TSM_AdvEmbSoft

Scheduling

Static Scheduling

P_x : période de la tâche

D_x : deadline ou délai auquel la tâche doit être faite

C_x : pire temps d'éxecutiton ($C_x < P_x$) le temps le plus long qu'une tâche met à s'éxecuter

Ce n'est pas toujours possible de trouver un agencement de tâche qui permet de tout accomplir dans les temps. Cet arrangement est défini "offline" par le programmeur:

- Définir le plus petit cycle comme le plus petit multiple commun des P_x
- Est-ce que c'est possible de construire une time table où toutes les tâches pourront être effectuée durant ce cycle? Si oui faire la table
- Faire la table de manière cyclique
- Certaines tâches peuvent se découper en plusieurs tâches
- On peut ajouter un temps mort pour arriver au bon temps de cycle

Les différentes tâches sont appelé dans la boucle while(1) qui ne s'arrête jamais

Avantages :

- Facile à implémenter

Désavantages :

- La table peut être très difficile à trouver
- Le système fait toujours la même chose peut importe les conditions
- Le délai maximum et un temps de cycle
- Le programme doit continuellement tester le statut de chaque device
- Cette approche est très difficilement scalable

Système basé sur les évenements

Les évenements sont générés par une source et reconnus par un récepteur

Ils peuvent arriver de manière prévisible ou non

La super boucle n'est pas capable de gérer les évenements non prévisibles

Un système basé sur les évenements permet d'éviter de checker continuellement les entrées

Gérer les évenements se fait avec les interruptions

- C'est efficace : tourne uniquement quand un évenement est présent
- C'est rapide : c'est un méchanisme hardware
- C'est scalable

Déroulement d'une interruption

Les processeurs actuels permettent de

- Prioriser certaines interruptions
- Masquer certaines interruptions
- Interruptions d'une interruptions (advanced player only)
- On peut choisir l'endroit dans la mémoire ou sera l'interruption

Règles :

- Interruption la plus courte possible
- Pas d'actions bloquante (comme accéder à un Mutex)

Scheduling Dynamic

Les tâches sont réagencée durant exécution

C'est basé sur la priorité (ou parfois la durée)

Une tâche moins prioritaire ne fera pas attendre une plus prioritaire

2 principaux types

- Réagencement à la fin d'une tâche uniquement : Run-To-Completion (RTC)
- Réagencement en tout temps : préemptif

RTC

Cela peut-être acceptable dans certains cas mais on peut facilement imaginer des scénarios où c'est problématique

Le scheduler choisit la tâche à effectuer parmi celle "Ready" et selon leur priorité

Un évenement peut changer l'état d'une tâche

Règles

- Si aucune tâche n'est prête, le scheduler reste en idle
- Si aucune tâche est "Running", le scheduler commence la tâche Ready la plus prioritaire
- Une fois qu'une tâche est commencée, elle se termine de toute façon
- Une tâche complétée entre en mode waiting jusqu'à ce qu'elle redevient ready

"""Personne n'écrit soit-même un scheduler"""

Dynamic premptive scheduler

Premption :

- Le scheduler arrête sa tâche pour aller à une autre

Règles

- Une activité de tâche peut être mise en attente (bloquée)
- Une tâche qui attend ne revient pas au CPU. Elle doit être signalée par une ISR ou une autre tâche
- Seul le scheduler déplace les tâches entre ready et running
- Sinon : round-robin entre les tâches de même priorités

Comparaison du temps de réponse

La préemption offre de meilleurs temps de réponses

- On peut faire plus de processing
- On peut baisser la vitesse du processeur pour économiser de l'argent et de l'énergie

Préemption est compliqué à coder (ça tombe bien on va pas le faire)

La préemption amène des vulnérabilités (Mutex, sémaphore à utiliser)

Définir et agencer des tâches est une features importantes des RTOS

Différence de besoin en RAM :

Comparaison de performances de scheduler

Difficile voir impossible de faire des comparaisons définies

Il faut à la fois des analyses théoriques et des mesures empiriques

- Analyse théorique : théorie des files, modélisation
- Mesures empiriques : avec de la simulation ou des mesures sur des systèmes existants

Mesures de performances

- Temps d'attente : temps perdu par les tâches à attendre lorsqu'elles sont ready
- Temps turnaround : temps pris de la soumission de la tâche à sa complétion
- Utilisation du CPU
- Throughput : nombre de tâches exécutée par unités de temps
- Temps de réponse
- Equitabilité entre tâches

Mbed OS EventQueue

Pas préemptif

Mécanisme simple et puissant pour une boucle d'événements

Les tâches périodiques peuvent être posté à une queue

La queue peut être utilisé pour dire qu'un évènement doit être fait suite à une interrupiton

Les évènements sont dispatché par un thread

Tâches et concurrences

Organiser un code embarqué en plusieurs tâches

Ce n'est pas souvent qu'un programme peut tenir dans une seule boucle.

