第8章 时钟与定时器

实验目的

- 深入了解文件管理涉及的概念和功能
- 理解文件系统如何组织和管理信息
- 通过编程模拟实现一个文件系统

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

定时器机制的作用

- 操作系统的许多活动由定时测量来驱动。
- 时钟是操作系统进行调度工作的重要工具。
- 时间标记必须由内核自动地设置，Linux操作系统中，有两种定时测量:
  - 维护系统当前时间和日期;
  - 间隔定时器，又称定时器。

Linux与其他操作系统的时间标准

- 1)以国际标准时间而非本地时间计时;
- 2)可自动进行转换，例如变换到夏令时;
- 3)将时间和日期作为一个量值保存。

time( )函数

time( )函数返回当前时间和日期

- #include <time.h>
- time_t time(time_t *tloc);
- 时间值作为函数值返回。如果参数非NULL，则时间值也存放在由tloc指向的主存单元内。

time( )函数的用法

```
/*****envtimetest.c*****/
#include <time.h>
#include <stdio.h>
#include <unistd.h>
int main( )
{
    int i;
    time_t the_time;
    for ( i=1; i<=10; i++ ) {
        the_time = time( (time_t *) 0 );
        printf("The time is %ld\n ", the_time);
            sleep(2);
        }
        exit(0);
}
```

difftime（）函数

difftime（）函数用来计算两个time_t值之间的秒数并以double类型返回

- #include <time.h>
- double difftime(time_t time1, time_t time2);
- difftime（ ）函数计算两个时间值之间的差，并将time1-time2的值作为浮点数返回。
- 对Linux来说，time（ ）函数的返回值是秒数，但考虑到最大限度地增加可移植性，则最好使用difftime（ ）。

