Systèmes temps réel
POSIX temps réel

Bertrand Dupouy

Systèmes temps réel

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

Rappel sur le rôle d'un système d'exploitation

- Objectifs :
  - Gestion et partage des ressources (logicielles et matérielles),
  - Présenter une machine virtuelle (indépendance vis-à-vis du matériel),
  - Efficacité ou vivacité (progression), consistance, flexibilité, robustesse,

- Ressources à gérer :
  - processus (ordonnancement, synchronisation)
  - mémoire (allocation, protection, partage, organisation),
  - fichiers (conservation/intégrité des informations), base de données,
  - communications (réseau, intergiciels)
  - entrées-sorties, évènements,

Le temps dans un SE temps partagé

Temps

- 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 « physique »

- 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 une 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)

Le temps dans un SE temps partagé

Priorité et datation

- Le SE ne prend pas en compte de contraintes externes pour la gestion du temps, il décide :
  - des marques temporelles (dates de mises à jour,…)
  - de l'allocation du processeur,

- 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

Contraintes des STR

• Objectifs des STR :

- déterminisme logique (comme les SE) : les mêmes données en entrées donnent les mêmes résultats en sortie,
- déterminisme temporel (en plus des SE) : respect des échéances, prédictibilité : répondre à des contraintes temporelles (sur le début et/ou la fin des activités),
- fiabilité (plus que les SE)

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

• Les bases de temps peuvent être très diverses :

- radar : millisecondes,
- BdD : minutes,
- météo : heures,
- sonde spatiale : jours.

Exemples de contraintes

• Echéance (*deadline*) à respecter : évènement dont la date est fixée « au plus tard » ou « au plus tôt »,

• Cohérence temporelle à assurer pour la production de résultats, exemple : synchronisation son/image en multimédia,

• Cadence à préserver pour la présentation de résultats : régularité pour la sortie d'images vidéo

• Domaines d'application :

- transport (avionique, automobile, trains), les applications sont souvent souvent embarquées (enfouies, *embedded*)
- contrôle de production (nucléaire, …)
- télécommunications
- multimédia

Les STR : définitions

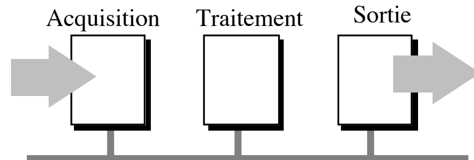
• *Hard and soft real time*

hard real time	strict respect des échéances
soft real time	on tolère le non respect occasionnel de certaines échéances.

• Un système TR se caractérise par sa capacité à respecter des échéances : les réponses doivent arriver en un temps fini et PREDICTIBLE.

• Un système TR est en **interaction** avec l'extérieur.Les échanges se font de manière :

– synchrone (scrutation, *clock based system*),
– asynchrone (interruption, *sensor based system*).



• les tâches temps réel sont donc périodiques (activation à fréquence fixe), apériodiques (fréq. activ. variable) ou sporadiques (fréq. activ. variable mais bornée).

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 (séquencement statique),
  - – tâches apériodiques (séquencement dynamique),
  - – Si on connaît l'application et son environnement : méthodes statiques, sinon, méthodes dynamiques
- Notion de qualité de la réponse :
  - – moins précise, plus rapidement,
  - – exacte, moins rapidement,
- Combien d'échéances peut-on manquer ?

  on peut associer des VALEURS aux terminaisons d'activités, le comportement du STR est déterminé par un algorithme qui maximalise ces valeurs (*time values*),

Systèmes temps réel

Sur l'ordonnancement dans les STR

- les tâches à exécuter gèrent des contraintes extérieures,
- 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,
- il faut assurer le respect des échéances pour les tâches **critiques** : une tâche de faible importance peut avoir de fortes contraintes de temps, et une tâche de forte importance peut avoir de faibles contraintes de temps, (passer de RMS à EDF),
- les STR ont souvent une connaissance a priori des tâches à exécuter,
- Les ordonnanceurs sont construits en fonction des informations dont ils disposent :

importance relative des tâches	Priorités + préemption
échéance + durée	LLF (marge = échéance - t. présent - t cpu demandé)
time value	
relation de dépendance entre activités	