Le code doit être découper en petits éléments tel que :

- Le code est lisible, structuré et documenté
- Le code peut-être testé modulairement
- Le développement réutilise des codes existants pour diminuer le temps de développement

- Le design du code permet à plusieurs de travailler sur le même projet
- Les améliorations peuvent être implémentés efficacement

=> Organiser ces éléments de codes en classes et méthodes

Les systèmes embarqués doivent répondre dans un temps imparti à des évènements. Cela veut dire qu'on doit pouvoir :

- Mesurer des temps
- Générer des évènements basé sur le temps, un seul ou répétitif
- Répondre rapidement à des évènements imprédictibles

Les différentes activités sont appelées tâches

Dès qu'un programme a plusieurs tâches c'est du multi-tâches

2 catégories de tâches

- Time-triggered
- Event-triggered

Multitasking et OS

On utilise un OS pour écrire des applications multitâches (c'est finalement son travail)

Un OS fait :

Les services basiques d'un OS

Threads vs Process

Un process peur contenir plusieurs threads est n'est pas dispo sur notre cible mais sur linux par exemple

Multitasking vs Multiprocessing

Multitasking : ça tourne manière concurrente sur un monoprocesseur

Multiprocessing : plusieurs processeurs qui partagent le même bus, parfois le même clock, mémoire et périphériques

Real-Time Operating Systems Goals

Définition du temps-réel : Le temps réel garanti la complétion d'un process dans un intervalle de temps défini. Par contre il ne dit pas quel temps ça doit prendre, ça doit juste être toujours prédictible.

Un RTOS est un OS qui :

- Sert des applications temps-réelles
- Répond à des requêtes dans une délai garantit et prédictible
- Proccess les données dans un nombre déterministe de cycles

Un RTOS est plus pour des performance déterministiques, plutôt que un high troughput

- Les performances peuvent être mesurées par latence (temps qu'il prend)
- Les performances peuvent être mesurées par gigue (variation du temps qu'il prend)
- Soft RTOS : plus de gigue, généralement répond aux deadlines
- Hard RTOS : moins de gigue, répond aux deadlines de manière déterministe

Tâches et RTOS

Un RTOS propose une approche de développement de programme où le contrôle du CPU et de toutes les ressources du systèmes sont généré par l'OS

C'est l'OS qui décide quelle partie du programme est à exécuter, pendant combien de temps et comment elle va accéder au ressources du systèmes.

- L'OS amène aussi la communication et la synchronisation entre les différentes tâches
- Il contrôle aussi les ressources partagées par les tâches comme la mémoire ou les périphérique hardware

Un programme écrit pour un RTOS est structuré en tâche ou chaque tâche :

- est presque exécutée dans un thread ou process séparé (pas obligé)
- est écrite comme un module du programme qui se suffit à lui même
- peut être priorisée

Mbed OS/RTX Task Scheduling

Mbed OS RTOS est basé sur RTX5

- RTX5 est l'implémentation ARM de CMSIS-RTOS
- RTX5 implémente un scheduler préemptif à faible latence

Scheduling est étroitement lié avec le concept de threads

Les processeurs Cortex-M supportent 2 modes d'opération : Thread et Handler

- Entrer dans le mode Thread : sur un Reset ou au résultat d'un retour d'exception (code privilégié ou non)
- Entrer dans le mode Handler : au résultat d'un retour d'exception (code privilégié)

Les Handlers sont utilisé pour du scheduling faible latence (Pas de préemption entre ces mécanismes)

- SysTick_Handler (time-based scheduling)
- SVC_Handler (lock-based scheduling)
- PendSV_Handler (interrupt-based scheduling)

Les priorité sont configurées telles qu'aucune préemption apparait entre ces handler

- Pas besoin de section critique pour protéger le scheduler

C'est une combinaison de priorité et d'un scheduling de round-robin

- Round Robin pour les tâches de même priorités
- Basé sur la priorité pour les autres tâches

RTX/Mbed OS Scheduling Options

Scheduling Préemptif

- Chaque tâche a une priorité différente et va run jusqu'à ce qu'elle se fasse préempté ou qu'elle a atteint un appel bloquant de l'OS

Round-Robin scheduling

- Chaque tâche a la même priorité et va run pour une période fixe, ou une tranche de temps, ou jusqu'à ce qu'elle atteint un appel bloquant de l'OS
- Le quantum est déterminé dans RTX_Config.h
- Si le quantum est expiré, l'état va être changé à READY

Multi-tasking coopératif

- Chaque tâche à la même priorité et le Round-Robin est désactivé
- Chaque tâche va run jusqu'un appel bloquant de l'OS ou l'utilisation de ThisThread::yield() call.