各种时间函数之间的关系

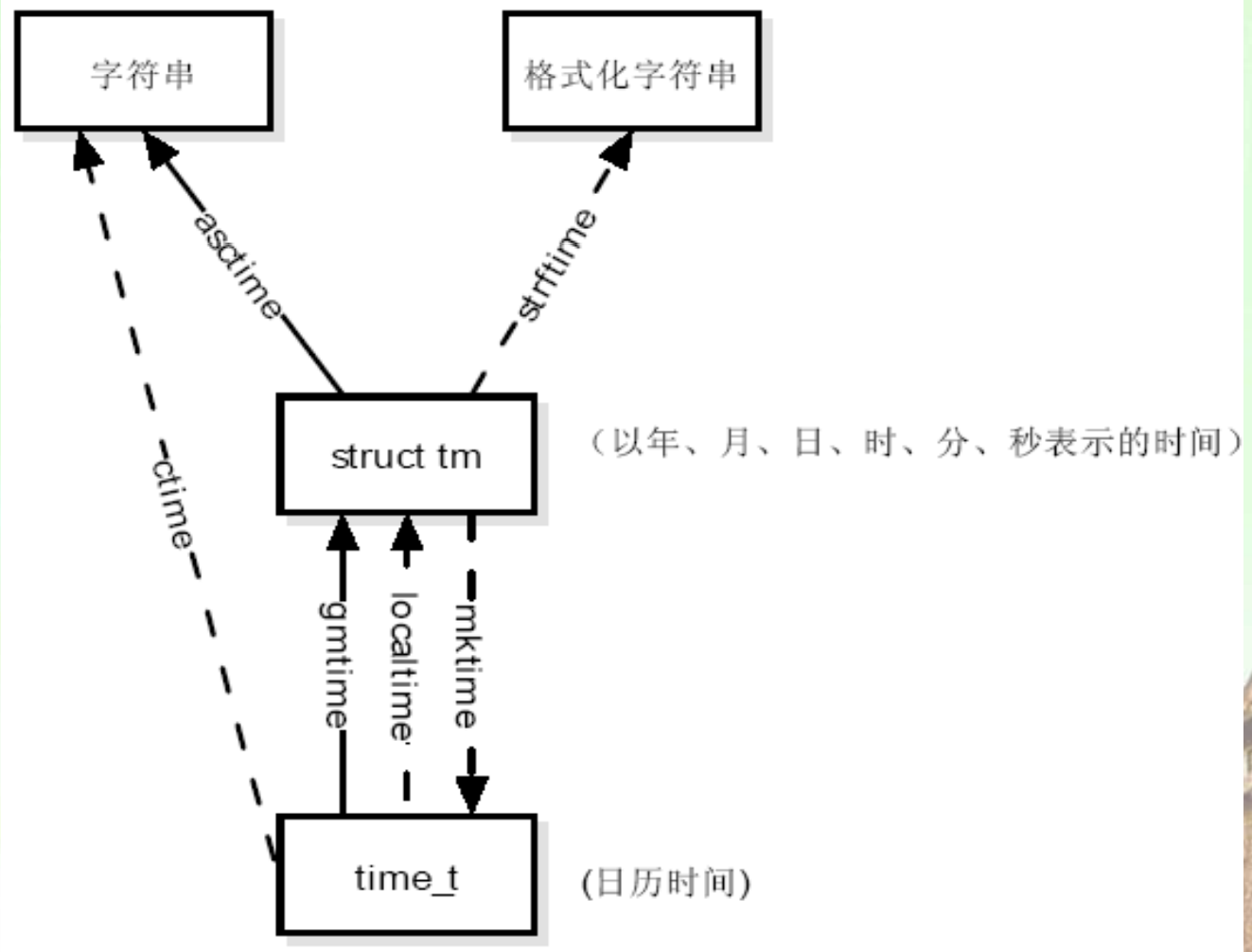

图 8-1 各种时间函数之间的关系

localtime（）和gmtime（） 函数

这两个函数将日历时间变换成以年、月、日、时、分、秒、星期几表示的时间，并将这些存放在一个tm结构中。

```
struct tm{
        int tm_sec;     /* 秒数*/
        int tm_min;     /* 分钟*/
        int tm_hour;    /*小时*/
        int tm_mday;    /*日*/
        int tm_mon;     /*月*/
        int tm_year;    /*年*/
        int tm_wday;    /*星期几*/
        int tm_yday;    /*从1月1日开始的时间*/
        int tm_isdst;   /*是否夏令时*/
}
```

localtime()和gmtime()函数原型

- #include <time.h>
- struct tm *gmtime(const time_t *calptr);
- struct tm *localtime(const time_t *calptr);
- 这两个函数都返回指向tm结构的指针。
- localtime（）和gmtime（）之间的区别是：localtime（）将日历时间变换成本地时间（考虑到本地时区和夏令时标志），而gmtime（）则将日历时间变换成国际标准时间的年、月、日、时、分、秒、星期几。

mktime()函数

函数mktime（ ）以本地时间的年、月、日等作为参数，将其变换成time_t值。

- #include <time.h>
- time_t mktime(struct tm *tmptr);
- 函数在成功时会返回日历时间，失败时返回-1;

asctime ()和ctime ()函数

asctime（ ）和ctime（ ）函数产生长度为26的字符串，这以date命令的系统默认输出形式类似:

Sun Jun 6 12:30:34 1999

- 两个函数的区别在于，asctime（ ）的参数是一个struct tm结构的指针，而ctime（ ）的参数是一个time_t指针。
- #include <time.h>
- char *asctime(const struct tm *timeptr);
- char *ctime(const time_t *timeval);
- ctime（ ）函数等效于调用asctime(localtime(timeval));

strftime ()函数

- strftime（ ）函数与printf（ ）函数非常相似，该函数的原型为:
- #include<time.h>
- size_t strftime(char *buf, size_t maxsize, const char *format, const struct tm *tmptr);
- 该函数的最后一个参数要求格式化的时间值。格式化结果存放在一个长度为maxsize个字符的buf数组中。format参数控制时间值的格式，如同printf( )函数一样，变换说明的形式是百分号后跟一个特定字符。format中的其他字符则按原样输出。

strftime转换控制符

时间函数与TZ

- **ctime（）、localtime（）、mktime（）、strftime（）4个函数受到环境变量TZ的影响，如果定义过TZ，则这些函数将使用其值以代替系统默认时区，如果TZ定义为空串，则使用国际标准时间。**

