Systèmes temps réel
POSIX temps réel

Bertrand Dupouy

Plan

- **Rappels sur la gestion du temps dans les SE**
- La gestion du temps dans un STR
- Facteurs intervenant dans la gestion du temps
- Propositions POSIX
- Unix et le temps réel

Le temps dans un SE temps partagé

- Un système d'exploitation classique, c'est-à-dire orienté temps partagé, (SE, par la suite) doit organiser et optimiser l'utilisation des ressources de façon à ce que l'accès à ces ressources soit **équitable**.
- Les traitements qu'on lui soumet sont effectués en parallèle au fil de l'allocation des **quanta de temps** aux diverses applications.
- Un SE n'a donc pour seule contrainte de temps que celle d'un **temps de réponse satisfaisant** pour les applications

contrainte = temps de réponse

La date dans un SE temps partagé

Temps logique / temps « physqiue »

- Le SE est capable de **dater** les événements qui surviennent au cours du déroulement de l'application :
  - mise à jour d'un fichier, envoi d'un message,
  - suspension d'un traitement pendant un certain délai (`sleep` sous Unix), lancement d'une tâche à une certaine date.
- Mais, en aucun cas, il ne **garantit** que les résultats seront obtenus pour une **date précise,** surtout si ces résultats sont le fruit d'un traitement déclenché par un événement **extérieur, (**réel, physique : émission d'une interruption sur réception d'un signal matériel)
  - `sleep(t)` bloque un processus pendant **au moins** t unités de temps

Le temps dans un SE

- le SE ne prend pas en compte les contraintes d'échéances pour l'ordonnancement :
  - la **priorité** (cf. *Round Robin with variable priority*) qui est attachée à un processus l'est en fonction du **type d'événement** attendu et de sa **consommation** de temps cpu, elle ne prend pas en compte la date de réalisation de la tâche.

Plan

- Rappels sur la gestion du temps dans les SE
- **La gestion du temps dans un STR**
- Facteurs intervenant dans la gestion du temps
- Propositions POSIX
- Unix et le temps réel

Contrainte Supplémentaire des STR :

- Objectif des STR :
  - déterminisme temporel (en plus du déterminisme logique où les mêmes données en entrées donnent les mêmes résultats):
    - respect des échéances, prédictibilité : répondre à des contraintes temporelles (sur le début et/ou la fin des activités),

  Résultat correct = résultat exact … **et fourni à la date voulue**

- Exemples :
  - Cohérence temporelle à assurer pour la production de résultats, (synchronisation son/image),
  - Cadence à préserver pour la présentation de résultats : régularité pour la sortie d'images video

Evaluation d'un SE, d'un STR

• SE :

Les performances sont jugées suivant le **rendement** : exécuter le plus de tâches possibles, le plus rapidement possible,

C'est le SE qui décide de la dynamique d'exécution (CONTRAINTES LOGICIELLES)

• STR

Le critère de performance est le suivant : **respect de toutes ou d'une partie (en cas de surcharge) des échéances**, qu'elles soient périodiques ou non,

Si on exige le respect de toutes les échéances, on parle de TR dur, sinon TR souple (mou, *soft*).

C'est l'environnement extérieur qui impose sa dynamique (CONTRAINTES PHYSIQUES)

Sur l'ordonnancement dans les STR

• Contexte de l'ordonnancement :

- tâches périodiques, tâches apériodique :
  - périodiques, on connaît à l'AVANCE les contraintes de temps (les échéances),
  - apériodiques, il faut prendre en compte dynamiquement l'arrivée de traitements,
- combien d'échéances peut-on manquer ?
- Notion de qualité de la réponse :
  - moins précise, plus rapidement,
  - plus précise, moins rapidement,
  - on peut associer des VALEURS aux terminaisons d'activités, le comportement du STR est déterminé par un algorithme qui maximalise ces valeurs (*time values*),
- il faut exécuter au moins un sous-ensemble de tâches critiques, même si la charge augmente (*gracefull degrade*), l'objectif n'est PAS le rendement

Rapidité temps réel (1)

• *real time computing is not fast computing :*

FAUX	VRAI
Information soumise à des contraintes temps réel = information à obtenir **rapidement**	Information soumise à des contraintes temps réel = information à obtenir **avant** une certaine date
traitement temps réel = traitement à effectuer **rapidement**	traitement temps réel = traitement à effectuer **avant** une certaine date

• Une machine très rapide qui exécute les tâches sans en respecter toutes les échéances est moins « temps réel » qu'une plus lente qui donne tous les résultats « temps réel » à la date voulue.