Le scheduling par défaut pour RTX est le Round-Robin préemptif

Mbed OS Task Scheduling

Mbed OS/RTX Scheduling

Task/Context Switching

Le Thread Control Block (TCB) rend le context switching un peu plus simple

- Le scheduler va commencer ou terminer un process en conséquence
- Il va stocker les info nécessaire
    - Hardware registers
    - Program Counter
    - Etat de la mémoire, stack et heap
    - Etat
- Similairement charge les informations du process B

Le context switching prend du temps

- Plusieurs milliers de cycles (200-300 pour Cortex-M RTX)
- C'est une surcharge est un goulot d'étranglement
- Un support hardware est aussi requis

Mais le multitasking est faisable, bien qu'à un seul processus à la fois

Mbed OS/RTX Threads

Sur RTOS, les tâches sont souvent associées à des threads

- Pas de concept de process
- Multi-tasking = multiple threads

Mbed OS fournit un API pour les Thread avec tout la doc

- Un thread est une instance de la classe Thread. Il doit être créé et ensuite lancé
- Des threads peuvent être créés avec différentes propriétés
- Important : à l'initialisation du système, un thread spécial qui exécute la fonction main est créé

Mbed OS/RTX Thread States

Running

- Currently running
- Seul 1 thread peut être à cet état

Ready

- Ready to run sont dans l'état Ready
- Une fois que le Running Thread a terminé ou est en train d'attendre, le prochain thread avec la priorité maximale devient le running threead

Waiting/Blocked

- Attente d'un évènement qui se produit

Inactive/Terminated

- Pas créé ou terminé
- Ces threads ne consomment aucune ressource du système

Threads synchronization

Dans un système multi-tâches, les différentes tâches peuvent tenter d'obtenir la même ressource ou peuvent attendre l'apparition de différents évènements

Dans certains cas, un tâche donnée peut entrer dans un état Waiting ou Blocked

Il y a plusieurs Waiting states

Events

Les events sont utiles pour attendre certaines condtions

- Un thread attends pour une conditions spécifique
    - La condition peut être faite d'un ou plusieurs flags (AND/OR)
    - Attente avec timeout est possible
- Un autre thread set cette condition spécifique
- Pas d'attente active

Dans MbedOS, on peut gérer les events avec EventFlagsAPI

Mutex

Même sur un mono-processeur, il y a un besoin de protéger quand on partage des ressources

Les mutex contrôlent l'accès sur la ressource partagée

- Ils s'assurent qu'un seul thread peut avoir accès à une section de code crititque
- On doit être attentif au deadlock ou starvation
    - Faire attention quand on a plusieurs mute
    - Toujours redonner le mutex après utilisation

RTX/Mbed OS implémente schéma de priorité héritée

Dealing with Deadlock

Autruche

Conditions de Coffman

Prévention : si toutes les conditions de Coffman sont fausses

- Le Mutex est inévitable
- Préemption est inévitable
- Prévenir les conditions circulaire d'attente. Solution avec Dijkstra (chemin le plus court)

Eviter les deadlock

- Evaluer les chances de deadlock lors de l'allocation, accepté ou interdire selon
- Algorithme du Banquier (Dijkstra)

Détecter une deadlock : que faire avec une deadlock existante

- Si possible
  - Kill une partie ou tout les process deadloqué
- Préemption de ressource?
- Restart? Watchdog sur système embarqué

Semaphore

Notion de producteur consomateur

Un sémaphore gère l'accès des threads à un set de ressources partagées d'un certain type

- A la différence des mutex, un sémaphore peut contrôler l'accès à plusieurs ressources partagées
- Par exemple, une sémaphore active l'accès et la gestion d'un groupe de périphériques identiques

Queue/Mail

Une Queue (file en français) permet de "filer" des entiers/des pointeurs d'un thread producteur à un thread consommateur

Une Mail fonctionne comme une Queue, mais fournit en plus des de la mémoire pour des messages

Inversion de priorité

Qu'est-ce qui se passe si les tâches ne sont pas indépendantes, qu'elles partagent des ressources et qu'elles interagissent?

- Le scheduling des tâches est affecté

Un principal problème arrive : inversion de priorité

- Les tâches de hautes priorités sont bloquées par des tâches de plus faible priorité

Les principales sources de d'inversion de priorité

- Sections non préemptables

- Ressources partagées
- Synchronisation et exclusion mutuelle (mutex)

Dans chaque cas, le temps de réponse (latence) est modifié

Cas sans limite : La tâche 1 pourrait être indéfiniment pas exécutée si la tâche 9 est sans arrêt préemptée alors qu'elle tient une ressource.

Solution

Les tâches sont forcées de suivre certaines règles quand elles bloquent ou débloquent des mutex

Ces règles sont appelées Resource Access Protocols

- Il en existe plusieurs

Mbed Os scheduling algorithm?

- Mbed OS/RTX implémente le priority inheritance mechanism

Extrait de la doc RTX :
Pour empêcher les inversions de priorité, le RTX CMSIS-RTOS implémente une méthode d'héritage de priorité pour les Mutex. Un thread de priorité inférieure hérite de la priorité de tout thread de priorité supérieure qui attend avec osMutexWait sur une ressource partagée. Pendant que le thread de priorité supérieure est en état d'ATTENTE, le thread de priorité inférieure s'exécute avec la même priorité qu'un thread en attente de priorité supérieure. Lorsque le thread de priorité inférieure cesse de partager une ressource avec osMutexRelease, la priorité originale est à nouveau attribuée à ce thread.

