<https://blog.csdn.net/qq_33336155/article/details/52037726>

一、已有的定时器接口  
   时空管理是计算机系统的主要任务。在时间管理中，我们经常利用定时器处理事情：比如tcp协议中利用定时器管理包超时，视频显示中利用定时器来定时显示视频帧，web服务中利用定时器来管理用户的超时。windows系统提供了SetTimer和timeSetEvent等定时器接口，linux中则提供了setitimer等接口。这些函数的接口很类似，大体上都是用户提供回调函数和超时时间向OS注册一个定时器事件，OS在超时时间到了的时候，调用用户提供的回调函数来完成用户想要做的事情。windows下的接口支持单进程中拥有多个定时器，而linux则只允许单进程拥有一个定时器，因此在linux下的单进程中要使用多个定时器，则需要自己维护管理，这是本文写作的出发点。另外，OS提供的定时器管理算法在大规模定时器的管理方面可能还不尽人意，这时候就需要用户去优化管理算法了，本文在这方面提供了一点素材。  
  
二、一个最简单的多定时器的实现（linux版）  
1、实现细节   
  这个实现允许用户使用多个自定义的定时器，每个自定义的定时器将周期地被触发直到其被删除。实现的主要思路是：  
    i）首先在初始化多定时器（init\_mul\_timer）时利用setitimer注册一个基本的时间单位（如1s）的定时事件；  
    ii）用户需要set\_a\_timer注册自定义定时器时，在timer\_manage管理结构中记录这个定时器的回调函数和定时周期等参数；  
    iii）当基本的时间单位到期后（如SIGALRM信号到达时），遍历整个timer\_manage，如果有自定义定时器的超时时间到了，就执行相应的回调函数，并将自定义定时器的超时时间置为最初值；否则将自定义定时器的超时时间相应地减一个基本的时间单位；  
    iv）用户通过del\_a\_timer来删除某个定时器，通 过destroy\_mul\_timer来删除整个多定时器。  
2、代码  
i) mul\_timer.h

|  |
| --- |
| /\* This file provides an interface of multiple timers. for convenience, it simplify signal processing.  \* Also, it can't be used in multithreading environment.  \* Author:bripengandre  \* Date:2009-04-29  \*/ #ifndef \_MUL\_TIMER\_H\_ #define \_MUL\_TIMER\_H\_  #include <sys/time.h>  #define MAX\_TIMER\_CNT 10 #define MUL\_TIMER\_RESET\_SEC 10 #define TIMER\_UNIT 60 #define MAX\_FUNC\_ARG\_LEN 100 #define INVALID\_TIMER\_HANDLE (-1)  typedef int timer\_handle\_t;  typedef struct \_timer\_manage {     struct \_timer\_info     {         int state; /\* on or off \*/         int interval;         int elapse; /\* 0~interval \*/         int (\* timer\_proc) (void \*arg, int arg\_len);         char func\_arg[MAX\_FUNC\_ARG\_LEN];         int arg\_len;     }timer\_info[MAX\_TIMER\_CNT];      void (\* old\_sigfunc)(int);     void (\* new\_sigfunc)(int);     struct itimerval value, ovalue; }\_timer\_manage\_t;  /\* success, return 0; failed, return -1 \*/ int init\_mul\_timer(void); /\* success, return 0; failed, return -1 \*/ int destroy\_mul\_timer(void); /\* success, return timer handle(>=0); failed, return -1 \*/ timer\_handle\_t set\_a\_timer(int interval, int (\* timer\_proc) (void \*arg, int arg\_len), void\*arg, int arg\_len); /\* success, return 0; failed, return -1 \*/ int del\_a\_timer(timer\_handle\_t handle);  #endif /\* \_MUL\_TIMER\_H\_ \*/ |

