Temps Réel

Jérôme Pouiller <j.pouiller@sysmic.org>

Troisième partie III

Le multitâches

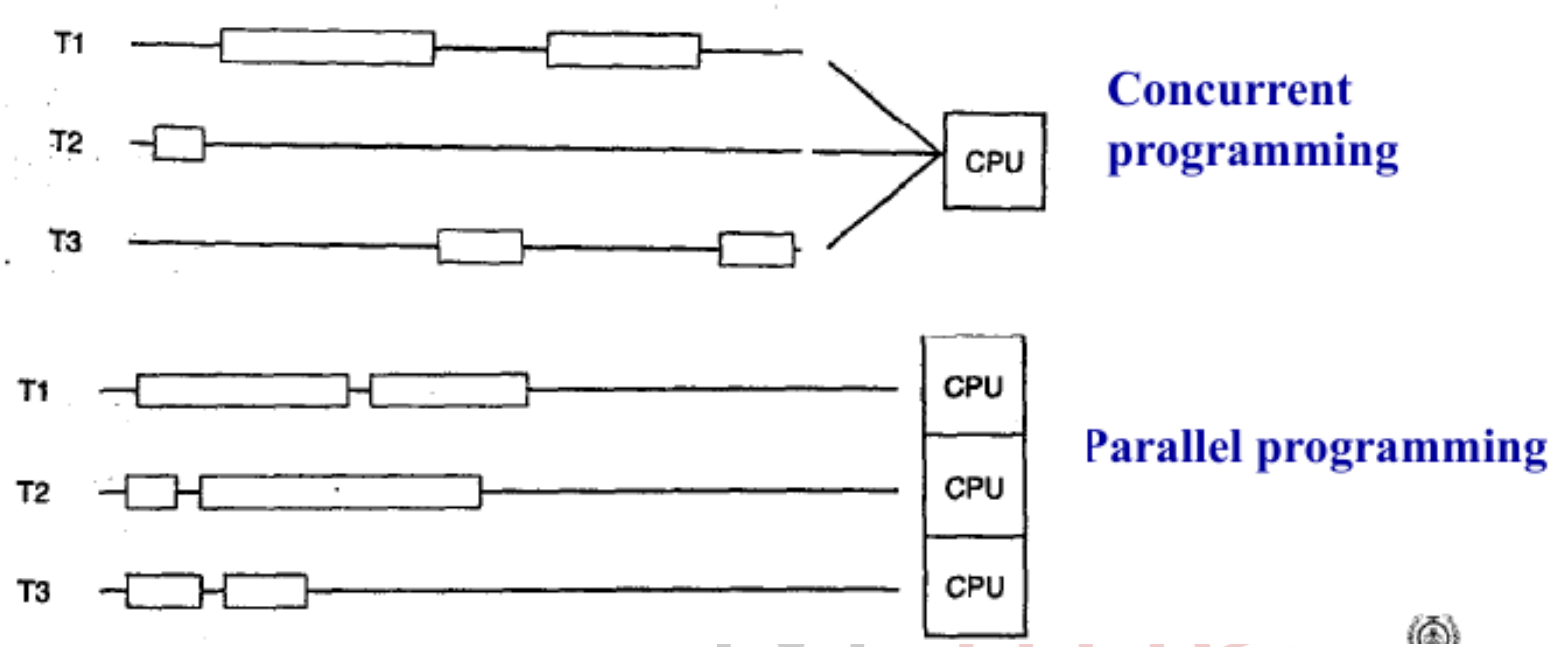
sysmic

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) :



