

Real-time systems

Problématiques de temps dans les systèmes Unix

Laure Gonnord

University of Lyon/ LIP

MIF18 - M1 optional course. Feb-June 2019

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Problématique

Objet : parler de l'**implémentation des systèmes temps-réel**.

Rappels :

- Souvent embarqués : souvent *all-in-one* : processeur, mémoire, E/S.
 - Problématiques : la sûreté, et les contraintes de temps.
 - Des applications différentes suivant si TR dur ou souple.
- Différentes implémentations/approches, mais “pensées système”.

Première génération

Idée : le moins de fonctionnalités possibles.

► Bibliothèque minimale (1 à 10Ko) de fonctions permettant de gérer les tâches, les signaux, la communication et l'exclusion mutuelle.

Exemples :

- SCEPTRE (20 fonctions)
- XEC de Real Time Craft (27 fonctions)

Inconvénients :

- Toujours plus de fonctionnalités à ajouter
- Sûreté pas facile à prouver (plateforme de test = celle de dev.)

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Problématique

Essayons de rendre Unix temps-réel.
(à priori pour des systèmes temps-réel souples)

Unix classiques, problèmes déjà vus

(dernier cours)

- Ordonnanceur inadapté : Changer la méthode d'ordonnancement.
 - Faible résolution d'horloge : Ajouter de nouveaux appels systèmes.
 - Gestion des signaux lents et aléatoire : Améliorer la préemptivité du noyau.
 - Architecture monolithique (fiabilité et taille) : Redéfinition de l'architecture de manière modulaire.
- ▶ Cf la suite.

Ordonnanceur

Principe :

- On ajoute une classe de processus temps réel.
 - Ceux-ci interrompent les autres tâches (sans leur laisser le temps de finir leur quantum) quand ils sont prêts.
- ▶ raccourcir le temps de démarrage des processus temps réel.

Préemptivité

Rappel : en Unix classique, un appel système n'est pas préemptible.

Solutions :

- insérer le “bon nombre” de points de préemptions.
- OU rendre le noyau complètement préemptif (en protégeant toutes les structures de données par des sémaphores).

Remarque Ce problème est le même en multi-processeurs quand on veut que plusieurs processeurs exécutent le code du noyau.

Fichiers

Les fichiers “temps réel” sont préalloués de manière contiguë sur le disque.

Vérouillage mémoire : *sticky bit* ✓

La notion de réentrance - 1/4

Fonction **réentrante** : fonction pouvant être utilisée par plusieurs tâches en concurrence, sans risque de corruption de données.

Exemple

```
int f(void)
{
    static int x = 0;
    x = 2* x +7;
    return x ;
}
```

Problème :

- l'instruction $x = 2*x+7$ se compile en plusieurs instructions machine.
- un thread peut être interrompu au milieu de ces instructions.

La notion de réentrance - 2/4

Une fonction non réentrante ne peut être partagée par plusieurs processus ou threads que si on assure l'exclusion mutuelle : au plus un thread pourra exécuter son code à un moment donné.

Une fonction réentrante peut être interrompue à tout instant et reprise plus tard sans corruption de données.

Pour écrire des fonctions réentrantes :

- Pas de données statiques utilisées d'un appel à l'autre.
- Pas de retour de pointeur statique : données fournies par l'appelant.
- Travail sur variables locales,
- Pas d'appel de fonctions non réentrantes

La notion de réentrance - 3/4

Fonctions connues non réentrantes :

- qq appels systèmes (*via* des bibliothèques de base). cf man.
 - implique de réécrire une nouvelle fonction
 - POSIX.1 donne une liste de fonctions réentrantes.
Typiquement `malloc()`, `free()`, les fonctions E/S de la `libc` ne sont pas garanties réentrantes.
- N'utiliser qu'en connaissance de cause des fonctions non réentrantes dans les programmes utilisant des signaux, ou multi-thread.

La notion de réentrance - 4/4

Ne pas oublier !

Ajouter le flag `-D_REENTRANT` à la compilation quand on utilise des *threads*.

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Approche hôte-cible

Principe : un système temps réel tourne sur sa propre machine, communique avec un système UNIX qui tourne sur une autre machine.

Communication : par réseau si les machines sont différentes ou bien par bus si c'est une machine avec une carte dédiée temps réel.

