

18 Lecture: Reliable Signal Handling and Timers

Outline:

- Announcements
- The rest of the quarter
 - Subjects
 - interaction
- Critical Sections
- Communicating with signal handlers
- From Last Time: Unreliable Signal Handling: `signal(2)`
- Reliable Signal Handling (POSIX)
- Signal Example
- Signal generation: Alarm clocks
- `alarm(2)`
- `setitimer(2)`
- `sleep(3)`, `pause(2)`, and `sigsuspend(2)`
 - `sleep(3)`
 - `pause(2)`
 - `sigsuspend(2)`
- Unwanted Interactions: `sleep(3)`
- Mid(end?)term
 - Style
 - Resources
- Mid(end?)term
 - Style
 - Resources
 - Material
- Mid(end?)term
 - Style
 - Resources
 - Material

18.1 Announcements

- Coming attractions:

Event	Subject	Due Date			Notes
asgn4	mytar	Mon	Nov 27	23:59	
asgn5	mytalk	Fri	Dec 1	23:59	
lab07	forkit	Mon	Dec 4	23:59	
asgn6	shell	Fri	Dec 8	23:59	

Use your own discretion with respect to timing/due dates.

- Dumb mypwd: 4:16.56
- Midterm coming
- Be working on Asgn4 (now's the chance to make it good.)
- Lab05:mypwd() due today

```

Possible: 60.0
    High: 60.0    (100.0%)
    Low: 10.0    ( 16.7%)
    Mean: 43.1    ( 71.8%)
    Median: 45.0  ( 75.0%)
    S.D.: 10.4    ( 17.4%)

```

```

Enrolled:    60
Submitted:   49    ( 81.7%)

```

```

lagniappe% gradescale 50 12.5
Max Possible: 50.0
    Buckets: 12.5% of 50.0

```

```

Min A-: 43.8  (87.5%)
Min B-: 37.5  (75.0%)
Min C-: 31.2  (62.5%)
Min D-: 25.0  (50.0%)

```

18.2 The rest of the quarter

18.2.1 Subjects

- Signals
- tty line disciplines (IO device management) (maybe)
- Process Management (fork(), the exec()s, process groups, etc.)
- Process Environments (env, resource limits)
- Other useful stuff (strtok(), setjmp(), longjmp())

The next two subjects (Terminal IO and Signals) show the most differentiation between the UNIX family members.

18.2.2 interaction

Communication with signal handlers must be done through globals or functions with side-effects.

18.3 Critical Sections

Sometimes a program just has to say “not now”...

(E.g. updating a linked list and a handler writes the list)

Can ignore the signal (might miss it) or set a flag for later(tricky). (POSIX lets the signal be blocked and delivered later...tune in later.)

One possible approach is to change the signal handler to something else while handling one instance.

18.4 Communicating with signal handlers

Communicating with signal handlers is a little crude because a signal handler can't return anything since it doesn't know when it's being called.

Requires the use of globals...

18.5 From Last Time: Unreliable Signal Handling: `signal(2)`

```
void (*signal(int signum, void (*sighandler)(int)))(int);
```

On delivery:

- signal handler is called with the signal number
- (maybe) the handler is reset to `SIG_DFL`
- (maybe) the further deliver of the same signal is blocked
- (maybe) slow system calls are interrupted, returning error and setting `errno` to `EINTR`

Signal Generation

- From Hardware: `SIGILL`, `SIGPWR`, `SIGFPE`
- From Software: via `kill(1)`, `kill(2)`, `killpg(2)`, `raise(3)`

```
#include <sys/types.h>
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
int raise (int sig);
```

- From Terminal Line discipline (non-canonical (cooked) mode processing)

18.6 Reliable Signal Handling (POSIX)

Components that are of interest:

- **sigaction** — reliable signal handling
- masking
- generation.
- waiting (`sigsuspend()`, `pause()`)

Look into `sigaction(2)` and `sigsetops(2)`. (See Figure 75.) The reliable signal semantics allows for signal handlers to be installed until told otherwise. By default, when a signal is caught, further instances are blocked (and possibly queued) until the handler completes.

