第五章 UNIX 时间与定时器

操作系统为进程调度、网络协议超时以及定期更新系统的统计信息等目的使用定时器。应用程序通过对系统时间和定时器函数的访问来测量性能或确定事件发生的事件。

1、POSIX 时间

POSIX 规定系统应该记录从 Epoch 开始的以秒为单位的时间，每天都被精确的计为 86400 秒。Epoch（新纪元）被定义为协调世界时（也称为 UTC，格林尼治标准时间或 GMT）的 1970 年 1 月 1 日，00：00（午夜）。

1.1 用从 Epoch 开始的秒数来表示时间

POSIX 基本标准只支持秒级的分辨率，并用类型 time_t 来表示从 Epoch 开始的时间，time_t 类型通常用 long 类型来实现。程序可以调用 time 函数来访问系统时间（用从 Epoch 开始的秒数表示），如果 tloc 不为 NULL，time 函数还会将时间存储在*tloc 中。

```
SYNOPSIS
#include <time.h>
time_t time(time_t, *tloc);
double difftime(time_t time1, time_t time0);       POSIX:CX
```

如果成功，time 返回从 Epoch 开始计算的秒数，否则返回（time_t）- 1，POSIX 没有为 time 定义任何必须检测的错误。

对一个 32 比特的 long 类型来说，时间大约会在 1970 年 1 月 1 日之后 68 年溢出，对使用 unsigned long 类型的 time_t 值来说，溢出会在 2106 年，对一个 64 比特的数据类型来说，溢出会在 2920 亿年之后出现。

difftime 函数负责计算两个 time_t 类型的日历时间之间的差值。

下面的程序计算了执行 function_to_time 函数所花的时钟时间：

```
#include <stdio.h>
#include <time.h>
void function_to_time(void);
```

```
int main(void)
{
    time_t tstart;
    tstart = time(NULL);
    function_to_time();
    printf("function_to_time took %f seconds of elapsed time\n",
difftime(time(NULL), tstart));
    return 0;
}
```

上例使用的时间分辨率为秒，一般不准确。

1.2 显示日期和时间

对那些需要计算时间差值的计算来说，使用 time_t 类型很方便，但用它来打印日期就很繁琐。

函数 localtime 的参数用来说明从 Epoch 开始的秒数，其返回的结构中带有根据本地需求调整过的时间成份。asctime 函数将 localtime 返回的结构转换成字符串。ctime 函数的功能等同于 asctime（localtime（clock））。gmtime 函数的参数也是从 Epoch 开始的秒数，其返回的结构中带有协调世界时表示（UTC）的时间成份。

```
SYNOPSIS
#include <time.h>
struct tm *localtime(const time_t *timer);
char *asctime(const struct tm *timer);
char *ctime(const time_t *clock);
struct tm *gmtime(const time_t *timer);        POSIX:CX
```

没有为这些函数定义错误。ctime 函数有一个参数，一个指向 time_t 类型变量的指针，并返回一个由 26 个字母组成的字符串的指针。字符串可能按照下列形式存储：

```
Sun Oct  06 02：21：35 1986\n\0
```

ctime 函数用静态存储的方式来保存字符串，对 ctime 的两次调用会将字符串存储在同一个位置上，在使用时要注意。

下面程序正确显示了函数 function_to_time 执行之前和之后的时间：

```
#include <stdio.h>
#include <time.h>

void function_to_time(void)
{
    sleep(2);
}

int main(void)
{
    time_t t;
    char s[26];

    t = time(NULL);
    strncpy(s, ctime(&t), 26);
    function_to_time();
    t = time(NULL);
    printf("The time before was %sThe time after was %s", s, ctime(&t));
    return 0;
}
```

1.3 用 struct timeval 来表示时间

对程序定时或者控制程序事件来说，用秒作为时间的尺度太粗糙了。struct timeval 结构包含以下成员：

time_t tv_sec;　　/* 从 Epoch 开始的秒数 */

time_t tv_usec;　　/* 从 Epoch 开始的微秒数 */

gettimeofday 函数用来获取自 Epoch 以来的、用秒和微秒表示的系统时间，tp 指向的 struct timeval 结构负责接收获取的时间，由于历史的原因，tzp 必须为 NULL。

```
SYNOPSIS
#include <sys/time.h>
```

```
int gettimeofday(struct timeval *restrict tp, *tzp);POSIX:CX
```