- VxWorks (Wind River Systems, ils vendent aussi un Linux de niveau certifié industriel)
- VMEexec (Motorola)

Approche points de préemption

Principe : Ajout de points de préemption dans un noyau existant Unix :

- Unix System V
- Linux Low Latency

Ou l'on réécrit un noyau Unix :

- AIX (IBM)
- REAL/IX (MODCOMP)
- LYNX OS (Lynx Real Time System) Compatible binaire Linux
- Solaris (Sun)

Approche micro-noyau 1/2

Principe : Le noyau lui-même est temps réel, les modules que l'on ajoute ne le sont pas. Les gestionnaires de périphériques et les systèmes de fichiers ne font pas partie du noyau.

- Mach (CMU)
- Chorus (Chorus System), est devenu VirtualLogix qui fait tourner des sous-os temps réel ou non et aussi Linux.
- QNX (compatible Unix)
- eCos (libre, très configurable)
- OS9
- OpenRTOS
- ITRON (Japon)
- Symbian OS (téléphone portable, PDA)
- Windows CE

Approche micro-noyau 2/2

Caractéristiques :

- Ces micro-noyaux sont plus lents (5-15%) (nombreux messages échangés entre processus).
- Plus fiables (modularité).
- Résistance aux pannes : un *driver* qui plante ne fait pas planter le système et peut être relancé.

Nano/Pico/Exo kernel : Les processus ont un accès direct au matériel mais fait de manière sécurisé. Les processus peuvent s'échanger des données sans passer par le kernel.

Approche machine virtuelle

Le système de base est un micro-noyau temps réel faisant tourner des tâches qui sont des systèmes d'exploitation (ou des applications).

- VirtualLogix
- PikeOS

Approche sous-tâche

On considère que le système d'exploitation généraliste est la tâche de fond du système temps réel. Les deux suivants fonctionnent avec Linux.

- RTLinux (on lance un programme en chargeant un module, l'ordonnanceur est aussi un module). Système pas très libre...
- RTAI : Aegis est un système minimal avec 2 tâches, l'une est RTAI pour le temps réel (ou Xenomai) et l'autre est Linux. (utilisé par ElinOS).

Xenomai permet de simuler (skin) différents système temps réel existants.

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Problématique

Objectif : Faire avec ce que l'on a, ie un Unix.

Pour quelles applis ? Applications Temps réel souples, qui vont cohabiter avec d'autres applis temps-partagés.

► La norme POSIX propose des *services* pour ça

Credits

Sources :

- Transparents de F. Singhoff (univ Brest).
- Transparents d'I. Puaut (univ Rennes).

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

La norme POSIX

Objectif Définir une **interface standard des services offerts** par Unix (portabilité). Publié conjointement ISO/ANSI. Remarques :

- portabilité difficile (diversité des systèmes)
- tout ne peut être normalisé
- quelques divergences dans l'implémentation (ex threads Unix).

Norme POSIX, organisation

La norme est découpée en chapitres, sous-chapitres. L'implémentation de certains (sous-) chapitres sont obligatoires, mais pas les autres.

Exemples de chapitres :

- 1003.1 : services de base (ex :fork, exec, ...)
- 1003.2 : commandes shell (ex : sh)
- 1003.5 : POSIX en ADA

POSIX, ce qui nous intéresse dans la suite

Chapitres :

- Extensions temps-réel 1003.1b. Presque tous les composants sont optionnels.
- Extensions pour le multithread 1003.1c

Résumé rapide 1/2

Threads POSIX : `pthread_`, `pthread_mutex`, `pthread_cond`,
`pthread_attr`, `pthread_rwlock...`

Résumé rapide 2/2

Les extensions temps-réel POSIX et leurs préfixes :

- ordonnancement sched_
- synchronisation (sémaphores, signaux) sem_ intr_
- messages
- gestion mémoire shm_ mémoire partagée
- gestion du temps (horloges clock_, timers timer_)
- entrées/sorties asynchrones aio_

Is is implemented ?

Deux possibilités :

- à la compilation, macros définies dans `unistd.h` : `if _POSIX_VERSION, ifdef POSIX_PRIORITY_SCHEDULING.`
- à l'exécution (`unistd.h`) : `long sysconf(int name);`

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

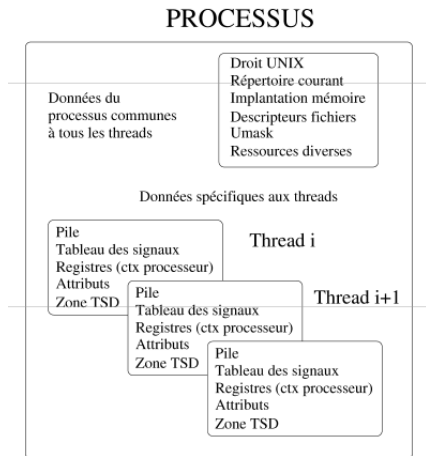
Ce chapitre contient les threads et les outils de synchro associés (mutex).

