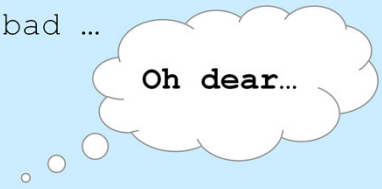
CS 33

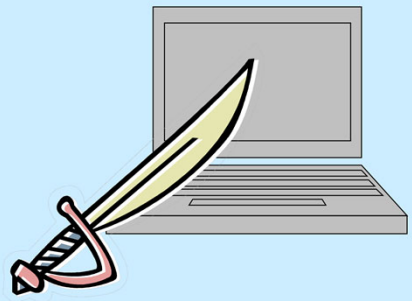
Signals

Whoops …

```
$ SometimesUsefulProgram xyz
Are you sure you want to proceed? Y
Are you really sure? Y
Reformatting of your hard drive will
begin in 3 seconds.
Everything you own will be deleted.
There's little you can do about it.
Too bad …
```



One Approach …



A Gentler Approach

- Signals
  - get a process's attention
    - » send it a signal
  - process must either deal with it or be terminated
    - » in some cases, the latter is the only option

Stepping Back …

- What are we trying to do?
  - interrupt the execution of a program
    - » cleanly terminate it

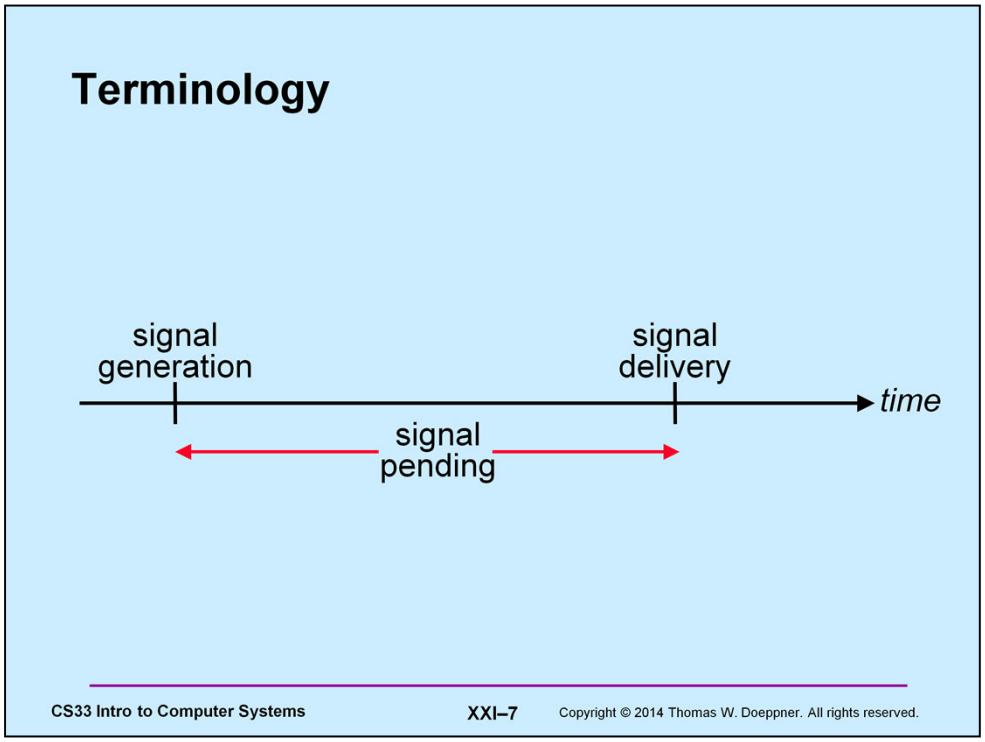
      or
    - » cleanly change its course
  - not for the faint of heart
    - » it's difficult
    - » it gets complicated
    - » (not done in Windows)

