Embedded Real Time Operating System (EmRTOS)

Systèmes d'exploitation embarqués Temps Réel

Application : (FreeRTOS / Embedded RTLinux)

STUDENT Edition

Ver 2.4.0

Sadok Bazine <sbazine.ens@gmail.com>

Google Classroom EmRTOS-ING3-uElec-25/26

2025/2026

Table des matières

Problématique

Le système temps réel et son environnement

Système : ensemble d'"activités" correspondant à un ou plusieurs traitements effectués en séquence ou en concurrence et qui communiquent éventuellement entre eux.

Un système temps réel interagit avec son environnement

— Capteurs : signaux et mesure de signaux

— Unité de traitement

— Actionneurs : actions sur l'environnement à des moment précis

Une définition informelle

Système temps réel : un système dont le comportement dépend, non seulement de l'exactitude des traitements effectués, mais également du temps où les résultats de ces traitements sont produits.

En d'autres termes, un retard est considéré comme une erreur qui peut entraîner de graves conséquences.

Échéances et systèmes temps réel

On distingue différents types d'échéances

— **Échéance dure** : un retard provoque une exception (traitement d'erreur) (exemple : carte son)

— **Échéance molle ou lâche** : un retard ne provoque pas d'exception (exemple : IHM)

On distingue par conséquent différents types de systèmes temps réels

— **Temps réel dur** : les échéances ne doivent en aucun cas être dépassées

— **Temps réel lâche ou mou** : le dépassement occasionnel des échéances ne met pas le système en péril

Criticité

En plus de la tolérance à l'erreur, nous devons prendre en compte la criticité de l'erreur :

— Téléphone portable

— Carte son professionnelle

— Système de commande d'un robot industriel

— Système de commande d'un avion de ligne

Cadencement temporel

— selon une mesure du temps (système piloté par le temps - *time-driven system*)

— selon des événements (système piloté par les événements - *event-driven system*)

— réponse en temps limité : système réactif

Prévisibilité, déterminisme, fiabilité

Systèmes temps réels : prévus pour le pire cas.

Prévisibilité : Pouvoir déterminer à l'avance si un système va pouvoir respecter ses contraintes temporelles. Connaissance des paramètres liés aux calculs.

Déterminisme : Enlever toute incertitude sur le comportement des tâches individuelles et sur leur comportement lorsqu'elles sont mises ensemble.

— variation des durées d'exécution des tâches

— durée des E/S, temps de réaction

— réaction aux interruptions, etc.

Fiabilité : comportement et tolérance aux fautes.

Autour du temps réel

Par conséquent, le temps réel possède plusieurs facettes :

— Ingénierie informatique : Algorithmique, développement

— Ingénierie électromécanique : Maîtrise de l'environnement physique

— Processus : Maîtrise de la qualité du produit, garantie sur les bugs

— Administrative : Certification

Exemple : Un airbag est un système temps réel très dur avec une échéance de temps très faible. La puissance du système n'est pas dans sa conception, mais dans sa garantie de qualité. C'est très facile de faire un airbag, c'est beaucoup plus complexe de le garantir.

Limite des systèmes classiques

Les systèmes classiques s'appuient sur un système d'exploitation en général mal adaptés au temps réel.

— Politique d'ordonnancement visant à équilibrer équitablement le temps alloué à chaque tâche

— mécanismes d'accès aux ressources partagées et de synchronisation comportent des incertitudes temporelles

— gestion des interruptions non optimisées

— la gestion de la mémoire virtuelle, des caches engendrent des fluctuations temporelles

— la gestion des temporisateurs qui servent à la manipulation du temps pas assez fin

Système monotâches

1 Système monotâches

Quelques définitions

— **Temps de réponse** : temps entre un évènement et la fin du traitement de l'évènement.

— Il est possible de formaliser cette définition comme nous le verrons plus tard

1.1 Scrutation des évènements

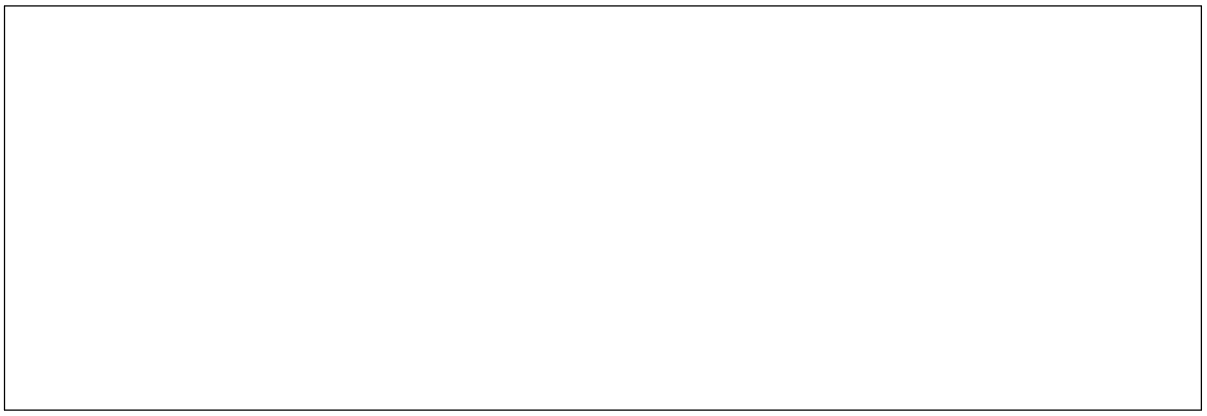
Scrutation des évènements

— Aussi appelé *polling*

— Boucle infinie

— On teste des valeurs des entrées à chaque tour de boucle

```
#define sensor1 *((char *) 0x1234)
#define sensor2 *((char *) 0xABCD)
void action1();
void action2();
int main() {
```



```
        return 0;
}
```

Scrutation

— Temps de réponse au évènements en pire cas facile à calculer : Pire temps pour parcourir la boucle

— Simple à programmer lorsqu'il y a peu de périphériques (ayant des temps de réaction similaires). On peut les scruter en même temps

— Utilisation du CPU sous optimal. Beaucoup de temps est utilisé pour lire la valeur des entrée. Ceci est particulièrement vrai si les évènements sont peu fréquents

— Si certains évènements entraînent des temps de traitement long ou si il y a beaucoup d'entrées à scruter, le temps de réponse en pire cas peut rapidement devenir très grand

— Tous les évènements sont traités avec la même priorité

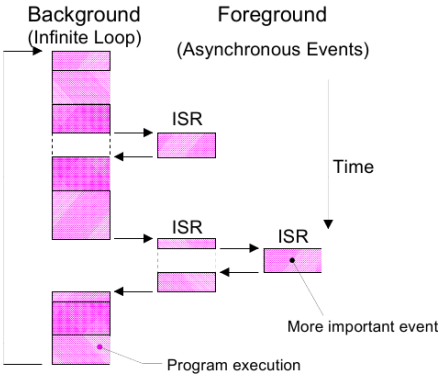
— Mauvaise modularité du code. Ajouter des périphériques revient à repenser tout le système

1.2 Gestion d'interruptions synchrones

Interruptions synchrones

— Appelé aussi Background/Foreground

— Gestion des évènements dans les interruptions



Interruptions synchrones

Concrètement :

```
#define PTR_DATA ((char *) 0x1234)
void isr() {
        action1(*PTR_DATA);
        *PTR_DEVICE_ACK = 1;
}
int main() {
```

```
        return 0;
}
```

Interruptions synchrones

— Temps de réponse au évènements plutôt bon

— Temps de réponse assez simple à calculer. Somme de

- Temps de traitement de l'évènement
- Temps de traitement des évènements de priorité supérieures
- Temps du changement de contexte (plus ou moins constant)
- Pire intervalle de temps ou les interruptions sont désactivées

→ Dans un système simple, ça peut se calculer à la louche

— Le temps de réponse en pire cas des calculs en tâche de fond est quasiment identique au traitement par scrutation (attention tout de même à la fréquence maximum des interruptions)

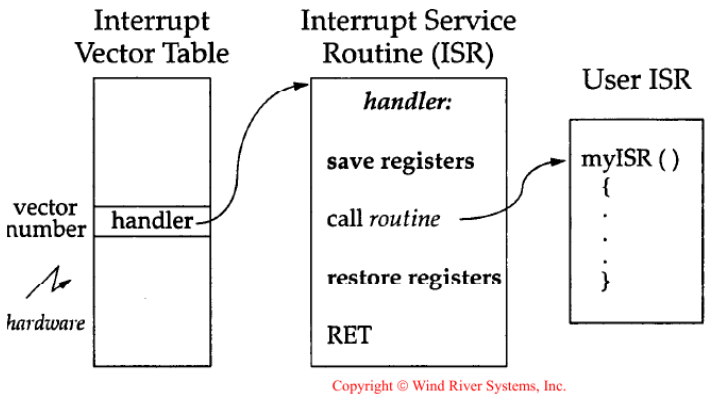
Qu'est-ce qu'une interruption ?

Il existe trois type d'interruptions :

— Les interruptions matérielles :
  - **IRQ** (aussi appelé Interruption externe). Asynchrone. Exemples : clavier, horloge, bus, DMA, second processeur, etc...
  - **Exception**. Asynchrone ou Synchrone. Exemples : Division par zéro, Erreur arithmétique, Erreur d'instruction, Erreur d'alignement, Erreur de page, Break point matériel, Double faute, etc...

— **Logicielle**. Déclenchée par une instruction. Synchrone.

Fonctionnement d'une interruption



Fonctionnement d'une interruption

Quand une interruption est levée :

— le CPU sauvegarde en partie ou en totalité le contexte d'exécution (principalement le pointeur d'instruction) sur la pile

— Le CPU passe en mode superviseur (nous y reviendrons)

— Le CPU recherche dans l'IVT (*Interrupt Vector Table* aussi appelée IDT, *Interrupt Description Table*) l'ISR (*Interruption Service Routine*) associée

— Le CPU place le pointeur d'instruction (le registre `IP/PC`) sur l'ISR

— L'ISR traite l'évènement (le traitement d'une E/S, etc...)

— L'ISR acquitte la réception de l'interruption indiquant qu'une nouvelle donnée peut-être traitée.

— L'ISR restaure le contexte sauvegardé.

Fonctionnement d'un PIC

Le PIC (Programmable Interrupt Controller) est un composant matériel permettant la gestion des IRQ. Il peut-être intégré au CPU ou externe (ou à cheval entre les deux...). Il permet en particulier :

— Activer ou de désactiver des IRQ

— De masquer temporairement une IRQ

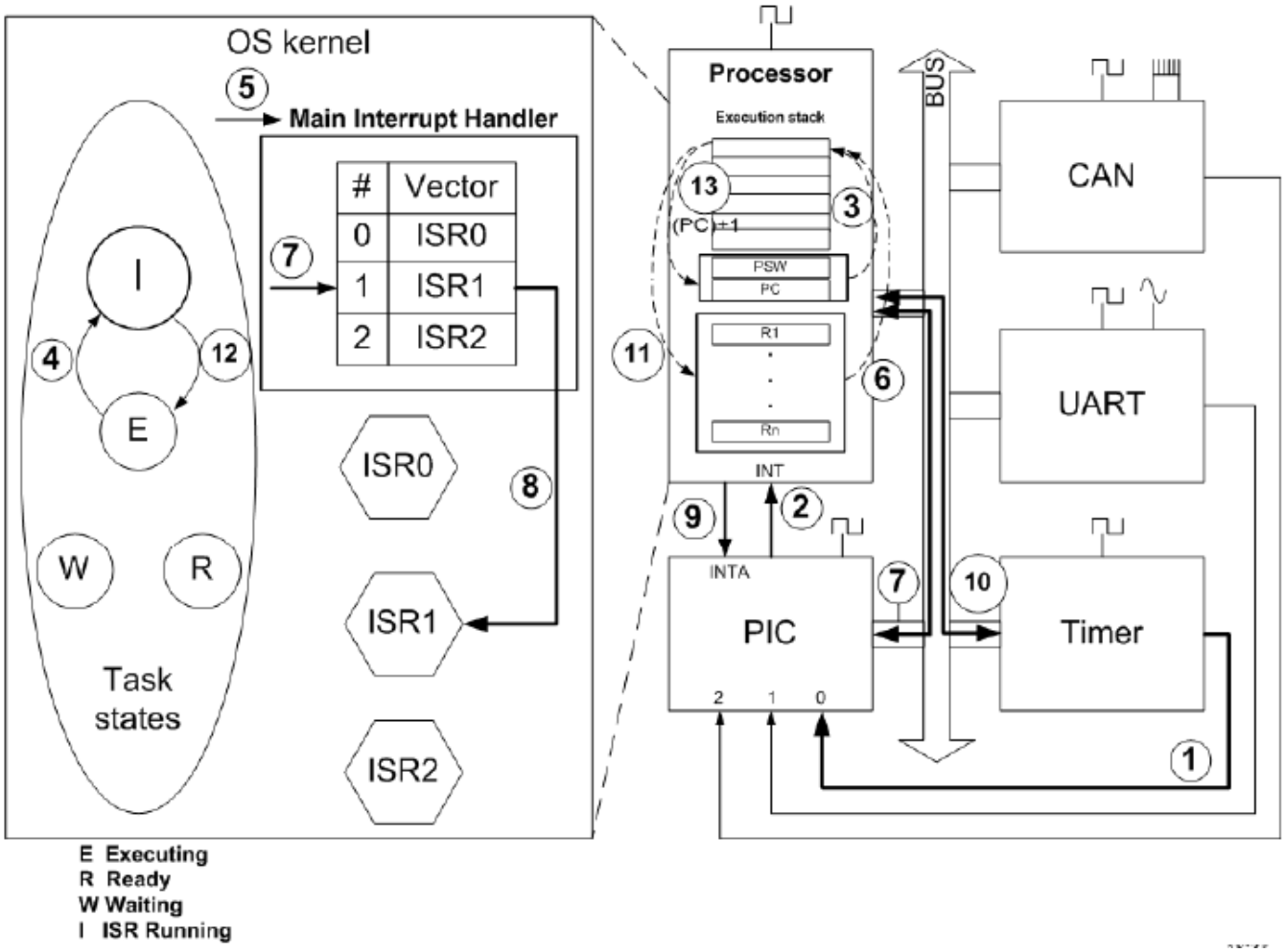
— De mettre en queue une interruption temporairement masquée

— De contrôler la priorité des interruptions

Il arrive fréquemment qu'un PIC soit multiplexé sur une seule ligne d'IRQ. Dans ce cas, le premier étage d'ISR lit un registre du PIC pour connaître le numéro de l'IRQ. (Cas notoire du 8259A sur les architectures x86)

Exemple

Exemple classique d'intégration d'un PIC multiplexé sur une IRQ :



Exemple :IRQ

1. Le périphérique *Timer* lève sa ligne d'IRQ
2. Le PIC reçoit l'interruption et lève une IRQ du processeur
3. Le processeur complète l'instruction courante et sauve le registre d'instruction `PC` et le registre d'état `PSW`
4. La tâche courante devient interrompue (Nous y reviendrons)
5. Le premier étage d'ISR est appelé
6. Le gestionnaire d'interruption complète la sauvegarde des registres
7. Le gestionnaire d'interruption demande au PIC quelle interruption à été appelée et il lit dans l'IVT quelle ISR est associée
8. Le gestionnaire d'interruption se branche sur l'ISR associée (ici, `ISR0`)
9. L'IRQ du processeur est acquittée. Les autre IRQ peuvent ainsi être levées

10. L'`ISR0` lit la valeur provenant du *Timer* et acquitte l'interruption du *Timer*. Ce périphérique peut de nouveau lever des IRQ.
11. Les registres généraux sont restaurés
12. Le contexte d'exécution est restauré
13. Le registre PC est restauré

Scrutation vs Interruption

Exemple de différence d'approche entre la gestion par scrutation et la gestion par interruption : Prenons l'acquisition de donnée à partir d'un convertisseur analogique/numérique asynchrone

— Dans le cas du traitement par scrutation, nous allons périodiquement voir si un résultat est arrivé. Beaucoup de temps est consommé pour rien et lorsque le résultat arrive, le traitement du résultat sera retardé

— Une interruption est levée quand une nouvelle donnée est disponible. Le processeur peut alors la traiter.