The POSIX semantics provide support for all sorts of tuning one might want to do, but this makes the interface more complicated.

sigaction
<code>int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);</code> <code>int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);</code>
sigsetops
<code>int sigemptyset(sigset_t *set);</code> <code>int sigfillset(sigset_t *set);</code> <code>int sigaddset(sigset_t *set, int signum);</code> <code>int sigdelset(sigset_t *set, int signum);</code> <code>int sigismember(const sigset_t *set, int signum);</code>
Other
<code>int sigsuspend(const sigset_t *mask);</code> <code>int pause(void);</code>

Figure 75: System calls associated with POSIX signal handling

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
}
```

The `siginfo_t` parameter to `sa_sigaction` is a struct with the following elements

```
siginfo_t {
    int      si_signo; /* Signal number */
    int      si_errno; /* An errno value */
    int      si_code;  /* Signal code */
    pid_t    si_pid;   /* Sending process ID */
    uid_t    si_uid;   /* Real user ID of sending process */
    int      si_status; /* Exit value or signal */
}
```

```

        clock_t si_utime; /* User time consumed */
        clock_t si_stime; /* System time consumed */
        sigval_t si_value; /* Signal value */
        int si_int; /* POSIX.1b signal */
        void * si_ptr; /* POSIX.1b signal */
        void * si_addr; /* Memory location which caused fault */
        int si_band; /* Band event */
        int si_fd; /* File descriptor */
    }

```

(sa_sigaction is active if SA_SIGINFO flag is set) Example flags:

```

SA_RESTART    restart interrupted system calls
SA_RESETHAND  this handler only handles once
SA_SIGINFO    use the siginfo parameter
SA_NODEFER    Does not block signals during handling

```

SA_RESETHAND and SA_NODEFER is equivalent to the old unreliable semantics.

With sigaction():

sigprocmask can have three possible values of *how*:

```

SIG_BLOCK     The set of blocked signals is the union of the current set and the
               set argument.
SIG_UNBLOCK   The signals in set are removed from the current set of blocked
               signals. It is legal to attempt to unblock a signal which is not
               blocked.
SIG_SETMASK   The set of blocked signals is set to the argument set.

```

18.7 Sigaction Example

See Figures 76 and 77 and 78.

Note the difference between “pause()” and “for(;;)”

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
    /* Handler for both SIGINT and SIGQUIT */
    fprintf(stderr,"Caught signal %d.    That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
    struct sigaction sa;

    sa.sa_handler      = handler; /* set up the handler */
    sigemptyset(&sa.sa_mask); /* mask nothing */
    sa.sa_flags    = 0;          /* no special handling */

    caught = 0;

    if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
        perror("sigaction");
        exit(-1);
    }

    if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
        perror("sigaction");
        exit(-1);
    }

    /* wait for a signal */
    while ( caught < 3 )
        pause();          /* wait for a signal */

    return 0;
}

```

Figure 76: A program that catches SIGINT and SIGQUIT using sigaction

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <signal.h>

static int caught;

void handler(int signum){
    /* Handler for both SIGINT and SIGQUIT */
    fprintf(stderr,"Caught signal %d.    That's %d.\n", signum, ++caught);
}

int main(int argc, char *argv[]){
    struct sigaction sa;
    sigset_t mask;

    sa.sa_handler      = handler; /* set up the handler */
    sigemptyset(&sa.sa_mask); /* mask nothing */
    sa.sa_flags = 0;          /* no special handling */

    caught = 0;

    if ( -1 == sigaction(SIGINT, &sa,NULL) ) {
        perror("sigaction");
        exit(-1);
    }

    if ( -1 == sigaction(SIGQUIT, &sa,NULL) ) {
        perror("sigaction");
        exit(-1);
    }

    /* wait only for SIGQUIT */
    sigfillset(&mask);
    sigdelset(&mask, SIGQUIT);
    while ( caught < 3 )
        sigsuspend(&mask); /* wait for a signal */

    return 0;
}

```

Figure 77: A program that catches SIGINT and SIGQUIT using sigaction using sigsuspend()

```
% one
^C
Caught signal 2. That's 1.
^\
Caught signal 3. That's 2.
^C
Caught signal 2. That's 3.

% two
^C^\
Caught signal 3. That's 1.
Caught signal 2. That's 2.
^C^\
Caught signal 3. That's 3.
Caught signal 2. That's 4.
%
```

Figure 78: The output from these programs

18.8 Signal generation: Alarm clocks

18.9 alarm(2)

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

`alarm(2)` is not very flexible, but it is a simple interface. after `seconds` seconds have elapsed a `SIGALRM` is sent.

The process is not guaranteed to be scheduled immediately, though, and timing errors can compound (illustrate clock drift).

Make sure to illustrate the problem of clock drift with repeated alarms, relative to `mytimer`

`alarm(0)` clears alarms.

`alarm()` returns the number of seconds remaining.

18.10 setitimer(2)

The interval timer is finer-grained and more accurate:

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

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value,
              struct itimerval *ovalue);

struct itimerval {
    struct timeval it_interval; /* next value */
    struct timeval it_value;    /* current value */
};

struct timeval {
    long tv_sec;                /* seconds */
    long tv_usec;               /* microseconds */
};
```