函数 gettimeofday 返回 0，没有保留其它的值来指示错误，但是很多系统实现 gettimeofday 时都使其不成功时返回-1 并设置 errno。

下面的程序通过在一个循环中调用 gettimeofday 来测试 gettimeofday 的分辨率，循环一直执行到产生 20 个差值为止，最后显示了调用次数，每次的微秒数，不同值时的差值及平均差值。大多数系统中，分辨率都是几个微妙。在大多数现代系统中，对 gettimeofday 的多次连续调用都会返回相同的值。注意 MMILLION 和 MILLION 的定义，数组 timecount 必须定义为 long long 型，其个数 NUMDIF*3 是经验值。

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000L
#define MMILLION 1000000LL
#define NUMDIF 20

int main(void)
{
   int i;
   int numcalls = 1;
   int numdone = 0;
   long sum = 0;
   long timedif[NUMDIF];
   long long timecount[NUMDIF * 3];
   struct timeval tlast;
   struct timeval tthis;

   if (gettimeofday(&tlast, NULL)){
      fprintf(stderr,"Failed to get first gettimeofday.\n");
      return 1;
   }
```

```
    timecount[numcalls] = MMILLION * tlast.tv_sec + tlast.tv_usec;
    while (numdone < NUMDIF){
        numcalls++;
        if (gettimeofday(&tthis, NULL)){
            fprintf(stderr, "Failed to get later gettimeofday.\n");
            return 1;
        }
    timecount[numcalls] = MMILLION * tthis.tv_sec + tthis.tv_usec;
        timedif[numdone] = MILLION*(tthis.tv_sec-tlast.tv_sec) +
                  tthis.tv_usec-tlast.tv_usec;
        if (timedif[numdone] != 0){
            numdone++;
            tlast = tthis;
        }
    }
    printf("%d calls to gettimeofday were required\n", numcalls);
    for (i =1; i <= numcalls; i++)
        printf("%2d:  %lld microseconds\n", i, timecount[i]);
    printf("Found %d differences in gettimeofday:\n", NUMDIF);
    for (i =0; i<NUMDIF; i++){
        printf("%2d:%10ld microseconds\n", i, timedif[i]);
        sum += timedif[i];
    }
    printf("The average defference is %f\n", sum/(double)NUMDIF);
    return 0;
}
```

练习 5-1：编写测试函数运行时间精确到微秒的程序。

1.4 使用实时时钟

时钟（clock）是一个计数器，它的值以固定间隔增加，这个固定间隔被称为时钟分辨

率（clock resolution）。POSIX：TMR 扩展中包含了各种用 clockid_t 类型的变量表示的时钟，POSIX 时钟可能是系统范围的时钟，也可能只在某一个进程中可见。所有支持 POSIX 标准的 UNIX 实现都必须支持一个系统范围内的、clockid_t 的值为 CLOCK_REALTIME 的时钟，这个时钟对应于系统时钟。clockid_t 被定义为 int 类型，表示时钟的编号。struct timespec 结构用来为 POSIX：TMR 时钟指定时间（在定时器中也需要 timespec 结构）。struct timespec 结构至少包含下列成员：

```
time_t    tv_sec          /* 秒 */
long      tv_nsec         /* 纳秒 */
```

POSIX 提供了设置时钟时间的函数（clock_settime），获取时钟时间的函数（clock_gettime）和确定时钟分辨率的函数（clock_getres）。每个函数都有两个参数：用来标识特定时钟的 clockid_t 和一个指向 struct timespec 结构的指针。

```
SYNOPSIS
#include <time.h>
int clock_getres(clockid_t cid, struct timespec *res);
int clock_gettime(clockid_t cid, struct timespec *tp);
int clock_settime(clockid_t cid, struct timespec *tp);POSIX:TMR
```

如果成功，这些函数就返回 0，否则返回-1 并设置 errno。如果 cid 指定的不是一个已知的时钟，那么这三个函数都将 errno 设置为 EINVAL，如果 tp 的值超出了 cid 的范围，或者 tp->tv_nsec 不在[0，10^9]范围内，clock_settime 也会将 errno 设置为 EINVAL。

下面的程序用 POSIX：TMR 时钟测量了 function_to_time 的运行时间。

```
#include <stdio.h>
#include <time.h>
#define MILLION 1000000L

void function_to_time(void)
{
   int x,i;
   for (i = 0; i < 10; i++)
      x = i * sin(i);
```

