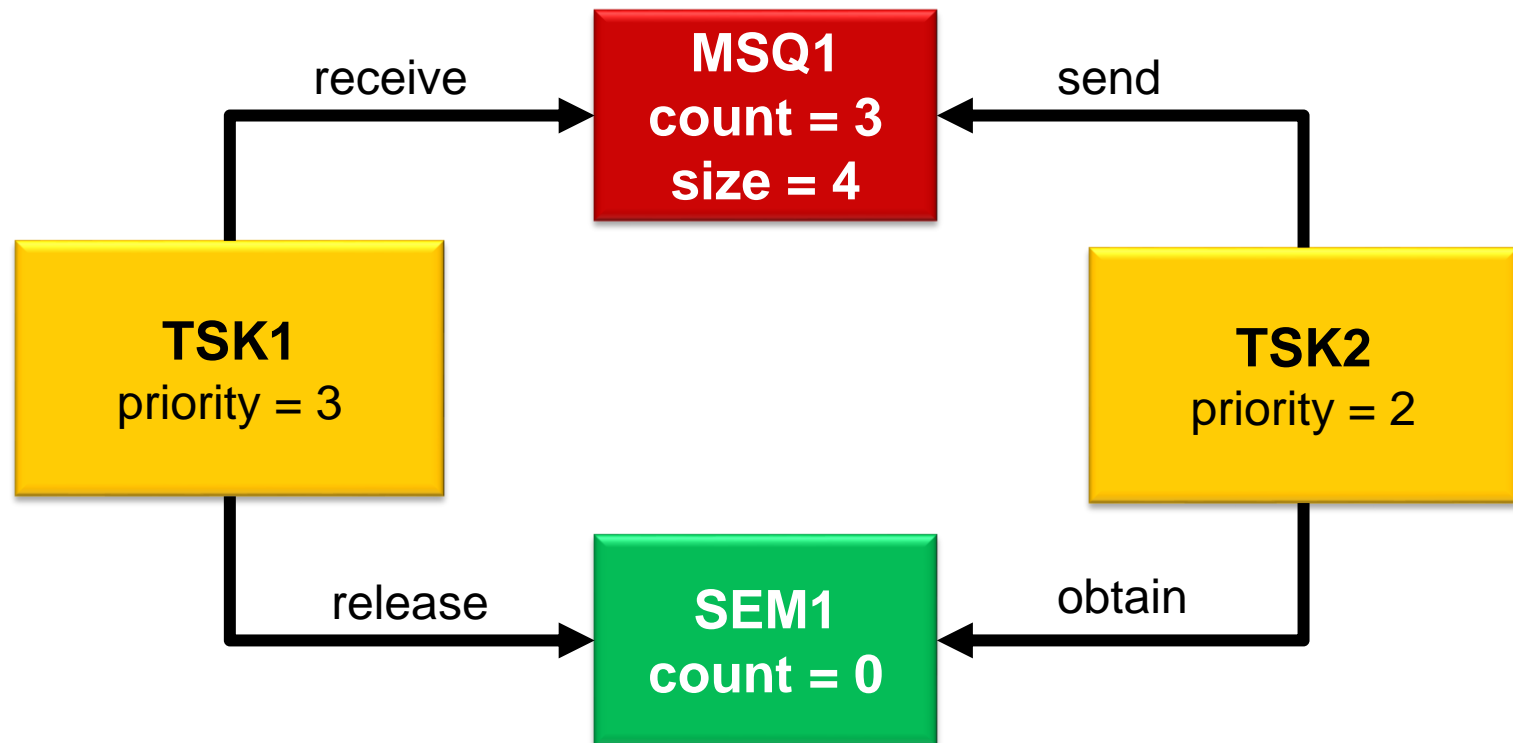
Synchronisation des tâches / événements

- Un RTOS offre généralement un ou plusieurs services de synchronisation
 - Sémaphores
 - Un compteur est associé à chaque sémaphore.
 - Service « put » / « release » incrémente la sémaphore.
 - Service « get » / « obtain » décrémente la sémaphore. Si une tâche appelle le service « get » / « obtain » sur une sémaphore dont le compteur vaut 0, cela a pour effet de bloquer la tâche (état suspendu).
 - 2 cas d'utilisation :
 1. exclusions mutuelles (protection des ressources partagées, sections critiques)
 2. notification d'événements
 - Groupes d'événements : utilisés pour gérer la notification d'événements multicast (1 émetteur, n récepteur).
 - Mutex : sémaphore binaire permettant de gérer les exclusions mutuelles.

Services et composant d'un RTOS

Synchronisation des tâches / évènements

- SEM1 est une sémaphore binaire dont le compteur est initialisé à 0 au démarrage.
- Quand TSK2 a terminé de remplir MSQ1, elle demande à « obtenir » le sémaphore → elle se bloque en attente que le sémaphore soit positionné à 1 par TSK1 via un appel « release ».



Services et composant d'un RTOS

Synchronisation des tâches / évènements

```
#define TASK1_PRIORITY    3
#define TASK2_PRIORITY    2
#define TASK_STACK_SIZE 10240
#define MESSAGE_QUEUE_COUNT    3
#define MESSAGE_QUEUE_SIZE 4
```

```
rtems_id task_id_1;
rtems_id task_id_2;
rtems_id message_queue_id_1;
rtems_id semaphore_id_1;
```

```
rtems_task Init(rtems_task_argument argument) {
    rtems_status_code status;
```

```
    status = rtems_task_create(rtems_build_name('T','S','K', '1'),
        TASK1_PRIORITY, TASK_STACK_SIZE,
        RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_ASR | RTEMS_INTERRUPT_LEVEL(0),
        RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_1);
```

```
    status = rtems_task_create(rtems_build_name('T','S','K', '2'),
        TASK2_PRIORITY, TASK_STACK_SIZE,
        RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_ASR | RTEMS_INTERRUPT_LEVEL(0),
        RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_2);
```

```
    status = rtems_message_queue_create(rtems_build_name('M','S','Q', '1'),
        MESSAGE_QUEUE_COUNT, MESSAGE_QUEUE_SIZE,
        RTEMS_LOCAL | RTEMS_PRIORITY, &message_queue_id_1);
```

```
    status = rtems_semaphore_create(rtems_build_name('S','E','M', '1'),
        0, RTEMS_PRIORITY | RTEMS_SIMPLE_BINARY_SEMAPHORE ,
        0, &semaphore_id_1);
```

...

Création d'une
sémaphore
binaire dont le
compteur est
initialisé à 0

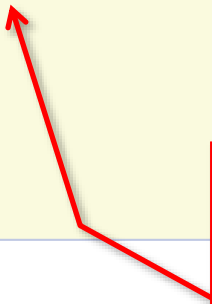
Services et composant d'un RTOS

Synchronisation des tâches / évènements

- Le code de task_2() est modifié de cette façon :

```
rtems_task task_2(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    rtems_status_code status;

    while (1) {
        buffer[0] = task_counter[T2];
        status = rtems_message_queue_send(message_queue_id_1, buffer, sizeof(uint32_t));
        if (status == RTEMS_TOO_MANY) {
            printf("message_queue_id_1 is full\n");
            rtems_semaphore_obtain(semaphore_id_1, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        }
        task_counter[1]++;
        print_task_counter();
    }
}
```



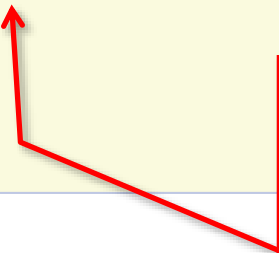
Quand TSK2 a terminé de remplir MSQ1, elle demande à « obtenir » le sémaphore → elle se bloque en attente que le sémaphore soit positionné à 1 par TSK1 via un appel « release ».

Services et composant d'un RTOS

Synchronisation des tâches / évènements

```
rtems_task task_1(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    size_t size;
    rtems_status_code status;
    uint32_t count;

    while (1) {
        status = rtems_message_queue_receive(message_queue_id_1, buffer, &size,
                                             RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        if (status == RTEMS_SUCCESSFUL) {
            task_counter[T1]++;
            print_task_counter();
            printf("message_queue_id_1 = %d\n", buffer[0]);
            rtems_message_queue_get_number_pending(message_queue_id_1, &count);
            printf("count = %d\n", count);
            if (count == 0) {
                status = rtems_semaphore_release(semaphore_id_1);
            }
        }
    }
}
```



Quand TSK1 a terminé de vider la queue de messages, alors elle positionne le sémaphore à 1 en appelant la fonction « release ». Ceci va avoir pour effet de débloquer TSK2 et donc de suspendre TSK1 qui est moins prioritaire.

Services et composant d'un RTOS

Synchronisation des tâches / évènements

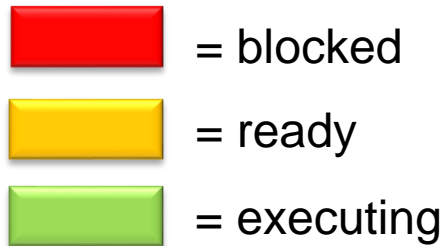
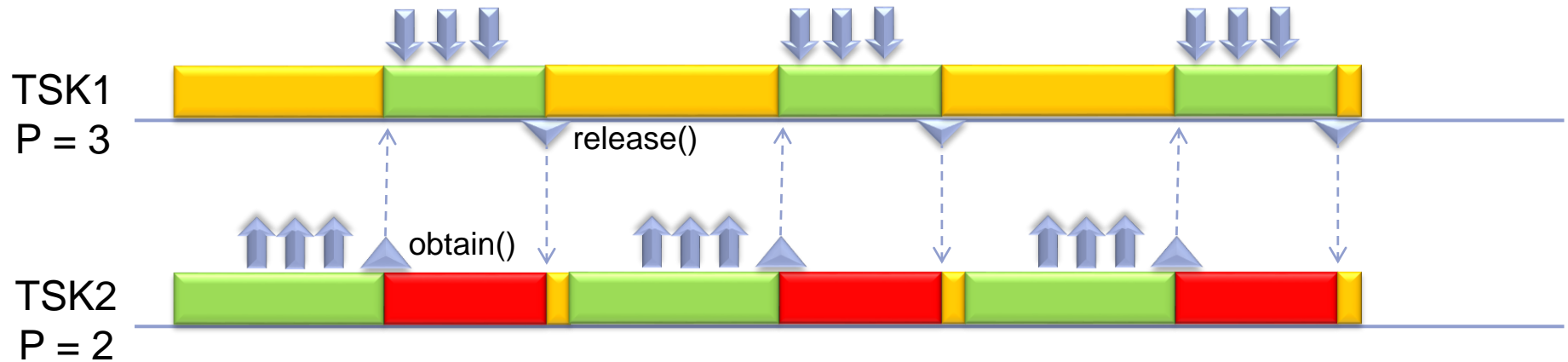
Résultat de l'exécution de l'application : quand TSK2 a terminé de remplir MSQ1, TSK1 commence à s'exécuter et à dépiler les messages.

```
0.000000      T1 = 0      T2 = 1
0.000000      T1 = 0      T2 = 2
0.000000      T1 = 0      T2 = 3
MSQ1 is full
-----
0.000000      T1 = 1      T2 = 3
MSQ1 = 0
count = 2
0.010000      T1 = 2      T2 = 3
MSQ1 = 1
count = 1
0.010000      T1 = 3      T2 = 3
MSQ1 = 2
count = 0
-----
0.010000      T1 = 3      T2 = 4
0.010000      T1 = 3      T2 = 5
0.010000      T1 = 3      T2 = 6
0.020000      T1 = 3      T2 = 7
MSQ1 is full
-----
status = 0
0.020000      T1 = 4      T2 = 7
MSQ1 = 4
count = 2
0.020000      T1 = 5      T2 = 7
MSQ1 = 5
count = 1
0.020000      T1 = 6      T2 = 7
MSQ1 = 6
count = 0
-----
0.030000      T1 = 6      T2 = 8
0.030000      T1 = 6      T2 = 9
...
```