Programmation multitâches

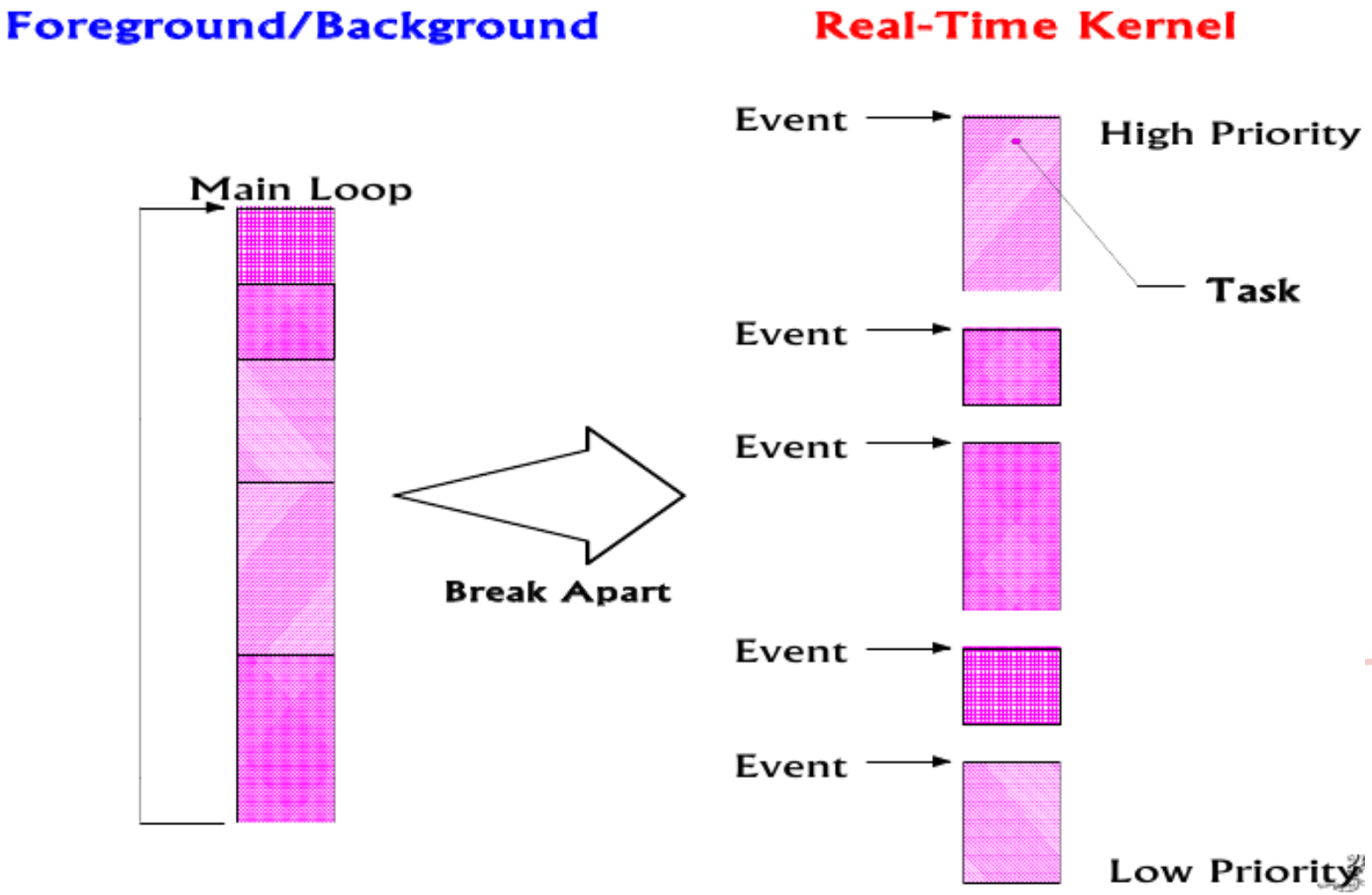
```
#include <unistd.h>

int main() {
  int r;

  r = fork();
  if (r < 0) {
     // Error
  } else if (r > 0) {
    // Parent
  } else /* r == 0 */ {
    // Child
  }
}
```

