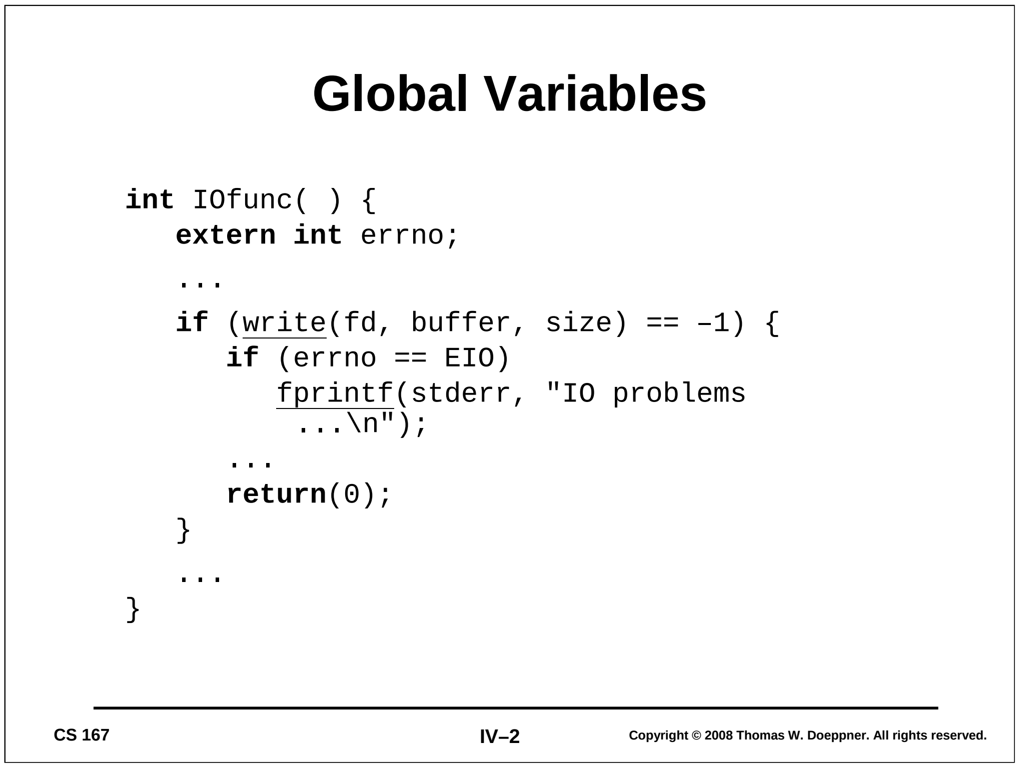
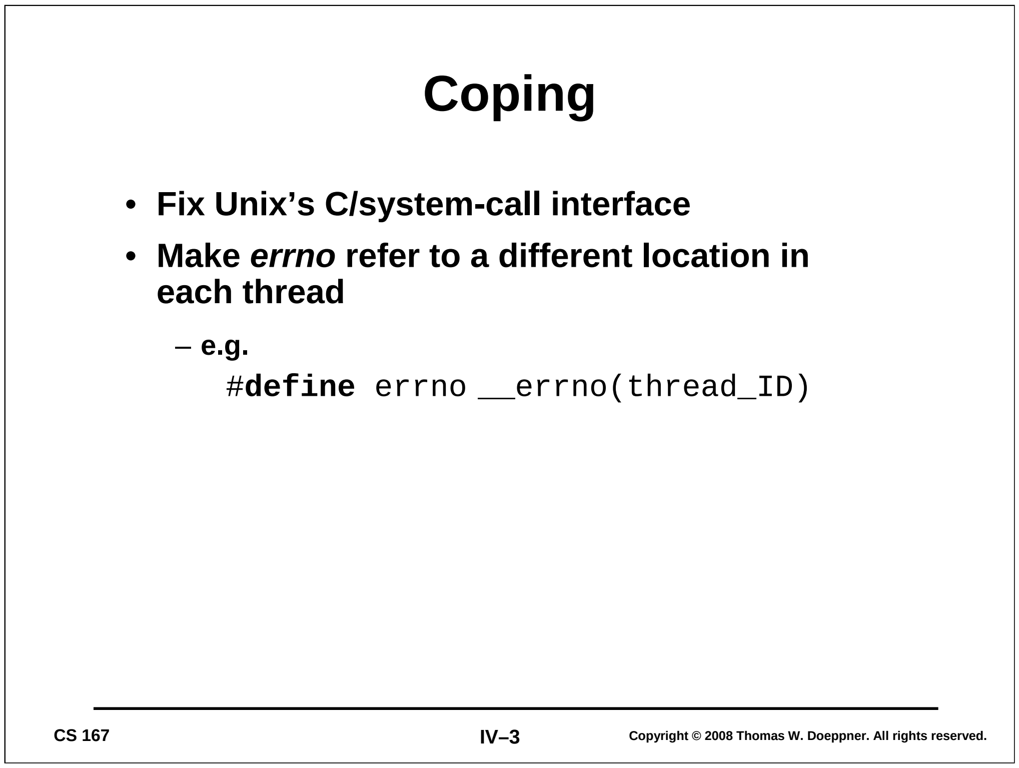
Programming with POSIX Threads II



Global Variables

```
int IOfunc( ) {
   extern int errno;
   ...
   if (write(fd, buffer, size) == –1) {
      if (errno == EIO)
         fprintf(stderr, "IO problems
          ...\n");
      ...
      return(0);
   }
   ...
}
```

Unix was not designed with multithreaded programming in mind. A good example of the implications of this is the manner in which error codes for failed system calls are made available to a program: if a system call fails, it returns –1 and the error code is stored in the global variable *errno*. Though this is not all that bad for single-threaded programs, it is plain wrong for multithreaded programs.



Coping

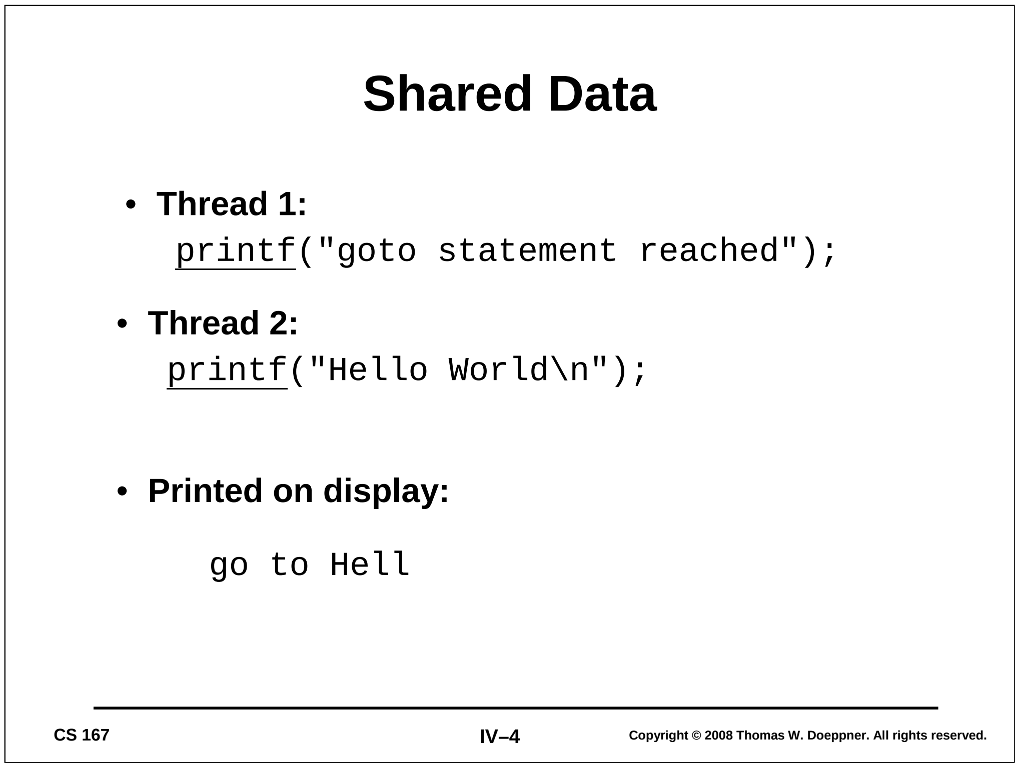
- **Fix Unix's C/system-call interface**
- **Make *errno* refer to a different location in each thread**
  - **e.g.**

```
#define errno __errno(thread_ID)
```