Caractéristiques :

- Un thread POSIX est défini par un id local au processus.
- Il possède une pile, un contexte, des attributs . . .
- Il peut être implanté en forme utilisateur ou noyau (la norme ne définit que l'interface).

Rappel : code réentrant.

Caractéristiques des threads POSIX, avec un schéma moche



TSD = thread-specific data. ctx = contexte.

Listing des fonctions principales

<i>pthread_create</i>	Création d'un thread. Paramètres : code, attributs, arg.
<i>pthread_exit</i>	Terminaison d'un thread. Paramètre : code retour.
<i>pthread_self</i>	Renvoie l'identifiant d'un thread
<i>pthread_cancel</i>	Destruction d'un thread. Paramètre : identifiant du thread.
<i>pthread_join</i>	Suspend un thread jqa la terminaison d'un autre.
<i>pthread_detach</i>	Suppression du lien de parenté entre thread.
<i>pthread_kill</i>	Emet un signal vers un thread.
<i>pthread_sigmask</i>	Modifie le masque de signal d'un thread.

Attention, changement de sémantique

Deux nouveaux comportements pour `fork()` et `exit()`

- `fork` : nouveau processus contenant un thread dont le code est la fonction `main()`
- `exit` : termine un processus et donc tous les threads contenus dans celui-ci.

Exemple : create et join

Ne pas oublier de compiler avec `-D_REENTRANT -lpthread`.

```
#include <pthread.h>

void* th(void* arg)
{
    printf("Je suis le thread %d ; processus %d\n",
           pthread_self(),getpid());

    pthread_exit(NULL);
}

int main(int argc, char* argv[])
{
    pthread_t id1 ,id2;

    pthread_create(&id1,NULL,th,NULL);
    pthread_create(&id2,NULL,th,NULL);

    pthread_join(id1,NULL);
    pthread_join(id2,NULL);

    printf("Fin du thread principal %d ; processus %d\n",
           pthread_self(),getpid());
    pthread_exit(NULL);
}
```

Attributs d'un thread

Attributs = caractéristiques données à la création, pas d'héritage.

Exemples :

Nom d'attribut	Signification
<code>detachstate</code>	<i>pthread_join</i> possible ou non
<code>policy</code>	Politique d'ordonnancement
<code>priority</code>	Priorité
<code>stacksize</code>	Taille de la pile spécifiée par le programmeur ou non

(voir le man, plein de possibilités)

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Services POSIX pour l'ordonnancement

- ils doivent s'appliquer au niveau thread et/ou niveau processus ;
- doivent implémenter ≥ 32 niveaux de priorités (fixes) ;
- une file d'attente par niveau de priorité ;
- des politiques différentes de gestion de file : SCHED_FIFO, SCHED_RR, SCHED_OTHERS


Un peu de doc ? man 7 sched :

SCHED_FIFO: First in-first out scheduling

SCHED_FIFO can be used only with static priorities higher than 0 [...]

SCHED_RR: Round-robin scheduling

Niveaux de priorité

`sched_get_priority_min()` → 


Niveau de priorité le plus faible

... → 

n → 

n+1 → 

... → 

`sched_get_priority_max()` → 

Niveau de priorité le plus fort

Listing des fonctions principales

<i>sched_get_priority_max</i>	Consulte la valeur de la priorité maximale.
<i>sched_get_priority_min</i>	Consulte la valeur de la priorité minimale.
<i>sched_rr_get_interval</i>	Consulte la valeur du quantum.
<i>sched_yield</i>	Libère le processeur.
<i>sched_setscheduler</i>	Positionne la politique d'ordonnancement.
<i>sched_getscheduler</i>	Consulte la politique d'ordonnancement.
<i>sched_setparam</i>	Positionne la priorité.
<i>sched_getparam</i>	Consulte la priorité.
<i>pthread_setschedparam</i>	Positionne la priorité.
<i>pthread_getschedparam</i>	Consulte la priorité.

Voir le man pour les détails.