Latence des interruptions

— Un périphérique ne génère pas d'IRQ si la précédente n'est pas acquittée (en principe)

— Vu que les interruptions sont souvent multiplexées, les interruptions sont souvent désactivées lors de la première phase de traitement

— Pour des raisons techniques, il est parfois nécessaire de désactiver les interruptions

— Le partage de l'information entre les interruptions et le reste du programme nécessite parfois de désactiver les interruptions (Nous y reviendrons)

Les conséquences :

— Augmentation du temps de réponses

— Temps réponse plus difficile à calculer

— Risque de perdre des interruptions.

Précautions avec les interruption

— Acquitter l'interruption le plus tôt possible

— Rester le moins de temps possible dans une interruption

— Accéder à un minimum de données pour éviter d'avoir à partager des données avec le background

— Transférer un maximum de traitement hors de l'interruption

⇒ *Gestion des interruptions asynchrones*

1.3 Gestion d'interruptions asynchrones

Interruptions asynchrones

— Interruption séparée en deux parties : *top half* et *bottom half*

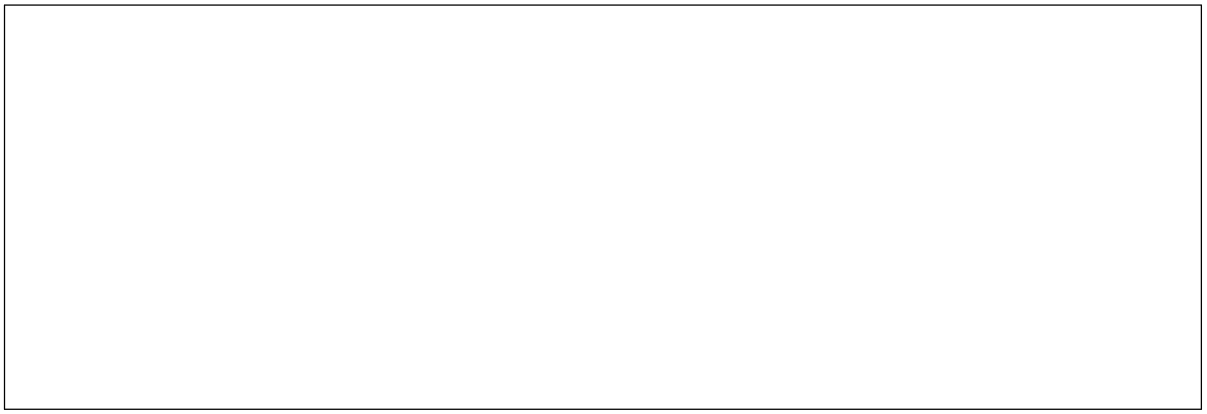
— On délègue le maximum de traitement au *bottom half*

— Permet de décharger les interruptions

— Permet de plus facilement prendre en compte des interactions entre les évènements (exemple, possibilité d'attendre deux évènements avant d'effectuer une action)

Interruptions asynchrones

```
#define PTR_DATA ((char *) 0x1234)
int a = 0;
void isr() {
        a++;
        *PTR_DEVICE_ACK = 1;
}
int main() {
```



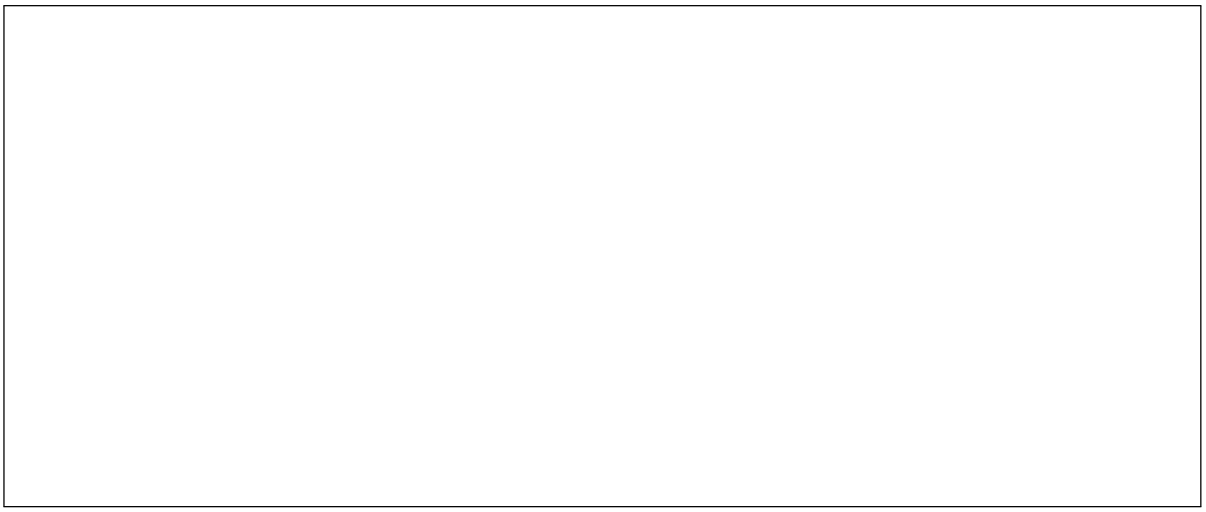
```
        // Optionnal background computing
}}
```

1.4 Protection des structures de données

Exemple de partage de données

Imaginons le code suivant :

```
#define PTR_DATA ((char *) 0x1234)
int a = 0;
char t[255];
void isr() {
        t[a++] = *PTR_DATA;
```



```
                // Optionnal background computing
        }
}
```

Exemple de partage de données

Prenons le cas où `f` traite l'interruption précédente (donc `a = 1`) et qu'une nouvelle interruption est déclenchée :

```
--a; // a = 0;




action1(t[a]);
// Lecture de t[1] au lieu de t[0]!
```

```

t[a] = *PTR_DATA;
// t[0] est {é}cras{é}!
a++;
// a = 1 maintenant
*PTR_DEVICE_ACK = 1;

```

Au lieu de lire correctement la première valeur retournée par l'ISR puis la seconde, nous lirons tout d'abord une valeur aléatoire puis la valeur retournée par la seconde interruption.

Comment éviter le problème ?

Les problèmes d'accès concurrents se traduisent très souvent par des *races conditions*. C'est à dire des problèmes aléatoires produit par une séquence particulière d'évènements

- Les *races conditions* sont souvent difficiles à reproduire et à identifier
- Les *races conditions* peuvent être latente dans le code et se déclarer suite à une modification de l'environnement externe
- Une race condition coûte chère (difficulté de correction, peut potentiellement atterrir en production)

Comment s'en protéger ?

— Ne pas partager de données avec les interruptions

— Utiliser des accès atomiques

— Utiliser des structures adaptées : buffers circulaires et queues

— Désactiver les interruptions lors d'accès à des données partagées

Comment éviter le problème ?

— Buffer circulaire,

— Queue,

— Désactivation des interruptions,

— Spin Lock.

1.5 Les limites du monotâche

Problèmes de la gestion des interruptions asynchrones

Nous n'avons pas résolu notre problème récurent :

— Le partage de l'information entre les interruptions et la boucle principale entraîne des latences

On retrouve certains problèmes que l'on avait avec la scrutation :

— Ne permet pas de prioriser les traitements dans la boucle principale

— Interaction entre les évènements complexe

Système multitâches

2 Tâches Concurrentes

Concurrence

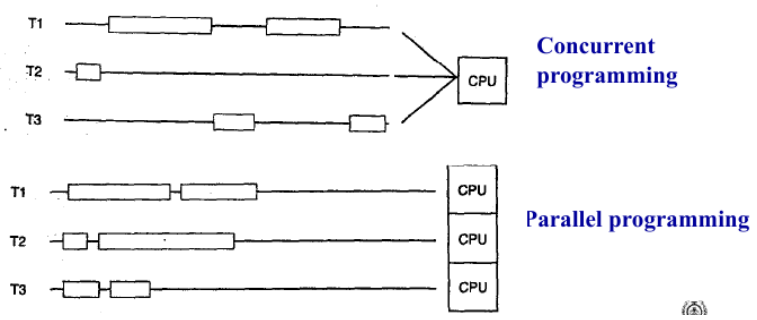
— Des tâches concurrentes sont des tâches exécutées séquentiellement sur un seul processeur en entrelaçant l'exécution de chaque tâches

— Pour les tâches, le temps partagé est transparent. Chaque tâche à l'impression d'avoir le CPU pour elle-seule

— On trouvera aussi le terme de *multitâches* ou de *temps partagé*

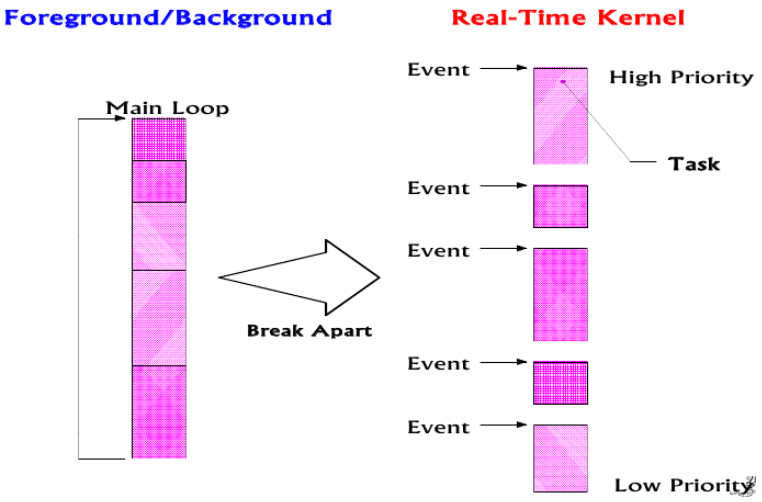
Concurrence

La programmation concurrente N'EST PAS de la programmation parallèle (même les système multi-coeurs sont souvent concurrent et parallèle) :



Concurrence

Migration d'un système avec gestion asynchrone des interruptions vers un système multitâches :



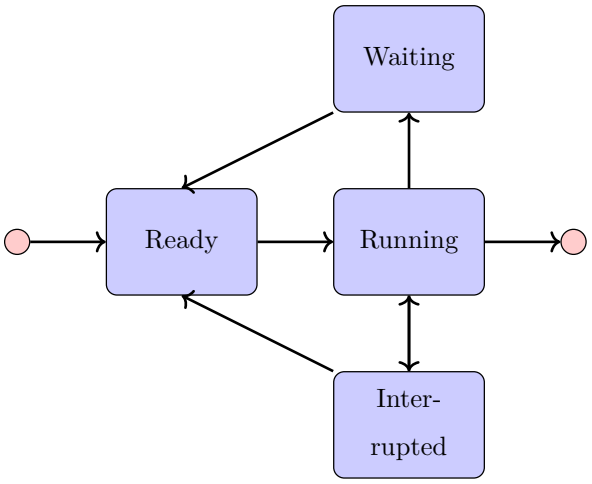
tâches concurrentes

Pour les systèmes plus complexes ou pour facilité la réutilisation, un système multitâches est plus approprié qu'un système monotâches (*Foreground/Background*).

— Facilite la gestion des évènements

— Permet de prioriser les traitements

États des tâches



2.1 Changement de contexte

Le changement de contexte

Chaque tâche possède une pile en mémoire. Une liste globale contient :

— les états de toutes les tâches

— l'emplacement de la pile en mémoire

— le contexte d'exécution, c'est-à-dire une sauvegarde des registres

Lors du changement de contexte

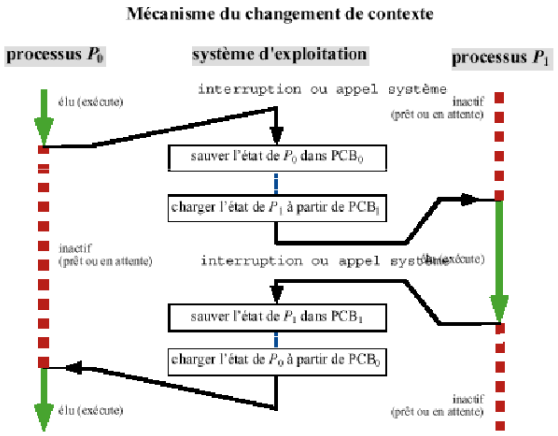
— on sauvegarde le contexte de la tâche précédente, en particulier son pointeur de pile et son pointeur d'instruction

— on restaure le contexte de la nouvelle tâche

— on restaure le pointeur d'instruction

Dans la pratique, il y a des petites subtilités selon la manière dont le changement de contexte à été amené.