Ressource Access Protocol

Il faut considérer

- Est-ce que l'inversion de priorité est bornée? Peut-on fixer un temps maximal
- Est-ce que le protocole évite les deadlock?
- Est-ce que le protocole évite des blocages non-nécessaire?
- Est-ce que c'est facile de calculer la limite supérieure du temps de blocage?
- Quel est le nombre maximal de blocage?
- Est-ce que c'est simple à implémenter?

Plusieurs protocoles sont présenté

Non Preemption Protocole (NPP)

Une tâche se voit attibuer la priorité la plus élevée si elle réussit à verrouiller une section critique

Une tâche reprend sa priorité quand elle relâche la section critique

Ceci évite l'inversion de priorité

Cela évite les deadlock et limite le nombre de blocage de toute tâche à 1

Mais cela permet aux tâches de basse priorité de bloquer celle de hautes incluant celles qui ne requièrent pas l'accès à la ressource partagée

Priority Inheritance Protocol (PIP)

Idée :

- A prend le mutex S
- B avec une priorité supérieure essaie et prendre S, et est bloquée S
- B transfère sa priorité à A (A reprend et tourne avec la priorité de B)

Comportement run time :

- A tout moment où une tâche de plus basse priorité bloque une tâche de plus haute priorité, elle hérite de la priorité de la tâche bloquée

C'est le protocole appliqué dans beacoup de RTOS dont Mbed OS/RTX

Le temps de blocage est limité par la longueur maximale (somme) des sections critique des tâches de basse priorité.

L'idée est d'élever la priorité de la tâche à plus faible priorité (si elle bloque une tâche de plus haute priorité) à la plus haute priorité des tâches bloquées

Et elle revient à sa priorité originelle quand elle quitte la section critique

Malheureusement les deadlock sont possibles avec ce protocole

Blocage en chaine

Highest Locker's Priority Protocol (HLP)

Idée : définir le ceiling ("plafond") $C(S)$ d'une sémaphore S pour être la plus haute priorité de toutes les tâches qui vont utiliser S durant l'exécution. Demande une analyse de code (offline) pour créer la table $C(S)$

A tout moment ou une tâche arrive à tenir un sémaphore S sa priorité va changer dynamiquement changé aux maximum entre sa priorité actuelle et $C(S)$

Quand elle a fini avec S elle revient à sa priorité originelle

C'est "Deadlock-free", une fois que la tâche 2 a S1, elle va tourner avec la priorité H, la tâche 1 va être bloquée et ne peut pas avoir S2 avant la tâche 2

Les tâches ne vont être bloqué au maximum une fois

Priority Ceilling Protocol

Idée : combiner le PIP et le HLP

- Supposons que S est le sémaphore avec le plafond le plus élevé verrouillé par d'autres tâches en cours
- Si une tâche A veut bloquer un sémaphore (pas forcément S), elle doit avoir une priorité strictement plus grande que le plafond de toutes les sémaphores bloquées par d'autres tâches, P(A) > C(S)
- Sinon A est bloquée et elle transmet sa priorité à la tâche qui tient S

- Priority Ceiling n'accepte les demandes d'entrée dans une section critique que si la priorité de la tâche A est
  supérieure à PC (toutes les ressources auxquelles accèdent les tâches autres que A). Cela résout les blocages
  en empêchant les conditions d'attente circulaire.

- Si la tâche A est bloquée à cause de cela, alors la priorité de la tâche B qui bloque la tâche A sera élevée à la priorité de A.
  sera élevée à la priorité de A (si A est la plus élevée) comme le protocole habituel d'héritage de priorité PIP.
- Une autre amélioration est qu'elle réduit le temps de blocage à la longueur d'une section critique d'une tâche de faible priorité.
- Mais il est difficile à mettre en œuvre et n'est pas implémenté dans RTX/Mbed OS.
- RTX/Mbed OS implémente le Protocole d'Héritage

Sommaire des Resource Access Protocols

Mémoire

Les tâches ont besoin de mémoire

La mémoire d'un ordinateur est utilisé pour stocker

- Le code du programme
- Les données de l'application
- Les données modifiées/calculées

La mémoire est d'habitude organisée en sections différentes

- Code ou texte
  - Instructions binaires à être exécutée
  - Elle est habituellement read-only
  - Le Program Counter (PC) pointe sur la prochaine instruction à être exécutée
- Données statiques
  - Variables globales/constantes/statiques partagées entre tâches/threads
- Heap
  - Dynamique allouée avec malloc/free (C) ou new/delete (C++)
- Stack
  - Utilisé pour exécuter du code, méthode/fonction appel et retour
  - Position est dans le stack Pointer

What Does Memory Management Do?