Exemple : changement de paramètres de 2 processus

```
struct sched_param parm;
int res=-1;
...
/* Tache T1 ; P1=10 */
parm.sched_priority=15;
res=sched_setscheduler(pid_T1,SCHED_FIFO,&parm)
if(res<0)
    perror("sched_setscheduler tache T1");

/* Tache T2 ; P2=30 */
parm.sched_priority=10;
res=sched_setscheduler(pid_T2,SCHED_FIFO,&parm)
if(res<0)
    perror("sched_setscheduler tache T2");
```

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Vue d'ensemble 1/2

Caractéristiques :

- Signal = événement délivré de manière **asynchrone** à un processus/thread ► interruption logicielle.
- Un signal peut être bloqué (ou masqué), pendant (du verbe pendre!) ou délivré (méchant anglicisme).
- Un signal a un comportement standard ► peut être modifié par l'utilisateur.
- Une table de signaux par thread/processus

Vue d'ensemble 2/2

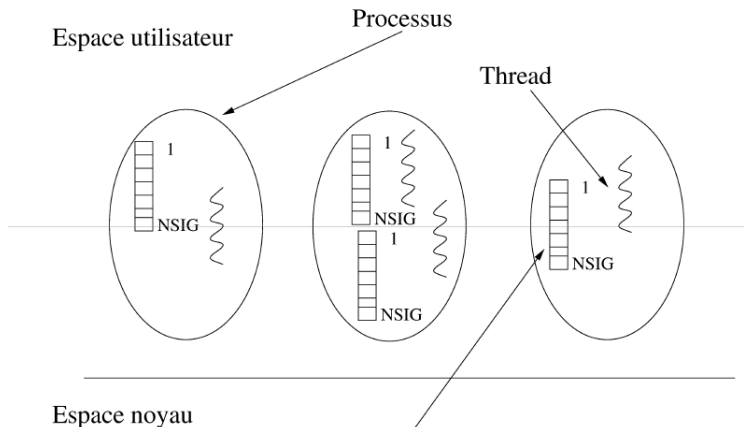


Table des signaux = une entrée par signal

- Contenant :
- pointeur sur handler
 - bit de masque
 - bit signal pendant

Signaux en POSIX 1003.1 (non TR!), exemple

```
void handler(int sig)
{
    printf("Signal %d recu\n",sig);
}

int main(int argc, char * argv[])
{
    struct sigaction sig;

    sig.sa_flags=SA_RESTART;
    sig.sa_handler=handler;
    sigemptyset(&sig.sa_mask);

    sigaction(SIGUSR1,&sig,NULL);

    while(1); // le programme bosse !!
}
```

Exécution :

```
$> sig &
```

```
$> kill -USR1 14090
```

```
Signal 10 recu
```



Signaux en POSIX 1003.1

Pb des implémentations de POSIX 1003.1 :

- livraison non fiable des signaux pendants (oups, pas temps-réel !)
- ordre d'émission \neq ordre de réception
- ne véhicule pas d'information
- grande latence.

Ajouts dans 10031b

Un certain nombre d'ajouts pour le temps-réel (man, man!)

- nouveaux signaux numérotés de SIGRTMIN à SIGRTMAX.
- signaux valués!
- plus de perte des signaux : une file pour les signaux pendants.
- livraison ordonnée (respect de la politique ordo + ordre venant du numéro du signal).
- émissions par kill, sigqueue, timer, ...

<i>sigqueue</i>	Emission d'un signal temps réel.
<i>sigwaitinfo</i>	Attente d'un signal sans lancement du handler
<i>sigtimedwait</i>	Idem ci-dessus + timeout sur le temps de blocage.

Signaux temps-réel + handler, exemple 1/2

```
int main(int argc, char * argv[])
{
    struct sigaction sig;
    union sigval val;
    int cpt;

    sig.sa_flags=SA_SIGINFO;
    sig.sa_sigaction=handler;
    sigemptyset(&sig.sa_mask);
    if(sigaction(SIGRTMIN,&sig,NULL)<0)
        perror("sigaction");

    for(cpt=0;cpt<5;cpt++)
    {
        struct timespec delai;
        delai.tv_sec=1;
        delai.tv_nsec=0;

        val.sival_int=cpt;
        sigqueue(0,SIGRTMIN,val);

        nanosleep(&delai,NULL);
    }
}
```

Signaux temps-réel + handler, exemple 2/2

```
void handler (int sig, siginfo_t *sip, void *uap)
{
    printf("Reception signal %d, val = %d \n",
          sig, sip->si_value.sival_int);
}
```

- Exécution :

```
$rt-sig
Reception signal 38, val = 0
Reception signal 38, val = 1
Reception signal 38, val = 2
Reception signal 38, val = 3
Reception signal 38, val = 4
```

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

Services liés au temps

- Quelle heure est-il ?
- Bloquer un thread/un processus pendant une durée donnée.
- Réveiller un thread/un processus régulièrement (timers)

La précision est **intrinsèquement liée** au matériel.