The ideal way to solve the "errno problem" would be to redesign the C/system-call interface: system calls should return only an error code. Anything else to be returned should be returned via result parameters. (This is how things are done in Windows NT.) Unfortunately, this is not possible (it would break pretty much every Unix program in existence).

So we are stuck with *errno*. What can we do to make *errno* coexist with multithreaded programming? What would help would be to arrange, somehow, that each thread has its own private copy of *errno*. I.e., whenever a thread refers to *errno*, it refers to a different location from any other thread when it refers to *errno*.

What can we do? As shown in the slide, we might use the C preprocessor to redefine *errno* as something a bit more complicated—references to *errno* result in accessing a function that retrieves this thread's private errno value. This is how things are actually done in Linux (and other implementations of POSIX threads). Please see the textbook for information on how this approach is generalized.



Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

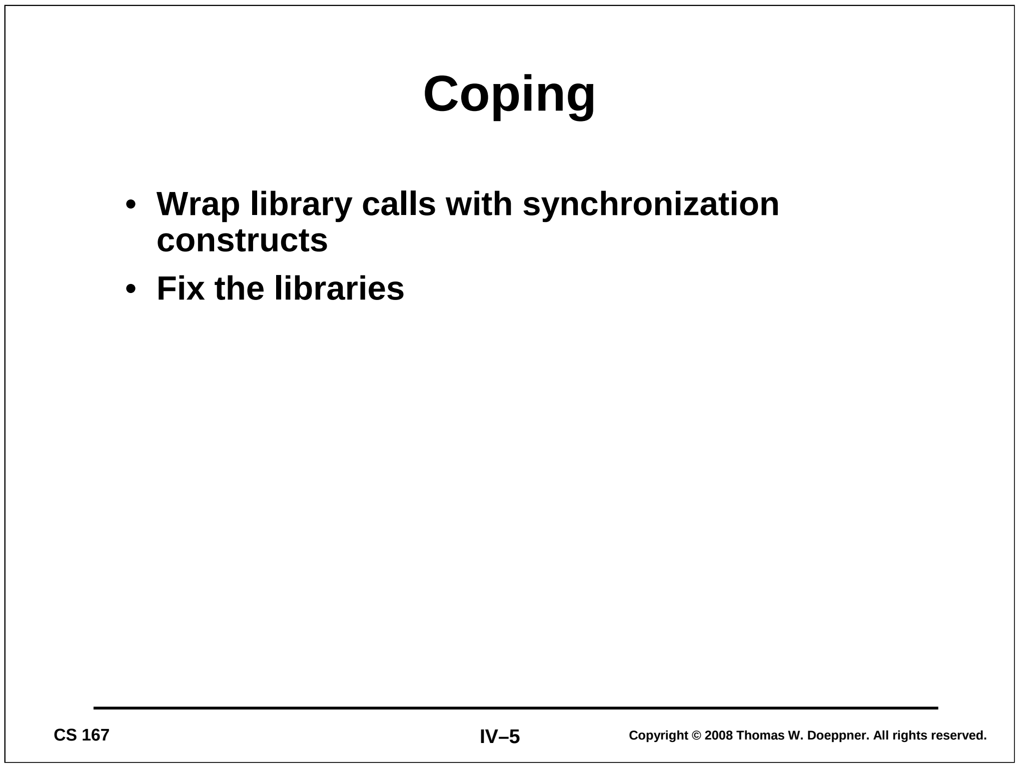
- **Thread 2:**

```
printf("Hello World\n");
```

- **Printed on display:**

```
go to Hell
```

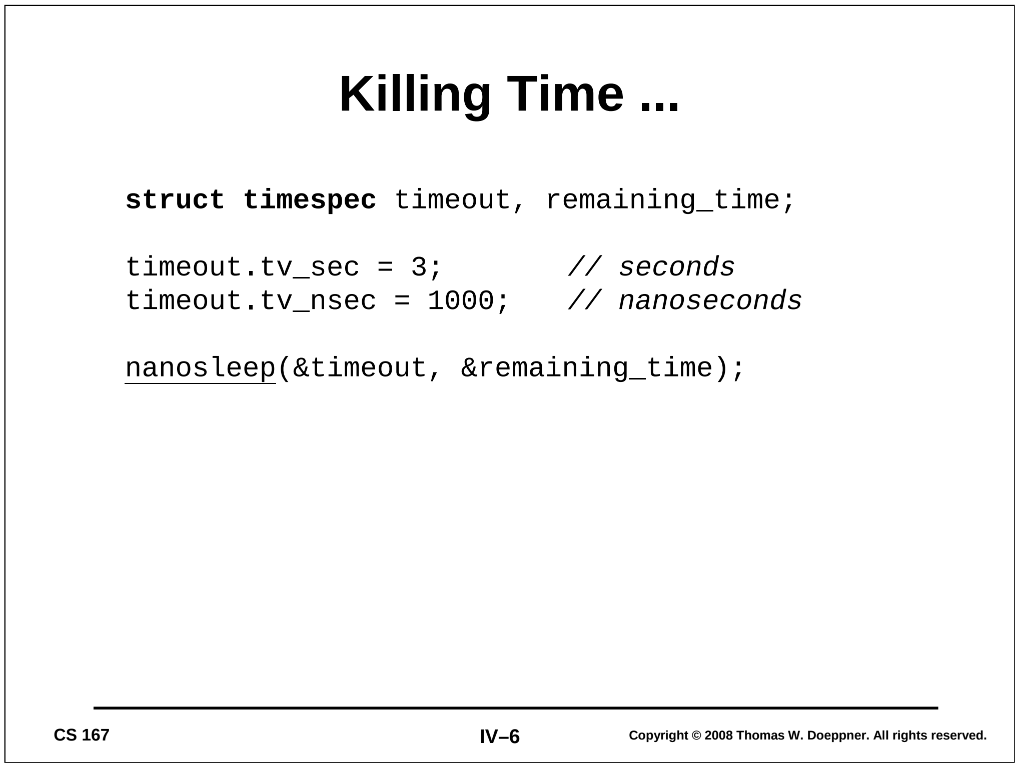
Yet another problem that arises when using libraries that were not designed for multithreaded programs concerns synchronization. The slide shows what might happen if one relied on the single-threaded versions of the standard I/O routines.



Coping

- **Wrap library calls with synchronization constructs**
- **Fix the libraries**

To deal with this *printf* problem, we must somehow add synchronization to *printf* (and all of the other standard I/O routines). A simple way to do this would be to supply wrappers for all of the standard I/O routines ensuring that only one thread is operating on any particular stream at a time. A better way would be to do the same sort of thing by fixing the routines themselves, rather than supplying wrappers (this is what is done in most implementations).

