Lecture - X
Signals -II

Tevfik Koşar
Louisiana State University
October 5th, 2010

Signals from a Process

#include <signal.h>

int kill(pid_t pid, int sig);

If pid > 0: Sig is sent to the process with pid
If pid = 0: Sig is sent to all processes whose group ID is
        equal to the process group ID of the sender
If pid = -1: If root, sig is sent to all processes excluding
        system processes
        If not root, sig is sent to all processes with
        the same uid as the user, excluding the sender.
        sig=0; error checking, i.e. validity of pid

Signal Semantics

• A signal is pending if it has been sent but not yet
  received.
  – There can be at most one pending signal of any particular type.
  – Signals are not queued!
• A process can block the receipt of certain signals.
  – Blocked signals can be delivered, but will not be received until
    the signal is unblocked.
  – There is one signal that can not be blocked by the process.
    (SIGKILL)
• A pending signal is received at most once.
  – Kernel uses a bit vector for indicating pending signals.

Implementation

• Kernel maintains pending and blocked bit vectors in
  the context of each process.
  • pending – represents the set of pending signals
    » Kernel sets bit k in pending whenever a signal of type k is
      delivered.
    » Kernel clears bit k in pending whenever a signal of type k is
      received.
  • blocked – represents the set of blocked signals
    » Can be set and cleared by the application using the sigprocmask
      function.

Receiving Signals

• Handling signals
  • Suppose kernel is returning from exception handler and
    is ready to pass control to process p.
  • Kernel computes pnb = pending & ~blocked
    – The set of pending nonblocked signals for process p
  • if (pnb != 0) {
    – Choose least nonzero bit k in pnb and force process p to
      receive signal k.
    – The receipt of the signal triggers some action by p.
    – Repeat for all nonzero k in pnb.
  } 
• Pass control to next instruction in the logical flow for p.

Overlapping Signals

• SIGY interrupts SIGX
  • ex: phone then door
  • When you press CTRL-C then CTRL-\, the program first jumps to
    inthandler, then to quithandler, then back to inthandler, then
    back to main loop.
• SIGX interrupts SIGY
  • ex: two people coming to your door
  • Three ways this can be handled:
    1. Recursively call the same handler
    2. Ignore the second signal, like a phone without call waiting
    3. Block the second signal until done handling the first
  • Method 3 is safest and default in modern systems
• Interrupted System Calls
  • receiving a signal while waiting for input
Example

```c
main(int ac, char *av[])
{
    void inthandler(int);
    void quithandler(int);
    char input[100];

    signal( SIGINT, inthandler ); // set trap
    signal( SIGQUIT, quithandler ); // set trap

    int i=1;
    while (i<5){
        sleep(1);
        printf("main:%d
",i++);
    }
}
```

Ignore other Intermittent Intermittent Handlers

```c
void quithandler(int s)
{
    printf("Received signal %d .. waiting
", s );
    ....
    printf("Leaving quithandler \n");
}

void inthandler(int s)
{
    signal(SIGQUIT, SIG_IGN);
    printf("Received signal %d .. waiting
", s );
    ....
    printf("Leaving inthandler \n");
    signal( SIGQUIT, quithandler );
}
```

Masking Signals - Avoid Race Conditions

- The occurrence of a second signal while the signal handler function executes.
  - The second signal can be of different type than the one being handled, or even of the same type.
- The system also contains some features that will allow us to block signals from being processed.
  - A global context which affects all signal handlers, or a per-signal type context.

Masking Signals (cont.)

- Each process maintains a signal mask which controls which signals are immediately delivered to the process and which have delivery deferred.
- If a signal is in the signal mask, it is said to be blocked.
- The signal mask for a process is initially empty, which means any signal can be delivered.
- Some signals (in particular, SIGINT and SIGABRT) cannot be deferred. Including them in a signal mask is not an error, but will not be effective.
- Blocking a signal is a temporary measure; don’t confuse it with ignoring (with sig_ign) a signal.

sigprocmask() Function

- The system call allows to specify a set of signals to block, and returns the list of signals that were previously blocked.

```c
sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
```

1. int how:
   - ADD(SIG_BLOCK)
   - DELETE(SIG_UNBLOCK)
   - SET(SIG_SETMASK).
2. const sigset_t *set:
   - The set of signals.
3. sigset_t *oldset:
   - If this parameter is not NULL, then it’ll contain the previous mask.