ii)mul\_timer.c

|  |
| --- |
| #include <stdio.h> #include <string.h> #include <signal.h> #include <time.h>  #include "mul\_timer.h"  static struct \_timer\_manage timer\_manage;  static void sig\_func(int signo);  /\* success, return 0; failed, return -1 \*/ int init\_mul\_timer(void) {     int ret;          memset(&timer\_manage, 0, sizeof(struct \_timer\_manage));     if( (timer\_manage.old\_sigfunc = signal(SIGALRM, sig\_func)) == SIG\_ERR)     {         return (-1);     }     timer\_manage.new\_sigfunc = sig\_func;          timer\_manage.value.it\_value.tv\_sec = MUL\_TIMER\_RESET\_SEC;     timer\_manage.value.it\_value.tv\_usec = 0;     timer\_manage.value.it\_interval.tv\_sec = TIMER\_UNIT;     timer\_manage.value.it\_interval.tv\_usec = 0;     ret = setitimer(ITIMER\_REAL, &timer\_manage.value, &timer\_manage.ovalue);           return (ret); }   /\* success, return 0; failed, return -1 \*/ int destroy\_mul\_timer(void) {     int ret;          if( (signal(SIGALRM, timer\_manage.old\_sigfunc)) == SIG\_ERR)     {         return (-1);      }      ret = setitimer(ITIMER\_REAL, &timer\_manage.ovalue, &timer\_manage.value);     if(ret < 0)     {         return (-1);     }      memset(&timer\_manage, 0, sizeof(struct \_timer\_manage));          return(0); }   /\* success, return timer handle(>=0); failed, return -1 \*/ timer\_handle\_t set\_a\_timer(int interval, int (\* timer\_proc) (void \*arg, int arg\_len), void \*arg,int arg\_len) {     int i;          if(timer\_proc == NULL || interval <= 0)     {         return (-1);     }           for(i = 0; i < MAX\_TIMER\_CNT; i++)     {         if(timer\_manage.timer\_info[i].state == 1)         {             continue;         }                  memset(&timer\_manage.timer\_info[i], 0, sizeof(timer\_manage.timer\_info[i]));         timer\_manage.timer\_info[i].timer\_proc = timer\_proc;         if(arg != NULL)         {             if(arg\_len > MAX\_FUNC\_ARG\_LEN)             {                 return (-1);             }             memcpy(timer\_manage.timer\_info[i].func\_arg, arg, arg\_len);             timer\_manage.timer\_info[i].arg\_len = arg\_len;         }         timer\_manage.timer\_info[i].interval = interval;         timer\_manage.timer\_info[i].elapse = 0;         timer\_manage.timer\_info[i].state = 1;         break;     }          if(i >= MAX\_TIMER\_CNT)     {         return (-1);     }     return (i); }   /\* success, return 0; failed, return -1 \*/ int del\_a\_timer(timer\_handle\_t handle) {     if(handle < 0 || handle >= MAX\_TIMER\_CNT)     {         return (-1);     }          memset(&timer\_manage.timer\_info[handle], 0, sizeof(timer\_manage.timer\_info[handle]));          return (0); }   static void sig\_func(int signo) {     int i;     for(i = 0; i < MAX\_TIMER\_CNT; i++)     {         if(timer\_manage.timer\_info[i].state == 0)         {             continue;         }         timer\_manage.timer\_info[i].elapse++;         if(timer\_manage.timer\_info[i].elapse == timer\_manage.timer\_info[i].interval)         {             timer\_manage.timer\_info[i].elapse = 0;             timer\_manage.timer\_info[i].timer\_proc(timer\_manage.timer\_info[i].func\_arg,timer\_manage.timer\_info[i].arg\_len);         }     } }   #define \_MUL\_TIMER\_MAIN   #ifdef \_MUL\_TIMER\_MAIN  static void get\_format\_time(char \*tstr) {     time\_t t;          t = time(NULL);     strcpy(tstr, ctime(&t));     tstr[strlen(tstr)-1] = '/0';          return; }   timer\_handle\_t hdl[3], call\_cnt = 0; int timer\_proc1(void \*arg, int len) {     char tstr[200];     static int i, ret;          get\_format\_time(tstr);     printf("hello %s: timer\_proc1 is here./n", tstr);     if(i >= 5)     {         get\_format\_time(tstr);         ret = del\_a\_timer(hdl[0]);         printf("timer\_proc1: %s del\_a\_timer::ret=%d/n", tstr, ret);     }     i++;     call\_cnt++;               return (1); }  int timer\_proc2(void \* arg, int len) {     char tstr[200];     static int i, ret;          get\_format\_time(tstr);     printf("hello %s: timer\_proc2 is here./n", tstr);     if(i >= 5)     {         get\_format\_time(tstr);         ret = del\_a\_timer(hdl[2]);         printf("timer\_proc2: %s del\_a\_timer::ret=%d/n", tstr, ret);     }     i++;     call\_cnt++;          return (1); }   int main(void) {     char arg[50];     char tstr[200];     int ret;          init\_mul\_timer();     hdl[0] = set\_a\_timer(2, timer\_proc1, NULL, 0);     printf("hdl[0]=%d/n", hdl[0]);     hdl[1] = set\_a\_timer(3, timer\_proc2, arg, 50);     printf("hdl[1]=%d/n", hdl[1]);     hdl[2] = set\_a\_timer(3, timer\_proc2, arg, 101);     printf("hdl[1]=%d/n", hdl[2]);     while(1)     {         if(call\_cnt >= 12)         {             get\_format\_time(tstr);             ret = destroy\_mul\_timer();             printf("main: %s destroy\_mul\_timer, ret=%d/n", tstr, ret);             call\_cnt++;         }         if(call\_cnt >= 20)         {             break;         }     }          return 0; }  #endif |