Rapidité et temps réel (2) : illustration

• Soit deux machines :

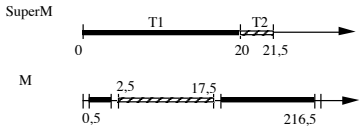
- *M,* durée changement de contexte = 0,5, ordonnancement : EDF
- *SuperM*, durée changement de contexte = 0 (!), processeur 10 fois plus rapide que celui de M,ordonnancement : FCFS

• Deux tâches T1 (prête en à t = 0, échéance t = 250) et T2 (prête en à t = 2, échéance t = 21), de même priorité, sont exécutées sur M et sur SuperM, ce qui donnera :

	Début	Echéance
T1	0	250
T2	2	21

Durée sur M	Durée sur SuperM
200	20, c.a.d (200/10)
15	1,5 c.a.d (15/10)

• Schéma d'exécution sur ces deux processeurs :



T2 ne respecte pas son échéance sur SuperM, tandis qu'elle le fait sur M, qui est plus lente mais dont le système utilise une politique d'ordonnancement adéquate.

Tâche urgente et tâche critique

• Chaque tâche a un degré :

- d'urgence, lié à la date de son échéance;
- de criticité, lié à son importance relative.

• Mais : une tâche très importante peu avoir de faibles contraintes de temps et tâche peu importante de fortes contraintes de temps

Pb. des ordonnancements du type RMS où le seul critère est la période.

• Comment conjuguer ces deux critères ? c'est à dire comment refléter l'urgence ET l'importance des tâches ?

- MUF (Maximum Urgency First),

Mixte criticality
ARINC 653 (systèmes partitionnés)

Ordonnancement Hors-ligne, en ligne

• Hors–ligne :

- l'ordonnancement est calculé **a priori**, c'est à dire avant l'exécution (*time driven scheduling*), l'ordonnanceur se réduit à un séquenceur,

• En–ligne :

- l'ordonnancement décidé à **l'exécution**, la détection des surcharges est plus difficile,

• Ne pas confondre :

- Hors ligne / en ligne,
- Priorité fixe / priorité dynamique,
- Préemptif / non préemptif,
- Priorité / criticité,

Réalisations d'application TR

Objectifs :

• Déterminisme temporel, il faut donc :

- maitriser les temps d'exécution (début/fin),
- garantir l'ordre d'exécution des fonctions,
- prouver l'ordonnnançabilité,

• Lisibilité, maintenance, il faut donc :

- éviter l'assembleur qui n'est pas portable et difficile à maintenir,
- utiliser des techniques d'ordonnancement, de gestion de la concurrence éprouvées,
- ne pas se contenter des tests qui ne sont pas toujours assez près des conditions réelles (cf. accumulation des dérives d'horloge dans une application très longue)

• Sûreté de fonctionnement, il faut pouvoir :

- détecter les erreurs,
- corriger les erreurs (cf. redondance matérielle)

– confiner les erreurs

Services attendus D'un STR ou d'une API TR

• **Les services** attendus d'un système, d'une API temps réel :

- politiques d'ordonnancement respectant les échéances,
- calcul de l'ordonnançabilité d'un jeu de tâches (contrôle d'admission),
- gestion adaptée du partage de ressources (sémaphore du type PIP ou PCP) ,
- traitement des événements (périodiques ou non),
- prise en compte du temps dans les entrées-sorties, et dans les communications (bus/ réseau temps réel, intergiciel temps réel),

• **L'API va** fournir des outils qui permettent d'adapter le modèle de l'application temps réel (lot de tâches RMS et synchro. PCP, par exemple à adapter sur un ordonnancement par priorité) au support fourni par l'architecture logicielle de base :

- rôle d'une *runtime* Ada,
- threads POSIX
- JVM temps réel

Limites des services Rendus:

• **Limites de ces services** :

- ils peuvent être plus ou moins élaborés, par exemple l'ordonnancement peut prendre en compte:
  - o la différence entre criticité et priorité (une valeur peut être associée à la terminaison d'une activité pour en évaluer l'importance),
  - o le traitement des surcharges, …
- ils ne **suppriment pas tous** les facteurs d'indéterminisme, il faudra prendre aussi en compte :
  - o la gestion (logicielle et matérielle) de la mémoire (le *garbage collector* des machines Java, l'utilisation de la mémoire virtuelle et des caches, …) doit être repensée pour le temps réel,
  - o le traitement matériel des entrées-sorties et des interruptions,
  - o éventuellement, le fonctionnement du processeur (pipe line),

Allowance
↳ Si une tache dépasse son tps d'exec et que les autres taches ne sont pas dans leur WCET doit-on la laisser s'exec ?

