Date: 22 octobre 2001

École Polytechnique de Montréal

INF4600 : Systèmes temps réel

Questionnaire du QUIZ : Automne 2001

PRENEZ NOTE DES DIRECTIVES SUIVANTES:

- Documentation non permise.
- Durée de l'examen : 120 minutes i.e. 19:00 à 21:00
- Ne pas écrire en rouge.
- L'examen comprend donc 5 questions pour un total de 20 points et il compte pour 20% de la note finale. La distribution des numéros est la suivante:

Question 1	**Caractéristiques des systèmes temps réel**	**4 points**
Question 2	**Vrai ou faux**	**3 points**
Question 3	**Concepts temps réel dans MicroC et VxWorks**	**5 points**
Question 4	**Programmation en MicroC**	**4 points**
Question 5	**Programmation en VxWorks**	**4 points**

Professeur: Guy Bois
Chargé de cours: Niculina Ignat

QUESTION #1 (4 points) Caractéristiques des systèmes temps réel

a) Donnez la définition des termes suivants (quelques lignes par définition au maximum). Au besoin, utilisez un exemple pour faciliter l'explication:

- Micronoyau
- Système embarqué
- Système distribué temps réel
- Instabilité du déclenchement d'une tâche (*release jitter*)

b) Le fait de donner des priorités différentes aux processus ne garanti pas qu'un processus temps réel ayant la priorité maximal ne puisse reprendre le contrôle au moment désiré.

b.1) Nommez un phénomène qui valide cette affirmation dont un cas est bien connu.
b.2) Avec 3 tâches et 1 mutex, démontrez graphiquement ce phénomène et donc cette affirmation. Faites les hypothèses nécessaires.

c) Dans un système d'exploitation temps réel tel que MicroC ou VxWorks :

c.1) Quels sont les avantages de minimiser le pas d'ordonnancement (*tick*)?
c.2) Quels sont les désavantages de trop minimiser ce même pas d'ordonnancement (*tick*)?

QUESTION #2 (3 points) Répondre par vrai ou par faux aux questions suivantes et justifiez en quelques lignes lorsque demandé.

1) L'identificateur d'une tâche microC/OS-II est l'adresse du TCB (*Task Control Block*) de la tâche dans la mémoire RAM.

2) Lors de l'appel de la routine OSTaskDel (task1) de microC/OS-II, task1 passe à l'état « waiting ».

3) Une tâche microC/OS-II prête à rouler s'exécute quand toutes les tâches plus prioritaires de l'application se trouve dans l'état « waiting » ou « dormant ».

4) Si une application microC/OS-II crée 20 tâches, elle doit leur assigner les priorités de 0 à 19. **Justifiez la réponse**.

5) En microC/OS-II, la queue des tâches en attente (« waiting ») est une structure de données spéciale, qui permet que le temps d'insertion ou de suppression d'une tâche demeure constant, peu importe la priorité de la tâche.

6) Si dans une application VxWorks il existe une ressource partagée entre des tâches et des interruptions, protéger la ressource par le biais d'un sémaphore mutex s'avère suffisant. **Justifiez la réponse.**

QUESTION 3 (5 points) Concepts temps réel dans MicroC et VxWorks

a) Si une section de code est critique du point de vue temporel (son temps d'exécution ne doit jamais être dépassé), elle doit être protégée par le verrouillage des interruptions (voir la routine Ma_routine).

Ma_routine:

…….

Désactiver les interruptions;

Exécuter la section de code critique;

Réactiver les interruptions;

……..

a.1) Expliquez pourquoi le verrouillage des interruptions n'est pas toujours suffisant.
a.2) Ajoutez deux lignes en pseudo-code qui pourraient assurer une protection supplémentaire à la section de code critique.

b) Soit le pseudo-code d'une routine de traitement des interruptions (ISR) en microC/OS-II :

Mon ISR1:

Sauvegarder tous les registres du CPU;

Appeler OSIntEnter();

Exécuter le code de l'ISR utilisateur;

Appeler OSIntExit();

Restaurer tous les registres du CPU

Si la ligne **Appeler OSIntEnter();** est substituée par **OSIntNesting++;**, le code devient:

Mon ISR2:

Sauvegarder tous les registres du CPU;

OSIntNesting++;

Exécuter le code de l'ISR utilisateur;

Appeler OSIntExit();

Restaurer tous les registres du CPU

Le code obtenu (Mon ISR2) est-il équivalent au code initial (Mon ISR1) du point de vue logique et sécurité de l'exécution du programme? **Justifiez la réponse**.