Services et composant d'un RTOS

Synchronisation des tâches / événements

- Le diagramme ci-dessous montre le chronogramme des tâches TSK1 et TSK2



Services et composant d'un RTOS

Les timers logiciels

- Dans le domaine des applications temps réel et embarquées, il est souvent nécessaire de déclencher des traitements périodiques.
- Ceci peut être réalisé en utilisant des tâches qui vont s'endormir pendant x ticks systèmes.
- La plupart des RTOS intègre des composants « timer logiciel » permettant de faciliter la mise en œuvre de ces traitements périodiques.
- Les timers logiciels peuvent aussi être utilisés pour déclencher un traitement qui ne s'exécutera qu'une seule fois mais de façon différée, au bout de x ticks système : on parle de timer « one-shot ».

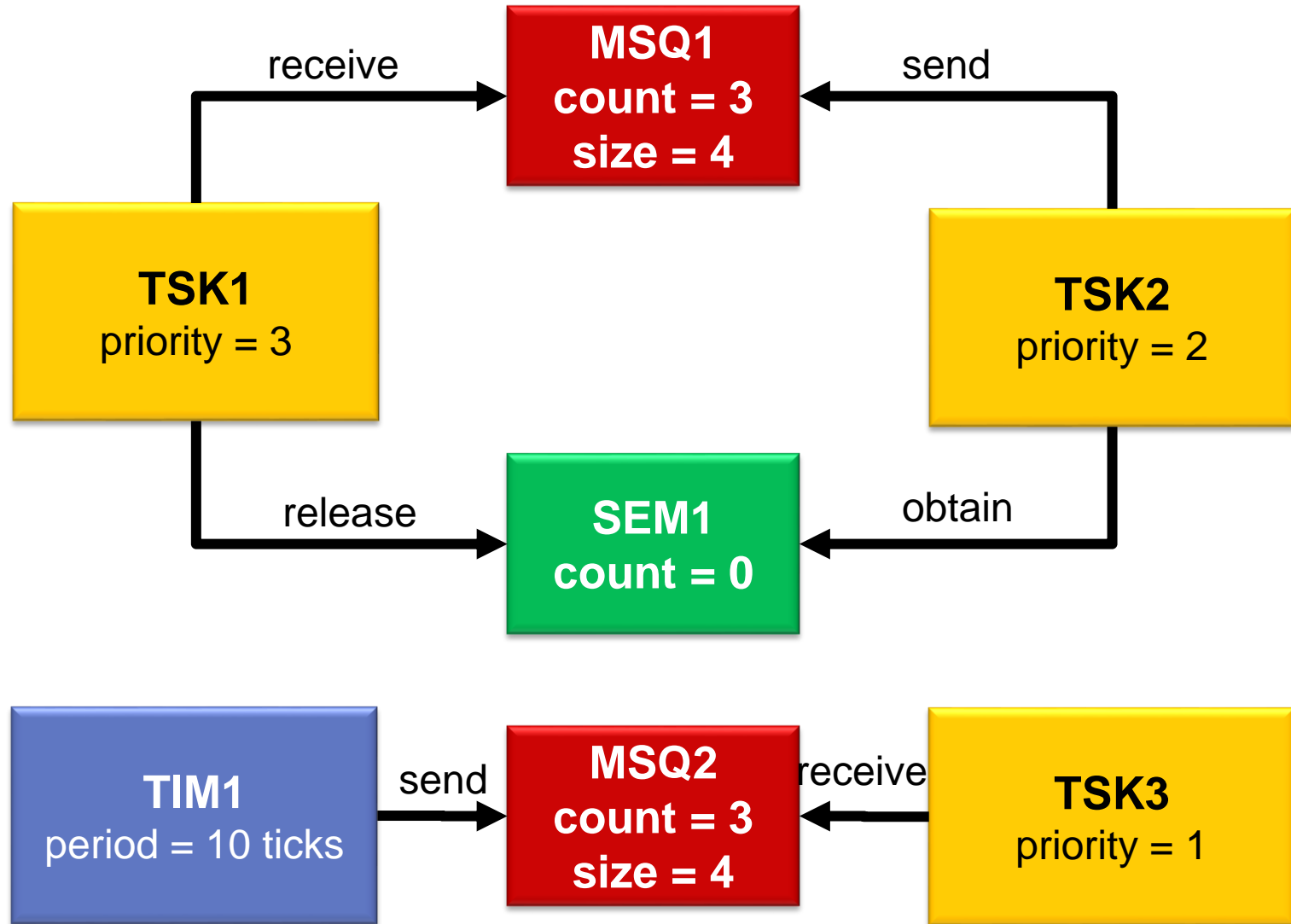
Services et composant d'un RTOS

Les timers logiciels

- On souhaite complexifier notre application en ajoutant une tâche TSK3 qui sera déclenchée de façon périodique, toutes les 100 ms (10 ticks), par une timer logiciel TIM1.
- On utilise par ailleurs une nouvelle queue de message MSQ2 pour assurer la communication entre TIM1 et TSK3.

Services et composant d'un RTOS

Les timers logiciels



Services et composant d'un RTOS

Les timers logiciels

On définit un timer logiciel TIM1 en lui associant une fonction de traitement `timer_1_entry()` qui sera déclenchée au bout de 10 ticks (100 ms)

```
#define TASK3_PRIORITY 1
...

rtems_id task_id_3;
rtems_id message_queue_id_2;
rtems_id timer_id_1;
...

rtems_task Init(rtems_task_argument argument) {
...
    status = rtems_task_create(rtems_build_name('T','S','K', '3'),
        TASK3_PRIORITY, TASK_STACK_SIZE,
        RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_ASR | RTEMS_INTERRUPT_LEVEL(0),
        RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_3);
...

    status = rtems_message_queue_create(rtems_build_name('M','S','Q', '2'),
        MESSAGE_QUEUE_COUNT, MESSAGE_QUEUE_SIZE,
        RTEMS_LOCAL | RTEMS_PRIORITY, &message_queue_id_2);
...

    status = rtems_timer_create(rtems_build_name('T','I','M', '1'), &timer_id_1);
    status = rtems_timer_fire_after(timer_id_1, 10, timer_1_entry, 0); // 10 ticks = 100 ms
    status = rtems_task_start(task_id_3, task_3, 1);
...
}
```

Services et composant d'un RTOS

Les timers logiciels

- Ci-dessous le code de la fonction `timer_1_entry()` associée au timer logiciel TIM1 :

```
#define BUFFER_SIZE 1

void timer_1_entry(rtems_id timer_id, void* timer_input) {
    static uint32_t data = 0;
    rtems_status_code status;

    data++;

    status = rtems_message_queue_send(message_queue_id_2, &data, sizeof(uint32_t));

    status = rtems_timer_fire_after(timer_id_1, 10, timer_1_entry, 0); // 10 ticks = 100 ms
}
```

La fonction RTEMS `rtems_timer_fire_after()` permet de réarmer le timer → ainsi, le timer TIM1 a bien un déclenchement périodique

Services et composant d'un RTOS

Les timers logiciels

- Ci-dessous le code de la tâche TSK3 :

```
#define BUFFER_SIZE 1

rtems_task task_3(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    size_t size;
    rtems_status_code status;

    while (1) {
        status = rtems_message_queue_receive(message_queue_id_2,
                                             buffer, &size, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        task_counter[2]++;
        print_task_counter();
    }
}
```


Services et composant d'un RTOS

Les timers logiciels

La tâche TSK3 se déclenche toutes les 100 ms.

```
0.090000      T1 = 15          T2 = 20          T3 = 0
0.090000      T1 = 15          T2 = 21          T3 = 0
0.090000      T1 = 15          T2 = 22          T3 = 0
0.090000      T1 = 15          T2 = 23          T3 = 0
MSQ1 is full
-----
0.100000      T1 = 15          T2 = 23          T3 = 1
0.100000      T1 = 16          T2 = 23          T3 = 1
MSQ1 = 20
count = 2
0.100000      T1 = 17          T2 = 23          T3 = 1
MSQ1 = 21
count = 1
0.100000      T1 = 18          T2 = 23          T3 = 1
MSQ1 = 22
...
0.190000      T1 = 31          T2 = 43          T3 = 1
MSQ1 = 40
count = 2
0.190000      T1 = 32          T2 = 43          T3 = 1
MSQ1 = 41
count = 1
-----
0.200000      T1 = 33          T2 = 43          T3 = 2
0.190000      T1 = 33          T2 = 43          T3 = 2
MSQ1 = 42
count = 0
0.200000      T1 = 33          T2 = 44          T3 = 2
0.200000      T1 = 33          T2 = 45          T3 = 2
0.200000      T1 = 33          T2 = 46          T3 = 2
```

