Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

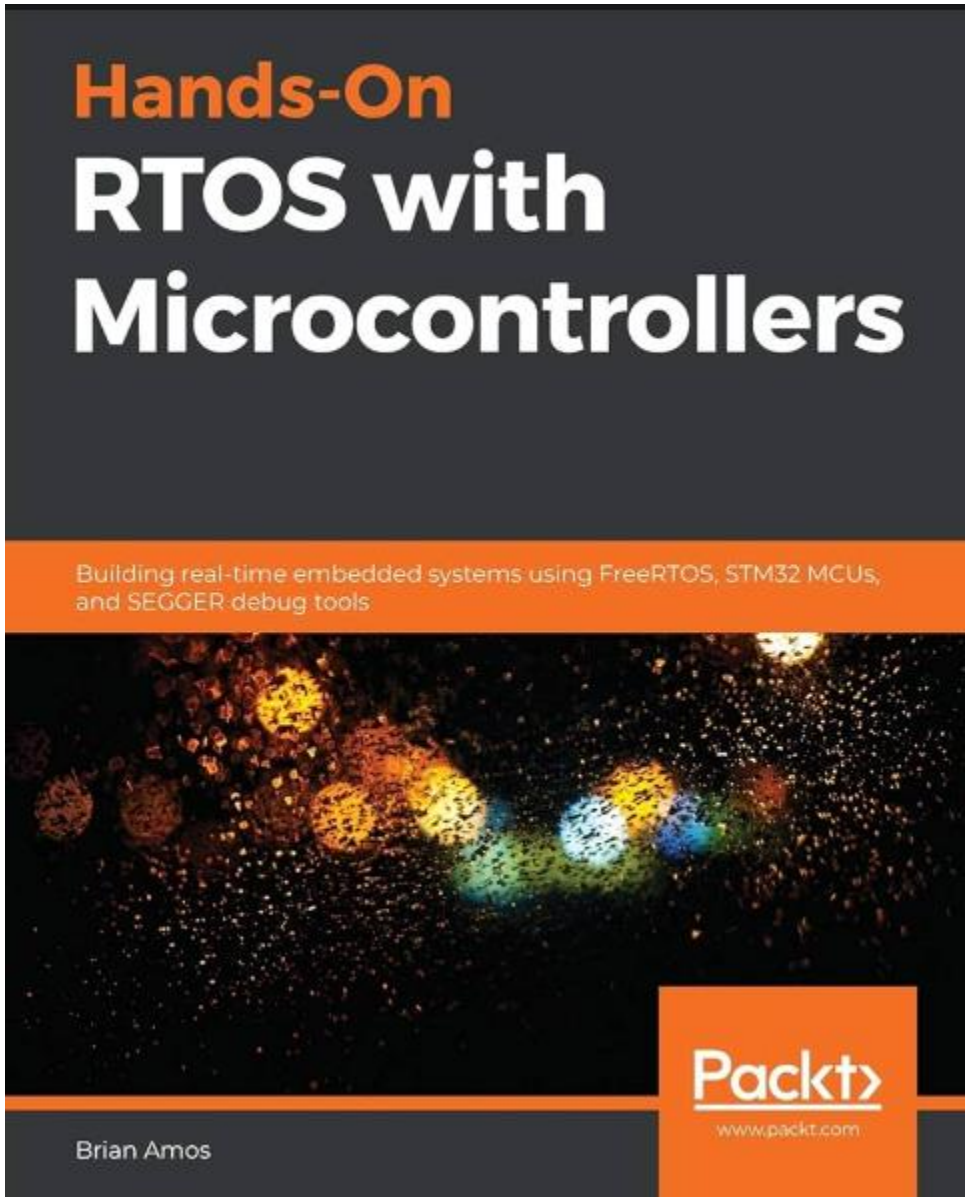
CEG4566/CSI4541/SEG4545

Conception de systèmes informatiques en temps réel

Hiver 2024

Professeur : Mohamed Ali Ibrahim, ing., Ph.D.

Source :


Hands-On
RTOS with
Microcontrollers
Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools
Packt
www.packt.com
Brian Amos

Chapitre 3 :

