Examen de temps-réel : partie I

Durée conseillée : 1h00
Pondération : 10 points sur 20
Documents de cours seuls autorisés.

0.1 Ordonnancement Rate Monotonic (RM) [6 points]

On considère l'ordonnancement RM de tâches, avec une échéance des tâches égale à leur période. A chaque tâche i, on associe les propriétés suivantes :
- A_i, la date de première activation de la tâche (état prêt) ;
- C_i, la durée d'exécution CPU nécessaire à l'exécution complète de la tâche (sans préemption) ;
- T_i, la période d'activation de la tâche périodique ;

Si le critère de Liu et Layland n'est pas vérifié, on peut conclure sur l'ordonnançabilité par une simulation avec un chronogramme.
Une autre méthode consiste à calculer R_i, le temps de réponse dans le pire cas de la tâche i. On définit le temps de réponse d'une tâche comme le temps entre son instant d'activation A_i et l'instant où son exécution se finit. Le temps de réponse dans le pire cas, R_i, est égal à la durée d'exécution C_i de la tâche additionnée au temps d'attente induit par l'exécution des tâches de priorité supérieure :

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil Cj \tag{1}$$

avec
- $hp(i)$, l'ensemble des tâches de priorité supérieure à la tache i
- $\lceil x \rceil$, l'opérateur plafond (ceil) qui est le plus petit entier supérieur ou égal à x

En pratique, on calcul R_i de manière itérative par :

$$R_i^0 = C_i \tag{2}$$

$$R_i^{k+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i^k}{T_j} \rceil C_j \tag{3}$$

avec arrêt de l'itération si il existe k' tel que :
- $R_i^{k'+1} = R_i^{k'}$ (Réussite)

ou
- $R_i^{k'} > T_i$ (Échec de l'ordonnancement)

Ainsi dans le second exemple d'ordonnancement RM traité en cours ($C_1 = 25, T_1 = 50$, $C_2 = 30, T_2 = 75$), on aurait pu conclure sur l'échec de l'ordonnancement en montrant que $R_2 > T_2$ au lieu de faire le chronogramme :
- Tâche 1 : $hp(1) = \emptyset$

$$R_1^0 = C_1 = 25 \quad \text{et} \quad R_1^1 = C_1 = R_1^0 \quad \text{(Réussite)} \tag{4}$$

- Tâche 2 : $hp(j) = \{1\}$ (tâche 1 plus prioritaire)

$$\begin{aligned} R_2^0 &= C_2 = 30 \\ R_2^1 &= C_2 + \lceil \frac{R_2^0}{T_1} \rceil C_1 = 30 + \lceil \frac{30}{50} \rceil 25 = 30 + 1 \times 25 = 55 \\ R_2^2 &= C_2 + \lceil \frac{R_2^1}{T_1} \rceil C_1 = 30 + \lceil \frac{55}{50} \rceil 25 = 30 + 2 \times 25 = 80 > T_2 \quad \text{(Échec)} \end{aligned}$$

1. On considère un système embarqué muni d'un microprocesseur gérant 2 tâches indépendantes et périodiques :

Tâche i	A_i (ms)	C_i (ms)	T_i (ms)
1	2	2	6
2	0	3	10

(a) Calculer le taux d'occupation du CPU. D'après le critère de Liu et Layland, que peut on dire de l'ordonnançabilité RM de ces 2 tâches ?

$\frac{2}{6} + \frac{3}{10} = 0{,}33 + 0{,}3 = 0{,}63$

→ ordonnançable

(b) Tracer un chronogramme de l'usage du CPU par les tâches sur un horizon de 20 ms seulement avec un ordonnancement RM et le choix adéquat pour les priorités des tâches. On néglige le temps CPU consommé par le système d'exploitation.
(On conseille d'indiquer en premier lieu tous les instants d'activation sur le chronogramme.)