Quel langage pour les applications temps réel ?

• Langage dédié ou généraliste ?

• Services attendus par l'utilisateur :

- • expressions des contraintes temporelles,
- • expression et gestion du parallélisme,
- • spécification des périphériques à bas niveau (description des structures de données au niveau bit),
- • environnement de développement croisé (pour l'embarqué),
- • interface avec les autres langages,

• Ada, Java temps réel (RTSJ), API RT-POSIX

Plan

• Rappels sur la gestion du temps dans les SE

• La gestion du temps dans un STR

• **Facteurs intervenant dans la gestion du temps**

• Propositions POSIX

• Unix et le temps réel

Facteurs intervenant dans la gestion du temps

• Facteurs logiciels :
- – entrées-sorties (pas de priorité),
- – gestion de la mémoire : (GC ou pagination),
- – gestion des fichiers (caches, cf. mode `block` de Unix),
- – gestion des disques (algorithme de parcours, allocation),
- – synchronisation (partage de ressources),

• Facteurs matériels :
- – gestion des interruptions,
- – mémoires caches et TLB,
- – pipe-line,
- – entrées-sorties matérielles,

• et encore :

- format de l'exécutable
  - o édition de liens statique/dynamique,
- charge de la machine,
- protocole réseau pour les SE répartis
- la mesure doit-elle se faire dans la configuration la plus défavorable (*worst case execution time, WCET*) ?

Gestion de la mémoire

• Gestion statique :

- le nombre, la taille et l'emplacement des objets sont connus, ou bornés, lorsque l'application est lancée,
- avantage : accès aux objets en temps constant et faible (objets implantés sous forme de tableaux)
- inconvénient : pas flexible

• Gestion dynamique :

- avantage : souplesse
- inconvénients :
  - temps d'accès et d'allocation difficile à prédire ou à borner,
  - fragmentation (respect de la localité ?,

• Remarques :

- Impact sur le temps de changement de contexte
- Attention à l'édition de liens dynamique

Gestion dynamique : Allocation contigue

• En allocation contigue :

- comment gérer la liste des blocs libres et occupés ?
- le temps d'allocation (c.a.d de placement dans les blocs libres) est variable,
- si plusieurs placements sont possibles, quel algorithme de placement choisir : First fit, best fit, worst fit ?
- si il n'y a pas assez de place en mémoire :
  - compactage de trous
  - ramasse miette (*garbage collecting*) mais problème de déterminisme (cf. *real time gc* pour RT Java)
  - *swapping* (swap out) d'un ou plusieurs processus

Gestion dynamique : Pagination

• Problèmes :

- temps d'accès aux informations difficile à prédire :
  - pagination à deux niveaux,
  - utilisation d'un cache de pagination : *TLB*,
- ce temps d'accès peut être très long : accès disques possibles,
- la pagination est donc peu utilisée en TR, ou bien avec des mécanismes de verrouillage des pages en mémoire

• Rappel sur la traduction d'adresse :

- hit sur le TLB ?
  - oui, on y trouve le numéro de page physique
  - non, accès aux tables. Au pire un accès disque et aun accès mémoire

Mémoires caches et temps réel

• Avantage :

- diminue le temps d'exécution des tâches de manière probabiliste (cf. *hit ratio*),

ARINC → certification de l'OS et des matériels qui tournent dessus.
OSEC

• Inconvénients :

- augmentation du temps de changement de contexte (réinitialisation des caches) si les espaces d'adressage sont séparés, d'où utilisation de *threads* dans les STR
- moins de prédictibilité, il existe diverses méthodes d'estimation du comportement des caches, par exemple, classement des instructions :
  - *compulsory hit, compulsory miss*
  - *first hit, first miss*

• Techniques mises en œuvre :

- verrouillage, partage des caches,
- techniques différentes en intra-tâche et inter-tâches,

Entrées sorties et temps réel

• Perturbations dues au fonctionnement des entrées-sorties :

- le *DMA* (*direct memory access*) n'est pas déterministe à cause des problèmes de latence sur le bus,
- priorité des entrées-sorties non liée à celle des processus,
- utilisation de caches par le *file system* (et les périphériques),

• On doit pouvoir gérer les E/S comme on gère les processus :

- la priorité d'une E/S doit correspondre à celle du processus qui l'a demandée,
  - annuler une demande d' E/S lorsque son échéance est dépassée,

satisfaire la plus prioritaire d'abord,

Place des interruptions

• Remarque : dans la réalité il n'y existe pas d'évènement se déroulant en temps nul !