- Pb du pire cas (*WCET, worst case execution time*): son temps d'exécution est bien supérieur à la moyenne

Récapitulatif

- Le comportement doit être prédictible dans deux domaines :
  - logique (comme dans les SE : sûreté, vivacité, ...),
  - temporel (à la différence des SE !).
- Dans une application temps réel, le début et la fin des activités doivent respecter des contraintes de temps.
- Le comportement du STR est déterminé par des algorithmes qui gèrent le respect des échéances
- Une valeur peut être associée à la terminaison d'une activité pour en évaluer l'importance (criticité vs priorité)

- Types d'activités gérées par les STR :
  - périodiques, on connaît à l'AVANCE les contraintes de temps (les échéances),
  - apériodiques, il faut prendre en compte dynamiquement l'arrivée de traitements soumis à des contraintes de temps,

Rapidité et temps réel (1)

- *real time computing is not fast computing :*

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

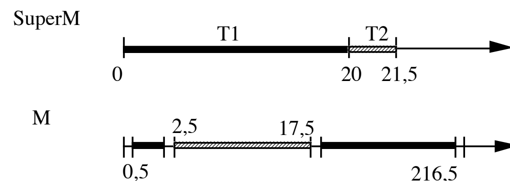
- *SuperM*, durée changement de contexte = 0 (!), processeur 10 fois plus rapide que celui de M,ordonnancement : FCFS

• 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),

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,

Il ne faut pas de STR,
pourquoi :
(de mauvais arguments)

• Exemples de mauvais arguments :

- on n'a pas à reconfigurer un système déjà existant
- pas besoin de licence,
- on gagne de la place,
- on gère directement les interruptions, donc on gagne du temps,
- on optimise le code,

Pourquoi faut-il un STR (1)?

- Exécution des traitements dans les routines d'interruptions : trop lent, on perd des événements :

```
void Traite_IT_N (…){
/* Gerer tout le traitement lié à l'IT */
}
```

- Donc : mise à jour de variables globales dans les routines d'interruptions et scrutation dans `main`. Ceci revient à faire FCFS dans une boucle d'attente active sans doute manquer des échéances -> **il faut** passer à la notion de tâche (bloquée ou non, …)

void Traite_IT_N (…){
…
Var_IT_N = 1 ;
…
}
	main (…) {
	while (1) {
	…
	if (Var_IT_N == 1) {…..}
	…
	}
	…
	}

Pourquoi faut-il un STR (2) ?

- Il faut gérer des tâches et non pas des fonctions (état bloqué vs attente active) avec un algorithme d'ordonnancement approprié,

- Il faut disposer de primitives de synchronisation (atomicité, …),

- On a besoin d'outils de communication (E/S) et d'allocation de mémoire,

- Le système masque les particularités du matériel, augmente la portabilité, offre des algorithmes fiables

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)

Services attendus D'un STR

• **Les services** qu'on va attendre d'un système temps réel :

- politiques d'ordonnancement respectant les échéances, on les classe en plusieurs familles:
  - algorithmes à priorité fixe (RMS),
  - algorithmes à priorité dynamique (EDF, LLF, MUF),
- 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),

Limites des services Rendus par un STR :

• **Limites de ces services** :

- ils peuvent être plus ou moins élaborés, par exemple l'ordonnancement peut prendre en compte:
    - la différence entre criticité et priorité (une valeur peut être associée à la terminaison d'une activité pour en évaluer l'importance),
    - le traitement des surcharges, …
- ils ne **suppriment pas tous** les facteurs d'indéterminisme, il faudra prendre aussi en compte :
    - la gestion 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,
    - le fonctionnement du processeur (pipe line),
    - le traitement matériel des entrées-sorties et des interruptions,

Systèmes temps réel

Quel langage pour les applications temps réel ?

• Langage dédié ou généraliste ?

• Services attendus par l'utilisateur :

- expression 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,

• Java temps réel (RTSJ), Ada,

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 :
  - 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*) ?