c) Soit les états possibles d'une tâche VxWorks : *pended, pended&suspended, suspended, delayed&suspended, delayed* et *ready/exec*. Quel serait l'état d'une tâche VxWorks qui attend un sémaphore mutex avec une certaine temporisation (pour un nombre spécifié de « ticks » d'horloge système)? **Justifiez la réponse.**

d) Supposons qu'une application VxWorks crée deux tâches : tâche1, de priorité 25 et tâche2, de priorité 50. La tâche1, étant est la plus prioritaire, commence à s'exécuter, ensuite elle se bloque, à l'attente d'un sémaphore mutex, qui n'est pas disponible. Alors, l'ordonnanceur va lancer en exécution la tâche2. Indiquez si la tâche2 va entrer en exécution immédiatement ou dans le cadre de l'ISR qui va traiter le « tick » d'horloge suivant. **Justifiez la réponse.**

e) En VxWorks, pour éviter qu'une tâche soit supprimée pendant qu'elle exécute une zone de code critique, on utilise habituellement la routine taskSafe():

```
taskSafe() ;
semTake(semId,WAIT_FOREVER);
Section de code critique
semGive(semId);
taskUnsafe();
```

Indiquez une méthode qui permet que l'appel des routines taskSafe() et taskUnsafe() soit éliminé, en assurant toutefois la sécurité de suppression de la tâche qui exécute la section de code critique.

Question 4 (4 points) Programmation en MicroC

a) Soit le code C suivant, exécuté par microC/OS-II, dans un système temps-réel avec des interruptions imbriquées:

```
int Temp ;

void swap (int* x, int* y)
{
        Temp = *x ;
        *x = *y ;
        *y = Temp ;
 }
```

Remarquez que la fonction **swap** n'est pas réentrante.

a.1) Indiquez trois méthodes pour rendre réentrante la fonction **swap**. Pour chaque méthode, écrivez le code C modifié.

a.2) Précisez les avantages et les inconvénients de chacune des trois méthodes indiquées au point a.1).

b) Soit une tâche *task1*, qui à un moment donné dans son code désire accéder à une ressource partagée dont au maximum 5 tâches différentes peuvent accéder en même temps (de manière concurrente). À l'aide de microC/OS-II, montrez comment *task*1 pourra accéder à cette ressource tout en respectant la contrainte du nombre maximum de 5 tâches et **en utilisant uniquement une queue comme mécanisme de synchronisation**. Donnez le code pour *task1*, ainsi que le bout de code du *main* qui initialise la queue et démarre *task1*. Dans *task1*, considérez que l'accès à la ressource partagée se fait via une fonction prédéfinie *put*.

Question 5 (4 points) Programmation en VxWorks

a) Soit l'application VxWorks de la figure 5.1 (page 8) :

a.1) Indiquez dans quelle zone de mémoire RAM les paramètres suivants de la routine taskSpawn sont mémorisées :

- semId
- "tMyTask"
- la priorité de la tâche "tMyTask"

a.2) Indiquez quel sera l'état de la tâche "tMyTask" après l'exécution de la routine "taskSpawn".

a.3) La vérification de la valeur de retour dans le cas des routines "taskSpawn", "semMCreate" et "semTake" est absolument nécessaire? Justifiez la réponse.

a.4) Est-ce que la ligne "**int count = 0;**" pourrait être remplacée par "**int count;** ", sans que le résultat logique de l'application soit affecté? Justifiez la réponse.

b) Soit le code C de la figure 5.2 (page 9), exécuté par VxWorks :

b.1) Trouver trois erreurs dans ce code.

b.2) Expliquez le rôle de la variable **busy**, dans ce code.

b.3) Modifiez les appels de taskDelay(300) et taskDelay(180), afin que la durée de retard soit de 5 secondes, respectivement 3 secondes, peu importe le microprocesseur sur lequel le code ci-dessus est exécuté.

b.4) Modifiez le code et substituez la variable **busy** par un sémaphore mutex (utilisez l'annexe qui explique les routines de création et de manipulation des sémaphores VxWorks).

```
#include <vxWorks.h>
#include <stdio.h>
#include <taskLib.h>
#include <semLib.h>

int v1;
int count = 0;

void init(void)
    {
        SEM_ID semId;

        if ((semId = semMCreate (SEM_Q_PRIORITY |
                                 SEM_DELETE_SAFE |
            SEM_INVERSION_SAFE)) == NULL)
                {
                printf ("semMCreate failed\n");
                return;
                }

                if(taskSpawn("tMyTask",100,0,20000,(FUNCPTR)mutex,semId,0,0,0,0,0,0,0,0,0)
          == ERROR)
                {
                printf ("taskSpawn failed\n");
                return;
                }
}

void mutex (SEM_ID semId)
{
        FOREVER
        {
                if(semTake (semId, WAIT_FOREVER) == ERROR)
                {
                        printf ("semTake failed\n");
                        return;
                }

                v1 = count;
                count++;
                semGive (semId);
        }
}
```

Figure 5.1

```
#include "vxWorks.h"
#include "taskLib.h"

void foo();
int busy = FALSE;

STATUS startFoo (char *printStr)
        {
        char flag;
        flag = *printStr;
        if (taskSpawn ("tFoo", 200, 0, 5000, foo, flag) == ERROR)
                {
                printf (" taskSpawn failed ");
                return (ERROR);
                }
        taskDelay (300);
        return (OK);
        }

void foo (char flag)
        {
        FOREVER
                {
                if (busy == FALSE)
                        {
                        busy = TRUE;
                        if (*flag == 'y')
                                printf ("In the critical region\n");
                        busy = FALSE;
                        }
                taskDelay (180);
                }
                                                }
```

Figure 5.2