```
}

int main(void)
{
    long long timedif;
    struct timespec tpstart,tpend;

    if (clock_gettime(CLOCK_REALTIME, &tpstart) == -1){
        perror("Failed to get starting time");
        return 1;
    }
    function_to_time();
    if (clock_gettime(CLOCK_REALTIME, &tpend) == -1){
              perror("Failed to get ending time");
              return 1;
        }
    timedif = MILLION * (tpend.tv_sec - tpstart.tv_sec) +
         (tpend.tv_nsec - tpstart.tv_nsec)/1000;
    printf("It took %ld microseconds\n", timedif);
}
```

CLOCK_REALTIME 的分辨率通常都高于 gettimeofday。下面的程序类似于前面测量 gettimeofday 分辨率的程序，通过测量时钟读的过程中 20 次变化的平均值来测试 CLOCK_REALTIME 的分辨率。程序还在前面调用 clock_getres 来显示设置时钟和定时器中断时使用的用纳秒表示的标称分辨率，标称分辨率通常都很粗，达到了毫秒的量级，而且与计时中使用的 clock_gettime 的分辨率无关。

```
#include <stdio.h>
#include <time.h>
#define BILLION 1000000000L
#define NUMDIF 20
```

```
    int main(void)
    {
       int i;
       int numcalls = 1;
       int numdone = 0;
       long sum = 0;
       long timedif[NUMDIF];
       struct timespec tlast;
       struct timespec tthis;

       if (clock_getres(CLOCK_REALTIME, &tlast))
          perror("Failed to get clock resolution");
       else if (tlast.tv_sec != 0)
          printf("Clock resolution no better than one second\n");
       else
          printf("Clock    resolution:    %ld    nanoseconds\n",
(long)tlast.tv_nsec);
       if (clock_gettime(CLOCK_REALTIME, &tlast)){
          perror("Failed to get first time");
          return 1;
       }
       while (numdone < NUMDIF){
          numcalls++;
          if (clock_gettime(CLOCK_REALTIME, &tthis)){
             perror("Failed to get a later time");
             return 1;
          }
          timedif[numdone] = BILLION*(tthis.tv_sec-tlast.tv_sec) +
```

```
                    tthis.tv_nsec-tlast.tv_nsec;
            if (timedif[numdone] != 0){
                numdone++;
                tlast = tthis;
            }
        }
        printf("%d calls to CLOCK_REALTIME were required\n",
numcalls);
        printf("Found %d differences in CLOCK_REALTIME:\n", NUMDIF);
        for (i =0; i<NUMDIF; i++){
            printf("%2d:%10ld nanoseconds\n", i, timedif[i]);
            sum += timedif[i];
        }
        printf("The average nonzero defference is %f\n",
sum/(double)NUMDIF);
        return 0;
    }
```

1.5 响应时间和 CPU 时间的对比

响应时间是一个程序从开始运行到结束时的全部时间，CPU 时间则指在响应时间中 CPU 真正执行程序的时间。前面的例子测量的都是响应时间。times 函数可以测量 CPU 时间。

```
SYNOPSIS
#include <sys/time.h>
clock_t times(struct tms *buffer);                    POSIX
```

如果成功，times 函数用时间账单填充参数 buffer 指向的 struct tms 结构，struct tms 结构至少包含以下成员：

```
clock_t     tms_utime       /* 进程的用户 CPU 时间 */
clock_t     tms_stime       /* 由进程使用的系统 CPU 时间 */
clock_t     tms_cutime      /* 进程及其已终止的子进程的用户 CPU 时间 */
```

clock_t tms_cstime /* 由进程及其子进程使用的系统 CPU 时间 */

如果 times 失败，就返回（clock_t）-1 并设置 errno。

times 返回用时钟嘀嗒计数表示的实际耗费时间，这个时间是从过去的任意一点开始计算的，比如可以从系统或进程的起始时间开始计算。典型的嘀嗒计数值为每秒钟 100 次，这个值对记帐来说是适合的，但要对短暂的事件进行性能测试，这个分辨率就不够了。

2、睡眠时间

自愿地阻塞一段时间的进程被称为在睡眠，sleep 函数使调用进程挂起，直到经过了特定的秒数，或者调用进程捕捉到信号的时候为止。

```
SYNOPSIS
#include <unistd.h>
unsigned sleep(unsigned seconds);                    POSIX
```

如果请求的时间到了，sleep 函数就返回 0，如果被中断了，sleep 函数就返回还剩的时间。sleep 与 SIGALRM 之间有交互，所以应该避免在同一个进程中同时使用它们。