Signals

- Generated (by OS) in response to
  - exceptions (e.g., arithmetic errors, addressing problems)
    - » synchronous signals
  - external events (e.g., timer expiration, certain keystrokes, actions of other processes)
    - » asynchronous signals
- Effect on process:
  - termination (possibly after producing a core dump)
  - invocation of a procedure that has been set up to be a signal handler
  - suspension of execution
  - resumption of execution

Signals are a kernel-supported mechanism for reporting events to user code and forcing a response to them. There are actually two sorts of such events, to which we sometimes refer as *exceptions* and *interrupts*. The former occur typically because the program has done something wrong. The response, the sending of a signal, is immediate; such signals are known as *synchronous* signals. The latter are in response to external actions, such as a timer expiring, an action at the keyboard, or the explicit sending of a signal by another process. Signals send in response to these events can seemingly occur at any moment and are referred to as *asynchronous* signals.

Processes react to signals using the actions shown in the slide. The action taken depends partly on the signal and partly on arrangements made in the process beforehand.



A signal is *generated* for (or sent to) a process when the event that causes the signal first occurs; the same event may generate signals for multiple processes. A signal is *delivered* to a process when the appropriate action for the process and signal is taken. In the period between the generation of the signal and its delivery the signal is *pending*.

Much like how hardware-generated interrupts can be masked by the processor, (software-generated) signals can be *blocked* from delivery to the process. Associated with each process is a vector indicating which signals are blocked. A signal that's been generated for a process remains pending until after it's been unblocked.

Signal Types

SIGABRT	*abort* called	term, core
SIGALRM	alarm clock	term
SIGCHLD	death of a child	ignore
SIGCONT	continue after stop	cont
SIGFPE	erroneous arithmetic operation	term, core
SIGHUP	hangup on controlling terminal	term
SIGILL	illegal instruction	term, core
SIGINT	interrupt from keyboard	term
SIGKILL	kill	forced term
SIGPIPE	write on pipe with no one to read	term
SIGQUIT	quit	term, core
SIGSEGV	invalid memory reference	term, core
SIGSTOP	stop process	forced stop
SIGTERM	software termination signal	term
SIGTSTP	stop signal from keyboard	stop
SIGTTIN	background read attempted	stop
SIGTTOU	background write attempted	stop
SIGUSR1	application-defined signal 1	stop
SIGUSR2	application-defined signal 2	stop

This slide shows the complete list of signals required by POSIX 1003.1, the official Unix specification. In addition, many Unix systems support other signals, some of which we'll mention in the course. The third column of the slide lists the default actions in response to each of the signals. *term* means the process is terminated, *core* means there is also a core dump; *ignore* means that the signal is ignored; *stop* means that the process is stopped (suspended); *cont* means that a stopped process is resumed (continued); *forced* means that the default action cannot be changed and that the signal cannot be blocked or ignored.

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
- Also
  - *kill* shell command
  - type ctrl-c
    - » sends signal 2 (SIGINT) to current process
  - type ctrl-\
    - » sends signal 3 (SIGABRT) to current process
  - type ctrl-z
    - » sends signal 20 (SIGTSTP) to current process
  - do something illegal
    - » bad address, bad arithmetic, etc.

Note that the signals generated by typing control characters on the keyboard are actually sent to the current process group of the terminal, a concept we discuss soon.

Handling Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signo,
    sighandler_t handler);

sighandler_t OldHandler;

OldHandler = signal(SIGINT, NewHandler);
```


Special Handlers

- SIG_IGN
  - ignore the signal
  - `signal(SIGINT, SIG_IGN);`
- SIG_DFL
  - use the default handler
    - » usually terminates the process
  - `signal(SIGINT, SIG_DFL);`

Example

```
int main() {
  void handler(int);

  signal(SIGINT, handler);
  while(1)
    ;
  return 1;
}
void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```


Note that the C compiler implicitly concatenates two adjacent strings, as done in printf above.