Mécanismes de signalisation et de communication des tâches

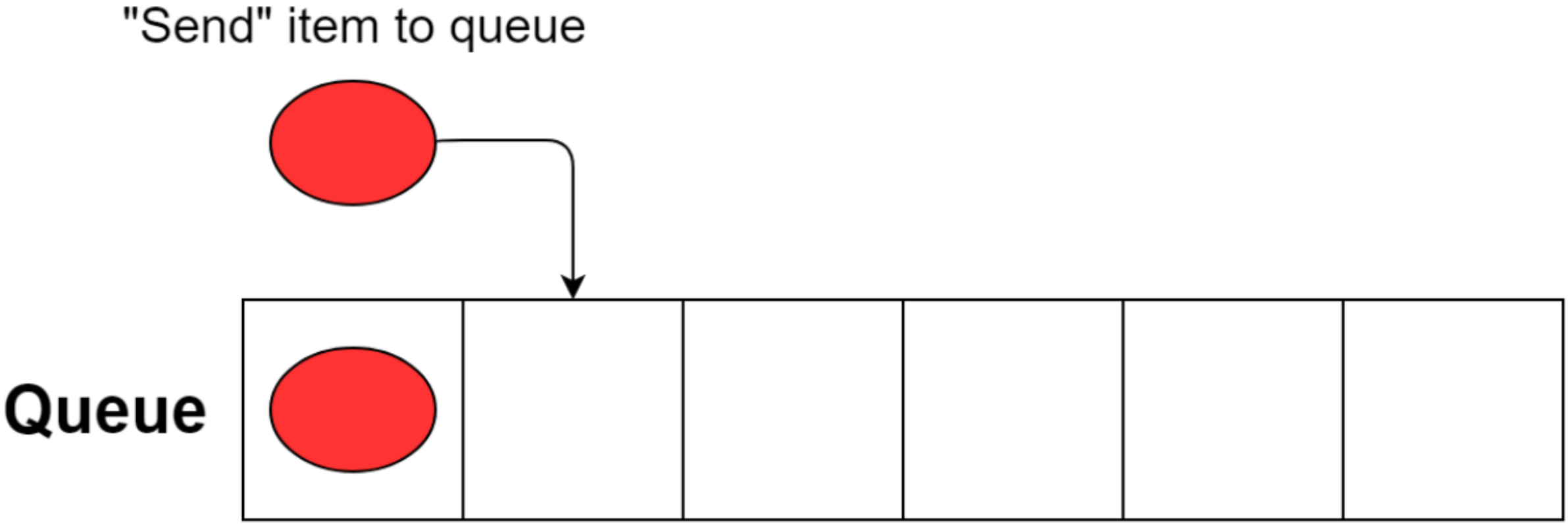
Plan

- Files d'attente RTOS
- Sémaphores RTOS
- Mutex RTOS

Files d'attente RTOS

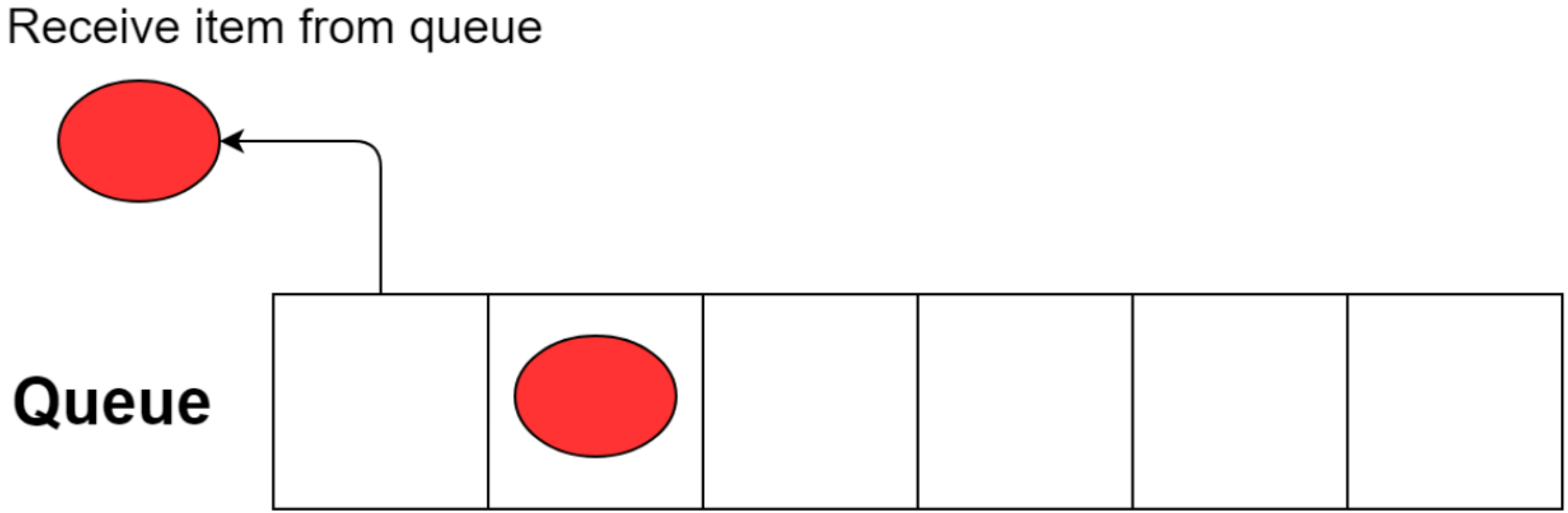
- Les files d'attente sont un concept assez simple, mais elles sont aussi extrêmement puissantes et flexibles, en particulier si vous avez traditionnellement programmé en C sur du bare metal.
- Au fond, une file d'attente est simplement une mémoire tampon circulaire.
- Toutefois, ce tampon présente des propriétés très particulières, telles que la sécurité multithread native, la possibilité pour chaque file d'attente de contenir n'importe quel type de données et de réveiller d'autres tâches qui attendent qu'un élément apparaisse dans la file d'attente.
- Par défaut, les données sont stockées dans des files d'attente selon la méthode FIFO (First In First Out)
  - le premier élément à être mis dans la file d'attente est le premier élément à être retiré de la file d'attente.

Envoi d'une simple file d'attente



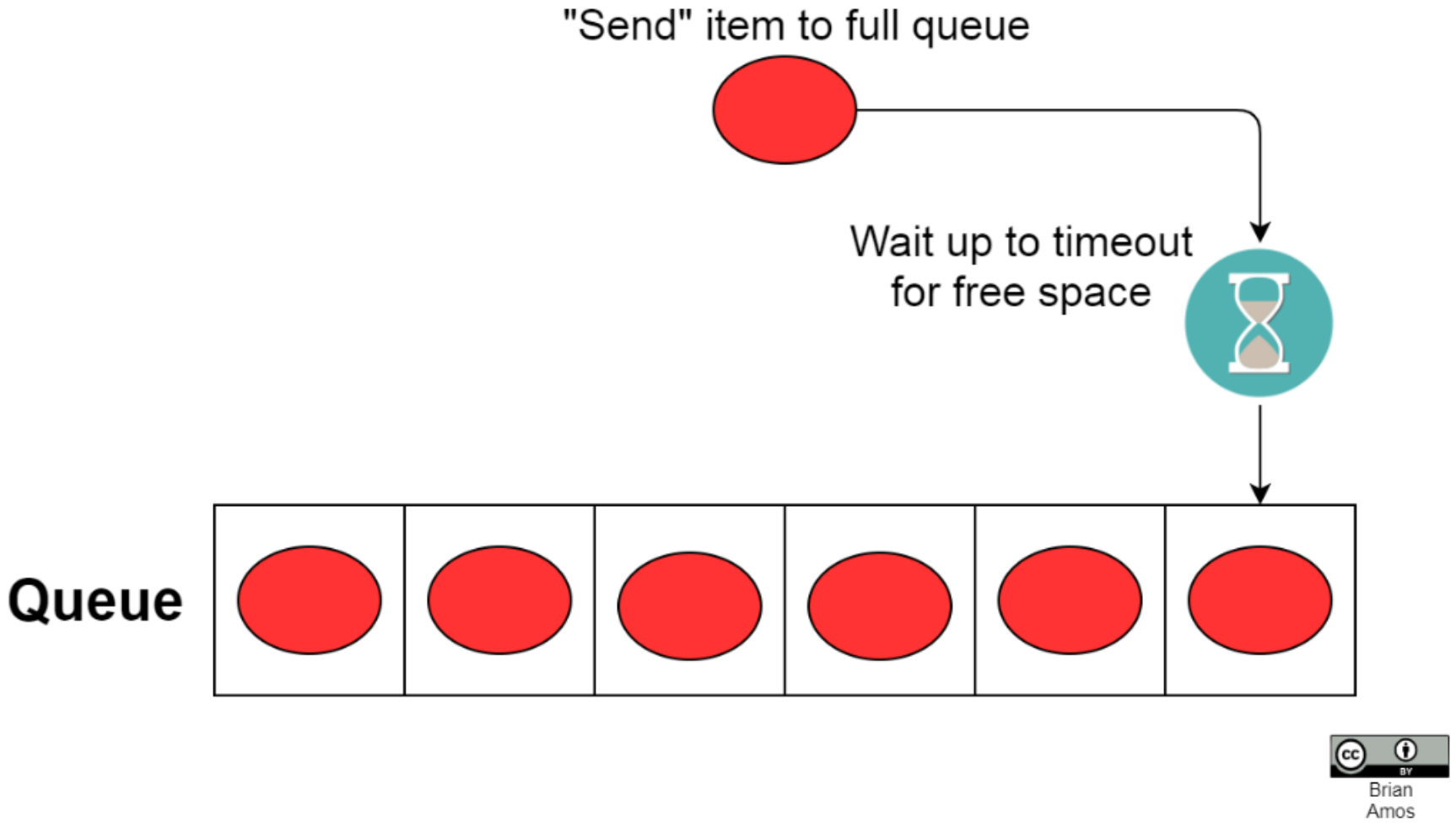
- Lorsqu'un élément est ajouté à une file d'attente disposant d'un espace disponible, l'ajout se fait immédiatement.
- Comme il y avait de la place dans la file d'attente, la tâche qui *envoie l'*élément à la file d'attente continue à fonctionner, à moins qu'une autre tâche plus prioritaire n'attende qu'un élément apparaisse dans la file d'attente.

