<https://www.cnblogs.com/a-lai/p/7298463.html>

为什么需要这个功能，因为大多数计算机软件时钟系统通常只能有一个时钟触发一次中断。当运行多个任务时，我们会想要多个定时器 的时钟跟踪并发这样可以生成正确的时间重叠，操作系统这样做。

本例子是为了实现使用Linux下的一个定时器，实现任一数量的定时器功能。

首先我们需要一些数据类型用来描述时钟数据结构

[复制代码](javascript:void(0);)

#include <stdio.h>

#include<time.h>

#define TRUE 1

#define FALSE 0

#define MAX\_TIMERS ... 最大时钟数量

typedef timerval TIME; 定义时间类型

#define VERY\_LONG\_TIME ... 最大时间长度

struct timer {

int inuse; 时钟是否可用

TIME time; 定时时间长度

char \*event; 是否超时

} timers[MAX\_TIMERS]; /\* set of timers \*/

[复制代码](javascript:void(0);)

每个定时器都以这个数据结构来描述，第一个成员用来描述时钟是否正在使用，第二个成员是这个定时器的定时时间，第三个成员是是一个指针，\*event初始化应该为0，当他被置为1，我们知道这个定时器已经超时了，和他相关的任务可以执行。

接下来是定时器数组的初始化，这里将每个时钟inuse成员设置为FALSE,表示时钟不可用。

[复制代码](javascript:void(0);)

void

timers\_init() {

struct timer \*t;

for (t=timers;t<&timers[MAX\_TIMERS];t++)

t->inuse = FALSE;

}

[复制代码](javascript:void(0);)

现在开始是结构实现部分

首先写到的timer\_undeclare这个函数，这个函数与后面的timer\_declare相对立。主要作用是清除一个定时器。

有很多方法可以用来保存定时器的定时记录。没有复杂时钟硬件的机器通常在每一个时钟周期处理一个中断处理程序。然后软件就在处理程序中获取系统时间，然后判断是否设置的定时器超时。

很多比较聪明的机器可以在硬件中设置定时时间，一旦时间超时，就触发一个硬件中断。这同样适用与软件中断。

他们通过一个 定义一个time\_now来记录当前的系统时间，volatile告诉机器每次从寄存器取值，防止数据被系统优化。

volatile TIME time\_now

接下来定义一系列数据来记录 timer\_next 指接下来要我们想要计时的定时器。time\_timer\_set保存最后一次获取的系统时间。

[复制代码](javascript:void(0);)

struct timer \*timer\_next = NULL;/\* timer we expect to run down next \*/

TIME time\_timer\_set; /\* time when physical timer was set \*/

//取消一个定时器

void timer\_undeclare(struct timer \*t)

{

　　disable\_interrupts();

　　if(!t->inuse)

　　{

　　　　enable\_interrupts();

　　　　return ;

　　}

　　t->inuse=0;

　　if(t==timer\_next)

　　{

　　　　if(time(&time\_now)<0)

　　　　perror("time error");

　　　　timers\_update(time\_now-time\_timer\_set);

　　　　if(timer\_next)

　　　　{

　　　　　　start\_physical\_timer(&timer\_next->time);

　　　　　　time\_timer\_set=time\_now;

　　　　}

　　}

　　enable\_interrupts();

}

[复制代码](javascript:void(0);)

timer\_undeclare作用为取消一个定时器。首先让中断失效，这很重要，因为时钟数据结构数据是在各进程中共享的，是可以在其他中断中被修改的，为了防止不必要的错我，这个取消操作应该为一个原子操作。开始我们先检测是否这个时钟已经无效了。如果有效，则设置inuse使其失效。如果我们要取消的定时器正好是下一个期望等待的定时器。那我们要重新指定下一个期望等待的定时器。在此之前所有定时器都要更新一下前一个定时器已经走过的时间。

接下来我们看到timers\_update（time\_t ti）函数

[复制代码](javascript:void(0);)

//更新定时器表时间

void timers\_update(time\_t time)

{

　　static struct timer timer\_last={

　　0,

　　{},

　　NULL

　　};

　　timer\_last.time.tv\_sec=10;

　　struct timer \*t;

　　timer\_next=&timer\_last;

　　for(t=timers;t<&timers[MAX\_TIMERS];t++)

　　{

　　　　if(t->inuse)

　　　　{

　　　　　　if(time<t->time.tv\_sec){

　　　　　　t->time.tv\_sec-=time;

　　　　　　if(t->time.tv\_sec<\

　　　　　　timer\_next->time.tv\_sec)

　　　　　　　　timer\_next=t;

　　　　　　}else

　　　　　　{

　　　　　　　　\*(t->event)=1;

　　　　　　　　t->inuse=0;

　　　　　　}

　　　　}

　　}

　　if(!timer\_next->inuse)timer\_next=0;

}

[复制代码](javascript:void(0);)

此函数作用是更新所有有效定时器的时间长，同时将timer\_next指向当前延时时间最短的一个定时器。如没有定时器，则将timer\_next设置为空。

timer\_declare 加入一个定时器

[复制代码](javascript:void(0);)

struct timer \* timer\_declare(TIME \*ti,char \*event)