• Conséquence : bien qu'elles ne soient pas fondamentales, l'ignorance des caractéristiques temporelles du matériel peut perturber le fonctionnement de l'application, par exemple :

- les latences dues aux interruptions
- le fonctionnement de l'horloge dont peut prendre en compte les interruptions par l'ajout d'une tâche périodique

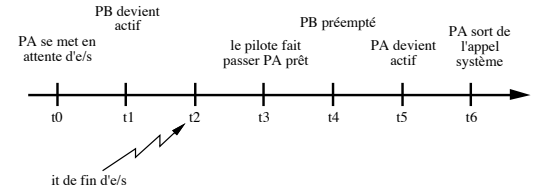
Temps de réponse à un événement

<table>
<tr><td colspan="4">Temps de réponse de l'application à un événement</td></tr>
<tr><td colspan="3">Dispatch latency</td><td></td></tr>
<tr><td colspan="2">Temps de réponse de l'interruption</td><td>Temps de commutation ou de préemption</td><td>Temps passé dans le processus élu</td></tr>
<tr><td>Latence interruption</td><td>Traitement interruption</td><td>Aller chercher le processus le plus prioritaire.
Changement de contexte.</td><td>Entrer dans le traitement écrit par l'utilisateur.</td></tr>
<tr><td>Terminer instruction en cours.
Sauvegarde du contexte.
Inhiber les IT.
Aller dans la procédure de traitement de l'IT.</td><td>Le traitement de l'IT envoie un message au processus bloqué.

Puis RTI.</td><td></td><td></td></tr>
</table>

Exemple

• Soient deux processus P_A et P_B, seuls sur le système. P_A est le plus prioritaire ; au temps t0, il lance une entrée-sortie, donc passe dans l'état bloqué. Le dessin ci-dessous schématise la suite des événements :



t0 à t1	P_A passe dans l'état bloqué et changement de contexte en faveur de P_B
t2 à t3	Gestion de l'it de fin d'E/S, qui finit par faire passer P_A dans l'état prêt
t3 à t4	temps de préemption pour arrêter P_B
t4 à t5	temps de changement de contexte en faveur de P_A
t5 à t6	temps de retour de l'appel système qui avait bloqué P_A (read, par exemple), ce temps dépend du nombre de paramètres, etc

Les élements du temps de réponse

temps de commutation	changement de contexte entre deux tâches de même priorité
temps de préemption	(passage à l'exécution d'une tâche plus prioritaire, différent du précédent : reconnaître l'événement, élire la tâche)
temps de latence des interruptions	(délai entre la reception de l'it et l'exécution de la première instruction du sp de traitement)

latence ordonnancement ou *dispatch latency time*	(délai entre la réception de l'it et l'entrée dans la tâche de traitement écrite par l'utilisateur)

sémaphore shuffle	(temps écoulé entre la libération d'un sémaphore et la réactivation de la tâche qui était bloquée sur celui-ci)

- en TR, tous ces éléments doivent être **prédictibles** et **finis**.
- Rappel : la rapidité n'est pas une condition suffisante pour le respect des échéances,

Récapitulatif : Ce que doit proposer le matériel

- Le support que doit proposer l'architecture matérielle :
  - gestion fine des horloges et des *timers*,
  - moyens de prise en compte précise des interruptions,
  - outils de contrôle des caches mémoire (inhiber, geler, …),
  - gestion efficace des E/S,

Plan

- Rappels sur la gestion du temps dans les SE
- La gestion du temps dans un STR
- Facteurs intervenant dans la gestion du temps
- **Propositions POSIX**
- Unix et le temps réel

