Time & Timers

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss the use of time in C including the representation of time and converting time into human readable forms
- Explain the process of waiting using sleep
- Discuss using timers and interval timers

Time

- Time serves a variety of purposes and the kernel keeps track of time in 3 ways:
  - Wall time (real time)
    - Actual time and date
  - Process time
    - Time that a process spends executing on a processor
      - Could be:
        - » ***User Time***: time the process itself spent executing
        - » ***System Time***: time the kernel spent working on the process
  - Monotonic time
    - Time source is strictly linearly increasing
    - Many systems (including linux) use uptime

Time Representation

- All three measurements of time can be represented in one of two formats:
  - Relative time
    - Value relative to a benchmark (such as current instant)
  - Absolute time
    - Represents time without any benchmark
    - UNIX represents this as the number of seconds since the epoch (1/1/1970 @ 00:00:00)

Operating Systems & Time

- OS track progression of time using a ***software clock*** which is a clock maintained by the kernel in software
  - Kernel instantiates a periodic timer, known as the system timer that pops at a specific frequence
  - When the timer interval ends, the kernel increments the elapsed time by one unit known as a ***tick*** or ***jiffy***

Time's Data Structures

- Original Spec:
  - Defined in <time.h>
    - **typedef long time_t**
    - Represents the number of seconds since the epoch
- And Now, Microsecond precision
  - #include <sys/time.h>

```
struct timeval {
      time_t          tv_sec;
      suseconds_t     tv_usec; };
```

- Even Better, Nanosecond precision
  - #include <time.h>

```
struct timespec {
      time_t     tv_sec;
      long       tv_nsec; };
```

Breaking Down Time

```
#include <time.h>
struct tm {
      int tm_sec;
      int tm_min;
      int tm_hour;
      int tm_mday;
      int tm_mon;
      int tm_year;
      int tm_year;
      int tm_wday;
      int tm_yday;
      int tm_isdst;
};
```


Time of the Day?

```
Call:
#include <time.h>
time_t time(time_t *t)

EX:
time_t t;
long val = (long)time(&t);
printf("Time: %ld\n", val);
```


A Better Interface

- We can get better resolution:
  - #include <sys/time.h>
  - int gettimeofday(struct timeval *tv, struct timezone *tz)
    - NOTE: tz is obsolete -> always pass NULL

Even More Advanced

```
Call:
#include <time.h>
int clock_gettime(clockid_t clock_id,
                  struct timespec *ts)

Ex:
struct timespec ts;
ret = clock_gettime(CLOCK_REALTIME, &ts)
```


Getting Process Time

```
#include <sys/times.h>
struct tms {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
};
clock_t times(struct tms *buf)
```


POSIX Clocks

- Several of the system calls discussed use POSIX clocks (type clockid_t) of which there are 5 we need to care about
  - **CLOCK_REALTIME**
  - CLOCK_MONOTONIC
  - CLOCK_MONOTONIC_RAW
  - CLOCK_PROCESS_CPUTIME_ID
  - CLOCK_THREAD_CPUTIME_ID

Time Source Resolution

```
#include <time.h>
int clock_getres
    (clockid_t clock_id,
     struct timespec *res)
```


Clock Resolution Example

```
clockid_t clocks[] = { CLOCK_REALTIME,
CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW,
CLOCK_PROCESS_CPUTIME_ID,
CLOCK_THREAD_CPUTIME_ID };
for (int i=0; i<5; i++) {
   int ret;
   struct timespec res;
   ret = clock_getres(clocks[i], &res);
   if (ret != 0) {
      printf("clock=%d sec=%ld nsec=%ld\n",
         clocks[i], res.tv_sec, res.tv_nsec)
}  }
```


Playing with Time

- One of the more important series of functions a PL has is taking time and making it human understandable
  - We can take the broken-down time into an ASCII representation of the time

```
#include <time.h>
char * asctime(const struct tm *tm)
char * asctime_r(const struct tm *tm,
                 char * buf)
```