Signals and Handlers

- What happens when signal is delivered to process that has a handler?
  - original Unix: current handler is called, but for subsequent occurrences, handler set to default
  - BSD (1981): new system call, sigset, introduced
    - » signal/handler association is permanent
    - » signal is blocked (masked) while handler is running
  - Sun Solaris (~1992): meanings of *signal* and *sigset* switched
  - Linux
    - » man page doesn't say what happens
      - signal and sigset do same thing:
        - handler stays associated with signal

sigaction

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

int main() {
   struct sigaction act; void sighandler(int);
   sigemptyset(&act.sa_mask); // zeroes the mask
   act.sa_flags = 0;
   act.sa_handler = sighandler;
   sigaction(SIGINT, &act, NULL);
   …
}
```


The *sigaction* system call is the primary means for establishing a process's response to a particular signal. Its first argument is the signal for which a response is being specified, the second argument is a pointer to a *sigaction* structure defining the response, and the third argument is a pointer to memory in which a *sigaction* structure will be stored containing the specification of what the response was prior to this call. If the third argument is null, the prior response is not returned.

The *sa_handler* member of *sigaction* is either a pointer to a user-defined handler function for the signal or one of SIG_DFL (meaning that the default action is taken) or SIG_IGN (meaning that the signal is to be ignored). The *sig_action* member is an alternative means for specifying a handler function; we discuss it in an upcoming slide.

When a a user-defined signal-handler function is entered in response to a signal, the signal itself is masked until the function returns. Using the *sa_mask* member, one can specify additional signals to be masked while the handler function is running. On return from the handler function, the process's previous signal mask is restored.

The *sa_flags* member is used to specify various other things which we describe in upcoming slides.

Note that, in general, *sigaction* is preferred over *signal* (and *sigset*). This is partly because there is no general agreement as to what *signal* and *sigset* actually do. However, *signal* works fine on Linux and we will use it in examples, mainly because it takes less space on slides than does *sigaction*. But you should normally use *sigaction* in the code you write.

Example

```
int main() {
  void handler(int);
  struct sigaction act;
  act.sa_handler = handler;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);

  while(1)
    ;
  return 1;
}

void handler(int signo) {
  printf("I received signal %d. "
     "Whoopee!!\n", signo);
}
```


This has behavior identical to the previous example; we're using sigaction rather than signal to set up the signal handler.

Getting More Out of Signals (1)

- Getting more than the signal number
  - for example, which arithmetic problem caused a SIGFPE?
- Use sa_sigaction rather than sa_handler

```
struct sigaction act;
act.sa_sigaction = arith_error;
   /* not sa_handler! */
sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;
   /* means that we're using sa_sigaction */
sigaction(SIGFPE, &act, 0);
```


Getting More Out of Signals (2)

```
void arith_error(int signo, siginfo_t *infop,
      void *ctx) {

   if (infop->si_code == FPE_INTDIV) {
      /* deal with integer divide by zero */
      …
   }
   …
}
```


The slide illustrates the signature of the handler procedure used with *siginfo*, as well as a partial example of its use. The third parameter is, on some implementations (but not on Linux), a pointer to a structure of type *ucontext_t* and contains the register context of the process at the time of interruption by the signal. We won't be discussing it further in this course, but information about its use can be found in the man page for *ucontext*.

The *siginfo* structure (of type *siginfo_t*) contains the following:

```
int           si_signo    /* signal number */
int           si_errno    /* error number */
int           si_code     /* signal code */
union sigval  si_value    /* signal value */
```

- if *si_errno* is not zero, it contains whatever error code is associated with the signal
- if *si_code* is positive, the signal was generated in response to a kernel-detected event (such as an arithmetic exception) indicated by *si_code* (there are a great number of possibilities—see *siginfo*'s man page for details)
  - *si_value* may contain other useful information, such as the problem address in the case of SIGSEGV and SIGBUS, and the child's PID and status in the case of SIGCHLD
- if *si_code* is less than or equal to zero, then the signal was generated by a user process via the *kill* system call
  - *si_value* contains the signaller's PID and UID, which can be referenced as:

