Timers and Time Service

5

Abstract

This chapter covers timers and timer services. It explains the principle of hardware timers and the hardware timers in Intel x86 based PCs. it covers CPU operations and interrupts processing. It describes timer related system calls, library functions and commands for timer services in Linux. It discusses process interval timers, timer generated signals and demonstrates process interval timers by examples. The programming project is to implement timer, timer interrupts and interval timers in a multitasking system. The multitasking system runs as a Linux process, which acts as a virtual CPU for concurrent tasks inside the Linux process. The real-time mode interval timer of the Linux process is programmed to generate SIGALRM signals periodically, which acts as timer interrupts to the virtual CPU, which uses a SIGALRM signal catcher as the timer interrupt handler. The project is for the reader to implement interval timers for tasks by a timer queue. It also lets the reader use Linux signal masks to implement critical regions to prevent race conditions between tasks and interrupt handlers.

5.1 Hardware Timer

A timer is a hardware device comprising a clock source and a programmable counter. The clock source is usually a crystal oscillator, which generates periodic electrical signals to drive the counter at a precise frequency. The counter is programmed with a count-down value, which decrements by 1 on each clock signal. When the count decrements to 0, the counter generates a **timer interrupt** to the CPU, reloads the count value into the counter and repeats the count-down again. The period of the counter is known as a timer tick, which is the basic timing unit of the system.

5.2 PC Timers

The Intel x86 based PC has several timers (Bovet and Cesati 2005).

(1). Real-Time Clock (RTC): The RTC is powered by a small backup battery. It runs continually even when the PC's power is turned off. It is used to keep real time to provide time and date information. When Linux boots up, it uses the RTC to update a system time variable to keep track of the current

© Springer International Publishing AG, part of Springer Nature 2018

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_5

time. In all Unix-like systems, the time variable is a long integer containing the number of seconds elapsed since the beginning of January 1 of 1970.

(2). Programmable Interval Timer (PIT) (Wang 2015]**:** The PIT is a hardware timer separated from the CPU. It can be programmed to provide timer ticks in resolutions of milliseconds. Among all I/O devices, the PIT interrupts at the highest priority IRQ0. PIT timer interrupts are handled by the timer interrupt handler of the Linux kernel to provide basic timing units for system operation, such as process scheduling, process interval timers and a host of other timing events.

(3). Local Timers in Multicore CPUs (Intel 1997; Wang 2015): In a multicore CPU, each core is an independent processor, which has its own local timer driven by the CPU clock.

(4). High Resolution Timers: Most PCs have a Time Stamp Counter (TSC), which is driven by the system clock. Its contents can be read via a 64-bit TSC register. Since the clock rates of different system boards may vary, the TSC is unsuited as a real-time device but it can provide timer resolutions in nanoseconds. Some high-end PCs may also be equipped with a special high speed timer to provide timer resolutions in nanoseconds range.

5.3 CPU Operations

Every CPU has a Program Counter (PC), also known as the Instruction Pointer (IP), a flag or status register (SR), a Stack Pointer (SP) and several general registers, where PC points to the next instruction to be executed in memory, SR contains current status of the CPU, e.g. operating mode, interrupt mask and condition code, and SP points to the top of the current stack. The stack is a memory area used by the CPU for special operations, such as push, pop call and return, etc. The operations of a CPU can be modeled by an infinite loop.

```
while(power-on){
  (1). fetch instruction: load *PC as instruction, increment PC to point to the
       next instruction in memory;
  (2). decode instruction: interpret the instruction's operation code and
       generate operands;
  (3). execute instruction: perform operation on operands, write results to
       memory if needed; execution may use the stack, implicitly change PC,
       etc.
  (4). check for pending interrupts; may handle interrupts;
}
```

In each of the above steps, an error condition, called an **exception** or trap, may occur due to invalid address, illegal instruction, privilege violation, etc. When the CPU encounters an exception, it follows a pre-installed pointer in memory to execute an exception handler in software. At the end each instruction execution, the CPU checks for pending interrupts. **Interrupts** are external signals from I/O devices or coprocessors to the CPU, requesting for CPU service. If there is a pending interrupt request but the CPU is not in the state of accepting interrupts, i.e. its status register has interrupts masked out, the CPU will ignore the interrupt request and continue to execute the next instruction. Otherwise, it will direct its execution to do interrupt processing. At the end of interrupt processing, it will resume the normal execution of instructions. Interrupts handling and exceptions processing are handled in the operating system kernel. For the most part they are inaccessible from user level programs but they are keys to understanding timer services and signals in operating systems, such as Linux.

