1 Introduction

Ce document montre, à travers l'implémentation de la fonction Wait, plusieurs concepts reliés aux systèmes d'exploitation préemptifs. On y retrouve des notions sur l'exécution concourante des tâches, les fonctions réentrantes, les appels de système et plus.

2 Exercice

Supposons un système d'exploitation préemptif exécutant trois tâches (T1, T2 et T3) à tour de rôle, comme suit :

Tâche 1	Tâche 2	Tâche 3

void T1(void)
{
  initT1();
 	
  while(1)
  {
  T1A();
  Wait(101);
  T1B();
  Wait(102);
  }
}
	void T2(void)
	{
	  initT2();
	
	  while(1)
	  {
	  T2A();
	  Wait(201);
	  T2B();
	  Wait(202);
	  T2Dure5mins();
	  }
	}
		void T3(void)
		{
		  initT3();
		
		  while(1)
		  {
		  T3A();
		  Wait(301);
		  T3B();
		  }
		}

Interruption périodique du système d'exploitation

void OSTimerISR()
{
  SauveContexteTacheActive();
  TacheActive = DetermineProchaineTacheActive()
  RestoreTacheActive();
}

Dans ce système, un programmeur novice a d'abord programmé la fonction Wait comme suit :

Wait – Implémentation A

void Wait(int Xms)
{
  int i;
  int j;
  for(i = 0; i < Xms; i++)
  for(j = 0; j < 127695; j++){ }; //Le microprocesseur exécute cette boucle en 1 ms
}

Malheureusement, son supérieur a rejeté l'implémentation. Ça ne marche pas. Aussi, le programmeur novice est revenu avec une seconde implémentation, dans laquelle un autre timer que celui du OS est configuré pour générer une interruption à toutes les millisecondes :

Wait – Implémentation B	
Fonction Wait	Interruption de Timer

void Wait(int Xms)
{
  MsCtr = Xms;
  while(MsCtr != 0){};
}
	void ISRTimerPeriodiqueMs(int Xms)
	{
	  if(MsCtr != 0)
	  MsCtr--;
	}

Cette deuxième implémentation ne fonctionnant pas, un troisième essai a été fait :

Wait – Implémentation C	
Fonction Wait	Interruption de Timer

void Wait(int Xms)
{
  int Timestamp = MsCtr;
  while(MsCtr – Timestamp < Xms){};
}
	void ISRTimerPeriodiqueMs(int Xms)
	{
	  MsCtr++;
	}

Cette fois-ci, les essais ont démontré que le code fonctionne. Cependant, il n'est pas optimal et une autre implémentation a été requise. Répondez aux questions suivantes :

1) Pourquoi l'implémentation A ne fonctionne pas?
2) Pourquoi l'implémentation B ne fonctionne pas?
3) Pourquoi l'implémentation C n'est pas optimale?
4) Comment devrait être implémentée la fonction Wait avec un système d'exploitation préemptif?

Réponses :

1) La boucle j dure une milliseconde lorsqu'elle utilise tout le temps de microprocesseur. Or, le système d'exploitation interrompt la boucle pour exécuter d'autres tâches. Cela aura pour conséquence d'augmenter le temps d'attente d'une manière imprévisible pour chaque tâche.

Dans le système en exemple, si les trois tâches attendent en même temps et qu'elles sont exécutées à tour de rôle, on peut présumer que Wait(100) produira une attente de 300ms…

2) La variable MsCtr est une variable globale écrite dans plusieurs contextes : l'interruption, la tâche 1, la tâche 2, la tâche 3 et toutes les autres tâches qui appellent Wait… Quand la tâche 1 commence son attente, elle écrit MsCtr = 101 et commence à attendre. La tâche 1 sera interrompue par le système d'exploitation qui exécutera la tâche 2. Lorsque celle-ci sera exécutée, elle attendra aussi à son tour, mettant MsCtr = 201 : le délai d'attente de la tâche 1 sera faussé. Conclusion : la dernière tâche à attendre déterminera le délai d'attente de toutes les tâches…

En d'autres termes, la fonction Wait n'est pas ré-entrante parce qu'elle utilise une variable globale. Elle ne peut donc pas être utilisée par les tâches du système.

3) MsCtr n'est plus écrit que dans un seul contexte et la variable Timestamp est locale (probablement de la mémoire réservée sur la pile de chaque tâche) : la fonction Wait attendra le bon délai pour chaque tâche.

Cependant, du temps de microprocesseur est gaspillé inutilement à attendre. Par exemple, on peut tourner en rond dans la tâche 1 pendant que la tâche 2 pourrait être exécutée pour réaliser du travail utile. Il est inutile d'accorder du temps de microprocesseur à une tâche qui attend pour qu'elle tourne en rond dans un while…

4) Pour éviter qu'une tâche ne soit exécutée alors qu'elle attend, il faut changer l'état de la tâche et gérer celle-ci à même le système d'exploitation :

Fonction Wait	Interruption de Timer pour le OS

void Wait(int Xms)
{
  TacheActive->Timestamp = Xms;
  TacheActive->Etat = ATTENTE;
  while(TacheActive->Etat == ATTENTE){};
}
	void OSTimerISR() //Aux millisecondes!
	{
	  SauveContexteTacheActive();
	  for(int i = 0; i < NbreTaches; i++)
	  if(Tache[i].Timestamp != 0)
	  {
	  Tache[i].Timestamp--;
	  if(Tache[i].Timestamp == 0)
	  Tache[i].Etat = PRET;
	  }
	  TacheActive = DetermineProchaineTacheActive()
	  RestoreTacheActive();
	}

Dans cet exemple, la fonction DeterminerProchaineTacheActive ne choisit que les tâches dans l'état PRET…