```
pid_t   si_pid
uid_t   si_uid
```

Waiting for a Signal …

```
signal(SIGALRM, RespondToSignal);

…

struct timeval waitperiod = {0, 1000};
      /* seconds, microseconds */
struct timeval interval = {0, 0};
struct itimerval timerval;
timerval.it_value = waitperiod;
timerval.it_interval = interval;

setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */
pause();  /* wait for it */
```


Here we use the *setitimer* system call to arrange so that a SIGALRM signal is generated in one millisecond. (The system call takes three arguments: the first indicates how time should be measured; what's specified here is to use real time. See its man page for other possibilities. The second argument contains a *struct itimerval* that itself contains two *struct timeval*s. One (named *it_value*) indicates how much time should elapse before a SIGALRM is generated for the process. The other (named *it_interval*), if non-zero, indicates that a SIGALRM should be sent again, repeatedly, every it_interval period of time. Each process may have only one pending timer, thus when a process calls setitimer, the new value replaces the old. If the third argument to setitimer is non-zero, the old value is stored at the location it points to.)

The *pause* system call causes the process to block and not resume until *any* signal that is not ignored is delivered.

Note that there is a race condition here: it's possible that the SIGALRM might be delivered after the process calls *setitimer*, but before it calls *pause* (the system might be very busy). If this were to happen, then the process might get "stuck" within *pause*, since no other signals are forthcoming.

Doing It Safely

```
sigset_t set, oldset;
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, &oldset);
      /* SIGALRM now masked */
…
setitimer(ITIMER_REAL, &timerval, 0);
      /* SIGALRM sent in ~one millisecond */

sigsuspend(&oldset);    /* wait for it safely */
      /* SIGALRM masked again */
…

sigprocmask(SIG_SETMASK, &oldset, (sigset_t *)0);
      /* SIGALRM unmasked */
```


Here's a safer way of doing what was attempted in the previous slide. We mask the SIGALRM signal before calling *setitimer*. Then, rather than calling *pause*, we call *sigsuspend*, which sets the set of masked signals to its argument and, at the same instant, blocks the calling process. Thus if the SIGALRM is generated before our process calls *sigsuspend*, it won't be delivered right away. Since the call to *sigsuspend* reinstates the previous mask (which, presumably, did not include SIGALRM), the SIGALRM signal will be delivered and the process will return (after invoking the handler). When *sigsuspend* returns, the signal mask that was in place just before it was called is restored. Thus we have to restore *oldset* explicitly.

As with *pause*, *sigsuspend* returns only if an unmasked signal that is not ignored is delivered.

Signal Sets

- **To clear a set:**

```
int sigemptyset(sigset_t *set);
```

- **To add or remove a signal from the set:**

```
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
```

- **Example: to refer to both SIGHUP and SIGINT:**

```
sigset_t set;

sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
```


A number of signal-related operations involve sets of signals. These sets are normally represented by a bit vector of type *sigset_t*.

Masking (Blocking) Signals

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
      sigset_t *old);
```

- used to examine or change the signal mask of the calling process
  - » *how* is one of three commands:
    - SIG_BLOCK
      - the new signal mask is the union of the current signal mask and set
    - SIG_UNBLOCK
      - the new signal mask is the intersection of the current signal mask and the complement of set
    - SIG_SETMASK
      - the new signal mask is set

In addition to ignoring signals, you may specify that they are to be blocked (that is, held pending or masked). When a signal type is masked, signals of that type remains pending and do not interrupt the process until they are unmasked. When the process unblocks the signal, the action associated with any pending signal is performed. This technique is most useful for protecting critical code that should not be interrupted. Also, as we've already seen, when the handler for a signal is entered, subsequent occurrences of that signal are automatically masked until the handler is exited, hence the handler never has to worry about being invoked to handle another instance of the signal it's already handling.