```
SYNOPSIS
#include <time.h>
int nanosleep(const struct timespec *rqtp,
                      struct timespec *rmtp);     POSIX:TMR
```

nanosleep 允许使用纳秒级的分辨率。系统时钟 CLOCK_REALTIME 决定了 rqtp 的分辨率。如果成功，nanosleep 返回 0，如果不成功，返回-1 并设置 errno。

下面的程序测试了 nanosleep 函数的分辨率，它对 nanosleep 执行了 100 次调用，每次使用的睡眠时间都是 1000 纳秒。如果 nanosleep 的分辨率为 1 纳秒，那么执行会在 100 微妙内结束，在一个分辨率为 10ms 的系统上，程序执行完大约要花 1 秒的时间。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#define COUNT 100
#define D_BILLION 1000000000.0
```

```
#define D_MILLION 1000000.0
#define MILLION 1000000L
#define NANOSECONDS 1000

int main(void)
{
   int i;
   struct timespec slptm;
   long tdif;
   struct timeval tend, tstart;

   slptm.tv_sec = 0;
   slptm.tv_nsec = NANOSECONDS;
   if (gettimeofday(&tstart, NULL) == -1){
      fprintf(stderr, "Failed to get start time\n");
      return 1;
   }
   for (i = 0; i < COUNT; i++)
      if (nanosleep(&slptm, NULL) == -1){
         perror("Failed to nanosleep");
         return 1;
      }
   if (gettimeofday(&tend, NULL) == -1){
      fprintf(stderr, "Failed to get end time\n");
         return 1;
   }
   tdif = MILLION * (tend.tv_sec - tstart.tv_sec) +
           tend.tv_usec - tstart.tv_usec;
   printf("%d  nanosleeps  of  %d  nanoseconds\n",  COUNT,
```

```
NANOSECONDS);
        printf("Should take %d microseconds or %f seconds\n",
            NANOSECONDS * COUNT / 1000, NANOSECONDS * COUNT / D_BILLION);
        printf("Actually took %ld microseconds or %f seconds\n",
            tdif, tdif / D_MILLION);
        printf("Number of seconds per nanosleep was %f\n",
            (tdif / (double)COUNT) / MILLION);
        printf("Number of seconds per nanosleep should be %f\n",
            NANOSECONDS / D_BILLION);
        return 0;
    }
```

在 Linux 中，系统调用 nanosleep 使调用进程进入睡眠，但是时钟中断的周期为 10 毫秒，如果进程进入睡眠而循正常途径由时钟中断处理程序而唤醒的话，就只能达到 10 毫秒的精度。因此，系统对那些要求睡眠时间小于 2 毫秒的进程采用延迟 2 毫秒而不是睡眠。因此可以解释上例程序的运行结果。

3、POSIX：XSI 间隔定时器

前面讨论的时钟采用增量的方式来记录所经过的时间。定时器不同于时钟，定时器通常减少它的值，并在为 0 时产生一个信号。计算机系统中通常只有少量的硬件间隔定时器，操作系统通过使用这些硬件定时器可以实现多个软件定时器。

在操作系统中，定时器有许多重要的应用。如操作系统调度一个进程时，就会启动一个时间长度为调度时间片（scheduling quantum）的间隔定时器，如果这个定时器到而进程还在执行，调度程序就将该进程转移到一个就绪队列中去。再比如，大多数调度算法都有提升那些已经等待了很长时间的进程的优先级的机制，调度程序可以为此设置一个间隔定时器来进行优先级管理。每次定时器到时，调度程序都会提升那些还未执行的进程的优先级。

POSIX：XSI 扩展的间隔定时器使用了一个包含下列成员的 struct itimerval 结构：

```
struct    timeval    it_value        /* 到下一次到期为止剩余的时间 */
struct    timeval    it_interval     /* 重新装载进定时器的值 */
```

回想一下，struct timeval 结构中是包含秒和微秒字段的。

一个遵循 POSIX：XSI 的实现必须为每个进程提供下列三种用户间隔定时器：

ITIMER_REAL：按照实际时间递减，并在定时器到期时产生一个 SIGALRM 信号；ITIMER_VIRTUAL：按照虚拟时间（进程使用的时间）递减，并在定时器到期时产生一个 SIGVTALRM 信号；ITIMER_PROF：按照虚拟时间和进程的系统时间递减，并在定时器到期时产生一个 SIGPROF 信号。

