Signals & Timers

CS241 Discussion Section
Fall 2009 (Oct. 8, 2009)
Week 6

Outline

- Announcements
- MP4
- Signals: processes and threads
- Blocking & waiting on signals
- Handling signals
- Timers

Announcements

- Homework #1
  - Due at the beginning of class (11:00am) on Monday, October 12th!
  - Solutions will be discussed in class, so no late submissions will be accepted at all. (Sorry!)

Announcements

- Midterm Exam is **6 days from now**!
  - TA review session tentatively planned for Monday night at 7:00pm
    - See our post in the newsgroup
    - Final time/place will be announced on the newsgroup by Monday morning
  - Due to the exam and our review session, there will be **no section next week**.

MP4

Multiple Clients / One Server

In MP #3, the general idea was a client/server model where one server interacted with N clients.

Major Problems we saw:

Clients must run simultaneously

- Launch all threads in one for loop
- Join all threads in another for loop
- DON'T do _create() and _join() in the same loop!

Semaphores must be initialized to the correct value

Review: signals

Asynchronous notification to a process indicating some action should be taken

Sending signals to a process:

```
kill -<signal> <pid>
int kill(pid_t pid, int sig);
```

We can signal individual threads, too:

```
int pthread_kill(thread_t tid, int sig);
```

What can we do with signals?

- Handle them!
  - Default or Ignore
  - Custom function with sigaction
- Block them!
  - Delay delivery with masks
  - Then we can sigwait to get them.

Lots of signal functions

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset)
int sigaction(int signo, const struct sigaction *act, struct sigaction *oact);
int sigwait(const sigset_t *restrict sigmask, int *restrict signo);
```

Process Signal Masks

Setting SIGINT to be blocked

```
if ((sigemptyset(&set) == -1) ||
          (sigaddset(&set, SIGINT) == -1))
   perror("Failed init signal set");
else if
(sigprocmask(SIG_BLOCK, &set, &oldset) == -1)
   perror("Failed to block SIGINT");
```

`SIG_BLOCK` adds `set` to current mask

`oldset` will store the previous signal mask

Thread Signal Masks

`pthread_sigmask()`:

Takes same parameters as `sigprocmask`

Only affects the signal mask of a single thread

Signal mask is inherited on thread creation

Signal Handlers

Allow us to change what happens when a signal is received

```
void handler(int signo) { … }
struct sigaction act;

act.sa_flags = 0;
act.sa_handler = handler;
// additional signals blocked in the handler
sigemptyset(&act.sa_mask);
sigaction(SIGUSR1, &act, NULL);
```

sa_handler can also be set to SIG_DFL (default) or SIG_IGN (ignore)

sa_handler vs. sa_sigaction

We can get additional information about the signal

```
void handler(int signo, siginfo_t* info,
  void* context);
act.sa_flags = SA_SIGINFO;
// fill sa_sigaction instead of sa_handler
act.sa_sigaction = handler;
```

Extra information contains, e.g., the source of the signal (`info->si_code`):

`SI_USER` – user-created signal (with abort, kill, etc.)

`SI_TIMER` – a POSIX:RTS timer expired

etc.

Example 1: Return of Hello World

See d6-example1.c and d6-example1-soln.c

Another "hello world" variant

Prints out "Hello" and "World" periodically

Let's change the code to stop the hello thread when receiving `SIGUSR1` and the world thread when receiving `SIGUSR2.`

Hint 1: we do not have to change the signal handler

Hint 2: what about the main thread?

Example 3 (d6-example3.c)

Windows users will sometimes press Ctrl-C (for copy to clipboard), inadvertently killing a UNIX process.

Let's change the `SIGINT` handler to kill the process only if the user *really* means it!

I.e., presses Ctrl-C three times within a 5-second "tick"

Hint: Use sa_sigaction

And let's only count signals sent by the kernel (based on keyboard input)

```
info->si_code == SI_KERNEL
```

pause()

Waits for *any* signal that is not blocked/ignored

But if a signal is generated before pause() is called, pause() will never see it

If we use the sigmask to block the signal until pause() is called, it will be queued until we remove it

However, pause() will just sit there waiting for the signal that is blocked; it will never check the queue

In summary: pause() only returns if called before the signal is generated!

sigwait()

Takes as parameter a sigset corresponding to which signals it should wait for

You should block the signals first

sigwait() will remove a signal from the queue that is in its sigset

Must also pass a pointer to an integer for it to store signal that was removed

```
sigwait(sigset_t *set, int *signo);
```

Example 4 (d6-example4.c)

Counting signals

Use sigwait() to count how many times a process receives `SIGUSR1` or `SIGUSR2`

Don't forget to block them first!

Timers

Using POSIX:TMR timers…

Send the SIGALRM signal to the process

If we set up a signal handler for SIGALRM, we have a programmable timer!

!! Signals sent for timers or interrupts need to be unblocked for the thread that will be receiving them !!

Accessing the clock

The POSIX:TMR extension allows us to get and set time from the real-time clock

```
struct timespec {
   time_t tv_sec;  /* seconds */
   long   tv_nsec; /* nanoseconds */ }
```

Timers need two of these: one for how long from now to begin, another for how often to generate the interrupt (timer interval)

```
struct itimerspec {
   struct timespec it_interval; /* period */
   struct timespec it_value; }
```

Setting a timer

Create the timer

```
timer_t timerid;
timer_create(CLOCK_REALTIME, NULL, &timerid);
```

Set up the structs (fire first at 5 s, then every 2.5 s)

```
struct itimerspec value;
value.it_interval.tv_sec = 2;
value.it_interval.tv_nsec = 500000000L;
value.it_value.tv_sec = 5;
value.it_value.tv_nsec = 0;
```

Start the timer

```
timer_settime(timerid, 0, &value, NULL);
```

Resetting (or disabling) a timer

How do we turn off a timer?
Simple: just “set” it to 0

We can also restart a timer
Just call `timer_settime` with the same parameters as before; the timer will be reset
Or pass in a different time to change its behavior

Example 2: Tick tock

See d6-example2.c
Let’s see how timers are used in this simple example program

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

```
gcc –o d6-example2 d6-example2.c
  –lrt
```

Timing your code

clock_gettime() fills in a struct timespec with elapsed time in nanoseconds since the epoch (Jan 1, 1970)
Difference between two structs can time a function/action
Useful to keep track of how long threads are waiting or executing

```
struct timespec tend, tstart;
clock_gettime(CLOCK_REALTIME, &tstart);
function_to_time();
clock_gettime(CLOCK_REALTIME, &tend);
double timediff = tend.tv_sec – tstart.tv_sec
  +((double)tend.tv_nsec – tstart.tv_nsec)/1e9;
```

Example 5 (d6-example5.c)

Time how long 1e5, 1e6, etc. iterations of an empty for loop take to execute

Time how long 1e5, 1e6, etc. sched_yields take to execute