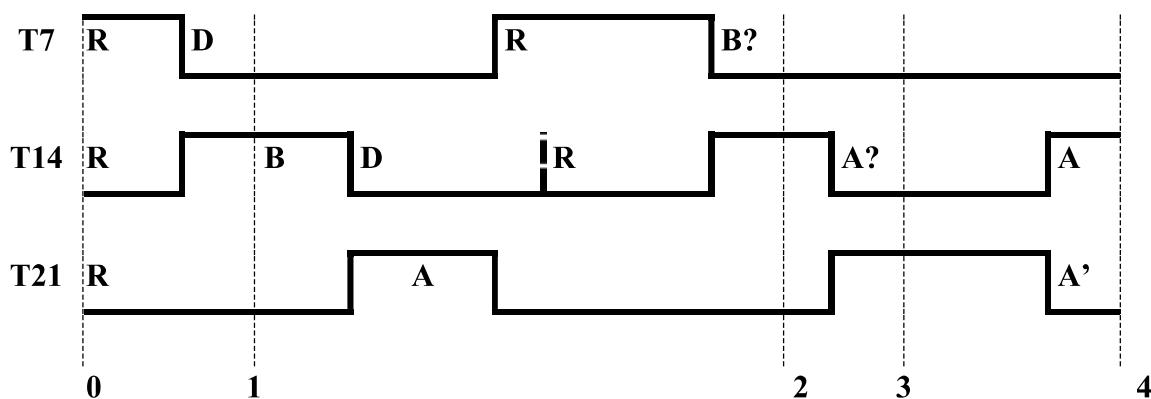


## Question 2 (4 points) Ordonnancement et héritage de priorité

Soit un système multitâche fonctionnant sous µCOS-III et comprenant 3 tâches et 2 mutex. Les tâches ont les priorités 7, 14 et 21 et les mutex s'appellent A et B. Le protocole héritage de priorité est utilisé.

La figure 2.1 illustre une trace d'exécution de ce système. La lettre R signifie que la tâche vient de passer à l'état prêt, D qu'elle vient de demander un délai (*OSTimeDly*), A qu'elle a demandé et obtenu le mutex A, A? qu'elle a demandé et n'a pas obtenu le mutex A, A' qu'elle a rendu le mutex A. Il en est de même pour le mutex B.



**Figure 2.1** N.B. On assume que B est demandé au tout début du tick 1 par T14

a) (2 pts) Pour chacun des instants 1, 2 et 3 et 4 :

- Remplissez la table *OSPrioTbl* (page 4 et 5) pour les tâches actives (i.e. à 1)
- Si c'est le cas, indiquez qui possède le mutex A, s'il y a héritage de priorité et qui est dans la liste d'attente.
- Si c'est le cas, indiquez qui possède le mutex B, s'il y a héritage de priorité et qui est dans la liste d'attente.
- Si c'est le cas indiquez la liste d'attente de *OSTickListDly*.

b) (2 pts) Donnez le détail des évènements :

- i. entre l'instant 0 et 1
- ii. entre l'instant 1 et 2
- iii. entre l'instant 2 et 3
- iv. entre l'instant 3 et 4

Plus précisément, donnez les détails internes de ce qui se passe dans le noyau lorsque que les fonctions *pend*, *post* et *OSTimeDly* sont appelées c'est-à-dire décrire ce qui se passe au niveau des structures de données (en faisant le lien avec votre réponse en a) et des appels système. S'il y a lieu, les changements de contexte (à la suite de l'appel de l'ordonnanceur), en indiquant la prochaine tâche à être exécutée (ici retour de la tâche courante ou une nouvelle tâche).

## Instant1 :

Liste d'attente de mutex A = {}

Liste d'attente de mutex B = {}

Liste d'attente d'un délai (OSTimeDly) = T7

Possession de A : Possession de B : T14

OSPrioTbl[] :

	0	1	2	3	4	5	6	7
[0]								
[1]							1	
[2]						1		
[3]								
[4]								
[5]								
[6]								
[7]							1	

## Instant2 :

Liste d'attente de mutex A = {}

Héritage : T14 a hérité de T7

Liste d'attente de mutex B = { T7 }

Liste d'attente d'un délai (OSTimeDly) = {}

Possession de A : T21

Possession de B : T14

OSPrioTbl[] :

	0	1	2	3	4	5	6	7
[0]								1
[1]								
[2]						1		
[3]								
[4]								
[5]								
[6]								
[7]							1	