函数 getitimer 用来获取某定时器当前的时间间隔，setitimer 函数用来启动和终止用户的间隔定时器。参数 which 用来指定定时器（即 ITIMER_REAL、ITIMER_VIRTUAL 和 ITIMER_PROF）。getitimer 将定时器 which 中当前的时间值保存在 value 指向的位置上。setitimer 用 value 指向的值来设置 which 指定的定时器。如果 ovalue 不为 NULL，setitimer 就将定时器以前的值放在 ovalue 指向的位置中。如果定时器在运行中，*ovalue 的成员 it_value 就为非零，并包含了定时器到期之前的剩余时间。

```
SYNOPSIS
#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value,
              struct itimerval *restrict ovalue); POSIX:XSI
```

如果成功，这两个函数返回 0，否则返回-1 并设置 errno。如果 value 中的微秒数不在[0，10^6]范围内，setitimer 就将 errno 置为 EINVAL。如果*value 的成员 it_interval 不为 0，定时器到期时就用这个值重启。如果*value 的 it_interval 为 0，定时器到期后就不会重启了。如果*value 的 it_value 为 0，那么如果定时器在运行，setitimer 就将其停止。

下面的程序用一个 ITIMER_PROF 定时器，每使用 2 秒的 CPU 时间，就打印一个星号。程序先调用 setupinterrupt 来安装 myhandler，作为 SIGPROF 的信号处理程序。然后程序调用 setupitimer 把 ITIMER_PROF 设置成为一个 2 秒的间隔定时器（将 it_value 和 it_interval 设置一样，就成了间隔定时器）。进程每使用 2 秒的 CPU 时间，定时器 ITIMER_PROF 就产生一个 SIGPROF 信号。进程捕捉到该信号，并用 myhandler 对它进行处理。这个处理程序向标准错误输出一个星号。

```
#include <errno.h>
#include <signal.h>
#include <stdio.h>
```

```
    #include <unistd.h>
    #include <sys/time.h>

    static void myhandler(int s)
    {
       char aster = '*';
       int errsave;

       errsave = errno;
       write(STDERR_FILENO, &aster, 1);
       errno = errsave;
    }

    static int setupinterrupt(void)
    {
       struct sigaction act;

       act.sa_handler = myhandler;
       act.sa_flags = 0;
       return (sigemptyset(&act.sa_mask) || sigaction(SIGPROF, &act,
NULL));
    }

    static int setupitimer(void)
    {
       struct itimerval value;

       value.it_interval.tv_sec = 2;
       value.it_interval.tv_usec = 0;
```

```
    value.it_value = value.it_interval;
    return (setitimer(ITIMER_PROF, &value, NULL));
}

int main(void)
{
    if (setupinterrupt() == -1){
        perror("Failed to set up handler for SIGPROF");
        return 1;
    }
    if (setupitimer() == -1){
        perror("Failed to setup the ITIMER_PROF interval timer");
        return 1;
    }
    for (;;);
}
```

利用 POSIX：XSI 扩展的间隔定时器及其操作函数，同样可以实现测量一个程序运行时间的功能。下面的程序是其模型。程序测量的是进程的响应时间，故用 ITIMER_REAL 间隔定时器，将其间隔时间设置得很大，在进程开始时启动，进程结束时用 getitimer 函数读取剩余时间，最后算得响应时间。

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000L

void function_to_time(void)
{
    int i;
    float x;
```

```
        for (i = 0; i < 10000; i++)
            x = i * sin(i);
        sleep(1);
    }

    int main(void)
    {
        long diftime;
        struct itimerval value, ovalue;