5.4 Interrupt Processing

Interrupts from external devices, such as the timer, are fed to predefined input lines of an **Interrupt Controller** (Intel 1990; Wang 2015) which prioritizes the interrupt inputs and routes the interrupt with the highest priority as an interrupt request (IRQ) to the CPU. At the end of each instruction execution, if the CPU is not in the state of accepting interrupts, i.e. it has interrupts masked out in the CPU's status register, it will ignore the interrupt request, keeping it pending, and continue to execute the next instruction. If the CPU is in the state of accepting interrupts, i.e. interrupts are not masked out, the CPU will divert its normal execution sequence to do interrupt processing. For each interrupt, the Interrupt Controller can be programmed to generate a unique number, called an **interrupt vector,** which identifies the interrupt source. After acquiring an interrupt vector number, the CPU uses it as an index to an entry in an **Interrupt Vector Table** (AMD64 2011) in memory, which contains a pointer to the entry address of an **Interrupt handler,** which actually handles the interrupt. When interrupt processing finishes, the CPU resumes normal execution of instructions.

5.5 Time Service Functions

In almost every operating system (OS), the OS kernel provides a variety of time related services. Time services can be invoked by system calls, library functions and user level commands. In this section, we shall cover some of the basic time service functions of Linux.

5.5.1 Gettimeofday-Settimeofday

```
#include <sys/time.h>
int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv, const struct timezone *tz);
```

These are system calls to the Linux kernel. The first parameter, tv, points to a timeval structure.

```
struct timeval {
      time_t          tv_sec;         /* seconds */
      suseconds_t     tv_usec;        /* microseconds */
};
```

The second parameter, timezone, is obsolete and should be set to NULL. The gettimeofday() function returns the current time in seconds and microseconds of the current second. The settimeofday() function sets the current time. In Unix/Linux, time is represented by the number of seconds elapsed since 00:00:00 of January 1, 1970. It can be converted to calendar form by the library function ctime (&time). The following examples demonstrate gettimeofday() and settimeofday().

(1). Gettimeofday system call

Example 5.1 Get system time by gettimeofday()

```
/********* gettimeofday.c file *********/
#include <stdio.h>
```

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

struct timeval t;

int main()
{
  gettimeofday(&t, NULL);
  printf("sec=%ld usec=%d\n", t.tv_sec, t.tv_usec);
  printf((char *)ctime(&t.tv_sec));
}
```

The program should display the current time in seconds, microseconds and also the current date and time in calendar form, as in

```
sec=1515624303 usec=860772
Wed Jan 10 14:45:03 2018
```

(2). Settimeofday system call

Example 5.2 Set system time by settimeofday()

```
/********* settimeofday.c file *********/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>

struct timeval t;

int main()
{
  int r;
  t.tv_sec = 123456789;
  t.tv_usec= 0;
  r = settimeofday(&t, NULL);
  if (!r){
     printf("settimeofday() failed\n");
     exit(1);
  }
  gettimeofday(&t, NULL);
  printf("sec=%ld usec=%ld\n", t.tv_sec, t.tv_usec);
  printf("%s", ctime(&t.tv_sec)); // show time in calendar form
}
```

The output of the program should show something like

```
sec=123456789 usec=862
Thu Nov 29 13:33:09 1973
```

Based on the printed date and year (1973), it seems that the settimeofday() operation has succeeded. However, in some Linux systems, e.g. Ubuntu 15.10, the effect may only be temporary. If the reader runs the gettimeofday program again, it will show that Linux has changed the system time back to the correct real time. This shows that the Linux kernel has the ability to use the real-time clock (and other time synchronization protocols) to correct any deviations of system time from real time.

5.5.2 The Time System Call

Example 5.3 The **time** system call

time_t time(time_t *t)

returns the current time in seconds. If the parameter t is not NULL, it also stores the time in the memory pointed by t. The limitation of the time system call is that it only provides resolutions in seconds, not in microseconds. This example shows how to get system time in seconds.

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

time_t start, end;

int main()
{
  int i;
  start = time(NULL);
  printf("start=%ld\n", start);
  for (i=0; i<123456789; i++); // delay to simulate computation
  end = time(NULL);
  printf("end  =%ld time=%ld\n", end, end-start);
}
```

The output should print the start time, end time and the number of seconds from start to end.