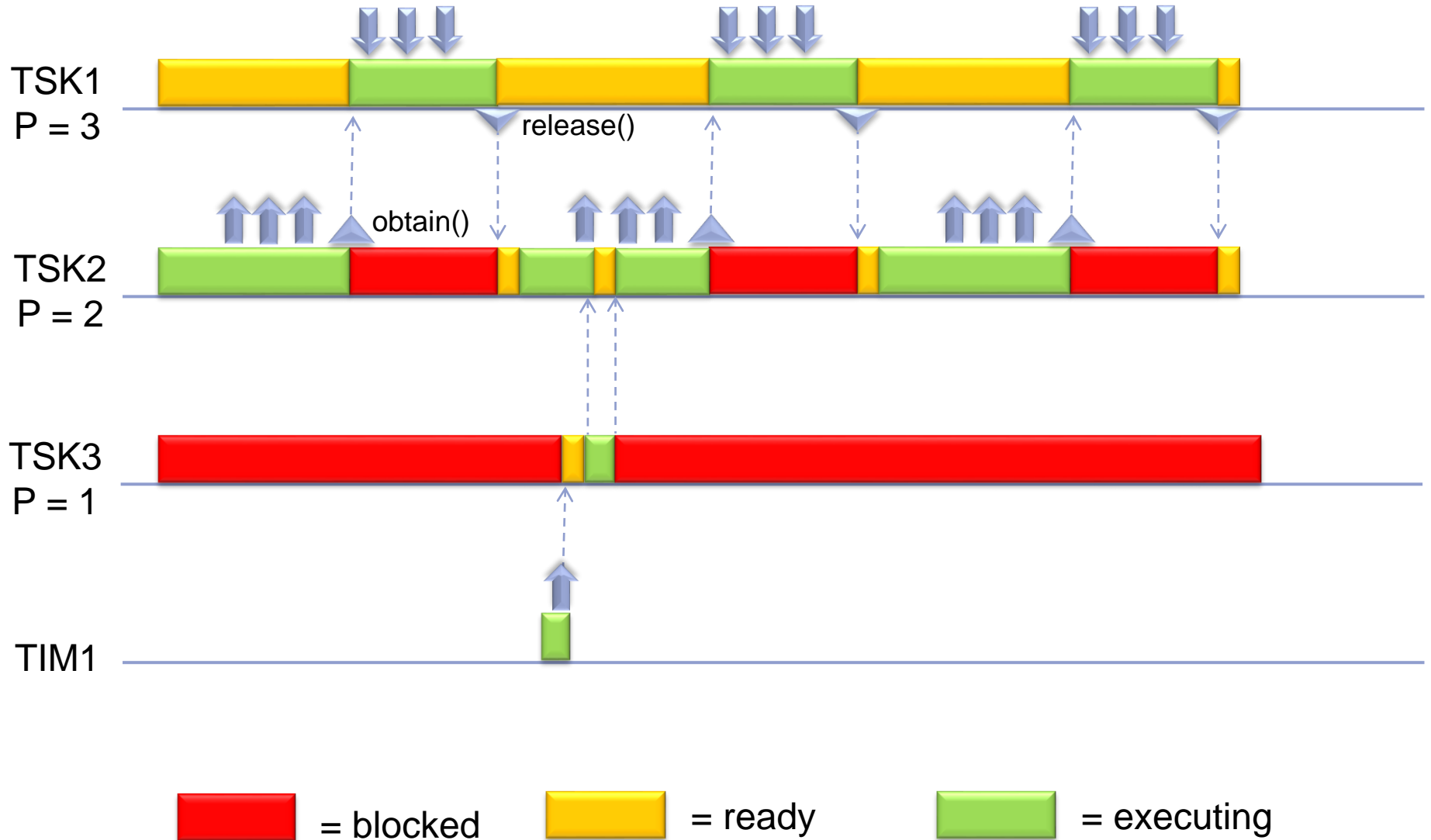
Services et composant d'un RTOS

Les timers logiciels

- Les fonctions appelées par les timers logiciels (comme `timer_1_entry()` dans notre exemple) s'exécutent dans le thread système du RTOS qui a la priorité la plus forte.
- Il est conseillé de ne pas implémenter directement les traitements dans ces fonctions associées aux timers.
- Il faut poster un message (ou déclencher un événement) qui permettra d'activer une tâche qui effectuera le traitement dans son propre thread.

Services et composant d'un RTOS

Les timers logiciels



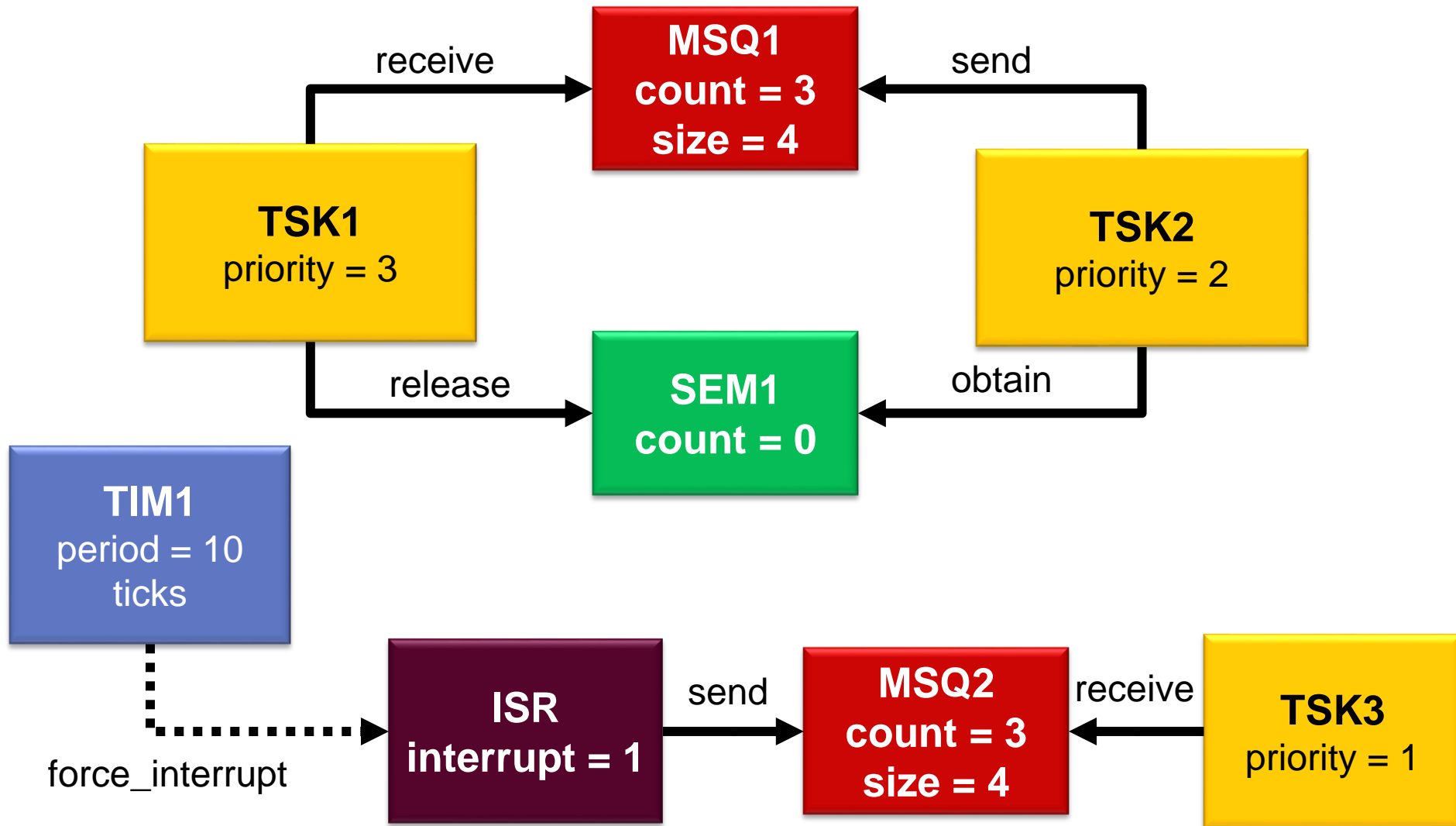
Services et composants d'un RTOS

Gestionnaire d'interruptions

- Les RTOS offrent des mécanismes permettant d'installer des gestionnaires d'interruption, c'est-à-dire les routines (ISR) permettant de répondre aux interruptions.
- Comme pour les routines de traitement des timers logiciels, il est fortement conseillé de ne pas implémenter directement les traitements dans les gestionnaires d'interruptions.
 - L'objectif est de pouvoir réactiver au plus vite l'interruption afin de ne pas retarder le traitement des nouvelles interruptions.
 - Dans la routine de traitement de l'interruption, il faut donc poster un message (ou déclencher un événement) qui permettra d'activer une tâche qui effectuera le traitement dans son propre thread avec sa propre priorité.

Services et composants d'un RTOS

Gestionnaire d'interruptions



Services et composants d'un RTOS

Gestionnaire d'interruptions

```
...
rtems_task Init(rtems_task_argument argument) {
...
    set_vector(isr_handler, (rtems_vector_number) (0x10+1), 1);
...
}
```

Installation d'une routine
d'interruption `isr_handler()`
associée à l'interruption
matérielle n°1

```
#define BUFFER_SIZE 1

rtems_isr isr_handler(rtems_vector_number vector) {
    static uint32_t data = 0;
    rtems_status_code status;

    data++;
    status = rtems_message_queue_send(message_queue_id_2, &data, sizeof(uint32_t));
}
```

Routine de traitement de
l'interruption

```
void timer_1_entry(rtems_id timer_id, void* timer_input) {
    rtems_status_code status;

    force_interrupt(1);

    status = rtems_timer_fire_after(timer_id_1, 10, timer_1_entry, 0); // 10 ticks = 100 ms
}
```