Réception d'une simple file d'attente



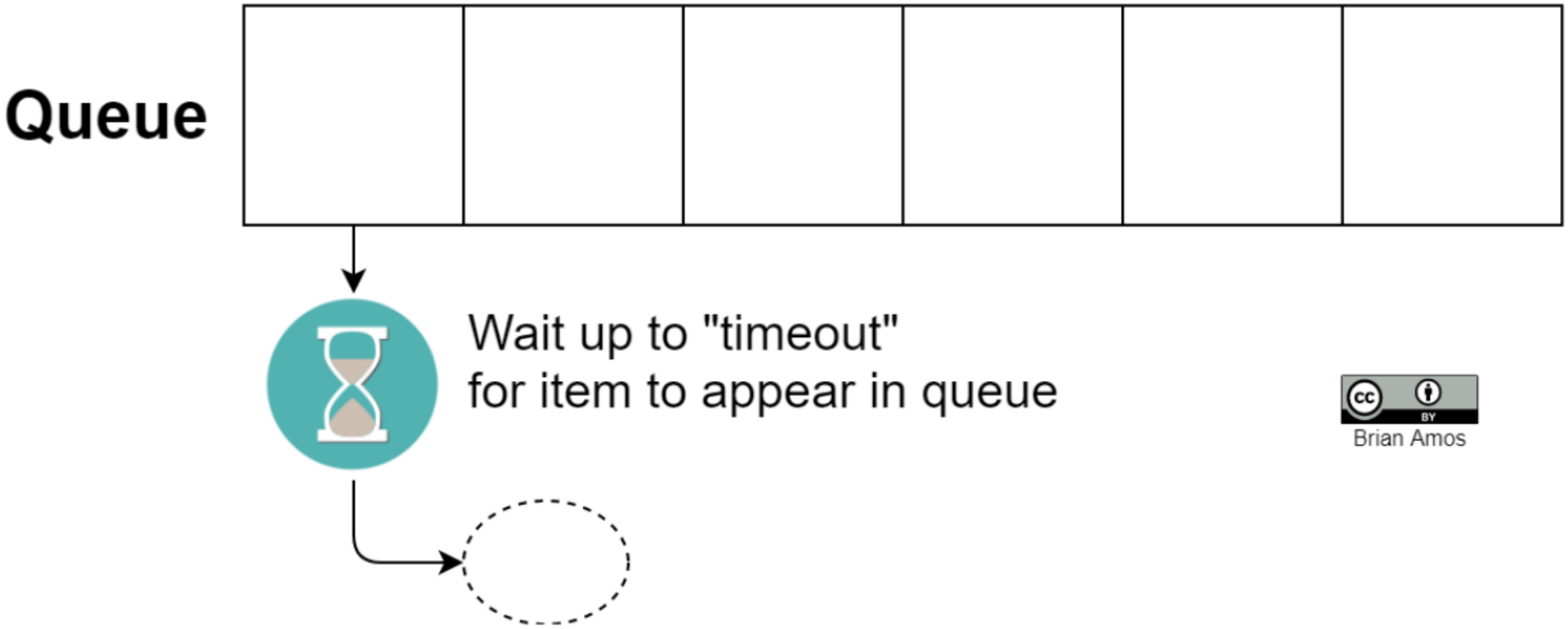
- Lorsqu'une tâche est prête à recevoir un élément d'une file d'attente, elle récupère par défaut l'élément le plus ancien.
- Dans cet exemple, comme il y a au moins un élément dans la file d'attente, la réception est traitée immédiatement et la tâche continue à s'exécuter.

Envoi d'une file d'attente complète



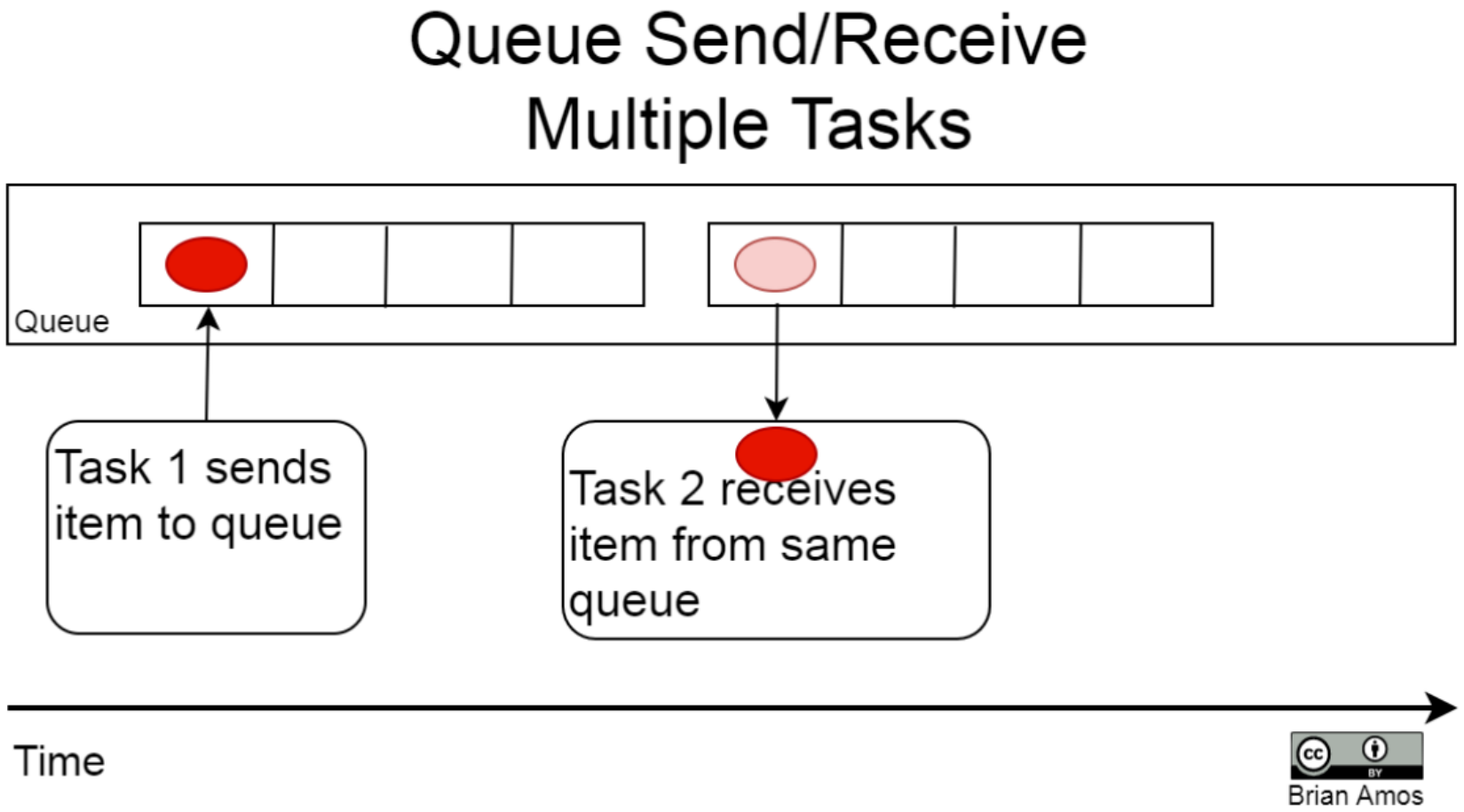
Lorsqu'une file d'attente est pleine, la tâche qui tente d'envoyer un élément à la file d'attente attendra jusqu'à ce qu'une place se libère dans la file d'attente, mais seulement jusqu'à la valeur du délai spécifié.

Réception d'une file d'attente vide



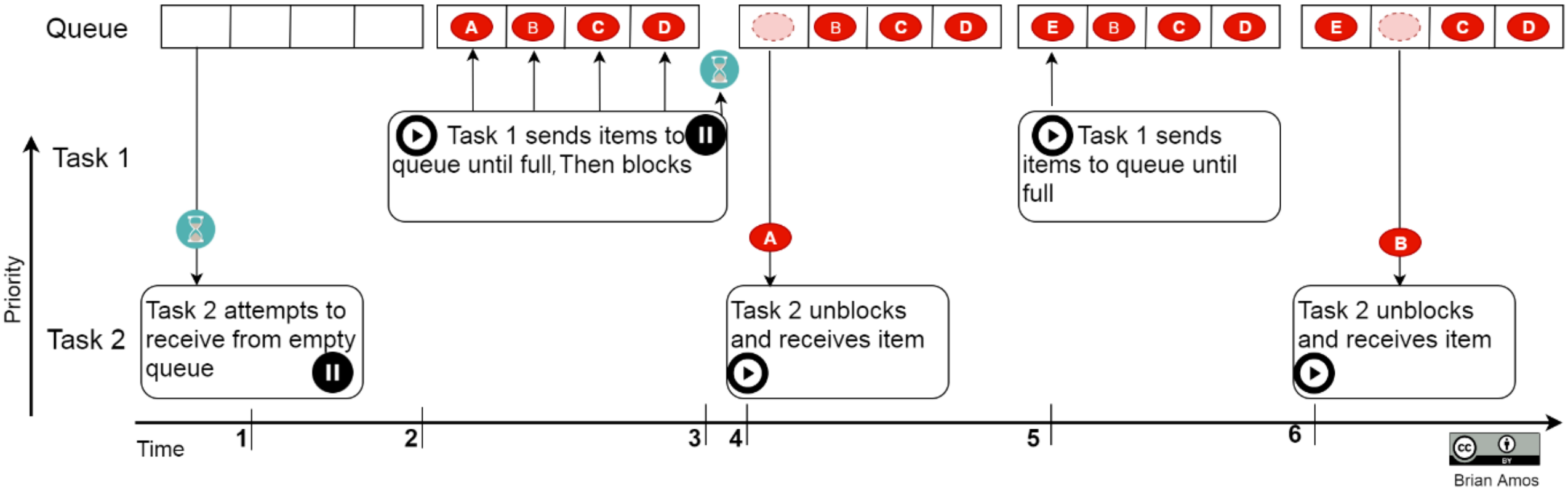
- Dans le cas d'une file d'attente vide, la tâche qui tente de recevoir un élément de la file d'attente sera bloquée jusqu'à ce qu'un élément apparaisse dans la file d'attente.
- Si aucun élément n'est disponible avant l'expiration du délai, le code appelant est informé de l'échec.
- Là encore, la marche à suivre exacte varie.