Concurrence

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

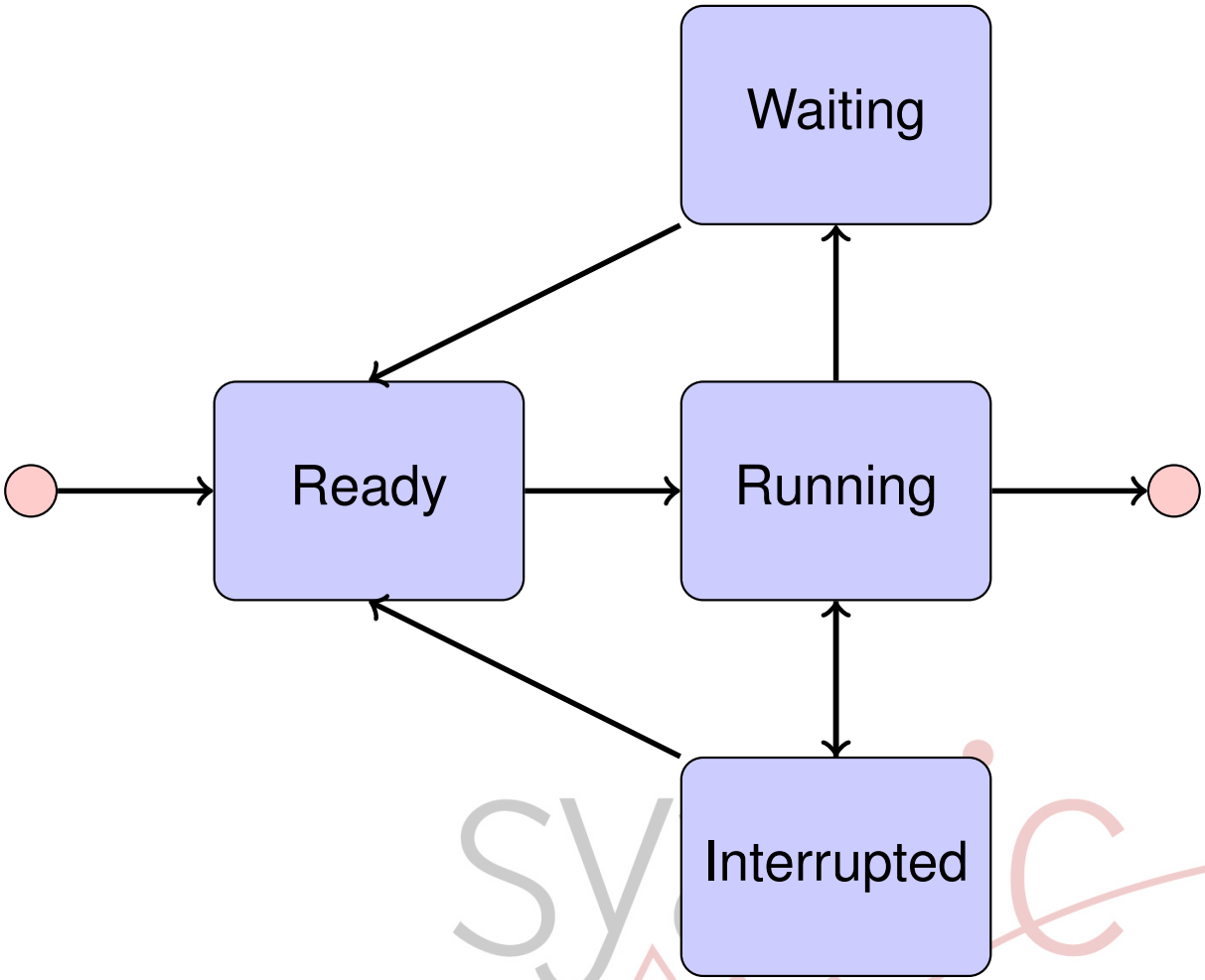


tâches concurrentes

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

- Facilite la gestion des évènements
- Permet de prioriser les traitements

États des tâches



Le changement de contexte

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

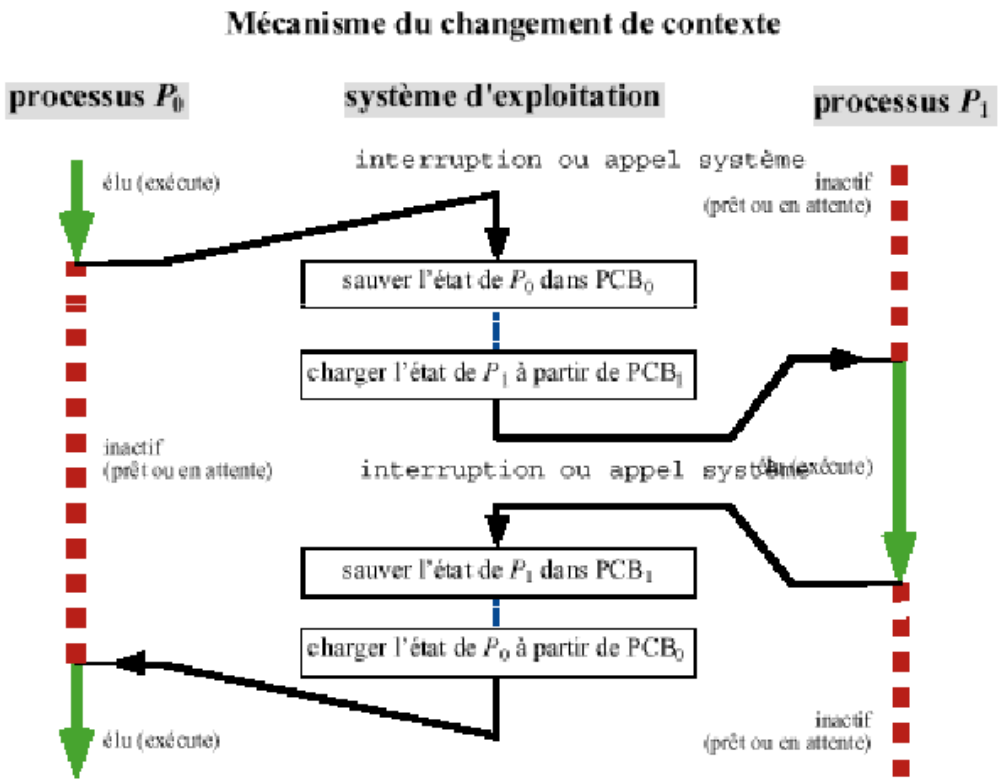
- les états de toutes les tâches
- l'emplacement de la pile en mémoire
- le contexte d'exécution, c'est-à-dire une sauvegarde des registres

