Signals 1: Overview

1. **Overview**
   - **what are signals**
   - **signal terminology**
   - **disposition, handlers, generation**
   - **standard signals**
   - **key signals**
   - **signals and terminals**
   - **real-time signals**
2. System Calls
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Signals

Signals are a mechanism for informing processes that some *event* has occurred.

They are effectively a type of **software interrupt**.

(An **interrupt** is an **asynchronous** notification that can literally interrupt the execution of a program or instruction.)

Most signals are sent to a process by the *kernel*.

However, it is also possible for a process to signal another process and even for a process to signal itself.

This allows signals to be used for process **synchronization** and (very simple) **IPC**.

Basic Signal Terminology

A signal is **generated** (or **raised**) by some event and will usually be destined for one process, though some events generate signals destined for multiple processes.

A generated signal is said to be **pending** before the kernel makes it available to a process.

When the kernel makes the signal available to a (running) process, it is said to be **delivered** and then **received** by the process.

How a process will respond to the delivery of a signal is called the signal's **disposition** (in the process).

A process may **block** particular signals, in which case they are held by the kernel (i.e., remain *pending*) until **unblocked**.

A process' **signal mask** indicates its currently *blocked* signals.

Signal Disposition

There are three possible *classes* of **signal dispositions**:

- **default** – take the **default action** for this signal
- **ignore** – *discard* the signal at delivery time (so no effect)
- **catch** – execute a function called a **signal handler**

There are five possible **default actions**:

- **ignore** – ignore the signal (discard it)
- **terminate** – terminate the process
- **core** – terminate the process and create a **core dump**
- **stop** – stop the process (job control)
- **continue** – continue the process (job control)

Signal Disposition (contd.)

When a signal is set to be *ignored*, the kernel actually just discards the signal at delivery time, so the process knows nothing about it and it has no effect on the process.

A **core dump** is a file containing the contents of the process address space, which can be used with `gdb` to examine the process state at termination.

Even if the *default disposition* is *ignore*, a different behavior may result from *explicitly* setting the disposition to ignore (e.g., for `SIGCHLD`).

Termination by a signal is considered **abnormal termination** (vs. **normal termination** as by calling `exit()`).

Signal Handlers

One of the unique aspects of signals is that signal *disposition* can be set so that a *user-defined function* gets run when a particular signal is delivered.

These functions are known as a **signal handlers**.

When a handler function gets run, signals are said to be **caught** or **handled**.

Signals can be delivered **asynchronously** (i.e., at any point in a program's execution).

Thus signal handlers provide *asynchronous execution* capability.

This can allow a program to respond immediately to some *event* (i.e., without having to **poll** to see if the event has occurred).

Signal Generation

Signals can be **generated** in several ways:

- hardware exceptions, such as:
  - divide by zero
  - illegal memory access
- OS software event, such as:
  - child process terminated (SIGCHLD)
  - writing to pipe with no open read ends (SIGPIPE)
- user types special characters at terminal, such as:
  - `ctrl-c` (*interrupt* character) (SIGINT)
- another process:
  - using `kill()` or `killpg()`
- the process itself:
  - using `abort()` or `alarm()`

Standard Signals

The term **standard signals** is commonly used to refer to **POSIX reliable signals**.

These are the most commonly used/seen signals.

Signals are identified by *positive integers*, but the portable method of denoting standard signals is with *symbolic names* like `SIGKILL`.

This is because the values for different standard signals may vary among UNIXes.

While you may have learned to to kill a process with the command "`kill -9` *PID*" using "`kill -SIGKILL` *PID*" is more portable (9 is the typical numeric value for SIGKILL).

See "`man 7 signal`" for info on the numeric values of signals.

Key Standard Signals

Here are the most important/common standard signals:

- **SIGTERM** – termination signal (`kill` default)
- **SIGKILL** – kill signal (cannot be *ignored* or *blocked*)
- **SIGINT** – "interrupt char" typed on terminal
- **SIGQUIT** – "quit char" typed on terminal
- **SIGCHLD** – child terminated (or stopped)
- **SIGSEGV** – invalid memory reference
- **SIGPIPE** – broken pipe (write to pipe with no readers)
- **SIGALRM** – timer signal (e.g., from `alarm()`)

Key Standard Signals (contd.)

continuing:

- **SIGHUP** – hangup detected on **controlling terminal** or termination of controlling process
- **SIGABRT** – abort signal from `abort()`
- **SIGUSR1** – user-defined signal 1
- **SIGUSR2** – user-defined signal 2
- **SIGILL** – illegal Instruction
- **SIGFPE** – floating point exception
- **SIGBUS** – bus error (bad memory access)
- **SIGPOLL** – pollable event (synonym for SIGIO)
- **SIGCONT** – continue if stopped
- **SIGSTOP** – stop process

Key Standard Signals (contd.)

The *default disposition* for most of these symbols is *termination* or *termination with core dump*.

The following signals have *ignore* as their *default dispositions*: `SIGCHLD` and `SIGURG`.

The following signals have *continue/stop* as their *defaults*: `SIGCONT` and `SIGSTOP` (relates to **job control**).

Signals that result from CPU/hardware issues are referred to as **synchronous signals**: SIGBUS, SIGFPE, SIGILL, SIGSEGV.

Synchronous signals are **thread-specific** (see more later).

Properties of Standard Signals

All standard signals—except for two—can be *ignored or blocked* by processes.

The two signals that *cannot be ignored or blocked* are: `SIGKILL` and `SIGSTOP`.

