

---

# CS 241 System Programming

## Timers & Signals

---

Discussion Section 5

20 Feb – 23 Feb

---

# Outline

- Review: Scheduling
  - Signals
  - Signal/Timer Handlers
  - `pause()`, `sigwait()`
  - POSIX:TMR Timers
  - `clock_gettime()`
  - `nanosleep()`
-

---

# Scheduling (methods for MP2)

- Scheduler is another process that runs and schedules the other processes on a system
    - Round-robin – When the scheduler runs, it suspends the current thread, and starts the one at the front of the queue.
    - Priority – Except when preempted, scheduler keeps the present thread running until it is done. When a thread of higher priority enters the queue, next time the scheduler runs, that thread will preempt the current thread.
-

# Scheduling Example

- Schedule the following threads with quantum of 2
  - Scheduler only runs at quantum, so threads can overrun
  - What happens under Round-Robin scheduling? Priority?
    - Average wait time, turn-around time?
  - What effect does preemption have?

Process	Duration	Priority #	Arrival Time
P1	6	4	2
P2	4	1	1
P3	7	2	6
P4	5	3	0

# Signals

(Table 8.1)

- System signals sent to processes/threads indicating some action should be taken
  - CTRL-C sends SIGINT
  - Division by Zero sends SIGFPE
  - SIGUSR1, SIGUSR2 we can define ourselves
- Unix `kill` sends signals to processes
  - `kill -s <signal> <pid>`
- C function `pthread_kill` sends to threads
  - `pthread_kill(thread_t tid, int signo)`

# Signal Masks in Processes

(Section 8.3, Example 8.10)

- **Setting SIGINT to be blocked:**

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

- **SIG\_BLOCK** adds `set` to current mask
- **SIG\_UNBLOCK** removes `set` from current mask
- **SIG\_SETMASK** sets current mask to be `set`
- `oset` will store the previous signal mask

---

# Signal Masks in Threads

(Section 13.5)

- `pthread_sigmask`
    - Takes same parameters as `sigprocmask`
    - Only has effect on the sigmask for a single thread
    - Signal masks inherited during thread creation
  - Need to use in MP2
    - Block the timer signal in non-scheduler threads
    - Block resume signal in non-scheduler threads
    - (potentially) Block non-timer signals in scheduler thread
-

---

# Effect of a Generated Signal

- Table 8.1 shows default actions for signals
  - SIGKILL/SIGSTOP cannot be changed
  - Every other signal we pick what action to take
    - Let it take default action
    - Block/ignore it indefinitely (using sigmask)
    - Setup a signal handler for it
      - Function that is executed when the process/thread receives the signal
      - `sigaction`
-

# Setting up Signal Handlers

(Section 8.4)

- Use a struct sigaction
- Similarly to threads, we need to craft a function with a particular signature
  - `sa_handler = handler, sa_flags = 0`  
`void handler(int signo)`
  - `sa_sigaction = handler, sa_flags = SA_SIGINFO`  
`void handler(int signo, siginfo_t*, void* context)`
- Can also specify a set of additional signals to block while executing the handler
  - `sa_mask`
    - Not necessary for MP2

---

# sa\_handler vs sa\_sigaction

(Section 9.4)

- Cannot use both the handler and the sigaction inside the struct
  - Both handlers are called in the same manner, but the latter receives extra information
    - Cause of the signal (info->si\_code)
      - SI\_USER – user created signal with raise/abort/kill/etc
      - SI\_TIMER – a POSIX:RTS timer expired
      - etc
-

---

# Setting up Signal Handlers

(Example 8.16)

## ■ Catching CTRL-C and running handler

```
void catchctrlc(int signo) {
    write(STDERR_FILENO, "CTRL-C\n", 7);
}

struct sigaction act;
act.sa_handler = catchctrlc;
act.sa_flags = 0;
if ((sigemptyset(&act.sa_mask) == -1) ||
    (sigaction(SIGINT, &act, NULL) == -1))
    perror("Failed to set handler for CTRL-C");
```

---

---

# Another way to ignore signals

(Example 8.15, Example 8.18)

- There are a few special values for `sa_handler`
    - `SIG_DFL` represents the default action (Table 8.1)
    - `SIG_IGN` will make the process ignore the signal
      - Not too useful with threads, as the signal will be ignored for all threads
        - Need to use `pthread_sigmask` as we did earlier
  - Can use `sigaction` to get old `sigaction`
    - As in example 8.15, if the handler was previously the default handler, now set the signal to be ignored
-

