GIF-3002 Partage de temps de CPU et RTOS

Ce cours discute du partage de temps de CPU entre les diverses tâches exécutées par un système microprocesseur. Il discute d'abord de diverses approches de partage du temps de processeur, puis approfondit deux approches communes : la super-boucle et le RTOS.

1 Introduction

La plupart des systèmes microprocesseurs doivent exécuter plusieurs tâches et doivent séparer le temps de calcul du microprocesseur parmi ces tâches. Dans les systèmes microprocesseurs, le microprocesseur est vu comme une ressource : il faut planifier les diverses tâches (scheduling) et leur alloué du temps de microprocesseur.

1.1 Concepts reliés

Préemption : La préemption, dans un contexte de SMI, signifie d'interrompre le processus en cours. Certains algorithmes d'allocation de temps de microprocesseur sont préemptifs : ils peuvent exécuter des tâches en plusieurs parties. D'autres algorithmes d'allocation de temps de microprocesseur ne sont pas préemptifs : lorsqu'un tâche commence, elle monopolise le microprocesseur jusqu'à ce qu'elle soit complétée.

Fonction réentrante : Une fonction réentrante est une fonction qui peut être appelée de différents contextes. Il est donc possible de rentrer de nouveau dans la fonction alors qu'elle est en cours d'exécution. Supposons par exemple que vous appelez MaFonction dans la boucle principale du main (while(1){… MaFonction()…} et que vous appeliez MaFonction également dans l'ISR du timer0, alors, il est possible d'exécuter MaFonction dans le timer0 alors qu'elle est exécutée dans le main… Par construction, les fonctions utilisant des variables locales et des variables passées en paramètre sont intrinsèquement réentrantes tandis que des fonctions utilisant des variables globales ne le sont pas.

Pointeur de fonction : Lorsque l'on parle de planification de tâches ou de sélection de tâche, les pointeurs de fonctions deviennent rapidement utiles dans le code. Un pointeur de fonction est une variable qui contient l'adresse d'une fonction. Void* ptrFonction (void) serait, par exemple, un pointeur de fonction sans paramètre d'entrée ou de sortie.

Section critique : Dans le texte qui suit, une section critique est un bout de code (comme une interaction avec un périphérique) qui doit s'exécuter sans interruption sous peine d'erreur.

Temps de microprocesseur (CPU time): Le temps de microprocesseur est le temps que prend une tâche afin d'être exécutée par le microprocesseur si elle est exécutée en continu. Le temps de microprocesseur théorique est obtenu en parcourant les instructions d'un programme et en additionnant le temps de toutes les instructions exécutées. Cependant, compter ce temps est très laborieux, voire très difficile. Habituellement, on mesure le temps de microprocesseur à l'oscilloscope (en activant/désactivant un GPIO lorsque la tâche commence/fini) ou en programmant la tâche pour qu'elle envoie un message par une interface du microprocesseur lorsque la tâche commence/finit.

Déterministique: Dans le texte qui suit, le mot déterministique signifie que l'on peut prédire, non lié au hasard. Un système déterministique produira toujours, pour un groupe d'entrées précis, les mêmes sorties.

Interruptions et exceptions : Dans le texte qui suit, une interruption est un signal généré par un périphérique pour indiquer un évènement alors qu'une exception est une erreur survenant lors de l'exécution d'instructions qui fait en sorte que l'exécution ne peut pas se poursuivre.

2 Partage de temps de CPU dans les SMI

2.1 Main, interruptions et évènements

Il y a deux approches classiques pour allouer du temps de CPU : allouer le temps en fonction d'évènements ou allouer le temps en fonction d'une horloge. Les évènements ou l'horloge peuvent être détectés par polling (on interroge successivement toutes les sources d'évènements ou l'horloge du système) ou par interruption.

Un microprocesseur exécute des tâches en arrière-plan (background), c'est-à-dire dans la boucle principale du main. Un microprocesseur exécute aussi des tâches en avant-plan (foreground), c'est-à-dire dans les interruptions.

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
InitWhile();
 	
while(1)
{
  ExecuteTachesMain();
}
	
	void MonISR(void)
	{
	  ExecuteTachesISR();
	}

2.2 Modèles de planification des tâches

Il y a plusieurs façons de planifier les tâches dans un système microprocesseur en fonction de leurs priorités respectives. Voici quelques façons de faire qui illustrent les nombreuses possibilités qui s'offriront à vous.

2.2.1 Interruptions et évènements seulement

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{};
	void MonISR1(void)
	{
	  ExecuteTache1();
	}
	…
	void MonISRN(void)
	{
	  ExecuteTacheN();
	}

2.2.2 Boucle de main et évènements seulement

Main.c	ISR.c
void InitWhile(void);	

InitInterruptions();
 	
while(1)
{
  if (Evenement1())
  ExecuteTache1();
…
  if (EvenementN())
  ExecuteTacheN();
};	

2.2.3 Boucle de main, interruptions et évènements

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
  if (Evenement1())
  ExecuteTache1();
…
  if (EvenementN())
  ExecuteTacheN();
};
	void MonISR_A(void)
	{
	  ExecuteTacheA();
	}
	…
	void MonISR_Z(void)
	{
	  ExecuteTacheZ();
	}

2.2.4 Boucle de main et interruption d'horloge pour OS seulement

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
  ExecuteTacheDeterminéeParLeOS();
};
	void ISR_OS(void)
	{
	  SaveContexteDeTacheEnCours();
	  VerifieEvenements();
	  PlanifieProchaineTache();
	  LoadContexteProchaineTache();
	}

2.2.5 Boucle de main et interruption d'horloge pour OS et interruptions de plus basse priorité

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
	void ISR_OS(void)
	{
	  SaveContexteDeTacheEnCours();
	  VerifieDrapeaux();

while(1)
{
  ExecuteTacheDeterminéeParLeOS();
};
	PlanifieProchaineTache();
	LoadContexteProchaineTache();
	}
	