La mémoire a besoin d'être gérée

- Autant l'OS que les tâches de l'utilisateur ont besoin de la mémoire

Allocation/Partition

- Comment allouer des sections de mémoire spécifiques pour chaque utilisation (code, statique, dynamique)
- C'est fait au build, link et à l'exécution

Réallocation

- Changer l'espace mémoire dynamiquement, la translation est idéalement faite par hardware

Protection

- Référence illégale à la mémoire d'autres process devrait être détectée et stoppée au durant l'exécution

- Les tâches sur Cortex-M peuvent implémenter un Memory Protection Unit (MPU)

Partage

- Plusieurs tâches/threads peuvent accéder à des parties communes de la mémoire

Typical Program-Generation Flow

La génération d'un programme suit un développement typique

- Compile → Assembler → Link → Download
- Le fichier exécutable généré (ou program image) est stocké dans la mémoire programme (normalement dans la mémoire flash on-chip), pour être fetché par le processeur

Compilation using Arm-Based Tools

Compiler Stages

Pre-processing

- Remplace les macros qui sont définie par un # dans le code
- Fusionne les sous-fichiers (.c/.cpp, .h) en un seul fichier (remplace les includes .h par son texte interne)

Parser (analyseur)

- Lit le code C
- Vérifie les erreurs de syntaxes
- Créer un code intermédiaire (représentation en arbre)

Optimisateur Haut-niveau : modifie le code intermédiaire (indépendant du processeur)

Générateur de code (dès qu'on connait l'architecture)

- Crée le code assembleur étapes par étapes pour chaque noeud du code intermédiaire
- Alloue les registres à différentes utilisations

Optimisateur Bas-niveau : modifie le code assembleur (les parties sont spécifiques au processeur)

Linker/Loader : créer une image exécutable du fichier objet

Cortex-M4 Program Image

Qu'est-ce qu'une image programme?

- L'image programme (parfois appelé fichier exécutable) est une pièce rempli de code prêt à être exécuté

Dans le Cortex-M4, l'image programme inclus :

- Table des vecteurs : inclus l'adresse de départ des exceptions (vecteurs) et la valeur du main stack pointer (MSP)
- Routine C de start-up
- Programme : code et donnée de l'application
- Code de librairie C : code programme pour des fonctions de librairie C.

Sur notre cible

C Start-up code

- Utilisé pour préparer la mémoire donnée et l'initialisation de valeur pour les variables de données globales
- Est inséré automatiquement par le compilateur/linkeur, labelé comme '_main' par un compilateur ARM, ou '_start' par un compilateur GNU

Code Programme

- Le code du programme réfère aux instructions générées (code de l'application) du programme application et des données application qui inclus:
    - Valeurs initiales de variables : variables locales qui sont initialisées dans les fonctions ou subroutines durant l'exécution du programme
    - Constantes : utilisées dans les valeurs de données, adresses de périphériques, characters strings, etc.
        - Parfois stockées ensembles dans des blocs de données appelés literal pools
        - Données constantes comme les lookup tables, données images graphiques (ex. bitmap) peuvent être fusionnées dans l'image du prgramme

C library code :

- Codes objets insérés dans le programme par le linker

Program Image in Global Memory

L'image du programme est stockée dans le Code Region dans la mémoire globale

- Jusqu'à 512 MB d'espace mémoire de $0x00000000$ à $0x1FFFFFFF$
    - Notre cible : 1MB ($0x0800\_0000$ à $0x0810\_0000$, aussi mapé de $0x000\_0000$)
- Habituellement implémenter dans la mémoire non-volatile, comme une one-chip FLASH memory
- Normalement séparé des données programme, qui sont allouées dans la région mémoire SRAM (data region)
    - Notre cible
        - SRAM1 (96 KiB, $0x2000\_0000$ à $0x2001\_8000$)
        - SRAM2 (32 KiB, $0x1000\_0000$ à $0x1000\_8000$)

The Mbed Memory Model

Il suit le modèle Cortex-M

- Il inclus une bootloader additionnel et optionnel
- Un bootloader est un programme qui charge Mbed OS quand le board est activé
- Habituellement, le bootloader vient avant l'application dans la ROM et l'application démarre immédiatement après le bootloader
- Une séquence de boot peut avoir plusieurs étages de bootloaders, avant d'arriver à l'application
    - Les différents étages (incluant l'application) peuvent avoir besoin d'évoluer avec le temps, pour ajouter des features ou corriger des bugs

- La plupart des séquences de boot sont composés de 3 étages
    - Boot selector (aussi connu comme root bootloader bootloader étage zéro) ne change jamais
    - Bootlader : peut être mis à jour avec plusieurs versions stockées sur l'appareil
    - Application : peut être mise à jour avec plusieurs versions stockées sur l'appareil

How is Data Stored in RAM?