---

# Timer Handlers

- We will be using POSIX:TMR timers
    - By default they send the SIGALRM signal
    - Setup a signal handler for SIGALRM
      - Bam! Now we have a timer handler.
  - !! Signals sent for timers or interrupts need to be unblocked for the thread that will be receiving them !!
    - Or we can use a special function, sigwait()
-

---

# pause()

(Section 8.5.1, Exercise 8.21, Exercise 8.22)

- Waits for any signal that is not blocked/ignored
  - If a signal is generated (and does not terminate the process) before pause() is called, pause() will never see it
  - If we use sigmask to block the signal until pause() is called, it will be queued until we remove it
    - However, pause() will sit waiting for the signal that is blocked; it will never check the queue
    - pause() only returns if called before the signal
-

---

# Enter sigwait()

(Section 8.5.3)

- Takes as parameter a sigset corresponding to which signals it should wait for
    - Block the signals first
    - sigwait() will remove a signal from the queue that is in its sigset
  - Must also pass a reference to an integer for it to store signal that was removed
    - CANNOT PASS NULL
  - `sigwait(sigset_t *set, int *signo)`
-

---

# Counting signals

(Program 8.11)

```
int main(void) {
    int signalcount = 0, signo, signum = SIGUSR1;
    sigset_t sigset;

    if ((sigemptyset(&sigset) == -1) ||
        (sigaddset(&sigset, signum) == -1) ||
        (sigprocmask(SIG_BLOCK, &sigset, NULL) == -1))
        perror("Failed to block signals before sigwait");
    fprintf(stderr, "This process has ID %ld\n", (long)getpid());
    for ( ; ; ) {
        if (sigwait(&sigset, &signo) == -1) {
            perror("Failed to wait using sigwait");
            return 1;
        }
        signalcount++;
        fprintf(stderr, "Number of signals so far: %d\n", signalcount);
    }
}
```

---

---

# Accessing the Clock

- The POSIX:TMR extension allows us to get and set time from `CLOCK_REALTIME`

- ```
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 up the Timer

(Example 9.16)

- **Create the timer**

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

- **Set up the structs (first at 10 seconds, then every 3.5)**

```
struct itimerspec value;  
value.it_interval.tv_sec = 3;  
value.it_interval.tv_nsec = 500000000L;  
value.it_value.tv_sec = 10;  
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: “set” it to 0

```
struct itimerspec offtime;  
value.it_interval.tv_sec = 0;  
value.it_interval.tv_nsec = 0;  
value.it_value.tv_sec = 0;  
value.it_value.tv_nsec = 0;  
timer_settime(timerid, 0, &offtime, NULL);
```

- We can also restart a timer

- Just call `timer_settime` with the same parameters as before, timer will be reset
    - Or pass in a different time to change the value
-

---

# Using the timer effectively (MP2 advice)

- POSIX:TMR timers by default send `SIGALRM`
    - Use `struct sigevent` to change this if needed
    - Setup a signal handler with `sigaction` to catch the signal
  - Timer doesn't start counting until `timer_settime`
  - Use infinite `pause` or `sched_yield` to sleep your scheduler thread while it waits for the timer
    - `RunThreads` will actually be creating new threads until it has created an arbitrary number, then you need to do this
    - If your scheduler thread terminates while the timer is still armed, the signal may be delivered to a different thread (if not ignored/blocked), or dropped completely
-

---

# clock\_gettime()

(Example 9.8)

- Fills in a struct timespec with elapsed time since the epoch
  - Difference of two times can measure a function/action
  - Useful to keep track of how long threads are waiting or executing (needed for MP2 evaluation)

```
struct timespec tpend, tpstart;
clock_gettime(CLOCK_REALTIME, &tpstart);
function_to_time();
clock_gettime(CLOCK_REALTIME, &tpend);
double timedif = tpend.tv_sec - tpstart.tv_sec
    + (double)(tpend.tv_nsec -
    tpstart.tv_nsec)/1000000000.0;
```

---

---

# nanosleep()

(Section 9.2)

- Uses same `struct timespec` we use for timers and `clock_gettime`
- First parameter specifies how long to sleep
- Second parameter returns how much time was remaining
  - Returns -1 when it exits early and this value has been set
  - We'll use this value to restart the `nanosleep`

```
struct timespec try, remains;
try.tv_sec = 2; try.tv_nsec = 0.5;
while (nanosleep(&try, &remains) == -1) {
    try.tv_sec = remains.tv_sec;
    try.tv_nsec = remains.tv_nsec;
}
```

---

---

# Recap

- How to generate a signal
  - How to use signal masks properly
  - How to wait for a signal
  - How to initialize and start a timer
  - How to time a function or action
-