POSIX

- Cette norme (*Portable Operating System Interface*) proposée par l'IEEE définit un jeu d'appels système,
- POSIX était destiné aux systèmes UNIX, mais de nombreux systèmes d'exploitation proposent cette interface (Windows NT par exemple),
- La section POSIX 1003.1c (ex POSIX 4.a) concerne les aspects threads et POSIX 1003.1b (ex POSIX 4.b) ajoute des extensions pour le temps réel,
- Points principaux :
  - plusieurs politiques d'ordonnancement, (FIFO, serveurs sporadiques, …),
  - protocoles de gestion de la concurrence : PIP et PCP,
  - gestion mémoire : fonctions de verrouillage de pages en mémoire et des outils de partage de l'espace d'adressage,
  - signaux et des timers déterministes,
  - entrées-sorties asynchrones et fichiers « mappés » en mémoire,
  - fonctions de trace.

Propositions POSIX

- Pourquoi les threads :
  - partage d'informations facile,
  - changement de contexte court,
  - utilisation des architectures SMP (*symetric multi processor*).
- Norme POSIX 1003.4 pour la portabilité des applications TR :
  - définit une interface standard entre l'application et le système,
  - ne spécifie PAS l'implantation, mais propose des outils de mesure des performances
  - **peu de fonctionnalités obligatoires**,
- POSIX 4 définit la panoplie TR minimale, 4a les threads et 4b les extensions. POSIX 4b propose des outils tels que :
  - l'accès direct aux interruptions depuis les applications,
  - l'ordonnancement "serveur sporadique",
  - les ordonnancements: SCHED_FIFO, SCHED_RR, SCHED_OTHER
  - une fonction qui permet à un thread de suivre la consommation cpu d'un autre,
  - les files de message (`mq_open, mq_receive,` ...)
  - les signaux temps réel

Propositions POSIX.4 (temps réel)

- Objectif : définir une interface standard pour rendre les applications **portables**
- Services proposés:
  - threads, ordonnancement, synchronisation,
  - files de messages, signaux,
  - gestion du temps, e/s asynchrones, gestion mémoire
- Problèmes :
  - la norme POSIX.4 contient plus de parties optionnelles que de parties obligatoires !
  - différences d'implantations (cf. les threads Linux (*clone*) et Solaris),
- Implantations : Linux, Solaris, de nombreux systèmes temps réel (Lynx, VxWorks, RTEMS, ...)

Portabilité et POSIX.4 (rôle des standards ?)

- Norme POSIX pour la portabilité des applications TR :
  - définit une interface standard entre l'application et le système,
  - ne spécifie PAS l'implantation,
  - mais propose des outils de mesure des performances
- POSIX 4a définit une panoplie TR **minimale**,
- Pour vérifier si la partie de la norme que l'on veut utiliser est bien implantée, utiliser `ifdef` et `error` pour avoir être averti par le préprocesseur, ou `sysconf` pour un message à l'exécution:

```
#include <unistd.h>
#ifndef _POSIX_PRIORITY_SCHEDULING
#error POSIX : pas d'ordonnancement TR
#endif
```


Ordonnancement POSIX.4

- priorités fixes avec préemption, 32 niveaux doivent être proposés,
- les politiques de gestion des files d'attentes associées à ces priorités sont : `FIFO, RR, OTHERS`
  - → les politiques d'une tache dans son niveau de priorité
- seul l'utilisateur privilégié (*root*) peut accéder à ce service d'ordonnancement pour choisir `FIFO` ou `OTHERS,`
- toutes ces fonctions sont applicables aux threads, quand on peut en appliquer aux processus aux processus, son nom ne contient pas le mnémonique `pthread,`

Threads POSIX.4

• Exemple de modification de politique d'ordonnancement et de priorité:

```
...
pthread_t             tid;
pthread_attr_t        attr;
struct sched_param    param;
...
pthread_attr_init (&attr)

/******** politique d'ordonnancement ********/
(#include <sched.h>)
pthread_attr_setschedpolicy(&attr, SCHED_FIFO)

/******** priorité du thread ********/
param.sched_priority = 1;
pthread_attr_setschedparam (&attr, &param)


/******** création du thread ********/
pthread_create (&tid, &attr, fonc, NULL)
```

• Divers :

- Niveaux de priorité

```
prio_max = sched_get_priority_max(policy);
prio_min = sched_get_priority_min(policy);
```

- Quantum pour la priorité RR

```
struct timespec qtm;
sched_rr_get_interval_max(0,&qtm);
```


pthread_detach: "Je ne ferais pas de join sur a thread, donc lorsqu'il a fini, on peut libérer les ressources qui lui sont associées".