Files d'attente pour la communication entre les tâches



- Dans l'exemple précédent, la tâche 1 et la tâche 2 interagissent avec la même file d'attente.
- La tâche 1 enverra un élément dans la file d'attente.
- Tant que la tâche 2 a une priorité plus élevée que la tâche 1, elle recevra immédiatement l'élément.

File d'attente Envoyer/Recevoir plusieurs tâches Priorité affectée (1/2)

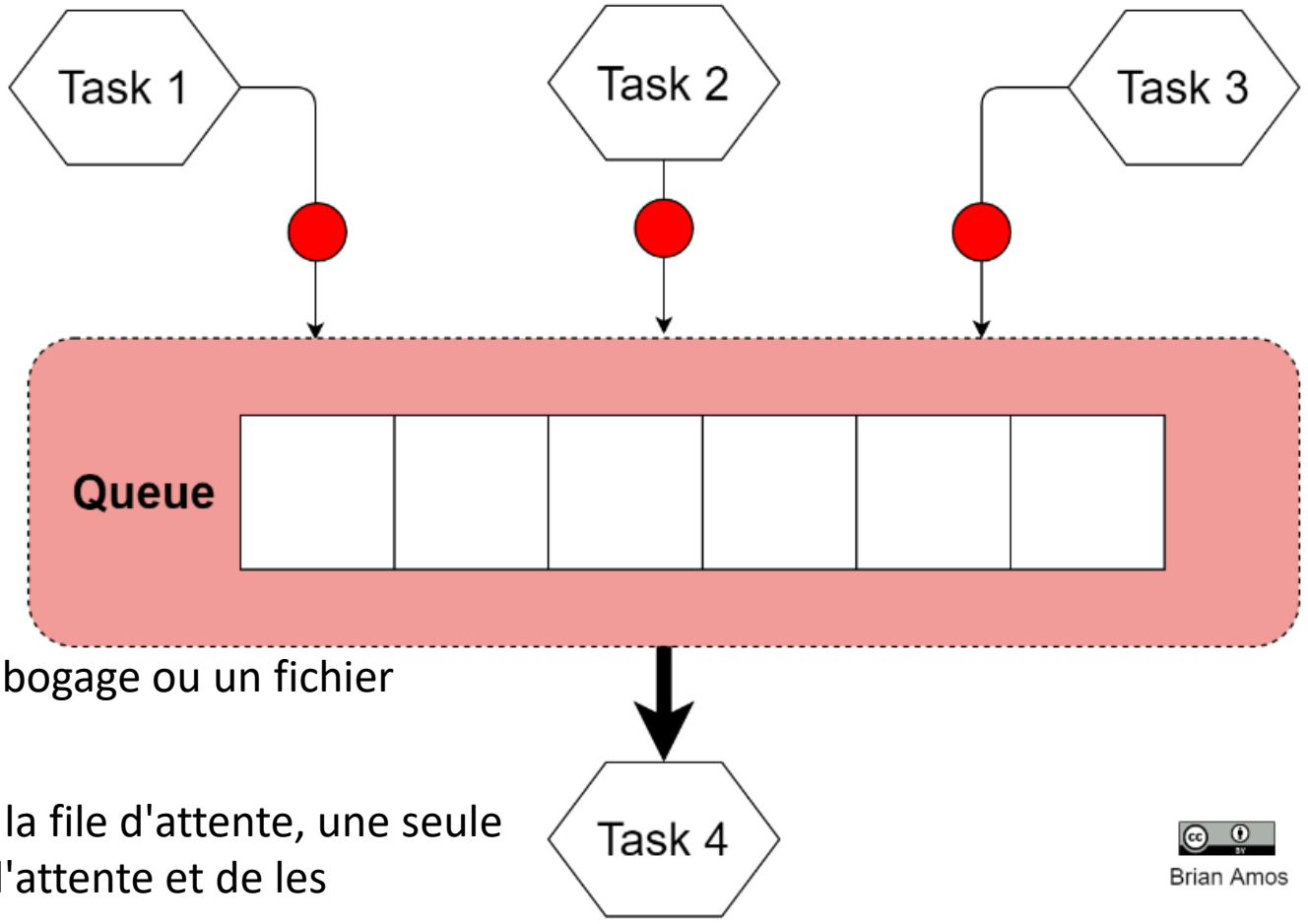


File d'attente d'envoi/réception de tâches multiples Affectation des priorités (2/2)

Les numéros suivants correspondent à des indices sur l'axe du temps :

1. La tâche 2 tente de recevoir un élément de la file d'attente vide. Aucun élément n'étant disponible, la tâche 2 se bloque.
2. La tâche 1 ajoute des éléments à la file d'attente. Étant donné qu'il s'agit de la tâche la plus prioritaire du système, la tâche 1 ajoute des éléments à la file d'attente jusqu'à ce qu'elle n'ait plus d'éléments à ajouter ou jusqu'à ce que la file d'attente soit pleine.
3. La file d'attente est remplie, la tâche 1 est donc bloquée.
4. La tâche 2 se voit attribuer un contexte par le planificateur puisqu'elle est désormais la tâche la plus prioritaire qui peut être exécutée.
5. Dès qu'un élément est retiré de la file d'attente, la tâche 1 retrouve son contexte (il s'agit de la tâche la plus prioritaire du système, et elle peut maintenant s'exécuter car elle était bloquée en attendant qu'un espace se libère dans la file d'attente). Après l'ajout d'un seul élément, la file d'attente est pleine et la tâche 1 est bloquée.
6. La tâche 2 reçoit un contexte et un élément de la file d'attente.

File d'attente Tâche multiple Écriture Lecteur unique



- Ceci est particulièrement utile pour un port série de débogage ou un fichier journal.
- De nombreuses tâches différentes peuvent écrire dans la file d'attente, une seule tâche étant chargée de recevoir les données de la file d'attente et de les transmettre à la ressource partagée.
- Alors que les files d'attente sont généralement utilisées pour transmettre des données entre les tâches, les sémaphores peuvent être utilisés pour signaler et synchroniser les tâches.

Sémaphores RTOS

- Les sémaphores sont utilisés pour indiquer que quelque chose s'est produit ; ils signalent des événements.