Typiquement, les données peuvent être séparées en 3 sections : static data, stack et heap

- Static Data : contient les variables globales et les variables statiques
- Stack : contient les données temporaires pour les variables locales, les paramètres passés dans les appels de fonctions, les registres sauvegardés durant les exceptions, etc...
- Heap : contient les parties d'espace mémoire qui sont réservées dynamiquement avec calloc(), malloc() ou new

The Mbed Memory Model

Dans la RAM, on peut distinguer 2 types logiques : static et dynamic memory

Static memory : alloué au moment de la compilation

- Table des vecteurs (read and write)
- Crash Data RAM
- Global data
- Static data
- Stacks des threads par défaut (main, timer, idle et scheduler/ISR)

Dynamic Memory est alloué at runtime

- Heap (données dynamiques)
- Stacks pour les threads utilisateurs

Le check du stack est activé pour tous les threads et le noyau produit une erreur si il détecte une condition d'overflow

L'adresse et la taille du stack et du heap sont définies au moment du build

- Elles peuvent être définies en C (avec le fichier linker) ou en assembleur

Le linker utilise aussi un scatter file qui décrit la location des différentes régions de mémoire. Ce fichier contient la définition de

- The Code Region (LR_IROM1)
- The Crash Data Region (RW_m_crash_data)
- The two RAM Regions (RW_IRAM1, RW_IRAM
- The Heap Region (ARM_LIB_HEAP)
- The Stack Region (MBED_RAM_START)

Data Storage Through An Example

What Memory Does a Program Need?

Est-ce que l'information peut changer?

- No : il faut la mettre dans la read-only pour sauver de la RAM
- Oui : il faut la mettre dans la read/write

Combien de temps la donnée a besoin de durer? La durée de vie doit être la plus courte possible

- Scope du programme : alloué statiquement
- Scope d'une fonction/méthode : alloué automatiquement dans la stack
- De l'allocation explicite à la désallocation explicite : dans le heap
- Toujours définir le scope le plus restrictif
- Utiliser l'allocation dynamique sur le heap avec attention

Data and Memory

Un nombre de type de données standard sont supporté par le C/C++

Néanmoins leur implémentation dépend de l'architecture et du compilateur C/C++

Dans la programmation ARM, la taille des données et référée comme un byte, half word, word et double word

- Byte : 8-bit
- Half word : 16-bit
- Word : 32-bit
- Double word: 64-bit

Le tableau montre l'implémentation des différents types (en jaune bonne pratique)

Class Qualifiers

Const

Jamais écrit par le programme, peut être mis dans la ROM pour sauver de la RAM

Volatile

Peut être changé en dehors du flux normal du programme : ISR, Hardware register

Le compilateur doit être prudent avec les optimisations

Static

Déclaré dans des fonctions ou méthodes et gardent leur valeurs entre les appels

Déclaré dans les classes, le champ est instancié une fois pour toutes les instances de la classe et la valeur est gardé pour la longueur de vie du programme

Activation Record/Stack Frame

Les activation records sont situé sur le stack

- Appeler une fonction va créer un activation record
- Revenir d'une fonction va supprimer l'activation record

Les variables automatiques (variables qui sont allouées et désalloués dans le scope, de base les variables sont automatiques si on dit rien en C) et les informations de gestion sont stockés dans l'activation record d'une fonctions

C'est utilisé pour des fonctions récursives par exemple, ça permet de stocker les arguments d'entrées l'adresse de retour et les résultats

Accessing Data

Qu'est-ce que ça prend d'obtenir une variable dans la mémoire

- ça dépend la location, qui dépend du type de stockage (static, automatic, dynamic)
- donc le temps/coût associé est variable

Memory Hierarchy

- Register : usuellement un cycle CPU pour y accéder
- Cache
  - Static RAM
- Main memory
  - Dynamic RAM
  - Volatile data
- Secondary Memory : Flash/Hard disk
- Tertiary Memory : Tape libraries

- Localité temporelle
- Localité spatiale

- Memory Hierarchy - pour exploiter la localité de la mémoire

Memory Protection Unit (MPU)

Qu'est que le MPU pour ARM?

- Unité programmable
- Permet au software privilégié comme l'OS kernel d'aller définir les permissions d'accès mémoire
- Il monitor les transactions, qui inclut les instructions fetches et data access
- Il trig un fault exception quand une violation d'accès est détectée

Le software privilégié/OS kernel

- Définit les régions mémoires
- Assigne des permissions d'accès mémoire et des attributs de mémoires à chacun d'eux

C'est un composant puissant pour améliorer la sécurité du système

- Il peut empêcher le mode utilisateur/logiciel d'application (c'est-à-dire le logiciel fonctionnant en mode non privilégié) d'accéder aux régions critiques de la mémoire.

Par exemple, un OS peut faire :

- Définir une région de la mémoire, par exemple $0x4000\_0000$ à $0x4000\_FFFF$
- Faire que cette région n'est accessible seulement quand le code du processeur tourne en mode privilégié
- Faire de cette région une read-only

- Faire de cette région une Execute-Never

MPU Programming

A travers les registres du MPU, qui peuvent être lus/écrits seulement quand le processeur est au niveau privilégié d'accès

8 régions de mémoire sont permises, identifiées par la base address et la size

Chaque région peut avoir des droit d'accès différents et le MPU peut activé/désactivé chaque région

Chaque transaction du processeur est checked contre la configuration du MPU

- Si les attributs de la transactions matches les droits d'accès de la région, la transaction est successful, et est produite à l'interface du processeurs
- Dans le cas d'un mismatch, une exception est générée et le processeur saute à l'handler de l'exception

MPU on Mbed OS

La protection de mémoire pour Mbed OS est activée automatiquement pour les devices qui suppport le MPU API