	Void MonISR1(void)
	{
	EvenementPreTraitement();
	EcritDrapeau();
	}

2.2.6 Boucle de main et interruption d'horloge pour OS et interruptions (basse et haute priorité)

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
  ExecuteTacheDeterminéeParLeOS();
};
	void ISR_OS(void)
	{
	SaveContexteDeTacheEnCours();
	VerifieDrapeaux();
	PlanifieProchaineTache();
	LoadContexteProchaineTache();
	}
	
	Void MonISR1(void)
	{
	EvenementPreTraitement();
	EcritDrapeau();
	}
	
	Void MonISRN(void)
	{
	ExecuteTacheN();
	}

3 La Super-Boucle

La super-boucle est un système de partage du temps de CPU où plusieurs tâches sont exécutées dans la boucle du main. Il y a plusieurs façons de distribuer les tâches avec une super-boucle.

3.1 Foreground (avant plan) et BackGround (arrière plan)

Dans la plupart des super boucles, les tâches moins prioritaires sont exécutées en arrière-plan, dans la boucle du main et les tâches critiques, c'est-à-dire celles qui doivent être effectuées rapidement, sont exécutées lors d'interruptions.

Les tâches exécutées en arrière-plan sont dirigés par des évènements et le système est non-préemptif : les tâches en arrière-plan sont exécutées en continu, sauf lors d'interruptions.

Voir 2.2.3, Boucle de main, interruptions et évènements, pour un exemple de code.

La plus grande force de cette implémentation de partage de temps de microprocesseur est la simplicité : l'implémentation est facile. Les plus grandes faiblesses de cette implémentation sont le partage inégal du temps de CPU, le grand intervalle de temps possible avant l'exécution d'une tâche et l'absence de contrôle sur l'exécution des tâches (il est très difficile d'arrêter qui tourne en rond): si une tâche de la boucle du main prends énormément de temps (parce qu'elle attend après un I\O par exemple), l'exécution des autres tâches est retardée…

Il faut noter que la super-boucle permet d'exécuter des fonctions sur requête de l'usager et qu'elle n'est pas limitée à l'exécution de requêtes prédéfinies lors de la programmation de la mémoire d'instruction : des fonctions de la super-boucle peuvent être indiquées par des pointeurs de fonction…

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
  if (Evenement1())
  ExecuteTache1();
  VerifieRequeteDeLUsager();
  if (RequeteUsager())
  (*ptrFonctionDeLUsager)();
  …
  if (EvenementN())
  ExecuteTacheN();
};
	void MonISR_A(void)
	{
	  ExecuteTacheA();
	}
	…
	void MonISR_Z(void)
	{
	  ExecuteTacheZ();
	}

3.2 Tâches coopératives

Une variante intéressante des super-boucles sont les tâches coopératives. Les tâches coopératives sont des tâches divisées en machine d'état. À toutes les boucles du main, un état de chaque tâche est exécuté. Ainsi la tâche elle-même décide du temps de CPU quelle prend (à moins d'interruption) et il est possible de d'arrêter une tâches lors de son exécution :

Main.c	ISR.c

void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
ExecuteUnEtatDeTache1();
ExecuteUnEtatDeTache2();
ExecuteUnEtatDeTache3();
};
 	
void ExecuteUnEtatDeTache1(void)
{
static int EtatDeTache1 = 0;
 	
If(Tache1DoitEtreAnnulée)
EtatDeTache1 = 0;
 	
switch(EtatDeTache1)
{
case 0:
if(Evenement1()){
EtatDeTache1 = 1;}
break;
case 1:
ExecutePartie1DeTache1()
EtatDeTache1 = 2;
break;
…
}
	void MonISR_A(void)
	{
	ExecuteTacheA();
	}
	…
	void MonISR_Z(void)
	{
	ExecuteTacheZ();
	}

Cette variante de la super-boucle a moins de désavantages que la super-boucle de base : la boucle de main est exécutée plus rapidement et les fonctions retardent moins l'exécution des autres fonctions. En contrepartie, le code est plus lourd et les machines d'états sont des implémentations peu efficaces : il faut prendre du temps de

microprocesseur et plusieurs instructions pour les gérer. Par ailleurs, une tâche qui gèle peut toujours geler le système entier…

Comme pour la super boucle de base, il est aussi possible d'appeler des fonctions de l'utilisateur avec des tâches coopératives. Cela se fait également avec des pointeurs de fonction…

3.3 Économie d'énergie

Lorsqu'il n'a pas d'évènements qui se produisent ou lorsque les intervalles de temps entre chaque évènement sont grands, il arrive fréquemment que le microprocesseur soit endormi pour une durée prédéfinie entre chaque itération de la super-boucle :

Main.c	ISR.c

Void InitWhile(void);
 	
InitInterruptions();
 	
while(1)
{
  if (Evenement1())
  ExecuteTache1();
  VerifieRequeteDeLUsager();
  if (RequeteUsager())
  (*ptrFonctionDeLUsager)();
  …
  if (EvenementN())
  ExecuteTacheN();
 	
  //Économie d'énergie
  If(PasDEvenenment())
  Sleep(TEMPS_DE_SOMMEIL);
};
	void MonISR_A(void)
	{
	  ExecuteTacheA();
	}
	…
	void MonISR_Z(void)
	{
	  ExecuteTacheZ();
	}

Une façon traditionnelle de sortir un microprocesseur du sommeil est une interruption d'un périphérique (timer, par exemple!) qui n'est pas désactivé lorsque le microprocesseur est mis en veille.

4 Système d'exploitation temps réel (RTOS)

4.1 Introduction

Un système d'exploitation temps réel est un système d'exploitation qui exécute des tâches à l'intérieur d'un temps précis. Il dispose d'un certain temps pour répondre à des évènements extérieur et le RTOS sera fiable s'il répond toujours dans les délais prescrits.