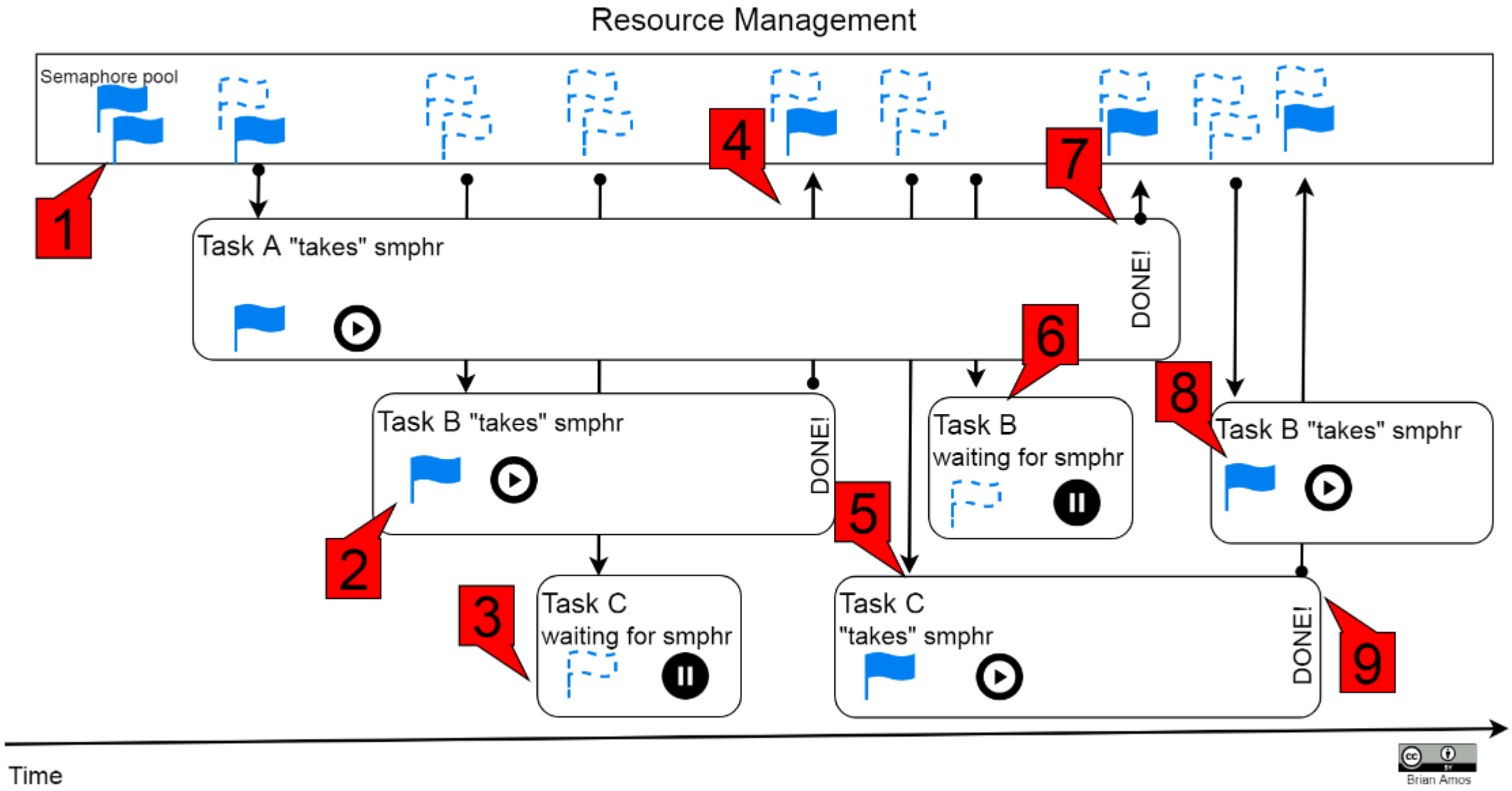
- Voici quelques exemples d'utilisation des sémaphores :
  - Un ISR a terminé l'entretien d'un périphérique. Il peut donner un sémaphore pour fournir aux tâches un signal indiquant que les données sont prêtes pour un traitement ultérieur.
  - Une tâche a atteint un point où elle doit attendre que les autres tâches du système la rattrapent avant de continuer. Dans ce cas, un sémaphore peut être utilisé pour synchroniser les tâches.
  - Restreindre le nombre d'utilisateurs simultanés d'une ressource restreinte.

- Il existe deux types de sémaphores différents
  - les sémaphores de comptage et les sémaphores binaires.

Compter les sémaphores (1/4)

- Les sémaphores de comptage sont le plus souvent utilisés pour gérer une ressource partagée dont le nombre d'utilisateurs simultanés est limité.
- Lors de leur création, ils peuvent être configurés pour contenir une valeur maximale, appelée plafond.
- Dans le diagramme suivant (diapositive suivante), nous avons une ressource réseau partagée qui peut accueillir deux connexions de socket simultanées.
- Cependant, trois tâches nécessitent un accès.
- Un sémaphore de comptage est utilisé pour limiter le nombre de connexions simultanées.
- Chaque fois qu'une tâche a fini d'utiliser la ressource partagée (c'est-à-dire que son socket se ferme), elle doit rendre son sémaphore pour qu'une autre tâche puisse accéder au réseau.
- Si une tâche donne par hasard un sémaphore qui a déjà atteint son nombre maximum, le nombre restera inchangé.

Compter les sémaphores (2/4)



Compter les sémaphores (3/4)

Examinons l'exemple étape par étape pour voir comment ce scénario se déroule :

1. Au départ, un sémaphore est créé avec un maximum (plafond) de 2 et un compte initial de 0.
2. Lorsque les tâches A et B tentent de prendre un *échantillon*, elles y parviennent immédiatement. À ce stade, elles peuvent chacune ouvrir un socket et communiquer sur le réseau.
3. La tâche C étant un peu plus tardive, elle devra attendre que le compte de *semphr* soit inférieur à 2, c'est-à-dire qu'un socket réseau soit libre d'être utilisé.
4. Lorsque la tâche B a fini de communiquer sur son socket, elle renvoie le sémaphore.
5. Maintenant qu'un sémaphore est disponible, la tâche C termine sa prise et est autorisée à accéder au réseau.
6. Peu après que la tâche C ait obtenu l'accès, la tâche B a un autre message à envoyer, elle tente donc de prendre un sémaphore, mais doit attendre qu'un sémaphore soit disponible, elle est donc mise en veille.
7. Pendant que la tâche C communique sur le réseau, la tâche A se termine et renvoie son sémaphore.
8. La tâche B est réveillée et sa prise est terminée, ce qui lui permet de commencer à communiquer sur le réseau.
9. Après que la tâche B a reçu son sémaphore, la tâche C termine sa transaction et rend le sémaphore qu'elle avait.

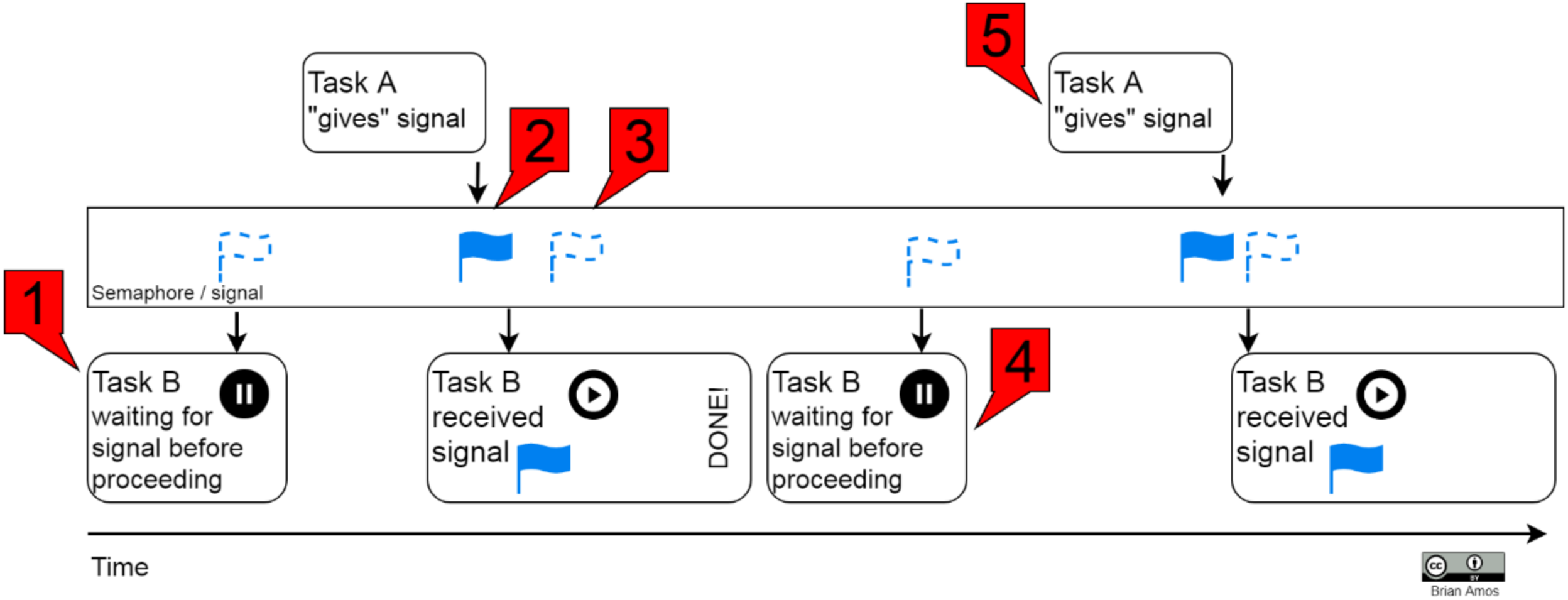
Compter les sémaphores (4/4)