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

系统定时器

Linux系统定时器又分两种，
第1种是老定时器机制，由一个32位指针的静态数组定义，每个指针可指向一个timer_struct结构。

```
struct timer_struct {
    unsigned long expires;      /*定时器激活时间*/
    void (*fn)(void);           /*定时器处理函数*/
};
```

- 第2种是新定时器机制，使用timer_list数据结构的链表，按定时器到期时间的升序排列，expires给出该定时器被激活的时间，而*function ( )指出定时器激活后的处理函数。

```
struct timer_list {
    struct list_head entry;     /*定时器链表*/
    unsigned long expires;      /*定时器激活时间*/
    unsigned long data;         /*传给处理函数的参数*/
    void (*function)(unsigned long);/*定时器处理函数*/
};
```

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

进程定时器

- 进程运行时分用户态和内核态，所以进程执行时也有进程自身在用户模式下花费的执行时间，及内核代表进程在内核模式下花费的执行时间，
- 内核为每个进程累计时间并管理进程三种不同的进程定时器：**ITIMER_REAL**、**ITIMER_VIRTUAL**和**ITIMER_PROF**。
- **ITIMER VIRTUAL**和**ITIMER PROF**定时器的处理方式相同，每一次时钟周期，当前进程的定时器值递减，如果到期，就直接产生适当的定时信号。实现**ITIMER REAL**定时器时，使用系统的**timer**链表，当它到期的时候，是由时钟**bottom half**处理例程把它从队列中删除并调用定时器处理程序，处理例程完成的工作就是产生**SIGALRM**信号。

alarm( )函数

- 如果进程对定时器的要求不是很精确的话，用**alarm( )**函数就可以实现定时器。
- **alarm( )**函数先设置一个时间值（以秒为单位），在将来某个时刻该时间值会被超过，当所设置的时间值被超过后，产生**SIGALRM**信号。如果不忽略或不捕捉此信号，则其默认动作是终止该进程。
- **#include <unistd.h>**
- **unsigned int alarm(unsigned int seconds);**
- 参数**seconds**的值是秒数，经过指定秒后产生信号**SIGALRM**。这种情况下，进程只能有一个定时器时间。如果在调用**alarm( )**时，以前已为该进程设置过定时器，而且它还没有超时，则该定时器时间的余留值作为本次**alarm( )**函数的值返回，以前登记的定时器时间则被新值替换。如果拥有以前登记的尚未超过的定时器时间，而且**seconds**的值为**0**，则取消以前的定时器时间，其余留值仍然作为函数的返回值。

alarm( )函数用法(1)

进程设置了一秒后被激活的定时器，定时器被激活时，程序将打印“alarm!”。

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void sigalrm_fn(int sig)
{
    printf("alarm!\n");
    alarm(2);
    return;
}
int main(void)
{
    signal(SIGALRM, sigalrm_fn);
    alarm(1);
    while(1) pause( );
}
```

利用alarm( )和pause( )函数模拟实现sleep( )函数(1)

```
/*****alarmtest.c*****/
#include <signal.h>
#include <unistd.h>
static void sig_alrm( int sig no )
{
        return ;
 }
unsigned int mysleep( unsigned int nsecs )
{
   if ( signal(SIGALRM, sig_alrm) == SIG_ERR)
                return (nsecs);
        alarm(nsecs);        /*设置定时器*/
        pause( );            /*等待信号*/
        return ( alarm(0) );
}
```

利用alarm( )和pause( )函数模拟实现sleep( )函数(2)