This is why one is told to do "`kill -SIGKILL` *PID*" instead of just "`kill` *PID*": the default `SIGTERM` signal can be blocked/ignored in processes so process may not terminate.

Using `-SIGKILL` (or `-9`) ensures process will be terminated.

(Note that *stopped processes* cannot be terminated by `SIGTERM`.)

Properties of Standard Signals (contd.)

When a signal is *generated* for a process that has that signal *blocked*, the signal is **queued** by the kernel for possible later delivery (if eventually *unblocked*).

Queueing of standard signals has two key limitations:

- at most *one instance* of each standard signal can be queued
- the *order* queued signals are *delivered* once unblocked is *indeterminate*

This means that if a process blocks signals for a period of time and then unblocks them, the process cannot determine how many of each signal was generated during the block period, nor what order signals were generated in during that period.

(**Real-time signals** address these limitations—see below.)

Signals and Terminals

By default, the Linux/UNIX **terminal driver** recognizes a number of *special characters* to interrupt running programs, do command line editing, send EOF, etc.

Instead of passing special characters through to programs, the terminal driver takes other actions.

Three of the terminal driver special characters cause the terminal driver to *generate signals* and send them to processes associated with the terminal.

This is actually one of the most common ways that users interact with signals (though they are often not aware of this).

Signals and Terminals (contd.)

The terminal driver special characters that generate signals are:

- **INTR** – set to `ctrl-c` by default, when the user types this character, the terminal driver sends a `SIGINT` signal
- **QUIT** – set to `ctrl-\` by default, when the user types this character, the terminal driver sends a `SIGQUIT` signal
- **SUSP** – set to `ctrl-z` by default, when the user types this character, the terminal driver sends a `SIGTSTP` signal

In all cases, the signals are sent to all processes in the **foreground process group** (for which the terminal is a **controlling terminal**).

(Special characters and other terminal characteristics can be changed with the `stty` command or `tcsetattr()` syscall.)

Real-Time Signals

So far we have discussed **standard signals**—i.e., **POSIX reliable signals**.

These are the most commonly seen/used types of signals.

Linux also supports the newer **POSIX real-time signals**.

Real-time signals address some key limitations of standard signals.

Unlike standard signals, real-time signals have no predefined uses (and no symbolic names).

This means that the entire set of real-time signals can be used for application-specific purposes.

The **default disposition** for real-time signals is to *terminate* the receiving process.

Real-Time Signals (contd.)

As with standard signals, real-time signals are represented by positive integers.

The range of supported real-time signals is defined by the macros `SIGRTMIN` and `SIGRTMAX`.

Linux supports 33 real-time signals (32 to 64), the glibc Pthreads implementation uses two or three of these internally, so adjusts `SIGRTMIN` suitably (to 34 or 35).

Because the range of available real-time signals can vary among Linux/UNIX systems, programs should *never refer to real-time signals using hard-coded numbers*.

Instead, programs should always refer to real-time signals using the notation `SIGRTMIN+n` (and possibly include run-time checks that `SIGRTMIN+n` does not exceed `SIGRTMAX`).

Real-Time Signals (contd.)

Real-time signals address two limitations of standard signals:

- they are *fully queued:* multiple instances of each real-time signal can be queued while blocked
- they are delivered in a *guaranteed order:* queued real-time signals of the same type are delivered in the order they were generated, different queued real-time signals are delivered starting with the *lowest numbered* (i.e., lower numbered real-time signals have higher "priority")

Note that if both standard and real-time signals are pending for a process, Linux gives priority to standard signals (though POSIX does not require this).

Signals 2: System Calls

1. Overview
2. **System Calls**
   - **unreliable vs. reliable API**
   - **setting disposition: signal() and sigaction()**
   - **blocking: sigprocmask(), etc.**
   - **sending: kill(), etc.**
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Signal System Calls

There a large number of system calls dealing with signals:

- **signal** – set signal disposition
- **sigaction** – set signal disposition
- **sigemptyset** – set no signals in signal set/mask
- **sigfillset** – set all signals in signal set/mask
- **sigaddset** – add signal to signal set/mask
- **sigdelset** – remove signal from signal set/mask
- **sigismember** – check if signal in signal set
- **sigprocmask** – change signal mask (blocked signals)
- **sigpending** – check the pending blocked signals

Signal System Calls (contd.)

continuing:

- **pause** – suspend process until receive signal
- **sigsuspend** – suspend process until receive signal
- **sigwait,sigwaitinfo** – wait for queued signals
- **sigtimedwait** – wait for queued signals, with time limit
- **kill,sigqueue** – send a signal to a process(es)
- **raise** – send a signal to self
- **sleep** – suspend process for n seconds or signal
- **alarm** – send alarm signal to self after n seconds
- **abort** – send abort signal to self

Unreliable vs. Reliable Signals

The original UNIX signal API left many behaviors unspecified and unable to be set as desired (with different UNIXes having different behaviors).

Among the key issues:

- does a signal handler *stay in effect* after it is invoked or not? (not resetting is called *"mousetrap behavior"*)
- can a signal handler be *interrupted* by the *same signal* (so another call to that handler)?
- can a signal handler be *interrupted* by a *different signal* (so another signal handler)?
- is an *interrupted syscall* automatically *restarted* or not?

Because you could not rely on the behavior you would get across UNIXes, this is often called the **unreliable** signal interface.

Unreliable vs. Reliable Signals (contd.)

`signal()` is the older, **unreliable** signals API syscall.

`sigaction()` is the newer, **reliable** signals API syscall.

