Linux et le temps

Christophe Pallier *

1er octobre 2001

Connaître l'heure, mesurer le temps écoulé, effectuer des opérations à intervalles réguliers, etc... sont autant de tâches concernant la gestion du temps sous Linux qu'il est intéressant de savoir réaliser, que ce soit en ligne de commande, ou dans un programme en C.

1 À la bonne heure

À l'instar de la plupart des appareils électroménagers, les ordinateurs modernes possèdent une "montre" interne. Celle-ci conserve la date et l'heure, même quand la machine est mise hors-tension. Sur un PC, cette horloge matérielle est appelée, indifféremment, horloge "BIOS", horloge "CMOS", ou encore "RTC" (Real Time Clock).

Lors du démarrage, Linux initialise sa propre "horloge système" avec l'heure stockée dans le RTC, auquel, normalement, il ne touche plus par la suite. Une interruption "timer", mise en place au démarrage, incrémente régulièrement l'horloge système, qui n'est rien d'autre qu'une adresse mémoire. Le contenu de celle-ci peut être affiché avec la commande *date*, fournie par le paquetage GNU "shellutils" (cf. info date).

```
$date
sam sep 15 17:23:15 CEST 2001
```

La même commande permet de changer la date et l'heure, à condition d'avoir les droits de superutilisateur. Le format de la date à fournir peut prendre plusieur formes, notemment "MMJJHHmmYYYY". Par exemple, pour régler l'horloge sur le 15 septembre 2001 à 16h48 :

```
$sudo date 091516482001
```

Date modifie également la date et l'heure de l'horloge matérielle. Certaines distributions fournissent la commande *hwclock* qui permet de manipuler séparément ou de synchroniser les horloges matérielle et système (cf. man hwclock). Notons aussi qu'il existe une option de compilation du noyau qui permet de spécifier que l'horloge matérielle doit être régulièrement mise à jour (en fait toutes les 11 minutes) à partir de la valeur de l'horloge système.

Il faut signaler que ce n'est pas vraiment une bonne idée de modifier l'horloge d'un système en cours de fonctionnement : la discontinuité temporelle, si elle est importante, peut perturber certains services qui suppose un déroulement linéaire du temps. C'est encore plus risqué dans le cas d'un retour en arrière dans le temps. Dans la mesure du possible, il est donc plutôt recommandé de faire un shutdown, de modifier l'horloge matérielle dans le BIOS, et de rebooter.

Il n'est pas rare d'observer une décalage entre l'horloge système et l'heure exacte : une station Linux qui n'est pas rebootée pendant des mois peut voir son horloge décalée de plusieurs minutes par rapport à l'horloge murale. Une cause possible est que certaines interruptions "timer" ne sont pas traitées. Toutefois, si la station est connectée à un réseau où se trouve un serveur respectant le protocole ntp (Network Time Protocol), on peut synchroniser régulièrement l'horloge système avec ce serveur pour conserver l'heure correcte (voir *ntpdate* et *ntptimeset*).

Dans un programme en C, on peut connaître la date et l'heure grâce aux fonctions *time* et *gettimeofday*. La fonction *time* retourne, par convention, le nombre de secondes écoulées depuis le 1 janvier 1970, minuit (cf. man 2 time). Ce n'est pas un format forcément très pratique, mais il existe plusieurs fonctions qui permettent de convertir cette valeur entière, soit dans une chaîne ascii, soit dans une structure comprenant des champs séparés pour les minutes, heures, jours..etc (cf. man 3

ctime). Voici un exemple à compiler avec ‘gcc -o testtime testtime.c’ :

```
/* testtime.c: test de la fonction time */
#include <time.h>
#include <stdio.h>
int
main()
{
  time_t a;
  time(&a);
  printf(ctime(&a));
  return 0;
}
```

La seconde fonction, *gettimeofday*, fournit une résolution temporelle nettement plus fine, de l’ordre de la microseconde. C’est donc elle qu’il faudra utiliser pour des mesures d’une précision inférieure à la seconde. Par exemple, le programme suivant mesure le temps écoulé pendant l’exécution d’une boucle :

```
/* testgettimeofday.c */
#include <sys/time.h>
#include <unistd.h>
#include <stdio.h>

int
main() {
  struct timeval tv1,tv2;
  struct timezone tz;
  long long diff;
  int i;
  gettimeofday(&tv1, &tz);
   for (i=1;i<10000;i++) {}  /* code à timer */
  gettimeofday(&tv2, &tz);
  diff=(tv2.tv_sec-tv1.tv_sec) * 1000000L + \
           (tv2.tv_usec-tv1.tv_usec);
  printf("durée=%d usec\n",diff);
}
```

Au cas où une précision de l’ordre de la microseconde ne suffirait pas (!), il est également possible, sur les PCs équipés de processeurs de la classe Pentium, de

lire directement dans un timer interne appelé TSC (Time Stamp Counter). Ce registre compte en fait le nombre de cycles processeurs et permet ainsi de déduire le temps écoulé quand on connaît la fréquence du processeur. Pour une fréquence de 1 GHz, la précision est de l'ordre de la nanoseconde ! Voici une fonction (trouvée dans le code du paquetage latencycheck) qui permet de lire ce registre :

```
__inline__ unsigned long long int rdtsc()
 {
    unsigned long long int x;
    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
    return x;
 }
```

Au delà de sa non-portabilité (Linux tourne sur différentes architectures), l'utilisation de ce registre n'est pas sans problème : par exemple, si le processeur peut modifier sa fréquence, comme ça semble être le cas sur les Pentiums mobiles, le nombre de cycles ne reflétera pas le temps. Hormis des cas exceptionnels, il vaut donc mieux employer *gettimeofday*. (Signalons tout de même que la précision de *gettimeofday* n'est pas du tout garantie sur des systèmes unix non-Linux).