Le changement de contexte



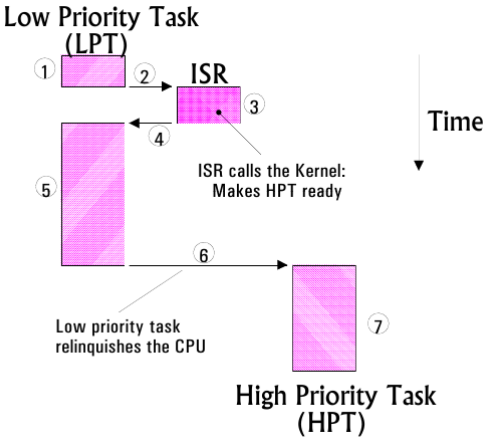
2.2 Multitâches non-préemptif

Multitâches non-préemptif

Le changement de contexte peut-être volontaire par les tâches. Dans ce cas, la tâche ayant terminé son traitement appellera explicitement la fonction *schedule* qui effectuera la changement de contexte. le système est dit non-préemptif ou multitâches collaboratif.

Multitâches non-préemptif

Ce type de système implique une latence difficilement quantifiable entre un évènement et sont traitement :



Multitâches non-préemptif

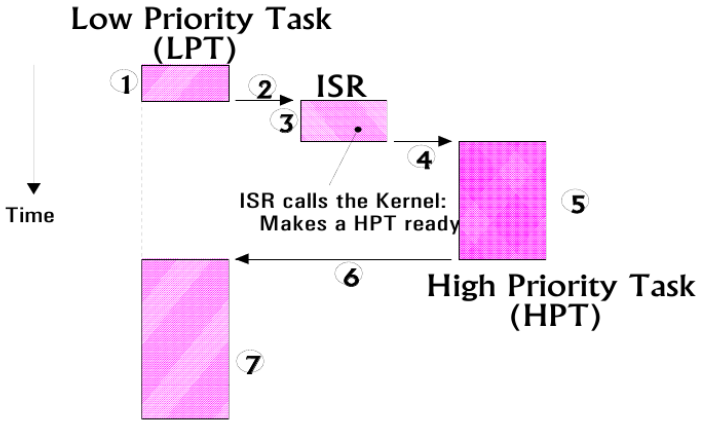
1. Une tâche non prioritaire est en cours d'exécution et est interrompue par un évènement (une IRQ)
2. L'ISR est appelée
3. L'IRQ rend une tâche de haute priorité prête à être exécutée
4. A la fin de l'ISR, le système rend le CPU à la tâche non-prioritaire

5. Quand la tâche non-prioritaire termine sont traitement, elle appelle `schedule`
6. L'ordonnanceur donne la main à la tâche de forte priorité
7. La tâche de haute priorité peut (enfin) s'exécuter

2.3 Multitâches préemptif

Multitâches préemptif

Un système multitâches préemptif va être capable de changer de contexte lors des interruptions :



Multitâches préemptif

1. Une tâche non prioritaire est en cours d'exécution et est interrompue par un évènement (une IRQ)
2. L'ISR est appelée
3. L'IRQ rend une tâche de haute priorité prête à être exécutée
4. A la fin de l'ISR, le système appel le scheduler
5. Le scheduler donne la main a la tâche de haute priorité
6. Quand la tâche prioritaire termine sont traitement, elle appelle `schedule`
7. Vu qu'il n'y a plus de tâches prioritaires à exécuter, l'ordonnanceur redonne la main à la tâche de faible priorité

Round robin

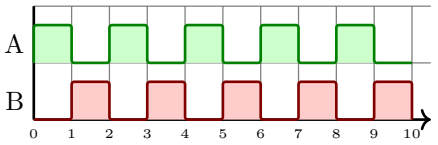
Examinons le cas de deux tâches de priorité égales n'effectuant jamais de relâchement volontaire :

```
1 task1() {
2   for(;;) ;
3 }
4 task2() {
5   for(;;) ;
6 }
```

Round robin

Dans ce cas, si aucune interruption ne se produit, la première tâche à avoir pris la main ne la rendra jamais. Afin de reprendre la main, on utilise une interruption d'horloge. Celle-ci garanti que le système pourra périodiquement réordonnancer les tâches. La période l'horloge utilisée est appelée quantum de temps ou HZ dans le cas de Linux.

Dans ce cas-ci, l'ordonnanceur devra donner une période à *task1* puis une période à *task2* et ainsi de suite. Ce comportement s'appelle *Round-Robin* ou *Tourniquet*.



3 MMU et temps réel

3.1 La MMU

La MMU

Les CPU modernes intègrent un composant appelé MMU (*Memory Management Unit*) :

- Unité de translation d'adresses mémoire
- On parle d'adresses physiques et d'adresses virtuelles
- Lorsque le MMU est actif (cas nominal), toutes les adresses du code assembleur sont des adresses virtuelles
- Il est possible de configurer le MMU avec une instruction spéciale en lui donnant un pointeur sur un tableau (dans la pratique, il s'agit plutôt d'un arbre) associant les adresses physiques et les adresses virtuelles

La MMU (2)

- Il est possible de changer les associations simplement en chargeant un pointeur sur une autre table
- On défini alors une table par tâche. Lors du changement de contexte, on change aussi de table
- Le CPU possède alors deux modes :
  - Utilisateur
  - Superviseur
- Seul le mode superviseur (l'OS) permet de modifier les associations de la MMU

3.2 MMU et gestion des exceptions

La MMU - gestion des exceptions

Toutes les adresses physiques ne sont pas associées à des adresses virtuelles

- Une tâche A ne peut pas accéder à la mémoire d'une tâche B
- Protection contre les erreurs de programmation
- Permet d'assurer la sécurité des systèmes multi-utilisateurs
- Une tâche à l'impression d'avoir toute la mémoire pour elle

La MMU - gestion des exceptions

Le MMU permet à l'OS de mieux utiliser la mémoire :

- L'OS peut donner des espaces d'adressage virtuel contigu alors que la mémoire physique est fractionnée
- Le système n'alloue jamais la plage $[0, 1024]$
  - Cela donne une plage de valeurs spéciales (ex : NULL)
  - Ainsi, lors du debug, vous êtes certains qu'un pointeur $\in [0, 1024]$ est non valide
  - En dehors des pointeurs, les nombres que l'on manipule sont très souvent < 1024. Ce système nous permet de rapidement repérer des casts abusifs entre des integers et des pointeurs
- "Sun a inventé le SegFault"

Gestion de la mémoire

Retarder l'association :

- Une tâche demande une allocation
- Le système enregistre la demande dans le Memory Manager mais ne modifie pas le MMU
- Le système indique à la tâche que l'allocation s'est correctement déroulée
- Lorsque la tâche accède à cette page, une exception est levée
- Le système reprend la main
- Il remarque qu'il avait promis cette page
- Il alloue un bloc physique et met à jour la MMU
- Il rend la main à la tâche
- Tout est transparent pour la tâche

Gestion de la mémoire

Utilisation de la Swap :

- Lorsque le système n'a plus assez de mémoire
- Il choisit une page physique qu'il copie sur le disque dur

— Il supprime la page de la MMU de la (les) tâche(s) concernée(s)

— Lorsque la tâche accède à la page supprimée, une exception est levée

— Le système récupère alors la page sur le disque

— Le système réécrit la page dans la mémoire physique

— Il associe l'adresse virtuelle demandée avec la nouvelle page physique

— L'OS rend la main à la tâche

Gestion de la mémoire

Gestion des droits sur les pages

— Il est possible d'affecter des droits en lecture/écriture/éxécution sur les pages gérées par la MMU

— Si la tâche essaye d'écrire sur une page contenant des données constantes, il s'agit d'un bug et une exception est levée

— On garantie que les pages *read-only* ne seront pas modifiées

— Une page contenant des données constantes (donné ou code) peut être mappée dans plusieurs tâches différentes

— En retirant les droits en exécution sur les pages de données, on améliore la sécurité du système (impossible d'exécuter une page contenant des données)

— Une page accessible en écriture peut être mappée dans deux tâches afin de leur permettre de partager des données

Gestion de la mémoire

Simplification des accès au IO

— La tâche demande de mapper un fichier en mémoire

— Le système alloue un espace d'adressage virtuel égal à la taille du fichier

— Le fichier en lui même n'est pas chargé en mémoire

— Lorsque la tâche accès à un espace du fichier, une exception est levée et la page demandée est chargée

— Le système marque la page comme Read-Only. Lorsque la tâche tente d'écrire dans la page, une exception est levée, la page est marquée *dirty* , les droits en écriture sont données et le système rend la main

— Lorsque le système à besoin de mémoire, il peut écrire les pages modifiées sur le disque et décharger la page de la mémoire

— Lorsqu'une tâche demande un fichier déjà présent en mémoire, on mappe simplement la MMU sur la page déjà présente

4 Programmation multitâches sous Linux

Threads et Processus

— On appelle les tâches ayant des contextes mémoires différents des *Processus* (cf. *fork(2)*)

— Il est possible d'exécuter plusieurs tâches dans un même contexte mémoire

— Ces tâches sont appelées *threads* ou *processus légers* (cf. *clone(2)*)

— Le fonctionnement est alors identique au mode sans MMU, avec les mêmes défauts et avantages :

  — Pas de protection contre les erreurs de programmation des autres threads

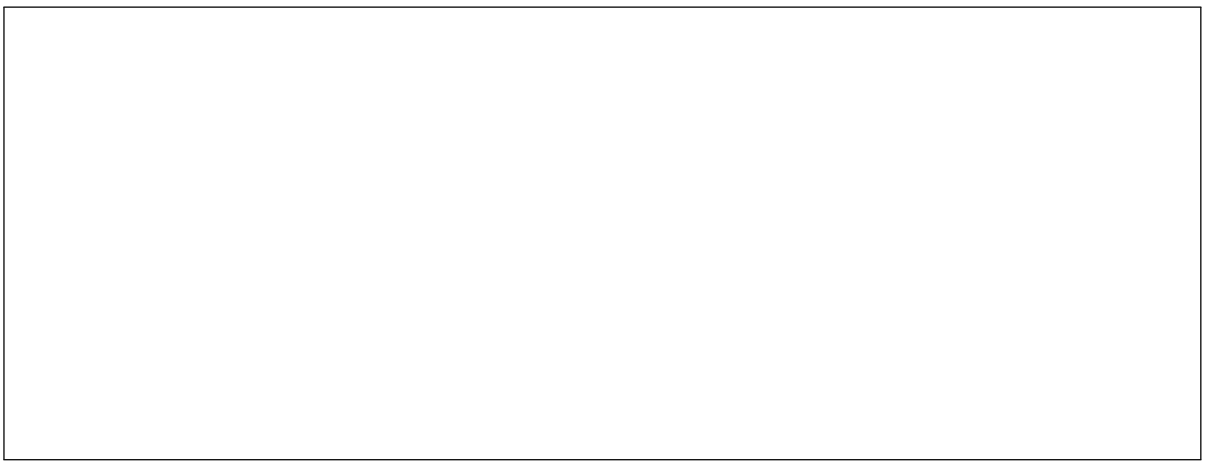
  — Partage de l'information simplifiée

  — Passage d'une thread à une autre est beaucoup plus rapide

4.1 Création de processus

Création de processus

```
1 #include <unistd.h>
2
3 int main() {
4         pid_t pid_fils;
5         pid_fils = fork();
```



```
1 }
```

4.2 Création de threads

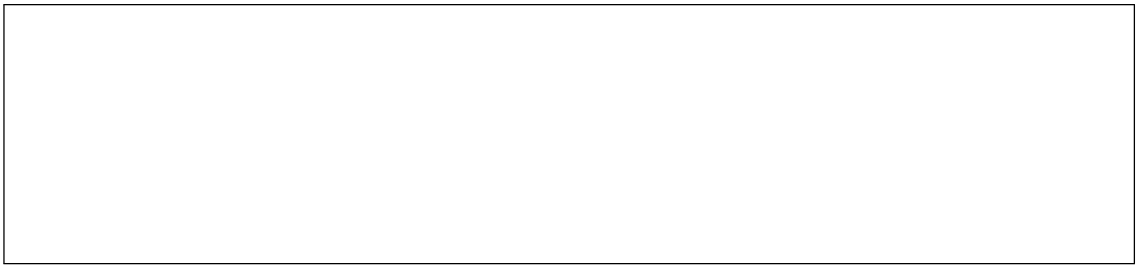
Création de threads

— Un programme minimal permettant de créer des threades est donné par les lignes suivantes :

```
1 #include <pthread.h>
2
3 void* task(void* arg) {
```

```
  int val = *(int *) arg;
  // Child
}

int main() {
```



```
        // Parent
}
```

— Pour plus de détail sur le fonctionnement et la manipulation des *threades* vous pouvez consulter vos fascicules de travaux pratiques.

4.3 Gestion et manipulation des tâches

Commandes ps, pmap et /proc

— `pmem, \%mem, rss, rsz, rssize` : Mémoire résidente, c'est à dire quantité de mémoire physqiue effectivement utilisé par le processus (par consequent, la memoire swapée n'est pas prise en compte).

— `vsz,vsize` : Taille de la plage d'addressage virtuelle du processus (sans les mapping de devices). Peut-être très supérieure à rss

— `sz` : size in physical pages of the core image of the process. This includes text, data, and stack space. Device mappings are currently excluded ; this is subject to change. See vsz and rss.

...

