Chapter 9
Times and Timers

Mingyu Lim
CCSLab (http://ccslab.konkuk.ac.kr)
Konkuk University
Objectives

- Learn how time is represented
- Experiment with interval timers
- Explore interactions of timers and signals
- Use timers to assess performance
- Understand POSIX real-time signals
POSIX Times

• Systems should keep time in terms of seconds since the Epoch
• The Epoch is defined as 00:00 (midnight), January 1, 1970, Coordinated Universal Time (also called UTC, Greenwich Mean Time or GMT)
• POSIX does not specify how an implementation should align its system time with the actual time and date
Time in seconds

```c
#include <time.h>
time_t time(time_t *tloc);
```

- Access the system time (expressed in seconds since the Epoch)
- Parameters
  - tloc: if not NULL, the function also stores the time in *tloc
    - time_t: long type
- Return values
  - If successful, The number of seconds since the Epoch
  - If unsuccessful, (time_t)-1
  - No mandatory errors defined
- For a 32-bit long, time would overflow in approximately 68 years from January 1, 1970 (2038)
#include <time.h>
double difftime(time_t time1, time_t time0);

• Computes the difference between two calendar times of type time_t

• Parameters
  – Two time_t parameters

• Return values
  – A double containing the first parameter minus the second
  – No errors defined
Displaying date and time

```c
#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);
```

- **localtime()**
  - Takes a parameter specifying the seconds since the Epoch
  - Returns a structure with the components of the time adjusted for local requirements
- **asctime()**
  - Converts the structure returned by localtime to a string
- **ctime()**
  - Equivalent to asctime(localtime(clock))
  - Returns a pointer to a 26-character English string that ends in a newline
  - Uses static storage to hold the time string
- **gmtime()**
  - Returns a structure with the components of time expressed as UTC
- **asctime, ctime, localtime are not thread-safe**
  - The POSIX:TSF Thread Safe Extension specifies thread-safe alternatives that have an additional buffer parameter
Displaying date and time

- **struct tm structure**
  - Used by gmtime and localtime functions
  - **Members**
    - int tm_sec; /* seconds [0,59]*/
    - int tm_min; /* minutes [0,59] */
    - int tm_hour; /* hours [0,23] */
    - int tm_mday; /* day of the months [1,31] */
    - int tm_mon; /* months [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; /* flag for daylight-saving time */
struct timeval

#include <sys/time.h>
int gettimeofday(struct timeval *restrict tp, void *restrict tzp);

• struct timeval
  – The POSIX:XSI Extension uses the structure to express time on a finer scale
  – Members
    • time_t tv_sec; /* seconds since the Epoch */
    • time_t tv_usec; /* and microseconds */

• gettimeofday()
  – Retrieves the system time in seconds and microseconds since the Epoch
  – Parameters
    • tp: receives the retrieved time
    • tzp: must be NULL (included for historical reasons)
  – Return values
    • 0, if successful
    • No errors defined (may systems implemented their own errors)
  – Program 9.1
    • Shows how to measure the running time of function_to_time by using gettimeofday function
    • If a long is 32 bits, the maximum duration is $2^{31} - 1$ microseconds, or approximately 35 minutes
Using real-time clocks

#include <time.h>
int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);

• Clock
  – A counter that increments at fixed intervals called the clock resolution
  – The POSIX:TMR Timers Extension contains clocks that are represented by variables of type clockid_t
  – All implementations must support a systemwide clock with a clockid_t value of CLOCK_REALTIME corresponding to the system realtime clock

• struct timespec structure
  – time_t tv_sec; /* seconds */
  – long tv_nsec; /* nanoseconds */

• Clock functions
  – Returns 0 if successful, -1 and sets errno if unsuccessful
Example using real-time clocks

```c
#include <stdio.h>
#include <time.h>
#define MILLION 1000000L

void function_to_time(void);

int main (void) {
    long timedif;
    struct timespec tpend, tpstart;

    if (clock_gettime(CLOCK_REALTIME, &tpstart) == -1) {
        perror("Failed to get starting time");
        return 1;
    }
    function_to_time();                           /* timed code goes here */
    if (clock_gettime(CLOCK_REALTIME, &tpend) == -1) {
        perror("Failed to get ending time");
        return 1;
    }
    timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +
               (tpend.tv_nsec - tpstart.tv_nsec)/1000;
    printf("The function_to_time took %ld microseconds\n", timedif);
    return 0;
}
```
Processor time

```c
#include <sys/time.h>
clock_t times(struct tms *buffer);
```