- C'est dans l'attente des sémaphores qu'un RTOS se distingue de la plupart des autres implémentations de sémaphores.
  - une tâche peut perdre du temps pendant l'attente d'un sémaphore.
- Si une tâche ne parvient pas à acquérir le sémaphore à temps, elle ne doit pas accéder à la ressource partagée.
- Au lieu de cela, elle doit adopter une autre ligne de conduite.
- Cette alternative peut prendre la forme d'un certain nombre d'actions allant d'une défaillance si grave qu'elle déclenche une séquence d'arrêt d'urgence, à quelque chose de si bénin qu'elle est simplement mentionnée dans un fichier journal ou transmise à un port série de débogage en vue d'une analyse ultérieure.
- En tant que programmeur, c'est à vous de déterminer la marche à suivre, ce qui peut parfois donner lieu à des discussions difficiles avec d'autres disciplines.

Sémaphores binaires

- Les sémaphores binaires sont des sémaphores de comptage avec un nombre maximum de 1.
- Ils sont le plus souvent utilisés pour la synchronisation.
- Lorsqu'une tâche doit se synchroniser sur un événement, elle tente de prendre un sémaphore et se bloque jusqu'à ce que le sémaphore devienne disponible ou jusqu'à ce que le délai spécifié se soit écoulé.
- Une autre partie asynchrone du système (une tâche ou un ISR) donnera un sémaphore.
- Les sémaphores binaires peuvent être donnés plus d'une fois ; il n'est pas nécessaire que ce morceau de code les renvoie.
- Dans l'exemple suivant, la tâche A ne donne qu'un sémaphore, tandis que la tâche B ne prend qu'un sémaphore (diapositive suivante).

Sémaphore binaire (1/2)



Sémaphore binaire (2/2)

Examinons l'exemple étape par étape pour voir comment ce scénario se déroule :

1. Initialement, TaskB tente de prendre le sémaphore, mais il n'était pas disponible, et TaskB s'est donc endormi.
2. Un peu plus tard, la TaskA donne le signal.
3. La tâche B est réveillée (par le planificateur ; cela se passe en arrière-plan) et possède maintenant le sémaphore.
   - Il accomplira les tâches qui lui incombent jusqu'à ce qu'il soit terminé.
   - Notez cependant que TaskB n'a pas besoin de rendre le sémaphore binaire.
   - Au lieu de cela, il l'attend à nouveau.
4. La tâche B est à nouveau bloquée parce que le sémaphore n'est pas disponible (comme la première fois), elle se met donc en veille jusqu'à ce qu'un sémaphore soit disponible.
5. Le cycle se répète.

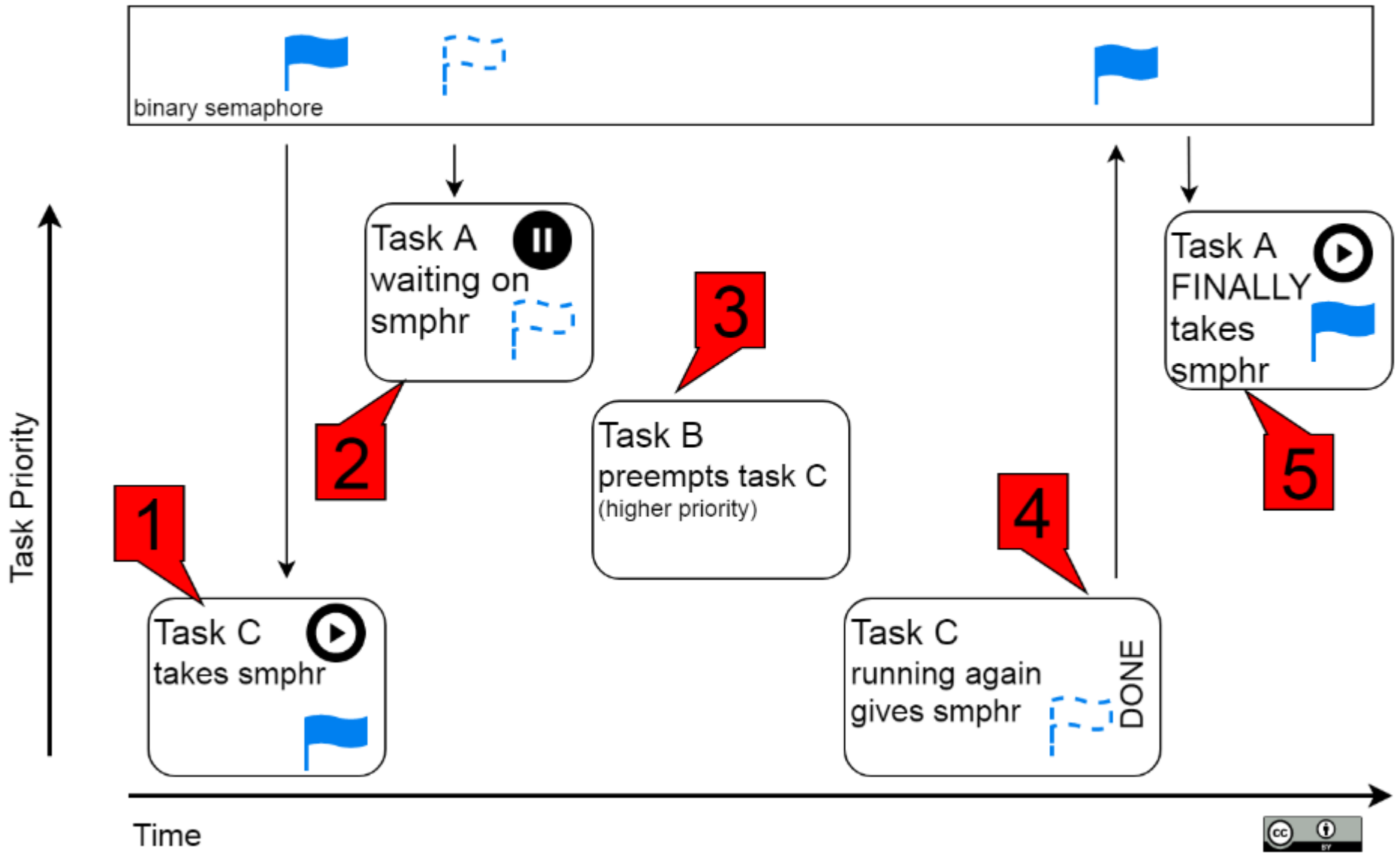
- Si la tâche B devait "rendre" le sémaphore binaire, elle s'exécuterait à nouveau immédiatement, sans recevoir le feu vert de la tâche A.
- Le résultat serait simplement une boucle tournant à plein régime, plutôt que d'être signalée sur une condition signalée par la tâche A.