Timed Out!

```
int TimedInput( ) {
   signal(SIGALRM, timeout);
   …
   alarm(30);    /* send SIGALRM in 30 seconds */
   GetInput();   /* possible long wait for input */
   HandleInput();
   return(0);
nogood:
   return(1);
}

void timeout( ) {
   goto nogood;   /* not legal but straightforward */
}
```


This slide sketches something that one might want to try to do: give a user a limited amount of time (in this case, 30 seconds — the *alarm* routine causes the system to send the process a SIGALRM signal in the given number of seconds) to provide some input, then, if no input, notify the caller that there is a problem. Here we'd like our timeout handler to transfer control to someplace else in the program, but we can't do this. (Note also that we should cancel the call to *alarm* if there is input. So that we can fit all the code in the slide, we've left this part out.)

Doing It Legally

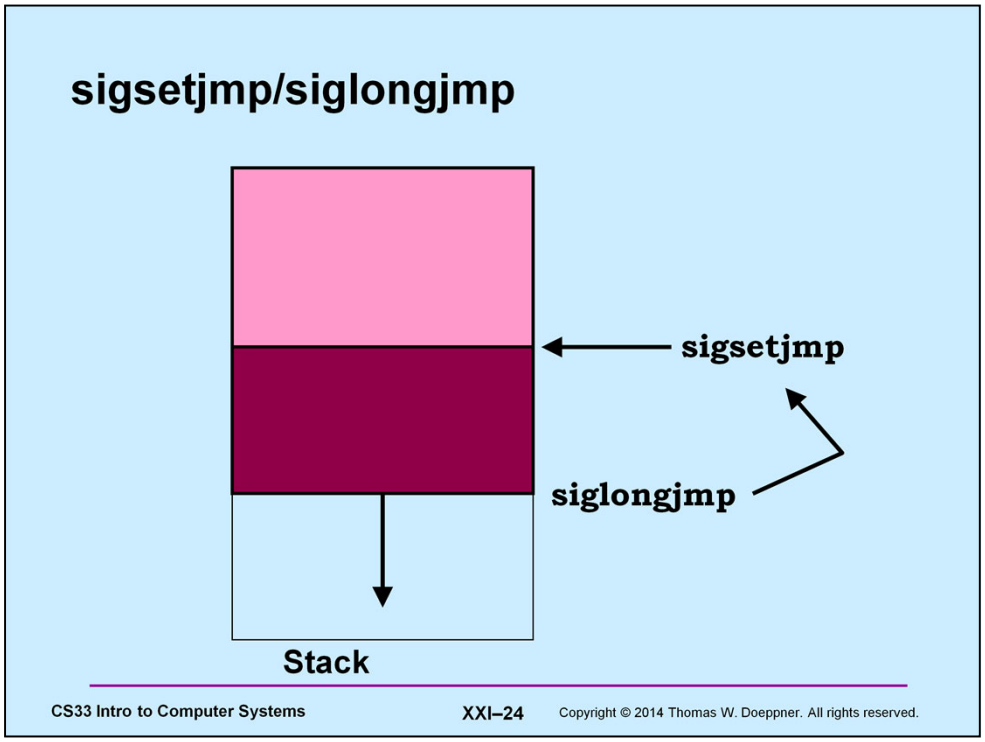
```
sigjmp_buf context;

int TimedInput( ) {
   signal(SIGALRM, timeout);
   if (sigsetjmp(&context, 1) == 0) {
      alarm(30);
      GetInput(); /* possible long wait for input */
      HandleInput();
      return(0);
   } else
      return(1);
}

void timeout() {
   siglongjmp(&context, 1);   /* legal but weird */
}
```