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

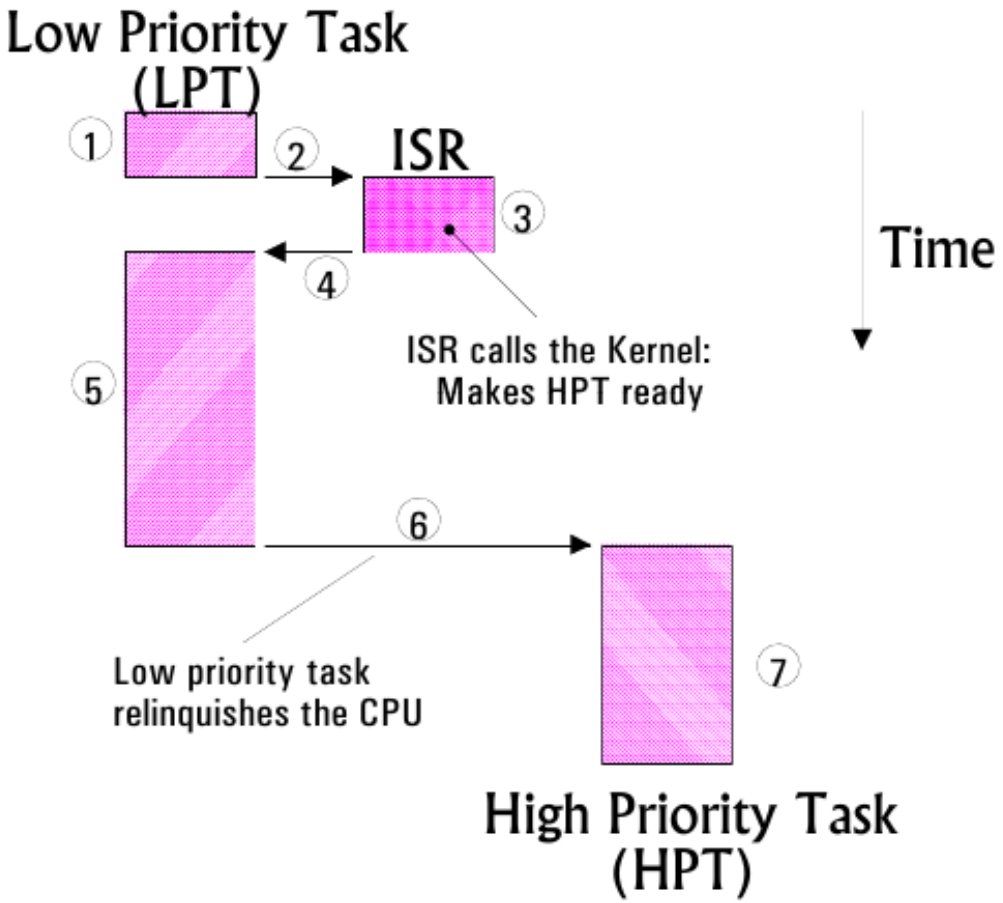


Multitâches non-préemptif

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 :