2 À la recherche du temps passé

Du point de vue d'un programme, il existe en réalité deux sortes de temps : d'une part le temps externe, c'est à dire celui que pourrait mesurer une horloge murale ; et d'autre part le temps CPU, c'est à dire le temps de calcul qui a été dévolu au programme. Dans un système multitâches comme Linux, un ordonnanceur de tâches distribue par petites tranches du temps CPU à tous les processus qui tournent sur la machine (voir le fichier linux/kernel/sched.c).

Dans l'exemple montrant l'utilisation de *gettimeofday* pour mesurer la durée d'exécution d'une portion de code, on mesurait en fait le temps externe. Or ce temps apparent d'exécution peut varier en fonction d'évènements externes tels que des interruptions ou des basculements de processus par l'ordonnanceur de tâches de Linux (d'ailleurs, si vous lancez ce programme de façon répétée, vous pourrez observer une variabilité de la durée d'autant plus importante que votre système est chargé). Si l'on veut mesurer le temps d'exécution CPU spécifique au processus courant,

on doit utiliser la fonction *clock* :

```
/* testclock.c */
#include <time.h>
#include <stdio.h>

int main() {
 clock_t a=clock();
 int i;
   for (i=1;i<1000000;i++) {}  /* code à timer */
 printf("durée= %f sec\n",(clock()-a)/CLOCKS_PER_SEC);
}
```

Si vous compilez cet exemple, vous réaliserez que la précision de *clock* n'est pas fameuse. Je soupçonne que l'information de temps CPU est gérée par l'ordonnanceur. Or ce dernier travaille sur une base temporelle de 100 Hz (10 msec). Donc *clock* n'est vraisemblablement pas adéquate pour mesurer des durées de l'ordre de la milliseconde ou moins (mais d'autres personnes, connaissant mieux le noyau Linux, me démentirons peut-être (?)). Signalons l'existence de la fonction *times*, qui détaille le temps CPU selon les modes utilisateur ou système.

3 Coincer la bulle

Dans un script shell, il est parfois nécessaire de devoir attendre qu'une certaine durée s'écoule avant de continuer un traitement ; la commande pertinente est *sleep*. Elle prend en argument le nombre de secondes pendant lesquelles le processus doit "dormir".

Dans un programme en C, de la même façon, on peut avoir besoin d'attendre un certain temps. Une idée naturelle est d'additionner le temps d'attente voulu à la valeur courante de l'horloge, puis de boucler jusqu'à ce que l'horloge atteigne cette nouvelle valeur. Cette technique, qui porte le nom de "busy wait", possède un défaut rédhibitoire dans certains cas : elle consomme "à fond" du temps CPU essentiellement pour ne rien faire. Les autres tâches risquent d'en pâtir, car cela réduit le temps CPU que leur alloue l'ordonnanceur. Il est donc plus sympathique de "laisser la main", c'est à dire mettre en sommeil le processus. La fonction pertinente est encore *sleep*, dont le prototype est définit dans `unistd.h`. Mais la

précision de *sleep* n'est que de l'ordre de la seconde. Pour une meilleure précision, on utilisera *nanosleep* définie dans `time.h`. Néanmoins, le nom de cette fonction ne doit pas faire illusion : loin d'être de l'ordre de la nanoseconde, la précision est plutôt de l'ordre du centième de seconde, du moins sur les PCs, et dans la version actuelle de Linux (cf. man nanosleep). Il faut également savoir que certains signaux peuvent "réveiller" le processus avant la fin de la période d'attente. Si tel est le cas, *nanosleep* renvoie par l'intermédiaire d'un de ses arguments, le temps d'attente qui reste à effectuer. Pour pouvoir attendre une fraction de milliseconde, une solution est d'utiliser l'interruption rtc, comme nous le verrons dans la section suivante.

4 Lancer des tâches périodiques

Une possibilité intéressante est de pouvoir lancer certaines tâches de façon périodique. Pour cela on dispose sur la plupart des systèmes Linux du mécanisme "cron". Lors de l'amorçage, Linux démarre un processus "cron" qui inspecte, à chaque minute, des tables dans lesquelles sont spécifiées des listes de tâches à effectuer, avec leur moments de lancement.