Les `mutex` et les `sémaphores`

• Les **mutex** sont destinés à la gestion des accès aux sections critiques, la file d'attente qui leur est associée est gérée par ordre de priorités décroissantes, ils peuvent être utilisés **entre threads ou processus,** suivant les options :

```
pthread_mutex_init(),
pthread_mutex_lock(), pthread_mutex_trylock(),
pthread_mutex_unlock(),
pthread_mutex_setATT(), pthread_mutex_getATT(), ...
```

• Les **sémaphores** sont l'implantation classique de l'outil défini par Dijkstra. La file d'attente est gérée par ordre de priorités décroissantes, les sémaphores peuvent être utilisés entre threads ou processus, suivant les options :

```
sem_init(), sem_destroy(), sem_open(), sem_close()
sem_post(), sem_wait(), sem_trywait()
```


Les variables conditionnelles POSIX.4

• Les fonctions de gestion :

```
pthread_cond_init(&VarCond,NULL),
pthread_cond_destroy(&VarCond),
pthread_cond_wait(&VarCond, &Verrou),
pthread_cond_timedwait(&VarCond, &Verrou, &Tempo),
pthread_cond_signal(&VarCond),
pthread_cond_broadcast(&VarCond).
```

• le `cond_wait` est **toujours** bloquant, à la différence d'une opération P sur un sémaphore. Il fait passer le thread à l'état bloqué ET rend le verrou associé de façon **atomique**.
Quand un thread en attente d'un `cond_signal` ou d'un `cond_broadcast` sort de l'état bloqué il **essaie de reprendre** le verrou,

• l'évènement de réveil (`signal, broadcast`) n'est pas mémorisé : si aucun thread ne l'attend, il est **perdu** (différent de V sur un sémaphore)

Les variables conditionnelles

• Dans l'exemple suivant, un thread incrémente une variable, les autres attendent qu'elle franchisse un seuil :

Thread de calcul

```
/******    variables partagees     ******/
pthread_mutex_init(&Verrou, NULL);
pthread_cond_init(&VarCond, NULL);

...
while (...){
   ...
   pthread_mutex_lock(&Verrou);
   Compteur ++;
   if (Compteur > N) pthread_cond_broadcast(&VarCond);
   pthread_mutex_unlock (&Verrou);
...
}
...
```

Threads en attente

```
...
...
pthread_mutex_lock (&Verrou);
     while (N < Compteur) {
         pthread_cond_wait(&VarCond, &Verrou);
     }
printf ("Seuil atteint! "\n);
...
pthread_mutex_unlock (&Verrou);
```

Les signaux

- Dans l'implémentation TR les différentes occurrences d'un même signal sont conservées, le nombre de signaux reçus correspond toujours au nombre de signaux émis.
- Pas de perte : gestion d'une liste de signaux en attente
- La priorité liée au signal est respectée (celle du thread) dans la gestion de la file d'attente
- Emission par `sigqueue`, par un `timer`, par un fin d'e/s
- nouveaux signaux : `RTSIG_MAX` signaux, numérotés de `SIGRTMIN` à `SIGRTMAX`

Fonction	Description
`sigqueue`	Mettre un signal dans la **file d'attente** associée au processus destinataire
`sigwaitinfo`	Attendre un signal et une info.
`sigtimedwait`	Attendre temporisée d'un signal

Les timers POSIX temps réel

- Avec l'option **CLOCK_REALTIME :**

Fonction	Description
`clock_settime`	Initialiser l'horloge
`clock_gettime`	Lire la valeur de l'horloge
`clock_getres`	Lire la résolution de l'horloge
`nanosleep`	Sleep haute résolution
`timer_create`	Création d'un timer
`timer_delete`	Destruction d'un timer
`timer_settime`	Armement/désarmement d'un timer
`timer_gettime`	Lire le délai restant sur un timer
`timer_getoverrun`	Lire le délai dépassé sur un timer

Les messages POSIX temps réel

- Ils sont similaires à ceux proposés par les IPC System V, mais à chaque message est associée une **priorité.** Le problème de l'inversion de priorité n'est pas géré :