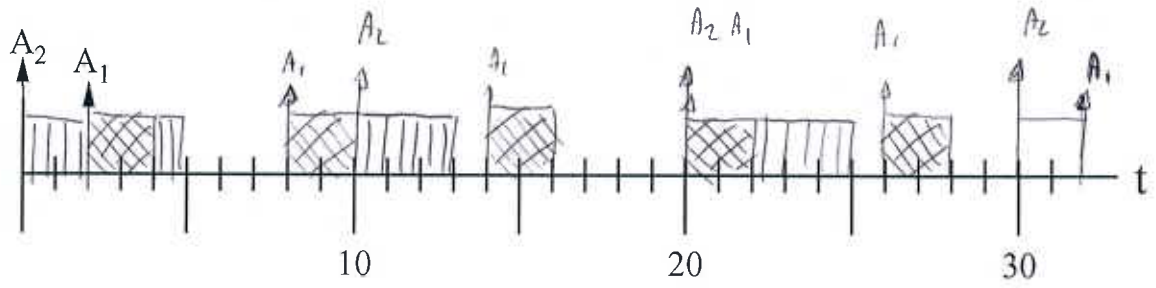


FIGURE 1 – Chronogramme d'exécution des 2 tâches.

Conclure sur l'ordonnançabilité d'après le chronogramme : ordonnançable

(c) Pourquoi pouvons-nous nous limiter à un horizon de 30ms pour le chronogramme ? Que représente 30ms par rapport aux périodes des tâches d'un point de vue mathématique.

plus petit commun multiple.

(d) On souhaite vérifier notre chronogramme, en calculant les temps de réponse dans le pire cas R_1 et R_2 des 2 tâches (on donnera R_2^0, R_2^1,....) :

$R_1^0 = C_1 = 2$ (immédiat)

$R_2^0 = C_2 = 3$

$R_2^1 = C_2 + \left\lceil \frac{R_2^0}{6} \right\rceil \cdot 2 = 3 + 1 \cdot 2 = 5$

$R_2^2 = C_2 + \left\lceil \frac{R_2^1}{6} \right\rceil 2 = 3 + 2 = 5$ fin.

Ces temps de réponse sont-ils conforment à votre chronogramme ? Justifier.

pire cas de réponse de τ_2 = 5 ms.

2. La survenue d'événements sporadiques peut nécessiter la création de tâches apériodiques pour traiter ces événements. Une des approches pour prendre en compte ces tâches apériodiques dans l'ordonnancement RM est de définir une tâche périodique spéciale s, appelée serveur de tâches apériodiques.

   Ce serveur se voit allouer un temps d'exécution CPU C_s pendant lequel il peut traiter les tâches apériodiques en attente. Lorsqu'il est actif (état running), ce serveur exécute les tâches apériodiques jusqu'à :
   - l'épuisement des tâches
   - ou l'expiration du temps de CPU C_s accordé au serveur. Les tâches apériodiques non traitées le seront lors de la prochaine période T_s.

   Soient les propriétés suivantes pour la tâche s, le serveur de tâches apériodiques :

Tâche	A_i (ms)	C_i (ms)	T_i (ms)
s	0	?	15

   (a) D'après le critère de Liu et Layland, quel choix de temps de CPU C_s en millisecondes entières permettrait d'assurer l'ordonnançabilité des 3 tâches (1,2 et s) ? Justifier.

si $C_s = 1$ $\quad U = 0{,}63 + \frac{1}{15} = 0{,}69$ ok

$C_s = 2$ $\quad U = 0{,}63 + \frac{2}{15} = 0{,}76$ ok

si $C_s = 3$ $\quad U = 0{,}63 + \frac{3}{15} = 0{,}83 > 0{,}78$ C_s ne doit pas être supérieur

   (b) On choisit finalement $C_s = 3$. D'après le critère de Liu et Layland, que peut-on dire de l'ordonnançabilité RM de ces 3 tâches ?

On ne peut conclure.

   (c) Tracer un chronogramme de l'usage du CPU par les 3 tâches sur un horizon adéquat avec un ordonnancement RM et le choix adéquat de priorités pour les tâches.