To get around the problem of not being able to use a *goto* statement to get out of a signal handler, we introduce the setjmp/longjmp facility, also known as the *nonlocal goto*. A call to *sigsetjmp* stores context information (about the current locus of execution) that can be restored via a call to *siglongjmp*. A bit more precisely: *sigsetjmp* stores into its first argument the values of the program-counter (instruction-pointer), stack-pointer, and other registers representing the process's current execution context. If the second argument is non-zero, the current signal mask is saved as well. The call returns 0. When *siglongjmp* is called with a pointer to this context information as its first argument, the current register values are replaced with those that were saved. If the signal mask was saved, that is restored as well. The effect of doing this is that the process resumes execution where it was when the context information was saved: inside of sigsetjmp. However, this time, rather than returning zero, it returns the second argument passed to *siglongjmp* (1 in the example).

To use this facility, you must include the header file *setjmp.h*.



The effect of *sigsetjmp* is, roughly, to save a description of the current stack location. A subsequent call to *siglongjmp* restores the stack to where it was at the time of the call to *sigsetjmp*. Note that *siglongjmp* should be called only from a stack frame that is farther on the stack than the one in which *sigsetjmp* was called.

Signals, Fork, and Exec

```
// set up signal handlers ...
if (fork() == 0) {
   // what happens if child gets signal?
   ...
   signal(SIGINT, SIG_IGN);
   signal(SIGFPE, handler);
   signal(SIGQUIT, SIG_DFL);
   execv("new prog", arg, NULL);
   // what happens if SIGINT, SIGFPE,
   // or SIGQUIT occur?
}
```


As makes sense, the signal-handling state of the parent is reproduced in the child.

What also makes sense is that, if a signal has been given a handler, then, after an exec, when the handler no longer exists, the signal reverts to default actions.

What at first glance makes less sense is that ignored signals stay ignored after an exec (but signals with default action stay that way after the exec). The intent is that this allows one to run a program protected from certain signals.

Dealing with Failure

- ***fork***, ***execv***, ***wait***, ***kill*** **directly invoke the operating system**
- **Sometimes the OS says no**
  - **usually because you did something wrong**
  - **sometimes because the system has run out of resources**
  - **system calls return -1 to signify a problem**

Reporting Failure

- Integer error code placed in global variable *errno*

```
extern int errno;
```

  - "man 3 errno" lists all possible error codes and meanings
  - to print out most recent error

```
perror("message");
```


Fork

```
int main( ) {
  pid_t pid;
  while(1) {
    if ((pid = fork()) == -1) {
      perror("fork");
      exit(1);
    }
    ...
  }
}
```


Exec

```
int main( ) {
  if (fork() == 0) {
      char *argv[] = {"garbage", 0};
      execv("/garbage", argv);
      /* if we get here, there was an
         error! */
      perror("execv");
      exit(1);
  }
}
```

Signals and Blocking System Calls

- What if a signal is generated while a process is blocked in a system call?
  1) deal with it when the system call completes
  2) interrupt the system call, deal with signal, resume system call

  or

  3) interrupt system call, deal with signal, return from system call with indication that something happened

The kernel normally checks for pending, unmasked signals when a process is returning to user mode from privileged mode. However, if a process is blocked in a system call, it might be a long time until it returns and notices the signal. If the blocking time is guaranteed to be short (e.g., waiting for a disk operation to complete), then it makes sense to postpone handling the signal until the system call completes. Such waits and system calls are termed "non-interruptible." But if the wait could take a long time (e.g., waiting for something to be typed at the keyboard), then the signal should be dealt with as quickly as possible, which means that the process should be forced out of the system call.

What happens to the system call after the signal handling completes (assuming that the process has not been terminated)? One possibility is for the system to automatically restart it. However, it's not necessarily the case that it should be restarted — the signal may have caused the program to lose interest. Thus what's normally done for such "interruptible" system calls is that some indication of what has happened is passed to the program, as is shown in the next slide.