4.4 Gestion de la mémoire

Gestion de la mémoire

Sécurisation des accès aux périphériques

— Lorsque les registres des périphériques sont mappés en mémoire, on utilise la MMU pour y accéder

— Il est possible d'autoriser l'accès à un périphérique à une tâche sans lui donner d'accès au reste du système

— Un système utilisant très fortement cette méthode est appelée micro-kernel

— La méthode est peu utilisée sous Linux

— cf. *ioperm(2)*

Linux et l'MMU

— Le multitâches permet une meilleure gestion de la concurrence

— MMU a de multiple avantages (sécurité, optimisation)

— En revanche le fonctionnement de la MMU entraîne de multiples exceptions

— Une allocation mémoire peut d'un seul coup changer tout le mapping

— Un accès en mémoire peut être immédiat 100 fois mais demander un accès aux disque la 101ème fois

— Il devient difficile de garantir le temps de calcul d'une fonction

— Les fonctions systèmes `mlock` et `mlock_all` permettent de demander à Linux de garder des pages (ou la totalité en mémoire)

— Il ne faut pas oublier d'allouer une pile suffisante avant d'appeler `mlock_all`

— Néanmoins, cela ne change pas que l'allocation dynamique ne se fait pas en temps constant

Utilisation de `mlock`

```c
#include <sys/mman.h>
void alloc_stack_1k(){
        char t[1024];
}

int main() {
```

4.5 Le mode superviseur

Passage en mode superviseur

Un processus utilisateur ne peut pas passer en mode superviseur.

Comment passer en mode superviseur ?

— Lorsqu'une interruption/exception est déclenchée

— Cela nous permet de faire fonctionner les optimisations précédentes

Comment appeler une fonction du système ?

— Les tâches ont besoin de faire des demandes au système (exemple : allouer de la mémoire)

— Ces fonctions système s'appellent des *appels système* ou *syscall* (section 2 des pages de man)

— Elles ont très peu de points communs avec les appels de fonctions classiques

— Chaque *syscall* est associé à un numéro (cf `sys/syscall.h` `asm/unistd_32.h`, *syscalls(2)*)

Passage en mode superviseur

Pour utiliser les *appels systèmes* (cf *syscall(2)*) :

— On place les arguments sur la pile

— On place le numéro de l'interruption sur la pile

— On déclenche une interruption logicielle (`int 0x80`)

— Le CPU passe en mode superviseur et appelle l'ISR de l'interruption

— L'OS prend la main, regarde le premier élément de la pile et appelle la fonction correspondante (`asm-generic/unistd.h`)

Il existe maintenant des instructions spéciales sur les CPU pour optimiser les *syscall* (instructions *sysenter*, *sysexit*)

5 Ordonnancement

5.1 Quelques notions théoriques

5.1.1 Modèle de tâches

Modèle de tâches

Paramètres définissant une tâche i :

— Date d'arrivée de la tâche dans le système : S_i

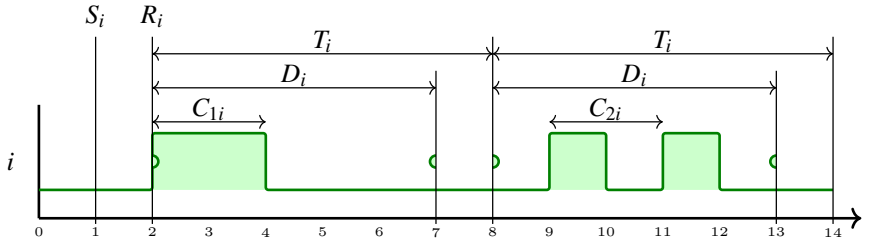
— Première date d'activation : R_i

— Période d'activation : T_i

— Délai critique ou *deadline* (Délai maximum acceptable pour son exécution) : D_i

— Capacité (Temps CPU nécessaire à l'exécution de la tâche) : C_i

Modèle de tâches



Tâche	Arrivée	Période	Capacité	Délai
i	1	6	2	5

Modèle de tâches

— Une tâche périodique est définie par : $(S_i, R_i, T_i, D_i, C_i)$

— Une tâche apériodique est définie par : $(S_i, R_i, 0, D_i, C_i)$

Très souvent :

— Les tâches sont connues au début de l'ordonnancement : $S_i = 0$

— Les tâches périodiques sont actives dès le début de l'ordonnancement : $R_i = 0$

— si $D_i = T_i$: tâche à *échéance sur requête*

5.1.2 Paramètres statiques d'une tâche

Paramètres statiques :

— Date de réveil sur une période k : r_{ik}

— Échéance sur une période k : $d_{ik} = r_{ik} + D_i$

— Facteur d'utilisation du processeur : $U_i = C_i / T_i$

— Facteur de charge du processeur : $CH_i = C_i/D_i$ ($CH_i = U_i$ si la tâche est à échéance sur requête)

— Laxité (ou *Slack*) de la tâche (retard maximum acceptable pour l'exécution de la tâche) : $L_i = D_i - C_i$

5.1.3 Paramètres dynamiques d'une tâche

Paramètres dynamiques

Paramètres dynamiques (dépendant de l'ordonnancement) :

— Priorité (peut être dynamique ou statique) : P_i

— Date du début de l'exécution d'une période k : s_{ik}

— Date de la fin de l'exécution d'une période k :e_{ik}

— Temps de réponse de la tâche $TR_{ik} = e_{ik} - r_{ik}$

— Durée d'exécution résiduelle à la date t : $C_i(t)$ $(0 \leq C_i(t) \leq C_i)$

— Délai critique résiduel à la date t : $D_i(t) = d_{ik} - t$ $(0 \leq D_i(t) \leq D_i)$

— Charge résiduelle à la date t : $CH_i(t) = C_i(t)/D_i(t)$ $(0 \leq CH_i(t) \leq CH_i)$

— Laxité résiduelle à la date t : $L_i(t) = D_i(t) - C_i(t)$ $(0 \leq L_i(t) \leq L_i)$

— Laxité conditionnelle à la date t (somme sur les tâches déclenchées à la date t et qui sont devant i du point de vue de l'ordonnancement) : $LC_i(t)$ (calcul complexe)

5.1.4 Paramètres du système

Paramètres du système

— Configuration : ensemble des tâches mises en jeu par l'application

— Taux d'utilisation du processeur : $U = \sum_i U_i$

— Taux de charge du processeur : $CH = \sum_i CH_i$

— Intervalle d'étude : intervalle de temps minimum pour prouver l'ordonnancabilité d'une configuration

  — Dans le cas de tâches périodiques : $ppcm_i(T_i)$

  — Dans le cas de tâches apériodiques : $[\min_i\{R_i\}, \max_i\{R_i + D_i\} + 2 \times ppcm_i(T_i)]$

— Laxité du processeur : intervalle de temps pendant lequel le processeur peut rester inactif tout en respectant les échéances : $LP(t) = \min_i\{LC_i(t)\}$

5.2 Algorithmes à priorité statique

Typologie des algorithmes

On distingue diverses typologies d'algorithmes :

— *on line* ou *off line* : Choix dynamique ou prédéfini à la conception

— à priorité *statique* ou *dynamique* : La priorité d'une tâche est-elle fixe ou une variable dépendante d'autres paramètres

— *préemptif* ou *non préemptif* : Une tâche peut-elle perdre le processeur (au profit d'une tâche plus prioritaire) ou non

— stratégie du meilleur effort ou inclémence

  — en TR mou, meilleur effort = faire au mieux avec les processeurs disponibles

  — en TR dur, obligation de respecter les contraintes temporelles : inclémence aux fautes temporelles

— centralisé ou réparti

5.2.1 Théorème de l'instant critique

Théorème de l'instant critique

Si toutes les tâches arrivent initialement dans le système en même temps et si elles respectent leur première échéance, alors toutes les échéances seront respectées par la suite, quel que soit l'instant d'arrivée des tâches.

— C'est une condition nécessaire et suffisante si toutes les tâches du système sont initialement prêtes au même instant.

— Dans le cas contraire, c'est une condition nécessaire

5.2.2 Rate Monotonique (RMA)

Rate Monotonique (RMA)

Aussi appelé *RMA* (*Rate Monotonic Algorithm*)

Ordonnancement à priorité statique où les priorités sont inversement proportionnelles aux périodes des tâches.

Fonctionne en version préemptive. La version non-préemptive n'est pas garantie.

Liu et Layland ont démontré qu'un système est ordonnançable si le taux d'occupation du processeur U vérifie la condition suffisante (non nécessaire) suivante :

$$U = \sum_{i}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

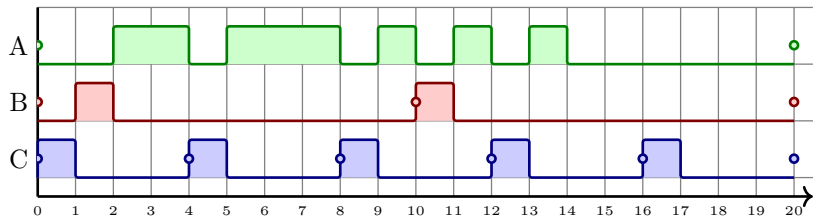
n	limite d'occupation
1	100.0%
2	82.8%
3	78.0%
4	75.7%
5	74.3%
10	71.9%
∞	69.3%

Exemple 1

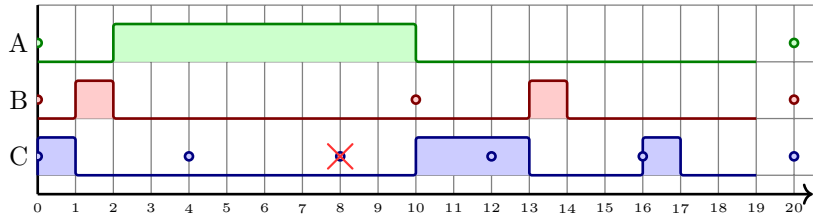
Tâche	Arrivée	Période	Capacité	Délai
A	0	20	8	Fin de période
B	0	10	1	Fin de période
C	0	4	1	Fin de période

Charge du CPU :

Mode préemptif T=



Mode non-préemptif :

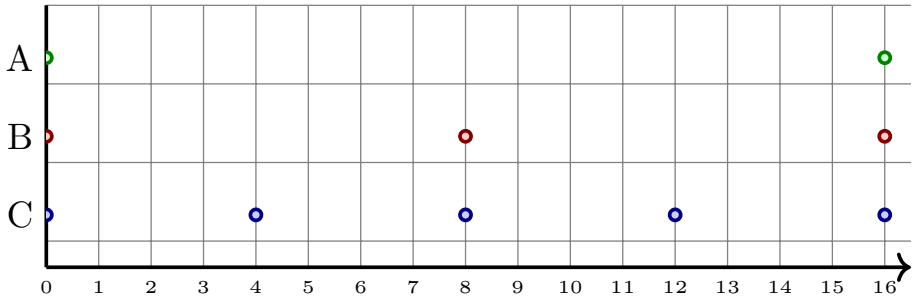


Exemple 2 (Mode préemptif)

Tâche	Arrivée	Période	Capacité	Délai
A	0	16	8	Fin de période
B	0	8	2	Fin de période
C	0	4	1	Fin de période

Charge du CPU :

Chronogramme :



5.2.3 Deadline Monotonic (DMA)

Deadline Monotonic (DMA)

— Aussi appelé *DMA* (*Deadline Monotonic Algorithm*

— Algorithme à priorité statique

— Généralisation de Rate Monotonic pour les tâche avec $D_i < T_i$

— Basé sur le délai critique :

  — La tâche de plus petit délai critique est la plus prioritaire

— Test d'acceptabilité (suffisant mais pas nécessaire) :

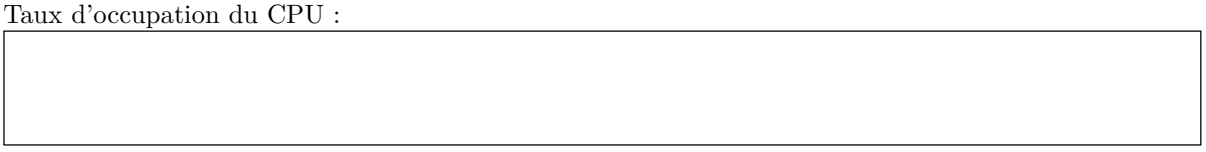
$$CH = \sum_{i}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right)$$

— Équivalent à Rate Monotonic dans le cas des tâches à échéance sur requête, meilleur dans les autres cas

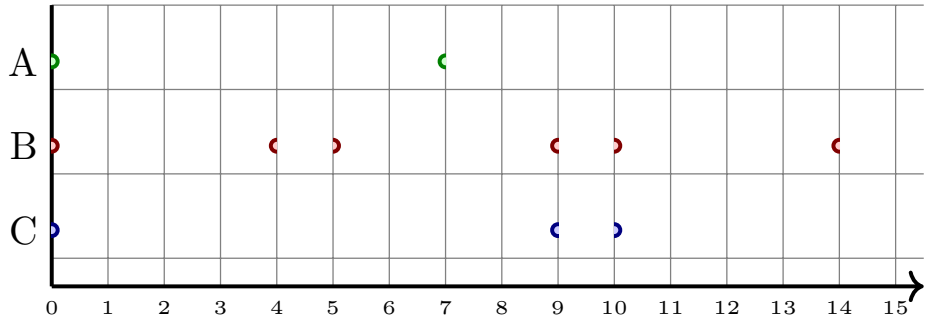
Exemple

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	7
B	0	5	2	4
C	0	10	2	9

Taux d'occupation du CPU :

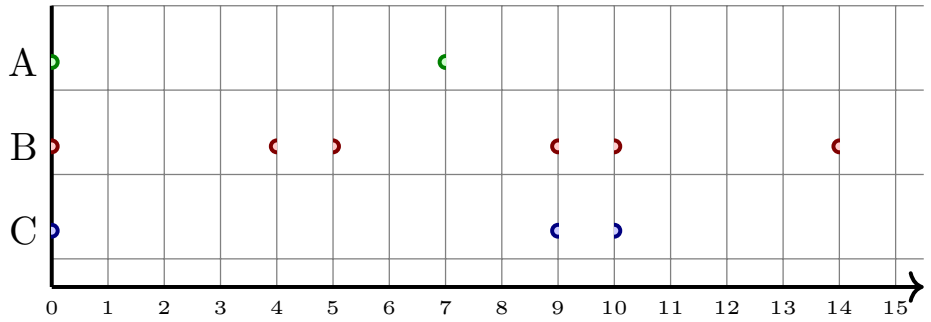


Par RMA :



Test d'acceptabilité pour (DMA) :

Par DMA :



