Atelier sur les Mutex

Ce document explique ce que sont les mutex et à quoi ils servent.

Supposons un appareil avec un système d'exploitation préemptif exécutant deux tâches : T1 et T2. Les deux tâches sont exécutées à tour de rôle et elles sont interrompues par le système d'exploitation à n'importe quel moment. Lorsque le système d'exploitation décide d'exécuter une tâche dans son interruption périodique, il reprend l'exécution à l'endroit où elle s'était arrêtée.

Dans ce système, le microcontrôleur est relié à une mémoire EEPROM par un bus SPI. Pour lire des informations dans la EEPROM, il faut envoyer une commande de lecture, suivie de l'adresse des informations sur le bus SPI, puis envoyer des coups d'horloge pour lire des octets.

La mémoire EEPROM supporte des ***transactions atomiques*** seulement. Autrement dit, lorsqu'on commence à lire la EEPROM, il faut finir la transaction. On ne peut pas envoyer deux commandes de lecture consécutives ou deux adresses consécutives...

Le pseudo-code des tâches 1 et 2 est le suivant :

Tâche 1	Tâche2	LectureEEPROM

void T1(void)
{
  InitT1();
  while(1)
  {
  T1partieA();
  T1partieB();
  Octet=LisEEPROM(400);
  T1partieC();
  }
}
	void T2(void)
	{
	  InitT2();
	  while(1)
	  {
	  T2partieA();
	  Octet=LisEEPROM(100);
	  T2partieB();
	  T2partieC();
	  }
	}
		char LisEEPROM(int adresse)
		{
		  SPI_ENVOI(CMD_LIRE);
		  SPI_ENVOI(MSB(adresse))
		  SPI_ENVOI(LSB(adresse))
		  return SPI_LIS_OCTET();
		}

Comme implémenté, ce code ne fonctionne pas? Pourquoi?

Si le système d'exploitation interrompt la Tâche 1 pendant l'exécution de LisEEPROM et si la fonction LisEEPROM est exécutée dans la Tâche 2 ensuite, il est possible d'envoyer deux commandes de lectures consécutives sans envoyer d'adresse ou d'envoyer des séquences comme « commande (T1) – Adresse (T1)- Commande (T2)-Adresse(T2)-Lire Octet(T2) »… Le même problème survient si la Tâche 2 est interrompue pendant la lecture de l'EEPROM.

La fonction LisEEPROM n'est pas ***réentrante*** : on ne peut pas entrer dans la fonction lorsqu'on exécute déjà la fonction dans une autre tâche ou contexte. Il s'agit d'une ***section critique*** qui doit être exécutée en totalité avant de passer à une autre tâche.

La correction suivante règle-t-elle le problème?

Tâche 1	Tâche2	LectureEEPROM

void T1(void)
{
InitT1();
while(1)
{
T1partieA();
T1partieB();
if(EEPROMDisponible)
{
EEPROMDisponible = 0;
Octet=LisEEPROM(400);
EEPROMDisponible = 1;
}
T1partieC();
}
}
	void T2(void)
	{
	InitT2();
	while(1)
	{
	T2partieA();
	if(EEPROMDisponible)
	{
	EEPROMDisponible = 0;
	Octet=LisEEPROM(200);
	EEPROMDisponible = 1;
	}
	T2partieB();
	T2partieC();
	}
	}
		char LisEEPROM(int adresse)
		{
		SPI_ENVOI(CMD_LIRE);
		SPI_ENVOI(MSB(adresse))
		SPI_ENVOI(LSB(adresse))
		return SPI_LIS_OCTET();
		}

On peut penser qu'ajouter un drapeau (EEPROMDisponible) règle le problème. Toutefois, ce n'est pas le cas. En effet, un problème survient encore si l'interruption du système d'exploitation se produit entre « **if(EEPROMDisponible)** » et « **EEPROMDisponible = 0; »**. Dans ce cas, il est encore possible de faire deux accès entremêlés à la EEPROM…

<u>Une ébauche de solution :</u>

Tâche 1	Tâche2	LectureEEPROM

void T1(void)
{
InitT1();
while(1)
{
T1partieA();
T1partieB();
DesactiveINTduOS();
if(EEPROMDisponible)
{
EEPROMDisponible = 0;
ActiveINTduOS();
Octet=LisEEPROM(400);
EEPROMDisponible = 1;
}
ActiveINTduOS();
T1partieC();
}
}
	void T2(void)
	{
	InitT2();
	while(1)
	{
	T2partieA();
	DesactiveINTduOS();
	if(EEPROMDisponible)
	{
	EEPROMDisponible = 0;
	ActiveINTduOS();
	Octet=LisEEPROM(200);
	EEPROMDisponible = 1;
	}
	ActiveINTduOS();
	T2partieB();
	T2partieC();
	}
	}
		char LisEEPROM(int adresse)
		{
		SPI_ENVOI(CMD_LIRE);
		SPI_ENVOI(MSB(adresse))
		SPI_ENVOI(LSB(adresse))
		return SPI_LIS_OCTET();
		}

Dans cette ébauche de solution on désactive l'interruption du système d'exploitation avant de vérifier un drapeau afin de savoir si la ressource (la mémoire SPI-EEPROM) est disponible.