The newer signals API both specifies default behaviors for the above issues and provides mechanisms for changing those behaviors.

Because it is simpler, `signal()` can still be used to set *default or ignore disposition*.

However, because its behavior is "unreliable," *it should never be used to setup a signal handler*.

Setting Disposition: signal()

`signal()` is the older, *unreliable* API to set signal disposition:
`sighandler_t signal(int signum, sighandler_t handler)`

- `sighandler_t` is defined as:
  - `typedef void (*sighandler_t)(int)`
- `signum` is the signal (symbol) whose disposition is being changed
- `handler` is the name of a **signal handler** function or one of the special values:
  - `SIG_DFL` (default disposition)
  - `SIG_IGN` (ignore disposition)
- returns the previous value of the signal handler, else `SIG_ERR`

An example call might be: `signal(SIGCHLD,SIG_IGN);`

Setting Disposition: signal() (contd.)

`signal()` is what we call a **higher-order function**, as it takes a function as one of its parameters.

In C, you pass a function as an argument by giving the function's name, but what is actually passed is a *pointer to the function*.

This is why type `sighandler_t` is specified as it is, the notation "(**funcname*)" denotes a **function pointer**.

As already noted, `signal()` should be used only to set default or ignore disposition—*it should not be used to setup a handler*.

Thus `signal()` should be called with second arguments that are only either `SIG_DFL` or `SIG_IGN`.

Setting Disposition: sigaction()

`sigaction()` is the newer, *reliable* API to set signal disposition:
`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`

- `signum` is the signal (symbol) whose disposition is being changed
- `act` contains the (new) disposition settings
- `oldact` receives the old disposition settings
  (so they can be easily restored)
- `oldact` can be `NULL` if the old disposition is no longer needed
- returns 0 on success, -1 on error

Setting Disposition: sigaction() (contd.)

`sigaction()` makes use of the `sigaction` struct type:

```
struct sigaction {
  void     (*sa_handler)(int);   /* Disposition/handler specification */
  void     (*sa_sigaction)(int, siginfo_t *, void *);
                                 /* Alternative "siginfo handler" */
  sigset_t sa_mask;              /* Signal mask during handler run*/
  int      sa_flags;             /* Flags/options */
  void     (*sa_restorer)(void); /* Obsolete */
}
```


Setting Disposition: sigaction() (contd.)

Example of registering a handler using `sigaction()`:

```
struct sigaction act;

memset(&act,0,sizeof(act));
act.sa_handler = sigint_handler;  //Name of handler function
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGCHLD);
act.sa_flags = SA_RESTART;

sigaction(SIGINT,&act,NULL)
```


Setting Disposition: sigaction() (contd.)

The default behavior for the key issues mentioned earlier are:

- a signal handler *stays in effect* after it is invoked (i.e., not "mousetrap behavior)
- a signal handler cannot be *interrupted* by the *same signal*
- a signal handler can be *interrupted* by a *different signal*
- an *interrupted syscall* is *not* automatically *restarted*

These behaviors can all be changed by including appropriate `sa_flags` and/or `sa_mask` values in the sigaction struct.

Setting Disposition: sigaction() (contd.)

Key values for the sigaction struct `sa_flags` field:

- `SA_RESETHAND` – restore default disposition once the signal handler has been invoked ("mousetrap behavior")
- `SA_NODEFER` – do not block signal while its own handler is being run (default is to block even signal if not in `sa_mask`)
- `SA_RESTART` – automatically restart interrupted system calls
- `SA_SIGINFO` – use alternative handler format

Signal Blocking

The asynchronous delivery of a signal can cause problems:

- the signal might interrupt a *"slow"* syscall like `read()`
- termination might interrupt a sequence of program steps, leaving a database (or similar) in an inconsistent state
- child might signal parent before parent has updated its state to reflect `fork()`

Thus, there may be **critical sections** in programs where we do not want particular (or even any) signals to be delivered.

Most signals can be **blocked**: held by the kernel for possible later delivery (when unblocked).

A process' **signal mask** is its current set of blocked signals.

Signal Blocking (contd.)

Two signals *cannot be blocked*: `SIGKILL` and `SIGSTOP`.

(They also cannot be ignored.)

This is why a sure way to terminate a process using the `kill` command is to use `-9` or `-SIGKILL` to send `SIGKILL`.

Note that with *standard signals*, only *one copy* of a blocked signal is maintained by the kernel, even if multiple instances of the signal were generated while the signal was blocked.

Also, there are no guarantees that blocked signals will be delivered in the order they were generated once they are unblocked.

Blocking: sigprocmask

A process' **signal mask** of blocked signals is changed using `sigprocmask`:

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

- changes the *signal mask* based on a **signal set**
- `how` determines what changes to make:
  - `SIG_BLOCK` – add signals in `set` to signal mask
  - `SIG_UNBLOCK` – remove signals in `set` from signal mask
  - `SIG_SETMASK` – set the signal mask to be `set`
- `set` is the signal set that specifies the signals to act on
- `oldset` stores the old signal mask (so it can be restored)

Blocking: sigprocmask (contd.)

Example of setting the signal mask to block only the `SIGUSR` signals:

```
sigset_t sset_siguser;

sigemptyset(&sset_siguser);
sigaddset(&sset_siguser,SIGUSR1);
sigaddset(&sset_siguser,SIGUSR2);

sigprocmask(SIG_SETMASK,&sset_siguser,NULL);
```

Blocking: sigprocmask (contd.)