Extensions POSIX 1001b

Entre autres :

- Support de plusieurs horloges.
- Impose la présence d'au moins une horloge : `CLOCK_REALTIME` (précision d'au moins 20 ms).
- Structure `timespec`.
- Services : consultation et modifs des horloges, mise en sommeil d'une tâche, **timer périodique couplé avec les signaux UNIX**.

Fonctions

<i>clock_gettime</i>	Consulte la valeur d'une horloge.
<i>clock_settime</i>	Modifie la valeur d'une horloge.
<i>clock_getres</i>	Obtention de la précision d'une horloge.
<i>timer_create</i>	Crée un timer.
<i>timer_delete</i>	Détruit un timer.
<i>timer_getoverrun</i>	Donne le nombre de signaux non traités.
<i>timer_settime</i>	Active un timer.
<i>timer_gettime</i>	Consulte le temps restant avant terminaison du timer.
<i>nanosleep</i>	Bloque un processus/thread pendant une durée donnée.

Timer exemple 1/2

```
int main(int argc, char * argv[])
{
    timer_t monTimer;
    struct sigaction sig;
    struct itimerspec ti;

    timer_create(CLOCK_REALTIME, NULL, &monTimer);

    sig.sa_flags=SA_RESTART;
    sig.sa_handler=trop_tard;
    sigemptyset(&sig.sa_mask);
    sigaction(SIGALRM, &sig, NULL);

    ti.it_value.tv_sec=capacite;
    ti.it_value.tv_nsec=0;
    ti.it_interval.tv_sec=0; // ici le timer n'est pas
    ti.it_interval.tv_nsec=0; // automatiquement réarmé
    timer_settime(monTimer, 0, &ti, NULL);

    printf("Debut capacite\n");
    while(go>0)
        printf("Je Bosse ..... \n");

    printf("Debloquee par timer : echeance ratee ..\n");
}
```


Timer exemple 2/2

```
int go=1;
```

```
void trop_tard(int sig)
{
    printf("Signal %d recu\n",sig);
    go=0;
}
```

- Exécution :

```
$timer
Debut capacite
```

```
Je Bosse .....
Je Bosse .....
Je Bosse .....
Je Bosse .....
Je Bosse .....
Signal 14 recu
Debloque par timer : echeance ratee ..
```

Timer + SIGRTMIN exemple 1/2

```
timer_t monTimer;
struct sigevent event;
struct sigaction sig;
struct itimerspec ti;

event.sigev_notify=SIGEV_SIGNAL;
event.sigev_value.sival_int=capacite;
event.sigev_signo=SIGRTMIN;
timer_create(CLOCK_REALTIME,&event,&monTimer);

sig.sa_flags=SA_SIGINFO;
sig.sa_sigaction=trop_tard;
sigemptyset(&sig.sa_mask);
sigaction(SIGRTMIN,&sig,NULL);

ti.it_value.tv_sec=capacite;
ti.it_value.tv_nsec=0;
ti.it_interval.tv_sec=0; // timer non réarmé
ti.it_interval.tv_nsec=0;
timer_settime(monTimer,0,&ti,NULL);

printf("Debut capacite\n");
while(go>0)
    printf("Je Bosse ....\n");

printf("Debloquee par timer : echeance ratee ..\n");
```



Timer + SIGRTMIN exemple 2/2

```
int go=1;

void trop_tard(int sig, siginfo_t *info, void *uap)
{
    printf("Reception signal %d, capacite = %d depassee \n",
          sig, info->si_value.sival_int);
    go=0;
}
```

- Exécution :

```
$timer-rt
Debut capacite
Je Bosse .....
Je Bosse .....
Je Bosse .....
Je Bosse .....
Je Bosse .....
Reception signal 38, capacite = 100 depassee
Debloque par timer : echeance ratee ..
```

Plan

Introduction

Unix temps-réel

Unix temps-réel, quelques approches

Posix temps-réel

Généralités sur POSIX

Les threads POSIX

Services d'ordonnancement

Signaux

Manipulation du temps

Conclusion

On a vu

Abstractions pour le temps réel, plus ou moins précise, un continuum complet de solutions, de UNIX + POSIX aux systèmes complets. temps-réel.

▶ à choisir selon le degré de criticité des contraintes TR.

Non traité aujourd'hui

On n'a pas montré d'exemples pour les sémaphores, les variables conditionnelles pour les threads, les entrées/sorties, les files de messages.

► Biblio perso, et lire le MAN.