Un RTOS est un système d'exploitation conçu pour rencontrer des échéances strictes

Les tâches planifiées par un RTOS peuvent être planifiée en fonction d'évènements ou en fonction d'une horloge. Les RTOS qui changent de tâche en fonction d'une horloge changent parfois trop de tâche ou perdent du temps à changer de tâche. Toutefois, ils donnent l'impression que chaque tâche a le microprocesseur pour elle seule et leur mécanisme de réponse à un évènement est plus facilement prévisible que les RTOS dépendant des évènements (la séquence d'évènement précédant un évènement X peut changer beaucoup!).

On retrouve event-driven RTOS et time-sharing RTOS dans la littérature pour qualifier les RTOS en fonction de la façon dont ils planifient les tâches.

Les évènements susceptibles d'appeler le scheduler du RTOS sont divers: fin d'une tâche, tâche bloquée, interruption d'un périphérique ou d'un timer, tâche coopérative qui appelle le scheduler elle-même, etc.

Dans un RTOS, les performances sont mesurées en fonctions de la capacité à respecter les contraintes temporelles. Elles ne se mesurent pas en termes de nombre de tâches exécutées par unité de temps ou vitesse d'exécution moyenne de l'ensemble des tâches ou avec d'autres critères propres aux systèmes d'exploitation de base. Pour respecter les contraintes temporelles, plusieurs stratégies sont employées:

- Réduire le nombre de tâches exécutées par le RTOS
- Annuler, réduire ou reporter certaines fonctions qui ne peuvent pas être exécutées sans mettre en danger le respect des échéances
- Lire les entrées et détecter les évènements souvent et rapidement
- Favoriser les tâches qui doivent être exécutées en temps-réel lors de l'allocation des ressources.
- Garder en réserve des ressources du système afin de répondre rapidement à un évènement lorsqu'utiliser ces ressources pour les tâches non temps-réel pourrait compromettre le temps de réponse. Refuser des tâches non temps-réel au besoin
- …

On distingue trois catégories de système temps-réel : les systèmes temps-réel durs (Hard real-time), les systèmes temps-réel soft (Soft real-time) et les systèmes temps-réel fermes (firm real-time). Dans un système temps-réel dur, le non-respect d'une échéance est vu comme un échec ; dans un système temps-réel soft, le non-respect d'une échéance en

moyenne est vu comme un échec ; dans un système temps-réel ferme, il y a des contraintes softs et des contraintes dures.

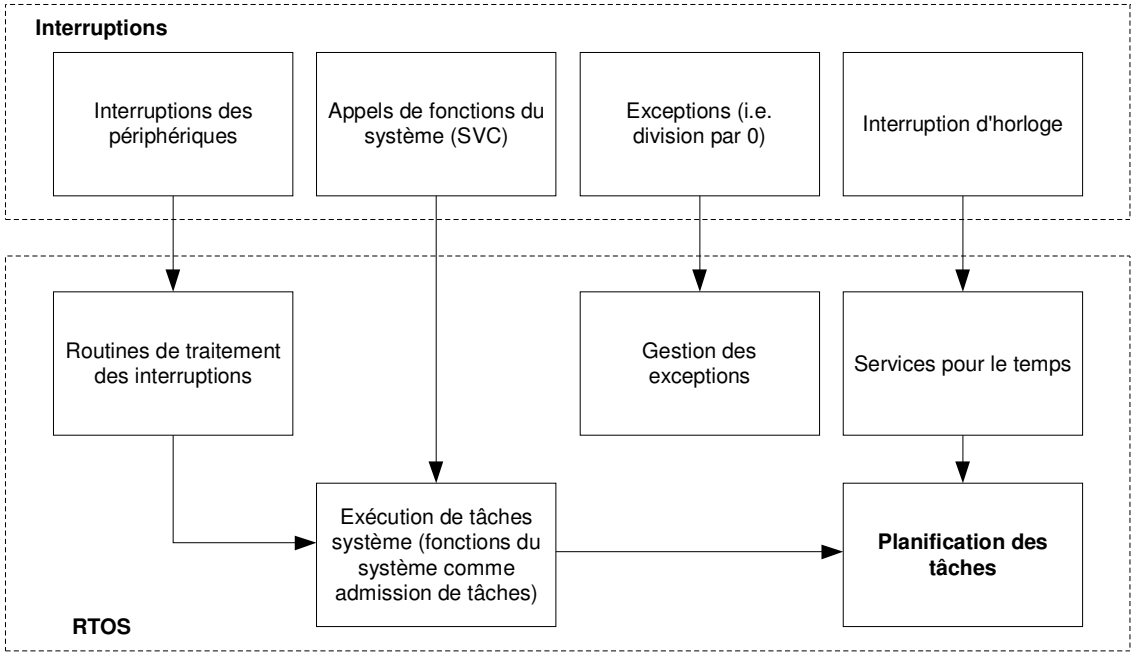
> *Les sections 1.4 et 1.5 de http://www.freertos.org/implementation/index.html seront présentées à titre d'exemple de tâche temps-réel et des échéances imposées au système d'exploitation.*

Dans un système microprocesseur embarqué, le noyau du système d'exploitation est un noyau réduit par rapport au noyau d'un ordinateur: par définition, un système embarqué est un système où les contraintes de mémoire sont importantes. Souvent, le noyau d'un RTOS comprend le démarrage du système, la planification des tâches et quelques fonctions, et encore. Les rares fonctions du RTOS, s'il y en a, doivent être petites et importantes afin de minimiser les ressources utilisées par le RTOS lui-même.