Interrupted System Calls

```
while(read(fd, buffer, buf_size) == -1) {
  if (errno == EINTR) {
    /* interrupted system call – try again */
    continue;
  }
  /* the error is more serious */
  perror("big trouble");
  exit(1);
}
```


If a system call is interrupted by a signal, the call fails and the error code is set to *errno*. The process then executes the signal handler and then returns to the point of the interrupt, which causes it to (finally) return from the system call with the error.

Interrupted While Underway

```
remaining = total_count;
bptr = buf;
for ( ; ; ) {
  num_xfrd = write(fd, bptr,
      remaining);
  if (num_xfrd == -1) {
    if (errno == EINTR) {
      /* interrupted early */
      continue;
    }
    perror("big trouble");
    exit(1);
  }
  if (num_xfrd < remaining) {
    /* interrupted after the
       first step */
    remaining -= num_xfrd;
    bptr += num_xfrd;
    continue;
  }
  /* success! */
  break;
}
```


However, the actions of some system calls are broken up into discrete steps. For example, if one issues a system call to write a megabyte of data to a file, the write will actually be split by the kernel into a number of smaller writes. If the system call is interrupted by a signal after the first component write has completed (but while there are still more to be done), it would not make sense for the call to return an error code: such an error return would convince the program that none of the write had completed and thus all should be redone. Instead, the call completes successfully: it returns the number of bytes actually transferred, the signal handler is invoked, and, on return from the signal handler, the user program receives the successful return from the system call.

Automatic Restart

```
struct sigaction act;

act.sa_handler = reap_child;
act.sa_mask = 0;
act.sa_flags = SA_RESTART;
sigaction(SIGCHLD, &act, 0);
```

```
remaining = total_count;
bptr = buf;
while ((num_xfrd =
    write(fd, bptr, remaining))
      != remaining) {
  if (num_xfrd == -1) {
    /* no EINTR from SIGCHLD */
    …
    break;
  }
  /* still must deal with
     partial completions */
  remaining -= num_xfrd;
  bptr += num_xfrd;
}
```


Sometime it's convenient to specify that system calls be automatically restarted when a particular signal occurs. For example, in the slide we've done this for the SIGCHLD signal by setting the SA_RESTART flag in the *sigaction* structure. However, automatic restart applies only if the system call was interrupted before any transfer took place. We still must deal with the case of a partial completion.

Note that the SIGCHLD signal, whose default action is to be ignored, is sent when a child process terminates or otherwise changes its status.

Asynchronous Signals (1)

```
main( ) {
   void handler(int);
   signal(SIGINT, handler);

   ...   /* long-running buggy code */

}

void handler(int sig) {
   ...   /* die gracefully */
   exit(1);
}
```


Let's look at some of the typical uses for asynchronous signals. Perhaps the most common is to force the termination of the process. When the user types control-C, the program should terminate. There might be a handler for the signal, so that the program can clean up and then terminate.

Asynchronous Signals (2)

```
computation_state_t  state;

main( ) {
  void handler(int);

  signal(SIGINT, handler);

  long_running_procedure( );
}
```

```
long_running_procedure( ) {
  while (a_long_time) {
    update_state(&state);
    compute_more( );
  }
}

void handler(int sig) {
  display(&state);
}
```


Here we are using a signal to send a request to a running program: when the user types control-C, the program prints out its current state and then continues execution. If synchronization is necessary so that the state is printed only when it is stable, it must be provided by appropriate settings of the signal mask.

Asynchronous Signals (3)

```
main( ) {
  void handler(int);

  signal(SIGINT, handler);

  ... /* complicated program */

  myput("important message\n");

  ... /* more program */

}
```

```
void handler(int sig) {

  ... /* deal with signal */

  myput("equally important "
      "message\n";
}
```


In this example, both the mainline code and the signal handler call *myput*, which is similar to the standard-I/O routine *puts*. It's possible that the signal invoking the handler occurs while the mainline code is in the midst of the call to *myput*. Could this be a problem?