## Instant3 :

Liste d'attente de mutex A = { *T14* }

Liste d'attente de mutex B = { *T7* }

Liste d'attente d'un délai (OSTimeDly) = { }

Possession de A : *T21*      Possession de B : *T14*

OSPrioTbl[] :

	0	1	2	3	4	5	6	7
[0]								1
[1]								
[2]								
[3]								
[4]								
[5]								
[6]								
[7]								1

*Héritage : T21 hérite de T14 donc de la priorité 7 car T14 est devenu temporairement T7 donc T21 devient aussi temporairement T7*

**Instant4 :**

Liste d'attente de mutex A = { }

Liste d'attente de mutex B = { T7 }

Liste d'attente d'un délai (OSTimeDly) = { }

Possession de A : T14      Possession de B : T14

OSPrioTbl[] :

	0	1	2	3	4	5	6	7
[0]								1
[1]								
[2]						1		
[3]								
[4]								
[5]								
[6]								
[7]							1	

Héritage : T21 redevient T21 donc priorité de 21, T14 est toujours T7 donc priorité de 7.

*T.B.*

*4/4*

Q2 i) - Au début toutes les tâches sont prêtes alors leur bit de priorité dans la table est mis à 1 avec OS\_PrioInsert(). Elles passent donc à l'état Ready.

- L'ordonnanceur est appelé et détermine la tâche avec la plus haute priorité avec OS\_PrioGetHighest(). C'est ainsi que T7 passe de Ready à Running.

- T7 fait ensuite un OSTimePly() ce qui fait en sorte qu'elle est ajoutée à la liste d'attente des délais. Elle est aussi retirée de la table Ready avec OS\_PrioRemove() comme elle passe de Ready à Pending avec le délai.

- L'ordonnanceur est appelé et détermine que T14 est maintenant la tâche prête la plus prioritaire.
- T14 passe donc à running. Le changement de contexte est fait.
- T14 fait un OSMutexPend() pour demander le mutex B et elle le reçoit comme il est disponible.

ii) - T14 fait un OSTimePly(). Même principe que T7 plus haut, elle passe de running à pending. Son bit n'est plus à 1 dans la table.

- L'ordonnanceur détermine que c'est maintenant T21 qui est la tâche prête la plus prioritaire.

- T21 passe de ready à running

- T21 demande le mutex A et l'obtient comme il est disponible.

- Le délai de T7 est échu alors elle est retirée de la file de délai. Elle est rajoutée à la table des tâches prêtes avec OS\_PrioInsert(). Elle passe donc de pending à ready.

- L'ordonnanceur est appelé (même principe).

T21 passe donc de running à ready

T7 passe de ready à running.

Le changement de contexte est effectué avec OSCtxSw().

- T7 demande le mutex B (OSMutexPend()) comme il n'est pas disponible, elle est ajoutée à la file des mutex avec OSPendListInsertPrior(). T7 passe de running à pending.

L'ordonnanceur est appelé (même principe)

T14 passe de ready à running

Le changement de contexte est effectué

- T14 a fini son délai. Elle quitte la liste de délai. Elle est ajoutée à la table des tâches prêtes. T14 passe de pending à ready.  
T7 est toujours la tâche à exécuter...

T14 hérite de la priorité 7 comme il détient le mutex B.

iii) - T14 demande le mutex A (même principe):  
T14 passe de running à pending

- T21 hérite de la priorité 7 comme il détient le mutex A.

- L'ordonnanceur est appelé. T21 passe de ready à running.

vi) - T21 libère le mutex A. Le prochain dans la liste du mutex A est T14 (OS\_PendListRemove) qui passe donc de pending à ready et son bit est mis à 1 dans la table.

- L'ordonnanceur est appelé (même principe):

T21 passe de running à ready

T14 passe de ready à running

Le changement de contexte est effectué.

### Question 3 (3.5 points) Analyse de code avec uC/OS-III

Soit 1 tâche producteur et N tâches consommateurs. Le code des tâches illustré à la figure 3.1 offre la fonctionnalité suivante:

- 1) Quand la variable globale *Producer\_done* est à faux, le consommateur consomme à travers une file circulaire de 256 éléments gérés par 2 sémaphores (*SemFull* initialisée à 0 et *SemEmpty* initialisée à 256) et un mutex *Mutex*, et
- 2) quand la variable globale *Producer\_done* est à vrai, le producteur s'arrête et les consommateurs doivent vider le fifo et se détruire.

```

1 void TaskCons(void* pdata)
2 {
3     OS_ERR err;
4     CPU_TS ts;
5
6     int out;
7
8     pdata = pdata;
9     safeprintf("Cons no %d: demarre\n", OSPrioCur);
10    while (1) {
11
12        OSMutexPend(&Mutex_done, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
13        if (Producer_done) {
14            OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
15            OSSemPend(&SemFull, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err);
16            if (err == OS_ERR_PEND_WOULD_BLOCK) {
17
18                OSTaskDel((OS_TCB*)0, &err);
19
20            }
21            else {
22                OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
23                out = buffer[rem++];
24                if (rem >= N)
25                    rem = 0;
26                OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
27                OSSemPost(&SemEmpty, OS_OPT_POST_1, &err);
28                OSTimeDly(100, OS_OPT_TIME_DLY, &err);
29            }
30        }
31        else {
32            OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
33            OSSemPend(&SemFull, 100, OS_OPT_PEND_BLOCKING, &ts, &err);
34
35            if (err != OS_ERR_TIMEOUT) {
36                OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
37                out = buffer[rem++];
38                if (rem >= N)
39                    rem = 0;
40                OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
41                OSSemPost(&SemEmpty, OS_OPT_POST_1, &err);
42                OSTimeDly(100, OS_OPT_TIME_DLY, &err);
43            }
44        }
45    }
46 }
47 }
```

Figure 3.1

### Question 3 (3.5 points) Analyse de code avec uC/OS-III

Soit 1 tâche producteur et N tâches consommateurs. Le code des tâches illustré à la figure 3.1 offre la fonctionnalité suivante:

- 1) Quand la variable globale *Producer\_done* est à faux, le consommateur consomme à travers une file circulaire de 256 éléments gérés par 2 sémaphores (*SemFull* initialisée à 0 et *SemEmpty* initialisée à 256) et un mutex *Mutex*, et
- 2) quand la variable globale *Producer\_done* est à vrai, le producteur s'arrête et les consommateurs doivent vider le fifo et se détruire.

```

1 void TaskCons(void* pdata)
2 {
3     OS_ERR err;
4     CPU_TS ts;
5
6     int out;
7
8     pdata = pdata;
9     safeprintf("Cons no %d: démarre\n", OSPrioCur);
10    while (1) {
11
12        OSMutexPend(&Mutex_done, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
13        if (Producer_done) { // Quand producteur a terminé.
14            OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
15            OSSemPend(&SemFull, 0, OS_OPT_PEND_NON_BLOCKING, &ts, &err);
16            if (err == OS_ERR_PEND_WOULD_BLOCK) {
17                // Si file vide c'est que tout
18                OSTaskDel((OS_TCB*)0, &err); ← On fait des appels
19                ← non bloquants.
20            }
21            else { // sinon on fonctionne comme d'habitude mais
22                OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
23                out = buffer[rem++];
24                if (rem >= N)
25                    rem = 0;
26                OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
27                OSSemPost(&SemEmpty, OS_OPT_POST_1, &err);
28                OSTimeDly(100, OS_OPT_TIME_DLY, &err);
29            }
30        }
31        else {
32            OSMutexPost(&Mutex_done, OS_OPT_POST_NONE, &err);
33            OSSemPend(&SemFull, 100, OS_OPT_PEND_BLOCKING, &ts, &err); // comme on veut
34
35            if (err != OS_ERR_TIMEOUT) { // si pas de timeout
36                OSMutexPend(&Mutex, 0, OS_OPT_PEND_BLOCKING, &ts, &err);
37                out = buffer[rem++]; // on y va comme
38                if (rem >= N)
39                    rem = 0;
40                OSMutexPost(&Mutex, OS_OPT_POST_NONE, &err);
41                OSSemPost(&SemEmpty, OS_OPT_POST_1, &err);
42                OSTimeDly(100, OS_OPT_TIME_DLY, &err); ← pas bloqué
43            }
44        }
45    }
46
47 }
48
49
50 }
```

*← On fait des appels non bloquants.*

*← comme on veut pas bloqué on utilise un deadline de 100 ticks.*

*← comme on y va comme d'habitude*

*← else // si timeout on va tester le producer\_done à la prochaine itération*

Figure 3.1

- a) **(2 pts)** Expliquez clairement comment cette fonctionnalité est réalisée dans le code de la figure 3.1. Suggestions expliquez les différentes branches de if then else et utilisez les numéros de lignes.

À inclure...