Plan

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

Gestion mémoire

- Gestion statique :
  - le nombre, la taille et l'emplacement des objets sont connus, ou bornés, lorsque le système est chargé,
  - donc : pas de fonction d'allocation de mémoire,
  - avantage : accès aux objets en temps constant et faible (objets implantés sous forme de tableaux),
  - inconvénient : pas flexible
- Gestion dynamique :
  - allocation de la mémoire à la demande,
  - avantage : souplesse,
  - inconvénients :
    - temps d'accès et d'allocation difficile à prédire ou à borner,
    - fragmentation,
    - respect de la localité ?
- Remarques :
  - Influence sur le temps de changement de contexte,
  - Attention à l'édition de liens dynamique,

Gestion dynamique : Allocation contigüe

• En allocation contigüe :

- comment gérer la liste des blocs libres et occupés ?
- le temps d'allocation (c'est-à-dire 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* (rangement sur disque) 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 un accès mémoire,

Protection mémoire

• Si chaque tâche a ses propres tables de pages ou de segments :

- pas de partage des espaces d'adressage entre tâches: il faut des outils système pour l'assurer, mais plus de sécurité,
- changement de contexte plus long,
- latence due à la traduction d'adresse non prédictible (plusieurs niveaux de tables, *hit* ou non dans le *TLB*)

Mémoires caches et temps réel

• Avantage :

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

• 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,

Plan

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

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,
    - utilisation de caches par les périphériques,
    - priorité des entrées-sorties non liée à celle des processus
    - utilisation de caches par le *file system* ,

Entrées sorties et temps réel

- Gestion du temps dans les entrées-sorties, il faudrait un délai d'attente maximum (*timeout*) :

```
-lire (périph. , données, délai)
```

- On devrait pouvoir gérer les E/S comme on gère les processus :
  - satisfaire la plus prioritaire d'abord,
  - la priorité des E/S doit correspondre à celle des processus qui les ont lancées,
  - annuler une demande d' E/S lorsque son échéance est dépassée,
  - réordonner la liste des demandes si une demande plus prioritaire arrive,

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, les caractéristiques techniques des architectures doivent être prises en compte :
  - les latences dues aux interruptions, à la préemption sont un facteur important, elles reflètent une technologie d'implantation plutôt qu'un style de fonctionnement,
    - les interruptions « polluent » les caches
- on peut prendre en compte les interruptions dans une étude d'ordonnançabilité, par exemple, en ajoutant une tâche périodique pour l'horloge,

Temps de réponse à un événement

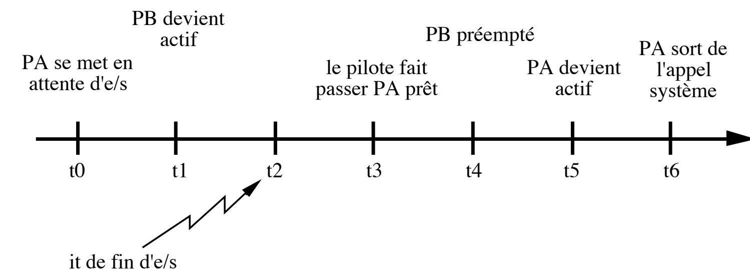
• (Cet événement va provoquer un changement de contexte)

<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 élu.
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 de niveau inférieur.
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 supposé en temps nul…
t2 à t6 supposé en temps nul…

t0 à t1	P_A passe dans l'état bloqué et changement de contexte en faveur de P_B
T1 à t2	P_B est actif
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 passer P_B en état prêt
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 clés 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 récéption 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,

Systèmes temps réel

Récapitulatif : Ce que doit proposer le matériel

- Rappel : ce que propose généralement le système :
  - efficacité gestion mémoire -> threads,
  - ordonnancement approprié -> RMS , EDF, serveur sporadique,…
  - E/S à échéance -> alarmes, timers,
  - Communications -> bus temps réel, réseau TR, intergiciels TR.
- Ce que doit proposer le matériel:
  - gestion efficace des interruptions,
  - gestion fine des horloges,
  - gestion des caches mémoire (verrouiller, geler, …),
  - gestion efficace des E/S (caches, …).