5.5.3 The Times System Call

The times system call

clock_t times(struct tms *buf);

can be used to get detailed execution time of a process. It stores the process time in a struct tms buf, which is

```
struct tms{
        clock_t tms_utime;   // user mode time
        clock_t tms_stime;   // system mode time
        clock_t tms_cutime;  // user time of children
        clock_t tms_cstime;  // system time of children
};
```

All times reported are in clock ticks. This provides information for profiling an executing process, including the time of its children processes, if any.

5.5.4 Time and Date Commands

date: print or set the system date and time
time: report process execution time in user mode, system mode and total time
hwclock: query and set the hardware clock (RTC), can also be done through BIOS.

5.6 Interval Timers

Linux provides each process with three different kinds of interval timers, which can be used as virtual clocks for process timing. Interval timers are created by the **setitimer()** system call. The **getitimer()** system call returns the status of an interval timer.

```
int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

Each interval timer operates in a distinct time domain specified by the parameter **which**. When an interval timer expires, a signal is sent to the process, and the timer is reset to the specified interval value (if nonzero). A **signal** is a number (1 to 31) sent to a process for the process to handle. Signals and signal processing will be covered in Chap. 6. For the time being, the reader may regard timer related signals as interrupts to a process, just like physical timer interrupts to a CPU. The 3 kinds of interval timers are

(1). ITIMER_REAL: decrement in real time, generate a **SIGALRM (14)** signal upon expiration.

(2). ITIMER_VIRTUAL: decrement only when the process is executing in user mode, generate a **SIGVTALRM (26)** signal upon expiration.

(3). ITIMER_PROF: decrement when the process is executing in user mode and also in system (kernel) mode. Coupled with ITIMER_VIRTUAL, this interval timer is usually used to profile the time spent by the application in user and kernel modes. It generates a **SIGPROF (27)** signal upon expiration.

Interval timer values are defined by the following structures (in <sys/time.h>):

```
struct itimerval {
   struct timeval it_interval;   /* interval for periodic timer */
   struct timeval it_value;      /* time until next expiration */
};
struct timeval {
   time_t      tv_sec;           /* seconds */
   suseconds_t tv_usec;          /* microseconds */
};
```

The function getitimer() fills the structure pointed to by curr_value with the current value i.e., the time remaining until the next expiration, of the timer specified by which (one of ITIMER_REAL, ITIMER_VIRTUAL, or ITIMER_PROF). The subfields of the field it_value are set to the amount of time remaining on the timer, or zero if the timer is disabled. The it_interval field is set to the timer interval (period); a value of zero returned in (both subfields of) this field indicates that this is a single-shot timer.

The function setitimer() sets the specified timer to the value in new_value. If old_value is non-NULL, the old value of the timer,i.e. the same information as returned by getitimer(), is stored there.

Periodic timers decrement from it_value to zero, generate a signal, and reset to it_interval. A timer which is set to zero (it_value is zero or the timer expires and it_interval is zero) stops the timer. Both tv_sec and tv_usec are significant in determining the duration of a timer. We demonstrate process interval timers by examples.

Example 5.4 This example shows how to set a VIRTUAL mode interval timer, which decrements its count only when the process is executing in user mode. The timer is set to start after an initial count of 100 msec has expired. Then it runs with a period of 1 second. When the timer count decrements to zero, it generates a SIGVTALRM(26) signal to the process. If the process did not install a catcher for the signal, it will handle the signal by default, which is to terminate. In that case, the process will terminate with a signal number 26. If the process has installed a signal catcher, the Linux kernel will let the process execute the signal catcher to handle the signal in user mode. Before starting the interval time, the program installs a signal catcher for the SIGVTALRM signal by