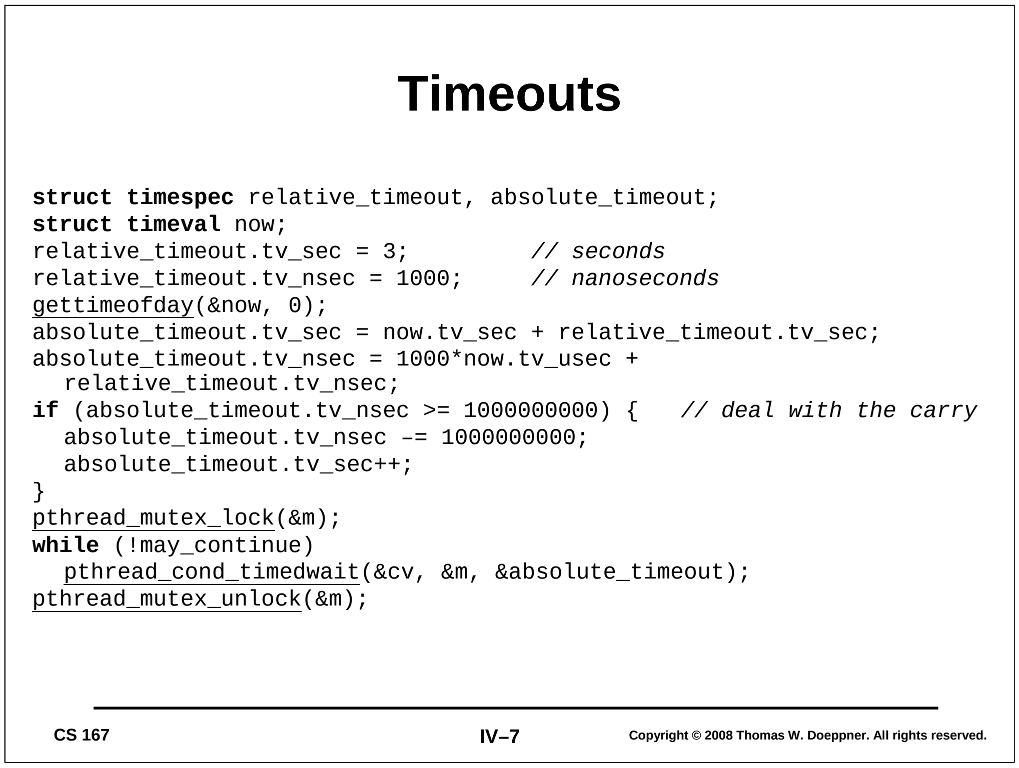


Killing Time ...

```
struct timespec timeout, remaining_time;

timeout.tv_sec = 3;       // seconds
timeout.tv_nsec = 1000;   // nanoseconds

nanosleep(&timeout, &remaining_time);
```

It is sometimes useful for a thread to wait for a certain period of time before continuing. The traditional Unix approach of using *alarm* and *SIGALRM* not only is not suitable for multithreaded programming, but also does not provide fine enough granularity. The routine *nanosleep* provides a better approach. A thread calls it with two arguments; the first indicates (in seconds and nanoseconds) how long the thread wishes to wait. The second argument is relevant only if the thread is interrupted by a signal: it indicates how much additional time remains until the originally requested time period expires.

Note that most Unix implementations do not have a clock that measures time in nanoseconds: the first argument to *nanosleep* is rounded up to an integer multiple of whatever sleep resolution is supported.

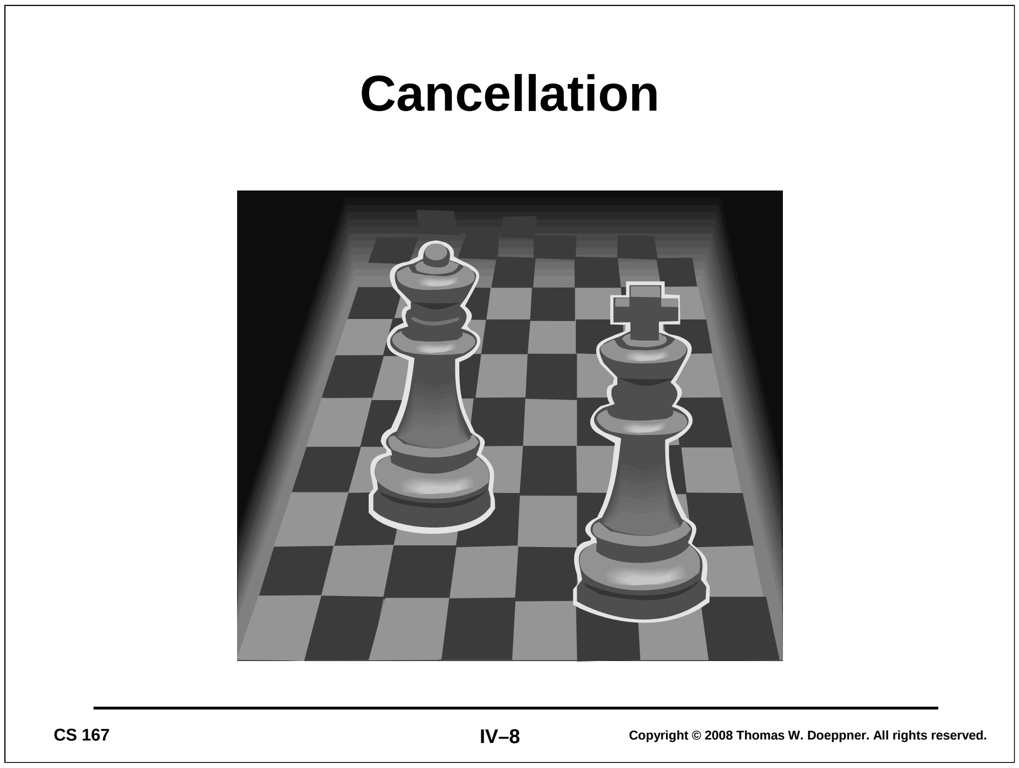


Timeouts

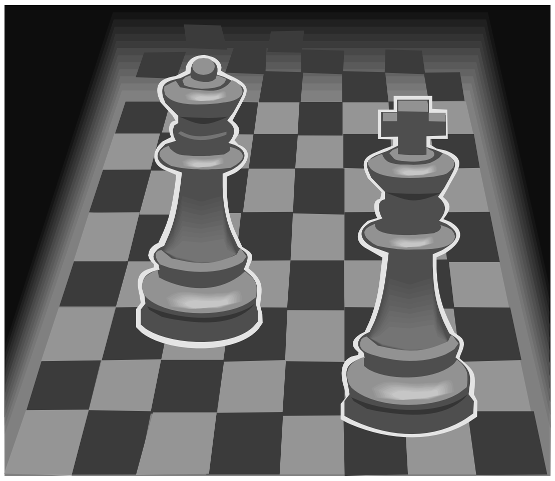
```
struct timespec relative_timeout, absolute_timeout;
struct timeval now;
relative_timeout.tv_sec = 3;         // seconds
relative_timeout.tv_nsec = 1000;     // nanoseconds
gettimeofday(&now, 0);
absolute_timeout.tv_sec = now.tv_sec + relative_timeout.tv_sec;
absolute_timeout.tv_nsec = 1000*now.tv_usec +
  relative_timeout.tv_nsec;
if (absolute_timeout.tv_nsec >= 1000000000) {   // deal with the carry
  absolute_timeout.tv_nsec -= 1000000000;
  absolute_timeout.tv_sec++;
}
pthread_mutex_lock(&m);
while (!may_continue)
  pthread_cond_timedwait(&cv, &m, &absolute_timeout);
pthread_mutex_unlock(&m);
```

POSIX threads provides a version of pthread_cond_wait that has a timeout: *pthread_cond_timedwait*. It takes an *additional argument indicating when the thread should give up on being awoken by a pthread_cond_signal*. This argument is an absolute time, as opposed to a relative time (as used in the previous slide); i.e., it is the clock time at which the call times out. To convert from an relative time to an absolute time, one must perform the machinations shown in the slide (or something similar)—note that *gettimeofday* returns seconds and microseconds, whereas *pthread_cond_timedwait* wants seconds and nanoseconds.