```
– int main( )
– {
–   printf(“sleep\n”);
–   mysleep(10);
–   printf(“wake up\n”);
– }
```

setitimer( )函数(1)

- alarm( )函数实现的定时器精度较低，如果进程需要使用精度较高的定时器，可以通过setitimer( )函数，函数原型为:
  - #include <sys/time.h>
  - int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue));

setitimer( )函数的参数(1)

- 该函数的第1个参数which指定定时器采用的策略，通常可选用下列三者之一，定时器的策略有：
  - ITIMER_REAL：这种定时器使用实时计数，反映进程走过的实际时间，不管进程在何种模式下运行，它总是在计数。当定时到达时，会发给进程一个SIGALRM信号。alarm( )函数实际上是采用ITIMER_REAL策略。
  - ITIMER_VIRTUAL：这种定时器使进程在用户模式（进程本身执行）执行的过程中计数，反映进程走过的虚拟时间，当计数完毕时发送SIGVTALRM信号给进程。
  - ITIMER_PROF：这个定时器是进程在用户模式（进程本身执行）和内核模式（系统代表进程执行）的时候都计时，反映进程处于活跃状态下走过的时间。与ITIMER_VIRTUAL比较，这个定时器记录的时间多于该进程在内核模式执行过程中消耗的时间。当定时到达时，会发送SIGPROF信号。

setitimer( )函数的参数(2)

- 内核为每个进程累计时间并管理进程的不同定时器，调度程序需要使用每个进程有关定时器的值。这些定时器周期性地被初始化为指定值，都进行递减操作，来反映时间的流逝，当定时器为0时，就发出一个中断信号，这时系统或应用程序就会进行相应处理。
- 间隔定时器只能执行一次，但能周期性循环。setitimer( )的第2个参数指向一个itimerval类型的结构，该结构指定定时器初始的持续时间（以秒和微秒为单位）以及定时器被自动重新激活后使用的持续时间（对于一次性执行的定时器而言为0）。setitimer( )的第3个参数是一个指针，它是可选的，指向一个itimerval类型的结构，函数将先前定时器的参数填充到该结构中。

setitimer( )函数的参数(3)

- struct itimerval {
- struct timeval it_interval;
- struct timeval it_value;
- };
- struct timeval {
- long tv_sec;
- long tv_usec;
- };

setitimer( )函数用法(1)

- setitimer（ ）函数调用的一个简单示范，在该例子中，每隔一秒发出一个SIGALRM，每隔0.5秒发出一个SIGVTALRM信号:
  - /*****setitimertest.c*****/
  - #include <stdio.h>
  - #include <stdlib.h>
  - #include <unistd.h>
  - #include <signal.h>
  - #include <time.h>
  - #include <sys/time.h>

setitimer( )函数用法(2)

```
void sigroutine(int signo)
{
    switch (signo){
        case SIGALRM:
            printf("Catch a signal -- SIGALRM \n");
            signal(SIGALRM, sigroutine);
            break;
        case SIGVTALRM:
            printf("Catch a signal -- SIGVTALRM \n");
            signal(SIGVTALRM, sigroutine);
            break;
    }
    return;
}
```

setitimer( )函数用法(3)