```
void timer_handler(int sig){ . . . .}
signal(SIGALRM, timer_handler)
```

After installing the signal catcher, the program starts the timer and then executes in a while(1) loop. While executing in the loop, every hardware interrupt, e.g. interrupts from hardware timers, causes the CPU, hence the process executing on the CPU, to enter the Linux kernel to handle the interrupt. Whenever a process is in kernel mode, it checks for pending signals. If there is a pending signal, it tries to handle the signal before returning to user mode. In this case, a SIGVTALRM signal would cause the process to execute the signal catcher in user mode. Since the interval timer is programmed to generate a signal every second, the process would execute timer_handler() once every second, making the printed message to appear once per second like a pulsar. The signal catcher function, timer_handler(), counts the number of times the interval timer has expired. When the count reaches a prescribed value, e.g. 8, it cancels the interval timer by setitimer() with a timer value 0. Although the timer has stopped, the process is still executing in an infinite while(1) loop. In this case, entering a Control-C key from the keyboard would cause the process to terminate with a SIGINT(2) signal. Details of signals and signal handling will be covered in Chap. 6. The following shows the setitimer.c program code of Example 5.5.

```
/*********** setitimer.c file *********/
#include <signal.h>
#include <stdio.h>
#include <sys/time.h>
int count = 0;
struct itimerval t;

void timer_handler(int sig)
{
  printf("timer_handler: signal=%d count=%d\n", sig, ++count);
  if (count>=8){
     printf("cancel timer\n");
     t.it_value.tv_sec = 0;
     t.it_value.tv_usec = 0;
     setitimer(ITIMER_VIRTUAL, &t, NULL);
  }
}

int main()
{
  struct itimerval timer;
  // Install timer_handler as SIGVTALRM signal handler
  signal(SIGVTALRM, timer_handler);
  // Configure the timer to expire after 100 msec
  timer.it_value.tv_sec = 0;
  timer.it_value.tv_usec = 100000; // 100000 nsec
  // and every 1 sec afterward
  timer.it_interval.tv_sec = 1;
  timer.it_interval.tv_usec = 0;
  // Start a VIRTUAL itimer
  setitimer(ITIMER_VIRTUAL, &timer, NULL);
  printf("looping: enter Control-C to terminate\n");
  while(1);
}
```

Figure 5.1 show the outputs of running the setitimer program.

```
looping: enter Control-C to terminate
timer_handler: signal=26 count=1
timer_handler: signal=26 count=2
timer_handler: signal=26 count=3
timer_handler: signal=26 count=4
timer_handler: signal=26 count=5
timer_handler: signal=26 count=6
timer_handler: signal=26 count=7
timer_handler: signal=26 count=8
cancel timer
^C
```

Fig. 5.1 Outputs of setitimer program

5.7 REAL Mode Interval Timer

Interval timers in both VIRTUAL and PROF modes take effect only when a process is executing. Information of such timers can be maintained in the PROC structure of each process. The (hardware) timer interrupt handler only needs to access the PROC structure of the current running process to decrement the timer count, reload the timer count when it expires and generate a signal to the process. The OS kernel does not have to use additional data structures to handle VIRTUAL and PROF timers of processes. However, REAL mode interval timers are different, because they must be updated by the timer interrupt handler whether the process is executing or not. Therefore, the OS kernel must use additional data structures for REAL mode timers of processes and take actions when their timers expire or are cancelled. In most OS kernels, the data structure used is a timer queue. We shall explain the timer queue in the programming project at the end of this chapter.

5.8 Programming Project

The programming project is to implement timer, timer interrupts and interval timers in a multitasking system with concurrent executing tasks. The programming project consists of four steps. Step 1 provides the base code of a multitasking system for the reader to get started. Step 2 is to add timer and timer interrupts to the base system. Step 3 is to implement interval timers for tasks. Step 4 is to implement Critical Regions in the system and task scheduling by time-slice. The object of the project is for the reader to learn not only how to use timers but also how they are implemented in an OS kernel.

5.8.1 System Base Code

The base code of the multitasking system is essentially the same as the programming project on user-level threads of Chap. 4. For ease of reference and completeness, we repeat the same code here. The following lists the base code of a multitasking system. It consists of a ts.s file in 32-bit assembly and a t.c file in C.