- b) **(1.5 pts)** Le code de la figure a été réalisé sous uC/OS-II, mais avec la venue de OS/III une nouvelle option nommée ***OS\_OPT\_DEL\_ALWAYS*** est apparue lors de la destruction d'un sémaphore. Plus précisément, en faisant *OSDel(&SemFull, OS\_OPT\_DEL\_ALWAYS, &err)* uC/OS-III procède à la suppression du sémaphore, que les tâches soient en attente sur le sémaphore ou non. S'il y a des tâches en attente sur le sémaphore, ces tâches deviennent donc prêtes à être exécutées. Ces dernières pourront en être informées par le biais d'un code d'erreur approprié lors de leur retour de l'appel à *OSSemPend(&SemFull, , , &err)*, qui retournera alors *err = OS\_ERR\_OBJ\_DEL*, indiquant que le sémaphore a été détruit.

Avec cette fonctionnalité, proposez une nouvelle version de *TaskCons* qui éviterait de mettre un timeout dans *OSSemPend()*. Expliquez les changements que vous feriez, pas besoin de recoder...

*On fait simplement un appel à OSSemPend(&SemFull, 0, ...) similaire à celui de la ligne 33 mais un deadline infini (0) plutôt que 100. À la ligne 35 on fait le test sur le message d'erreur OS\_ERR\_OBJ\_DEL plutôt que OS\_ERR\_TIME\_OUT. En fait si on ajoute un else (au if de la ligne 35) avec OSTaskDel(OS\_TCB\*) 0, &err, on pourrait éliminer les 12 à 30.*

#### Question 4 (2.5 points) Ordonnancement, RMA et contraintes dures vs douces

Considérant l'ordonnancement RMA (*Rate Monotonic Assignment*) (plus la période est petite plus la priorité est grande) :

- a) (1 pt) À l'aide de la condition d'ordonnancement de *Liu et Layland*, les tâches de la Table 4.1 peuvent-elles être ordonnancées.

Tâches	Période $P_i$	Pire temps d'exécution $C_i$
Task1	100	20
Task2	150	30
Task3	210	70
Task4	400	100

**Table 4.1**

$20/100 + 30/150 + 70/210 + 100/400 \approx .938 < .757$  on ne peut donc rien dire

- b) (.5 pt) Sachant d'après une analyse plus poussée que le vrai temps d'exécution  $R_i$  (temps CPU + temps d'interférence + temps de blocage + temps de changement de contexte) de ces 4 tâches est donné par  $R_1 = 20$ ,  $R_2 = 50$ ,  $R_3 = 140$  et  $R_4 = 410$ , les tâches de la Table 4.1 peuvent-elles être ordonnancées selon l'assignation  $T_i < T_j \rightarrow P_i < P_j$  où  $T_i$  est la période et  $P_i$  la priorité? Expliquez.

*Non car il faut toujours avoir  $R_i < P_i$  sinon on va manquer un deadline. Or ici,  $R4 = 410 > T4 = 400$ .*

- c) Supposons que les tâches Task1, Task2 et Task4 possèdent chacune une contrainte dure qui doit toujours être respectée, alors que la tâche Task3 possède une contrainte plus douce qui doit être respectée en moyenne. On décide de remplacer la tâche Task3 par deux tâches Task3' et Task3'', ayant chacune une période deux fois plus longue que Task3. De plus, on ajoute au tableau 4.1 une colonne qui donne pour le temps moyen d'exécution. Le résultat de ces 2 modifications est illustré à la figure 4.2:

- c.1) (.5 pt) À l'aide de la condition d'ordonnancement de *Liu et Layland*, démontrez comment on peut ainsi assurer un ordonnancement de Task1, Task2, Task4 qui respecte chaque contrainte dure.

*En séparant Task3 en 2 tâches Task3' et Task3'' on augmente la période de 210 à 420. Si on fait 2 tâches (Task3' et Task3'') à tous les 420 ticks de 70 ticks ça revient à une à toute les 210 de 70 ticks... Mais le fait d'avoir fait passer à 420 change les priorités selon RMA : Task1 est la petite période donc plus grande priorité, ensuite Task2, puis Task 4 et finalement Task3' et Task3''. Concentrons-nous sur les 3 premières tâches:*

$20/100 + 30/50 + 100/400 = .65 < .78$  donc OK! On peut implémenter les 3 premières tâches et on respectera toujours les contraintes (dure).

- b.2) (.5 pt) Ceci est illustré à la figure 4.2. Que peut-on dire alors de l'ordonnancement de Task1, Task2, Task3', Task3'' et Task4 d'un point de vue contrainte douce?