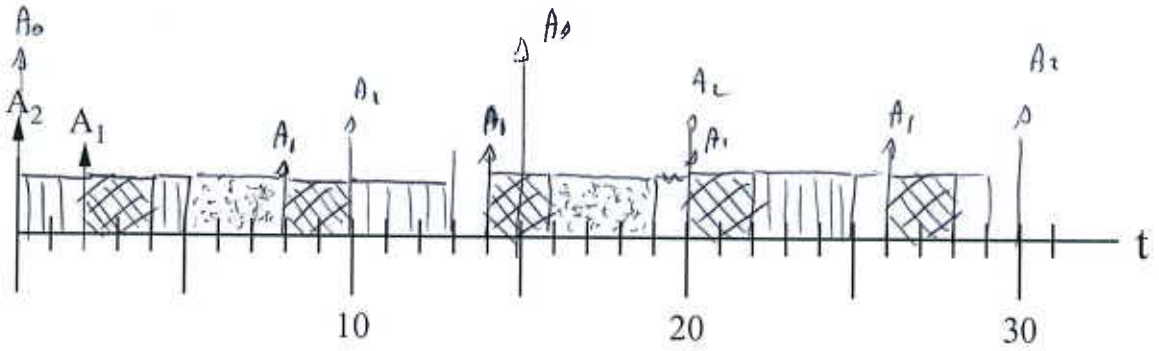


FIGURE 2 – Chronogramme d'exécution des 3 tâches.

Conclure sur l'ordonnançabilité d'après le chronogramme : Ordonnançable.

(d) Sur l'horizon de 30ms du chronogramme, combien de millisecondes le CPU est-il inutilisé ? Pouvions nous prévoir ce résultat avec entre autre le résultat de la question (b) ?

5 ms

30 - (0,833 × 30) = 5, ms -

(e) D'après le chronogramme (où $A_1 \neq 0$), quel est le pire temps de réponse de la tâche s ?

8 ms

(f) Calculer avec la formule le temps de réponse dans le pire cas du serveur de tâches apériodiques R_s (donner Rs^0, R_s^1,...). Le résultat est différent de celui obtenu sur le chronogramme, qui ne représente pas la pire situation d'ordonnancement, à savoir $A_1 = A_2 = A_s = 0$ ms.

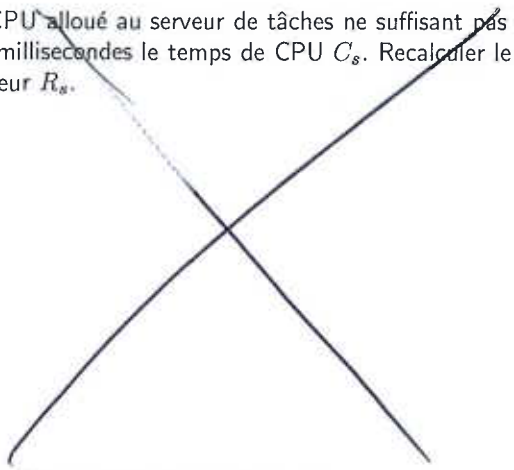
$R_0^0 = 3$.

$R_0^1 = 3 + \lceil \frac{3}{6} \rceil . 2 + \lceil \frac{3}{10} \rceil . 3 = 8$ ms

$R_0^2 = 3 + \lceil \frac{8}{6} \rceil . 2 + \lceil \frac{8}{10} \rceil . 3 = 3 + 4 + 3 = 10$ ms

$R_0^3 = 3 + \lceil \frac{10}{6} \rceil 2 + \lceil \frac{10}{10} \rceil . 3 = 3 + 4 + 3 = 10$ ms

(g) Le temps CPU alloué au serveur de tâches ne suffisant pas pour l'application, on souhaite porter à 4 millisecondes le temps de CPU C_s. Recalculer le temps de réponse dans le pire cas du serveur R_s.



Conclure sur l'ordonnancabilité si $C_s = 4$:

0.2 Héritage de priorité [4 points]

5

On travaille ici avec des sémaphores binaires (dont le compteur est initialisé à 1). Pour éviter des inversions de priorité, ceux-ci implémentent l'héritage de priorité :
Si une tâche A bloque car le sémaphore (une ressource) qu'elle tente d'acquérir est déjà acquis par une tâche B et si la priorité courante de B est inférieure à celle de A, alors :

- la priorité de la tâche B est augmentée au même niveau que la priorité de la tâche A.
- dès que B libère le sémaphore, la tâche B retrouve sa priorité antérieure.