```
#------------ ts.s file -------------
.global tswitch, scheduler, running
tswitch:
SAVE:    pushal
         pushfl
         movl running, %ebx
         movl %esp, 4(%ebx)
FIND:    call scheduler
RESUME:  movl running, %ebx
         movl 4(%ebx), %esp
         popfl
         popal
         ret

/********* t.c file *********/
#include <stdio.h>
#include <stdlib.h>
```

```
#include <signal.h>
#include <string.h>
#include <sys/time.h>

#define NPROC    9
#define SSIZE 1024
// PROC status
#define FREE     0
#define READY    1
#define SLEEP    2
#define BLOCK    3
#define PAUSE    4
#define ZOMBIE   5

typedef struct proc{
  struct proc *next;
  int ksp;            // saved sp when NOT running
  int pid;            // task PID
  int priority;       // task priority
  int status;         // status=FREE|READY, etc.
  int event;          // sleep event
  int exitStatus;
  int joinPid;
  struct proc joinPtr;
  int time;           // time slice in ticks
  int pause;          // pause time in seconds
  int stack[SSIZE];   // per task stack
}PROC;

PROC proc[NPROC];     // task PROCs
PROC *freeList, *readyQueue, *running;
PROC *sleepList;      // list of SLEEP tasks
PROC *pauseList;      // list of PAUSE tasks

#include "queue.c"    // same queue.c file as before
#include "wait.c"     // tsleep, twakeup, texit, join functions
int menu() // command menu: to be expanded later
{
  printf("*********** menu ***********\n");
  printf("* create  switch  exit  ps *\n");
  printf("****************************\n");
}

int init()
{
  int i, j;
  PROC *p;
  for (i=0; i<NPROC; i++){
    p = &proc[i];
    p->pid = i;
    p->priority = 1;
```

```
    p->status = FREE;
    p->event = 0;
    p->next = p+1;
  }
  proc[NPROC-1].next = 0;
  freeList = &proc[0];     // all PROCs in freeList
  readyQueue = 0;
  sleepList  = 0;
  pauseList  = 0;
  // create P0 as initial running task
  running = dequeue(&freeList);
  running->status = READY;
  running->priority = 0;  // P0 has lowest priority 0
  printList("freeList", freeList);
  printf("init complete: P0 running\n");
}

int do_exit()  // task exit as FREE
{
  printf("task %d exit: ", running->pid);
  running->status = FREE;
  running->priority = 0;
  enqueue(&freeList, running);
  printList("freeList", freeList);
  tswitch();
}

int do_ps()    // print task status
{
  printf("--------- ps ----------\n");
  printList("readyQueue", readyQueue);
  printList("sleepList ", sleepList);
  printf("----------------------\n");
}

int create(void (f)(), void *parm) // create a new task
{
  int i;
  PROC *p = dequeue(&freeList);
  if (!p){
     printf("create failed\n");
     return -1;
  }
  p->ppid = running->pid;
  p->status = READY;
  p->priority = 1;
  for (i=1; i<12; i++)
      p->stack[SSIZE-i] = 0;
  p->stack[SSIZE-1] = (int)parm;
  p->stack[SSIZE-2] = (int)do_exit;
```

```
  p->stack[SSIZE-3] = (int)f;
  p->ksp = &p->stack[SSIZE-12];
  enqueue(&readyQueue, p);
  printf("%d created a new task %d\n", running->pid, p->pid);
  return p->pid;
}

void func(void *parm) // task function
{
  char line[64], cmd[16];
  printf("task %d start: parm = %d\n", running->pid, parm);
  while(1){
    printf("task %d running\n", running->pid);
    menu();
    printf("enter a command line: ");
    fgets(line, 64, stdin);
    line[strlen(line)-1] = 0; // kill \n at end of line
    sscanf(line, "%s", cmd);
    if (strcmp(cmd, "create")==0)
       create((void *)func, 0);
    else if (strcmp(cmd, "switch")==0)
       tswitch();
    else if (strcmp(cmd, "exit")==0)
       do_exit();
    else if (strcmp(cmd, "ps")==0)
       do_ps();
  }
}

int main()
{
  int i;
  printf("Welcome to the MT multitasking system\n");
  init();
  for (i=1; i<5; i++)    // create tasks
      create(func, 0);
  printf("P0 switch to P1\n");
  while(1){
    if (readyQueue)
       tswitch();
  }
}

int scheduler()
{
  if (running->status == READY)
     enqueue(&readyQueue, running);
  running = dequeue(&readyQueue);
  printf("next running = %d\n", running->pid);
}
```