*Ici on va regarder selon des contraintes douces mais avec les 5 tâches :*

*10/100 + 25/100 + 20/400 + 40/420 + 40/420 ≈ .51 < .743 donc OK! On peut implémenter les 5 tâches et on respectera en moyenne les contraintes douces.*

Tâches	Période $P_i$	Pire temps d'exécution $C_i$	Temps d'exécution en moyenne i.e. $C_i$ moyen
Task1	100	20	10
Task2	150	30	25
Task3'	420	70	40
Task3''	420	70	40
Task4	400	100	20

**Question 5 (4 points) Laboratoire no 1**

Soit un système sous uC/OS-III faisant le routage des paquets fonctionnant avec une fréquence de 1000 Hz (`OS_CFG_TICK_RATE_HZ = 1000u`).

Il est composé des éléments suivants : une interface de réception qui est implémentée avec un ISR (`ISR_input_packet`), une tâche qui fait le traitement sur les paquets reçus (`TaskComputing`) et une tâche qui assigne les paquets au bon port de sortie (`TaskForwarding`).

L'acquisition des paquets se fait de manière discontinue par rafale: lors d'une rafale qui dure **500 ms** à 1 débit de 2 paquets/ms. Les paquets rentrent via une série d'interruptions en provenance d'un périphérique extérieur en suivant la cadence de 2 paquets/ms. Puis durant **500 ms** aucun paquet ne rentre au routeur. Puis on recommence ce traitement ainsi à chaque **1000 ms**. Durant la rafale, pour chaque nouveau paquet qui arrive au routeur via le ISR, `ISR_input_packet` prend **1/5 de tick** d'horloge et transfert les paquets dans une queue de messages `inputQ` pour un traitement éventuel.

Sur le CPU, la tâche `TaskComputing` traite les paquets un par un : elle en prend un dans `inputQ`, effectue un calcul (validation de l'adresse et rejet de paquets, CRC, etc.) et le place dans la bonne queue de messages d'une des 3 tâches `TaskForwarding` selon la priorité `highQ`, `mediumQ` et `lowQ`. Tout ce traitement demande à `TaskComputing` **1/2 tick** d'horloge par paquet.

Finalement, les `TaskForwarding` vident les queues de messages selon la priorité `highQ`, `mediumQ` et `lowQ` et envoie le paquet à la bonne adresse de destination (on suppose donc ici qu'il fait aussi le travail de `TaskOutputPort()`). Tout ce traitement demande à chaque `TaskForwarding` **1/8 tick** d'horloge par paquet.

- a) **(2.5 pts)** Calculez le pourcentage d'utilisation du CPU pour `ISR_input_packet`, `TaskComputing` et `TaskForwarding`. Reste-t-il suffisamment de place pour les changements de contextes et les autres ISR? Justifiez.
  
- b) **(1.5 pt)** Déterminez selon vous qu'elle devrait être la taille minimum des queues de messages pour assurer le bon fonctionnement du système. On suppose que la priorité de `TaskComputing` est plus grande que celle de `TaskForwarding`. De plus, le poids de distribution des queues `highQ`, `mediumQ` et `lowQ` est distribué uniformément : en moyenne de 1/3, 1/3 et 1/3.

1 tick = 1ms.

C  
1° 500 ms → 2 paquets/ms → 1000 paquet  
500 ms suivant → rien.

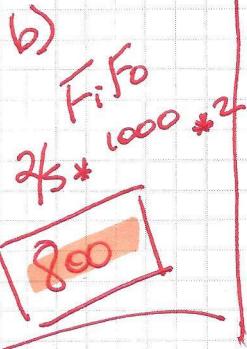
ISR demande  $\frac{1}{s}$  ticks →  $\frac{1}{s}$  ms.

- Pour le 1000 paquets de la rafale

ISRs demandent 200 ms  
 $\frac{1000}{5}$  ← du 500 ms.

Il reste 300 ms.