Dans le cas où plusieurs sémaphores sont utilisés, les sections critiques et donc les héritages de priorité peuvent être imbriqués, tels que représentés sur les figure 3.

1. Compléter la ligne indiquant l'évolution de la priorité courante de la tâche 1 dans les chronogrammes 3 et 4 où l'héritage de priorité est implémenté.
2. Les 2 opérations standard sem_wait et sem_post sur un objet sémaphore (S) sans héritage de priorité sont implémentées avec le pseudo-code suivant (schedule()= appel ordonnanceur) :

sem_wait	sem_post
(1) Disable-interrupt	(8) Disable-interrupt
(2) if S.counter > 0	(9) If non-empty(S.queue)
(3) then S.counter - - 1	(10) then {
(4) else {	(11) next-to-run := get-first(S.queue)
(5) insert(current-task, S.queue)	(12) insert(next-to-run, Ready-queue)
(6) schedule()}	(13) schedule() }
(7) Enable-interrupt	(14) else S.counter ++ 1
	(15) Enable-interrupt

(a) D'après l'implémentation proposée, dans l'ensemble des tâches en attente du sémaphore S, quelle tâche va être débloquée par l'opération sem_post(&S) ?
- ✗ la tâche avec le temps d'attente sur le sémaphore le plus long
- ○ la tâche avec le temps d'attente sur le sémaphore le plus court
- ○ la tâche avec la priorité la plus élevée
- ○ la tâche avec la priorité la plus faible

Justifier votre réponse :
si S.queue est une FIFO, la première tâche de liste est la + ancienne. 1/2 1/2

(b) On définit :
- current-task.prio, la priorité courante de la tâche courante (celle qui effectue l'appel à la fonction sem_wait ou sem_post) ;
- S.owner.prio, la priorité courante de la tâche ayant acquis le sémaphore.

Pour implémenter la protocole d'héritage de priorité sur un sémaphore binaire, choisir les 2 meilleures modifications à appliquer au code précédent parmi les propositions suivantes :

i. la ligne (5) doit être étendue avec "sauvegarder current-task.prio et if ~~S.owner.prio > current-task~~.prio then current-task.prio = s.owner.prio"

ii. la ligne (5) doit être étendue avec "sauvegarder current-task.prio et if S.owner.prio < current-task.prio then current-task.prio = s.owner.prio"

iii. la ligne (5) doit être étendue avec "sauvegarder S.owner.prio et if ~~S.owner.prio > current-task~~.prio then S.owner.prio = current-task.prio"

iv. la ligne (5) doit être étendue avec "sauvegarder S.owner.prio et if S.owner.prio < current-task.prio then S.owner.prio = current-task.prio"

v. la ligne (14) doit être étendue avec "restaurer current-task.prio à la priorité précédemment sauvegardée"

vi. la ligne (12) doit être étendue avec "restaurer current-task.prio à la priorité précédemment sauvegardée"

(Dernière question inspirée de Exam for Real Time Systems, 2010 Oct 25, Wang Yi)