Why is it done this way? Though at first (and most subsequent) glances it seems foolish to require an absolute timeout value rather than a relative one, the use of the former makes some sense if you keep in mind that *pthread_cond_timedwait* could return with the "may_continue" condition false even before the timeout has expired (either because it's returned spontaneously or because the "may_continue" was falsified after the thread was released from the condition-variable queue). By having the timeout be absolute, there's no need to compute a new relative timeout when *pthread_cond_timedwait* is called again.

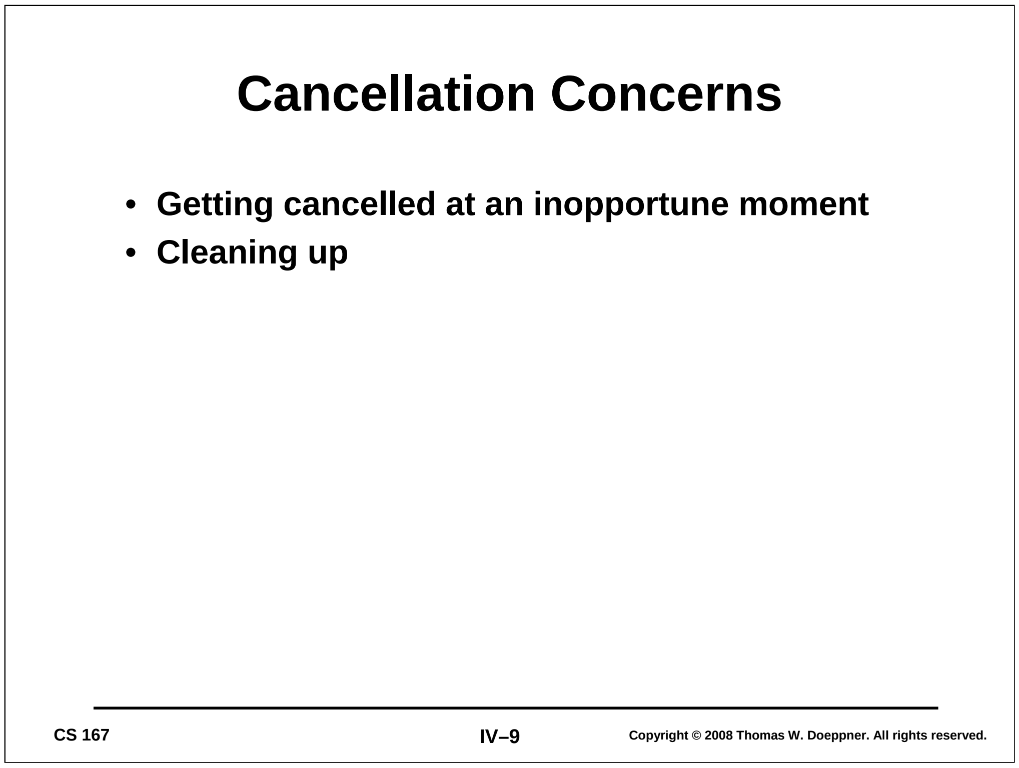


Cancellation



In a number of situations one thread must tell another to cease whatever it is doing. For example, suppose we've implemented a chess-playing program by having multiple threads search the solution space for the next move. If one thread has discovered a quick way of achieving a checkmate, it would want to notify the others that they should stop what they're doing, the game has been won.

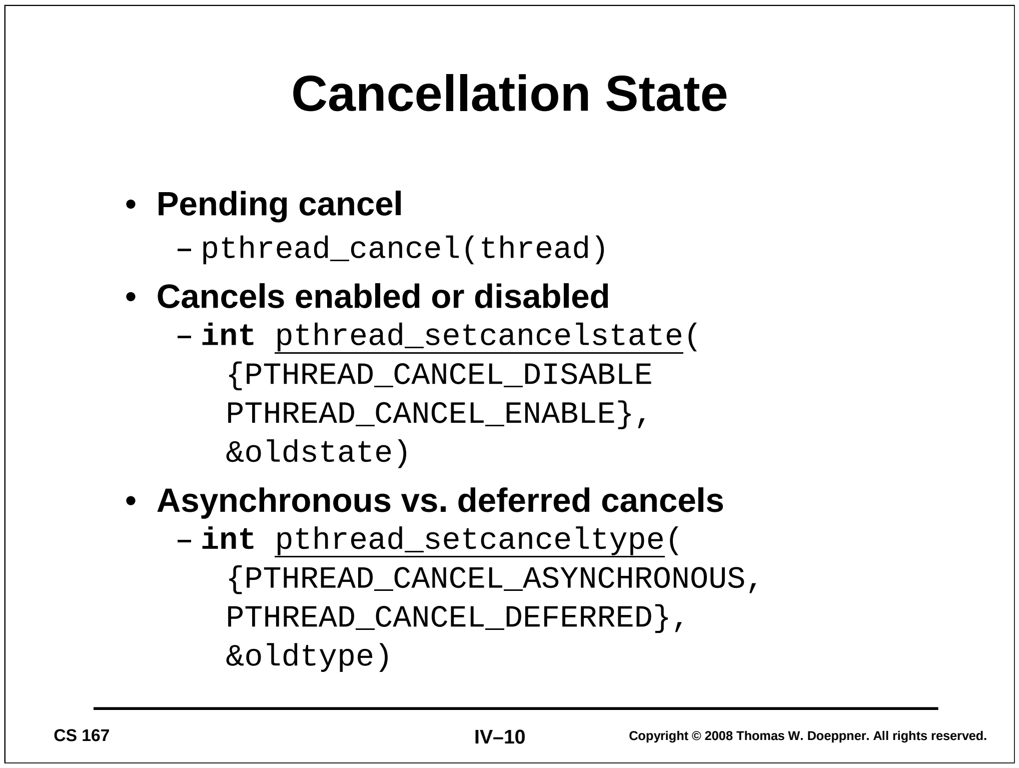
One might think that this is an ideal use for per-thread signals, but there's a cleaner mechanism for doing this sort of thing in POSIX threads, called cancellation.



Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning up

We have two concerns about the forced termination of threads resulting from cancellation: a thread might be in the middle of doing something important that it must complete before self-destructing; and a canceled thread must be given the opportunity to clean up.