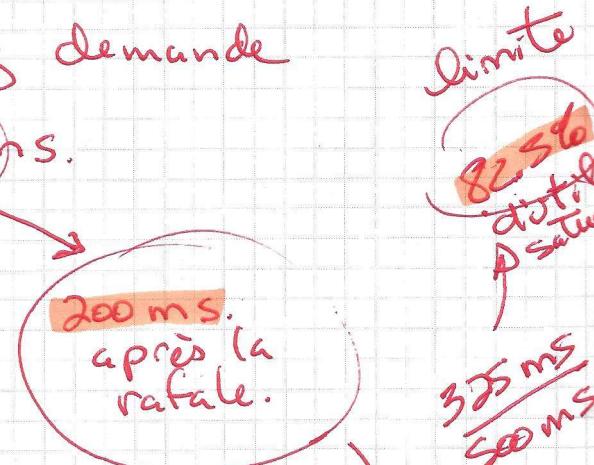
- Pour le 1000 paquets de la 2e rafale



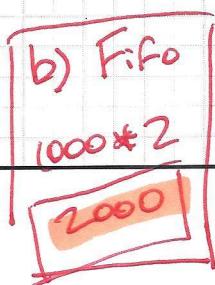
Task Computing demande

$$\left( \frac{1000}{2} \right) = 500 \text{ ms.}$$

300 ms  
durant la rafale



- Pour le 1000 paquets de la rafale



Task Forwarding demande

$$\frac{1000}{8} = 125 \text{ ms} \rightarrow$$

$\approx 600$   
charge

125 ms  
après la rafale

## Annexe

```
void OSMutexPend (OS_MUTEX *p_mutex,
                  OS_TICK timeout,
                  OS_OPT opt,
                  CPU_TS *p_ts,
                  OS_ERR *p_err)
```

**opt**

determines whether the user wants to block if the mutex is not available or not. This argument must be set to either:  
`OS_OPT_PEND_BLOCKING`, or  
`OS_OPT_PEND_NON_BLOCKING`

```
void OSMutexPost (OS_MUTEX *p_mutex,
                  OS_OPT opt,
                  OS_ERR *p_err);
```

**opt**

`OS_OPT_POST_NONE`  
 No special option selected.  
`OS_OPT_POST_NO_SCHED`  
 Do not call the scheduler after the post, therefore the caller is resumed even if the mutex was posted and tasks of higher priority are waiting for the mutex.  
 Use this option if the task calling OSMutexPost() will be doing additional posts, if the user does not want to reschedule until all is complete, and multiple posts should take effect simultaneously.

```
void OSSemPend (OS_SEM *p_sem,
                 OS_TICK timeout,
                 OS_OPT opt,
                 CPU_TS *p_ts,
                 OS_ERR *p_err)
```

**timeout**

allows the task to resume execution if a semaphore is not posted within the specified number of clock ticks. A timeout value of 0 indicates that the task waits forever for the semaphore. The timeout value is not synchronized with the clock tick. The timeout count begins decrementing on the next clock tick, which could potentially occur immediately.

**opt**

specifies whether the call is to block if the semaphore is not available, or not block.  
`OS_OPT_PEND_BLOCKING`  
 to block the caller until the semaphore is available or a timeout occurs.  
`OS_OPT_PEND_NON_BLOCKING`  
 If the semaphore is not available, OSSemPend() will not block but return to the caller with an appropriate error code.

**p\_err**

is a pointer to a variable used to hold an error code:

`OS_ERR_NONE` if the semaphore is available.  
`OS_ERR_PEND_WOULD_BLOCK` if this function is called as specified `OS_OPT_PEND_NON_BLOCKING`, and the semaphore was not available.  
`OS_ERR_TIMEOUT` if the semaphore is not signaled within the specified timeout.

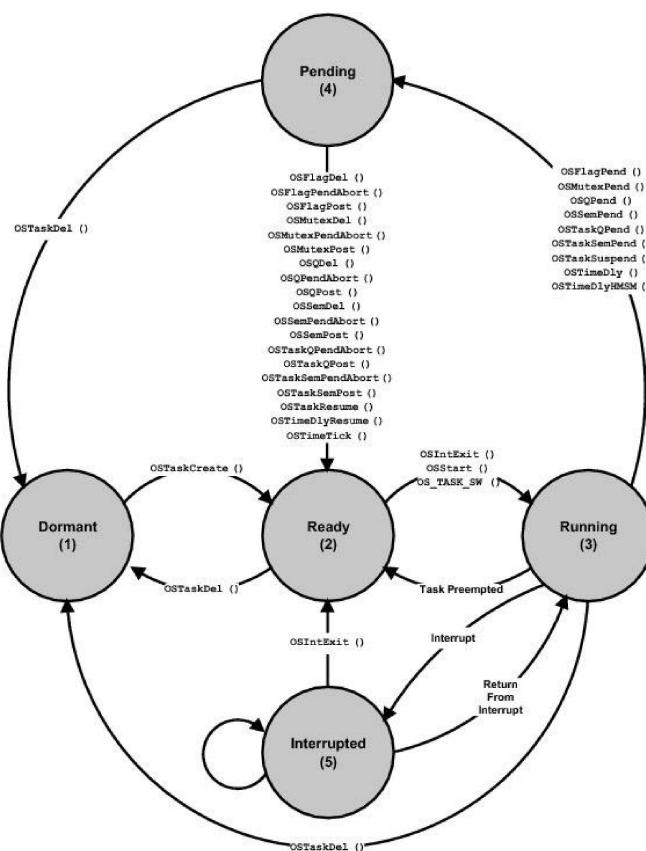
```
OS_SEM_CTR OSSemPost (OS_SEM *p_sem,
    OS_OPT opt,
    OS_ERR *p_err)
```