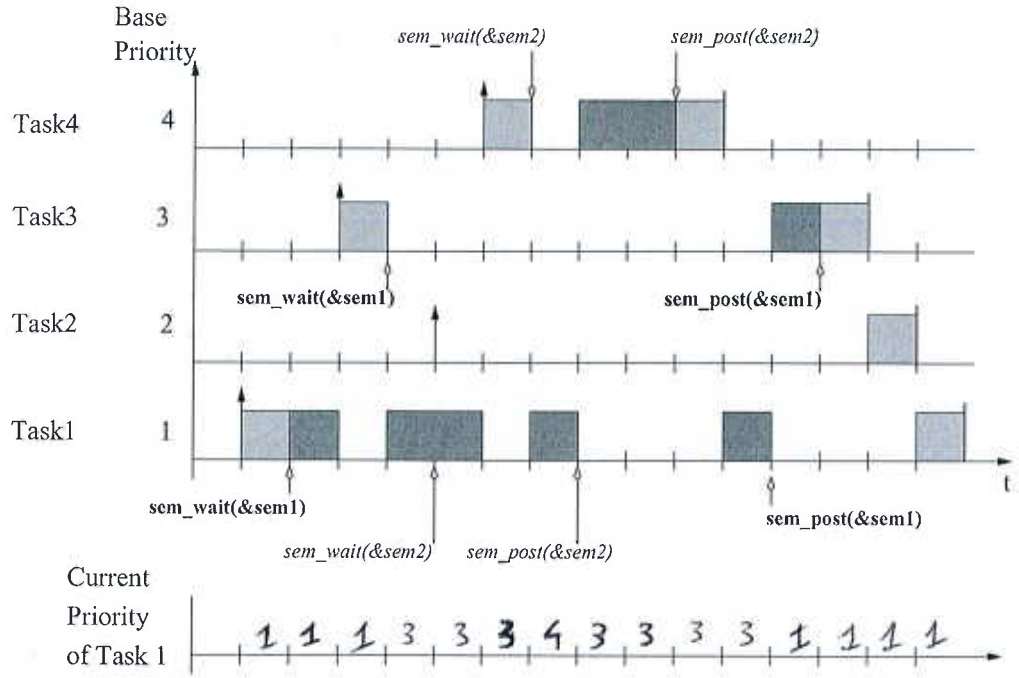


FIGURE 3 – Chronogramme avec héritages de priorité

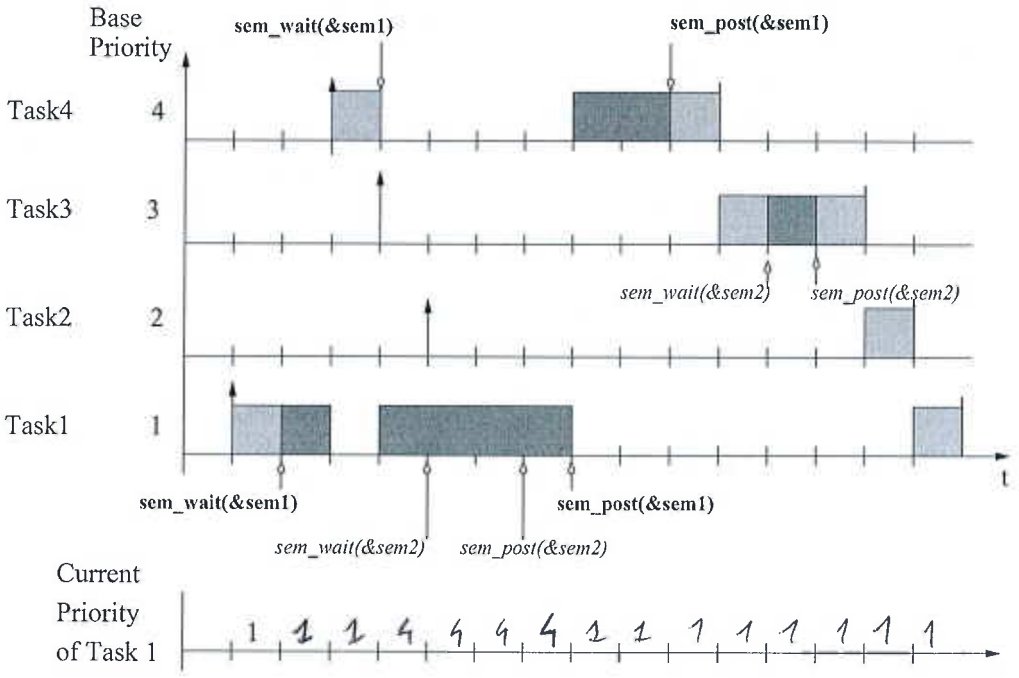


FIGURE 4 – Chronogramme avec héritages de priorité

Partie 2

1/ Il faut commenter 31, 32 et 51, 52.