Déclenchement de
l'interruption dans la routine
de traitement du timer

Services et composant d'un RTOS

Gestionnaire d'interruptions

La tâche TSK3 se déclenche toutes les 100 ms.

```

0.090000      T1 = 15          T2 = 20          T3 = 0
0.090000      T1 = 15          T2 = 21          T3 = 0
0.090000      T1 = 15          T2 = 22          T3 = 0
0.090000      T1 = 15          T2 = 23          T3 = 0
MSQ1 is full
-----
0.100000      T1 = 15          T2 = 23          T3 = 1
0.100000      T1 = 16          T2 = 23          T3 = 1
MSQ1 = 20
count = 2
0.100000      T1 = 17          T2 = 23          T3 = 1
MSQ1 = 21
count = 1
0.100000      T1 = 18          T2 = 23          T3 = 1
MSQ1 = 22
...
0.190000      T1 = 31          T2 = 43          T3 = 1
MSQ1 = 40
count = 2
0.190000      T1 = 32          T2 = 43          T3 = 1
MSQ1 = 41
count = 1
-----
0.200000      T1 = 33          T2 = 43          T3 = 2
0.190000      T1 = 33          T2 = 43          T3 = 2
MSQ1 = 42
count = 0
0.200000      T1 = 33          T2 = 44          T3 = 2
0.200000      T1 = 33          T2 = 45          T3 = 2
0.200000      T1 = 33          T2 = 46          T3 = 2

```

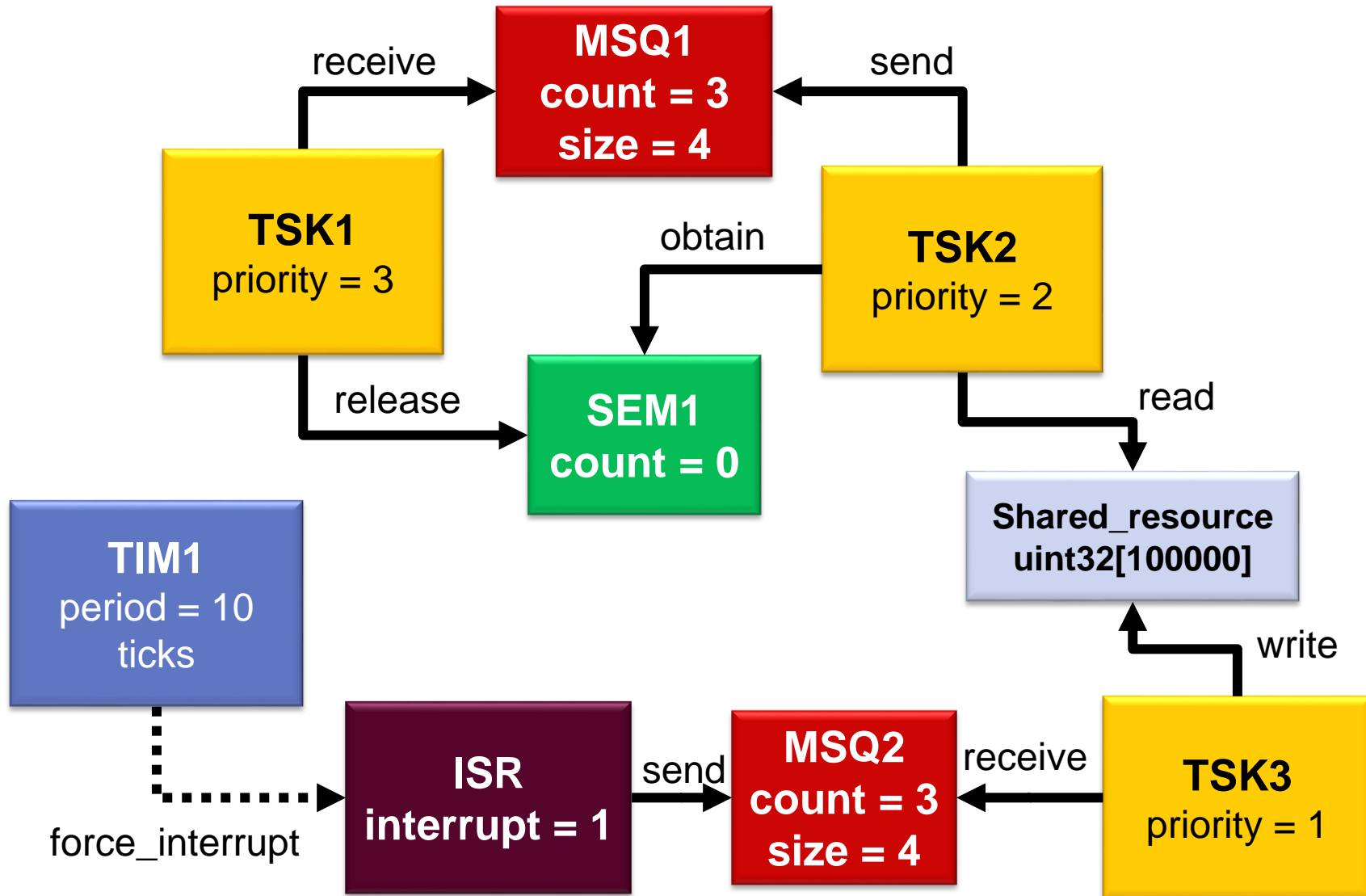
Services et composants d'un RTOS

Exclusion mutuelle

- On souhaite faire évoluer l'application en introduisant une ressource « partagée » entre 2 tâches :
 - Cette ressource partagée est un buffer (tableau) de 100000 entiers 32 bits.
 - La tâche TSK3 remplit chaque élément de ce tableau avec son compteur d'exécution.
 - La tâche TSK2 fait la sommation de tous les éléments de ce tableau et affiche le résultat de cette sommation.

Services et composants d'un RTOS

Exclusion mutuelle



Services et composants d'un RTOS

Exclusion mutuelle

```
#define SHARED_RESOURCE_SIZE 100000

extern uint32_t shared_resource[SHARED_RESOURCE_SIZE];
```

```
rtems_task task_3(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    uint32_t i;

    size_t size;
    rtems_status_code status;

    while (1) {
        status = rtems_message_queue_receive(message_queue_id_2, buffer, &size,
                                             RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        task_counter[T3]++;

        for (i=0; i < SHARED_RESOURCE_SIZE ; i++) {
            shared_resource[i] = task_counter[T3];
        }

        print_task_counter();
    }
}
```

La tâche TSK3 remplit
chaque élément du tableau
shared_resource[] avec son
compteur d'exécution.

Services et composants d'un RTOS

Exclusion mutuelle

```
rtems_task task_2(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    uint32_t shared_resource_sum = 0;
    uint32_t i;
    rtems_status_code status;

    while (1) {
        buffer[0] = task_counter[T1];
        status = rtems_message_queue_send(message_queue_id_1, buffer, sizeof(uint32_t));
        if (status == RTEMS_TOO_MANY) {
            rtems_semaphore_obtain(semaphore_id_1, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        }

        shared_resource_sum = 0;
        for (i=0; i < SHARED_RESOURCE_SIZE ; i++) {
            shared_resource_sum += shared_resource[i];
        }
        printf("shared_resource_sum = %d\n", shared_resource_sum);

        task_counter[1]++;
        print_task_counter();
    }
}
```

La tâche TSK2 fait la sommation de tous les éléments du tableau `shared_resource[]` et affiche le résultat de cette sommation.

Services et composants d'un RTOS

Exclusion mutuelle

```

shared_resource_sum = 0
0.050000      T1 = 0      T2 = 1      T3 = 0
0.140000      T1 = 0      T2 = 1      T3 = 1
shared_resource_sum = 26613
0.160000      T1 = 0      T2 = 2      T3 = 1
0.240000      T1 = 0      T2 = 2      T3 = 2
shared_resource_sum = 133133
0.260000      T1 = 0      T2 = 3      T3 = 2
0.260000      T1 = 1      T2 = 3      T3 = 2
0.260000      T1 = 2      T2 = 3      T3 = 2
0.270000      T1 = 3      T2 = 3      T3 = 2
0.340000      T1 = 3      T2 = 3      T3 = 3
shared_resource_sum = 251015
0.370000      T1 = 3      T2 = 4      T3 = 3
0.440000      T1 = 3      T2 = 4      T3 = 4
shared_resource_sum = 357015
0.470000      T1 = 3      T2 = 5      T3 = 4
0.540000      T1 = 3      T2 = 5      T3 = 5
shared_resource_sum = 463022
0.580000      T1 = 3      T2 = 6      T3 = 5
0.640000      T1 = 3      T2 = 6      T3 = 6
shared_resource_sum = 569028
0.680000      T1 = 3      T2 = 7      T3 = 6
0.680000      T1 = 4      T2 = 7      T3 = 6
0.680000      T1 = 5      T2 = 7      T3 = 6
0.690000      T1 = 6      T2 = 7      T3 = 6
0.740000      T1 = 6      T2 = 7      T3 = 7
shared_resource_sum = 686337
0.790000      T1 = 6      T2 = 8      T3 = 7
0.840000      T1 = 6      T2 = 8      T3 = 8

```

- On observe que le résultat des sommations effectuées par TSK2 n'est jamais un multiple du compteur d'exécution.
- Cela s'explique par le fait que TSK3, étant plus prioritaire que TSK2, préempte TSK2 quand TSK2 est en train de parcourir le tableau pour effectuer la sommation.
- TSK3 modifie l'état de la ressource partagée `shared_resource[]` alors que cette ressource est en cours d'utilisation.
- On peut faire évoluer l'architecture de l'application en protégeant l'accès à la ressource partagée à l'aide d'un service d'exclusion mutuelle