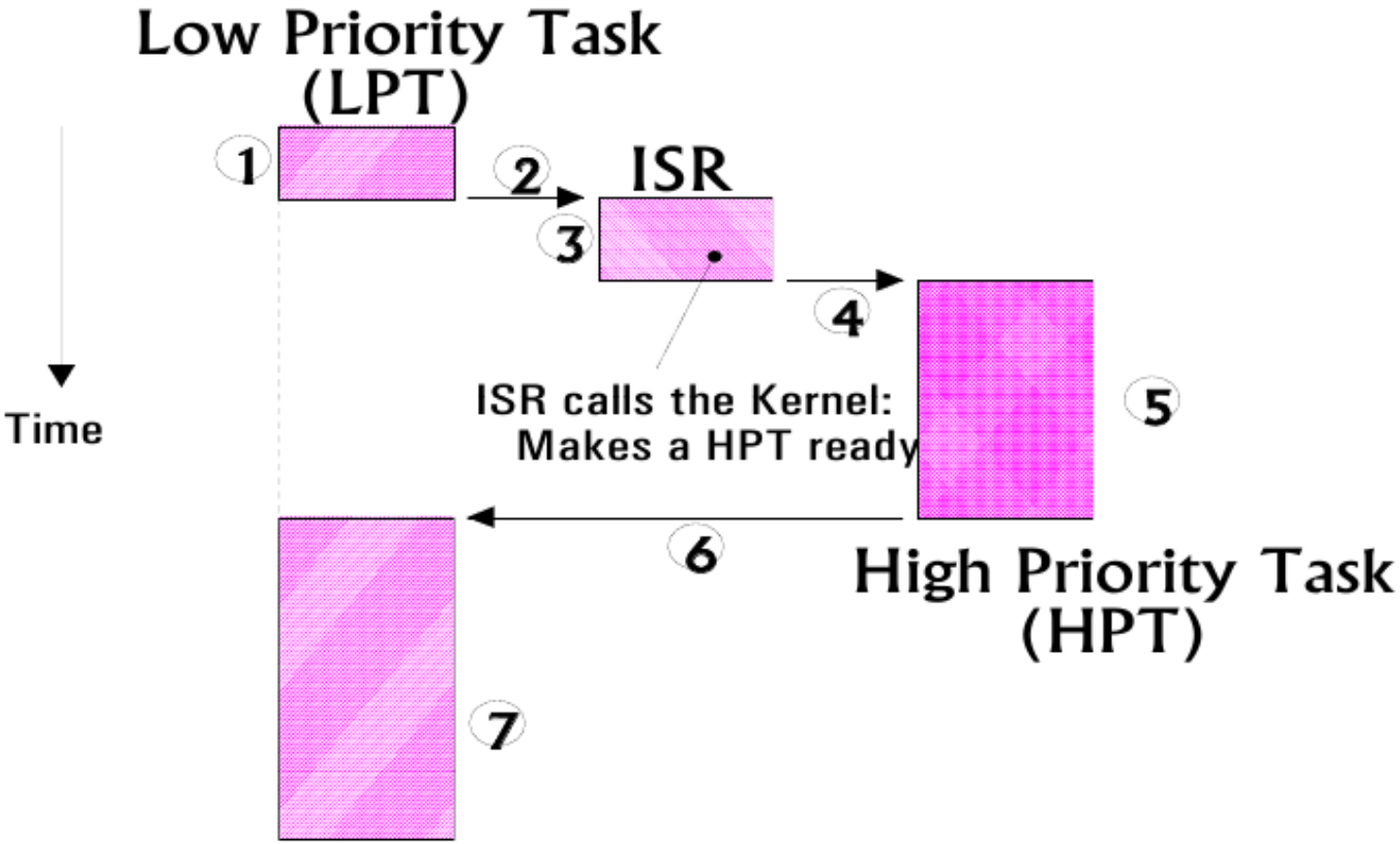


Multitâches non-préemptif

1. Une tâche non prioritaire est en cours d'exécution et est interrompue par un évènement (une IRQ)
2. L'ISR est appelée