Car : 1/ Les 4 mutex sont bloqués au démarrage du programme. (fermés)

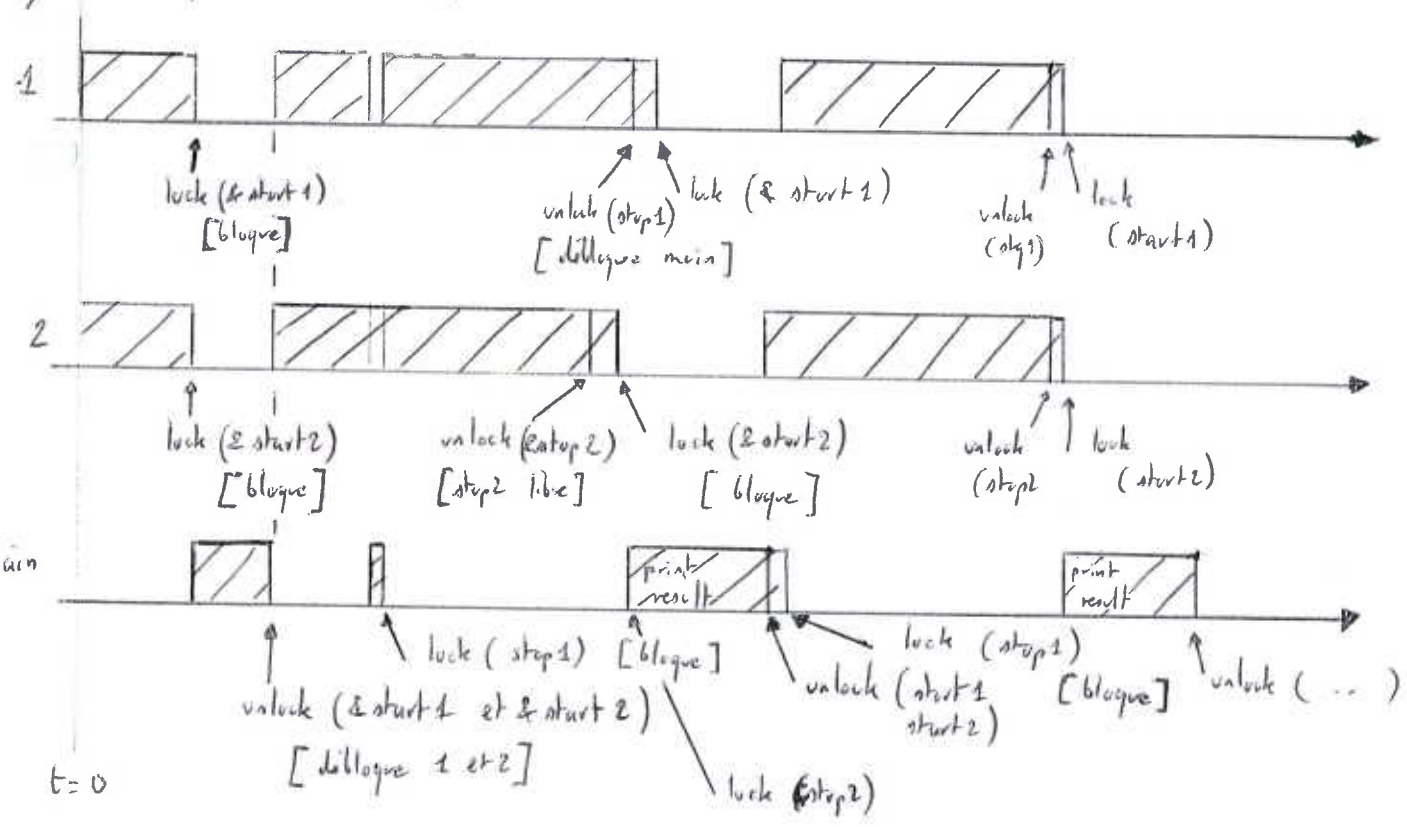
2/ le thread main est le seule pouvant s'exécuter car 1 et 2 bloquent sur les mutex fermés. Il libère 1 et 2 par une libération des mutex start 1 et start 2, puis bloque sur les mutex stop1 et stop2.

→ <u>thread main est bloqué</u>

3/ 1 et 2 acquièrent les mutex start 1 et start 2 et donc les ferment. Lignes 32 et 52, les threads 1 et 2 tentent d'acquérir start_1 et start_2 qu'ils ont déjà acquis / bloqués.

→ 1 et 2 bloquent sur la seconde acquisition de start 1 et start 2 alors que thread main est également bloqué sur d'autres mutex. (132 et 52)

2/ d'après les affichages :



à t=0, start 1, start 2, stop 1 et stop 2 sont acquis / fermés par le MAIN.