```
int main( )
{
    struct itimerval value, ovalue, value2;
    signal(SIGALRM, sigroutine);     /*设置信号处理函数*/
    signal(SIGVTALRM, sigroutine);   /*设置信号处理函数*/
    value.it_value.tv_sec = 1;                  /*设置定时器时间*/
    value.it_value.tv_usec = 0;
    value.it_interval.tv_sec = 1;
    value.it_interval.tv_usec = 0;
    setitimer(ITIMER_REAL, &value, &ovalue);
    value2.it_value.tv_sec = 0;      /*设置定时器时间*/
    value2.it_value.tv_usec = 500000;
    value2.it_interval.tv_sec = 0;
    value2.it_interval.tv_usec = 500000;
    setitimer(ITIMER_VIRTUAL, &value2, &ovalue);
    for( ; ; );
}
```

系统定时器与进程定时器的关系

POSIX 1003.1b标准定时器

- Linux支持POSIX 1003.1b标准定时器，该定时器为用户态程序引入一种新型定时器，尤其是针对多线程和实时应用程序。POSIX定时器比传统间隔定时器更灵活、更可靠。它们之间有两个显著区别:
  - 当传统间隔定时器到期时，内核会发送一个SIGALRM信号给进程来激活定时器。而当一个POSIX定时器到期时，内核可以发送各种信号给整个多线程应用程序，也可以发送给单个指定的线程。
  - 如果一个传统间隔定时器到期了很多次，但用户态进程不能接收SIGALRM信号（例如由于信号被阻塞或者进程不处于运行态），那么只有第1个信号被接收到，其他所有SIGALRM信号都丢失了。对于POSIX定时器来说会发生同样的情况，但进程可以调用timer_getoverrun( )函数来得到自第1个信号产生以来定时器到期的次数。

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

实验1 统计进程时间

- 实验说明

统计进程运行时在用户模式、内核模式以及总的运行时间。需要使用3种定时器来测量进程的处理器使用率，同时使用signal机制建立信号处理程序，记录和统计进程的实际的、虚拟的和活跃的3种时间。

- 解决方案

定义3个计时变量，用于记录进程的3种运行时间，然后定义3种不同的定时器，设定这些定时器的发生时间。在设置完毕后，每当定时器超时并产生一个信号后，只需要增加相应计时变量的值便粗略地得到进程运行的3种时间。

为了得到更加精确的运行时间，需要考虑如下情况：在定时器刚产生一个信号后，程序便运行结束；在定时器信号产生后至程序结束前的时间也需要进行统计。这需要利用设定定时器时使用的itimerval结构。通过setitimer( )设定定时器时，需要传入一个itimerval结构，当再次使用getitimer( )取出该结构时，该结构的it_value成员指明当前至下次定时器超时所需要的时间，通过在进程结束前取出各个定时器的itimerval结构，就能够更精确地统计进程运行的3种时间。

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

实验2 通过alarm( )实现sleep( )函数功能

- 实验说明

定时器一节给出一个通过alarm( )函数实现sleep( )函数的例子。但是该例子还有多个问题需要解决。在本实验中，需要对该方案进行改进，实现一个可靠的类sleep( )函数。

- 解决方案

需设定新的信号处理函数，保存原有信号处理函数；设置信号集，屏蔽SIGALRM信号；处理中如果原先设定的定时器尚未超时，则再次调用alarm( )。

主要内容

- 背景知识
  - 定时器机制的概念
  - 系统定时器
  - 进程定时器
- 实验内容
  - 统计进程时间
  - 通过alarm( )实现sleep( )函数功能
  - 基于单定时器实现任意数目的逻辑定时器

实验3 基于单定时器实现任意数目的逻辑定时器

- 实验说明
  - 在实验2中，读者可以发现，仅仅只能使用一个定时器在某些时候是不够的。为了让用户更加方便的使用定时器，在本实验中，需要基于原有的Linux定时器构造一组新的定时器函数，使得进程可以设置任意数目的定时器，这些定时器的精度以秒为单位即可。

解决方案(1)

- 可以定义一组API与数据结构，供用户使用自定义的定时器，自定义定时器采用进程的总运行时间进行计时:
- **unsigned long add_timer(unsigned long nsecs, timer_handler handler, void *data);** 添加一个定时器，该定时器会在nsecs秒后被激活，handler在定时器激活后会被调用。函数的返回值为自定义定时器的唯一标识符。
- **unsigned long del_timer(int id);**删除一个定时器。函数的返回值为该定时器距离超时的秒数
- **typedef void timer_handler(void *);** 自定义的定时处理函数，会在函数被激活时调用。该处理函数能够在add_timer( )时附带一个参数，当该函数被调用时，该参数会被传递给该函数。

- 解决方案(2)

用户在进程空间中维护一个自定义定时器的链表，在用户使用自定义定时器时：

- 先设定一个系统定时器，定时器每隔一秒被激活
- 为每一个自定义定时器维护一个数据结构
- 系统定时器被激活时，更新每一个自定义定时器的计数，如果发现一个自定义定时器超时，便执行其关联的定时处理函数。