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 des `ifdef`, ou `sysconf` à l'exécution:

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


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,`

• seul l'utilisateur privilégié (*root*) peut accéder à ce service d'ordonnancement,

• applicables, pour la plupart, aux threads et processus ; dans le cas des processus, le nom de la fonction ne contient pas le mnémonique `pthread,`

Threads POSIX.4

• Exemple :

```
int main() {

pthread_t            tid;
pthread_attr_t       attr;
struct sched_param   param;
...

pthread_attr_init (&attr)

/* choix de la politique d'ordonnancement */
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

/* choix de la  priorité */
param.sched_priority = 1;
pthread_attr_setschedparam (&attr, &param);

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

...
return 1 ;
}
```


Les `mutex` et sémaphores POSIX.4

• 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. Citons :

```
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, ils peuvent être utilisés entre threads ou processus, suivant les options. Citons :

```
sem_init(…), sem_destroy(…), sem_open(…),
sem_post(…),sem_wait(…), sem_trywait(…).
```

Les variables conditionnelles POSIX.4

- Elles servent à mettre un thread en attente de la vérification d'une condition,
- Implantation : association d'un mutex et d'une variable dite conditionnelle
- 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 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 de façon **atomique**.
  Quand le thread sort de l'état bloqué sur un `signal` ou `broadcast`, 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 POSIX.4

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

Thread de calcul

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

Threads en attente

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

Les signaux POSIX

- Dans l'implémentation TR, les différentes occurrences d'un même signal restent pendantes. Le nombre de signaux reçus correspond toujours au nombre de signaux émis.
- Pas de perte : gestion d'une liste de signaux en attente
- Priorité liée au signal qui est respectée, avec celle du thread, dans la gestion de la file d'attente
- Emission par `kill`, `sigqueue`, `timer`, ou sur fin d'e/s
- Nouveaux signaux : `RTSIG_MAX` signaux, numérotés de `SIGRTMIN` à `SIGRTMAX`
- Une valeur peut être passée à `sigwaitinfo` et `sigtimedwait`.
- API :

Fonction	Description
`sigqueue`	Envoyer un signal
`sigwaitinfo`	Attendre un signal et une info.
`sigtimedwait`	Attendre un signal avec une **échéance**

Les timers et les messages

- Extrait de l'API timers:

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_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 sont similaires à ceux des 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`	Renvoie les caractéristiques d'une file de messages
`mq_open`	Ouvrir une file de message

Threads POSIX et threads UNIX

• Verrous
```
  pthread_mutex_init                mutex_init()
  pthread_mutexattr_init
  pthread_mutexattr_setpshared
  pthread_mutexattr_getpshared
  pthread_mutexattr_setprotocol
  ...

 pthread_mutexattr_setprioceiling
  pthread_mutex_lock                mutex_lock()
 ...                               ...
  pthread_mutex_unlock              mutex_unlock()
  pthread_mutex_destroy             mutex_destroy()
```

• Variables conditionnelles
```
     pthread_cond_init                cond_init()
     ...                              ...
     pthread_condattr_setpshared      ___
     pthread_condattr_getpshared      ___
     pthread_condattr_destroy         ___
     pthread_cond_wait                cond_wait()
     pthread_cond_timedwait           cond_timedwait()
```

• Lecteurs-écrivains
```
                                    rwlock_init()
                                    rwlock_destroy()
```

• Sémaphores
```
             sem_init         sema_init()
             sem_open,        ___
             sem_close        ___
            ...               ...
             sem_getvalue     ___
             sem_unlink       ___
```


Systèmes temps réel

POSIX : Serveur sporadique Et PCP

• Initialisation du serveur sporadique et du verrou PCP:

```
char buffer[2000] ;

/******        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 );

/********        Trace en mémoire !!             **********/
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, 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

