

Signals

What's a Signal?

- A Unix signal is a simple form of a message.
- It usually is reporting the occurrence of an “event”
- The only information in a signal is its type.
- There are ~30 distinct signals.
- Examples:
 - SIGCHLD: a child process has terminated.
 - SIGSEGV: a memory address could not be accessed.
 - SIGKILL: die!

Signal Disposition

- The *disposition* of a signal specifies what should happen if the signal is sent.
- The signal could be ignored.
- It could cause the process to terminate, possibly with a core dump.
- It could be caught resulting in a programmer-supplied function (known as a *signal handler*) being called.
- It could be blocked, meaning that no action is taken for now, but the event is remembered for later when it is *unblocked*.
- All signals have a “default” action, most often terminating the process.

Catching a Signal

```
void myHandler(int);                // declare the handler

int main() {
    int i;
    signal( SIGINT, myHandler );    // install the handler
    for ( i=0; i<20; i++ ) {        // do something else
        printf("hello\n");
        sleep(1);
    }
}

void myHandler(int signum) {
    printf("OUCH!\n");
}
```

Signal Function

- Prototype:

```
void (*)(int) signal(int signum, void (*handler)(int))
```

- ```
typedef void (*sig_handler_t)(int num);
sig_handler_t signal(int signum, sig_handler_t *handler)
```

- Special handlers:

- ```
#define SIG_IGN (void (*)(int)) 1
```
- ```
#define SIG_DFL (void (*)(int)) 0
```
- ```
#define SIG_ERR (void (*)(int)) -1
```

Ignoring a Signal

```
// Convenient type definition for sig_handler_t
typedef void (*sig_handler_t)(int num);

int main() {
    // Remember prior disposition
    sig_handler_t old_handler = signal(SIGINT, SIG_IGN);
    for ( int i = 0; i < 1000; ++i ) {
        printf("hello\n");
        sleep(1);
    }
    // Restore prior disposition
    signal(SIGINT, old_handler);
}
```

Good practice to restore the signal's disposition when done.

Common Signals

- SIGINT: terminal handler detected the “interrupt character”
- SIGQUIT terminal handler detected the “quit character”
- SIGTERM: Terminate the process.
- SIGKILL: Terminate the process. Cannot be caught, ignored or blocked.
- SIGSTOP: Pause the process. Cannot be caught, ignored or blocked.
- SIGCHLD
 - Informs a parent that a child process has terminated.
- SIGFPE
 - Sent due to various arithmetic hardware errors, such as divide by zero.
- SIGPIPE
 - Process tried to write to a pipe after the last reader closed it.
- SIGSEGV
 - Memory violation. I.e. trying to access memory that you don’t have rights to.

Sending a Signal

- `int kill(pid_t pid, int signo);` // `signal.h`
 - `pid > 0`: send signal to process with ID `pid`
- `int raise(int signo);` // `signal.h`
 - Send `signo` to yourself
- `int pause(void)` // `unistd.h`
 - Causes the calling process to sleep until a signal is delivered.

“Unreliable” Signals

- Used to be signals could “get lost”
- And [with some implementations] the action was reset to the signal’s default.
- Couldn’t “block” a signal. Could only handle, ignore or allow default behavior.
- The function **sigaction** replaces signal, providing more reliability and flexibility.

sigaction

- `int sigaction(int signo,
 struct sigaction* act,
 struct sigaction* oldAct)`
- `struct sigaction {
 void (*sa_handler)(int);
 sigset_t sa_mask; // signals to block during handler
 int sa_flags; // other flags
 void (*sa_sigaction)(int, siginfo_t*, void*);
};`
 - `sa_sigaction` allows info about origin of signal
 - man `sigaction` for details of `siginfo_t`
- Common Flags
 - `SA_RESTART`: “Slow” system calls interrupted are restarted.
 - `SA_RESETHAND`: Reset to default
 - `SA_SIGINFO`: Use `sa_sigaction` instead of `sa_handler`.

Blocking Signals

- Sometimes we want to *temporarily* ignore a signal without actually losing it.
- This is called “blocking”
 - `sigprocmask(int how, sigset_t *sigs, sigset_t *prev)`
 - Specifies the “set” of signals to block
 - how?
 - `SIG_BLOCK`: add to the set of signals being blocked
 - `SIG_UNBLOCK`: remove from signals being blocked
 - `SIG_SET`: set these as the signals to block, unblocking any others.
 - Sometimes we want to “suspend” our program with a particular set of signals blocked.
 - `sigsuspend(sigset_t *sigmask)`
 - Set signal mask to `sigmask` and suspend process till a signal is either caught or terminates the process.

sigset_t

- `sigemptyset(sigset_t *setp)`
 - Clear all signals from sigset
- `sigfillset(sigset_t *setp)`
 - Set all signals in sigset
- `sigaddset(sigset_t *setp, int signum)`
 - Add signum to signals in sigset
- `sigdelset(sigset_t *setp, int signum)`
 - Remove signum from signals in sigset

Using signals: sleeping

```
int main() {  
    printf("about to 'sleep' for 4 seconds\n");  
    signal(SIGALRM, wakeup);           // catch it  
    alarm(4);                          // set clock  
    pause();                           // freeze here  
    printf("Morning so soon?\n");      // back to work  
}  
  
void wakeup(int signum) {  
    printf("Alarm received from kernel\n");  
}
```

- alarm schedules a SIGALRM signal to be sent.
- Replaces current alarm
- Returns number of seconds that remained on current alarm, if any.
- Call alarm(0) to “turn off” the alarm
- pause returns after a signal is caught, the handler is executed and returns.

Simple sleep function

```
#include <signal.h>
#include <unistd.h>
static void sig_alrm(int signo){}
unsigned int toDream(unsigned int nsecs) {
    if (signal(SIGALRM, sig_alrm) == SIG_ERR)
        return(nsecs);
    alarm(nsecs);    // start the timer
    pause();         // next caught signal wakes us up
    return alarm(0); // turn off timer, return unslept time
}
```

- What happens if there was an alarm set?
- What was the disposition for SIGALRM?
- What about the race condition?

Sleep continued

- What happens if there was already an alarm set?
 - Sooner than our wakeup? We should sleep the shorter time.
 - Later? Reset the alarm to the “remaining” value when done.
- What was the disposition for SIGALRM?
 - We should reinstate any previous alarm handler.
- What race condition?
 - What happens if the alarm goes off before we pause?
 - Result: we might pause forever.
 - Solution? Don't pause. Instead, (see code for details)
 - block SIGALRM, using sigprocmask
 - Set the alarm
 - sigsuspend.

Jumping

- `#include <setjmp.h>`
- `int sigsetjmp(sigjmp_buf jmp_env, int savemask);`
 - `jmp_env` is commonly a global so it can easily be used from all locations
 - Sets current location as the place from a `siglongjmp` to “return” to.
 - Returns zero when called “directly”
 - `savemask == 0` means do not save / restore the current signal mask otherwise do save / restore the current signal mask.
This is what makes `sigsetjmp/siglongjmp` different from `setjmp/longjmp`.
- `void siglongjmp(sigjmp_buf env, int val);`
 - Returns to the most recent `sigsetjmp` that used the specified `env`.
 - `val` is used as the return value of the corresponding `sigsetjmp`.
 - “Unwinds” the stack.
- Like any “goto”, handle with care.