3. Le traitement 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

Multitâches préemptif

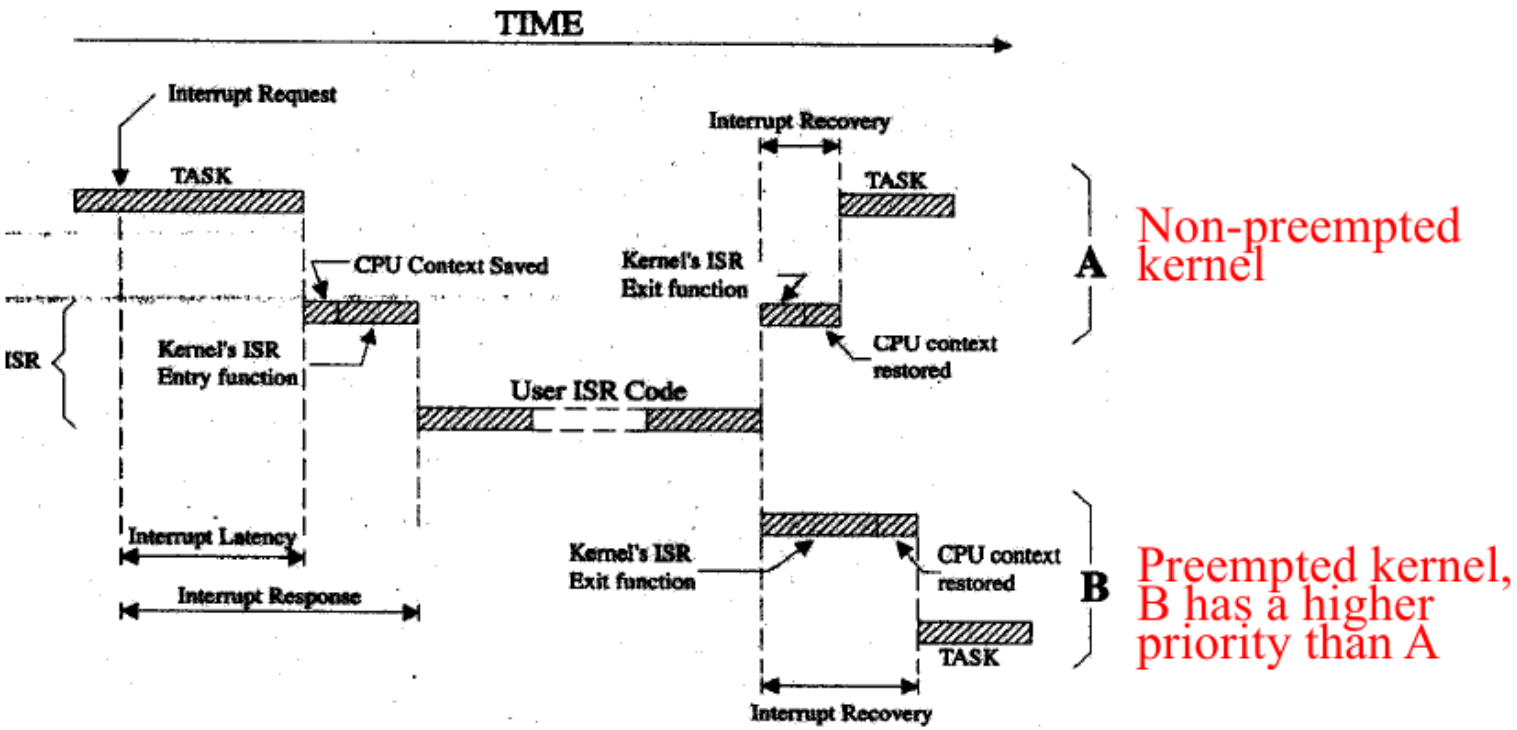
Un système multitâches préemptif va être capable de changer de contexte lors des interruptions :