The `usec` number is optimistic, however. It depends on the system clock's resolution. (linux box: 10ms)

It comes in three flavors:

<code>ITIMER_REAL</code>	decrements in real time, and delivers <code>SIGALRM</code> upon expiration.
<code>ITIMER_VIRTUAL</code>	decrements only when the process is executing and delivers <code>SIGVTALRM</code> upon expiration.
<code>ITIMER_PROF</code>	decrements both when the process executes and when the system is executing on behalf of the process. Coupled with <code>ITIMER_VIRTUAL</code> , this timer is usually used to profile the time spent by the application in user and kernel space. <code>SIGPROF</code> is delivered upon expiration.

18.11 sleep(3), pause(2), and sigsuspend(2)

18.11.1 sleep(3)

SYNOPSIS

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds);
```

DESCRIPTION

sleep() makes the current process sleep until seconds seconds have elapsed or a signal arrives which is not ignored.

RETURN VALUE

Zero if the requested time has elapsed, or the number of seconds left to sleep.

18.11.2 pause(2)

```
#include <unistd.h>
```

```
int pause(void);
```

DESCRIPTION

The pause library function causes the invoking process (or thread) to sleep until a signal is received that either terminates it or causes it to call a signal-catching function.

18.11.3 sigsuspend(2)

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

DESCRIPTION

The sigsuspend call temporarily replaces the signal mask for the process with that given by mask and then suspends the process until a signal is received.

RETURN VALUE

The function sigsuspend always returns -1, normally with the error EINTR.

18.12 Unwanted Interactions: sleep(3)

Flying at different levels can be dangerous: when you tweak system-dependent features, you may upset other implementations: The Solaris sleep(3) function uses the interval timer. You can't use both. You can use pause(2).

pause(2) means “wait for a signal”

18.13 Mid(end?)term

The midterm will be Monday, May 25th.

18.13.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.13.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.14 Mid(end?)term

The midterm will be Monday, May 25th.

18.14.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.14.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.14.3 Material

- The C Programming Language (background noise)
 - The Basics: Type system, flow control, calling protocols, etc.
 - Structures and types
 - Memory management: malloc() and free() and traversing dynamic data structures.
 - Preprocessor controls, headers, etc.
- Unix Systems Programming
 - System architecture:
 - * identity (users and groups (uid, gid, euid, egid))
 - * programs
 - * processes
 - * the filesystem
 - * IO (buffered vs. not and the issues involved)
 - * ipc
 - * system calls
 - The Unix Family: (Chapter 2 stuff)

- Unbuffered IO – the basic system calls and how they work
 - * open, close, creat, read, write, lseek
 - * dup, dup2
 - * fcntl() and ioctl()
- The filesystem: Directories and structure
 - * Structure of the filesystem (inodes and whatnot)
 - * opendir(), readdir(), etc.
 - * the stats
 - * link, unlink, symlink, readlink
 - * identity
 - * How times work

18.15 Mid(end?)term

The midterm will be Monday, May 25th.

18.15.1 Style

My usual midterm style is short-answer questions for breadth and a couple of longer ones to demonstrate depth of knowledge. I try to emphasize understanding over rote knowledge, but bear in mind that in a field such as this two are hard to separate.

18.15.2 Resources

- Any necessary system call prototypes will be listed on the exam. (Unnecessary ones, too.)

18.15.3 Material

- The C Programming Language (background noise)
 - The Basics: Type system, flow control, calling protocols, etc.
 - Structures and types
 - Memory management: malloc() and free() and traversing dynamic data structures.
 - Preprocessor controls, headers, etc.
- Unix Systems Programming
 - System architecture:
 - * identity (users and groups (uid, gid, euid, egid))
 - * programs
 - * processes
 - * the filesystem
 - * IO (buffered vs. not and the issues involved)
 - * ipc
 - * system calls
 - The Unix Family: (Chapter 2 stuff)

- Unbuffered IO – the basic system calls and how they work
 - * open, close, creat, read, write, lseek
 - * dup, dup2
 - * fcntl() and ioctl()
- The filesystem: Directories and structure
 - * Structure of the filesystem (inodes and whatnot)
 - * opendir(), readdir(), etc.
 - * the stats
 - * link, unlink, symlink, readlink
 - * identity
 - * How times work