More Conversions

Convert a tm to a time_t

```
#include <time.h>
time_t mktime (struct tm *tm)
```

Convert a time_t to it's ASCII representation:

```
#include time.h>
char * ctime(const time_t *time)
char * ctime_r(const time_t *time,
               char * buf)
```


More Conversions (2)

Convert time_t to tm (GMT)

```
struct tm* gmtime(const time_t *time)
struct tm* gmtime_r(const time_t *time,
                    struct tm* result)
```

Convert time_t to tm (Local)

```
struct tm* localtime(time_t *time)
struct tm* localtime_r(const time_t *time,
                       struct tm* result)
```


More Conversions (3)

- One last thing: on Linux, **`time_t`** is guaranteed to be an integer type so there's no need to do anything special
  - `(double)(time1 – time0)`
- However, on other systems, time_t is actually a data structure, so if you want portability
  - `#include <time.h>`
  - `double difftime(time_t time1, time_t time0)`

Sleeping & Waiting

- We've seen sleeping which suspends execution of the process for a given amount of time
  - `#include <unistd.h>`
  - `unsigned int sleep(unsigned int seconds)`
- We can sleep with microsecond precision
  - #BSD variants:
    - `void usleep(unsigned long usec)`
  - #SUS versions
    - `int usleep(useconds_t usec)`

Nanosecond Sleeping

- We can even sleep for nanoseconds

```
#include <time.h>
int nanosleep(const struct timespec *req,
              struct timespec *rem)
```


Alternatives to Sleeping

- Generally, you should avoid sleeping…
  - Although it's overused in educational settings for demonstration purposes, there's almost always a better way to do it
- However, how do we deal with events?
  - Instead of the process spinning in a loop, until the event happens, we can block the process until the event occurs

Timers

- Timers are the mechanism for notifying a process when a given amount of time has passed
  - The amount of time before a timer ***expires*** is called the ***delay*** or ***expiration***
  - How the kernel notifies the process depends on the timer in play

Simple Alarms

- Simple alarm() call:
  - `#include <unistd.h>`
  - `unsigned int alarm (unsigned int seconds)`
  - A call to this function schedules the delivery of a SIGALRM signal to this process
  - If a previously scheduled signal was pending, the call cancels the alarm, replaces it with the new alarm, and returns the number of seconds remaining in the old alarm
  - Successful use of this, of course, depends on signal handling the SIGALRM signal

Alarm() Example

```
void alarm_handler(int signal) {
    printf("Five Seconds passed!\n");
}

main() {
    signal(SIGALRM, alarm_handler);
    alarm(5);
    pause();
}
```

Interval Timers

- Interval timer system calls provide more control than alarm()
  - `#include <sys/time.h>`
  - `int getitimer(int which, struct itimerval * val)`
  - `int setitimer(int which, const struct itimerval *val, struct itimerval * oval)`

Interval Timer Modes

- The ***int which*** refers to one of three modes:
  - ITIMER_REAL
    - Measures real time
  - **ITIMER_VIRTUAL (profiling)**
    - Decrements only when user space code is executing
  - **ITIMER_PROF (profiling)**
    - Decrements both while the process is executing and while the kernel is executing

`struct itimerval`

```
struct itimerval {
    struct timeval it_interval;
    struct timeval it_value;
};
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

Example

```
void alarm_handler(int signal) {
     printf("Timer Hit!\n"); }
main() {
    struct itimerval delay;
    signal(SIGALARM, alarm_handler);
    delay.it_value.tv_sec =5;
    delay.it_value.tv_usec = 0;
    delay.it_interval.tv_sec = 1;
    delay.it_interval.tv_usec = 0;

    setitimer(ITIMER_REAL, &delay, NULL); }
```


Summary

- Discussed signals in terms of purpose, use, lifecycle and their symbolic identifiers
- Explained a common set of signals including their default operations and events
- Discussed basic signal management including sending signals, catching signals, ignoring, and waiting for signals
- Explained reentrancy as it relates to signals and discuss blocking and restoring signals

Questions?