Cette solution fonctionne, mais elle n'est pas parfaite : si la EEPROM n'est pas disponible, on ne lit pas d'octet et on passe à la suite du code... Il faudrait attendre après EEPROMDisponible pour continuer :

Tâche 1	Tâche2	LectureEEPROM

void T1(void)
{
InitT1();
while(1)
{
T1partieA();
T1partieB();
GetEEPROMMutex();
Octet=LisEEPROM(400);
FreeEEPROMMutex();
T1partieC();
}
}
	void T2(void)
	{
	InitT2();
	while(1)
	{
	T2partieA();
	GetEEPROMMutex();
	Octet=LisEEPROM(400);
	FreeEEPROMMutex();
	T2partieB();
	T2partieC();
	}
	}
		char LisEEPROM(int adresse)
		{
		SPI_ENVOI(CMD_LIRE);
		SPI_ENVOI(MSB(adresse))
		SPI_ENVOI(LSB(adresse))
		return SPI_LIS_OCTET();
		}

Avec les fonctions de mutex implémentées ainsi :

GetEEPROMMutex	FreeEEPROMMutex

void GetEEPROMMutex(void)
{
while(!**EEPROMDisponible)**
{
DesactiveINTduOS();
if(EEPROMDisponible)
{
EEPROMDisponible = 0;
ActiveINTduOS();
return;
}
ActiveINTduOS();
}
}
	void FreeEEPROMMutex (void)
	{
	EEPROMDisponible = 1;
	}

Il n'y a pas de problème d'exécution, mais ces fonctions sont-elles optimales?

Ces fonctions ne sont pas optimales. Lorsqu'une tâche attend après EEPROMDisponible, elle tourne en rond et du temps de microprocesseur est gaspillé. Il s'agit ***d'attente active*** : une tâche est exécutée inutilement alors qu'elle attend après une ressource. Par ailleurs, une tâche qui appellerait deux fois GetEEPROMMutex resterait gelée éternellement. Enfin, la fonction GetEEPROMMutex est spécifique à la mémoire EEPROM et ne peut être réutilisée pour les autres mutex potentiellement nécessaires dans le système.

Il faut que la fonction GetMutex déclare que la tâche attend après une ressource pour que le système d'exploitation ne la ré-exécute pas tant que le mutex n'est pas libre. Il faut aussi que la fonction GetMutex reçoive en argument l'adresse d'un mutex (qui est

fatalement une variable globale) et il faut que la fonction sorte si la tâche qui l'appelle possède déjà le mutex. Voici un exemple d'implémentation des fonctions pour obtenir un mutex et relâcher un mutex :

```
void OS_GetMutex(MutexStruct* Mutex)
{
  unsigned char IsMutexGotten = 0;

  if(Mutex->Owner == ActiveProcess)
    return;

  while(!IsMutexGotten)
  {
    OS_ENABLED = 0;
    if(Mutex->State == MutexFree)
    {
      Mutex->State = MutexGotten;
      Mutex->Owner = ActiveProcess;
      OS_ENABLED = 1;
      IsMutexGotten = 1;
    }
    else
    {
      ActiveProcess->ProcessState = Mutex->BlockedState;
      OS_ENABLED = 1;
      while(ActiveProcess->ProcessState == Mutex->BlockedState)
      {};   //OS interrupt occurs at this point
    }
  }
}

void OS_ReleaseMutex(MutexStruct* Mutex)
{
  int i;

  //Free the mutex
  Mutex->State = MutexFree;
  Mutex->Owner = 0;
  //Allow blocked processes to compete for the Mutex again
  for(i = 0; i < NumberOfAdmittedTasks; i++)
  {
    if(ListOfOSTasks[i]->ProcessState ==  Mutex->BlockedState)
    {
      ListOfOSTasks[i]->ProcessState = PSE_ProcessReady;
    }
  }

  //Wait to enter the OS interrupt:
  //This gives equal chances to each processes to compete for the mutex
  //OS_WaitX_10Ms(0); // Do not wait, just enter the OS Interrupt
}
```