Fonction	Description
`mq_close`	Fermer une file de messages
`mq_getattr`	Récupérer les caractéristiques d'une file de messages
`mq_open`	Ouvrir une file de message
`mq_receive`	Extraire un message d'une file
`mq_send`	Déposer un message dans une file
`mq_setattr`	Changer les attributs d'une file
`mq_unlink`	Détruire une file de messages

Exemple : producteur/consommateur

- Elements d'un programme :

```
#define TAILLE 10

mqd_t NomBAL;
struct mq_attr Attr;
timespec Ech;

 char Emis[TAILLE];
char Recu[TAILLE];

// nombre maximal de messages
Attr.mq_maxmsg = NbMess;

// taille des messages
Attr.mq_msgsize = TailleMess;

// si la boite n existe pas:
flag = O_CREAT|O_RDONLY|O_WRONLY|O_RDWR|O_NONBLOCK;
NomBAL = mq_open("/BAL", flag, mode, Attr);

// si la boite est deja cree :
flag = O_RDONLY|O_WRONLY|O_RDWR|O_NONBLOCK;
NomBAL = mq_open ("/msgQueue", flag);

// depot dans la BAL
mq_send (NomBAL, Emis, TAILLE, Prio);
// depot avec time-out
mq_timedsend (NomBAL, messageS, TAILLE, Prio, &Ech);

// attente de message
mq_receive (NomBAL, Recu, TAILLE, &Prio);

// attente de message TEMPORISEE
mq_timedreceive (NomBAL, Recu, TAILLE, &Prio, &Ech);
```

POSIX : Serveur sporadique Et PCP

• Initialisation du serveur sporadique et du verrou PCP:

```
/******        Serveur sporadique                  *********/
/******  Initialisation des priorites                ******/
/******  Initialisations de la periode et du budget  ******/
/******      a 1/2 seconde et 1/4 seconde            ******/
#define HIGH_PRIORITY   150
#define LOW_PRIORITY    100
schedparam.ss_replenish_period.tv_nsec = 500000000;
schedparam.ss_initial_budget.tv_nsec   = 250000000;
schedparam.sched_priority              = HIGH_PRIORITY;
schedparam.ss_low_priority             = LOW_PRIORITY;

/********  Creation d'un verrou avec option PCP  **********/

#define MEDIUM_PRIORITY 131
pthread_mutexattr_setprotocol( &attr, PTHREAD_PRIO_PROTECT );
pthread_mutexattr_setprioceiling( &attr, MEDIUM_PRIORITY );
pthread_mutex_init( &Mutex_id, &attr );

priority = schedparam.sched_priority;
sprintf( buffer, " -  nouvelle priorite = %d", priority );
print_current_time(  buffer );
```


POSIX : Serveur sporadique Et PCP

• Lancement du serveur, il va commencer à consommer son budget de temps, puis prendre le verrou:

```
/******** Boucle pour voir diminuer la priorite **********/
for ( ; ; ) {
   if ( schedparam.sched_priority != LOW_PRIORITY )
        continue;
   priority = schedparam.sched_priority;
   sprintf( buffer, " -  nouvelle priorite = %d", priority );
   print_current_time(  buffer );

  /********** L'appel a lock va augmenter la priorite
**********/
  puts( "Verrou va etre pris" );
  pthread_mutex_lock( &Mutex_id );
  priority = schedparam.sched_priority;
  sprintf( buffer, " -  nouvelle priorite = %d", priority );
  print_current_time(  buffer );
  break;
}
```


POSIX : Serveur sporadique Et PCP

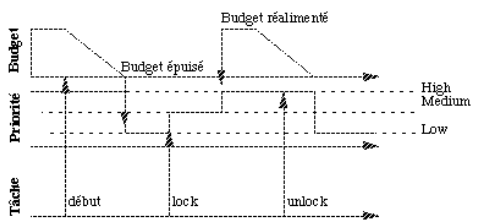
• Attendre. Le budget va être remis à son niveau initial. Puis on ve libérer le verrou, la priorité devarit baisser.

```
/***** Attendre pour voir le budget etre re-alimente
**********/
for ( ; ; ) {
 if (schedparam.sched_priority == HIGH_PRIORITY )
       break
}

priority = schedparam.sched_priority;
sprintf( buffer, " -  nouvelle priorite = %d", priority );
print_current_time(  buffer );


/***** Le unlock doit faire descendre la priorite **********/
puts( " On va rendre le verrou" );
pthread_mutex_unlock( &Mutex_id );

priority = schedparam.sched_priority;
sprintf( buffer, " -  nouvelle priorite = %d", priority );
print_current_time(  buffer );

for ( ; ; ) {
 if ( schedparam.sched_priority == LOW_PRIORITY )
      break;
}

priority = schedparam.sched_priority;
sprintf( buffer, " -  nouvelle priorite = %d", priority );
print_current_time(  buffer );
```