Les fonctions de management du MPU fournies par MBed sont limitée à couper la protection si nécessaire

- Voir MPU API

La protection de mémoire dans Mbed OS fait les choses suivantes

- Il prévient l'exécution de la RAM
- Il prévient l'écriture à la ROM

Mbed OS gère le MPU management automatiquement dans les situations suivantes

- La protection de mémoire est activée comme une partie de la séquence de boot
- La protection de mémoire est désactivée quand une nouvelle application démarre
- La protection de mémoire est désactivée quand on programme la flash

Blocage de l'exécution de la RAM (ScopedRamExecutionLock)

- Après le boot, exécution de la RAM n'est pas autorisée
- Les applications/librairies qui ont besoin de la possibilité d'exécuter depuis la RAM peuvent activer cette possibilité en acquérant le RAM Execution Lock

Blocage de l'écriture de la ROM (ScopedRomWriteLock)

- Après le boot, écrire sur la ROM n'est pas autorisé
- Les applications/librairies qui ont besoin de la possibilité d'écrire sur la ROM peuvent activer cette possibilité en acquérant le RAM Write Lock

Bootloader

Good enough, soon enough

Comment fait-on un logiciel assez bon, sans faire faillit?

- Suivre un bon plan

- Développer et tester de manière efficace
    - Faire des design et des codes reviews
    - Développer sans tester n'est pas efficace
- Tout est itératif
    - Etre capable d'améliorer le process de développement
    - Etre capable d'améliorer le logiciel

What is a good plan?

Commencer avec les requirements des consomateurs

Designer le building block du système

- Quels sont les tâches du système et les modules?

Ajouter les requirements manquants et les contraintes

Appliquer un bon process de développement

- Intégrer les tests dès le début
- Planifier des test unitaires et d'intégration, automatiquement si possible
- Appliquer continuous integration and delivery (CI/CD)

Toute version logicielle a un lifecycle

Tout est itératif

Les requirements vont évoluer avec le temps

- Le design a besoin d'être adapté
- Le plan de test doit être adapté

Les améliorations doivent être développé et déployé

- Adaptations (nouveau requirements)
- Corrections (en incluant des maintenance préventive)

What about deploying updates to embedded systems?

Des milliards de microcontrôleurs sont shippés et délivrer chaque année

Comment faire des amélioration de logiciels à ces microcontrôleurs

- On ne peut pas retourner l'équipement au fabricant
    - On ne renvoie pas notre ordinateur pour une mise à jour
- Bien que les mises à jour de certains système se font de manière assistée par l'utilisateur, ce n'est souvent pas le cas pour les systèmes embarqués
- Très souvent les mises à jour sont déployées remotely

Déployer des mises à jours sur de microcontrôleurs sont faites à travers une application bootloader

What is a bootloader?

C'est une application (logiciel)

Premier objectif : permettre à un software/firmware à être mis à jour

Sans l'utilisation d'un hardware spécialisé

- No Jtag programmer

Il peut utiliser différents protocoles pour recevoir le software update

What does it take to develop a bootloader?

Complète compréhension de

- Comment le microcontrôleur fonctionne
- Comment la mémoire est organisée
- Comment la flash est partitionnée et peut être écrite/écrasée

Bootloader requirements

Avoir la possibilité de switch le mode d'opération (application ou bootloader)

- Comment entrer de le mode bootloader?

Une interface de communication (USB, réseau, ...)

Format de l'update (hex, bin, ...)

Flash/EEPROM requirements (erase, writem read, map/location)

Protection contre la corruption (checksum)

Sécurité (protéger le bootloader et l'application)

Bootloader process

Différents processus sont possibles car il y a différents scénarios

Branching code

- Dans la plupart des cas, le bootloader est toujours exécuté pour faire des fonctions systèmes basiques (check de l'intégrité du système)
- Le branching code est donc inclus dans le bootloader
- Le branching code fait la décision si le bootloader check l'intégrité et les nouvelles version du firmware. Il fait aussi la décision d'installer un nouveau firmware

Application code

- Est exécuté après le branching code et le check d'intégrité
- Peut avoir une tâche pour télécharger un nouveau firmware - cette tâche peut aussi être fait par le bootloader

Bootloader application

Le bootloader n'est pas différent d'une application standard

Il a la capacité d'effacer et de programmer une nouvelle application à sa place en supportant les commandes suivantes :

- Erase the flash
- Write the flash
- Exit/restart (reboot en sautant au code de l'application)

Selon le scénario, il peut avoir besoin de l'accès au périphériques pour faire les fonctions de bootloading

Télécharger le nouveau firmware

- Peut arriver à travers le réseau ou basé sur un évènement spécifique (par exemple insérer une carte sd ou se connecter à un port série)
- C'est souvent fait dans une tâche dans l'application principale (en particulier si le téléchargement se fait via un réseau)
- Peut aussi faire partie de l'application bootloader

Memory Model with a bootloader

Les outils de Mbed OS supportent le modèle de mémoire suivant.

Quand on build l'application principale, cela défini les symboles qui peuvent être utilisé par l'application principale.