        ovalue.it_interval.tv_sec = 0;
        ovalue.it_interval.tv_usec = 0;
        ovalue.it_value.tv_sec = MILLION;
        ovalue.it_value.tv_usec = 0;
        if (setitimer(ITIMER_REAL, &ovalue, NULL) == -1){
            perror("Failed to set real timer");
            return 1;
        }
        function_to_time();
        if (getitimer(ITIMER_REAL, &value) == -1){
            perror("Failed to get real timer");
            return 1;
        }
        diftime    =    MILLION    *    (ovalue.it_value.tv_sec    -
value.it_value.tv_sec) +
                    ovalue.it_value.tv_usec - value.it_value.tv_usec;
        printf("The function took %ld microseconds or %f seconds\n",
            diftime, diftime / (double)MILLION);
        return 0;
```

}

4、实时信号

在基本的 POSIX 标准中，信号处理程序是一个带有单个整型参数的函数，这个整数代表了所产生信号的信号码。POSIX：XSI 扩展和 POSIX：RTS 实时信号扩展都对信号处理能力进行了扩展，包含了信号排队和向信号处理程序传递信息的能力。

5、POSIX：TMR 间隔定时器

POSIX：XSI 扩展的间隔定时器功能分配给每个进程少量的、固定数目的定时器，如 ITIMER_REAL、ITIME_VIRTUAL、ITIME_PROF 等每种类型各一个。POSIX：TMR 扩展采用了一种替换方法，在这种方法中只有少量的时钟（例如 CLOCK_REALTIME），但一个进程可以为每个时钟创建很多独立的定时器。

6、定时器漂移、超限和绝对时间

如前所述，一个 POSIX：TMR 定时器和 POSIX：XSI 定时器有关的问题就是根据相对时间来设置这些定时器的方式。假设设置了一个间隔 2 秒的周期性中断，当定时器到期时，系统自动的用另一个 2 秒的间隔来重启定时器，假设从定时器到期到定时器被重新设置之间的等待时间为 5 微秒，那么，定时器的实际周期就是 2.000005 秒，在 1000 次中断之后，定时器会偏移 5 毫秒。这种不准确性就被称为定时器漂移（timer drift）。如果定时器是从定时器的信号处理程序中重启的，问题就会更严重一些。

处理漂移问题的一种方法是记录定时器实际上什么时候应该到期，并调整每次设置定时器的值。称为用绝对时间而不是相对时间来设定定时器。

7、Linux 时钟中断机制

时间在一个操作系统内核中占据着重要的地位，它是驱动一个 OS 内核运行的“起博器”。一般说来，内核主要需要两种类型的时间：（1）、在内核运行期间持续记录当前的时间与日期，以便内核对某些对象和事件作时间标记（timestamp，也称为“时间戳”），或供用户通过时间 syscall 进行检索；（2）、维持一个固定周期的定时器，以提醒内核或用户一段时间已经过去了。

PC 机中的时间由三种时钟硬件提供，而这些时钟硬件又都基于固定频率的晶体振荡器来提供时钟方波信号输入。这三种时钟硬件是：（1）实时时钟（Real Time Clock，RTC）；（2）可编程间隔定时器（Programmable Interval Timer，PIT）；（3）时间戳计数器（Time Stamp Counter，TSC）。

实时时钟 RTC 是一个时钟芯片，主要记录 Epoch 时间，RTC 通过主板上的电池来供电的，因此当 PC 机关掉电源后，RTC 仍然会继续工作。RTC 不仅计时而且还提供不同频率的方波供系统用。RTC 与操作系统的交互有两点：其一是系统启动时，操作系统从 RTC 读入时间作为系统时间，将其放在 struct timeval xtime 中，xtime 是一个全局变量；其二是在需要的时刻将系统时间写入 RTC（比如修改时间）。

可编程间隔定时器 PIT 对 RTC 输出的方波进行计数。Linux 用宏 CLOCK_TICK_RATE 来表示 PIT 的输入方波的频率（在 PC 机中这个值通常是 1193180HZ），用宏 LATCH＝（CLOCK_TICK_RATE）/（HZ）来表示计数值，其中 HZ=100。即 PIT（严格来说是其 0 通道）每隔 10ms 就会计数到 0，它就在 IRQ0 上产生一次时钟中断，称为一次时钟滴哒。时钟嘀嗒非常重要，内核全局变量 jiffies（32 位无符号整数）用来表示内核本次启动以来的时钟滴答次数。对 jiffies 加 1 应是原子操作（(*（unsigned long *）&jiffies）++）。定时器大多基于时钟中断（但需要修正）。

TSC 是 CPU 中一个 64 位的专用寄存器，用来记录输入 CPU 的方波次数，若某 CPU 的主频为 500MHZ，则 TSC 每个 2ns 计数一次。利用 TSC 可以改善时钟中断的精度，利用 TSC 可以实现高精度的计时。下面程序片断在用户空间读取 TSC 的值：

```
#include <asm/msr.h>
#include <stdio.h>
#include <asm/types.h>

int main()
{
    __u64 i;
    rdtscll(i);
    printf("%llu\n", i);
    return 0;
```

```
}
```

更多信息，请参考资料“Linux 时钟机制”。