Une telle table contient une ligne par tâche. Les cinq premières colonnes sont des masques qui spécifient les minutes, heures, jours, mois et jour de la semaine où le processus doit être lancé (cf. "man 5 crontab" pour les détails sur la syntaxe de ces fichiers). Le reste de ligne spécifie la commande à exécuter.

La commande *crontab -e* ouvre une session dans l'éditeur 'vi' pour éditer la table des tâches. On peut ajouter dans cette table, par exemple, la ligne suivante :

```
1 * * * * /bin/date >>$HOME/aga
```

À partir de ce moment, à la première minute de chaque heure, la sortie de la commande *date* sera ajoutée dans le fichier "aga" du répertoire utilisateur (Il faut toutefois que les droits adéquats ait été spécifiés par le superutilisateur dans les fichiers /var/cron/allow et /var/cron/deny). Un *crontab -r* mettra fin à cette programmation.

Dans un programme en C, il n'est pas question d'utiliser le mécanisme "cron" pour lancer une fonction périodiquement. On doit gérer soi-même cette program-

mation. Une solution élégante fait appel aux threads : elle consiste à créer un thread effectuant une boucle qui vérifie l'horloge régulièrement et, aux moments adéquats, exécute la fonction désirée. Il n'est pas question de rentrer ici dans des explications sur la programmation par threads (voir un numéro précédent de Linux Magazine), mais nous voulons signaler un mécanisme qui assure une assez bonne périodicité. Cette solution est inspirée des fichiers `rtc.txt` et `rtc.c` fournis avec le noyau Linux.

On peut configurer l'horloge matérielle (le RTC) pour qu'elle envoit périodiquement (à une fréquence comprise entre 2Hz et 8192Hz) une interruption au processeur. Pour cela, il faut ouvrir le device `/dev/rtc` et utiliser des appels *ioctl* tels qu'ils sont documentés dans le fichier `rtc.txt`. Cela fait, le processus peut utiliser les fonctions *select* ou *read* pour dormir en attendant ces interruptions, et les traiter dès qu'elles arrivent. Nous nous servons de ce mécanisme dans le programme suivant, qui lit à chaque milliseconde le status du port parallèle, et s'arrête dès qu'un bit change (ce programme a pour but de détecter un "top synchro" envoyé par un appareil externe).

```
/* Lecture périodique (1024Hz) du port parallel */
/* compiler avec  gcc -Wall testrtc.c -lm -O2 -o testrtc */
#include <stdio.h>
#include <linux/mc146818rtc.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <asm/io.h>
#include <sys/io.h>
#include <sched.h>
#include <sys/time.h>

#define LPT1         0x378

int main(void) {

  int fd, retval, irqcount, e,a;
  unsigned long data;
  struct sched_param sp ;
  struct timeval tv,tv2;
  struct timezone tz;
```

```
  /* passage en priorité << temps réel >> */
  sp.sched_priority = 60 ;
  sched_setscheduler (0, SCHED_FIFO, &sp);

  /* ouverture du port parallèle */
  ioperm(LPT1,3,1);
  a=inb(LPT1+1);

  /* programmation du rtc à 1024Hz */
  fd = open ("/dev/rtc", O_RDONLY);
  retval = ioctl(fd, RTC_IRQP_SET, 1024);
  retval = ioctl(fd, RTC_PIE_ON, 0);

  irqcount = 0;
  e = 0;
  while ((irqcount<1024)&&(!e)) {
    gettimeofday(&tv,&tz);
    /* check if a bit on the status port has changed */
    if (inb(LPT1+1)!=a) e=irqcount;
    /* attend l'interruption suivante */
    retval = read(fd, &data, sizeof(unsigned long));
    irqcount++;
    gettimeofday(&tv2,&tz);
    printf("%f msec\n",(tv2.tv_sec-tv.tv_sec)*1000.0 + \
           (tv2.tv_usec-tv.tv_usec)/1000.0);
  }

  if (e) {fprintf(stderr,"Event detected at %dms\n",e);}

  /* Supprime l'interruption périodique */
  retval = ioctl(fd, RTC_PIE_OFF, 0);
  close(fd);
  return 0;
}
```

Ce programme doit absolument être exécuté sous l'id root, sous peine de genérer une faute de segmentation. Avoir les droits de root est nécessaire pour trois raisons : pour pouvoir activer la priorité "temps réel", pour lire le port parallèle, et pour autoriser le rtc à interrompre le processeur 1024 fois par seconde. Si vous n'avez pas un appareil qui vous permet de basculer un des bit des lignes de contrôle du port parallèle (cf. p.ex. : http://www.lscp.net/expe/doc/parallel/parallel.html),

ce programme affichera simplement la durée entre deux interruptions. Cet exemple montre qu'un noyau Linux standard, bien que son ordonnanceur fonctionne par tranches de 10 msec, permet de réaliser certaines applications qui nécéssitent des latences de détection de l'ordre de la milliseconde.

5 Références

Time-related functionality in Linux kernels 2.x.x. par A. Balsa.

Real Time Clock Driver for Linux par P. Gortmaker.

Paquetage latencytest-0.42-png: cf. http://www.gardena.net/benno/linux/audio

Real-time data collection in Linux: a case study par S. Finney.