5.3 Algorithmes à priorité dynamique

5.3.1 Earliest Deadline First (EDF)

Algorithme EDF

- Algorithme à priorité dynamique
- Basé sur l'échéance :
  - A chaque instant (i.e à chaque réveil de tâche), la priorité maximale est donnée à la tâche dont l'échéance est la plus proche
- Test d'acceptabilité :
  - Condition nécessaire :

$$U = \sum_{i}^{n} \frac{C_i}{T_i} \leq 1$$

  - Condition suffisante :

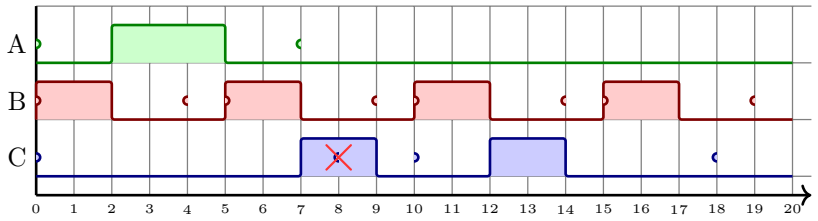
$$CH = \sum_{i}^{n} \frac{C_i}{D_i} \leq 1$$

  - Dans le cas des tâches à échéance sur requête, les deux conditions sont égales

Exemple

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	7
B	0	5	2	4
C	0	10	2	8

Par DMA :

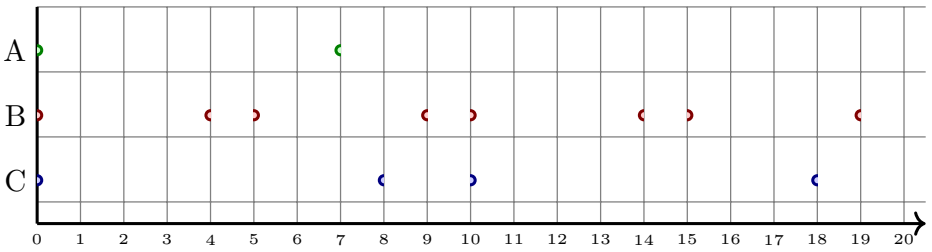


Par EDF :

Condition nécessaire :

Condition suffisante :

Par EDF :



5.3.2 Least Slack Time (LST)

Least Slack Time (LST)

- Algorithme à priorité dynamique
- Aussi appelé *Least Laxity First (LLF)*
- Basé sur la laxité résiduelle :
  - La priorité maximale est donnée à la tâche qui a la plus petite laxité résiduelle : $l_i(t) = D_i(t) - c_i(t)$
- Équivalent à EDF si on ne calcule la laxité qu'aux réveils des tâches

— Optimum à trouver entre la granularité du calcul et le nombre de changements de contexte provoqués

— Permet d'être parfois plus résistant aux erreurs

— Demande une connaissance de la capacité des tâches

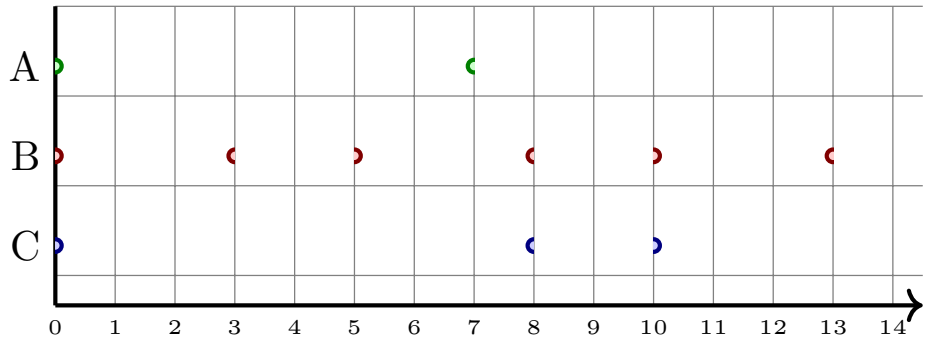
— Gain discutable

— Peu utilisé dans la pratique

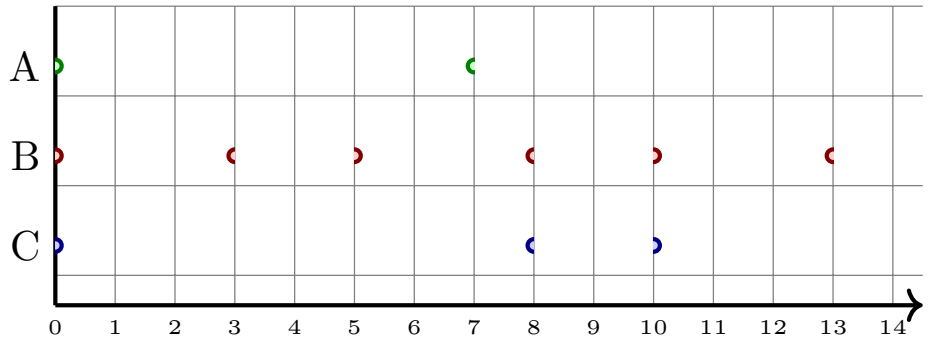
Exemple

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	1	7
B	0	5	2	3
C	0	10	3	8

Par EDF :



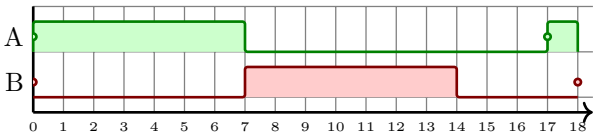
Par LST, en recalculant les priorités à chaque période :



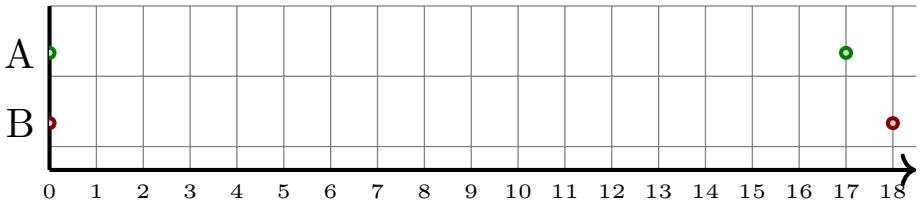
Exemple - 2

Tâche	Arrivée	Période	Capacité	Délai
A	0	17	7	Fin de période
B	0	18	7	Fin de période

Par EDF :



Par LST, en recalculant les priorités à chaque période :



5.4 Traitement des tâches apériodiques

Tâches apériodiques

- Tâches prises en compte dans une configuration comprenant déjà des tâches périodiques
- A priori, on ne connaît pas l'instant d'arrivée de la requête de réveil de la tâche apériodique
- Contraintes temporelles strictes ou relatives
- Buts à atteindre :
  - Maintenir le respect des échéances des tâches déjà présentes dans l'ordonnanceur
  - Si contraintes relatives : minimiser le temps de réponse
  - Si contraintes strictes : maximiser le nombre de tâches acceptées en respectant leurs contraintes

5.4.1 Ordonnancement conjoint

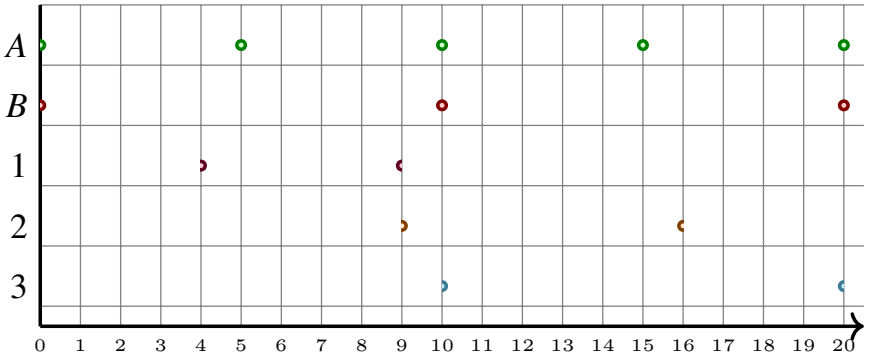
Ordonnancement conjoint

- Algorithme fonctionnant avec DMA, EDF ou LST et des tâches apériodiques avec des délais critiques
- On ordonnance les tâches apériodiques de la même manière que les tâches périodiques
- Avant d'accepter la tâche, il faut vérifier le critère d'acceptabilité (ce qui correspond à évaluer l'algorithme hors ligne)

Exemple : Ordonnancement conjoint

Avec ordonnancement EDF des tâches périodiques :

Tâche	Arrivée	Période	Capacité	Délai
A	0	5	2	Fin de période
B	0	10	2	Fin de période
1	4	-	2	9
2	9	-	2	16
3	10	-	3	20



5.4.2 Exécution en arrière-plan

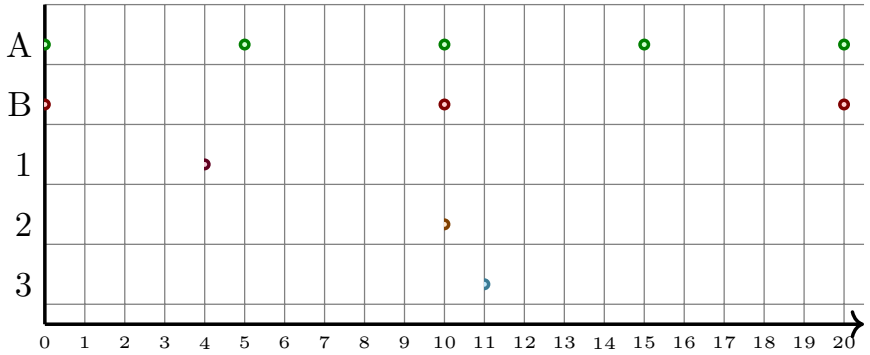
Traitement en arrière-plan

- Aussi appelé *backgound* ou sur *temps creux*
- Les tâches apériodiques sont ordonnancées quand le processeur est oisif
  - Les tâches périodiques restent les plus prioritaires
- Ordonnancement des tâches apériodiques en mode FIFO
- Traitement le plus simple, mais le moins performant
- Pas de marge de manoeuvre pour améliorer le temps de réponse des tâches apériodique. Potentiellement, les tâches apériodique peuvent avoir des temps de réponse long.

Exemple : Traitement en arrière-plan

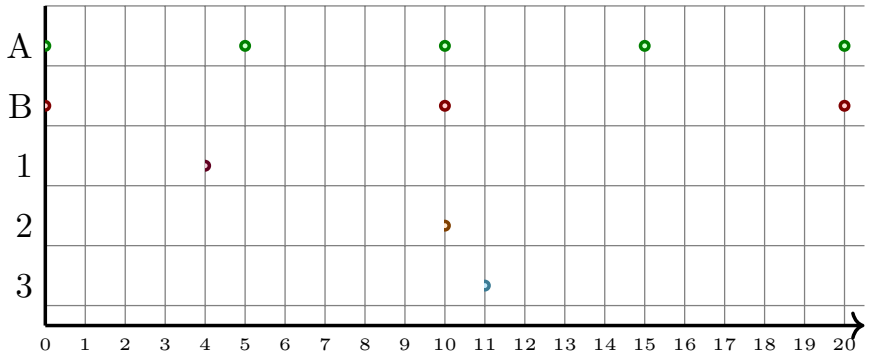
Avec ordonnancement RM des tâches périodiques :

Tâche	Arrivée	Période	Capacité	Délai
A	0	5	2	Fin de période
B	0	10	2	Fin de période
1	4	-	2	Aucune
2	10	-	1	Aucune
3	11	-	3	Aucune



Traitement en arrière-plan

Avec ordonnancement RM des tâches périodiques :



$TR_1 = ...$

$TR_2 = ...$

$TR_3 = ...$

5.4.3 Traitement par serveur

Traitement par serveur

- Un serveur est une tâche périodique créée spécialement pour prendre en compte les tâches apériodiques
- Un serveur est caractérisé par :
  - Sa période
  - Son temps d'exécution : capacité du serveur
- Un serveur est généralement ordonnancé suivant le même algorithme que les autres tâches périodiques
- Très souvent, afin de diminuer le temps de réponses des tâches apériodiques, la priorité du serveur est haute (sinon, ses performances sont identiques au traitement sur temps creux)
- Une fois actif, le serveur sert les tâches apériodiques dans la limite de sa capacité.
- l'ordre de traitement des tâches apériodiques ne dépend pas de l'algorithme général
- Il est possible de combiner ce mode avec un traitement en background (Temps de réponse+, prédictabilité-)

Serveur par scrutation

Serveur par scrutation

- Aussi appelé *polling*

— A chaque activation, traitement des tâches en attentes jusqu'à épuisement de la capacité ou jusqu'à ce qu'il n'y ait plus de tâches en attentes

— Si aucune tâche n'est en attente (à l'activation ou parce que la dernière tâche a été traitée) , le serveur se suspend immédiatement et perd sa capacité qui peut être réutilisée par les tâches périodiques

— Quand une instance (un événement) de tâche apériodique arrive, elle attend jusqu'à ce que la capacité du serveur soit disponible.