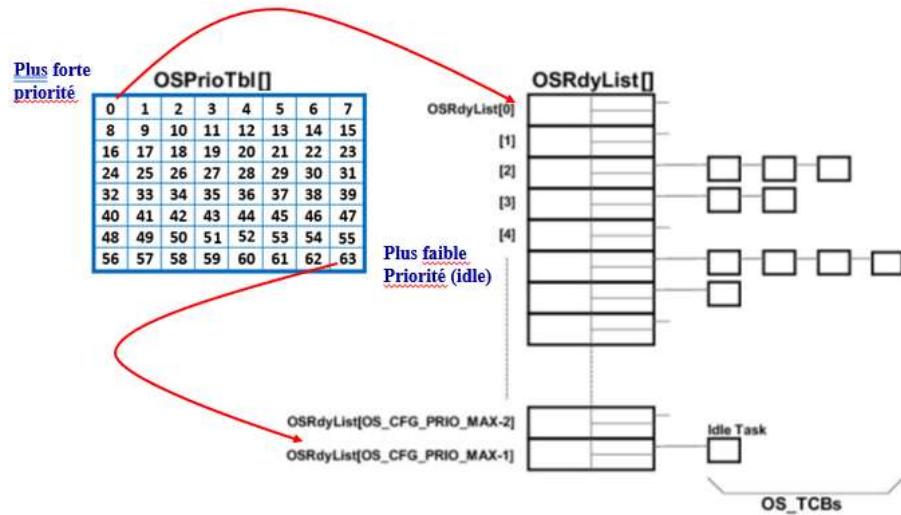
**opt**  
determines the type of post performed.

**OS\_OPT\_POST\_1**  
Post and ready only the highest-priority task waiting on the semaphore.

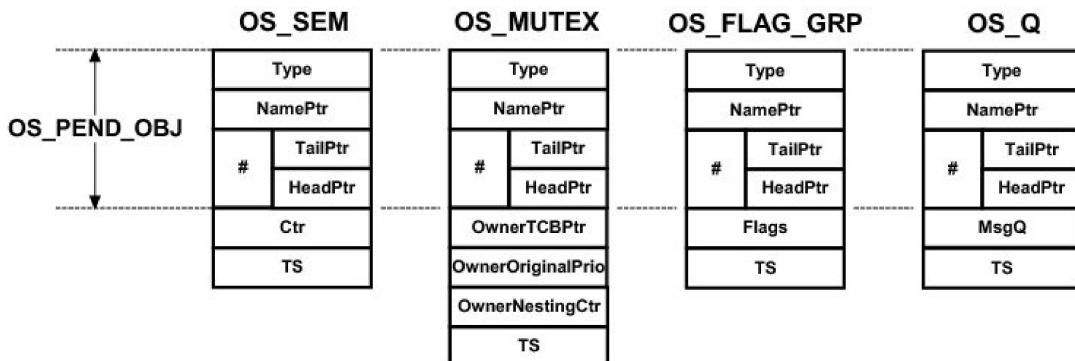
**OS\_OPT\_POST\_ALL**  
Post to all tasks waiting on the semaphore. You should only use this option if the semaphore is used as a signaling mechanism and never when the semaphore is used to guard a shared resource. It does not make sense to tell all tasks that are sharing a resource that they can all access the resource.

**OS\_OPT\_POST\_NO\_SCHED**  
This option indicates that the caller does not want the scheduler to be called after the post. This option can be used in combination with one of the two previous options.  
You should use this option if the task (or ISR) calling OSSemPost() will be doing additional posting and, the user does not want to reschedule until all done, and multiple posts are to take effect simultaneously.