- Accurate measurement of execution time
  - In a multiprogramming environment many processes share the CPU, so real time (by `time()`) is not an accurate measure of execution time
  - The virtual time for a process
    - The amount of time that the process spends in the running state
    - Execution times are usually expressed in terms of virtual time rather than wall-clock time

- `times()`
  - Fills the struct tms structure with time-accounting information
  - If successful, it returns the elapsed real time, in clock ticks
  - If it fails, returns (clock_t)-1 and set errno

- struct tms structure
  - `clock_t tms_utime;` /* user CPU time of process */
    - the CPU time spent executing instructions of the calling process
  - `clock_t tms_stime;` /* system CPU time on behalf of process */
    - the CPU time spent in the system while executing tasks on behalf of the calling process
  - `clock_t tms_cutime;` /* user CPU time of process and terminated children*/
    - the sum of the `tms_utime` and `tms_cutime` values for all waited-for terminated children
  - `clock_t tms_cstime;` /* system CPU time of process and terminated children */
    - the sum of the `tms_stime` and `tms_cstime` values for all waited-for terminated children
Example of processor time

- **Program 9.4**
  - Estimates the total of the amount of CPU time used by function_to_time as well as the fraction of the total CPU time used
  - Calls sysconf to determine the number of clock ticks in a second
    - `sysconf(_SC_CLK_TCK)`
  - The user time and the system time used on behalf of the process
    - `cticks = tmend.tms_utime + tmend.tms_stime - tmstart.tms_utime – tmstart.tms_stime`
  - The fraction of total CPU time may be inaccurate if a context switch occurs during the execution
    - `cticks / (tcend – tcstart)`

- **Program 9.5**
  - Estimates the total of the amount of CPU time of a child process
    - `cticks = tmend.tms_cutime + tmend.tms_cstime - tmend.tms_utime – tmend.tms_stime;`
Sleep functions

#include <unistd.h>
unsigned sleep(unsigned seconds);
#include <time.h>
int nanosleep(const struct timespec *rqtp, struct timespec *rmtp);

• sleep()
  – Causes the calling thread to be suspended either until the specified number of seconds has elapsed or until the calling thread catches a signal
  – Returns 0 if the requested time has elapsed
  – Returns the amount of unslept time if interrupted

• nanosleep()
  – Causes the calling thread to suspend execution until the time interval specified by rqtp has elapsed or until the thread receives a signal
  – If rmtt is not NULL, it contains the time remaining
  – Returns 0 if successful, -1 and sets errno if unsuccessful
  – Does not affect the use of any signals, including SIGALRM
POSIX:XSI Interval Timer

• Timers
  – A computer system typically has a small number of hardware interval timers
    • Clock vs. timer?
  – The OS implements multiple software timers by using these hardware timers
  – Time-sharing OS can also use interval timers for process scheduling

• POSIX:XSI Timers
  – Use a struct itimerval structure
    • struct timeval it_value; /* time until next expiration */
    • struct timeval it_interval; /* value to reload into the timer */
POSIX: XSI Interval Timer

#include <sys/time.h>
int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *restrict value, struct itimerval *restrict ovalue);

• getitimer()
  – Stores the current value of the time for timer ‘which’ in ‘value’
    • 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 SIGVTALRM signal when it expires
    • ITIMER_PROF
      – Decrements in virtual time and system time for the process and generates a SIGPROF signal when it expires

• setitimer()
  – Sets the timer specified by ‘which’ to ‘value’
  – If ovalue is not NULL, previous value is saved
  – If value->it_interval is not 0, the timer restarts with this value when it expires
  – If value->it_interval is 0, the timer does not restarts when it expires
  – If value->it_value is 0, the timer stops if it is running
Example of timer

- **Program 9.7**
  - Print out an asterisk for each two seconds
  - Use ITIMER_PROF timer

```c
... static int setupitimer(void) {
    struct itimerval value;
    value.it_interval.tv_sec = 2;
    value.it_interval.tv_usec = 0;
    value.it_value = value.it_interval;
    return (setitimer(ITIMER_PROF, &value, NULL));
} ...
```

- **Program 9.8**
  - Use ITIMER_VIRTUAL to measure the execution time of function_to_time()

```c
... struct itimerval ovalue, value;
    ovalue.it_interval.tv_sec = 0;
    ovalue.it_interval.tv_usec = 0;
    ovalue.it_value.tv_sec = 0;
    ovalue.it_value.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &ovalue, NULL);
    function_to_time();
    getitimer(ITIMER_VIRTUAL, &value);
... ```
Real-time signals

• POSIX standard
  – A signal handler is a function with a single integer parameter

• POSIX:XSI Extension and POSIX:RTS
  Realtime Signal Extension
  – Queuing of signals and passing of information to signal handlers
  – If _POSIX_REALTIME_SIGNALS is defined, the real-time signals are supported
Real-time signals