Par ailleurs, les applications embarquées ont souvent plus de libertés que les applications sur ordinateur. Par exemple, les applications embarquées sont souvent exécutées en mode kernel pour leur laisser un plus grand contrôle des ressources du système. Autre exemple : les applications embarquées peuvent parfois désactiver les interruptions alors que ce n'est pas possible sous Windows…

Les tâches d'un RTOS sont similaires à celles d'un OS normal, mais souvent implémentée différemment:

- Initialiser le système
- Admission des tâches
- Gestion de la mémoire
- Gestion des périphériques et de leurs interruptions
- Gestion des exceptions
- Gestion des ressources et synchronisation des tâches
- Gestion du temps : fournir une base de temps aux tâches du système
- Protection de la mémoire et librairies de fonctions pour l'accès aux périphériques
- **Planification des tâches**



4.2 Initialisation du système

Dans PC, un programme en mémoire non-volatile (BIOS) charge le système d'exploitation qui initie l'ordinateur. Dans la plupart des systèmes embarqués, les premières instructions exécutées par le microprocesseur sont des instructions du système d'exploitation.

Les premières instructions du système d'exploitation testent le système microprocesseur. Elles vérifient la mémoire et les périphériques, configurent les interruptions, configurent les périphériques et initialisent les variables qui gèrent le système.

Souvent, les initialisations des systèmes embarqués ressemblent à :

```
void main(void)
{
        DisableInterrupts();
        InitMemory();

        InitInterrupts();

        InitTimer0();
        InitTimer1();
        InitSPI();
        …
        InitUART();

        InitOS(); //InitScheduler();
```

```
        EnableInterrupts();

        while(1)
        {};
}
```

Souvent le premier code exécuté par le microprocesseur lors de la mise sous tension est un code écrit en assembleur qui configure certains registres de base du microcontrôleur (initialisation de la table des vecteurs d'interruption par exemple), effectue certaines tâches reliées à la mémoire et à la compilation (mettre à 0 tout le contenu de la mémoire RAM qui n'est pas no init par exemple), puis fait un branchement vers le main…

4.3 Admission des tâches

Un RTOS gère des Task Control Blocks (TCBs) qui sont analogues aux Process Control Control Blocks (PCBs) des systèmes d'exploitation conventionnels. Le RTOS doit inclure des fonctions permettant d'allouer de la mémoire pour les TCBs, de créer les TCBs, d'utiliser les TCBs (lire l'état d'une tâche par exemple) et de détruire les TCBs.

Le document RTOS_Processus et GPOS.ppt sera brièvement présenté en classe pour un rappel sur les PCBs (page 9) et sur l'admission des processus (page 14).

Dans un RTOS, la création de TCBs doit être optimisée pour les tâches temps-réel : si allouer de la mémoire et des ressources à la tâche est trop long, il est possible d'entraîner un échec du système…

4.4 Gestion de la mémoire

S'il manque de temps, cette section sera vue dans le cours Mémoire-Logiciel.

Le document RTOS_Mémoire et GPOS.ppt sera brièvement présenté en classe pour un rappel sur les stratégies d'allocation de mémoire dans les systèmes d'exploitation généraux.

Allocation de mémoire pour les tâches

Les systèmes d'exploitation embarqués allouent souvent des segments de mémoires contigus de taille variable ou fixe. Les stratégies d'allocation de mémoire non contigüe pour les tâches (comme les tables de page) sont souvent trop longues/complexes ou elles occupent trop d'espace mémoire (code et données) pour être réalisables.

En raison des contraintes de temps, la stratégie d'allocation de la mémoire la plus souvent employée dans les RTOS est le First Fit. Des méthodes comme le best fit sont trop longues…

Par ailleurs, les tâches avec contraintes dures (temps réel dure) ou même soft doivent être exécutées très rapidement. Pour cette raison, la mémoire allouée aux tâches temps réel est

souvent allouée définitivement (pas de swapping) et la translation d'adresse pour ces tâches est souvent minimaliste, voire évitée.

Allocation de mémoire pour les variables créées dynamiquement
L'allocation dynamique de mémoire (new, malloc) doit également être rapide pour les RTOS. De ce fait, on gère souvent le monceau en blocs de taille fixe pour accélérer le processus d'allocation, quitte à perdre de l'espace mémoire.

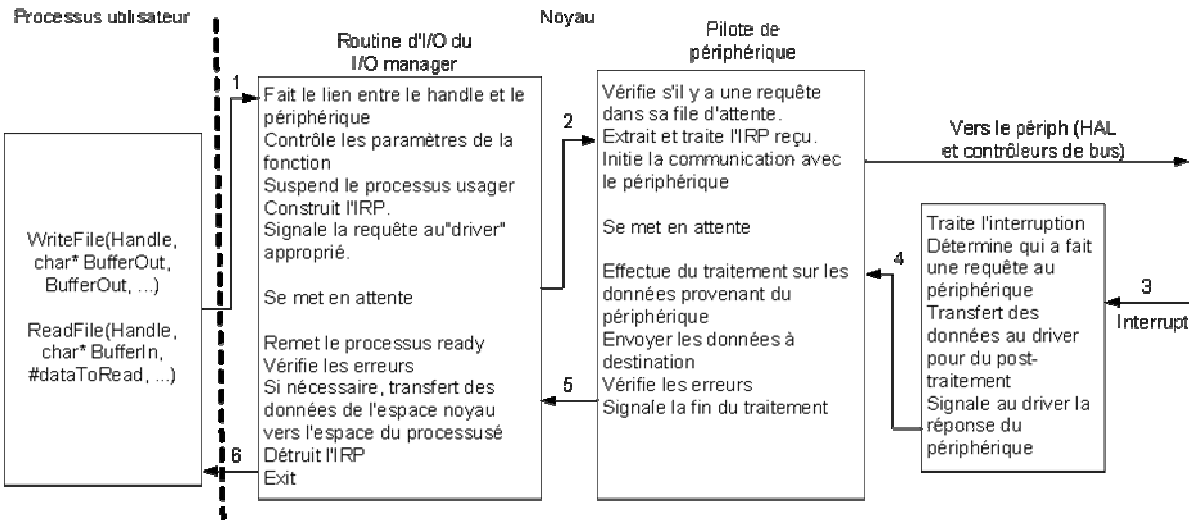
4.5 Périphériques et gestion des interruptions

Tous les périphériques du système sont susceptibles de générer des interruptions. Par exemple, il est possible de générer une interruption lorsqu'un octet est reçu du port SPI.