```
Welcome to the MT multitasking system
freeList = 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> NULL
init complete
task 0 created new task 1
task 0 created new task 2
task 0 created new task 3
task 0 created new task 4
P0 switch to P1
task 1 start: parm = 0
task 1 running
********** menu **********
* create  switch  exit  ps *
****************************
enter a command line: switch
task 2 start: parm = 0
task 2 running
********** menu **********
* create  switch  exit  ps *
****************************
enter a command line:
```

Fig. 5.2 Sample outputs of the base MT system

The reader may consult Chap. 4 for the create() function, which creates a new task to execute a specified function. To compile and run the base system under Linux, enter

gcc –m32 t.c ts.s # assembly code ts.s is for 32-bit Linux

Then run a.out. When a task runs, it displays a command menu, where the commands are

create: create a new task
switch: switch task
exit : task exit
ps : show status of tasks

Figure 5.2 shows the sample outputs of running the base code of the multitasking system.

5.8.2 Timer Interrupts

Step 2 The entire base system runs on a virtual CPU, which is a Linux process. Timer generated signals to the Linux process can be regarded as interrupts to the virtual CPU of the base system. Create a REAL mode interval timer for the Linux process. Program the interval timer to generate a SIGALRM signal every 10 msec. Install a SIGALRM signal catcher as the timer interrupt handler of the virtual CPU. In the timer interrupt handler, keep track of the number of seconds, minutes and hours elapsed. The needed extension code segments are shown below.

```
void thandler(int sig)
{
  // count the number of timer ticks; update ss, mm, hh
  // print a message every second
}
```

```
signal(SIGALRM, thandler);          // install SIGALRM catcher
struct itimerval t;                 // configure timer
t.it_value.tv_sec = 0;
t.it_value.tv_usec = 10000;         // start in 10 msec
t.it_interval.tv_sec = 0;
t.it_interval.tv_usec = 10000;      // period = 10 msec
setitimer(ITIMER_REAL, &t, NULL);   // start REAL mode timer
```

The outputs of Step 2 should be similar to Fig. 5.2, except it displays a message on each second.

5.8.3 Timer Queue

Step 3 Add interval timer support for tasks in the base system. Add the commands

pause t : task pauses for t seconds
timer t : task sets an (REAL mode) interval timer of t seconds

The pause command causes a task to go to sleep for the specified number of seconds, which will be woken up when the pause time expires. After setting an interval timer, the task may continue, which will be notified by a signal when its timer expires. Since the MT system can not generate and send signals yet, we shall assume that the task will go to sleep after executing the timer command, which will be woken up when its timer expires.

As pointed out in Sect. 5.7, the system must use a timer queue to keep track of the status of REAL mode timers of tasks. The timer queue consists of TQE (Timer Queue Element) entries of the form

```
typedef struct tqe{
   struct tqe *next;
   PROC *proc;       // pointer to requesting process
   int   time;       // expiration time count
   void (action)()   // action function = twakeup
}TQE;
TQE *timerQueue = 0; // initialized to NULL
```

The TQEs may be allocated and deallocated from a pool of TQEs. Since each process can have only one REAL mode timer, the number of TQEs is at most equal to the number of PROCs. If desired, they can be included in the PROC structure. Here we assume that the TQEs are allocated/deallocated dynamically from a pool of free TQEs. When a process requests a REAL mode interval timer, a TQE is allocated to record the requesting process, the expiration time, and the action to take when the timer expires. Then it enters the TQE into the timerQueue. The simplest timerQueue is a link list ordered by non-decreasing expiration time. If there are many requests for REAL mode timers, such a timerQueue may look like the following diagram, in which the numbers represents the expiration time of the TQEs.