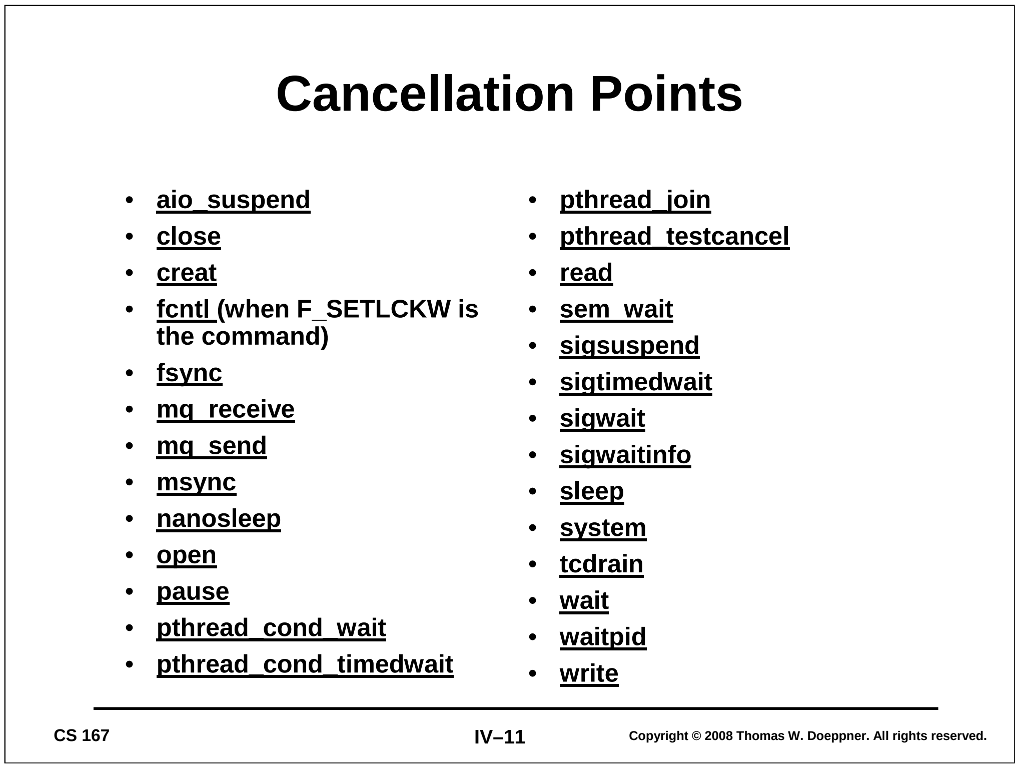
Cancellation State

- **Pending cancel**
  - `pthread_cancel(thread)`
- **Cancels enabled or disabled**
  -
```
int pthread_setcancelstate(
  {PTHREAD_CANCEL_DISABLE
  PTHREAD_CANCEL_ENABLE},
  &oldstate)
```
- **Asynchronous vs. deferred cancels**
  -
```
int pthread_setcanceltype(
  {PTHREAD_CANCEL_ASYNCHRONOUS,
  PTHREAD_CANCEL_DEFERRED},
  &oldtype)
```

A thread issues a cancel request by calling pthread_cancel, supplying the ID of the target thread as the argument. Associated with each thread is some state information known as its *cancellation state* and its *cancellation type*. When a thread receives a cancel request, it is marked indicating that it has a pending cancel. The next issue is when the thread should notice and act upon the cancel. This is governed by the cancellation state: whether cancels are *enabled* or *disabled* and by the cancellation type: whether the response to cancels is *asynchronous* or *deferred*. If cancels are disabled, then the cancel remains pending but is otherwise ignored until cancels are enabled. If cancels are *enabled*, they are acted on as soon as they are noticed if the cancellation type is *asynchronous*. Otherwise, i.e., if the cancellation type is *deferred*, the cancel is acted upon only when the thread reaches a *cancellation point*.

Cancellation points are intended to be well defined points in a thread's execution at which it is prepared to be canceled. They include pretty much all system and library calls in which the thread can block, with the exception of *pthread_mutex_lock*. In addition, a thread may call *pthread_testcancel*, which has no function other than being a cancellation point.

The default is that cancels are *enabled* and *deferred*. One can change the cancellation state of a thread by using the calls shown in the slide. Calls to *pthread_setcancelstate* and *pthread_setcanceltype* return the previous value of the affected portion of the cancellability state.

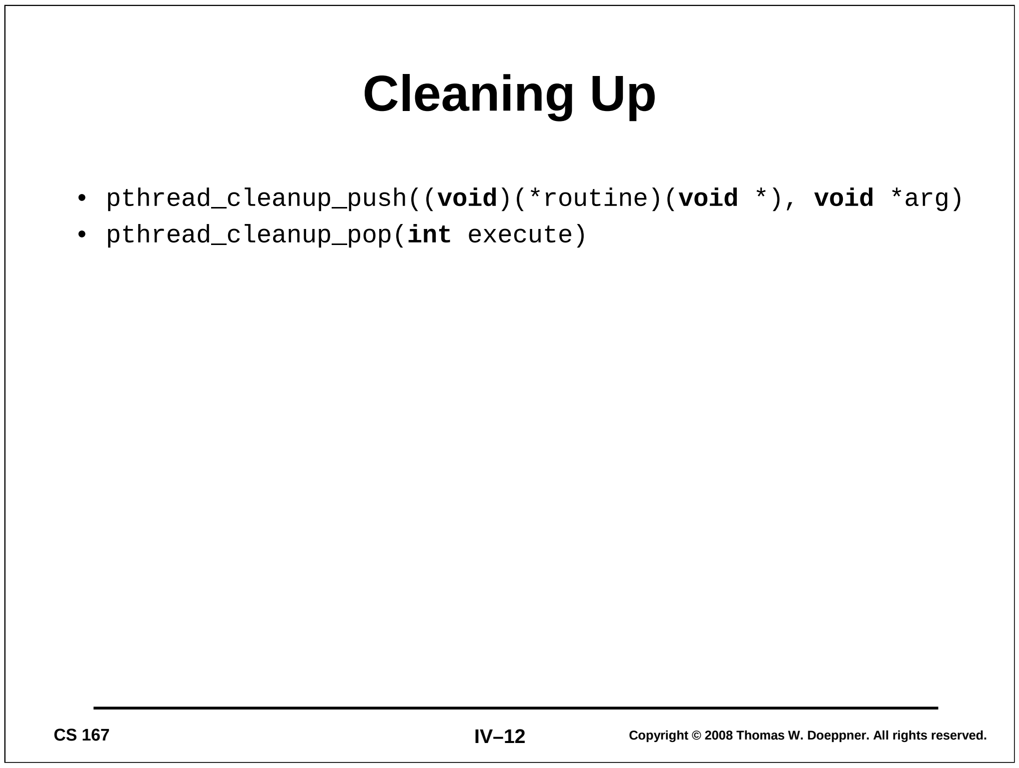


Cancellation Points

- aio_suspend
- close
- creat
- fcntl (when F_SETLCKW is the command)
- fsync
- mq_receive
- mq_send
- msync
- nanosleep
- open
- pause
- pthread_cond_wait
- pthread_cond_timedwait
- pthread_join
- pthread_testcancel
- read
- sem_wait
- sigsuspend
- sigtimedwait
- sigwait
- sigwaitinfo
- sleep
- system
- tcdrain
- wait
- waitpid
- write

CS 167 IV–11

The slide lists all of the required cancellation points in POSIX. Note, in particular, *pthread_testcancel*, whose effect is to be nothing but a cancellation point (i.e., if a cancel isn't pending, it does nothing).