Signal sets can be manipulated with several functions:

- `sigemptyset()`
- `sigfillset()`
- `sigaddset()`
- `sigdelset()`
- `sigismember()`

Blocking: sigpending

Sometimes one may want to check if there are any *pending blocked signals*, such as before unblocking signals.

This can be done with `sigpending`:
`int sigpending(sigset_t *set)`

- `set` receives the set of pending signals, as a *signal set*

Sending Signals: kill

A process can send a signal to another process using `kill`:
`int kill(pid_t pid, int sig)`

- `pid` can be one of:
  - a positive integer – send to PID of `pid`
  - 0 – send to every process in *process group* of caller
  - -1 – send to every process have permission to send to
  - a negative integer – send to *process group* |`pid`|
- `sig` is the signal to send, else if 0 "no signal is sent, but error checking is still performed; this can be used to check for the existence of a process ID or process group ID"

To be allowed to signal another process, the sender must have the same real or effective UID as the target process' real UID, or else have `root` as its effective UID (really `CAP_KILL`).

Sending Signals: sigqueue and raise

`sigqueue` is an alternative to `kill()`:
`int sigqueue(pid_t pid, int sig, const union sigval value)`

The primary reason to use `sigqueue()` is that it can be used to pass additional info to a handler ("**siginfo handlers**").

Siginfo handlers are installed by using `sigaction()`'s `SA_SIGINFO` flag and `sa_sigaction` struct field.

`raise` allows a process to *signal itself*:
`int raise(int sig)`

Abnormal Termination: abort

`abort` can be used by a process to terminate itself *abnormally*:
`void abort(void)`

- first unblocks `SIGABRT` then raises that signal for the calling process
- results in the abnormal termination unless `SIGABRT` is caught and the handler does not return (e.g., `siglongjmp()`)
- if `SIGABRT` is ignored or caught by a handler that returns, the process will still terminate: default disposition for `SIGABRT` will be restored and signal raised for a second time.
- never returns

Signals 3: Signal Handlers

1. Overview
2. System Calls
3. **Signal Handlers**
   - **signal handlers syntax**
   - **examples**
   - **passing data: global variables**
   - **siginfo handlers**
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

Registering a Signal Handler

As note previously, `signal()` should be used *only to set default or ignore disposition*.

Setting signal disposition to be *caught* by a *signal handler* should *always be done with* the newer `sigaction()`.

(This is often termed *establishing* or *registering* a handler.)

`sigaction()` requires creating an appropriate sigaction struct:

```
struct sigaction act;
memset(&act,0,sizeof(act));
act.sa_handler = sigint_handler;  //Name of handler function
sigemptyset(&act.sa_mask);
sigaddset(&act.sa_mask,SIGCHLD);
act.sa_flags = SA_RESTART;

sigaction(SIGINT,&act,NULL)
```


Signal Handlers Syntax

What is the *syntax* for defining a *signal handler* function?

`signal()`'s syntax is commonly shown using a `typedef` for the signal handler syntax to simplify things.

The typedef used with `signal()` specifies the signal handler as: `void (*sighandler_t)(int)`:

- `(*`*function*`)` denotes a function (pointer)
- `(int)` indicates the function takes a single, `int` argument
- `void` indicates the function has a `void` return type

Thus, to be acceptable as a *signal handler*, a function must take a single `int` argument and have `void` return type.

Signal Handlers Syntax (contd.)

The sigaction struct does not use a `typedef` for the handler element, but species the same syntax for handler functions:

```
void (*sa_handler)(int)
```

Note that without using a `typedef`, `signal()` could be defined as:

```
void (*oldhandler)(int) signal(int signum, void (*newhandler)(int))
```


Signal Handler Examples

Example signal handler function for use with SIGINT:

```
void sigint_handler(int signal)
{
  //Beep terminal instead of terminating process on ^-c:
  write(1,"\a",1);  //'\a' is the ASCII BEL character
  return;
}
```

Example signal handler function for use with SIGCHLD:

```
void sigchld_handler(int signal)
{
  //Collect all terminated children:
  while (waitpid(-1,NULL,WNOHANG) > 0);
  errno=0;  //in case waitpid() set it
  return;
}
```


Passing Data to Handlers

The only information passed to basic signal handlers is the signal that invoked the handler (but see "**siginfo handlers**" below).

Obviously this is not useful when the handler is registered for a single signal.

However, the same handler function may be registered for multiple signals, so the handler may need to check which signal invoked it to determine what to do.

Often, it is necessary for the program that setup a handler to be able to pass data to or from a handler.

Because of signal handlers' syntax, this requires **global variables**.

(In C, *global variables* are variables declared outside of `main` or any functions, giving them **global scope**.)

Example of Passing Data via Global Variable

Terminate loop in program when user types ^-c:

```
int done = 0;  //Global flag variable

int main()
{
  ...use sigaction() to register sigint_handler() as handler for SIGINT...
  ...
  while (!done) {
    ...do stuff...
  }
  return EXIT_SUCCESS;
}

void sigint_handler(int signal)
{
  done=1;
  return;
}
```


Siginfo Handlers

Standard signal handlers have to have prototypes like:

```
void sa_handler(int signo)
```

`sigaction()` can be used to setup such handlers via the sigaction struct's `sa_handler` field.

However, `sigaction()` can instead setup "**siginfo handlers**."

These handlers receive additional information about the signal context, and have prototypes:

```
void sa_sigaction(int signo, siginfo_t *si, void *cntxt)
```

To register such a handler:

- the `SA_SIGINFO` flag must be set in the sigaction struct
- the handler function is set in the `sa_sigaction` field
- no handler is set using the `sa_handler` field

Siginfo Handlers (contd.)