Multitâches préemptif

1. Une tâche non prioritaire est en cours d'exécution et est interrompue par un évènement (une IRQ)
2. L'ISR est rappelle
3. Le traitement 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 tâche prioritaire à exécuter, l'ordonnanceur redonne la main à la tâche de faible priorité

Le changement de contexte sur interruption



Round robin

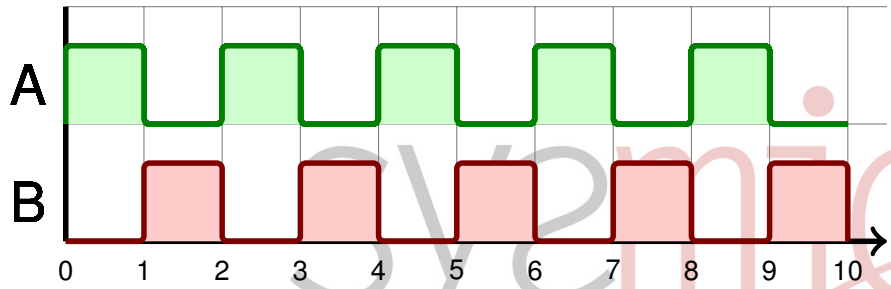
Examinons le cas de deux tâches de priorité égales n'effectuant jamais de relâchement volontaire :

```
task1() {
  for(;;) ;
}
task2() {
  for(;;) ;
}
```

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*.



La MMU

Le temps partagé permet de simuler que chaque tâche est la seule à utiliser le CPU.

En revanche, la mémoire est partagée entre les tâches. Ainsi, si une tâche A écrit par erreur sur l'espace d'une tâche B :

- La tâche B plante
- Le problème est complexe à trouver
- Il n'y a aucune moyen pour empêcher la tâche A de faire cette action.

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 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 et 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

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

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

- Lorsqu'une tâche accède à une adresse non associée. Une exception est déclenchée. Cela permet à l'OS de reprendre la main et de traiter l'erreur (souvent en tuant la tâche fautive)
- Lorsqu'une tâche souhaite allouer de la mémoire
  - La tâche demande à l'OS
  - L'OS choisi un (ou plusieurs) blocs de mémoire physique libres
  - L'OS marque le bloc comme appartenant à la tâche
  - L'OS choisi un espace d'adresse virtuelle où associer le bloc de mémoire
  - L'OS met à jour la MMU
  - L'OS retourne l'adresse virtuelle

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
- Tout est transparent pour 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 de manière transparente
- 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 (il faut alors gérer correctement le marquage *dirty* de la page)
- cf. champ *buffer* et *cache* de la commande `free`

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)*

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*)

Threads

Thread versus 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 beaucoup plus rapide

Utilisation de processus

```c
#include <unistd.h>

int main() {
  int r;

  r = fork();
  if (r < 0) {
     // Error
  } else if (r > 0) {
    // Parent
  } else /* r == 0 */ {
    // Child
  }
}
```

Utilisation de threads

```
#include <pthread.h>

void *task(void *arg) {
  int val = (int) arg;
  // Child
}

int main() {
  int arg = 42
  pthread_t id;
  pthread_create(&id, NULL, task, (void *) arg);
  // Parent
}
```

Multitâches, MMU et Temps réel

- 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`

```
#include <sys/mman.h>

void alloc_stack_1k() {
  char t[1024];
}

int main() {
  alloc_stack_1k();
  mlockall(MCL_CURRENT | MCL_FUTURE);
}
```