3、缺陷  
   i)新建定时器、遍历定时器和删除定时器（查找哪个定时器超时）时时间复杂度都为O(n)（n是定时器的个数）;  
   ii)适用环境是单线程环境，如要用于多线程，需添加同步操作。  
   iii)程序中有些小bug，如对新建超时时间为0的定时器没有妥善的处理。  
  
三、多定时器的改进版  
1、思路  
   改进定时器的实现，即是改善二种所指出的几个缺陷，如下是一个改进版,主要是将遍历超时时间的时间复杂度降为了O(1).  
   改善思路:各定时器以一个链表的形式组织起来,除链表头定时器的超时时间是用绝对时间纪录的外,其它定时器的超时时间均用相对时间(即超时时间-前一个定时器的超时时间)纪录.  
   注意,各定时器都是一次性的,当定时器的超时被处理后,定时器将被自动删除.另外如果将定时器的结点改为双向结构,可以将删除定时器的时间复杂度降为O(1).  
2、数据结构  
   每个定时器都有一个唯一的ID,这个ID是如下的结构体:

|  |
| --- |
| typedef struct \_timer\_handle {         unsigned long ptr;         unsigned long entry\_id; }\*timer\_handle\_t; |

   ptr纪录的是定时器结点的地址,entry\_id则是一个自多定时器初始化后自增的id.ptr和entry\_id一起组成定时器结点的key,一方面使得新建定时器时生成key的过程大为简化,另一方面使得删除定时器的时间复杂度降为O(1)（前提是定时器结点采用双向结构）。  
   定时器结点的数据结构如下：

|  |
| --- |
| /\* timer entry \*/ typedef struct \_mul\_timer\_entry {     char is\_use; /\* 0, not; 1, yes \*/     struct \_timer\_handle handle;     unsigned int timeout;     unsigned int elapse; /\* \*/     int (\* timer\_proc) (void \*arg, unsigned int \*arg\_len); /\* callback function \*/     void \*arg;     unsigned int \*arg\_len;     struct \_mul\_timer\_entry \*etr\_next; }mul\_timer\_entry\_t; |

其中的is\_use是用来防止这样一种情况：用户在回调函数中调用kill\_timer来删除定时器，这个时候kill\_timer和遍历定时器中都有删除结点的操作，有可能将整个链表搞混乱。所以在调用用户的回调函数前先将is\_use置1，在kill\_timer中需检查is\_use，只有在 is\_use为0的情况下，才执行清理定时器结点的操作。  
3、代码（windows版）  
i)mul\_timer.h