`siginfo_t` is a struct with additional info about signal context.

It is somewhat complex, since it contains a **union** that defines different fields depending upon the signal it results from.

Key fixed fields:

- `int si_signo` – the signal (number)
- `int si_errno` – errno value if nonzero
- `int si_code` – for some signals, code that identifies event

An important aspect of siginfo handlers is the ability for limited data to be *passed to the signal handler* when using `sigqueue()` (see below) to send a signal to a process.

Siginfo Handlers (contd.)

This involves the `siginfo_t` struct's `si_value` field, a *union*:

```
union {
  int   sival_int;
  void *sival_ptr;
}
```

Thus, we can pass either an integer value (in `si_value.sival_int` or a pointer value (in `si_value.sival_ptr`).

(Note that pointers will be valid only if a process uses `sigqueue()` to send a signal to *itself!*)

This allows data to be passed to handlers *without* the use of **global variables**.

That can be particularly useful with **multithreaded programs** (since globals are *shared* among threads).

Signals 4: Synchronization

1. Overview
2. System Calls
3. Signal Handlers
4. **Synchronization**
   - **IPC vs. synchronization**
   - **synchronization: pause() and sigsuspend()**
   - **synchronous handling: sigwait^{+}()**
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

IPC and Process Synchronization

A signal informs a process that some event has occurred, so if one process sends a signal to another process, information is transmitted, allowing a very simple form of IPC.

Often, this is all that is needed, as when one process needs to inform another that it has completed certain operations.

The `SIGUSR?` signals are particularly appropriate for this purpose.

It is common for such IPC to be used for **synchronization**—to have one process wait to proceed beyond a certain point in its code until the other informs it that certain (precondition) code has been completed.

Signals are useful for synchronization because there are calls that will cause a process to *suspend itself* until an appropriate signal is received.

Synchronization: pause and sigsuspend

`pause` suspends a process/thread until a signal is received:
```
int pause(void)
```

`sigsuspend` does the same thing, but temporarily replaces the process' *signal mask* before suspending the process:
```
int sigsuspend(const sigset_t *mask)
```

A key requirement for a signal to unpause/unsuspend a process is that *the signal's disposition must be one of two things*:
1. terminate (the process)
2. catch (invoke a handler function)

Because of this requirement, it is not uncommon to define and register an "*empty*" handler function (i.e., function does nothing).

This allows the signals to be used with `pause()`/`sigsuspend()` to unsuspend a process and continue running.

Synchronization: pause and sigsuspend (contd.)

The added functionality of `sigsuspend()` in changing the signal mask allows control over what signals might unsuspend the program, by changing the set of signals that are blocked while suspended.

(It also automatically restores the mask in effect prior to the call when the process is unsuspended.)

Thus, `sigsuspend()` has similar functionality to doing:
```
sigprocmask(SIG_SETMASK,&mask,&savemask);
pause();
sigprocmask(SIG_SETMASK,&savemask,NULL);
```

The problem with the above sequence of three calls is that they are not carried out as an **atomic operation**—like `sigsuspend()`.

This can lead to problems with **race conditions**.

Synchronization: pause and sigsuspend (contd.)

With the three-call sequence, you could unblock a signal that you want to unsuspend the process, but then have that signal get delivered before `pause()` gets executed.

This would result in the program *hanging*, because it would be suspended waiting for a signal that has already been delivered.

This is a *race condition*, because whether the problem occurs or not will depend on the exact relative timing of the instructions.

Race conditions are a serious problem that programmers must consider when writing programs that involve *asychronous events* like signals.

Synchronous Handling: sigwait$^+$

Asynchronously interrupting a process' normal execution to run a signal handler function has drawbacks, such as requiring handler code be reentrant, potentially interupting syscalls, and so forth.

We just saw how `sigsuspend()` can be used to have a process simply stop running and wait for a signal to be received, effectively meaning that normal execution does not get interrupted.

This approach to signal interaction with a program is often described as **synchronous signal handling**.

While `sigsuspend()` can be used to implement synchronous signal handling, we saw that this requires defining a handler function, which is often empty.

Because of this, three new system calls were added to SUS to make it easier to use signals to *synchronously* accept signals.

Synchronous Handling: sigwait$^+$ (contd.)

The three calls all allow a process or a thread to suspend itself and wait for any of a *set* of signals before proceeding.

This is similar to what can be done with `sigsuspend()` and *empty handlers*.

The three calls are:

- `int sigwait(const sigset_t *set, int *sig)`
- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)`
- `int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`

Synchronous Handling: sigwait$^+$ (contd.)

These functions are typically used by *blocking all signals* (using `sigprocmask()`) and then *waiting for particular signals*.

The functions all return when one of the waited-for signals is *generated and becomes pending* (but not delivered since blocked).

(The pending signal is cleared before the function returns.)

The functions provide similar functionality, differing as follows:

- `sigwait` – signal is placed in `sig` parameter
- `sigwaitinfo` – signal is return value, stores info in `siginfo_t` (struct) parameter
- `sigtimedwait` – like `sigwaitinfo` but can limit time to wait

Synchronous Handling: sigwait$^+$ (contd.)