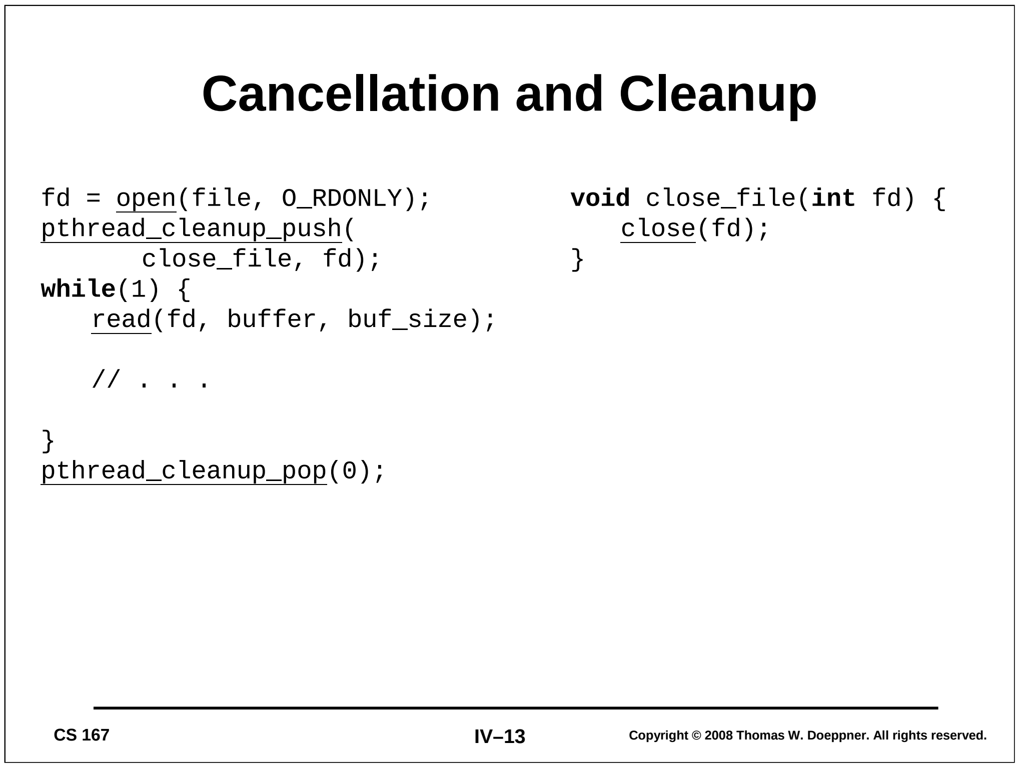


Cleaning Up

- `pthread_cleanup_push((void)(*routine)(void *), void *arg)`
- `pthread_cleanup_pop(int execute)`

When a thread acts upon a cancel, its ultimate fate has been established, but it first gets a chance to clean up. Associated with each thread may be a stack of cleanup handlers. Such handlers are pushed onto the stack via calls to *pthread_cleanup_push* and popped off the stack via calls to *pthread_cleanup_pop*. Thus when a thread acts on a cancel or when it calls *pthread_exit*, it calls each of the cleanup handlers in turn, giving the argument that was supplied as the second parameter of *pthread_cleanup_push*. Once all the cleanup handlers have been called, the thread terminates.

The two routines pthread_cleanup_push and *pthread_cleanup_pop* are intended to act as left and right parentheses, and thus should always be paired (in fact, they may actually be implemented as macros: the former contains an unmatched "{", the latter an unmatched "}"). The argument to the latter routine indicates whether or not the cleanup function should be called as a side effect of calling *pthread_cleanup_pop*.

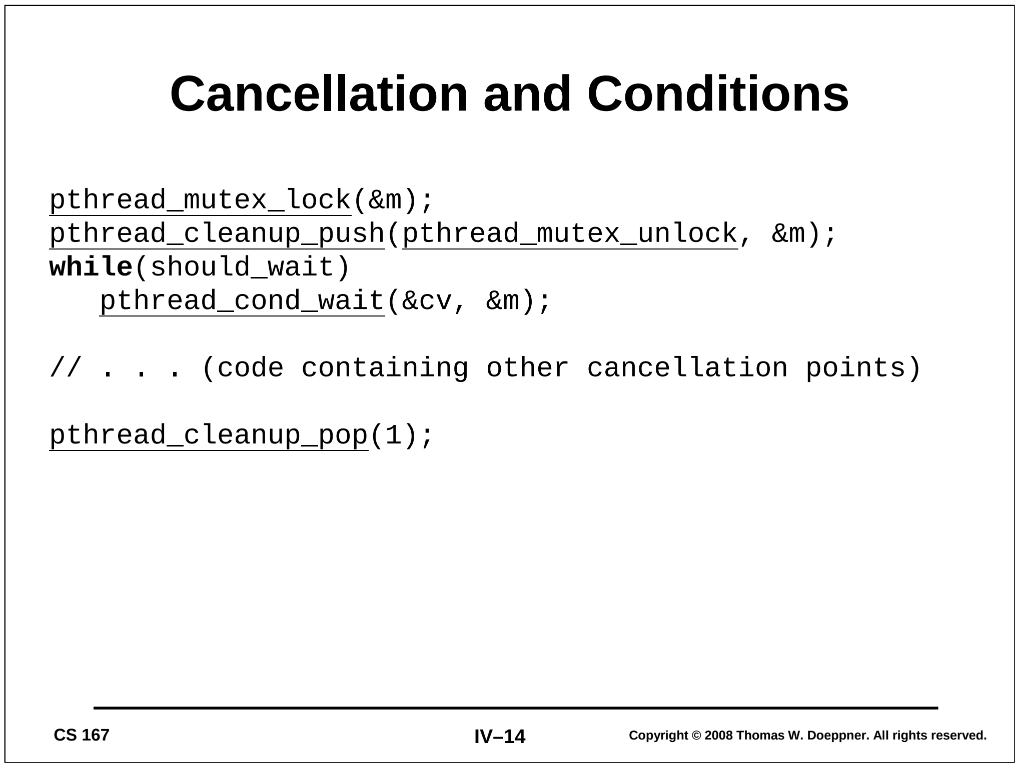


Cancellation and Cleanup

```
fd = open(file, O_RDONLY);
pthread_cleanup_push(
       close_file, fd);
while(1) {
   read(fd, buffer, buf_size);

   // . . .

}
pthread_cleanup_pop(0);
```

```
void close_file(int fd) {
   close(fd);
}
```

Here is a simple example of the use of a cleanup handler in conjunction with cancellation. The thread executing the code on the left, which has pushed a cleanup handler on its stack, makes successive calls to *read*, which is a cancellation point. Assuming the thread has cancellation enabled and set to deferred, if the thread is canceled, it will notice the cancel within *read*. It will then walk through its stack of cleanup handlers, first calling *close_file*, then any others that might be on the stack. Finally, the thread terminates.