Asynchronous Signals (4)

```
void myput(char *str) {
  static char buf[BSIZE];
  static int pos=0;
  int i;
  int len = strlen(str);
  for (i=0; i<len; i++, pos++) {
    buf[pos] = str[i];
    if ((buf[pos] == '\n') || (pos == BSIZE-1)) {
      write(1, buf, pos+1);
      pos = -1;
    }
  }
}
```


Here's the implementation of *myput*, used in the previous slide. What it does is copy the input string, one character at a time, into buf, which is of size BSIZE. Whenever a newline character is encountered, the current contents of buf up to that point are written to standard output, then subsequent characters are copied starting at the beginning of buf. Similarly, if buf is filled, its contents are written to standard output and subsequent characters are copied starting at the beginning of buf. Since buf is static, characters not written out may be written after the next call to myput. Note that printf (and other stdio routines) buffers output in a similar way.

The point of *myput* is to minimize the number of calls to write, so that write is called only when we have a complete line of text or when its buffer is full.

However, consider what happens if execution is in the middle of *myput* when a signal occurs, as in the previous slide. Among the numerous problem cases, suppose *myput* is interrupted just after *pos* is set to -1 (if the code hadn't have been interrupted, *pos* would be soon incremented by 1). The signal handler now calls *myput*, which copies the first character of *str* into *buf[pos]*, which, in this case, is *buf[-1]*. Thus the first character "misses" the buffer. At best it simply won't be printed, but there might well be serious damage done to the program.

Async-Signal Safety

- **Which library routines are safe to use within signal handlers?**

- abort
- accept
- access
- aio_error
- aio_return
- aio_suspend
- alarm
- bind
- cfgetispeed
- cfgetospeed
- cfsetispeed
- cfsetospeed
- chdir
- chmod
- chown
- clock_gettime
- close
- connect
- creat
- dup
- dup2
- execle
- execve
- _exit
- fchmod
- fchown
- fcntl
- fdatasync
- fork
- fpathconf
- fstat
- fsync
- ftruncate
- getegid
- geteuid
- getgid
- getgroups
- getpeername
- getpgrp
- getpid
- getppid
- getsockname
- getsockopt
- getuid
- kill
- link
- listen
- lseek
- lstat
- mkdir
- mkfifo
- open
- pathconf
- pause
- pipe
- poll
- posix_trace_event
- pselect
- raise
- read
- readlink
- recv
- recvfrom
- recvmsg
- rename
- rmdir
- select
- sem_post
- send
- sendmsg
- sendto
- setgid
- setpgid
- setsid
- setsockopt
- setuid
- shutdown
- sigaction
- sigaddset
- sigdelset
- sigemptyset
- sigfillset
- sigismember
- signal
- sigpause
- sigpending
- sigprocmask
- sigqueue
- sigsuspend
- sleep
- sockatmark
- socket
- socketpari
- stat
- symlink
- sysconf
- tcdrain
- tcflow
- tcflush
- tcgetattr
- tcgetpgrp
- tcsendbreak
- tcsetattr
- tcsetpgrp
- time
- timer_getoverrun
- timer_gettime
- timer_settime
- times
- umask
- uname
- unlink
- utime
- wait
- waitpid
- write

To deal with the problem on the previous page, we must arrange that signal handlers cannot destructively interfere with the operations of the mainline code. Unless we are willing to work with signal masks (which can be expensive), this means we must restrict what can be done inside a signal handler. Routines that, when called from a signal handler, do not interfere with the operation of the mainline code, no matter what that code is doing, are termed *async-signal safe*. The POSIX 1003.1 spec requires the routines shown in the slide to be async-signal safe.

Note that POSIX specifies only those routines that must be async-signal safe. Implementations may make other routines async-signal safe as well. In particular, pretty much all standard library routines are async-signal safe in Solaris (but this is not the case in Linux).