Example of waiting for the SIGUSR signals:

```
//Block all signals:
sigset_t blocked;
sigfillset(&blocked);
sigprocmask(SIG_SETMASK,&blocked,NULL);

//Set up signals to wait for:
sigset_t waiting;
sigemptyset(&waiting);
sigaddset(&waiting,SIGUSR1);
sigaddset(&waiting,SIGUSR2);

//Wait for and dispatch on SIGUSR? signal:
while(1) {
  switch(sigwaitinfo(&waiting,NULL) {
    case SIGUSR1:
      ...
    case SIGUSR2:
      ...
  }
}
```


Synchronous Handling: sigwait$^+$ (contd.)

The sigwait$^+$ calls are particularly useful with *threads:*

- they suspend only the *thread* they are called in
- signal masks can be set on a *per-thread* basis

(Signals and thread interactions are discussed further below.)

The calls also work with the **real-time signals** (see below).

Because they are "newish," the following feature test macro may be required to be able to use the sigwait$^+$ calls:

```
#define _POSIX_C_SOURCE 199309
```

Signals 5: fork, exec, Pthreads

1. Overview
2. System Calls
3. Signal Handlers
4. Synchronization
5. **Signals and fork, exec, Pthreads**
   - **signals across fork**
   - **signals across exec**
   - **signals and Pthreads**
6. Advanced Topics
7. Signal Handler Issues

Signals Across fork/exec

Since signals are often used in **multi-process** programs, it is critical to understand what happens to signal disposition across calls to `fork()` and `exec()`.

Consider, for example, that it is possible for a signal to be able to be delivered to a new process before the first statement in the new process can be executed.

Thus, the only way we could be guaranteed to have signal handling in place in time would be by setting it up *prior* to making the `fork()/exec()` call.

This would obviously also require that signal disposition remained unchanged across `fork()/exec()`.

Is this the case?

Signals Across fork

Here is the situation with signals and `fork()`:

- signal disposition (including handlers) is inherited
- signal mask is inherited
- pending signals are not inherited
- pending timers are not inherited

So signal disposition and blocking is unaffected by `fork()`.

The child will have the same settings as were in affect in the parent at the time of the `fork()`.

Signals Across exec

Here is the situation with signals and `exec()`:

- signals whose disposition is *catch* are reset to *default*
- signals whose disposition is default/ignore remain unchanged
- unspecified if this is true for ignored `SIGCHLD`, however Linux leaves it unchanged
- signal mask remains unchanged
- pending signals remain unchanged
- all pending timers are cleared

(Man page for `execve()` makes somewhat different claims!)

Signals and Pthreads

The signal model was developed long before **POSIX Threads**.

The interaction of signals with Pthreads can make it somewhat complex to use signals in **multithreaded programs**.

Nonetheless, it is possible to use signals in multithreaded programs.

In fact, the ability to have thread(s) dedicated to handling signals can be very useful.

When using signals in multithreaded programs , it is critical to understand that some aspects of signals apply **process-wide** and some are **thread-specific**.

E.g., while most older signal syscalls apply **process-wide**, several signal-related `pthread_` syscalls are **thread-specific**.

Signals and Pthreads (contd.)

Here are key points about signals and threads:

- **process-wide** vs. **thread-specific**:
  - most signals are *process-wide* by default
  - **synchronous signals** (e.g., `SIGFPE`) are *thread-specific*
  - `kill()` delivers a signal to an entire *process*
  - `raise()` delivers a signal to the calling *thread*
  - `pthread_kill()` and `pthread_sigqueue` will deliver a signal to a specific Pthread
  - target thread must be in same process as sender
  - there is no way to signal a single thread in another process
  - **timers** are *process-wide* resources (so shared by all threads)

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **blocking**:
  - **signal masks** (signals being **blocked**) are a *per-thread* attribute
  - this means signal *blocking* is *thread-specific*
  - use `pthread_sigmask()` to set *signal masks* in multithreaded programs
  - in multithreaded programs, the behavior of `sigprocmask()` is *undefined*
  - a newly created thread *inherits a copy* of its creator's *signal mask*
  - the set of **pending signals** for the new thread is *empty*
  - `sigpending()` gives the set of pending signals for the calling *thread* only
  - this is the *union* of pending thread-specific and process-directed signals

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **disposition**:
  - signal **disposition** is a *per-process* attribute
  - this means all threads will must have the *same disposition* for each signal
  - if a delivered signal's disposition is *termination*, the *entire process* (all threads) is terminated
- signal **handling**:
  - if a *process-wide* signal is *caught*, one thread is *randomly chosen* to run the handler
  - **synchronous signal handling** is better with Pthreads
  - approach: block signal(s) in all threads, then call `sigwait()` in *"handler thread"*

Pthreads Signal Syscalls

Signal-related syscalls that are specifically for use with *Pthreads*:

- `pthread_sigmask` – set signal mask for a thread (`sigprocmask()`'s behavior is undefined in multithreaded program)
- `pthread_kill` – send signal to one thread (in same process) (`kill()` sends signal to entire process)
- `pthread_sigqueue` – queue a signal and data to a thread in process (`sigqueue()` sends signal to entire process)

Synchronous Signal Handling

In multithreaded programs that must respond to particular signals, *asynchronous* signal handlers are generally *not* the best approach.