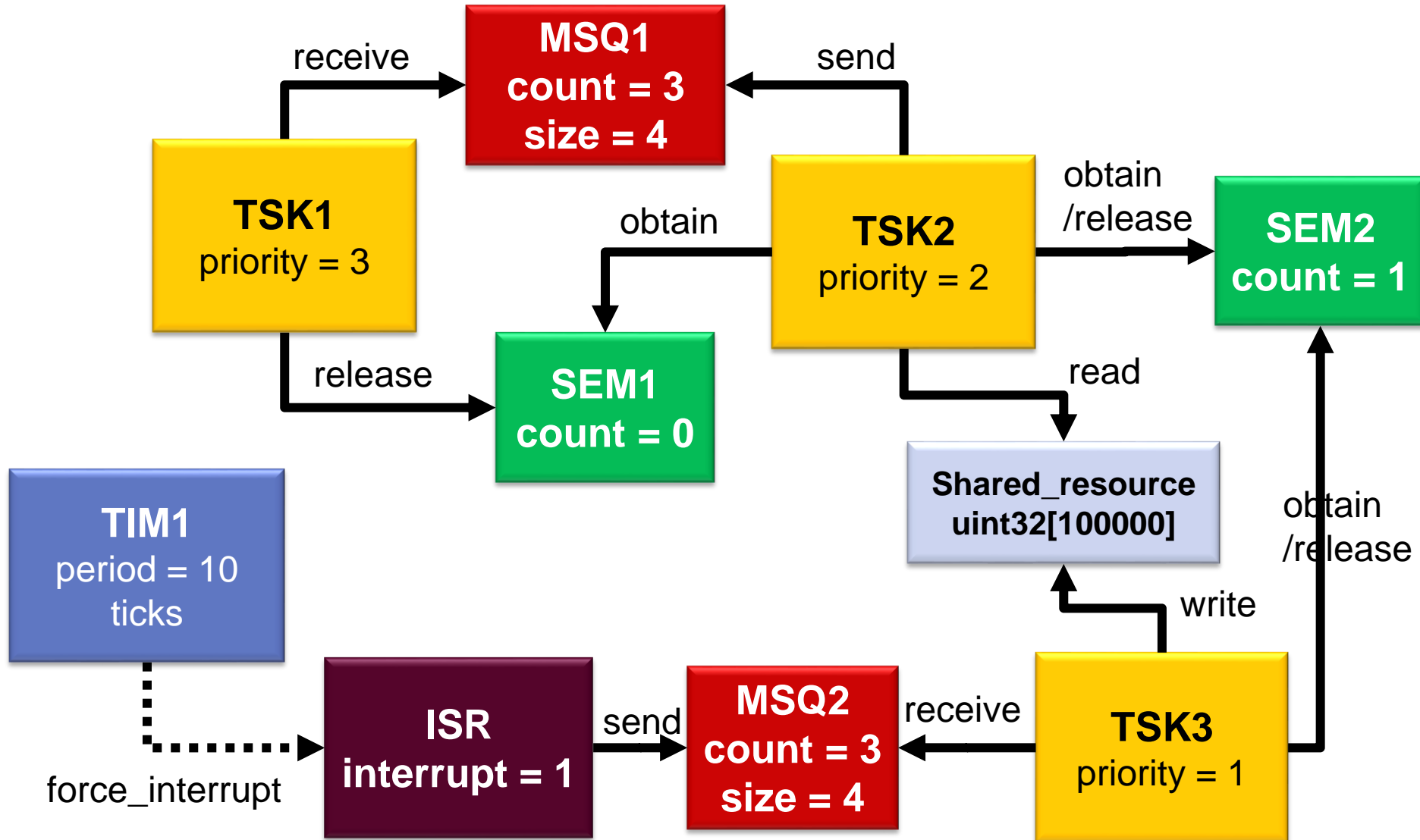
Services et composants d'un RTOS

Exclusion mutuelle

- Les RTOS offrent généralement des services d'exclusion mutuelles qui sont utilisés pour l'accès aux ressources partagées entre plusieurs tâches.
- Ces services reposent sur des composants appelés « mutex » ou sur l'utilisation de sémaphores binaires (cas de RTEMS par exemple).

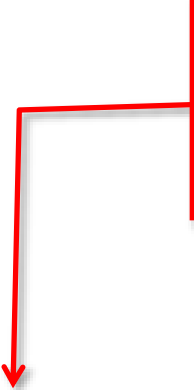
Services et composants d'un RTOS

Exclusion mutuelle



Services et composants d'un RTOS

Exclusion mutuelle



On ajoute à l'application un sémaphore binaire SEM2 qui est initialisé à 1 au démarrage de l'application

```
status = rtems_semaphore_create(rtems_build_name('S','E','M', '2'),1  
    RTEMS_PRIORITY | RTEMS_INHERIT_PRIORITY | RTEMS_BINARY_SEMAPHORE , 0,  
    &semaphore_id_2);
```

Services et composants d'un RTOS

Exclusion mutuelle

```
rtems_task task_3(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    uint32_t i;

    size_t size;
    rtems_status_code status;

    while (1) {
        status = rtems_message_queue_receive(message_queue_id_2, buffer, &size,
                                             RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        task_counter[T3]++;

        rtems_semaphore_obtain(semaphore_id_2, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

        for (i=0; i < SHARED_RESOURCE_SIZE ; i++) {
            shared_resource[i] = task_counter[2];
        }

        rtems_semaphore_release(semaphore_id_2);

        print_task_counter();
    }
}
```

La tâche TSK3 « verrouille »
la ressource partagée
shared_resource[] en
obtenant le sémaphore

La tâche TSK3
« déverrouille » la ressource
partagée shared_resource[]
en relâchant le sémaphore

Services et composants d'un RTOS

Exclusion mutuelle

```
rtems_task task_2(rtems_task_argument unused) {
    uint32_t buffer[BUFFER_SIZE];
    uint32_t shared_resource_sum = 0;
    uint32_t i;
    rtems_status_code status;

    while (1) {
        buffer[0] = task_counter[T1];
        status = rtems_message_queue_send(message_queue_id_1, buffer, sizeof(uint32_t));
        if (status == RTEMS_TOO_MANY) {
            rtems_semaphore_obtain(semaphore_id_1, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
        }

        rtems_semaphore_obtain(semaphore_id_2, RTEMS_WAIT, RTEMS_NO_TIMEOUT);

        shared_resource_sum = 0;
        for (i=0; i < SHARED_RESOURCE_SIZE ; i++) {
            shared_resource_sum += shared_resource[i];
        }
        printf("shared_resource_sum = %d\n", shared_resource_sum);

        rtems_semaphore_release(semaphore_id_2);

        task_counter[1]++;
        print_task_counter();
    }
}
```

La tâche TSK2 « verrouille » la ressource partagée `shared_resource[]` en obtenant le sémaphore

La tâche TSK2 « déverrouille » la ressource partagée `shared_resource[]` en relâchant le sémaphore

Services et composants d'un RTOS

Exclusion mutuelle

shared_resource_sum = 0		
0.050000	T1 = 0	T2 = 1 T3 = 0
shared_resource_sum = 0		
0.150000	T1 = 0	T2 = 1 T3 = 1
0.160000	T1 = 0	T2 = 2 T3 = 1
shared_resource_sum = 100000		
0.260000	T1 = 0	T2 = 2 T3 = 2
0.260000	T1 = 0	T2 = 3 T3 = 2
0.260000	T1 = 1	T2 = 3 T3 = 2
0.270000	T1 = 2	T2 = 3 T3 = 2
0.270000	T1 = 3	T2 = 3 T3 = 2
shared_resource_sum = 200000		
0.370000	T1 = 3	T2 = 3 T3 = 3
0.370000	T1 = 3	T2 = 4 T3 = 3
shared_resource_sum = 300000		
0.470000	T1 = 3	T2 = 4 T3 = 4
0.470000	T1 = 3	T2 = 5 T3 = 4
shared_resource_sum = 400000		
0.580000	T1 = 3	T2 = 5 T3 = 5
0.580000	T1 = 3	T2 = 6 T3 = 5
shared_resource_sum = 500000		
0.680000	T1 = 3	T2 = 6 T3 = 6
0.680000	T1 = 3	T2 = 7 T3 = 6
0.680000	T1 = 4	T2 = 7 T3 = 6
0.690000	T1 = 5	T2 = 7 T3 = 6
0.690000	T1 = 6	T2 = 7 T3 = 6
shared_resource_sum = 600000		
0.790000	T1 = 6	T2 = 7 T3 = 7
0.790000	T1 = 6	T2 = 8 T3 = 7

- On observe que le résultat des sommations effectuées par TSK2 est bien désormais toujours multiple du compteur d'exécution de la tâche TSK3.
- Quand TSK2 parcourt la ressource partagée shared_resource[], elle obtient le sémaphore SEM2.
- Ainsi, TSK3 ne peut plus préempter TSK2 quand TSK2 accède à la ressource partagée.
- La préemption est repoussée au moment où TSK2 relâche le sémaphore SEM2.

Exclusion mutuelle



= blocked



= ready



= executing



= shared
resource access

Services et composants d'un RTOS

Gestion de la mémoire

- Dans le domaine des logiciels embarqués, l'utilisation des fonctions d'allocation dynamique de la mémoire de type malloc() est fortement déconseillée voire interdite par les standards de programmation.
- La plupart des RTOS offre des services intégrés de gestion de la mémoire :
 - Pools de blocs de mémoire de taille fixe (allocation de mémoire rapide et déterministe).
 - Ex. : composant Partition Manager de RTEMS
 - Pools de blocs de mémoire taille non fixe (problématique car non déterministe, fragmentation / défragmentation).
 - Ex. : composant Region Manager de RTEMS

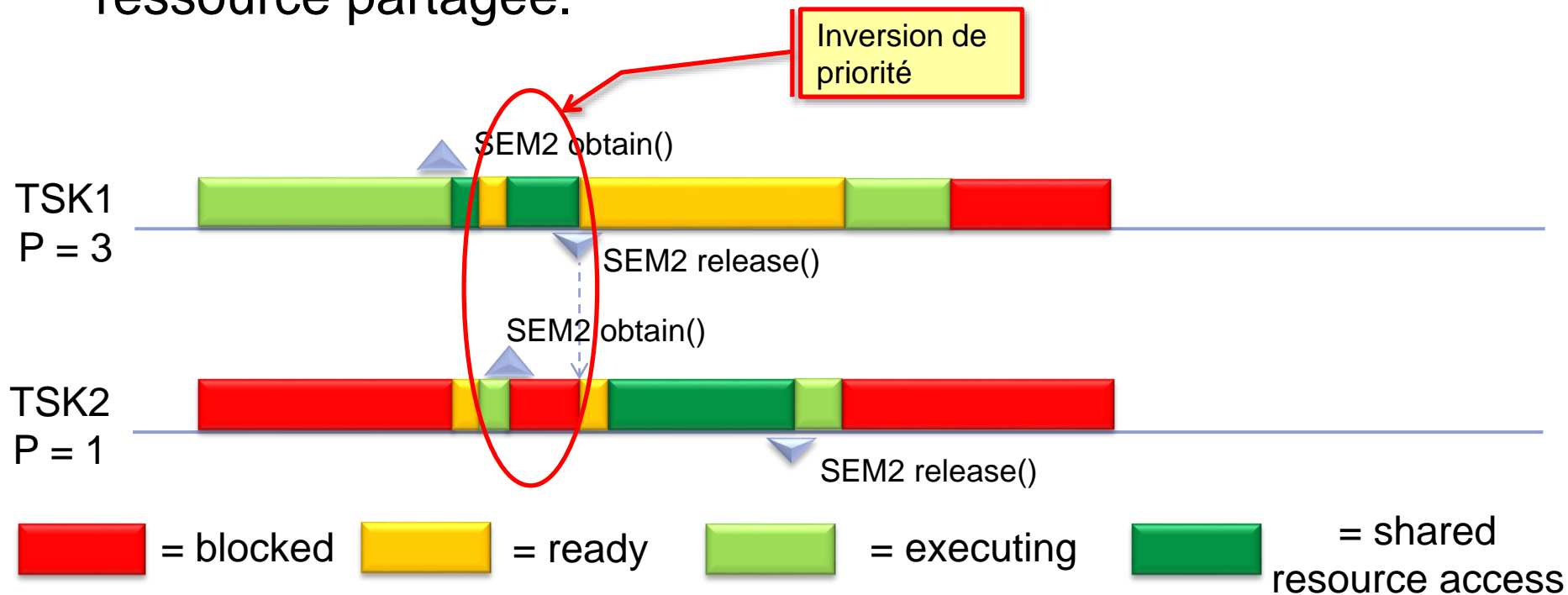
Systèmes d'exploitation temps réel (RTOS)

1. Définition
2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
4. Temps d'exécution, temps de réponse, ordonnançabilité.
5. Aperçu des produits disponibles sur le marché

Problèmes classiques liés aux RTOS

Inversion de priorité

- Le phénomène d'inversion de priorité se produit quand une tâche de plus haute priorité est suspendue (bloquée) à cause d'une tâche de plus basse priorité détenant une ressource partagée.