Les routines traitant les interruptions des périphériques sont généralement des routines du système d'exploitation. Lors d'une interruption, le système d'exploitation peut changer l'état d'une tâche et lui communiquer les informations appropriées.

La figure suivante, extraite des notes de cours de systèmes d'exploitation illustre le traitement des requêtes d'entrées/sorties dans un système d'exploitation conventionnel:

1) Un programme usager appelle une routine d'interruption du système d'exploitation pour s'adresser à un périphérique.
2) Cette routine fait une requête vers un pilote de périphérique (construit un I/o Request Packet –IRP- et appelle une fonction du pilote)
3) Le pilote de périphérique envoie des requêtes au module d'E/S du périphérique et attend une interruption de ce dernier.
4) Une routine-ISR du système d'exploitation traite l'interruption et appelle une fonction du pilote de périphérique
5) La routine d'E/S effectue un post-traitement sur les données provenant du périphérique et permet l'exécution du processus.



Un pilote de périphérique est un ensemble de fonctions appelées par le système d'exploitation lorsque se produit des évènements prédéterminés. Il ne s'agit pas d'un programme, mais de plusieurs routines exécutées lorsqu'un périphérique est détecté, retiré, mis en veille, adressé, etc.

Dans un RTOS, il arrive que plusieurs de ces étapes, voire toutes ces étapes, soient court-circuitées : le pilote de périphérique peut être absent (la routine d'E/S s'adresse directement au module d'E/S du périphérique), le processus peut s'adresser directement au périphérique et il est même possible d'exécuter des tâches à l'intérieur des interruptions. La gestion des périphériques dépend des tâches temps réels qui doivent être exécutées.

Habituellement, les interruptions des périphériques ont une priorité plus basse que l'interruption du système d'exploitation, mais une priorité plus haute que la tâche exécutée. Si la tâche exécutée est critique et s'il faut l'exécuter avant un certain temps, les interruptions peuvent causer un échec du système: ils peuvent interrompre la tâche critique en cours suffisamment longtemps pour que l'échéance ne soit pas respecter. Pour cette raison, les routines traitant les interruptions doivent toujours être courtes. Habituellement, la tâche exécutée lors d'une interruption est réduite au minimum. Pour reprendre l'exemple de l'octet du SPI ci-dessus, cela pourrait être: acquitter l'interruption, mettre un octet/message reçu dans un buffer, puis lever un drapeau de telle sorte qu'une tâche de moins haute priorité traite l'octet reçu plus tard.

De manière générale, avoir des routines traitant les interruptions le plus court possible est une bonne pratique de programmation embarquée. Cela est nécessaire pour plusieurs raisons: faciliter le partage du temps de CPU, éviter de perdre des interruptions si une interruption se produit à plusieurs reprises pendant le traitement de l'interruption, éviter d'affamer certaines tâches du système, ne pas ralentir les autres interruptions de même priorité…

Les interruptions sont aussi susceptibles de modifier des données du système d'exploitation ou d'appeler de fonctions de celui-ci. Dans ces cas, il est possible d'écrire une variable à partir de deux contextes différent (celui des tâches background et celui des interruptions; ou ceux de deux interruptions) ou encore, d'appeler des fonctions du RTOS qui ne sont pas nécessairement réentrantes. Il existe deux stratégies pour éviter ces erreurs:

1) Approche unifiée: Désactiver les interruptions quand les informations du RTOS sont modifiées (par une tâche ou par une interruption), en espérant ne pas perdre d'interruption pendant la désactivation ou ne pas retarder trop l'interruption du scheduler.
2) Approche segmentée: Tous les accès aux variables du RTOS ou appel de fonction du RTOS se font dans une interruption basse priorité (mais plus haute priorité que la tâche en cours…). Le travail effectué dans une interruption qui modifie les données du RTOS sera segmenté : une partie sera faite dans l'interruption elle-

même et une autre partie, dans une ISR de plus basse priorité. Le travail effectué dans une tâche qui modifie les données du RTOS sera également segmenté.

Si le temps le permet, une comparaison des deux approches basées sur le site http://rtos.com/articles/18835 sera présentée en classe.

4.6 Gestion des exceptions

Un système d'exploitation doit traiter convenablement les exceptions qui se produisent lors de l'exécution des tâches. Ces exceptions peuvent être de plusieurs natures et sont habituellement des interruptions synchrones.

Une interruption synchrone est une interruption synchronisée avec le code : elle se produit toujours au même endroit lorsqu'une séquence d'instruction est exécutée.

Quelques exemples d'exceptions propres aux RTOS ou aux systèmes embarqués : instruction invalide, accès mémoire invalide (exemple : adresse incorrecte), faute de protection de mémoire, non-respect d'une contrainte temps-réel dure, détection d'un deadlock, timeout général, …

Il y a plusieurs comportements possibles lors d'une exception. Habituellement, un reset du microprocesseur est effectué (avec un watchdog, l'éxécution d'instructions est simplement arrêtée…). Il est aussi possible d'arrêter ou recommencer la tâche ayant causé l'exception si elle n'est pas temps-réel ou autre stratégie.

4.7 Programmation Concurrente

Le système d'exploitation gère les ressources du système microprocesseur. Il doit attribuer ces ressources aux diverses tâches à exécuter en fonction de leurs besoins et de leur priorité.