```c
#include <signal.h>
struct sigaction{
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_sigaction) (int, siginfo_t *, void *); /* real-time handler */
};
```

- **sa_sigaction**
  - Used if sa_flags & SA_SIGINFO is nonzero
  - Handler form
    - `void func(int signo, siginfo_t *info, void *context);`
    - Context is not currently defined
  - `siginfo_t` structure
    - `int si_signo; /* signal number */`
      - Same as the value passed by signo of func
    - `int si_code; /* cause of the signal */`
      - SL_USER, SL_QUEUE, SL_TIMER, SL_ASYNCIO, SL_MESSGQ
    - `Union sigval si_value;/* signal value */`
      - Application specified signal value
        - `int sival_int;`
        - `void *sival_ptr;`
Signal queuing

• When multiple signals are pending, POSIX guarantees that at least one instance is delivered if the signal is unblocked
• Additional instances may be lost
• POSIX:RTS signal queuing facility for receiving every signal
Signal queuing

```c
#include <signal.h>
int sigqueue(pid_t pid, int signo, const union sigval value);
```

- An extension to kill command
- Sends signal ‘signo’ with value ‘value’ to the process with ID ‘pid’
- If signo is 0, error checking is performed, but no signal is sent
- If SA_SIGINFO in the sa_flags field was set, the signal is queued and sent to the receiving process
- **Program 9.9**
  - Sends queued signals to a process
- **Program 9.10**
  - Prints its process ID, sets up a signal handler for SIGUSR1, and suspends itself until a signal arrives
POSIX: TMR interval timers

- A small of clocks, such as CLOCK_REALTIME, and a process can create many independent timers for each clock
- Based on the struct itimerspec structure
  - struct timespec it_interval; /* timer period */
  - struct timespec it_value; /* expiration */
  - Better resolution than the struct timeval
POSIX: TMR interval timers

```c
#include <signal.h>
#include <time.h>
int timer_create(clockid_t clock_id, struct sigevent *restrict evp, timer_t *restrict timerid);
int timer_delete(timer_t timerid)
```

- **timer_create()**
  - Create per-process timers
  - Not inherited on fork
- **Parameters**
  - `clock_id`: which clock the timer is based on
  - `timerid`: holds the ID of the created timer
  - `evp`
    - Specifies the asynchronous notification to occur when the timer expires
    - If NULL, the timer generates the default signal (SIGALRM for CLOCK_REALTIME)
    - `evp->sigev_signo`: desired signal number
    - `evp->sigev_notify`: specifies the action to be taken when the timer expires
      - SIGEV_SIGNAL: the timer expiration generates a signal
      - SIGEV_NONE: prevent the timer from generating a signal
POSIX: TMR interval timers

```c
#include <time.h>
int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
                 struct itimerspec *ovalue);
```

- **timer_settime()**
  - Start and stop a timer
  - ‘flags’ specifies whether the timer uses relative or absolute time
  - If ‘ovalue’ is not NULL, it saves the previous value

- **timer_overrun()**
  - Timer overrun
    - One of signals generated may be lost when a timer expires while a signal is still pending from a previous expiration of the same timer
  - Returns the number of overruns
POSIX:TMR interval timers

- **Program 9.11**
  - Shows how to create a timer that generates periodic interrupts
- **Program 9.12**
  - Measure the running time of function_to_time()
  - Similar to 9.8 but it uses real time rather than virtual time
Timer drift

• **Reason**
  – The latency between when the timer was due to expire and when the timer was reset
    • 2 seconds interval is actually 2.000005 seconds if the latency is 5 microseconds

• **Example**
  – Consider a repeating timer with period of 22 ms when the timer has a resolution of 10 ms
    • The drift grows by 8 ms on each expiration

• **Solution using absolute time**
  – Keep track of when the timer should actually expire and adjust the value for setting the timer each time
    • Save $T = \text{current time} + 22 \text{ ms}$
    • Set the timer to expire in 22 ms
    • In the signal handler, set the timer to expire in $(T - \text{current time} + 22\text{ms})$, and $T = T + 22 \text{ ms}$
Timer drift

• Solution using POSIX:TMR timers with absolute time
  – The flags parameter of timer_settime can be set to TIMER_ABSOLUTE
    • the time given in the it_value member of the *value parameter represents the real time rather than a time interval
  – Determine the current time by using clock_gettime and add 22 ms. Save this as $T$
  – Set the timer to expire at time $T$ using TIMER_ABSOLUTE flag
  – In the timer signal handler, add 22 ms to $T$ and set the timer to expire at time $T$