Function	Description
<code>OS_PrioGetHighest()</code>	Find the highest priority level
<code>OS_PrioInsert()</code>	Set bit corresponding to priority level in the bitmap table
<code>OS_PrioRemove()</code>	Clear bit corresponding to priority level in the bitmap table



Function	Description
<code>OS_PendListChangePrio()</code>	Change the priority of a task in a pend list
<code>OS_PendListInit()</code>	Initialize a pend list
<code>OS_PendListInsertPrio()</code>	Insert a task in priority order in the pend list
<code>OS_PendListRemove()</code>	Remove a task from a pend list

## Ordonnancement:

$N$  = Nombre de tâches  $i$  dans le système ( $i$  varie donc de 1 à  $N$ )

$P_i$  = Priorité d'une tâche  $i$  (0 étant la plus prioritaire et 63 la moins prioritaire)

$T_i$  = Période d'exécution d'une tâche  $i$  (ex. 1 tâche doit s'exécuter à toutes les 4 ticks d'horloges)

$I_i$  = Temps maximum pour lequel une tâche  $i$  est préemptée (ou interférée) par une (ou des) tâche(s) de plus grande priorité durant  $T_i$ .

$B_i$  = Temps maximum de blocage d'une tâche  $i$  durant sa période  $T_i$  (e.g., tâche qui ne peut plus s'exécuter à cause qu'une (ou des) tâche(s) de moins grande priorité occupe une section critique requise par  $T_i$  ou par une tâche de priorité plus grande que  $T_i$ )

$C_i$  = Temps d'exécution maximum sur le CPU de la tâche  $i$

$U = \text{Utilisation du CPU par les différentes tâches } T_i \text{ d'un système} = \sum_{i=1}^N C_i / T_i$

$CS$  = Le temps maximum de changement de contexte d'une tâche  $i$

$R_i$  = Temps d'exécution réel d'une tâche  $i$  durant sa période  $T_i$

$$R_i = C_i + I_i + B_i + CS$$

Important: pour démontrer qu'une assignation est valide on devra avoir  $R_i \leq T_i$  pour toutes les tâches  $i$  du système.

<ul style="list-style-type: none"> <li>• RMS           <ul style="list-style-type: none"> <li>– Rate Monotonic scheduling</li> <li>– On assigne la priorité aux processus selon leur période respective.</li> <li>– Plus la période est petite, plus la tâche est prioritaire (petit <math>P_i</math>):</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Test d'ordonnancement           <ul style="list-style-type: none"> <li>– Liu and Layland ont démontré que si la condition suivante est rencontrée, on aura toujours un résultat d'ordonnancement:</li> </ul> </li> </ul>																																
<ul style="list-style-type: none"> <li>• Exemple:</li> </ul> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Processus</th><th>Période (<math>T_i</math>)</th><th>Priorité (<math>P_i</math>)</th></tr> </thead> <tbody> <tr><td>A</td><td>25</td><td>1</td></tr> <tr><td>B</td><td>60</td><td>3</td></tr> <tr><td>C</td><td>42</td><td>2</td></tr> <tr><td>D</td><td>105</td><td>5</td></tr> <tr><td>E</td><td>75</td><td>4</td></tr> </tbody> </table>	Processus	Période ( $T_i$ )	Priorité ( $P_i$ )	A	25	1	B	60	3	C	42	2	D	105	5	E	75	4	<p style="text-align: right;"><b>Référence p. 81 du User's Manual disponible sur le site web</b></p> $\sum_{i=1}^N \left( \frac{C_i}{T_i} \right) < N(2^{1/N} - 1)$ <p>Quand <math>N \rightarrow \infty</math> le terme de droite tend vers <math>\ln(2) = 69.3\%</math></p> <p style="text-align: right;"><a href="https://dl.acm.org/citation.cfm?doi=321738.321743">https://dl.acm.org/citation.cfm?doi=321738.321743</a></p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>N</th><th>Utilization Bound</th></tr> </thead> <tbody> <tr><td>1</td><td>100.0%</td></tr> <tr><td>2</td><td>82.8%</td></tr> <tr><td>3</td><td>78.0%</td></tr> <tr><td>4</td><td>75.7%</td></tr> <tr><td>5</td><td>74.3%</td></tr> <tr><td>10</td><td>71.8%</td></tr> </tbody> </table>	N	Utilization Bound	1	100.0%	2	82.8%	3	78.0%	4	75.7%	5	74.3%	10	71.8%
Processus	Période ( $T_i$ )	Priorité ( $P_i$ )																															
A	25	1																															
B	60	3																															
C	42	2																															
D	105	5																															
E	75	4																															
N	Utilization Bound																																
1	100.0%																																
2	82.8%																																
3	78.0%																																
4	75.7%																																
5	74.3%																																
10	71.8%																																