Comme les ressources sont limitées, il peut arriver que des tâches entrent en compétition pour obtenir une ressource. Il faut une stratégie pour allouer les ressources à chaque tâche.

Le terme ressource inclue des variables aussi bien que des périphériques…

Le système d'exploitation doit aussi faciliter les échanges de données entre les diverses tâches du système.

Enfin, le système d'exploitation doit synchroniser les diverses tâches du système.

4.7.1 Section Critique

Une section critique est une ressource (mémoire ou périphérique) partagée par plusieurs tâches qui ne peut être utilisée que par une seule tâche à la fois. L'utilisation de la ressource doit se faire de façon atomique.

Le mot atomique est employé dans le sens indivisible: on ne peut pas diviser la passage dans une section critique en deux!

Dans la vie courante, un exemple de section critique serait un pont à une seule voie : il s'agit d'une ressource que chaque automobiliste peut emprunter, mais qu'une seule voiture peut traverser à la fois. Lorsqu'une voiture s'accapare la ressource, les autres voitures ne peuvent y accéder.

En programmation, les sections critiques sont des espaces mémoires partagée entre plusieurs tâches (ou entre des tâches background et des interruptions) ou, encore, des accès à des périphériques qui doivent se faire en plusieurs étapes consécutives et bien définies.

4.7.2 Mutex et Sémaphores

Les sémaphores sont des variables qui indiquent si une tâche a pris le contrôle d'une section critique.

Les sémaphores fonctionnent avec deux opérations : 1) Tester la sémaphore et obtenir la ressource si elle est disponible (opération P), 2) Libérer la ressource (opération V). Lorsqu'on utilise des sémaphores, toutes les tâches voulant entrer dans une section critique vérifie si la ressource est disponible en faisant l'opération P. Si la ressource est disponible, la tâche s'exécute et libère la ressource en faisant l'opération V lorsque terminée.

Il y a deux types de sémaphores : les sémaphores binaires et les sémaphores compteurs.

Les sémaphores binaires, qui sont analogues aux MUTEX (mutual exlusion), fonctionnent de manière booléenne. Elles déterminent si on peut accéder à une ressource ou non.

Exemple d'utilisation d'un sémaphore binaire :

Tache N

```
…
bool PeutContinuerTache = 0 ;

while(PeutContinuerTache == 0)
{
        DisableInterrupt() ;
        if(SemaphoreRessourceUtilisée == 0)
        {
                SemaphoreRessourceUtilisée == 1; //Opération P
                EnableInterrupt() ;
                PeutContinuerTache = 1 ;
        }
        else
```

```
        {
                EnableInterrupt() ;
                IndiqueTacheBloquéeAuOS();
        }
}

UtiliseRessource() ;
SemaphoreRessourceUtilisée = 0;     //Opération V
…
```

L'exemple ci-dessus est un spin lock: on tourne en rond jusqu'à ce que la tâche ne soit plus verrouillée. En français, on parle d'attente active: le processus attend jusqu'à ce que la ressource soit disponible. Dans les systèmes d'exploitation généraux où le % d'utilisation du CPU peut être un critère de performances, l'attente active est inadmissible : il faut permettre l'exécution d'autres tâches par le CPU pendant l'attente. Aussi, la fonction IndiqueTacheBloquéeAuOS() *devrait, dans les faits, changer l'état du processus et appeler le scheduler. Dans les RTOS, l'attente active est aussi un gaspillage de ressources, toutefois, elle peut être requise afin de rencontrer les échéances/contraintes temps-réel.*

Les sémaphores compteurs intègrent des compteurs et des pointeurs sur des tâches. À toutes les fois qu'une tâche requiert la ressource (opération P), le compteur est incrémenté (ou décrémenté en fonction de l'implémentation), la tâche est mise dans une file d'attente pour la ressource et la tâche est bloquée. Lorsque la ressource est octroyée à une tâche, celle-ci s'exécute et libère la ressource à la fin (opération V). Lors de l'opération V, le compteur est décrémenté et la ressource peut être octroyée à une autre tâche dans la file d'attente du sémaphore.

Les sémaphores binaires sont un cas limite des sémaphores compteurs : on ne compte qu'une seule fois… Utiliser uniquement des sémaphores compteurs uniformise le code, mais l'utilisation de celles-ci prend plus de ressources (espace mémoire et temps de CPU) que l'utilisation des sémaphores binaires et c'est pourquoi on retrouve surtout des sémaphores binaires dans les systèmes embarqués.

4.7.3 Passage de Messages

Une autre stratégie permettant de distribuer les ressources aux diverses tâches est le passage de message. Dans le passage de message, les ressources sont gérer par une seule tâche et il faut envoyer des messages à cette tâche si on veut obtenir la ressource…

4.7.4 Inversions de priorité et Deadlock

Deux problèmes fréquents surgissent régulièrement lorsque le système d'exploitation attribue des ressources à différentes tâches : l'inversion de priorité et les deadlock.

L'inversion de priorité survient lorsqu'une ressource requise pour deux tâches a d'abord été réservée par la tâche de basse priorité qui a été lancée en premier. La deuxième tâche,

haute priorité, est lancée prioritairement, mais elle devient bloquée parce qu'elle attend après la ressource.

Un cas typique d'inversion des priorités implique trois tâches : A1, B2, C3 où A1 est plus prioritaire que B2 qui est plus prioritaire que C3 et où A1 et C3 ont une ressource commune. La séquence d'évènement suivant conduira à une inversion de priorité :

- C3 est la première tâche exécutée et C3 s'empare de la ressource commune.
- B2 est la seconde tâche exécutée. Comme B2 est plus prioritaire que C3, B2 est exécutée et C3 attends du temps de microprocesseur.
- A1 est exécutée, mais A1 est bloquée parce que la ressource commune est utilisée par C3. B2 continue d'être exécutée, même si sa priorité est plus basse que celle de A1.