Manipulating Signal Sets

The `sig_mask` component of the `sigaction` structure contains a set of signals. This set is modified using the following functions (or macros):

- `sigemptyset(sigset_t *set);` -- init to no signals
- `sigfillset(sigset_t *set);` -- init to all signals
- `sigaddset(sigset_t *set, int sig);` -- add signal
- `sigdelset(sigset_t *set, int sig);` -- remove signal
- `sigismember(sigset_t *set, int sig);` -- check signal
Block other Interrupts inside Handler

```c
void inthandler(int s)
{
    sigset_t sigset, sigoldset;
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGQUIT);
    sigprocmask(SIG_SETMASK, &sigset, &sigoldset);
    printf("Received signal %d .. waiting\n", s);
    ....
    printf("Leaving SIGINT Handler \n");
    sigprocmask(SIG_SETMASK, &sigoldset, NULL);
}
```

**sigaction() Function**

- The **sigaction()** function allows the calling process to examine and/or specify the action to be associated with a specific signal.

```c
int sigaction(int sig, struct sigaction *new_act, struct sigaction *old_act);
```

**sigaction() Function (cont.)**

- This function is "newer" than **signal**, and provides considerably more flexibility.
- Like **signal**, the first argument is a signal number (or name).
- The second argument is a pointer to a structure containing the new characteristics for the signal; the third argument points to a structure which will receive the old characteristics of the signal. Either or both of these pointers may be **NULL**, allowing any combination of setting or querying the action associated with a signal.

**sigaction Structure**

- **struct sigaction** has the following members:
  - **sa_handler**: set to **SIG_DFL, SIG_IGN**, or pointer to handler function (compare this with the second argument to **signal**).
  - **sa_mask**: a set of additional signals to be blocked during execution of the function identified by **sa_handler**.
  - **sa_flags**: special flags that affect the signal behavior.
  - **sa_sigaction**: (used only for POSIX real-time signals).

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

**sa_flags**

- **SA_NOCLDSTOP**: If signum is SIGCHLD, do not receive notification when child processes stop.
- **SA_NOCLDWAIT**: If signum is SIGCHLD, do not transform children into zombies when they terminate.
- **SA_RESETHAND**: Restore the signal action to the default state once the signal handler has been called.
- **SA_ONSTACK**: Call the signal handler on an alternate signal stack provided by **sigaltstack(2)**.
- **SA_RESTART**: Provide behaviour compatible with BSD signal semantics, by making certain system calls restartable across signals.
- **SA_NODEFER**: Do not prevent the signal from being received from within its own signal handler.
- **SA_SIGINFO**: The signal handler takes 3 arguments, not one. In this case, **sa_sigaction** should be set instead of **sa_handler**.

**sigaction() (cont.)**

- A new signal mask is calculated and installed only for the duration of the signal-catching function, which includes the signal being delivered.
- Once an action is installed for a specific signal, it remains installed until another action is explicitly requested.
sigaction() Example

```c
main()
{
    struct sigaction newhandler; /* new settings */
    sigset_t blocked;          /* set of blocked sigs */
    void inthandler();         /* the handler */

    newhandler.sa_handler = inthandler; /* handler function */
    sigfillset(&blocked);       /* mask all signals */
    newhandler.sa_mask = blocked; /* store blockmask */

    int i;
    for (i=1; i<31; i++)
        if (i!=9 && i!=17) /* catch all except these signals */
            if (sigaction(i, &newhandler, NULL) == -1)
                printf("error with signal %d\n", i);
    while(1)
}
```

Real-time Signals

- **POSIX.4 adds some additional signal facilities.** The key features are:
  - The real-time signals are in addition to the existing signals, and are in the range SIGRTMIN to SIGRTMAX.
  - Real-time signals are queued, not just registered (as is done for non-real-time signals).
  - The source of a real-time signal (kill, sigqueue, asynchronous I/O completion, timer expiration, etc.) is indicated when the signal is delivered.
  - A data value can be delivered with the signal.

sigqueue()

The `sigqueue()` function provides a means to queue a signal, and provide additional signal data. The function fails immediately if queueing is not possible.

```c
int sigqueue(pid_t pid, int signo, const union sigval value);
```

The union `sigval` has the following members:

- `int sival_int;` // a simple integer value
- `void *sival_ptr;` // a pointer

Summary

- **Signals**
  - Generating & Catching Signals
  - Overlapping Signals
  - Preventing Race Conditions
  - Masking Signals

Read Ch 10 from Stevens Book

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), and M. Shcklette (UChicago).