Problèmes classiques liés aux RTOS

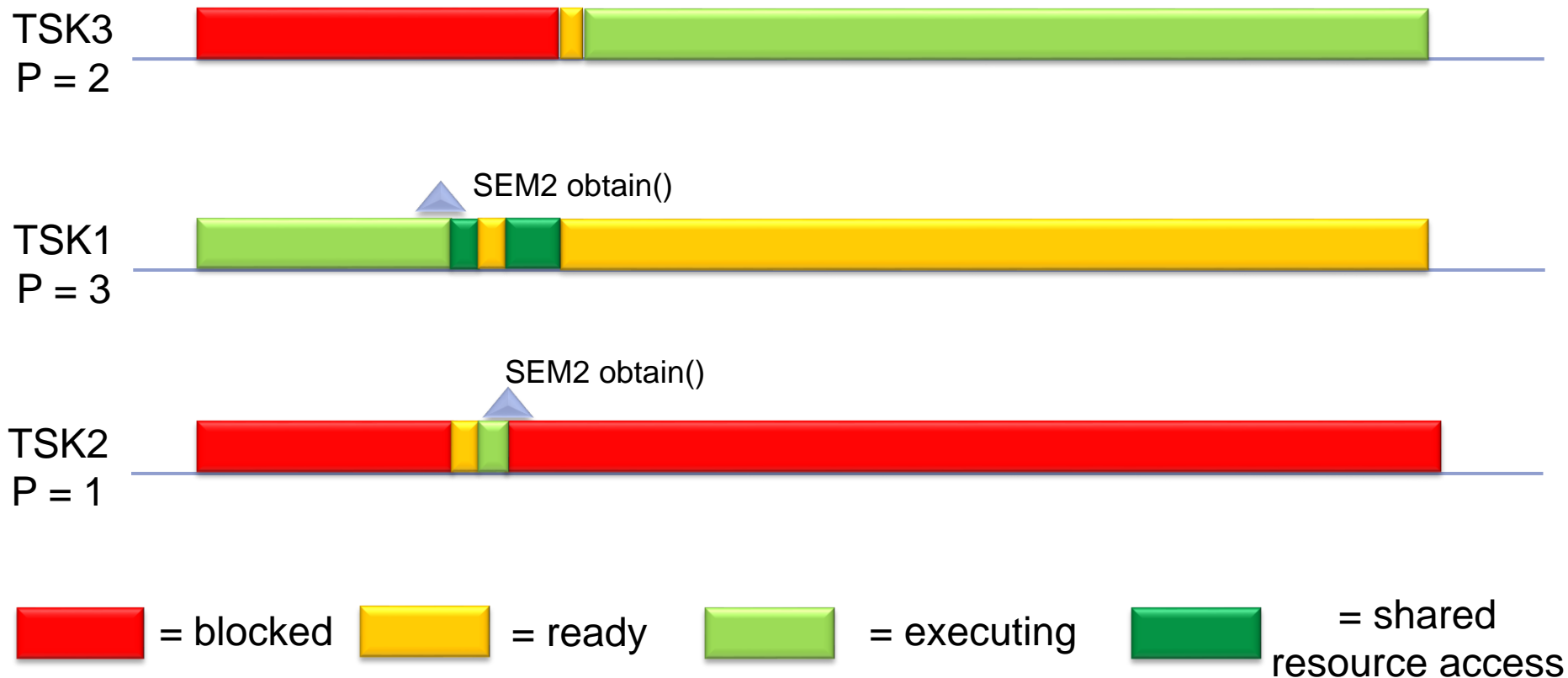
Inversion de priorité

- Ce phénomène est naturelle et courant quand 2 tâches de différentes priorités doivent se partager la même ressource.
- Si ces 2 tâches sont les seules à être actives, alors la durée pendant laquelle l'inversion de priorité a lieu est bornée par le temps durant lequel la tâche de priorité la plus basse détient la ressource partagée.
- Ceci est un comportement normal et déterministe.
- En revanche, si des tâches de priorité intermédiaire deviennent actives durant cette période d'inversion de priorité, alors la durée pendant laquelle l'inversion de priorité a lieu n'est plus déterministe et peut mettre en échec l'application.

Problèmes classiques liés aux RTOS

Inversion de priorité

- Exemple d'un cas d'inversion de priorité problématique : la durée de l'inversion de priorité n'est plus limitée au temps pendant lequel TSK1 détient la ressource partagée.



Problèmes classiques liés aux RTOS

Inversion de priorité

- Les RTOS implémentent généralement le mécanisme dit « d'héritage de priorité » qui permet d'éviter de tomber dans les problèmes d'inversion de priorité non-bornés.
- Le principe est qu'une tâche qui a acquis un mutex ou une sémaphore binaire va, au moment où une autre tâche plus prioritaire essaie d'acquérir la ressource, voir sa priorité modifiée de telle sorte qu'elle soit égale à celle de la tâche plus prioritaire qui est bloquée en attente de la ressource.

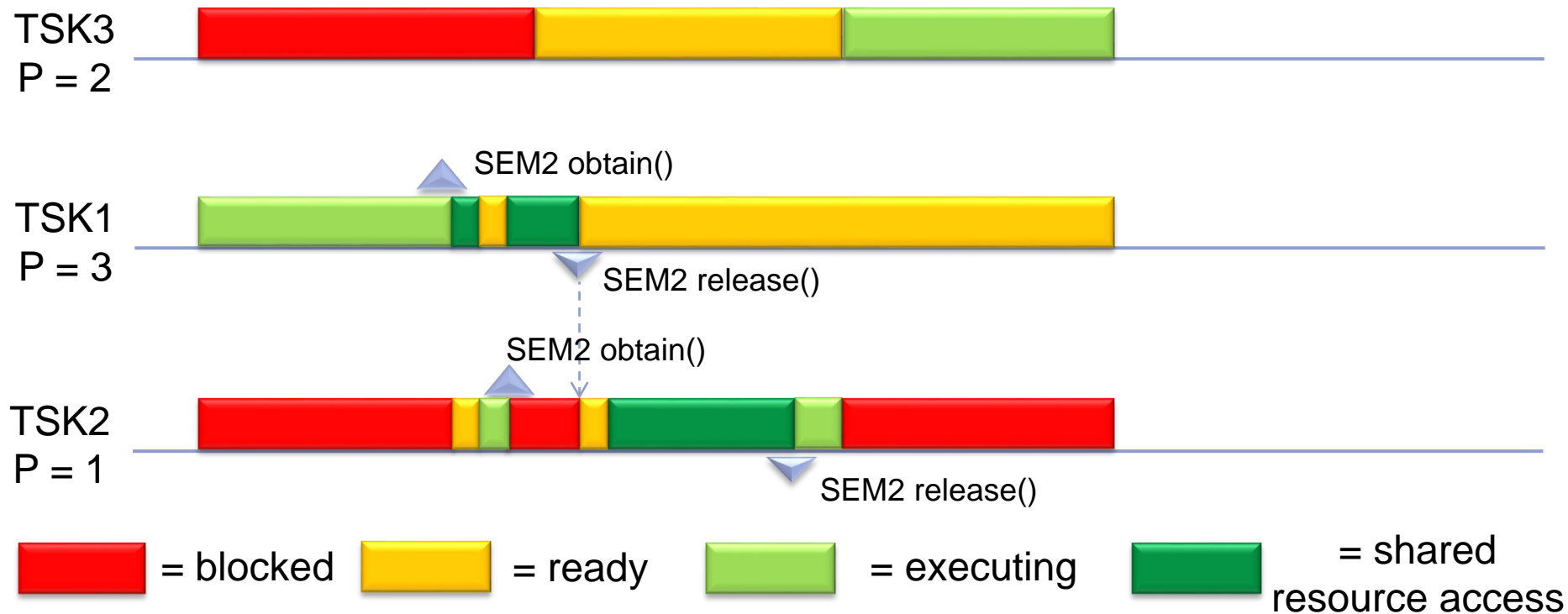
L'option `RTEMS_INHERIT_PRIORITY` permet d'activer pour la sémaphore binaire SEM2 l'héritage de priorité

```
status = rtems_semaphore_create(rtems_build_name('S','E','M','2'), 1,  
    RTEMS_PRIORITY | RTEMS_INHERIT_PRIORITY | RTEMS_BINARY_SEMAPHORE, 0,  
    &semaphore_id_2);
```

