Signals 6: Advanced Topics

1. Overview
2. System Calls
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. **Advanced Topics**
   - **signals as file descriptors**
   - **timers**
   - **timers as file descriptors**
   - **real-time signals**
   - **non-local jumps**
7. Signal Handler Issues

Signals as File Descriptors

In the past it was difficult to mix (asynchronous) signals with *asynchronous I/O* (done via calls `select()`, `poll()`, `epoll()`).

`signalfd` changed that situation:
`int signalfd(int fd, const sigset_t *mask, int flags)`

- `fd` is set to -1 to create a new FD, else to an existing `signalfd` FD that is to be modified
- `mask` is signal set to accept via this FD
- `flags` (limited options, only recent Linux)

The signals being accepted via `signalfd()` should be *blocked* (e.g., via `sigprocmask()`) to prevent normal signal disposition.

(Note: `signalfd()` is Linux specific.)

Signals as File Descriptors (contd.)

The man page says:
"signalfd() creates a file descriptor that can be used to accept signals targeted at the caller. This provides an alternative to the use of a signal handler or sigwaitinfo(2), and has the advantage that the file descriptor may be monitored by select(2), poll(2), and epoll(2).

If one or more of the signals specified in mask is pending for the process, then the buffer supplied to read(2) is used to return one or more signalfd_siginfo structures that describe the signals. The read(2) returns information for as many signals as are pending and will fit in the supplied buffer.

As a consequence of the read(2), the signals are consumed, so that they are no longer pending for the process (i.e., will not be caught by signal handlers, and cannot be accepted using sigwaitinfo(2))."

Signals as File Descriptors (contd.)

signalfd-based SIGINT handler example:

```
int main(void)
{
  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  sigprocmask(SIG_SETMASK, &mask, NULL);

  int sigfd = signalfd (-1, &mask, 0);

  struct pollfd fds[MAX_FDS];
  fds[0].fd = sigfd;
  fds[0].events = POLLIN;
  ...

  while ((poll(fds,MAX_FDS,-1) > 0) {
    if (fds[0].revents & POLLIN)
      sigint_read_handler(sigfd);

    ...main program code for other FDs...
  }

  return EXIT_SUCCESS;
}
```

Signals as File Descriptors (contd.)

signalfd-based example contd.:

```
void sigint_read_handler(int sigfd)
{
  struct signalfd_siginfo sfd_si;

  if (read (sigfd, &sfd_si, sizeof(sfd_si)) < sizeof(sfd_si))
    return;

   ...handler actions...
  return;
}
```

Timers

Timers in Linux make use of signals.

`alarm` is the simplest timer call:

```
unsigned int alarm(unsigned int seconds)
```

It arranges to have a `SIGALARM` signal sent to the process in a specified number of seconds.

`setitimer` provides more options:

```
int setitimer(int which, const struct itimerval *new_value,
                         struct itimerval *old_value)
```

The signal sent when an itimer expires depends on `which` (the type of time it is counting).

Timers (contd.)

A newer POSIX timer API consists of the functions:

- `timer_create()`
- `timer_settime()`
- `timer_delete()`
- etc.

This API allows more control over the signals sent, finer time resolution, and more info about signal expiration.

Sleep

Signals are also used in process sleeping.

`sleep` will *suspend* a process for a specified number of seconds:
`unsigned int sleep(unsigned int seconds)`

The sleep will be *interrupted* if a signal is received (and whose disposition is not ignore).

This allows a process to suspend itself for a certain number of seconds but possibly be awoken before that by a signal.

Note: `sleep()` may be implemented using `SIGALARM`, so `sleep()` and `alarm()` should not be used simultaneously.

Related "sleep" functions include:

- `nanosleep`
- `clock_nanosleep`

Timers as File Descriptors

Just as there is now a syscall to turn a signals into file descriptors, there are calls to create timers that deliver expiration notices via FDs.

As with `signalfd()`, this has the advantage that the timer FD can be monitored by `select()/poll()/epoll()` instead of signal handling.

The FD can also be monitored by a single thread, while timer signals are normally *process-wide*.

(`timer_create()` has a Linux-specific thread delivery option.)

Timers as File Descriptors (contd.)

Key timerfd functions:

- `int timerfd_create(int clockid, int flags)`
- ```
  int timerfd_settime(int fd, int flags,
                      const struct itimerspec *new_value,
                      struct itimerspec *old_value)
  ```

(Note: These calls are Linux specific.)
  ```

Real-Time Signals

We have so far discussed the so called **standard signals**.

An additional set of signals are the **POSIX real-time signals**.

According to the man pages:

- "Unlike standard signals, real-time signals have no predefined meanings: the entire set of real-time signals can be used for application-defined purposes."
- "The default action for an unhandled real-time signal is to terminate the receiving process."
- "Multiple instances of real-time signals can be queued. By contrast, if multiple instances of a standard signal are delivered while that signal is currently blocked, then only one instance is queued."

Real-Time Signals (contd.)

continuing:

- "Real-time signals are delivered in a guaranteed order. Multiple real-time signals of the same type are delivered in the order they were sent. If different real-time signals are sent to a process, they are delivered starting with the lowest-numbered signal."
- "If both standard and real-time signals are pending for a process...Linux...gives priority to standard signals in this case."

Linux typically supports a range of 32 different real-time signals, numbered from `SIGRTMIN` to `SIGRTMAX`.

Good style to specify a signal like `SIGRTMIN+1` instead of using a specific integer because glibc uses some RT signals internally (it adjusts `SIGRTMIN` so lowest available).

Non-Local Jumps in Handlers

Signals often represent an *error condition*—i.e., an **exception**.

If one is deep within a stack of function calls, it may be appropriate to be able to quickly *"jump"* out of all/most of these calls, aborting out of all of the functionality that caused the error.

This requires *"unwinding the calling stack"* to return to a much higher level.

This can often be accomplished with the following functions:

- `int setjmp(jmp_buf env)`
- `int sigsetjmp(sigjmp_buf env, int savesigs)`
- `void longjmp(jmp_buf env, int val)`
- `void siglongjmp(sigjmp_buf env, int val)`

Non-Local Jumps in Handlers (contd.)

`setjmp()/sigsetjmp()/` save the stack context/environment.

`longjmp()/siglongjmp()` "jump back" to a saved stack context.

Not surprisingly, the man pages say:
"make programs hard to understand and maintain."

However, as exception systems in many languages make clear, such an approach can be preferable to doing what is necessary to cause a stack of functions to immediately return (not to mention it can be much quicker).

Non-Local Jumps in Handlers (contd.)

Non-local jump from handler example:

```
jmp_buf back_at_main;

int main(void)
{
  signal(SIGINT,sigint_handler);  //Use sigaction() really

  if (!setjmp(back_at_main)) {
    //Code run initially after jmp point set:
    ...code that initiates stack of function calls...
    return EXIT_SUCCESS; }
  else {
    //Code run after jmp back due to ^-c:
    ...cleanup, etc...
    return EXIT_FAILURE; }
}

void sigint_handler(int sig)
{
  //Give up on all processing and go back to main:
  longjmp(back_at_main, 1);
}
```