```
timerQueue = TQE1 ->TQE2 ->TQE3
               2      7       15
```

At each second, the (hardware) timer interrupt handler decrements each and every TQE by 1. When a TQE's time decrements to 0, it is removed from the timerQueue and the action function invoked. The

default action is to generate a SIGALRM(14) signal to the process, but it may also be other kinds of actions. In general, a (hardware) interrupt handler should run to completion as quickly as possible. The drawback of the above timerQueue organization is that the timer interrupt handler must decrement each and every TQE by 1, which is time-consuming. A better way is to maintain the TQEs in a **cumulative** queue, in which the expiration time of each TQE is relative to the cumulative sum of all the expiration time of preceding TQEs. The following shows a cumulative timerQueue, which is equivalent to the original timerQueue.

```
timerQueue = TQE1 ->TQE2 ->TQE3
              2      5      8
```

With this organization of the tiemrQueue, the timer interrupt handler only needs to decrement the first TQE and handle any TQE whose time has expired, which greatly speeds up the timer interrupt processing.

Step 3 of the project is to add a REAL mode interval timer for the Linux process and implement interval timers for tasks in the MT system.

Figure 5.3 shows the sample outputs of running such a system. As the figure shows, task1 requested a timer of 6 seconds. Task 2 requested a timer of 2 seconds, 4 seconds after that of task 1. At this moment, the cumulative timerQueue contents are shown as

timerQueue = [2, 2] =>[1, 2]

In each TQE, the first number is the task PID and the second number is the expiration time. At each second, the timer interrupt handler only decrements the first TQE of task 2. When the TQE of task

