

Timers

- *Timer* – Keeps track of the passage of time
- *Simple Timer* – Measures elapsed time, reporting it when queried
- *Interval Timer* –
 - Generates an interrupt after a specific time interval
 - Can cause periodic interrupts (time quantum)

time

SYNOPSIS

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

```
time_t time (time_t *tloc);
```

POSIX: CX

- Returns time in seconds since Epoch
- If tloc is not NULL, time also stores time in *tloc
- On error, time returns -1 and sets errno
- If time_t is 32 bit unsigned long, it would overflow in about 68 years
- If time_t is 64 bit unsigned long, it would overflow in 292 billion years, long after the sun burned out (after a very long time)
- Useful in mathematical calculations

difftime

SYNOPSIS

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

```
double difftime (time_t time1, time_t time0);
```

POSIX: CX

- time1 and time0 are calendar times of type time_t

function_to_time

```
void function_to_time(void)
{
    int i;
    double sum = 0.0;
    for (i=1;i<100000000;i++)
        sum = sum + 1.0/i;
    printf("The sum is %5.3f\n",sum);
}
```

This is the function timed in most of the timing examples that follow

time/difftime example

```
#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;
}
```

ctime Library Function

SYNOPSIS

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

```
char *asctime(const struct tm *timeptr);
```

```
char *ctime(const time_t *clock);
```

```
struct tm *gmtime(const time_t *timer);
```

```
struct tm *localtime(const time_t *timer);
```

POSIX: CX

- ctime converts time to an ASCII string suitable for printing
- Takes a variable of type time_t and returns a pointer to a 26 character string
- Uses static storage
- For example: Sun Oct 06 02:21:35 1986\n\0
- For thread-safe function add underscore r, ie ctime_r

ctime Example

```
#include <stdio.h>
#include <time.h>
int main(void) {
    time_t tcurrent;
    tcurrent = time(NULL);
    printf("The current time is", ctime(&tcurrent));
    return 0;
}
```

gtime and localtime

int tm_sec;	seconds after minute (0,60)
int tm_min;	minutes after hour (0,59)
int tm_hour;	hours since midnight (0,23)
int tm_mday;	day of month (1,31)
int tm_mon;	months since January (0,11)
int tm_year;	years since 1900
int tm_wday;	days since Sunday (0,6)
int tm_yday;	days since January 1 (0,365)
int tm_isdst;	daylight savings time flag

localtime Example

```
struct tm *tcurrent  
tcurrent = localtime (time(NULL));  
printf(“%d days have elapsed since Jan 1\n”, tcurrent->tm_yday);
```

gettimeofday

SYNOPSIS

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

```
int gettimeofday(struct timeval *tp, void *tzp);
```

POSIX:XS/

- Handles timings where seconds is too coarse – returns 0 on success and -1 on error (and sets errno)
- struct timeval
 - time_t tv_sec /* seconds since Epoch */
 - time_t tv_usec /* and microseconds */
- tzp is no longer used – always set it to NULL
- If a long is 32 bits, max duration is $2^{31}-1$ μ secs or \approx 35 minutes.
- This can be extended by using *long long* (usually 64 bits) – format changes are necessary

gettimeofday - Example

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000
void main(void)
{ struct timeval tpstart;
  struct timeval tpend;
  long timedif;
  gettimeofday(&tpstart,NULL);
  function_to_time();          /* timed code goes here */
  gettimeofday(&tpend,NULL);
  timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +
           tpend.tv_usec - tpstart.tv_usec;
  fprintf(stderr, "It took %ld microseconds\n", timedif); }
```

Clocks

- POSIX Realtime Extension contains *clocks*
- A clock is a counter that increments at fixed intervals called the *clock resolution*

clock_gettime Library Function

SYNOPSIS

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

```
int clock_gettime(clockid_t clock_id, struct timespec *tp);
```

```
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

```
int clock_getres(clockid_t clock_id, struct timespec *res);
```

POSIX: *TMR*

- These functions return 0 on success and -1 and set errno on failure
- struct timespec
 - time_t tv_sec /* seconds */
 - long tv_nsec /* nanoseconds */

clock_gettime – Example

```
/* Example 6.3 */
#include <stdio.h>
#include <time.h>
#define MILLION 1000000
void main(void)
{ struct timespec tpstart;
  struct timespec tpend;
  long timedif;
  clock_gettime(CLOCK_REALTIME, &tpstart);
  function_to_time();          /* timed code goes here */
  clock_gettime(CLOCK_REALTIME, &tpend);
  timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +
           (tpend.tv_nsec - tpstart.tv_nsec)/1000;
  fprintf(stderr, "It took %ld microseconds\n", timedif); }
```

sysconf Library Function

SYNOPSIS

```
#include <unistd.h>
```

```
long sysconf(int name);
```

- sysconf is used to find the number of ticks per second in a system
- *Example* –
ticks = double sysconf(_SC_CLK_TCK)

times Library Function

SYNOPSIS

```
#include <sys/times.h>
```

```
clock_t times (struct tms *buffer);
```