POSIX : Serveur sporadique Et PCP

• Résultats et schéma correspondant :

```
/******************* RESULTATS *************************/


  Fri May 24 11:05:01  -  nouvelle priorite = 150
  Fri May 24 11:05:01  -  nouvelle priorite = 100
Verrou va etre pris
  Fri May 24 11:05:01  -  nouvelle priorite = 131
  Fri May 24 11:05:01  -  nouvelle priorite = 150
On va rendre le verrou
  Fri May 24 11:05:01  -  nouvelle priorite = 150
  Fri May 24 11:05:01  -  nouvelle priorite = 100
```



Plan

- Rappels sur la gestion du temps dans les SE
- La gestion du temps dans un STR
- Facteurs intervenant dans la gestion du temps
- Propositions POSIX
- **Unix et le temps réel**

Unix et le temps réel

- Objectifs :
  - adaptation au temps réel souple (*soft real time*) : par exemple, le multimédia,
  - Cohabitation avec le temps partagé,
- Moyens
  - Ordonnancement et synchronisation : threads POSIX
  - Gestion mémoire : pouvoir verrouiller les pages en mémoire,
  - Gestion disque : pré-allouer de la surface disque,

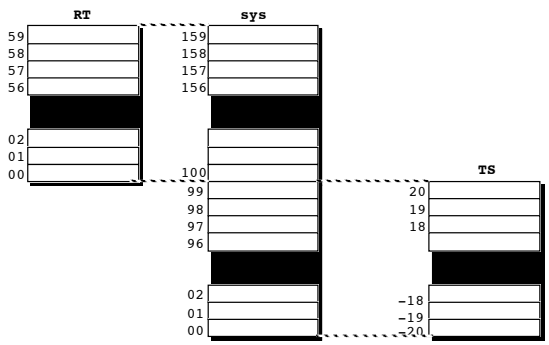
Priorités Unix SVR4

- Deux niveaux de priorité :

Niveau 1 : Priorités matérielles (*hardware dispatching*)	interruptions issues du matériel *les interruptions sont traitées immédiatement, indépendement de la priorité du processus qui a provoqué leur arrivée.*
Niveau 2 : Priorités logicielles (*software dispatching*)	Trois classes d'ordonnancement : - `RT`,la plus prioritaire. Un processus a une priorité fixe, que lui seul peut changer. Il hérite du quota cpu de son père. - `sys` pour gérer le swapper, la pagination, les STREAMS ... . Un processus a une priorité fixe définie par le noyau. - `TS`, le temps partagé traité en *round robin*.

Ordonnancement Unix SVR4

- Chaque classe gère son propre algorithme et propose des processus à la file globale du noyau :



- **Rappel : les interruptions sont plus prioritaires que n'importe quelle tâche**

Indications pour la mise en place d'un environnement TR

• Choisir l'ordonnancement et les outils de synchronisation ad hoc,

• Au niveau développement :

- Eviter l'édition de **liens dynamique** (accès disquespossibles lors de l'exécution).
- Pas d'allocation dynamique après la séquence d'initialisation de l 'application

• Contrôle fort des ressources :

- Verrouiller les pages en mémoire, pas de pagination, pas de swap, (cf `mmap, mlockall, memcntl, sysconf`)
- Les *e/s passent par le cache disque* : efficacité, mais pas de déterminisme temporel,
- Gestion des signaux : dédier un thread pour gérer l'arrivée aléatoire des interruptions,

Gestion des fichiers

• Déléguer les entrées-sorties à des threads, attention aux problèmes de synchronisation,

• Pour configurer le disque :

- `tunefs` qui définit en particulier la taille du plus grand espace contigu
- Configuration dynamique des zones de swap : `swap,`
- Pour effectivement écrire sur le disque utiliser `fsync(fd)` qui demande la mise à jour du disque et attend qu'elle soit faite,
  - Cet appel est différent de `sync` qui demande la mise à jour, mais n'attend pas la fin des écritures.
- Pour ne pas utiliser le cache : `mmap, /dev/zero`