It is better to use **synchronous signal handling**:

- *block* signals to be handled in all threads
- create a thread for signal handling (or can create separate thread for each signal)
- have the handling thread(s) use `sigwait()` (or related) to set the thread up to respond to its particular signal(s)
- when `sigwait()` returns, the handling thread carries out the actions that would traditionally have been in handler functions

Example: Asynchronous Signal Handler

Classic *asynchronous* SIGINT handler example:

```
int main(void)
{
  struct sigaction act;
  memset(&act,0,sizeof(act));
  act.sa_handler = sigint_handler;
  sigemptyset(&act.sa_mask);
  sigaction(SIGINT,&act,NULL);  //Set SIGINT to be caught

  ...main program code...

  return EXIT_SUCCESS;
}

void sigint_handler(int sig)
{
  ...handler actions...
  return;
}
```


Example: Synchronous Signal Handling

Thread-based *synchronous* SIGINT handler example:

```
int main(void)
{
  sigset_t mask;
  sigemptysetset(&mask);
  sigaddset(&mask, SIGINT);  //Block SIGINT for entire process
  sigprocmask(SIG_SETMASK, &mask, NULL);

  pthread_t tid;
  pthread_create(&tid, NULL, sigint_handler_thread, NULL);

  ...main program code...
  return EXIT_SUCCESS;
}

void *sigint_handler_thread(void *ignore)
{
  sigset_t catching;
  sigemptyset(&catching);
  sigaddset(&catching, SIGINT);
  int sig;
  while(1) {
    sigwait(&catching, &sig);  //Wait for SIGINT (to be pending)
    ...handler actions...
  }
  return NULL;
}
```

sigwait() and related

Three syscalls for synchronous signal handling with threads:

- `int sigwait(const sigset_t *set, int *sig)`
- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)`
- `int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`

Operation:

- suspend execution of the calling thread until one of the signals specified in `set` becomes *pending*
- *accept* the signal (removing from pending) and return
- if a signal in `set` is pending when called, return immediately
- if multiple signals in `set` are pending, retrieved signal is determined by usual ordering rules (see "`man 7 signal()`")

sigwait() and related (contd.)

`sigwait()`:

- passes signal number back in `sig`
- returns 0 on success, else a positive error number

`sigwaitinfo()`:

- passes `siginfo_t` structure describing the signal back in `info`
- returns signal number on success, else -1 with `errno` set

`sigtimedwait()` is just like `sigwaitinfo()` except:

- `timeout` argument specifies max time for which the thread can be suspended waiting for a signal

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);
}
```

Signals 7: Handler Issues

1. Overview
2. System Calls
3. Signal Handlers
4. Synchronization
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. **Signal Handler Issues**
   - **reentrant/async-signal-safe functions**
   - **atomicity of instructions**
   - **volatile sig_atomic_t**
   - **interrupted syscalls**

Signal Handler Issues

Signal handlers are a very powerful programming mechanism because they provide a way to *asynchronously execute code* in response to events (without inefficient polling or delays).

However, signal handlers can result in *asynchronous interruptions* of program flow:

- *library functions* generally involve multiple operations, so library calls can be interrupted by a handler
- *C statements* generally require multiple machine instructions, so may be interrupted by a handler
- *"slow" system calls* (i.e., those syscalls that can block) can be interrupted by a handler and not complete
- handlers may not return to interrupted code due to a **non-local jump** or process termination

Signal Handler Issues (contd.)

Writing reliable code inovlving signal handlers requires care:

- interrupted library functions must be able to properly continue after returning from the handler(s)—i.e., must be **reentrant**
- operations must be made **atomic** or else must be able to be interrupted without affecting program logic
- "slow" system calls have to be checked to see if they were interrupted and have to be *restarted*
- *non-local jumps* should be limited to serious error situations
- *cleanup code* must still be run before termination (so may want to raise different signal to terminate or raise signal second time)
- *global variables* for passing data to/from handlers are generally not reliable in *multithreaded* programs

Reentrant/Async-Signal-Safe Calls

A **reentrant function** is one that can start executing, be interrupted and called a second time, and yet eventually correctly continue the first execution (when second call returns).

Rentrancy is important with *signal handlers* and *multithreading:*

- a handler can interrupt a function call, and the original function must be able to continue properly if the handler returns
- multiple threads may concurrently invoke the same function

POSIX defines two concepts related to reentrancy:

- **async-signal-safe functions**
- **thread-safe functions**

Reentrant/Async-Signal-Safe Calls (contd.)

Functions are *not* reentrant when they use *global* or `static` data: if a first call gets interrupted, a second call could change data being used by the first call, leading to incorrect outcomes when the first call continues.

All functions called inside a signal handler should be async-signal-safe, because the handler may have interrupted a function (and some handlers can even be interrupted by another handler).

POSIX says: "If a signal interrupts the execution of an *unsafe* function, and (the) handler calls an unsafe function, then the behavior of the program is *undefined*."

Code that can have *undefined behavior* is incorrect code!

Reentrant/Async-Signal-Safe Calls (contd.)

SUS lists a set of functions that must be async-signal-safe; the list can be found with "man 7 signal."

Note that very few *C Standard Library functions* are listed, which means library functions should generally be avoided in handlers.

Many system calls are included however.

`printf()`/`fprintf()` are not included, but `write()` is!

Non-standard, *reentrant versions* of some library functions may exist, named as $func$`_r`: e.g., `asctime` and `asctime_r`.

Atomicity

Atomicity of operations is another issue that must be considered when writing asynchronous and concurrent programs.

In terms of an OS, a operation is **atomic** if once initiated, the operation cannot be suspended or be interrupted (e.g., by a signal).

Key: once started, no relevant data/state can be changed by something else.

Most system calls are atomic (though see below about *"slow" syscalls*), but *few library functions are*.

Individual *machine code instructions* are atomic, but individual *C statements* are generally *not* (since require multiple machine instructions).

Atomicity (contd.)

If your program makes use of signal handlers, program operations may be interrupted.

There are three possible approaches to ensuring program works as intended:

- use data types that are always accessed *atomically*
- design your program code so that nothing "bad" happens if an operation is interrupted
- *block* all signals during operations that cannot be interrupted

Atomicity: Example

GNU C library documentation includes this atomicity example:

```
struct two_words { int a, b; } memory;

void handler(int signum)
{
   printf ("%d,%d\n", memory.a, memory.b);
   alarm (1);
}

int main (void)
{
   static struct two_words zeros = { 0, 0 }, ones = { 1, 1 };
   signal (SIGALRM, handler);
   memory = zeros;
   alarm (1);
   while (1) {
       memory = zeros;
       memory = ones;
     }
}
```

"This program fills memory with zeros, ones, zeros, ones, alternating forever; meanwhile, once per second, the alarm signal handler prints the current contents."

Atomicity: Example (contd.)

"Clearly, this program can print a pair of zeros or a pair of ones. But that's not all it can do! On most machines, it takes several instructions to store a new value in memory, and the value is stored one word at a time. If the signal is delivered in between these instructions, the handler might find that memory.a is zero and memory.b is one (or vice versa)."

"On some machines it may be possible to store a new value in memory with just one instruction that cannot be interrupted. On these machines, the handler will always print two zeros or two ones."

Atomicity: volatile sig_atomic_t

It is common inside handler functions to test and/or set **global flag variables** (integers).

Unfortunately these operations may not be atomic, so if interrupted by another signal/handler, can lead to unexpected behavior.

To minimize this possibility, *global variables for use in signal handlers* should be declared as: `volatile sig_atomic_t`.

The `sig_atomic_t` type is an integer type, whose range can be determined from the constants: `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX`.

Doing this will guarantee that the following are *atomic:*

- variable access (retrieve value)
- variable assignment (set value)

Atomicity: volatile sig_atomic_t (contd.)

Notice that increment/decrement operations (e.g., ++/--) are still *not* guaranteed to be atomic!

The GNU C library documentation states that:
"In practice, you can assume that int and other integer types no longer than int are atomic. You can also assume that pointer types are atomic; that is very convenient. Both of these are true on all of the machines that the GNU C library supports."

The `volatile` keyword is a C keyword that keeps the compiler from optimizing a variable to be stored in a register (which can cause issues with transfers to handler code).

Atomicity: volatile sig_atomic_t (contd.)

Example of reliable use of a flag in a handler:

```
//Global variables:
volatile sig_atomic_t sig_flag = 0;

int main()
{
  ...
  act.sa_handler = handler;
  sigaction(SIGINT,&act,NULL)
  ...
  while(!sig_flag) {
  ...
}

void handler(int sig)
{
  sig_flag = 1;
  return;
}
```


Access Patterns that Avoid Issues

There GNU C Library documentation describes how to reason about access patterns to determine if they avoid issues even if an access is interrupted:

"A flag which is set by the handler, and tested and cleared by the main program from time to time, is always safe even if access actually requires two instructions.

To show that this is so, we must consider each access that could be interrupted, and show that there is no problem if it is interrupted.

An interrupt in the *middle of testing* the flag is safe because either it's recognized to be nonzero, in which case the precise value doesn't matter, or it will be seen to be nonzero the next time it's tested.

An interrupt in the middle of clearing the flag is no problem because either the value ends up zero, which is what happens if a signal comes in just before the flag is cleared, or the value ends up nonzero, and subsequent events occur as if the signal had come in just after the flag was cleared.

As long as the code handles both of these cases properly, it can also handle a signal in the *middle of clearing the flag*."

Interrupted System Calls

While most system calls are *atomic*, some syscalls, the so-called **"slow" system calls**, are *not*.

The "slow" system calls are those syscalls like `read()` that can take time to complete and/or that can *block indefinitely*.

It is possible for a signal to literally interrupt these syscalls: while `read()` is blocked waiting for data to become available, control is switched to a handler, but when control is transferred back, the instruction counter has already been incremented, so execution proceeds from the *next* instruction.

If a syscall is interrupted and does not complete, it will return an *error result* and set `errno` to `EINTR`.

(See "`man 7 signal`" for info on syscall interruptions.)

Interrupted System Calls (contd.)

One approach to deal with the possibility of certain syscalls being interrupted is to write your code to deal the possibility.

This will generally mean enclosing calls to functions like `read()` inside a loop, continuing only when it does not result in `EINTR`.

Instead of:

```
if ((nread = read(fd,buff,size)) == -1 {
  ...read-error response... }
...non-error continuation...
```

Must do:

```
while ((nread = read(fd,buff,size)) == -1) && errno == EINTR);
if (nread == -1) {
  ...read-error response... }
...non-error continuation...
```

Interrupted System Calls (contd.)

Another approach is to have the kernel *automatically restart* any syscalls that get interrupted by signals.

This can be done when using `sigaction()` to set disposition, by setting the `SA_RESTART` flag.

Different UNIXes taken have different approaches when `signal()` is used to set signal disposition: some automatically restart interrupted syscalls and others do not.

With `sigaction()`, interrupted syscalls are *not* automatically restarted unless the `SA_RESTART` flag is set.

Interrupted System Calls (contd.)

Most interruptable syscalls can be automatically restarted via the `SA_RESTART` flag, including:

- `read()`, `readv()`, `recv()`, etc.
- `write()`, `writev()`, `send()`, etc.
- `open()`
- `wait()`, `waitpid()`, etc.
- `ioctl()`
- `accept()`
- `connect()`