

# CS241 Operating Systems

## CPU Scheduling (5)

Klara Nahrstedt

Lecture 14

2/20/2006

---

# Content

- Timers:TMR
- gprof
- Timers
  - POSIX XSI
- Summary

# Administrative Notes

- MP2 on scheduling is running
- Quiz 4 on Friday, 2/24/06
- Quiz 4 covers readings of book chapters and lecture notes
  - Tanenbaum: Scheduling 2.4
  - R&R: Signals 8.1-8.5,
  - R&R: Signal Handling and Threads 13.5
  - R&R: 9.1
  - Lecture Notes:
    - lec10-sched, lec11-sched, lec12-sched, lec13-sched, lec14-sched

# POSIX Times

- POSIX specifies time in seconds since the Epoch
- Epoch is defined as 00:00 (midnight), January 1, 1970, Coordinated Universal Time (UTC)
- `#include <time.h>`
- `time_t time(time_t *calptr);`
- Day is 86,400 seconds
- `time_t` is usually a *long*
- If the *long* is 32 bits, time overflows in 2038
- Two POSIX Extensions
  - POSIX:TMR (used also in our MP)
  - POSIX:XSI

# Timing a Function

```
#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);
}
```

# Expressing, Displaying Time

- POSIX base standard supports only time resolution of seconds
- Expression of time as `time_t` type

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

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

- Displaying current time

```
#include <stdio.h>
```

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

```
int main(void) {
```

```
    time_t tcurrent;
```

```
    tcurrent = time(NULL);
```

```
    printf("the current time is %s", ctime(&tcurrent));
```

```
    return 0;
```

```
}
```

# POSIX:TMR Extension (1)

- A *clock* is a counter – increments at fixed intervals and represents the *clock resolution*
- POSIX:TMR represents clocks as variables of type **clockid\_t**
- All implementations of POSIX clocks must support systemwide clock with **clockid\_t** value **CLOCK\_REALTIME** (corresponds to system realtime clock)
- Finer granularity clock structure

**struct timespec** has the following parameters:

**time\_t tv\_sec** /\*seconds\*/

**long tv\_nsec** /\*nanoseconds\*/

# POSIX:TMR Extension (2)

- Important Clock Functions
  - Set the clock time – **clock\_gettime**
  - Retrieve the clock time – **clock\_gettime**
  - Determine the clock resolution – **clock\_getres**

```
#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);
```

# Time Measurements

## (Get Processing Time of Executing Function)

```
#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();
    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;
}

function_to_time(void) {
    int x;
    int i;
    x=1;
    for (i=0;i<100; i++) { x++;}
}
```

# gprof



To compile a source file for profiling, specify the ``-pg'` option when you run the compiler. (This is in addition to the options you normally use.)

To link the program for profiling, if you use a compiler such as `cc` to do the linking, simply specify ``-pg'` in addition to your usual options. The same option, ``-pg'`, alters either compilation or linking to do what is necessary for profiling.

Here are examples:

```
cc -g -c myprog.c utils.c -pg
```

```
cc -o myprog myprog.o utils.o -pg
```

The ``-pg'` option also works with a command that both compiles and links:

```
cc -o myprog myprog.c utils.c -g -pg
```

# gprof data

- `gmon.out' file is written in the program's *current working directory* at the time it exits
- your program must exit normally: by returning from main or by calling exit

# gprof analysis

- `gprof options [executable-file [profile-data-files...]] [> outfile]`
- Options include time spent in function, call graph, ...

# Flat profile

Flat profile: Each sample counts as 0.01 seconds.

% cumulative	self	self	self	total		
time seconds	seconds	calls	ms/call	ms/call	name	
33.34	0.02	0.02	7208	0	0	open
16.67	0.03	0.01	244	0.04	0.12	offtime
16.67	0.04	0.01	8	1.25	1.25	memccpy
16.67	0.05	0.01	7	1.43	1.43	write
16.67	0.06	0.01				mcount
0.00	0.06	0.00	1	0.00	50.00	main

# Call Graph

```
index % time self children called name <spontaneous>
[1]   100.0  0.00    0.05      start [1]
      0.00    0.05  1/1    main [2]
```

# gprof

- Profiling also involves watching your program as it runs, and keeping a histogram of where the program counter happens to be every now and then. Typically the program counter is looked at around 100 times per second of run time, but the exact frequency may vary from system to system.
- A special startup routine allocates memory for the histogram and sets up a clock signal handler to make entries in it. Use of this special startup routine is one of the effects of using ``gcc ... -pg'` to link. The startup file also includes an ``exit'` function which is responsible for writing the file ``gmon.out'`.

# gprof

- Number-of-calls information for library routines is collected by using a special version of the C library. The programs in it are the same as in the usual C library, but they were compiled with ``-pg'`. If you link your program with ``gcc ... -pg'`, it automatically uses the profiling version of the library.
- The output from gprof gives no indication of parts of your program that are limited by I/O or swapping bandwidth.

# POSIX XSI

```
struct timeval  
time_t tv_sec; /* seconds since the Epoch*/  
time_t tv_usec /* and microseconds*/
```

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

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

tzp is null, historical

# Measure running time using gettimeofday (R&R P.307)

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

void function_to_time(void);

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

    if (gettimeofday(&tpstart, NULL)) {
        fprintf(stderr, "Failed to get start time\n");
        return 1;
    }
}
```

# Measure running time using gettimeofday

```
function_to_time(); /* timed code goes here */  
if (gettimeofday(&tpend, NULL)) {  
    fprintf(stderr, "Failed to get end time\n");  
    return 1;  
}
```

```
timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +  
          tpend.tv_usec - tpstart.tv_usec;  
printf("The function_to_time took %ld microseconds\n", timedif);  
return 0;  
}
```

# Gettimeofday Limitations

- Resolution small number of microseconds
- Many consecutive calls; Will return same value

# A program to test the resolution of gettimeofday (R&R P.308)

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000L
#define NUMDIF 20

int main(void) {
    int i;
    int numcalls = 1;
    int numdone = 0;
    long sum = 0;
    long timedif[NUMDIF];
    struct timeval tlast;
    struct timeval tthis;
    if (gettimeofday(&tlast, NULL)) {
        fprintf(stderr, "Failed to get gettimeofday\n");
        return 1;
    }
}
```

# A program to test the resolution of gettimeofday

```
while (numdone < NUMDIF) {
    numcalls++;
    if (gettimeofday(&tthis, NULL)) {
        fprintf(stderr, "Failed to get a later gettimeofday.\n");
        return 1;
    }
    timedif[numdone] = MILLION*(tthis.tv_sec - tlast.tv_sec) +
                       tthis.tv_usec - tlast.tv_usec;
    if (timedif[numdone] != 0 {
        numdone++;
        tlast=tthis;
    }
}
```

.....

# Summary

- Introduction to POSIX Timers
- POSIX:TMR
  - clock\_gettime
- POSIX: XSI
  - gettimeofday
- Measuring time