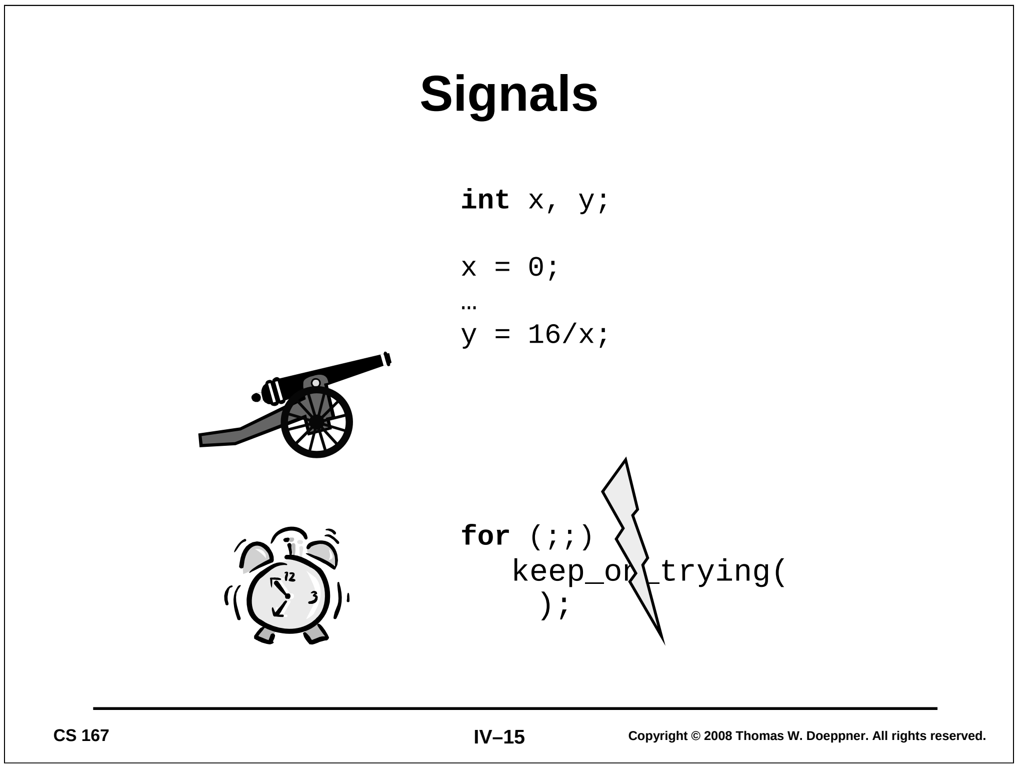


Cancellation and Conditions

```
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);
while(should_wait)
   pthread_cond_wait(&cv, &m);

// . . . (code containing other cancellation points)

pthread_cleanup_pop(1);
```

In this example we handle cancels that might occur while a thread is blocked within *pthread_cond_wait*. Again we assume the thread has cancels enabled and deferred. The thread first pushes a cleanup handler on its stack—in this case the cleanup handler unlocks the mutex. The thread then loops, calling *pthread_cond_wait*, a cancellation point. If it receives a cancel, the cleanup handler won't be called until the mutex has been reacquired. Thus we are certain that when the cleanup handler is called, the mutex is locked.

What's important here is that we make sure the thread does not terminate without releasing its lock on the mutex m. If the thread acts on a cancel within *pthread_cond_wait* and the cleanup handler were invoked without first taking the mutex, this would be difficult to guarantee, since we wouldn't know if the thread had the mutex locked (and thus needs to unlock it) when it's in the cleanup handler.

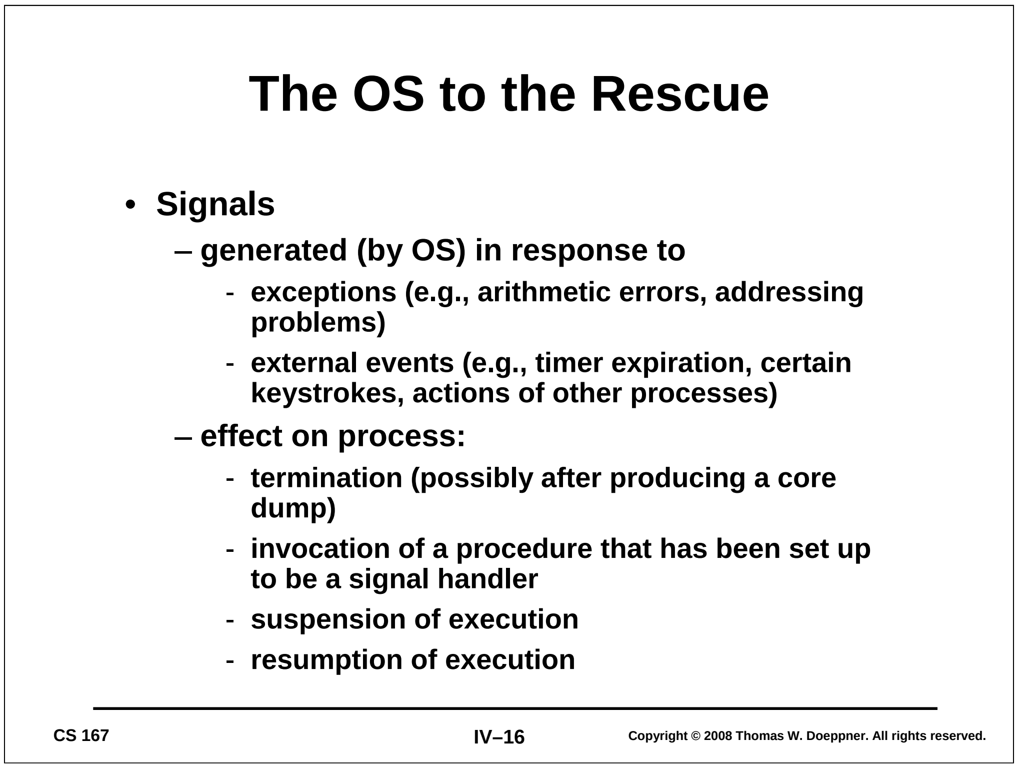


Signals

```
int x, y;

x = 0;
…
y = 16/x;
```

```
for (;;)
   keep_on_trying(
     );
```

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.



The OS to the Rescue

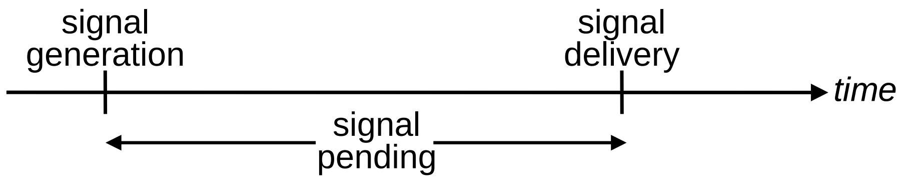
- Signals
  - generated (by OS) in response to
    - exceptions (e.g., arithmetic errors, addressing problems)
    - external events (e.g., timer expiration, certain keystrokes, actions of other processes)
  - effect on process:
    - termination (possibly after producing a core dump)
    - invocation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

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.