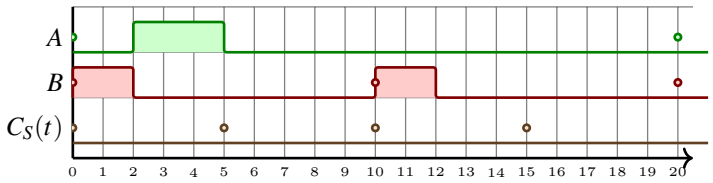
— Il est possible de rendre la main au CPU si aucunes taches apériodiques n'est en attente (TR des taches périodiques +, prédictabilité -)

— Dans le cas ou le serveur à la plus petite priorité, l'algorithme équivaut à peu près au traitement en background

Exemple

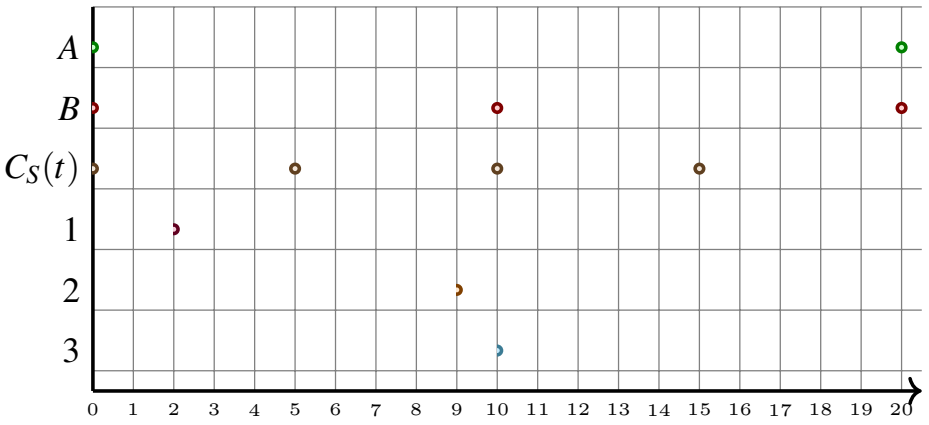
A vide, ordonnancement RM des tâches périodiques :

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	Fin de période
B	0	10	2	Fin de période
S	0	5	2	Fin de période

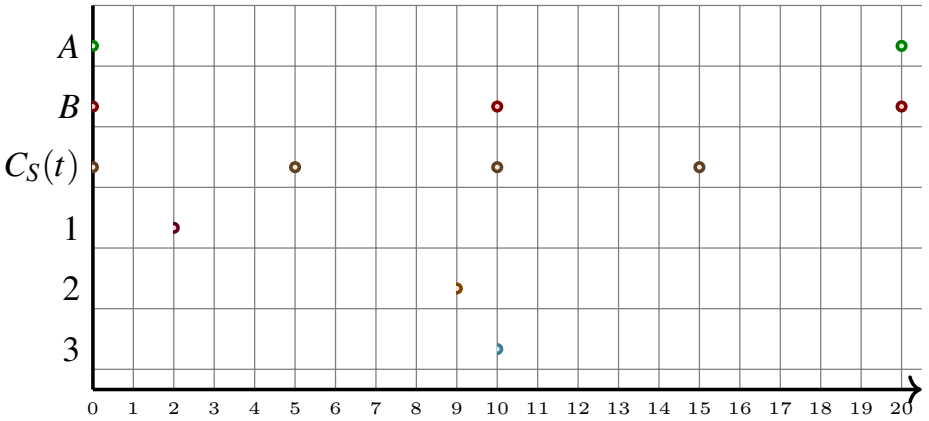


Exemple (avec les 3 tâches apériodiques)

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	Fin de période
B	0	10	2	Fin de période
S	0	5	2	Fin de période
1	2	-	2	Aucune
2	9	-	1	Aucune
3	10	-	3	Aucune



Exemple (avec les 3 tâches apériodiques)



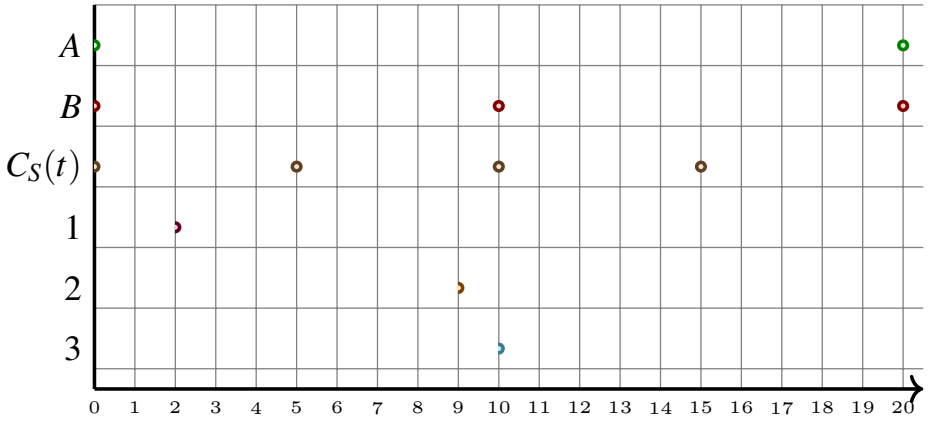
$TR_1 = ...$

$TR_2 = ...$

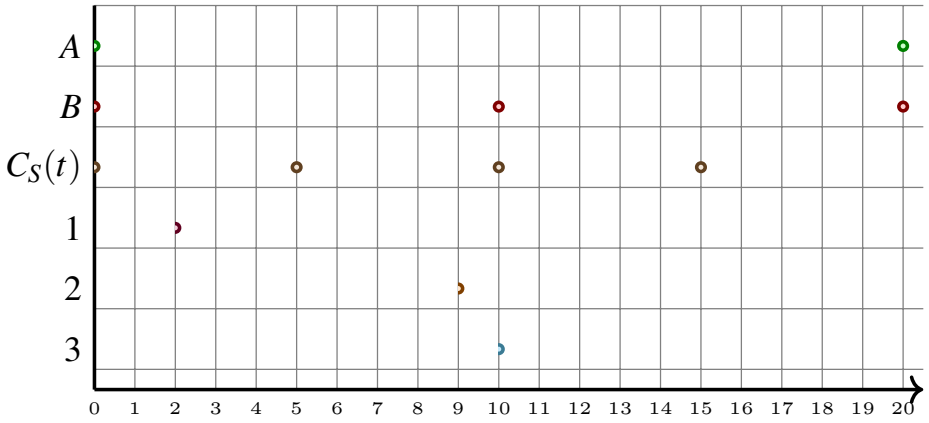
$TR_3 = ...$

Exemple (en utilisant les temps creux)

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	Fin de période
B	0	10	2	Fin de période
S	0	5	2	Fin de période
1	2	-	2	Aucune
2	9	-	1	Aucune
3	10	-	3	Aucune



Exemple (en utilisant les temps creux)



$$TR_1 = ...$$

$$TR_2 = ...$$

$$TR_3 = ...$$

Limitation

Limitations du serveur par scrutation

— perte de la capacité si aucune tâche apériodique en attente

— si occurrence d'une tâche apériodique alors que le serveur est suspendu, il faut attendre la requête suivante

Serveur différé

Serveur différé

— Aussi appelé serveur ajournable

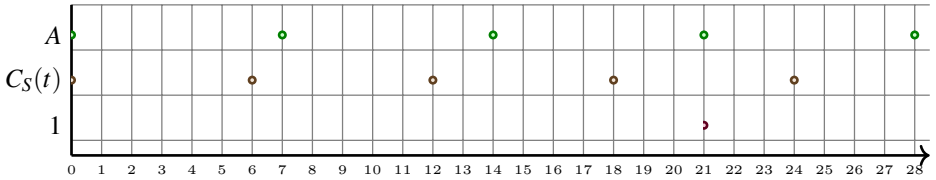
— Il peut provoquer des erreurs d'ordonnancement

— ...

Exemple

Avec ordonnancement RMA :

Tâche	Arrivée	Période	Capacité	Délai
A	0	7	2	Fin de période
S	0	6	3	Fin de période
1	21	-	6	Aucune



Serveur sporadique

Serveur sporadique

— Améliore le temps de réponse des tâches apériodiques sans diminuer le taux d'utilisation du processeur pour les tâches périodiques

— Très utilisé pour les IHM car permet une meilleure expérience utilisateur

— Comme le serveur différé mais

  — Ne retrouve pas sa capacité à période fixe,

  — La capacité retrouvée est égale à la capacité consommée

Serveur sporadique

— Calcul de la récupération de capacité :

  — le serveur est dit "actif" quand la priorité de la tâche courante $P_{exe} \geq P_S$,

  — le serveur est dit "inactif" si $P_{exe} < P_S$,

  — RT : date de récupération

    — calculée dès que le serveur devient actif(t_A)

    — $= t_A + T_S$

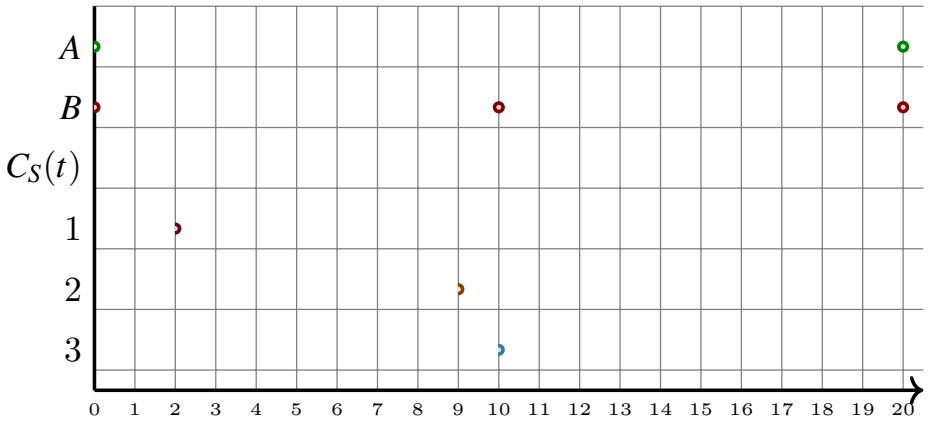
  — RA : montant de récupération à effectuer à RT

    — calculée à l'instant t_I où le serveur devient inactif ou que la capacité est épuisée

    — = la capacité consommée pendant l'intervalle $[t_A, t_I]$

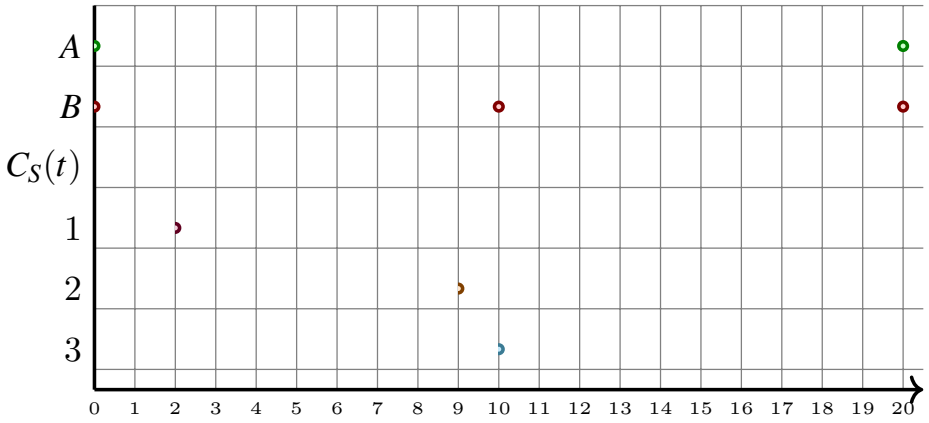
Exemple : Serveur sporadique (RMA)

Tâche	Arrivée	Période	Capacité	Délai
A	0	20	3	Fin de période
B	0	10	2	Fin de période
S	0	5	2	Fin de période
1	2	-	2	Aucune
2	9	-	1	Aucune
3	10	-	3	Aucune



Exemple : Serveur sporadique

Avec ordonnancement RMA :



$TR_1 = ...$

$TR_2 = ...$

$TR_3 = ...$

Création du noyau

6 Introduction à Linux embarqué

6.1 L'embarqué

Qu'est-ce que l'embarqué ?

D'après Wikipedia :

> Un système embarqué peut être défini comme un système électronique et informatique autonome, qui est dédié à une tâche bien précise. Ses ressources disponibles sont généralement limitées. Cette limitation est généralement d'ordre spatial (taille limitée) et énergétique (source d'énergie limitée).
> Les systèmes embarqués font très souvent appel à l'informatique, et notamment aux systèmes temps réel.
> Le terme de système embarqué désigne aussi bien le matériel que le logiciel utilisé.

6.2 Linux

Qu'est-ce que Linux ?

— Linux ne désigne que le noyau

— Linux est souvent associé aux outils GNU d'où le nom de GNU/Linux

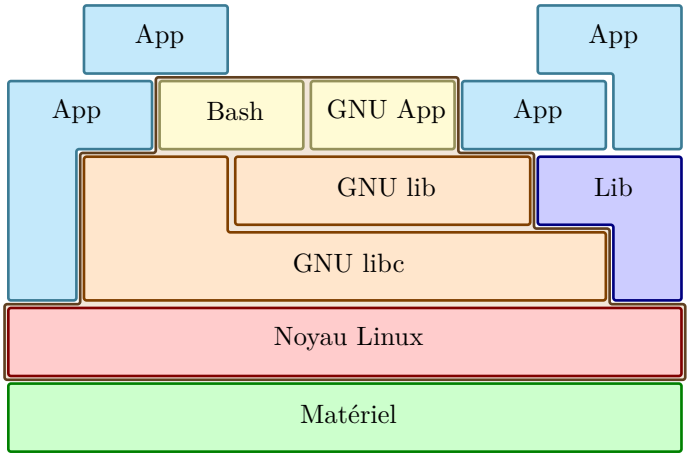
— Systèmes avec les outils GNU mais un noyau différent : GNU/Hurd, Solaris, etc...

— Systèmes Linux sans GNU : Android

— Le nombre de systèmes Linux installés est difficile à évaluer (en partie à cause des système Linux embarqués)

GNU/Linux

GNU/Linux est finalement un aggloméra :



La Norme Posix

- *Portable Operating System Interface [for Unix]*
- Uniformise les OS
- Première version publiée en 1988
- Souvent implémentée en partie
- ... et parfois s'en inspire simplement
- Posix ↛ Linux
- Linux ↛ Posix

Le Projet GNU

- Créé en 1983 par Richard Stallman
- Pose les bases politiques de GNU/Linux
  - GPL publiée en 1989
  - GPLv2 en 1991
  - GPLv3 en 2006
- `gcc` apparait en 1985
- `bash` et les Coreutils apparaissent en 1988 (inspirés de `sh` 1971/1977)
- Nombre d'architectures supportées incalculables

Le noyau Linux

- Créé en 1991 par Linus Torvalds
- Système communautaire
- 15 millions de lignes de code dans 30000 fichiers (+15%/an)
- Environ 1200 développeurs dans 600 entreprises (+35%/an)
- Environ 5000 contributeurs depuis la première version de Linux
- Environ 650 mainteneurs (c'est-à-dire responsbales d'une partie du noyau)
- Domaine d'application très large, du DSP au super-calculateurs en passant pas les grilles de calcul
- 24 architectures (= jeux d'instructions)
- Des centaines de plateformes
- Environ 1000 drivers
- Une centaine de versions publiées
- Environ 10000 contributions sur chaque version
- Enormément de "forks" et de version non-officielles

Qu'est-ce qu'une distribution ?