Il y a plusieurs stratégies pour contrer l'inversion de priorité. Dans le cadre du cours, je ne donnerai qu'un exemple (vous verrez plus de détails dans le cours sur les OS !): augmenter la priorité d'un processus lorsqu'il partage une ressource avec une tâche plus prioritaire.

Le Deadlock survient lorsque deux ou plusieurs tâches différentes partagent un ensemble de ressources qui sont distribuées parmi toutes les tâches de telle sorte que toutes les tâches sont bloquées : elles ne peuvent pas libérer les ressources qu'elles ont afin d'être terminée, mais elles attendent après les ressources des autres tâches pour finir. L'exemple le plus simple de Deadlock est deux tâches (1 et 2) utilisant toutes deux les ressources A et B. Si la tâche 1 réserve la ressource A et que la tâche 2 réserve la ressource B, puis si la tâche 1 attend après la ressource B et la tâche 2 attend après la ressource A, il y aura un Deadlock.

Il y a plusieurs stratégies pour contrer les Deadlock. Dans le cadre du cours, je ne donnerai que deux exemples (vous verrez plus de détails dans le cours sur les OS !): allouer une seule ressource à la fois à une tâche et toujours faire le verrouillage des ressources dans le même ordre.

4.7.5 Désactivation temporaire des interruptions

La désactivation temporaire des interruptions permet à un processus de garder le microprocesseur pour lui-même tant qu'il le désire. Il peut ainsi prendre une ressource, en faire usage et relâcher la ressource sans problème de Dead lock ou d'inversion de priorité.

Cependant, désactiver les interruptions avec un RTOS est une pratique à éviter le plus possible: si vous désactiver les interruptions longtemps, vous retardez d'autant le temps de réponse du système à un évènement et vous pouvez, de ce fait, causer des échecs du système. Habituellement, on ne désactive les interruptions que pour tester un drapeau et on les réactive aussitôt :

```
//Exemple de cas où la désactivation des interruptions est brève et pertinente…
```

```
DisableInterrupts()
If(RessourceIsAvailable) //Lecture d'une sémaphore binaire!
{
 GetRessource(); //Écriture d'une sémaphore binaire!
 EnableInterrupts();
 …
}
EnableInterrupts();
```

4.8 Gestion du temps pour les tâches

Le système d'exploitation doit fournir des fonctions ou des variables permettant aux tâches de gérer le temps. Ces fonctions ou variables vont du simple Tick d'horloge (un compteur incrémenté à toutes les unités de temps, dont l'unité de temps dépend de l'horloge du système et parfois de l'interruption du système d'exploitation) à une interface détaillée pour créer des timers.

Le système d'exploitation doit aussi effectuer certaines tâches selon le temps écoulé : gérer des timers (si ce service est offert par le RTOS), réveiller des tâches en attente de temps, et plus…

Voici un exemple de tâches parfois réalisée dans l'interruption d'horloge d'un RTOS:

- Sauvegarder le contexte de la tâche en cours d'exécution
- Augmenter le temps du système (Tick) de 1
- Mettre à jour les timers
- Réveiller les tâches (périodiques) qui attendent si c'est l'heure
- Planifier une nouvelle tâche (!)
- Autres actions exécutées pendant l'interruption :
    - Ôter les tâches terminées
    - Vérifier l'état du système
    - Nourrir le watchdog
    - …
- Lire le contexte de la tâche planifiée

4.9 Protection de la mémoire et librairies de fonctions pour l'accès aux périphériques

Dans plusieurs systèmes d'exploitation, les programmes de l'utilisateur n'ont pas directement accès aux ressources du système : ils doivent appeler des services du système d'exploitation pour y accéder. La plupart des systèmes d'exploitation offrent donc des services et des librairies de fonctions aux programmes de l'utilisateur.

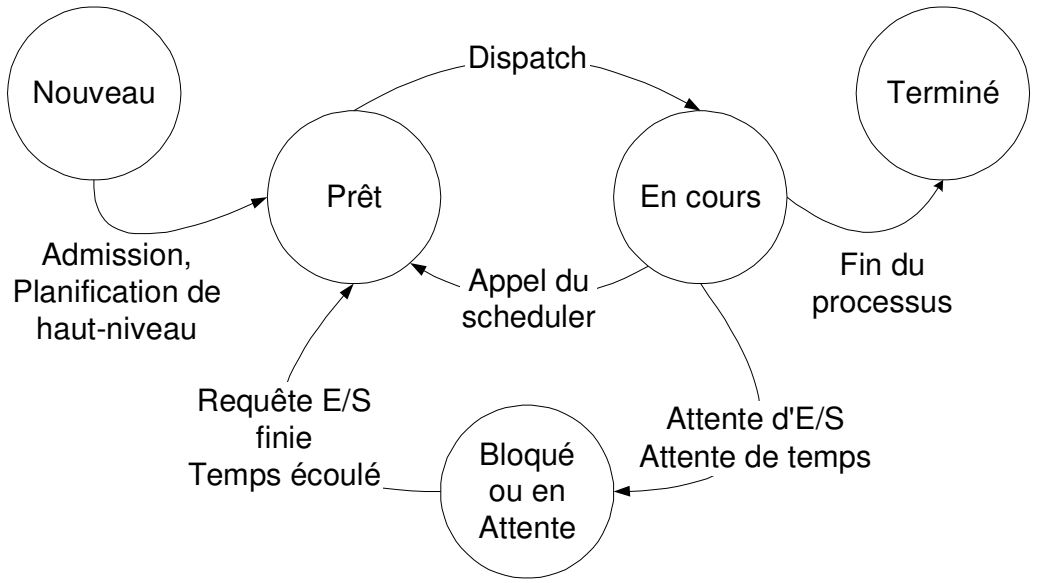
Les fonctions offertes par le système d'exploitation sont souvent accessibles sous la forme d'interruptions logicielles. On parle de SVC (service call) quand un programme de l'utilisateur appelle une routine d'interruption du système d'exploitation afin d'accéder à une ressource (lire périphérique) du système.