POSIX

- clock_t holds a number of clock ticks
- struct tms contains:
 - clock_t tms_utime /* user CPU time of process*/
 - clock_t tms_stime /* system CPU time on behalf of process */
 - clock_t tms_cutime /* user CPU time of process and terminated children */
 - clock_t tms_cstime /* system CPU time on behalf of process and terminated children */
- Times returns elapsed time since an arbitrary point in past (or -1 on error and sets errno)

times - Example

```
void main(void)
{ clock_t real_start;
  clock_t real_end;
  clock_t ticks_used;
  struct tms process_start;
  struct tms process_end;
  if ((real_start = times(&process_start)) == -1)
    perror("Could not get starting times");
  else {
    /* perform calculation to be timed */
    function_to_time();
    if ((real_end = times(&process_end)) == -1)
      perror("Could not get ending times");
    else {
      ticks_used = process_end.tms_utime + process_end.tms_stime
        - process_start.tms_utime - process_start.tms_stime;
      printf("Fraction of time running = %f\n",
        (double)(ticks_used)/(real_end - real_start)); } } }
```

sleep

SYNOPSIS

```
#include<unistd.h>
```

```
unsigned sleep(unsigned seconds);
```

POSIX

- Sleep blocks for the number of seconds specified
- It could be implemented with an interval timer

nanosleep

SYNOPSIS

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

```
int nanosleep(const struct timespec *rqtp, struct timespec  
    *rmtp);
```

POSIX:*TMR*

- Causes the calling thread to suspend execution until the time interval specified by `rqtp` has elapsed or signal is received
- If interrupted by signal and `rmtp` is not NULL, location pointed to by `rmtp` contains time remaining allowing `nanosleep` to be restarted
- The system clock `CLOCK_REALTIME` determines resolution of `rqtp`

Interval Timers

- POSIX:*XSI*
 - Gives each process a small fixed number of timers
 - One of each of the types
 - ITIMER_REAL
 - ITIMER_VIRTUAL
 - ITIMER_PROF
- POSIX:*TMR*
 - Takes an alternative approach where there are a small number of clocks, such as CLOCK_REALTIME
 - A process can create many independent timers for each clock

POSIX:XSI/ Interval Timers

- ITIMER_REAL: decrements in real time and generates a SIGALRM signal when it expires
- ITIMER_VIRTUAL: decrements in virtual time (time used by the process) and generates a SIGVTARM signal when it expires
- ITIMER_PROF: decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires

struct itimerval

- `it_value` /* time until next expiration */
- `it_interval` /* value to reload into timer */

setitimer

SYNOPSIS

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

```
int setitimer (int which, const struct itimerval *value,  
              struct itimerval *ovalue);
```

POSIX:XSI

ITIMER_PROF - Example (top)

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include <unistd.h>
char astbuf[] = "*";
static void myhandler(int s)
{
    write(STDERR_FILENO, astbuf, sizeof(char));
}
/* set up the myhandler handler for signal SIGPROF */
void init_timer_interrupt(void)
{
    struct sigaction newact;
    newact.sa_handler = myhandler;
    newact.sa_flags = SA_RESTART;
    sigemptyset(&newact.sa_mask);
    sigaction(SIGPROF, &newact, NULL);
}
```

ITIMER_PROF – Example (bottom)

```
/* set the ITIMER_PROF interval timer for 2-second intervals */  
void setup_interval_timer(void)  
{ struct itimerval value;  
  value.it_interval.tv_sec = 2;  
  value.it_interval.tv_usec = 0;  
  value.it_value = value.it_interval;  
  setitimer(ITIMER_PROF, &value, NULL); }  
void main(void)  
{ init_timer_interrupt();  
  setup_interval_timer();  
  /* execute rest of main program here */  
  function_to_time();  
  exit(0); }
```

getitimer

SYNOPSIS

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