— Debian, Ubuntu, Meego, Red Hat, Suse, ...

— Un ensemble de programmes disponibles pour GNU/Linux

— Ensemble de normes et de procédure

— Permet de garantir le fonctionnement des programmes distribués

— Notre distribution "Hôte" : Debian/Ubuntu

— Notre distribution "Cible" : une distribution personnalisée

6.3 Introduction au Shell Linux

Quelques notions du Shell

— Sous Linux, la ligne de commande est un outil très performant

— Très souvent, c'est le seul langage de script disponible sur la cible

— Referez vous à vos fascicule de TP pour se rappeler des commandes Shell

— Consulter l'aide-mémoire, Format A3 (*Google Classroom* : `Aide-mémoire des commandes Shell`) dont vous disposez.

— Consulter l'aide des commandes par l'option `--help` ou la documentation système par la commande `man`.

— Fonctions utiles : `mkdir, rm, ls, tree, pwd, mv, nano, touch, wc, ...`

— On utilise beaucoup les redirections des entrées/sorties sous Linux (`< >, >>, |, < >`)

— Les chemins absolus et relatifs

— Expressions régulières, Globbing, Alias, script shell ...

Shell : Expressions régulières

— Utiliser `ifconfig` et `grep` pour extraire l'adresse IP :

```
# ip a | grep -Eo 'inet ([0-9]*\.){3}[0-9]*' | grep -Eo '([0-9]*\.){3}[0-9]*' | grep
↪  -v '127.0.0.1'
```

— avec `sed` :

```
# ip a | sed -En 's/127.0.0.1//;s/.*inet (([0-9]*\.){3}[0-9]*).*/\2/p'
```

— `ifconfig eth0 | ...` pour extraire l'IP de la carte réseau `eth0`

— créer notre propre commande `myip` dans le fichier .bashrc :

```
alias myip="ifconfig | sed -En 's/127.0.0.1//;s/.*inet (([0-9]*\.){3}[0-9]*).*/\2/p'"
```

— comparer avec l'instruction `hostname -I`

Shell : Scripting

Les fonctionnalités d'un script shell sont identiques à celles de la ligne de commande.

— Ouvrir un nouveau fichier. On suffixe généralement les shell par .sh

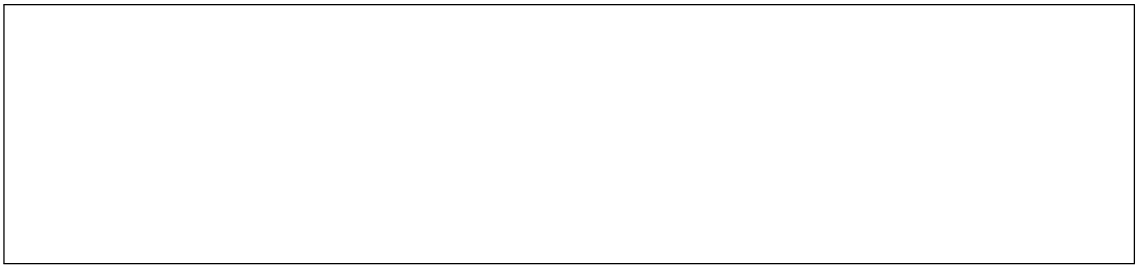
— Indiquer le shell utilisé sur la première ligne, préfixé de `#!`

```
#!/bin/sh
```

— La suite s'écrit comme sur la ligne de commande

```
echo TP embedded RTLinux
```

— Il est nécessaire de donner les droits en exécution au script



7 Le bootloader

7.1 Rôle d'un chargeur de démarrage

Le bootloader

Description

— Le bootloader se trouve souvent sur une *eeprom*. Celle-ci est directement mappée sur le bus d'adresse

— Au minimum, il doit initialiser :

  — les timing de la mémoire RAM

  — les caches CPU

Le bootloader

Description

— Il peut :

  — Initialiser une ligne série pour l'utiliser comme terminal

  — Offrir un prompt et accéder à des options de boot

  — Initialiser la mémoire flash

  — Copier le noyau en mémoire

  — Passer des arguments au noyau

  — Initialiser le chipset réseau

  — Récupérer des informations provenant d'un serveur DHCP (serveur où récupérer l'image du noyau, etc...)

  — Lire des fichiers provenant du réseau

  — Écrire sur la mémoire flash

  — Gérer un système de secours

  — Initialiser des fonctionnalités cryptographiques

Le bootloader

Description

— Il est très rare de pouvoir démarrer Linux sans bootloader fonctionnel

— Si votre bootloader n'est pas fonctionnel, vous aurez souvent besoin d'un matériel particulier pour le mettre à jour (un outil capable de flasher l'eeprom)

Bootloader connus :

— Grub

— Syslinux (et son dérivé Isolinux)

— U-Boot

— Redboot

— BareBox (successeur de U-Boot)

Le bootloader

Test

Testons notre bootloader :

— Démarrez une communication RS232 avec la cible

— Redémarrer la carte

```
[...]
Hit any key to stop autoboot:  0
```

— Obtenez la liste des commandes

```
uboot> help
```

— Attention aux caractères de contrôle (`<PgUp>`, `Ctrl+V`, ...)

  — Pas d'historique

  — Pas de flèches gauche/droite

— En cas de problème pour vous connecter, vérifiez vos paramètres RS232.

7.2 Démarrer par réseau

Démarrer par réseau

— Permet de travailler plus confortablement car évite les cycles de scp/flash

— Parfois, il faut démarrer la cible sous Linux pour pouvoir la flasher. C'est donc la seule manière de continuer.

— Fonctionne aussi très bien avec des PC

— Trois étapes pour démarrer un système

1. Le bootloader configure la carte réseau et place le noyau en mémoire
2. Le noyau s'exécute et monte un filesystem réseau
3. Le premier processus du système est lancé : `init`

7.2.1 TFTP

TFTP

Mise en place

— Identique au protocole `ftp` mais plus simple

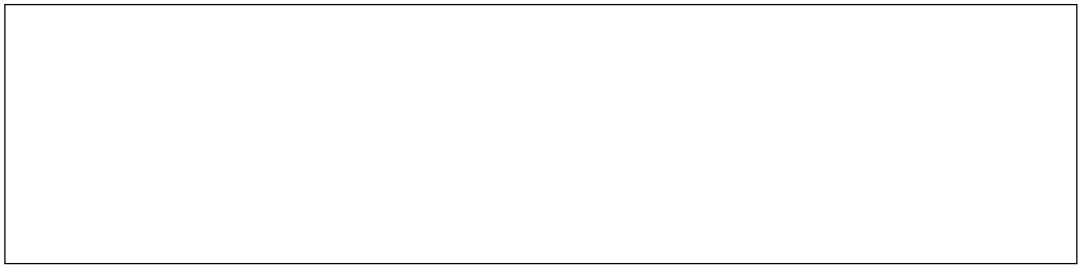
— Permet d'etre implémenté avec très peu de ressource

— Mise en place :

```
host% apt-get install tftp-hpa tftpd-hpa
host% cp ../Build/armhf/hello-static /var/lib/tftpboot/hello
```

— Test en local

— Par le shell interactif



— Par la ligne de commande

— Il est possible de modifier le répertoire partagé dans `/etc/default/tftpd-hpa`

— En cas de problème, consultez les logs (`/var/log/syslog` ou `/var/log/daemon.log`)

TFTP

Configuration de la cible pour télécharger et démarrer le noyau. Par RS232 :

— Nous supposons que :

— le répertoire `/var/lib/tftpboot/` contient aussi l'image `uImage` du noyau Linux,

— l'adresse IP du serveur est `192.168.1.10`,

— l'adresse IP de la carte cible est `192.168.1.12`

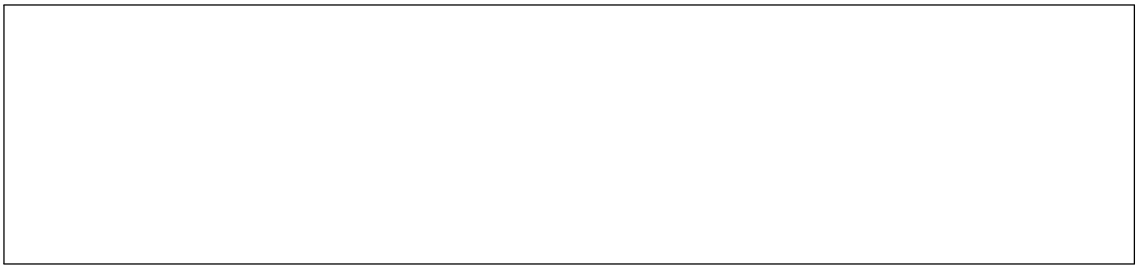
— Configuration de l'IP

— Vérification la configuration IP

```
uboot> ping 192.168.1.10
```

— Déclaration de notre *host* comme serveur `tftp`

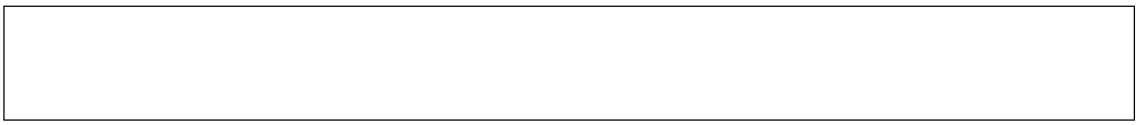
— Parfois il est nécessaire de passer la console au noyau :



TFTP

— Téléchargement du noyau dans une zone libre de la mémoire

— L'adresse de chargement du noyau dépend de plusieurs parmètres : topologie mémoire, utilisation de la mémoire par le bootloader, configuration du noyau, taille du noyau, etc..



— Exécution du noyau

— Le noyau trouve la flash, monte la flash et charge l'init de la flash

7.2.2 NFS

Nfs

Comparable au partage réseau de windows.

— Installation

```
host% sudo apt install nfs-kernel-server nfs-common
host% mkdir nfs
host% sudo nano /etc/exports
```

— Configuration du partage

```
/home/user/nfs 0.0.0.0/0.0.0.0(rw,no_root_squash)
```

Nfs

— Tester la configuration sur l'hôte

```
host% sudo exportfs -a
host% sudo systemctl restart nfs-kernel-server
host% mkdir nfs-mount
host% mount -t nfs 127.0.0.1:/home/user/nfs nfs-mount
```

— En cas de problème, vérifiez les logs : `/var/log/daemon.log`

Démarrage sur le NFS

— Modification des arguments passés au noyau

— Configuration IP

— Configuration NFS

— La variable `bootargs` permet de passer des arguments au noyau

— Démarrage

— Voir `Documentation/filesystem/nfs/nfsroot.txt`

— Après avoir monté le NFS, le noyau essaye de passer la main au programme `init`

8 Création d'un noyau personnalisé

8.1 Compilation du noyau

Récupération des sources

Où récupérer les sources du noyau ?

1. Utiliser les sources souvent fournies. Il arrive souvent qu'elles contiennent des drivers particuliers et qu'elles soient déjà configurées
2. Utiliser `git clone` ... ou
3. Télécharger le noyau :

```
host% wget https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.9.47.tar.xz
```

4. Extraire les codes sources du noyau :

```
host% tar xvf linux-4.9.47.tar.xz
```

5. Configurer ...
6. puis compiler ...

Configuration et personnalisation du noyau

— Avec Kconfig

```
host% make help
host% mkdir build
host% make O=build ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

— Beaucoup d'options, mais il y a l'aide (<h>) et la recherche (</>)

— La configuration est sauvegardée dans .config

— Charger une configuration pré-établie :

Exemple : `bcm2709_defconfig` permet de charger une configuration pour *Raspberry Pi 2, Pi 3, Pi 3+* et `bcm2711_defconfig` pour *Raspberry Pi 4* .

— Certains constructeurs fournissent un patch par cible `_defconfig`

— D'autres fournissent un .config

Personnalisation de la configuration du noyau

— Vérifier et modifier les paramètres du noyau `make ... menuconfig` :

— Cocher NFS ;

  - Désactiver la MMU `CONFIG\_MMU=n` ;
  - Activer le mode préemptif `PREEMPT\_RT\_FULL = y, HIGH\_RES\_TIMERS = y` ...

- Ou bien télécharger un *patch* temps réel du noyau :

```
host% wget
↪ https://cdn.kernel.org/pub/linux/kernel/projects/rt/4.9/older/patch-4.9.47-rt37.patch.xz
host% mv linux-4.9.47 linux-4.9.47-rt37
host% cd linux-4.9.47-rt37
host% xz -d ../patch-4.9.47-rt37.patch.xz
```

- et appliquer ce correctif par :

Compiler le noyau

- La compilation se lance avec :

- XX fait référence au format du binaire produit :
  - Le premier octet est-il du code ?
  - Respecte-t-il le format ELF ?
  - Y a-t-il un format particulier d'entête à respecter ?
- Dans le doute, il faut consulter la documentation de votre bootloader
- Compilation pour `U-Boot` (standard)

```
host% sudo apt install uboot-mkimage
host% make O=build ARCH=arm CROSS_COMPILE=arm-linux- uImage
```

- Partage de l'image par TFTP

Images productibles par la compilation du noyau

- `vmlinux` : L'image ELF du noyau. Lisible par les debugueurs, certains flasheurs, certains bootloaders
- `Image` : `vmlinux` strippé et préfixé par un mini-bootloader permettant de sauter sur la fonction `start_kernel` de `vmlinux`.

— `bzImage` et `zImage` : Comme `Image` mais compressé en `bz2` ou `gz`.

— `vmlinuz` : Normalement équivalent du `bzImage`.

— `xipImage` : Idem `Image` mais destiné à être exécuté directement sur un *eeprom* sans être copier en mémoire au préalable.

— `uImage` : `zImage` avec une entête spéciale pour *u-boot*.

Les modules

Deux mots sur les modules :

— Morceaux de noyaux externalisés

— Compilation avec `make [...] modules`

— Installation avec `make [...] INSTALL_MOD_PATH=~/rootfs modules_install`

8.2 Création de l'init

Démarrage du noyau

— A la fin du démarrage du noyau, ce dernier donne la main à l'exécutable déclaré avec `init=`. Par défaut, il s'agit de /sbin/init

— `init` ne se termine jamais

— Les arguments non-utilisés par le noyau sont passés à `init`

— On peut estimer que notre système démarre à partir du moment ou nous obtenons un shell (c'est en tous cas là que la plupart des intégrateurs Linux embarqué s'arrêteront)

— Du moins complexe au plus complexe à démarrer :

  — `init=/hello-static`

  — `init=/hello`

  — `init=/bin/sh`

  — `init=/sbin/init`

  — `systemd symlink`

Créer l'espace utilisateur

Créer l'arborescence

Nous travaillerons dans un répertoire vierge