|  |
| --- |
| /\* This file provides an interface of multiple timers. it can't be used in multithreading environment.  \* Author:bripengandre  \* Date:2009-07-19  \*/  #ifndef \_MUL\_TIMER\_H\_ #define \_MUL\_TIMER\_H\_ #include <windows.h>  typedef struct \_timer\_handle {         unsigned long ptr;         unsigned long entry\_id; }\*timer\_handle\_t;  /\* timer entry \*/ typedef struct \_mul\_timer\_entry {     char is\_use; /\* 0, not; 1, yes \*/     struct \_timer\_handle handle;     unsigned int timeout;     unsigned int elapse; /\* \*/     int (\* timer\_proc) (void \*arg, unsigned int \*arg\_len); /\* callback function \*/     void \*arg;     unsigned int \*arg\_len;     struct \_mul\_timer\_entry \*etr\_next; }mul\_timer\_entry\_t;  typedef struct \_mul\_timer\_manage {     unsigned long entry\_id;     unsigned int timer\_cnt;     unsigned int time\_unit;     struct \_mul\_timer\_entry \*etr\_head;     UINT timer\_id; };    struct \_mul\_timer\_manage \*init\_mul\_timer(unsigned int time\_unit); timer\_handle\_t set\_timer(struct \_mul\_timer\_manage \*ptimer, unsigned int time\_out, int (\*timer\_proc) (void \*arg, unsigned int \*arg\_len), void \*arg, unsigned int \*arg\_len); int kill\_timer(struct \_mul\_timer\_manage \*ptimer, timer\_handle\_t hdl); int get\_timeout\_byhdl(struct \_mul\_timer\_manage \*ptimer, timer\_handle\_t hdl); int get\_timeout\_bytimeproc(struct \_mul\_timer\_manage \*ptimer, int (\* timer\_proc) (void\*arg, unsigned int \*arg\_len)); int release\_mul\_timer(struct \_mul\_timer\_manage \*ptimer);  int is\_valid\_time\_hdl(timer\_handle\_t hdl);  #endif /\* \_MUL\_TIMER\_H\_ \*/ |