4.10 Planification de tâches (Ordonnacement, scheduling)

Le RTOS doit décider quelle tâche il exécutera dans le prochain intervalle de temps. Pour se faire, il se basera sur l'état de chaque tâche et un algorithme de décision afin de déterminer quelle tâche accomplir.

Un système d'exploitation préemptif sauvegardera toujours le contexte (TCB) de la tâche en cours d'exécution avant de passer à une autre tâche. Il choisira ensuite la prochaine à exécuter, puis chargera le contexte de la prochaine tâche à exécuter.

4.10.1 État des tâches



Une tâche peut être dans trois états possibles:

- En cours d'exécution (running)
  - processus OK, processeur OK
- Prêt (suspendu provisoirement pour qu'un autre processus s'exécute, ready)
  - processus OK, processeur occupé
- Bloqué ou en attente (attendant un événement ou un certain temps)
  - processus non OK, même si processeur OK

4.10.2 Planification de tâches

Certains concepts de *RTOS_Processus et GPOS.ppt (page 14)* et *Cours-10-1.pdf* seront brièvement revus en classe.

Dans un RTOS, les algorithmes de planification des tâches diffèrent souvent de ceux d'un système d'exploitation normal : les objectifs et critères de planification ne sont pas les mêmes. Par exemple, l'algorithme du round-robin (tourniquet, ou des variantes) est fréquemment utilisé pour les OS normaux, l'algorithme d'assignation de priorités fixes est utilisé à la fois pour les RTOS et pour les OS généraux et certains algorithmes sont spécifiques aux RTOS ou aux systèmes embarqués:

Earliest deadline: La tâche avec la contrainte la plus proche est exécutée.

Least Laxity: La tâche pour laquelle il reste le moins de temps est exécutée (le temps restant est le temps de l'échéance à respecter auquel on soustrait le temps de la tâchee et le temps actuel.

Maximum urgency: un mélange de Earliest Deadline et Least Laxity. Une variable booléenne est associé à chaque tâche et dit si elle est urgente ou non. Les tâches urgentes sont exécutées avec l'algo de Least Laxity et les tâches non-urgentes, avec le Earliest deadline.
Priorité proportionnelle à la fréquence (rate monotonique priority): la tâche exécuté le plus souvent est la tâche la plus prioritaire.

4.10.3 Sauvegarde et lecture de contexte

La sauvegarde d'un contexte consiste à sauvegarder les informations du TCBs qui varient d'une tâche à l'autre :

- Compteur de programme
- Registres
- Pointeur de pile de la tâche
- Drapeaux
- État de la tâche

Habituellement la sauvegarde de contexte se fait sur une pile réservée ou sur un espace mémoire utilisé comme une pile à l'intérieur du PCB. Une pile apparaît comme une structure de donnée naturelle afin de sauvegarder les contextes.

Un avantage de sauvegarder le contexte d'une tâche avec une pile est d'exploiter les mécanismes mis en place pour les interruptions (on doit sauvegarder le contexte du main…) ou pour les appels de fonctions afin d'effectuer une sauvegarde de contexte efficace.

La lecture de contexte est l'opération inverse de la sauvegarde de contexte : les registres de la tâche à exécuter sont restaurés, puis l'exécution de l'exécution de la tâche se poursuit en restaurant le compteur de programme.

4.10.4 Latence des interruptions et temps de changement de contexte

Dans tous les systèmes multitâches et particulièrement dans les systèmes où une interruption d'horloge appelle la sélection/planification d'une nouvelle tâche, la latence des interruptions et le temps de changement de contexte (sauvegarder le contexte de la tâche remplacée et charger le contexte de la nouvelle tâche) doivent être le plus court possible. En effet, il faut limiter la proportion de temps de microprocesseur utilisée pour planifier les tâches…

La latence des interruptions est le temps pris pour entrer dans une interruption. Lors d'une interruption, le microprocesseur sauvegarde automatiquement, au minimum,

l'adresse de retour sur la pile et, habituellement, les drapeaux accompagnés de certains registres. Ces opérations doivent être le plus courte possible afin d'éviter de perdre du temps dans les interruptions.

Le temps pour changer de contexte est également critique. Ce temps est également retrancher du temps disponible pour répondre à un évènement à l'intérieur d'un délai imposé. Pour réduire le temps de changement de contexte, il faut minimiser le temps de sauvegarde du contexte (créer des instructions PUSH multiple registers par exemple), minimiser le temps de sélection de la prochaine tâche à effectuer (en les triant par ordre de priorité par exemple) et minimiser le temps de chargement de contexte (créer des instructions POP multiple registers par exemple).

La latence des interruptions et le temps de changement de contexte ont été grandement améliorés au cours des dernières décennies. Sur environ 25 ans : les vitesses d'horloge ont augmenté d'un facteur 5 à 20 dans l'embarqué, le nombre d'instruction par coup d'horloge a augmenté d'un facteur 5 à 10, du matériel additionnel a permis une accélération de ces tâches (encore d'un facteur 10) et les stratégies logicielles se sont améliorées. Bref, de nos jours, le temps de changement de contexte est de l'ordre de quelques microsecondes (us) ou moins alors qu'il était de quelques millisecondes il y a 25 ans.

4.11 Exemple d'implémentation

Si le temps le permet, un RTOS pour le LM3S9B92, le FreeRTOS sera montré en classe. Il est possible de télécharger le code source de ce système d'exploitation à partir du lien suivant : http://focus.ti.com/docs/toolsw/folders/print/eks-lm3s9b92.html.