Problèmes classiques liés aux RTOS

Inversion de priorité

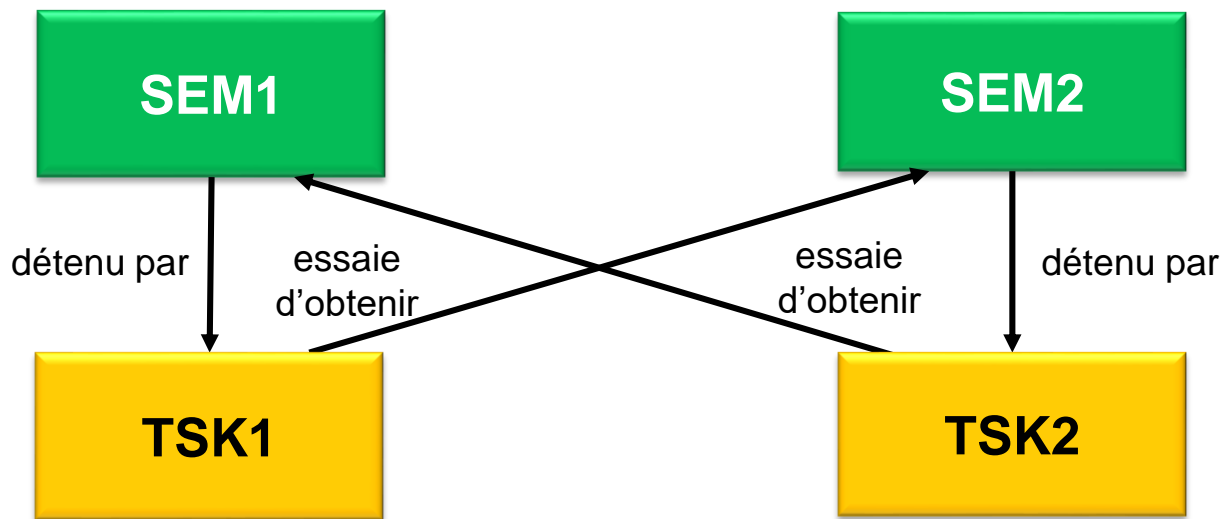
- En activant l'héritage de priorité, la durée de l'inversion de priorité est limitée au temps pendant lequel TSK1 détient la ressource partagée.



Problèmes classiques liés aux RTOS

Deadlock

- Le problème de deadlock est lié à l'utilisation des mécanismes d'exclusion mutuelle (sémaphores binaires, mutex).
- Un deadlock se produit quand une tâche 1 est suspendue (bloquée) indéfiniment en attente d'un sémaphore ou d'un mutex déjà pris par une autre tâche 2 qui elle-même est bloquée indéfiniment en attente d'une ressource détenue par la tâche 1.



Problèmes classiques liés aux RTOS

Deadlock

- Pour éviter les situations de deadlock, il faut, au moment de la conception de l'application, placer des restrictions sur la façon dont les tâches peuvent obtenir les ressources partagées : **les situations de deadlock seront évitées si les tâches ne peuvent acquérir qu'un seul mutex ou sémaphore à la fois.**
- Pour éviter les situations de deadlock, on peut aussi utiliser des appels à la fonction d'obtention du mutex ou du sémaphore utilisant des timeouts.

A la place de RTEMS_NO_TIMEOUT, on peut spécifier un délai de timeout exprimé en nombre de ticks système

```
rtems_semaphore_obtain(semaphore_id_2, RTEMS_WAIT, RTEMS_NO_TIMEOUT);
```

Problèmes classiques liés aux RTOS

Corruption de la pile des tâches

- Avec un RTOS, chaque tâche dispose de sa propre pile dans laquelle sont stockées : le compteur de programme (PC), les variables locales, les arguments des fonctions appelées, le contexte d'exécution de la tâche (copie des registres du processeur) lors des préemptions ou interruptions.
 - Un mauvais dimensionnement de la pile d'une tâche peut provoquer de grave dysfonctionnement : un débordement de pile va causer l'écrasement des données adjacentes à la pile.
 - Cela peut se traduire par exemple par la corruption du compteur de programme (PC) et un plantage du processeur.

C'est ici que l'on précise la taille de la pile de la tâche.

```
status = rtems_task_create(rtems_build_name('T','S','K', '1'),  
    TASK1_PRIORITY, TASK_STACK_SIZE,  
    RTEMS_PREEMPT | RTEMS_NO_TIMESLICE | RTEMS_INTERRUPT_LEVEL(0),  
    RTEMS_LOCAL | RTEMS_FLOATING_POINT, &task_id_1);
```

Problèmes classiques liés aux RTOS

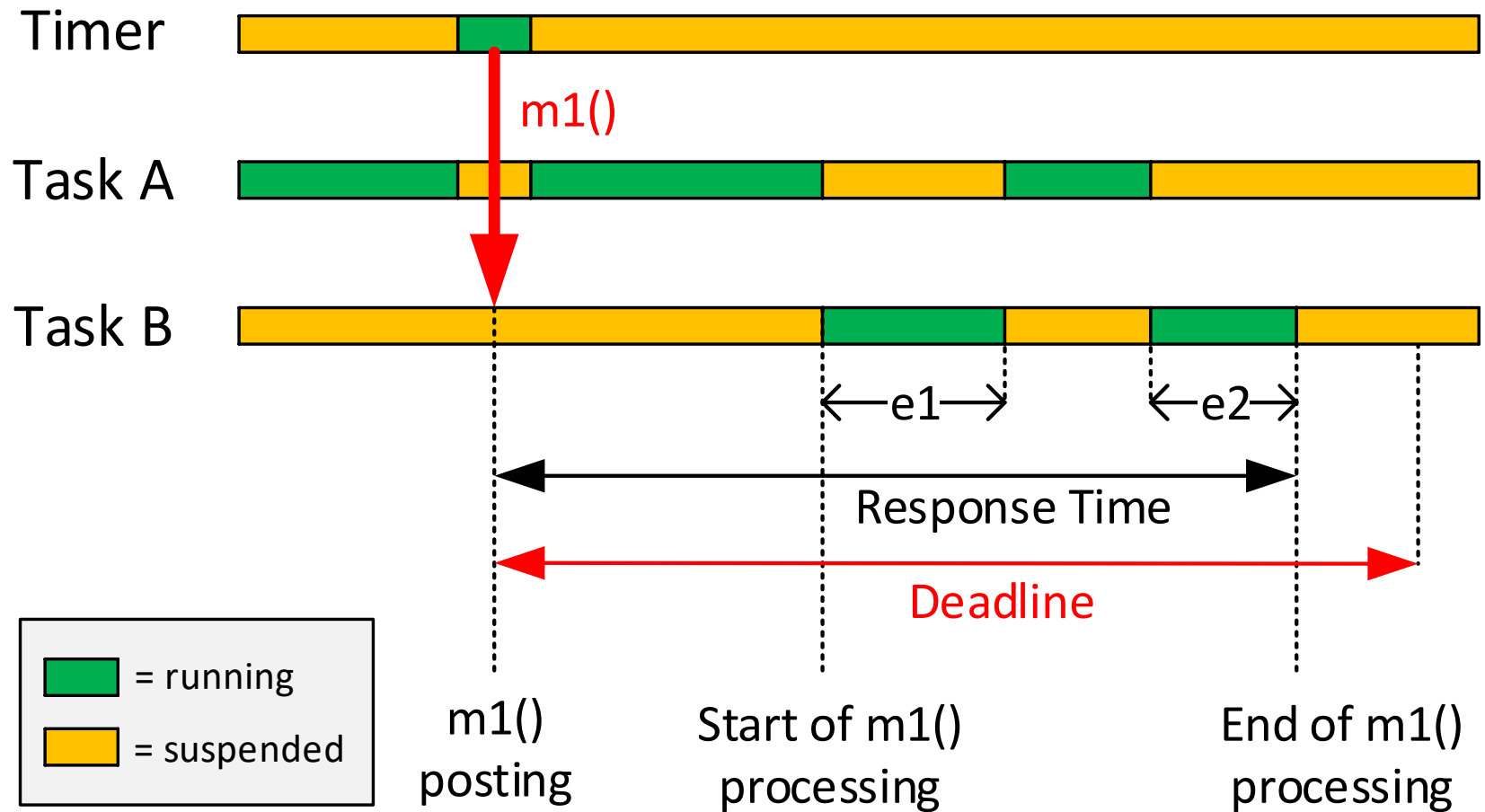
Corruption de la pile des tâches

- Certains RTOS offrent des services permettant de monitorer le taux d'occupation des piles.
 - Exemple :
 - Fonction `rtems_stack_checker_report_usage()` de RTEMS qui retourne la taille de chaque pile et son niveau d'utilisation maximum.
 - Fonction `rtems_stack_checker_is_blow()` de RTEMS qui détermine si le pointeur de pile pointe sur une zone en dehors de la pile.

Systèmes d'exploitation temps réel (RTOS)

1. Définition
2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
- 4. Temps d'exécution, temps de réponse, ordonnançabilité.**
5. Aperçu des produits disponibles sur le marché

Temps d'exécution, temps de réponse, ordonnançabilité.



Temps d'exécution, temps de réponse, ordonnançabilité.

- Temps de réponse d'une tâche (response time) = temps total écoulé entre l'instant où un message d'activation d'une tâche est émis et l'instant où le traitement associé est terminé
 - Le temps de réponse inclut tous les délais induits par
 - toutes les interférences avec les autres tâches (préemptions), timers et interruptions de l'application,
 - les attentes potentielles liées aux accès aux ressources partagées.
- Temps d'exécution maximum (WCET = worst case execution time) = temps maximum qu'une tâche prendrait pour effectuer son traitement si elle pouvait s'exécuter sans aucune interférence et sans attentes liées aux ressources partagées.

Temps d'exécution, temps de réponse, ordonnançabilité.

- Échéance (deadline) = durée au-delà de laquelle la réponse de la tâche n'est plus valide (échéance de terminaison au plus tard)
- Un système est dit « ordonnançable » si l'on peut garantir, pour chaque tâche, que tous les temps de réponse pire cas sont inférieurs aux deadlines.
- Différentes approches pour analyser l'ordonnançabilité d'un système :
 - Analyse WCET statique (sans exécution directe sur le matériel)
 - Approche basée sur des mesures sur cible matérielle

Analyse d'ordonnement et modèle AADL

- Une approche consiste à décrire avec un modèle AADL (Architecture Analysis and Design Language) l'architecture d'un système embarqué en termes de composants matériels et de composants logiciels interagissant.
 - Composants matériels : processeur(s), bus mémoire, accès DMA, accès RAM, caches, périphériques, ...
 - Composants logiciels : tâches, ressources partagées, mécanismes de synchronisation, ...
 - Connexions entre les différents composants
- L'approche AADL permet d'analyser l'interaction entre les composants logiciels et les composants matériels.
- Un modèle AADL est le point d'entrée pour des outils d'analyse d'ordonnement.
- Les analyses d'ordonnement peuvent être réalisées bien avant que le système final soit développé.