```
int getitimer (int which, struct itimerval *value);
```

POSIX:XS/

- Use getitimer to determine the amount of time remaining on a POSIX:XS/ interval timer
- getitimer sets the *value structure with the time remaining until the which timer expires
- getitimer returns 0 on success and returns -1 and sets errno on failure

POSIX:XSI/ Interval Timer to Time Code

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000 void main(void)
{ struct itimerval value;
  struct itimerval ovalue;
  long timedif;
  value.it_interval.tv_sec = 0;
  value.it_interval.tv_usec = 0;
  value.it_value.tv_sec = MILLION;    /* a large number */
  value.it_value.tv_usec = 0;
  setitimer(ITIMER_VIRTUAL, &value, NULL);
  getitimer(ITIMER_VIRTUAL, &ovalue);
  function_to_time();                /* timed code goes here */
  getitimer(ITIMER_VIRTUAL, &value);
  timedif = MILLION*(ovalue.it_value.tv_sec - value.it_value.tv_sec)
    + ovalue.it_value.tv_usec - value.it_value.tv_usec;
  printf("It took %ld microseconds\n", timedif); }
```

POSIX.*TMR* Interval Timers

- POSIX.*TMR* interval timers are per-process timers not inherited on a fork
- POSIX.*TMR* timers are based on the `itimerspec` structure with the following members:
 - `it_interval` /* timer period */
 - `it_value` /* timer expiration */

timer_create

SYNOPSIS

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

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

```
int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t  
    *timerid);
```

```
struct sigevent {
```

```
    int          sigev_notify    /* notification type */
```

```
    int          sigev_signo;    /* signal number */
```

```
    union sigval sigev_value;    /* signal value */
```

```
};
```

```
union sigval {
```

```
    int          sival_int;      /* integer value */
```

```
    void         *sival_ptr     /* pointer value */
```

```
};
```

timer_create returns 0 on success or -1 and sets errno on error

timer_create parameters

- `clock_id` specifies the clock the timer is based on
- `*timerid` holds the ID of the created timer
- `sigevent` structure and `sigval` union are required by POSIX.1b but there may be additional parameters
- `*evp` specifies signal to be sent – for `CLOCK_REALTIME` the default is `SIGALRM`
- The `evp->sigev_notify` member indicates action to be taken on timer expiration
- If `evp->sigev_notify` is set to `SIGEV_NONE`, no signal is sent
- If several timers generate the same signal, `evp->sigev_value` is used to distinguish which timer generated the signal – Use `SA_SIGINFO` flag in `sa_flags` member of `struct sigaction` to do this

POSIX. *TMR* Interval Timer Operations

SYNOPSIS

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

```
int timer_settime(timer_t timerid, int flags, const struct  
    itimerspec *value, struct itimerspec *ovalue);
```

```
int timer_gettime(timer_t timerid, struct itimerspec  
    *value);
```

```
int timer_getoverrun(timer_t timerid);
```

timer_settime

- Starts or stops a timer created by calling `timer_create`
- `flags` parameter indicates whether timer uses relative or absolute time – relative time is similar to Spec 1170 timers while absolute time allows for greater accuracy and control of timer drift
- `value` and `ovalue` parameters have same meaning as for `setitimer`

timer_gettime

- Gets the time remaining on an active timer
- It is possible for timer to expire while a signal is still pending from a previous expiration of the same timer – signals may be lost (called timer overrun)
- Timer overruns occur only in signals generated by the same timer

POSIX.*TMR* Interval Timer to Time Code (top)

```
/* Program 6.3 */
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#define MILLION 1000000
#define THOUSAND 1000

void main(void)
{ timer_t time_ID;
  struct itimerspec value;
  struct itimerspec ovalue;
  long timedif;
```

POSIX.*TMR* Interval Timer to Time Code (bottom)

```
if (timer_create(CLOCK_REALTIME, NULL, &time_ID) < 0) {
    perror("Could not create a timer based on CLOCK_REALTIME");
    exit(1); }
value.it_interval.tv_sec = 0;
value.it_interval.tv_nsec = 0;
value.it_value.tv_sec = MILLION;    /* a large number */
value.it_value.tv_nsec = 0;
timer_settime(time_ID, 0, &value, NULL);
timer_gettime(time_ID, &ovalue);
function_to_time();                  /* timed code goes here */
timer_gettime(time_ID, &value);
timedif = MILLION*(ovalue.it_value.tv_sec - value.it_value.tv_sec)
    + (ovalue.it_value.tv_nsec - value.it_value.tv_nsec)/THOUSAND;
printf("It took %ld microseconds\n", timedif); }
```