Inversion de priorité

- Examinons un problème courant qui survient lorsque l'on tente d'utiliser un sémaphore binaire pour fournir une fonctionnalité d'exclusion mutuelle.
- Considérons trois tâches, A, B et C, où A a la priorité la plus élevée, B la priorité moyenne et C la priorité la plus basse.
- Les tâches A et C s'appuient sur un sémaphore pour accéder à une ressource partagée entre elles.
- La tâche A étant la plus prioritaire du système, elle doit toujours être exécutée avant les autres tâches.
- Cependant, étant donné que les tâches A et C dépendent toutes deux d'une ressource partagée entre elles (gardée par le sémaphore binaire), il existe une dépendance inattendue (diapositive suivante).

Binary Semaphore

Mutual Exclusion Attempt



Sémaphore binaire (exemple)

Examinons l'exemple étape par étape pour voir comment ce scénario se déroule :

1. La tâche C (la tâche la moins prioritaire du système) acquiert un sémaphore binaire et commence à travailler.
2. Avant que la tâche C ne termine son travail, la tâche A (la plus prioritaire) s'interrompt et tente d'acquérir le même sémaphore, mais elle est obligée d'attendre parce que la tâche C a déjà acquis le sémaphore.
3. La tâche B préempte également la tâche C, car la tâche B a une priorité plus élevée que la tâche C. La tâche B effectue le travail qui lui incombe, puis se met en veille.
4. La tâche C exécute le reste de son travail avec la ressource partagée, puis rend le sémaphore.
5. La tâche A peut enfin être exécutée.

Sémaphore binaire

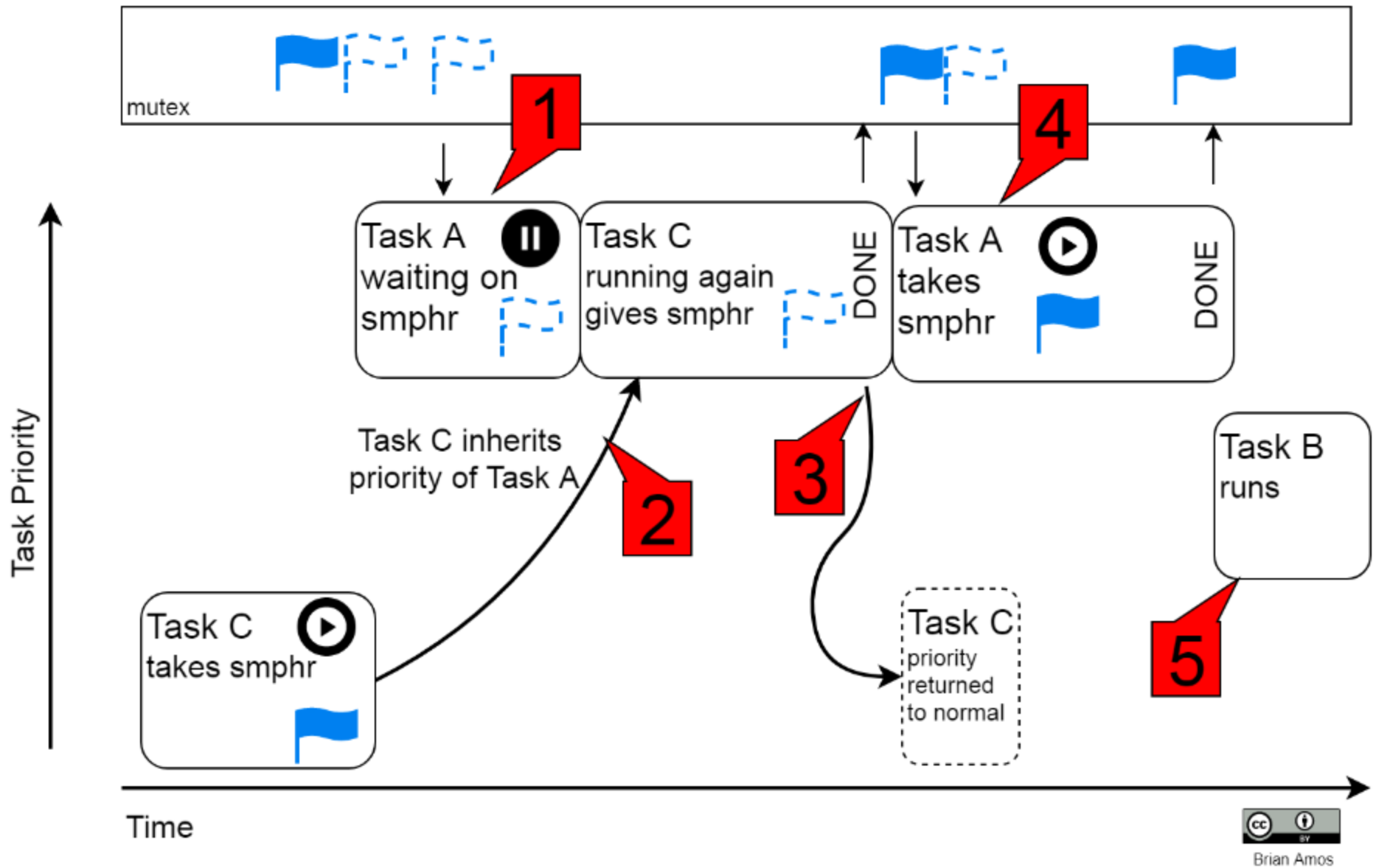
- La tâche A a finalement pu être exécutée, mais pas avant que DEUX tâches moins prioritaires aient été exécutées.
- La tâche C qui termine son travail avec la ressource partagée était inévitable (à moins qu'une modification de la conception ait pu être apportée pour l'empêcher d'accéder à la même ressource partagée que la tâche A).
- Cependant, la tâche B a également eu la possibilité de s'exécuter jusqu'à son terme, même si la tâche A était en attente et avait une priorité plus élevée !
- Il s'agit d'une inversion de priorité
  - une tâche plus prioritaire du système attend d'être exécutée, mais elle est forcée d'attendre alors qu'une tâche moins prioritaire s'exécute à sa place
  - les priorités des deux tâches sont effectivement inversées dans ce cas.

Les mutex minimisent l'inversion des priorités

- Plus tôt, nous avons dit que, sous FreeRTOS, les mutex étaient des sémaphores binaires avec une caractéristique supplémentaire importante.
- Cette caractéristique importante est l'héritage prioritaire
  - Les mutex ont la capacité de modifier temporairement la priorité d'une tâche afin d'éviter de provoquer des retards importants dans le système.
- Cela se produit lorsque le planificateur constate qu'une tâche de haute priorité tente d'acquérir un mutex déjà détenu par une tâche de moindre priorité.
- Dans ce cas précis, l'ordonnanceur augmentera temporairement la priorité de la tâche inférieure jusqu'à ce qu'il libère le mutex.
- À ce stade, la priorité de la tâche inférieure est ramenée à ce qu'elle était avant l'héritage des priorités.
- Examinons le même exemple que dans le diagramme précédent, mis en œuvre à l'aide d'un mutex (au lieu d'un sémaphore binaire) :

Mutex

Priority Inversion



Inversion de priorité des mutex (1/2)

Examinons l'exemple étape par étape pour voir comment ce scénario se déroule :

1. La tâche A attend toujours que la tâche C renvoie le mutex.
2. La priorité de la tâche C est ramenée au même niveau que celle de la tâche A, plus prioritaire. La tâche C s'exécute jusqu'à la fin puisqu'elle détient le mutex et est une tâche de haute priorité.
3. La tâche C rend le mutex et sa priorité est rétrogradée à ce qu'elle était avant de détenir un mutex qui retardait la tâche de haute priorité.
4. La tâche A prend le mutex et termine son travail.
5. La tâche B est autorisée à s'exécuter.