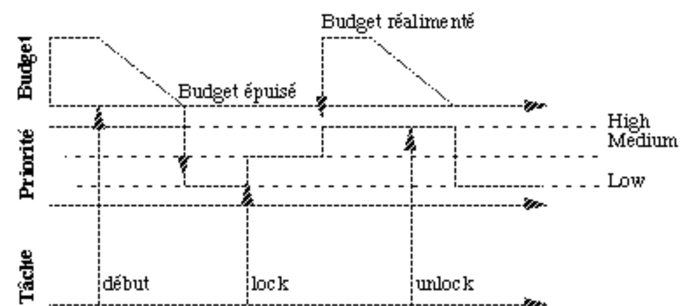
• Le budget va être remis à son niveau initial. Puis on va libérer le verrou, la priorité devrait baisser.

```
/**** 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

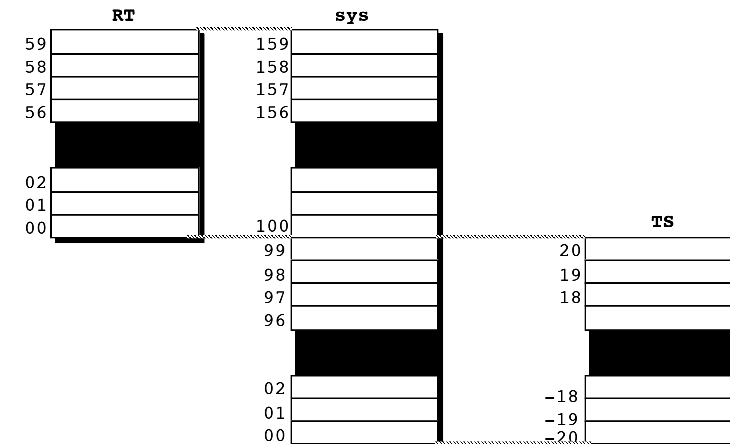
• **Unix et le temps réel**

Comment construire un STR ?

- En le dérivant d'un système temps partagé (UNIX ou autre), pour cela, il faut :
  - pouvoir verrouiller les pages en mémoire,
  - pré-allouer de la surface disque,
  - maitriser le temps de réponse aux interruptions,
  - réduire le temps de changement de contexte,

Unix : Ordonnancement SVR4

- Objectifs :
  - adaptation au temps réel mou : par exemple, le multimédia,
  - cohabitation avec le temps partagé,
- Chaque classe RT (Real Time) et TS (Time Sharing) gère son propre algorithme et propose des processus à la file globale du noyau :
  - `RT,` la plus prioritaire. Chaque processus a une priorité fixe.
  - `TS,` le temps partagé traité en *round robin*.



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

Mise en place d'une application TR : points à vérifier (1)

• Veiller à la pré-allocation des ressources :

- Pour la mémoire : pas d'allocation dynamique (cf. malloc (C), new(Java)),
- Pour les threads : pas d'allocation dynamique (cf. thread_create(C), new(Java)),

• Attention à l'édition de **liens dynamique** (accès disque implicite lors de l'exécution) :

- Utiliser l'édition de lien statique (C)
- Le pré-chargement des classes (Java),

• Utiliser l'ordonnancement et les outils de concurrence *ad hoc* :

- Attention au blocage éventuel sur une ressource détenue par un processus moins prioritaire ou temps partagé ! (**inversion** de priorité)

Mise en place d'une application TR : points à vérifier (2)

• Contrôler la gestion de la mémoire:

- verrouiller les pages en mémoire,
- vérifier le fonctionnement des caches (efficacité, mais pas de déterminisme temporel),

• Vérifier le fonctionnement des périphériques :

- par exemple, dans le cas des disques : surface disque disponible ; utilisation, ou non, de caches.

• Gestion des e/s :

- utiliser des threads dédiés ou des fonctions non bloquantes,
- gérer les échéances : fonctions du type `select` ou `poll`,
  - *Mais, dans tous les cas : l'ordre d'exécution sur un périphérique n'est pas forcément celui de la mise en attente.*

• Gestion des signaux : dédier un thread pour gérer leurs arrivées aléatoires