{

　　struct timer \*t;

　　disable\_interrupts();

　　for(t=timers;t<&timers[MAX\_TIMERS];t++)

　　{

　　　　if(!t->inuse)break;

　　}

　　if(t==&timers[MAX\_TIMERS])

　　{

　　　　enable\_interrupts();

　　　　return 0;

　　}

　　t->event=event;

　　t->time.tv\_sec=ti->tv\_sec;

　　t->time.tv\_usec=ti->tv\_usec;

　　if(!timer\_next)

　　{

　　　　if(time(&time\_now)<0)

　　　　perror("time() error");

　　　　time\_timer\_set=time\_now;

　　　　start\_physical\_timer(&((timer\_next=t)->time));

　　}else if((ti->tv\_sec+time\_now)<(\

　　timer\_next->time.tv\_sec+time\_timer\_set))

　　{

　　　　if(time(&time\_now)<0)

　　　　perror("time error");

　　　　timers\_update(time\_now-time\_timer\_set);

　　　　time\_timer\_set=time\_now;

　　　　start\_physical\_timer(&((timer\_next=t)->time));

　　}else

　　{

　　}

　　t->inuse=1;

　　enable\_interrupts();

　　return t;

}

[复制代码](javascript:void(0);)

首先找到一个可用的定时器表项，设置相关参数。

接下来判断如果timer\_next为空，那么说明定时器表项没有定时器需要定时，那我们直接将timer\_next指向新加入定时器，开始计时。

如果新加入定时器需要延时时间比当前正在延时的定时器的剩余时间还要短，则更新定时器表，并计时当前加入的定时器。

在处理完当前定时器事件后，将新加入的定时器的inuse置1.

接下来是定时器中断处理函数

[复制代码](javascript:void(0);)

//定时器中断处理函数

void timer\_interrupt\_hander(int signo)

{

　　printf("interrupt\_hander\n");

　　if(time(&time\_now)<0)

　　　　perror("time() error");

　　timers\_update(time\_now-time\_timer\_set);

　　if(timer\_next)

　　{

　　　　time\_timer\_set=time\_now;

　　　　start\_physical\_timer(&timer\_next->time);

　　}

}

[复制代码](javascript:void(0);)

这里我们打印一串字符来证明定时器时间的触发，首先要做的先更新定时器表，然后将time\_timer\_set设置成当前系统时间，继续进行下一个定时器事件，直到所有定时器都处理完毕。

接下来几个是LINUX系统相关函数

[复制代码](javascript:void(0);)

//失效定时器中断

void disable\_interrupts()

{

sigset\_t new\_mask;

sigemptyset(&new\_mask);

sigaddset(&new\_mask,SIGALRM);

if(sigprocmask(SIG\_BLOCK,&new\_mask,NULL)<0)

perror("SIG\_BLOCK error");

}

//使能定时器中断

void enable\_interrupts()

{

sigset\_t new\_mask;

sigemptyset(&new\_mask);

sigaddset(&new\_mask,SIGALRM);

if(sigprocmask(SIG\_UNBLOCK,&new\_mask,NULL)<0)

perror("SIG\_UNBLOCK error");

}

//开启一个定时器工作

void start\_physical\_timer(TIME\* time)

{

if(signal(SIGALRM,timer\_interrupt\_hander)==SIG\_ERR)

perror("signal error");

struct itimerval new\_value;

sigset\_t zero\_mask;

sigemptyset(&zero\_mask);

new\_value.it\_value.tv\_sec=time->tv\_sec;

new\_value.it\_value.tv\_usec=time->tv\_usec;

new\_value.it\_interval.tv\_sec=0;

new\_value.it\_interval.tv\_usec=0;

setitimer(ITIMER\_REAL,&new\_value,NULL);

sigsuspend(&zero\_mask);

}

[复制代码](javascript:void(0);)

主函数测试部分

[复制代码](javascript:void(0);)

#include<stdio.h>

#include<signal.h>

#include"multtime.h"

#include<stdlib.h>

#include<unistd.h>

int main()

{

pid\_t pid;

TIME time1,time2,time3;

time1.tv\_sec=6;

time1.tv\_usec=0;

time2.tv\_sec=4;

time2.tv\_usec=0;

time3.tv\_sec=2;

time3.tv\_usec=0;

timer\_init();

if((pid=fork())<0)

{

perror("fork() error");

}

else if(pid==0)

{

printf("child 1\n");

timer\_undeclare(timer\_declare(&time1,0));

}

else

{

if((pid=fork())<0)

{

perror("fork error");

}

else if(pid==0)

{

printf("child 2\n");

timer\_undeclare(timer\_declare(&time3,0));

}

else

{

printf("parent\n");

timer\_undeclare(timer\_declare(&time2,0));

}

}

exit(0);

}

[复制代码](javascript:void(0);)

实验结果：

parent

child 2

child 1

interrupt\_hander

interrupt\_hander

interrupt\_hander

本人初学LINUX是个菜鸟，以自己的理解写的，待深入学习后再完善改进。