ii)mul\_timer.c

|  |
| --- |
| #include "mul\_timer.h" #include <stdio.h> #include <stdlib.h> #include <time.h>   void CALLBACK traverse\_mul\_timer(UINT uTimerID, UINT uMsg, DWORD dwUser, DWORD dw1, DWORD dw2); static int print\_mul\_timer(struct \_mul\_timer\_manage \*ptimer);  struct \_mul\_timer\_manage \*init\_mul\_timer(unsigned int time\_unit) {     struct \_mul\_timer\_manage \*p;          if( (p = malloc(sizeof(struct \_mul\_timer\_manage))) == NULL)     {         return (NULL);     }          p->etr\_head = NULL;     p->timer\_cnt = 0;     p->time\_unit = time\_unit;     p->entry\_id = 0;          p->timer\_id = timeSetEvent(time\_unit, 0, (LPTIMECALLBACK )traverse\_mul\_timer,(DWORD)p, TIME\_PERIODIC);          return(p); }   timer\_handle\_t set\_timer(struct \_mul\_timer\_manage \*ptimer, unsigned int time\_out, int (\*timer\_proc) (void \*arg, unsigned int \*arg\_len), void \*arg, unsigned int \*arg\_len) {     struct \_mul\_timer\_entry \*p, \*prev, \*pnew;          if(ptimer == NULL || time\_out == 0)     {         return (NULL);     }               if( (pnew = malloc(sizeof(struct \_mul\_timer\_entry))) == NULL)     {         return (NULL);     }     pnew->is\_use = 0;     pnew->arg = arg;     pnew->arg\_len = arg\_len;     pnew->elapse = 0;     pnew->timer\_proc = timer\_proc;          p = ptimer->etr\_head;     prev = NULL;     while(p != NULL)     {         if(p->timeout < time\_out) /\* assume the latest time\_proc has higher priority \*/         {             time\_out = time\_out-p->timeout;             prev = p;             p = p->etr\_next;         }         else         {             p->timeout -= time\_out;             break;         }     }          pnew->timeout = time\_out;     pnew->etr\_next = p;     pnew->handle.ptr = (unsigned long )pnew;     pnew->handle.entry\_id = ptimer->entry\_id;     ptimer->entry\_id++;      if(prev == NULL)     {         ptimer->etr\_head = pnew;     }     else     {         prev->etr\_next = pnew;     }         ptimer->timer\_cnt++;          return (&pnew->handle); }   int kill\_timer(struct \_mul\_timer\_manage \*ptimer, timer\_handle\_t hdl) {     struct \_mul\_timer\_entry \*p, \*prev;          if(ptimer == NULL)     {         return (0);     }               p = ptimer->etr\_head;     prev = NULL;     while(p != NULL)     {         if(p->handle.ptr == hdl->ptr && p->handle.entry\_id == hdl->entry\_id)         {             break;         }          prev = p;         p = p->etr\_next;     }          /\* no such timer or timer is in use, return 0 \*/     if(p == NULL || (p != NULL && p->is\_use == 1))      {         return (0);      }          /\* has found the timer \*/     if(prev == NULL)     {         ptimer->etr\_head = p->etr\_next;     }     else     {         prev->etr\_next = p->etr\_next;     }          /\* revise timeout \*/     if(p->etr\_next != NULL)     {         p->etr\_next->timeout += p->timeout;     }          /\* delete the timer \*/     free(p);     p = NULL;     ptimer->timer\_cnt--;          return (1); }   int get\_timeout\_byhdl(struct \_mul\_timer\_manage \*ptimer, timer\_handle\_t hdl) {     struct \_mul\_timer\_entry \*p;     unsigned int timeout;          if(ptimer == NULL || (struct \_mul\_timer\_entry \*)(hdl) == NULL)     {         return (-1);     }               timeout = 0;     p = ptimer->etr\_head;     while(p != NULL)     {      if(p->handle.ptr == hdl->ptr && p->handle.entry\_id == hdl->entry\_id)         {             break;         }          timeout += p->timeout;         p = p->etr\_next;     }          if(p == NULL)     {         return (-1);     }     else     {         return ((int)timeout+p->timeout);     }          }   int get\_timeout\_bytimeproc(struct \_mul\_timer\_manage \*ptimer, int (\* timer\_proc) (void\*arg, unsigned int \*arg\_len)) {     struct \_mul\_timer\_entry \*p;     unsigned int timeout;          if(ptimer == NULL || timer\_proc == NULL)     {         return (-1);     }          p = ptimer->etr\_head;     while((p != NULL) && (p->timer\_proc != timer\_proc))     {         timeout += p->timeout;         p = p->etr\_next;     }          if(p == NULL)     {         return (-1);     }     else     {         return (timeout+p->timeout);     }     }   int release\_mul\_timer(struct \_mul\_timer\_manage \*ptimer) {     struct \_mul\_timer\_entry \*p, \*ptmp;          if(ptimer == NULL)     {         return (0);     }          timeKillEvent(ptimer->timer\_id);     /\* delete all timers \*/     p = ptimer->etr\_head;     while(p != NULL)     {         ptmp = p;         p = p->etr\_next;         free(ptmp);     }     /\* delete timer\_manage \*/     free(ptimer);     ptimer = NULL;          return (1); }  int is\_valid\_time\_hdl(timer\_handle\_t hdl) {     if(hdl == NULL)     {         return (0);     }     else     {         return (1);     }     }   void CALLBACK traverse\_mul\_timer(UINT uTimerID, UINT uMsg, DWORD dwUser, DWORD dw1, DWORD dw2) {     struct \_mul\_timer\_manage \*ptimer;     struct \_mul\_timer\_entry \*p, \*ptmp;     unsigned int timeout;          ptimer = (struct \_mul\_timer\_manage \*)dwUser;     if(ptimer == NULL)     {         return;     }          timeout = ptimer->time\_unit;     p = ptimer->etr\_head;     while(p != NULL)     {         if(p->timeout <= timeout)         {             p->is\_use = 1;             p->timer\_proc(p->arg, p->arg\_len);             ptmp = p;             timeout -= p->timeout;             p = p->etr\_next;             free(ptmp);             ptimer->etr\_head = p;         }         else         {             p->timeout -= timeout;             p->elapse += timeout;             ptimer->etr\_head = p;             break;         }     }     if(p == NULL)     {         ptimer->etr\_head = NULL;     }              return;  }   static int print\_mul\_timer(struct \_mul\_timer\_manage \*ptimer) {     struct \_mul\_timer\_entry \*p;     int i;          if(ptimer == NULL)     {         return (0);     }          printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*mul\_timer statistics start\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/n");     printf("this mul\_timer's time\_unit=%u, etr\_head=%p and has %d timers:/n", ptimer->time\_unit, ptimer->etr\_head, ptimer->timer\_cnt);          p = ptimer->etr\_head;     i = 0;     while(p != NULL)     {         printf("the %d timer: timeout=%u, elapse=%u, timer\_proc=%p, arg=%p, arg\_len=%p, etr\_next=%p/n"             , i+1, p->timeout, p->elapse, p->timer\_proc, p->arg, p->arg\_len,p->etr\_next);         p = p->etr\_next;         i++;     }     printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*mul\_timer statistics end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/n");          return (1); }   #define \_MUL\_TIMER\_MAIN  #ifdef \_MUL\_TIMER\_MAIN  static void get\_format\_time(char \*tstr) {     time\_t t;          t = time(NULL);     strcpy(tstr, ctime(&t));     tstr[strlen(tstr)-1] = '/0';          return; }  timer\_handle\_t hdl[100]; int call\_cnt = 0; struct \_mul\_timer\_manage \*ptimer;  int timer\_proc1(void \*arg, unsigned int \*len) {     char tstr[200];     static int i, ret;          get\_format\_time(tstr);     printf("call\_cnt=%d, hello %s: timer\_proc1 is here./n", call\_cnt, tstr);     i++;     call\_cnt++;               return (1); }  int timer\_proc2(void \* arg, unsigned int \*len) {     char tstr[200];     static int i, ret;          get\_format\_time(tstr);     printf("call\_cnt=%d, hello %s: timer\_proc2 is here: arg = %s, len = %d./n",call\_cnt, tstr, arg, \*len);     i++;     call\_cnt++;          return (1); }   int main(void) {     char arg[50] = "hello, multiple timers";     char tstr[200];     int ret;     int len = 50, i;          ptimer = init\_mul\_timer(1000);          for(i = 0; i < 10; i++)     {         hdl[i<<1] = set\_timer(ptimer, 1000\*(i+1), timer\_proc1, NULL, NULL);         printf("hdl[0i<<1=%d, is\_valid\_hdl=%d/n", hdl[i<<1],is\_valid\_time\_hdl(hdl[i<<1]));         hdl[(i<<1)+1] = set\_timer(ptimer, 3000\*(i+1), timer\_proc2, arg, &len);         printf("hdl[i<<1+1]=%d, is\_valid\_hdl=%d/n", hdl[(i<<1)+1],is\_valid\_time\_hdl(hdl[(i<<1)+1]));         print\_mul\_timer(ptimer);     }      ret = kill\_timer(ptimer, hdl[17]);     printf("ret=kill\_timer=%d/n", ret);     print\_mul\_timer(ptimer);         printf("hd[19]->timout=%d/n", get\_timeout\_byhdl(ptimer, hdl[19]));      while(1)     {         if(call\_cnt == 15)         {             get\_format\_time(tstr);             ret = release\_mul\_timer(ptimer);             printf("call\_cnt=%d, main: %s destroy\_mul\_timer, ret=%d/n",call\_cnt, tstr, ret);             call\_cnt++;         }     }          return 0; }  #endif |

3、缺陷  
i)新建定时器的时间复杂度为O(n),删除定时器的时间复杂度也为O(n)(简单地将定时器结点改为双向结构,可将复杂度降为O(1));  
ii)不能用于多线程环境  
四 、多定时器的工业级实现  
 1、time wheelz算法  
   以前的BSD内核以及现在的linux内核的实现与三中所用算法相似（未实证，只是据说），据说现在的BSD内核已采用了较好的time wheelz算法。  
   time wheez算法的优点：  
   i)将新建定时器的时间复杂度降近似为O(1)。它根据定时器的超时值，将新定时器散列到hash桶中；  
   ii)遍历检查定时器的时间复杂度也近似为O(桶大小)，如果散列均匀。  
   iii)删除定时器的时间复杂度近似为O(1)，通过hash算法或临时存储（空间换时间的算法）。  
2、time wheelz的实现

   请参考文末给出的两个论文，惭愧得很，文章我也只是稍微瞄了下，以后有用得着的时候再深究吧。

转自 http://blog.csdn.net/zhangxinrun/article/details/5914191