Inversion de priorité des mutex (2/2)

- En fonction du temps que prend la tâche C avec la ressource partagée et de l'importance du temps que prend la tâche A, il peut s'agir d'une source d'inquiétude majeure ou d'un problème mineur.
- Il est possible d'effectuer une analyse des délais pour s'assurer que la tâche A respecte toujours les délais, mais il peut s'avérer difficile d'identifier toutes les causes possibles d'inversion des priorités et d'autres événements asynchrones hautement prioritaires.
- Au minimum, l'utilisateur doit utiliser les délais intégrés prévus pour la prise de mutex et effectuer une action alternative appropriée si un mutex n'a pas été pris en temps voulu.
- Le chapitre 9, Communication inter-tâches, fournit plus de détails sur la manière de procéder.

- Les mutex et les sémaphores sont des mécanismes standard de signalisation entre les tâches.
- Ils sont très standardisés entre les différents RTOS et offrent une excellente flexibilité.

Exemple 1 queue (1/3)

```
int main(void)
{

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    xQueue = xQueueCreate(5,sizeof(int32_t));
    xTaskCreate(SenderTask,"sender task",100,NULL,1,&sender_handle);
    xTaskCreate(ReceiverTask,"receiver task",100,NULL,2,&receiver_handle);


    vTaskStartScheduler();

    while (1){ }
}
```

Exemple 1 queue (2/3)

```
void SenderTask (void *pvParameters)
{
    int32_t value_to_send = 3090;
    BaseType_t qStatus;
    while(1)
    {
        value_to_send++;


        qStatus = xQueueSend(xQueue,&value_to_send,0);
        if(qStatus !=pdPASS)
        {
                printf("Error: Data couldn't be sent\r\n");
        }
    }
    for (volatile int i=0;i<100000;i++);


}
```

Exemple 1 queue (3/3)

```
void ReceiverTask (void *pvParameters)
{
    int32_t value_received;
    const TickType_t wait_time = pdMS_TO_TICKS(100);
    BaseType_t qStatus;

    while(1)
    {


        qStatus = xQueueReceive(xQueue, &value_received, wait_time);
        if(qStatus==pdPASS)
        {
                printf("The value received is %ld \r\n",value_received);
        }
        else {printf("couldn't receive\r\n");}

    }
    for (volatile int i=0;i<100000;i++);
}
```

Exemple 2 queue (1/4)

```c
int main(void)
{

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    yQueue = xQueueCreate(5,sizeof(int32_t));

    xTaskCreate(SenderTask1,"sender task1",100,NULL,1,&sender_handle);
    xTaskCreate(SenderTask2,"sender task2",100,NULL,1,&sender_handle);
    xTaskCreate(ReceiverTask,"receiver task",100,NULL,2,&receiver_handle);

    vTaskStartScheduler();

    while (1){}

}
```

Exemple 2 queue (2/4)

```
void SenderTask1 (void *pvParameters)
{
	int32_t value_to_send= 2050;
	BaseType_t qStatus;
	while(1)
	{
		qStatus=xQueueSend(yQueue,&value_to_send,0);
		if(qStatus !=pdPASS)
		{
			printf("Error: Data couldn't be sent from sender 2\r\n");
		}
	}
    for (volatile int i=0;i<100000;i++);


}
```

Exemple 2 queue (3/4)

```
void SenderTask2 (void *pvParameters)
{
    int32_t value_to_send= 5050;
    BaseType_t qStatus;
    while(1)
    {
        qStatus=xQueueSend(yQueue,&value_to_send,0);
        if(qStatus !=pdPASS)
        {
        printf("Error: Data couldn't be sent from sender2\r\n");


        }
    }
    for (volatile int i=0;i<100000;i++);
}
```

Exemple 2 queue (4/4)

```
void ReceiverTask (void *pvParameters)
{

    int32_t value_received;
    const TickType_t wait_time = pdMS_TO_TICKS(100);
    BaseType_t qStatus;
    while(1)
    {
        qStatus = xQueueReceive(yQueue, &value_received, wait_time);
        if(qStatus == pdPASS)
        {
                printf("The value received is ID = %ld \r\n",value_received);
        }
        else {printf("couldn't receive\r\n");}


    }
    for (volatile int i=0;i<100000;i++);
}
```

Exemple 3: trois tâches sans synchronisation (1/4)

```c
int main(void)
{
    HAL_Init();

    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    xTaskCreate(redLedController,"red led task",100,NULL,1,NULL);
    xTaskCreate(yellowLedController,"yellow led task",100,NULL,1,NULL);
    xTaskCreate(blueLedController,"blue led task",100,NULL,1,NULL);

    vTaskStartScheduler();

    while (1){}

}
```

Exemple 3: trois tâches sans synchronisation (1/4)

```c
int main(void)
{
    HAL_Init();

    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    xTaskCreate(redLedController,"red led task",100,NULL,1,NULL);
    xTaskCreate(yellowLedController,"yellow led task",100,NULL,1,NULL);
    xTaskCreate(blueLedController,"blue led task",100,NULL,1,NULL);

    vTaskStartScheduler();

    while (1){}

}
```

Exemple 3: trois tâches sans synchronisation (2/4)

```
void redLedController(void *pvParameters)
{


    while(1)
    {
        printf("this is red task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_3);
    }

}
```

Exemple 3: trois tâches sans synchronisation (3/4)

```
void yellowLedController(void *pvParameters)
{

    while(1)
    {
        printf("this is yellow task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_5);
    }


}
```

Exemple 3: trois tâches sans synchronisation (4/4)

```
void blueLedController(void *pvParameters)
{


    while(1)
    {
        printf("this is blue task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_4);
    }


}
```

Exemple 4: Sémaphore binaire (1/4)

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();
    xBinarySemaphore=xSemaphoreCreateBinary();

    xTaskCreate(redLedController,"red led task",100,NULL,1,NULL);
    xTaskCreate(yellowLedController,"yellow led task",100,NULL,1,NULL);
    xTaskCreate(blueLedController,"blue led task",100,NULL,1,NULL);

    vTaskStartScheduler();

    while (1){}

}
```

Exemple 4: Sémaphore binaire (2/4)

```
void redLedController(void *pvParameters)
{
    xSemaphoreGive(xBinarySemaphore);
    while(1)
    {
        xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);


        printf("this is red task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_3);
        xSemaphoreGive(xBinarySemaphore);
        //vTaskDelay(100);
    }

}
```

Exemple 4: Sémaphore binaire (3/4)

```
void yellowLedController(void *pvParameters)
{
    while(1)
    {
        xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);

        printf("this is yellow task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_5);
        xSemaphoreGive(xBinarySemaphore);
        //vTaskDelay(100);
    }
}
```

Exemple 4: Sémaphore binaire (4/4)

```
void blueLedController(void *pvParameters)
{
    while(1)
    {
        xSemaphoreTake(xBinarySemaphore,portMAX_DELAY);

        printf("this is blue task\r\n");
        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_4);
        xSemaphoreGive(xBinarySemaphore);
        //vTaskDelay(100);
    }
}
```

Résumé

- Ce chapitre a présenté les files d'attente, les sémaphores et les mutex.
- Quelques cas d'utilisation courante de chacun de ces éléments de base pour les applications RTOS ont également été discutés à un niveau élevé et certains des comportements plus subtils de chacun ont été mis en évidence.
- Les diagrammes présentés dans ce chapitre devraient servir de point de référence auquel nous pourrons revenir lorsque nous passerons à des exemples réels plus complexes dans les chapitres suivants.
- Nous avons maintenant abordé certains des concepts fondamentaux des RTOS.
- Dans le prochain chapitre, nous nous intéresserons à une autre étape très importante du développement d'un système temps réel solide.
- Cette étape influe sur l'efficacité de l'exécution des microprogrammes et a un impact majeur sur les performances du système - sélection du MCU.