```
Welcome to the multitasking system
freeList = 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> NULL
P0 creates tasks
readyQueue = 1 -> 2 -> 3 -> 4 -> NULL
task 1 running
***************** menu ******************
* create  switch  exit  pause  ps  timer *
*****************************************
enter a command line: timer 6
timerQueue = [1, 6] =>
task 2 running
***************** menu ******************
* create  switch  exit  pause  ps  timer *
*****************************************
enter a command line: timerQueue = [1, 5] =>
timer timerQueue = [1, 4] =>
2
timerQueue = [2, 2] => [1, 2] =>
task 3 running
***************** menu ******************
* create  switch  exit  pause  ps  timer *
*****************************************
enter a command line: timerQueue = [2, 1] => [1, 2] =>
timerQueue = [2, 0] => [1, 2] =>
timer wakeup task 2
timerQueue = [1, 1] =>
timerQueue = [1, 0] =>
timer wakeup task 1
```

Fig. 5.3 Sample outputs of task interval timers

2 expires, it deletes the TQE from the timerQueue and wakes up task 2. After these, the TQE of task 1 will be the first one in the timerQueue. On the next second, it will decrement the TQE of task 1. Similarly, the pause command allows a task to pause for t seconds, which will be woken up when its pause time expires.

5.8.4 Critical Regions

In the base code system, there is only one kind of execution entities, namely tasks, which execute one at a time. A task keeps executing until is gets a switch command, goes to sleep or exits. Furthermore, task switch occurs only at the end of an operation, but never in the middle of any operation. Therefore, there are no competitions among tasks, hence no **Critical Regions** in the base code system. However, the situation changes once we introduce interrupts to the system. With interrupts, there are two kinds of execution entities, tasks and interrupt handler, which may compete for the same (shared) data objects in the system. For example, when a task requests an interval timer, it must enter the request as a TQE into the timerQueue. While a task is modifying the timerQueue, if a timer interrupt occurs, it would divert the task to execute the interrupt handler, which may modify the same timerQueue, resulting in a race condition. Thus, the timerQueue forms a Critical Region, which must be protected to ensure it can only be accessed by one execution entity at a time. Likewise, while a process is executing in the middle of the sleep() function, it may be diverted to execute the interrupt handler, which may execute wakeup(), trying to wake up the process before it has completed the sleep operation, resulting in yet another race condition. So the problem is how to prevent task and interrupt handler from interfering with each other.

Step 4: Implementation of Critical Regions
When an interrupt handler executes, the task is logically not executing, so tasks can not interfere with interrupt handler, but the reverse is not true. While a task executes, timer interrupts may occur, which divert the task to execute the interrupt handler, which may interfere with the task. In order to prevent this, it suffices for the executing task to mask out interrupts in Critical Regions. As pointed out before, the MT multitasking system runs on a virtual CPU, which is a Linux process. To the MT system, timer interrupts are SIGALRM signals to the Linux process. In Linux, signals other than SIGKILL(9) and SIGSTOP(19) can be blocked/unblocked by the following means.

(1). sigset_t sigmask, oldmask; // define signal mask sets
(2). sigemptyset(&sigmask); // initialize signal mask set to empty
(3). **sigaddset(&sigmask, SIGALRM); // add signal numbers to the set**
(4). To block signals specified in the mask set, issue the system call
sigprocmask(SIG_BLOCK, &sigmask, &oldmask);
(5). To unblock signals specified in the mask set, issue the system call
sigprocmask(SIG_UNBLOCK, &sigmask, &oldmask);
(6). For convenience, the reader may use the following functions to block/unblock signals
int_off(){ sigprocmask(SIG_BLOCK, &sigmask, &oldmask); }
int_on(){ sigprocmask(SIG_UNBLOCK, &oldmask, &sigmask); }

Step 4 of the programming project is to use sigprocmask() of Linux to block/unblock timer interrupts in Critical Regions in the MT system to ensure no race conditions between tasks and the timer interrupt handler.

5.8.5 Advanced Topics

As a topic for advanced study, the reader may try to implement task scheduling by time-slice and discuss its implications. In time-slice based task scheduling, each task is given a fixed number of timer ticks, known as the time-slice, to run. While a task runs, its time-slice is decremented by the timer interrupt handler. When the time-slice of the running task decrements to 0, the system switches task to run another task. These sounds simple and easy but there are serious implications. The reader is encouraged to think of the problems and possible solutions.

5.9 Summary

This chapter covers timers and timer services. It explains the principle of hardware timers and the hardware timers in Intel x86 based PCs. it covers CPU operations and interrupts processing. It describes timer related system calls, library functions and commands for timer services in Linux. It discusses process interval timers, timer generated signals and demonstrates process interval timers by examples. The programming project is to implement timer, timer interrupts and interval timers in a multitasking system. The multitasking system runs as a Linux process, which acts as a virtual CPU for concurrent tasks inside the Linux process. The real-time mode interval timer of the Linux process is programmed to generate SIGALRM signals periodically, which acts as timer interrupts to the virtual CPU, which uses a SIGALRM signal catcher as the timer interrupt handler. The project is for the reader to implement interval timers for tasks by a timer queue. It also lets the reader use Linux signal masks to implement critical regions to prevent race conditions between tasks and interrupt handlers.

Problems

1. The library function **ctime(&seconds)** converts time in seconds to calendar form, which is a string. Modify the Example 5.1 program to print the current date and time in calendar form.
2. Modify the Example 5.1 program to include a delay loop to simulate a lengthy computation of the program. Use gettimeofday() to get the current time both before and after the loop. Then print the difference between the two times. Can we measure the total execution time of a program this way? Justify your answer.
3. Modify the program in Example 5.1 as follows.

   (1). Install a signal catcher for the SIGPROF(27) signal.
   (2). Start another interval timer in the PROF mode with the same period as the VIRTUAL mode timer.
   (3). Let pcount=number signals generated by the PROF timer, and
   vcount=number signals generated by the VIRTUAL timer.
   In the signal handlers, print both pcount and vcount. When either count reaches 100, stop the timers and print the counts.
   (4). In the while(1) loop of the main program, add the system calls getpid() and getppid(), so that the process will spend some time in system mode.

   Compile and run the program, and answer the following questions.

   (5). Which timer will generate signals faster and WHY?
   (6). How to determine the process execution time in user mode and system mode?

(7). Run the program as time a.out. It will print the execution time of a.out in real mode, user mode and system mode. Explain how is the time command implemented?

4. Modify the program in Example 5.1 to set a REAL mode interval timer.
5. In a cumulative timer queue, the expiration time of each TQE is relative to the cumulative sum of the expiration time of all preceding TQEs. Design an algorithm to insert a TQE with an expiration time t into the timerQueue. Design an algorithm to delete a TQE when a timer request is cancelled before expiration.

References

AMD64 Architecture Programmer's manual Volume 2: System Programming, 2011
Bovet, D.P., Cesati, M., "Understanding the Linux Kernel, Third Edition", O'Reilly, 2005
Intel i486 Processor Programmer's Reference Manual, 1990
Intel MultiProcessor Specification, v1.4, 1997
Wang, K. C., "Design and Implementation of the MTX Operating System", Springer A.G., 2015