```
host% mkdir nfs-root-mine
host% ln -s nfs-root-mine nfs
host% cd nfs-root-mine
```

L'arborescence classique sera :

— bin

— sbin

— usr/bin

— usr/sbin

— etc

— dev

— proc

— sys

— tmp

— var

Il est possible de ne pas respecter cette arborescence

Créer l'espace utilisateur

Créer l'arborescence

Après le démarrage, le noyau ne trouve pas l'init :

```
[...]
Kernel panic - not syncing: No init found.  Try passing init= option to kernel.
```

— Pour que la machine puisse démarrer il faut choisir le fichier `init`, on peut par exemple transférer les fichiers `hello-static` et `hello` et essayer de démarrer ...

— Liste des bibliothèques nécessaires pour le l'exécutable dynamique

```
host% arm-linux-ldd --root . hello
```

— Il faut aussi les transférer :

```
host% mkdir lib
host% cp /opt/arm-linux-.../lib/ld-uClibc-0.9.30.2.so lib
host% cp /opt/arm-linux-.../lib/libuClibc-0.9.30.2.so lib
```

— ...

8.3 Compilation de l'espace utilisateur

8.3.1 Installation de Busybox

Busybox

— Contient la plupart des binaires nécessaires pour démarrer un système

— Attention, ce ne sont pas les mêmes outils que sur PC. Il y a souvent des options non-implémentées ou des comportements différents

— Téléchargement

```
host% wget http://busybox.net/downloads/busybox-1.19.4.tar.bz2
host% tar xvjf busybox-1.19.4.tar.bz2
host% mkdir build
```

— On retrouve `Kconfig`

```
host# make O=build CROSS_COMPILE=arm-linux- menuconfig
```

Busybox

On trouve plein d'outils :

`addgroup, adduser, adjtimex, ar, arping, ash, awk, basename, bunzip2, bzcat, cal, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, crond, crontab, cut, date, dc, dd, deallocvt, delgroup, deluser, devfsd, df, dirname, dmesg, dos2unix, dpkg, dpkgdeb, du, dumpkmap, dumpleases, echo, egrep, env, expr, false, fbset, fdflush, fdformat, fdisk, fgrep, find, fold, free, freeramdisk, fsck.minix, ftpget, ftpput, getopt, getty, grep, gunzip, gzip, halt, hdparm, head, hexdump, hostid, hostname, httpd, hush, hwclock, id, ifconfig, ifdown, ifup, inetd, init, insmod, install, ip, ipaddr, ipcalc, iplink, iproute, iptunnel, kill, killall, klogd, lash, last, length, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, losetup, ls, lsmod, makedevs, md5sum, mesg, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktemp, modprobe, more, mount, msh, mt, mv, nameif, nc, netstat, nslookup, od, openvt, passwd, patch, pidof, ping, ping6, pipe_progress, pivot_root, poweroff, printf, ps, pwd, rdate, readlink, realpath, reboot, renice, reset, rm, rmdir, rmmod, route, rpm, rpm2cpio, run-parts, rx, sed, seq, setkeycodes, sha1sum, sleep, sort, start-stop-daemon, strings, stty, su, sulogin, swapoff, swapon, sync, sysctl, syslogd, tail, tar, tee, telnet, telnetd, test, tftp, time, top, touch, tr, traceroute, true, tty, udhcpc, udhcpd, umount, uname, uncompress, uniq, unix2dos, unzip, uptime, usleep, uudecode, uuencode, vconfig, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat`

Installation de Busybox

— Configurons le chemin de destination vers ~/nfs

```
host% make O=build CROSS_COMPILE=arm-linux-
host% make O=build CROSS_COMPILE=arm-linux- CONFIG_PREFIX=~/nfs install
```

— L'installation créé des liens symboliques vers les binaires `busybox`

— Sans Busybox, toutes ces binaires seraient séparées et dispersées sur une dizaine de sources

— Nous pouvons maintenant démarrer avec `init=/bin/sh`

— `init=/sbin/init` pose encore quelques problèmes.

8.3.2 Le process `init`

Configuration de `init`

Le fichier `/etc/inittab`

Il est possible de configurer `init` avec le fichier /etc/inittab :

— Le fichier inittab décrit l'ensemble des processus qui doivent être lancés au démarrage du système (par exemple, `/etc/init.d/boot, /etc/init.d/rc, ...`).

— Le programme init(8) distingue différents modes de démarrage (`runlevel`), chacun pouvant avoir ses propres processus à démarrer.

— Les modes de démarrage valides sont `0-6`, et `A, B et C` pour des modes personnalisés.

— Typiquement, une entrée dans le fichier `inittab` a la forme suivante :

`id:runlevels:action:process`

id Séquence unique de 1 à 4 caractères identifiant une entrée dans `inittab`.

Note : pour les programmes de connexion comme les `gettys` ou d'autres, le champ id doit être le numéro du `tty` correspondant à la console, par exemple 1 pour `tty1`.

runlevels Liste des modes de démarrage pour lesquels l'action doit être faite.

action Décrit l'action à faire.

process Spécifie la commande à exécuter. Si ce champ commence par le caractère `+`, `init` ne lancera pas les commandes `utmp` et `wtmp` pour enregistrer les connexions.

Exemple : `/etc/inittab`

— Lancement automatique d'un shell

```
host% echo '::askfirst:/bin/sh' > etc/inittab
```

— Appel d'un script de démarrage.

```
host% echo '::sysinit:/etc/init.d/rcS' >> etc/inittab
host% mkdir etc/init.d
host% echo '#!/bin/sh' > etc/init.d/rcS
host% chmod +x etc/init.d/rcS
```

Documentation disponible sur la page de man *inittab(5,8)*.

Des fichiers de configuration de init et d'autres utilitaires de busybox sont disponibles dans busybox/examples.

Autres `init`

Il existe d'autres formes d'`init` :

— SystemV

— runit (proposé par Busybox)

— upstart (utilisé autrefois par Ubuntu)

— systemd (utilisé par le reste du monde)

Ces `init`, plus modernes offrent de nouvelles fonctionnalités et plus de robustesse pour le système.

`init`

Fichiers de configuration

Nous pouvons maintenant démarrer avec `init=/bin/init` mais certaines fonctionnalités sont absentes (`ps`, `ifconfig`, `top`, `lsusb`, etc...)

Il faut monter les partitions `/proc` et `/sys` :

```
host% echo "::sysinit:mount -t proc none /proc" >> etc/inittab
host% echo "::sysinit:mount -t sysfs none /sys" >> etc/inittab
```

Automatisation du montage avec inittab :

Nos commandes semblent maintenant correctement fonctionner.

Utilisation de `fstab`

Il est possible d'automatiser ce montage au démarrage avec fstab et `mount -a`

Nous pouvons utiliser le fichier etc/inittab pour monter nos partitions automatiquement.

Filesystem temporaire

Créer un *filesystem* en mémoire permet de protéger notre flash (à durée de vie limitée), de garantir que nos systèmes seront toujours identiques entre chaque démarrage et d'améliorer les performances.

— Création

— Droits d'accès

```
host% chmod 777 tmp
host% chmod +t tmp
```

— Montage d'un filesystem contenu en mémoire

Les fichiers devices

— Permettent de communiquer avec le noyau

— Il représente plus ou moins chacun un périphérique

— Les plus importants sont normés (Documentation/devices.txt)

— Il est possible de les créer avec `mknod` :

```
host% mknod dev/console     c  5  1
host% mknod dev/ttyS0       c  4 64
host% mknod dev/random      c  1  8
host% mknod dev/mtdblock1   b 31  1
host% mknod dev/sda         b  8  0
host% mknod dev/sda1        b  8  1
```

Utilisation de `MAKEDEV`

`MAKEDEV` permet d'automatiser la création des fichiers devices de base

```
host% cd dev
host% MAKEDEV std
host% MAKEDEV console
```

Utilisation de `mdev`

— Intégré dans Busybox

— Uniquement depuis 2.6, nécessite /sys compilé et monté

— Permet de créer les devices à la volée

— Sur les systèmes très petits et où l'utilisation de device dynamique n'est pas nécessaire, on se passe de `mdev` à cause des dépendances avec le noyau

— Création de /dev sur un disque mémoire

— Initialisation /dev lors du démarrage

— Installation de `mdev` comme *handler* pour les nouveaux périphériques

Utilisation de `mdev`

Automatisation du processus

```
host% echo 'none /dev tmpfs' >> etc/fstab
host% echo "mdev -s" >> etc/rcS
host% echo "echo /sbin/mdev > /proc/sys/kernel/hotplug" >> etc/rcS
```

Résumé

— Toujours commencer par *hello-static*, le moins dépendant

— Si il ne fonctionne pas, recherchez du coté du format de binaire de la chaîne de compilation et avec sa compatibilité avec les options du noyaux

— Si *hello-static* fonctionne, mais pas *hello* en dynamique, cherchez du coté du format de la *libc* et de sa compatibilité avec le format de binaire de *hello*

— Si *hello* fonctionne mais que vous ne pouvez pas lancer de shell, cherchez du coté des devices et des droits.

— Si le shell démarre mais pas init, recherchez du coté des fichiers de configuration et des devices

— Vérifier que votre cible ne change pas d'IP en démarrant

— Sinon, cherchez dans les parametres passés au noyau ou dans la configuration

— Si possible, toujours tester entre chaque modification

8.4 Quelques ajouts

Résolution DNS

— Ajout de la résolution DNS

— Si nous Utilisions la *glibc* au lieu de la *uclibc*, il serait alors nécessaire de configurer le fichier /etc/nsswitch.conf. Il serait possible de choisir parmi différents backends pour gérer les authentifications et le réseau.

— Le reste de la configuration réseau s'effectue normalement dans /etc/network, puis en utilisant `ifup` et `ifdown`

— Pour utiliser le dhcp, ne pas oublier de recopier le fichier examples/udhcp/simple.script fourni avec busybox vers /usr/share/udhcpc/default.script

Ajouts d'utilisateurs

— Ajout d'utilisateurs :

```
host# echo 'root:x:0:0:root:/root:/bin/sh' > etc/passwd
host# echo 'root:x:0:' > etc/group
host# echo "::sysinit:login" >> etc/inittab
```

— `root::0:0:root:/root:/bin/sh` créerait un utilisateur sans mot de passe

— Possibilité de calculer les mots de passe avec `mkpasswd`. Voir `mkpasswd -m help`

— Possibilité de gérer les mots de passe dans `/etc/shadow` (*shadow(5)*) pour plus de sécurité :

```
host# echo 'root:x:::::::' > etc/shadow
```

8.5 Initramfs

Initramfs

— Qu'est-ce qu'un initramfs ?

— Un petit filesystème chargé en mémoire par le bootloader

— Successeur de initrd (plus complexe d'utilisation)

— Permet déffectuer des actions avant le lancement d'init

— Il devrait rester petit et ne s'utiliser que pour continuer le boot

— Si vous intégrez busybox dans votre initramfs, il est préconisé de le compiler en statique en ne gardant que les composant utiles

— Cas classique : charger les modules permettent d'acceder au filesystème :

  — M-Sys Disc-On-Chip

  — Configuration RAID

  — Configuration réseau pour booter en NFS

— Dans notre cas, il n'est pas utile car nous n'avons pas besoin de charger de driver particuliers avant le démarrage

Initramfs

— Création d'un rootfs vraiment minimal :

```
host# mkdir initramfs
host# cd initramfs
host# mkdir dev
host# mknod dev/console c 5 1
host% cp ../hello-static init
host# find . |  cpio -o | gzip > initramfs.gz
```

— Comme l'image noyau, l'image initramfs doit être au format u-boot.

Initramfs

— On indique l'adresse du ramfs en second paramètre de `bootm`. Évidement, le noyau doit être configuré pour utiliser le ramfs.

```
uboot> setenv autostart no
uboot> tftp 22000000 initramfs.gz.img
uboot> tftp 21000000 uImage
uboot> bootm 21000000 22000000
```

8.6 Flasher le rootfs

Recopier le système sur la flash

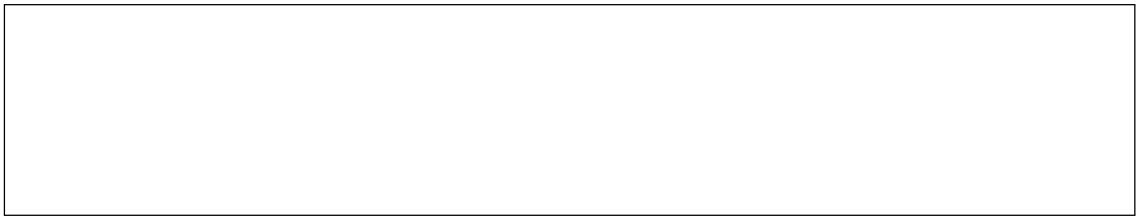
Il est possible d'effectuer cette opération à partir de U-Boot.

— Nous allons utiliser TFTP sans démarrer dessous. Désactivons l'autostart

```
uboot> set autostart no
```

— Toujours effacer la flash avant de flasher

— On place l'image en mémoire afin de la flasher



Recopier le noyau sur la flash

L'opération est similaire au filesystem. Dans la plupart des cas, on place le noyau à part sur une autre partition de la flash. Nous n'avons alors pas besoin de filesystem :

— Sous Linux

```
target% flash_erase /dev/mtd0 0x400000 0
target% cd /tmp
target% tftp -g -r uImage
target% nandwrite -s 0x400000 -p /dev/mtd0 uImage
```

— Sous U-boot