Quand on build l'application bootloader, cela défini les symboles qui peuvent être utilisé par l'application bootloader

Checking application integrity

Adding metadata application header

Les metadatas sont créés et ajouté au moment du build

- Version du header
- Version et taille du firmware
- Hash du firmware (signature, pas encryptée)

Metadatas utilisées par le bootloader

- Pour checker l'intégrité de l'application
- Pour installer des applications candidates

Memory Model with application header

Quand on build l'application principale, cela défini les symboles qui peuvent être utilisé par l'application principale.

Quand on build l'application bootloader, cela défini les symboles qui peuvent être utilisé par l'application bootloader

Downloading firmware candidates

Peut être fait en utilisant différents protocoles

- Sur un réseau IP
    - Peut être opérer depuis un système de management centralisé
- Sur une connexion BLE
    - A travers une IP gateway ou à travers une connexion locale BLE
- Accès physique
    - Par exemple sur un port série

Dans tous les cas, un update client doit tourner dans l'application

- Le client doit surveiller les updates availables
- Il doit télécharger les fimware candidates et les stocker à un endroit approprié
- Habituellement, plusieurs candidats peuvent être stockés sur la cible

- Il n'installe PAS le firmware - c'est le job du bootloader

Storing the firmware candidates

Les firmware candidates doivent être stockés sur la mémoire non volatile

La mémoire non volatile (Flash) peut être :

- La flash interne
  - Si elle est assez grande
  - En utilisant le FlashIAP API sur Mbed OS
- Une autre mémoire flash
  - Comme la mémoire 64-Mbit Quad-SPI Flash sur le DISCO
  - En utilisant le QuadSPI API sur Mbed OS

Memory model with firmwares candidates (Internal Flash)

Testing Embedded Systems

Motivations

Les développeurs qui testent "les changements qui ont fait" n'est pas suffisant.

- L'interdépendance être les composants software
- Dépendances à l'environnement
- Le test prend du temps et est ennuyeux

Le test doit être fait d'une manière telle que

- Il améliore la qualité avant tout
- N'empêche pas les développeurs de faire leur job préféré
- Est implémenté dans des procédures automatiques

The pyramid Test

Mike Cohn's concept

- 3 couches hiérarchiques
- La granularité est spécifique à chaque couche
- Beaucoup de tests à granularité fine
- Moins de tests avec plus d'intégration

Test Performance

Les tests ont des granularités différentes et sont donc très différents

Les bons tests partagent tout de même quelques caractéristiques:

- Ils sont déterministes
- Ils sont entièrement automatisés
- Ils sont réactifs

Unit tests

Test d'une seule fonction ou une seule classe

- Pas de dépendance sur la plateforme ou libraries externes
- Utilisation de stubs ou mocks pour l'isoler des dépendances externes

Peut tourner sur tout environnement

- L'environnement hôte est pivilégié

Integration tests

Plus grande granularité que les tests unitaires

- Des dépendances sur la plateforme ou sur des librairies externes peuvent exister (souvent)
- Mbed OS : Greentea est l'outil de test automatique

Doivent tourner sur la plateforme cible

- Les outils doivent permettre de build, flash, execution et générer les résultats
    - Une intégration avec la plateforme hôte est requise
- Peuvent être contrôlé par un hôte local mais devraient aussi permettre l'automatisation dans un environnement d'intégration continue

Greentea for Mbed OS

Permet de lancer des programmes de test sur la cible embarqué

- Build l'environnement hôte
- Flash depuis l'hôte
- Lance les tests sur la cible embarquée
- L'hôte collecte les résultats

Les test sur l'hôtes

- Ecrit par des scripts Python qui tournent sur l'ordinateur
- Peuvent communiquer en retour sur la cible embarquée
- On peut par exemple vérifier que les données sont envoyées correctement sur une plateforme cloud

Write Your Own Test

Suivre une certaine structure

Utiliser les frameworks suivants :

- Greentea client:
    - Test setup (with host integration)
- Unity
    - Test macros
- utest
    - Test clases definition
    - Test execution
    - Reporting handlers

Test Automation

Intégration dans un environnement CI/CD

Améliore la qualité du code

- Passer du temps à coder plutôt que de tester

Permet des améliorations de codes efficaces

- Faire en sorte que les bugs sont plus facilement détectables
- Filet de sauvetage pour les développeurs

Faire des tests qui sont

- Déterministes
- Entièrement automatisés
- Réactifs, avec un report efficace

CI/CD principle

Changements:

1. On crée une branche
2. On travail sur la branche
3. On attend que toutes les phases de test réussissent
4. Un report sort
5. Une autre personne regarde et approuve
6. On merge
7. On refait des tests et on déploie

Use Docker Images

Manière facile d'avoir un setup d'environnement de build pour des projets Mbed OS

- Elles inclusent des toolchains, cmake, Greentea

Les images Docker sont stockées dans les packages Githubs

- Peuvent être utilisées sur notre ordinateur
- Peuvent être utilisées sur Github
    - Restriction : tester sur un device physique