Analyse d'ordonnancement et modèle AADL

- Outils d'analyse d'ordonnancement :
 - Ex : Cheddar (<http://beru.univ-brest.fr/~singhoff/cheddar/>)
- Outils AADL :
 - AADL Inspector (<https://www.ellidiss.fr/public/wiki/inspector>)
 - Visualisation / Simulation / Analyse (via Cheddar)

Analyse d'ordonnancement et modèle AADL

- Exemple de définition d'un process simple avec AADL :

```
PROCESS ndpu_asw_process
END ndpu_asw_process;

PROCESS IMPLEMENTATION ndpu_asw_process.impl
SUBCOMPONENTS
    -- THREADS
    this_thread1_thr: THREAD thread1_thr.impl;
    this_thread2_thr: THREAD thread2_thr.impl;

    -- SHARED DATA
    this_sharedData_dat: DATA sharedData_dat.impl;

CONNECTIONS
    sharedData_conn: DATA ACCESS this_sharedData_dat -> this_thread1_thr.sharedData_dat;
    sharedData_conn2: DATA ACCESS this_sharedData_dat -> this_thread2_thr.sharedData_dat;

END ndpu_asw_process.impl;
```

Analyse d'ordonnancement et modèle AADL

- Exemple de définition d'un ensemble de tâches avec AADL :
 - thread1_thr est plus prioritaire que thread2_thr

```
THREAD thread1_thr
FEATURES
    sharedData_dat : REQUIRES DATA ACCESS sharedData_dat.impl;
PROPERTIES
    Priority => 255;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Dispatch_Protocol => Periodic;
    period => 5ms;
    Deadline => 5ms;
    Compute_Execution_Time => 1ms..1ms;
END thread1_thr;

THREAD IMPLEMENTATION thread1_thr.impl
END thread1_thr.impl;

THREAD thread2_thr
FEATURES
    sharedData_dat : REQUIRES DATA ACCESS sharedData_dat.impl;
PROPERTIES
    Priority => 251;
    POSIX_Scheduling_Policy => SCHED_FIFO;
    Dispatch_Protocol => Periodic;
    period => 10ms;
    Deadline => 10ms;
    Compute_Execution_Time => 8ms..8ms;
END thread2_thr;

THREAD IMPLEMENTATION thread2_thr.impl
END thread2_thr.impl;
```

Analyse d'ordonnancement et modèle AADL

- Exemple de définition d'une ressource partagée avec AADL :

```
DATA sharedData_dat
PROPERTIES
    Concurrency_Control_Protocol => Priority_Inheritance;
END sharedData_dat;
```

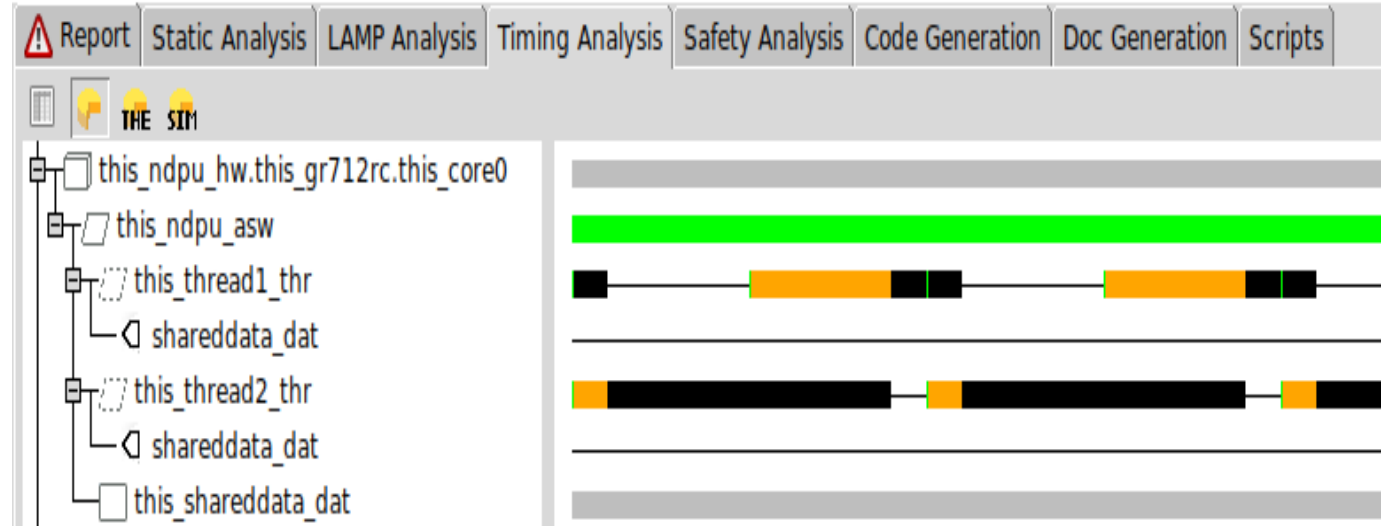
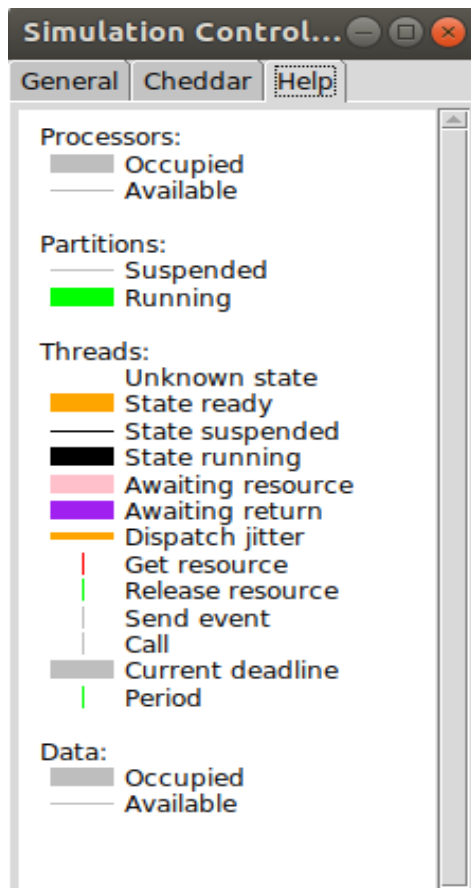
Analyse d'ordonnancement et modèle AADL

- Ordonnancement obtenu par analyse si la ressource partagée est désactivée :



Analyse d'ordonnancement et modèle AADL

- Ordonnancement obtenu par analyse si la ressource partagée est activée et que le protocole « priority inheritance » est sélectionné :
 - thread2_thr hérite de la priorité de thread1_thr aussi longtemps qu'il détient la ressource partagée :



Analyse d'ordonnancement et modèle AADL

- Exemple de résultat de vérification des timings via une analyse d'ordonnancement Cheddar lancée avec AADL Inspector :

	Deadline	Computed	Max Cheddar	Max Marzhin	Avg Cheddar	Avg Marzhin	Min Cheddar	Min Marzhin
▼ this_ndpu_hw.this_gr712rc.this_core0		141.19 %		20.27 %				
▼ this_ndpu_asw_process0								
this_intercoremanager_0_processintercorequeues_thr	45	1.00000	1	1	1.00	1.00	1	1
this_timemanager_0_processtimecode_thr	10	2.00000	2	2	2.00	2.00	2	2
this_timerecorder_0_recordtime_thr	700	3.00000	3	3	3.00	3.00	3	3
this_nicuspacewiremanager_0_processinterrupt_thr	18	4.00000	4		1.24		1	
this_nicuspacewiremanager_0_armpackettransmission_thr	36	5.00000	5		2.28		2	
this_nicuspacewiremanager_0_monitor_thr	112	5.00000	6	4	2.12	2.33	1	1
this_heartbeatproducer_0_produceheartbeatpacket_thr	100	7.00000	7	5	7.00	5.00	7	5
this_nfeespacewiremanager_0_processinterrupt_thr	36	8.00000	8		3.36		3	
this_nfeespacewiremanager_0_armpackettransmission_thr	36	9.00000	9		4.36		4	
this_nfeespacewiremanager_0_monitor_thr	112	9.00000	10	6	3.50	3.67	2	2
this_datapoolmanager_0_updatedatapool_thr	112	13.00000	13	9	6.50	6.67	5	5
this_eventproducer_0_processevent_thr	90	14.00000	14		5.80		3	
this_hkmanager_0_processperiodichk_thr	112	15.00000	15	16	8.38	13.67	6	10
this_monitor_0_monitor_thr	112	22.00000	22	16	15.12	13.00	12	11
this_tmmanager_0_processtmbuffers_thr	112	25.00000	25	19	18.12	16.00	15	14
this_cameraengine_0_processpacket_thr	36	32.00000	912		74.73		5	
this_cameraengine_0_processtimecode_thr	36	27.00000	31	25	25.25	25.00	19	25
this_cameraengine_0_notifyendofreadout_thr	20	26.00000	29	23	23.75	23.00	19	23
this_cameraengine_0_notifyendofframe_thr	90	28.00000	28	22	25.25	22.00	22	22
this_cameraengine_0_executewindowprocessing_thr	1750	911.00000	911		909.25		906	
this_scienceprocessor_0_executescienceprocessing_thr	4500	3672.00000	3672		3670.00		3664	
this_watchdogreloader_0_reloadwatchdog_thr	6250	3673.00000	3673		1164.48		27	
this_memoryscrubber_0_readmemory_thr	6250	4040.00000	4040		4035.25		4030	

Systèmes d'exploitation temps réel (RTOS)

1. Définition
2. Services et composants fonctionnels d'un RTOS (Tâches, queues de message, timers, interruptions, sémaphores, mutex, ...)
3. Problèmes classiques : inversion de priorité, deadlock, ...
4. Temps d'exécution, temps de réponse, ordonnançabilité.
5. Aperçu des produits disponibles sur le marché

RTOS

Principaux produits

■ Principaux RTOS :

- ThreadX: <https://rtos.com/solutions/threadx/real-time-operating-system/>
- FreeRTOS: <https://freertos.org/>
- RTEMS: <https://www.rtems.org/>
- eCos: <http://ecos.sourceware.org/>
- VxWorks: <https://www.windriver.com/products/vxworks/>
- QNX Neutrino RTOS:
<http://blackberry.qnx.com/en/products/neutrino-rtos/neutrino-rtos>