Signal Actions

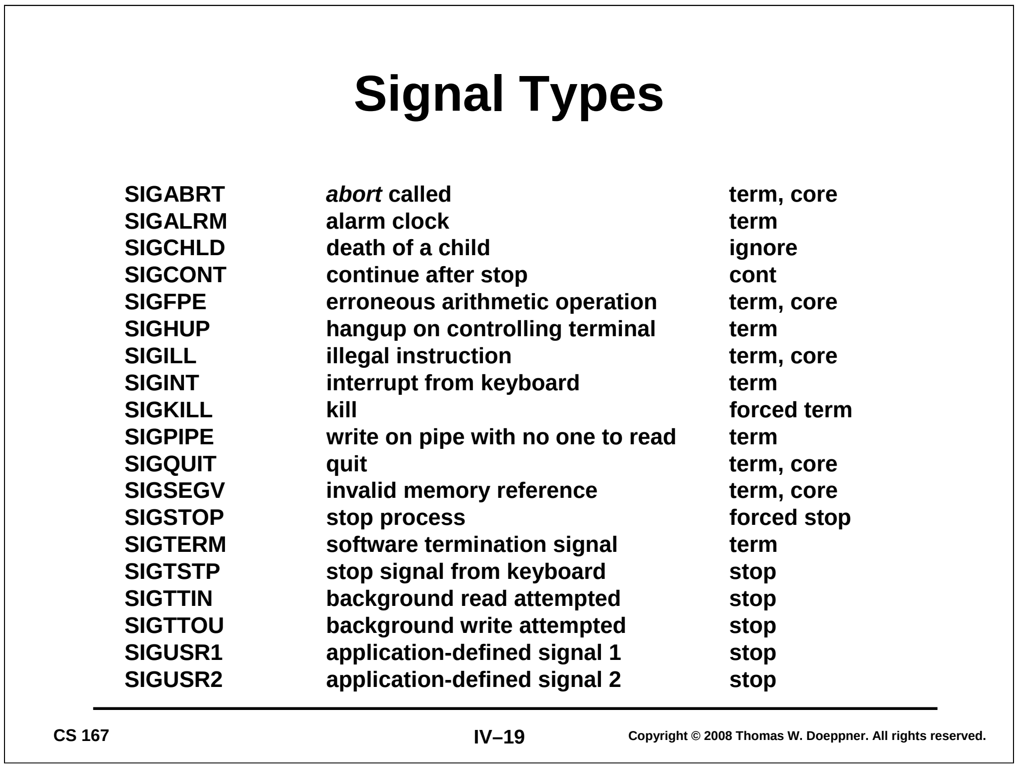
- For each signal type, a process can specify an action:
  - *abort* (with or without a core dump)
  - *ignore*
  - *hold*: temporarily ignore (delay delivery)
  - *catch*: call a signal handler function
- For each signal type, there is a default action

Terminology



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 indicated 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. 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.

Sending a Signal

- `int kill(pid_t pid, int sig)`
  - send signal *sig* to process *pid*
  - (not always) terminate with extreme prejudice
- Also
  - *kill* shell command
  - type ctrl-c
    - sends signal 2 (SIGINT) to current process
  - do something illegal
    - bad address, bad arithmetic, etc.

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);
```

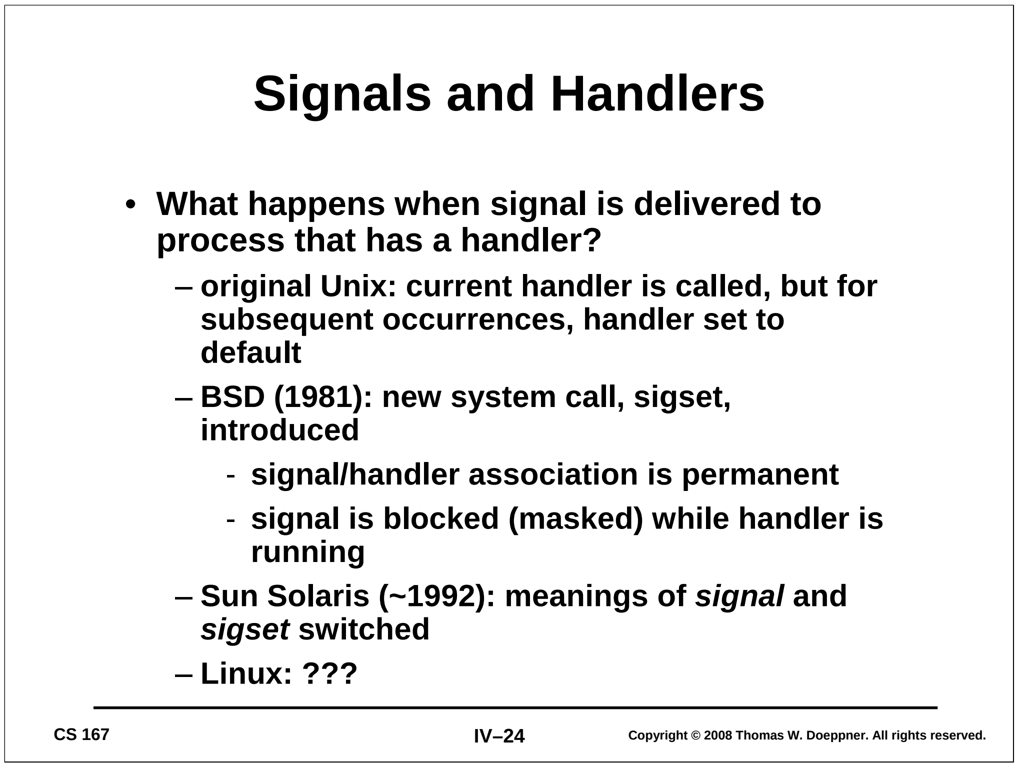
Special Handlers

- **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);
}
```

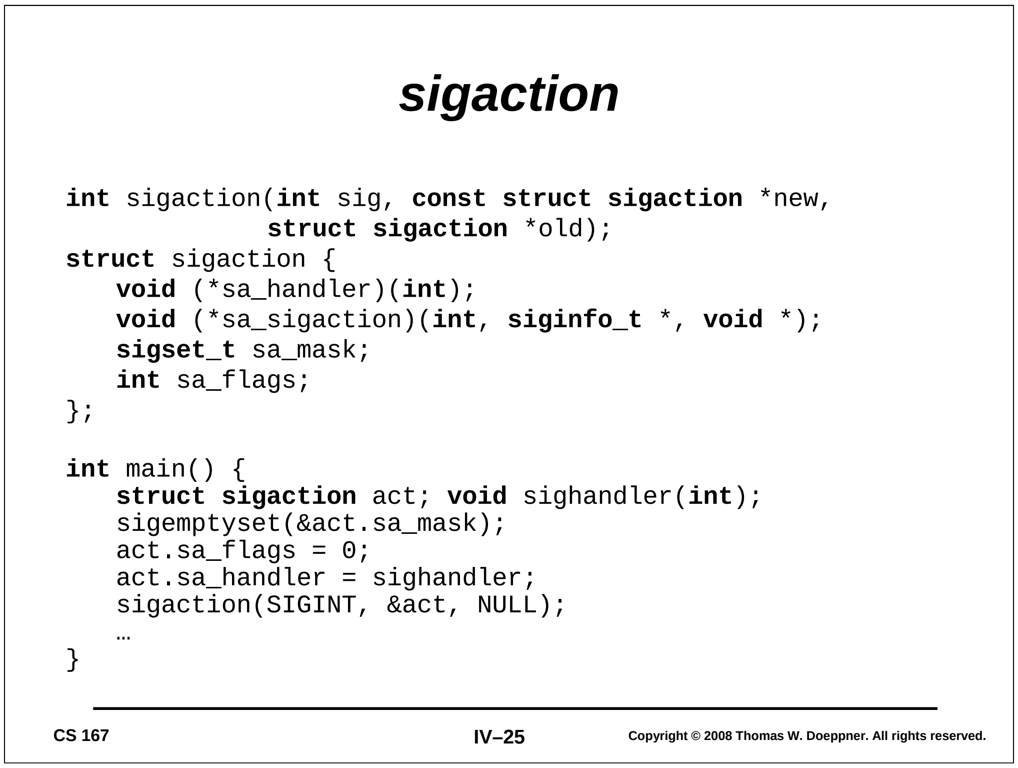
Note that the C compiler implicitly concatenates two adjacent strings, as done in printf above.



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: ???

Linux's man-page entry for *signal* is a bit ambiguous on whether it sets a signal's handler to default after an occurrence of the signal. In fact, it does not: once the handler is set, it stays set until explicitly changed.



sigaction

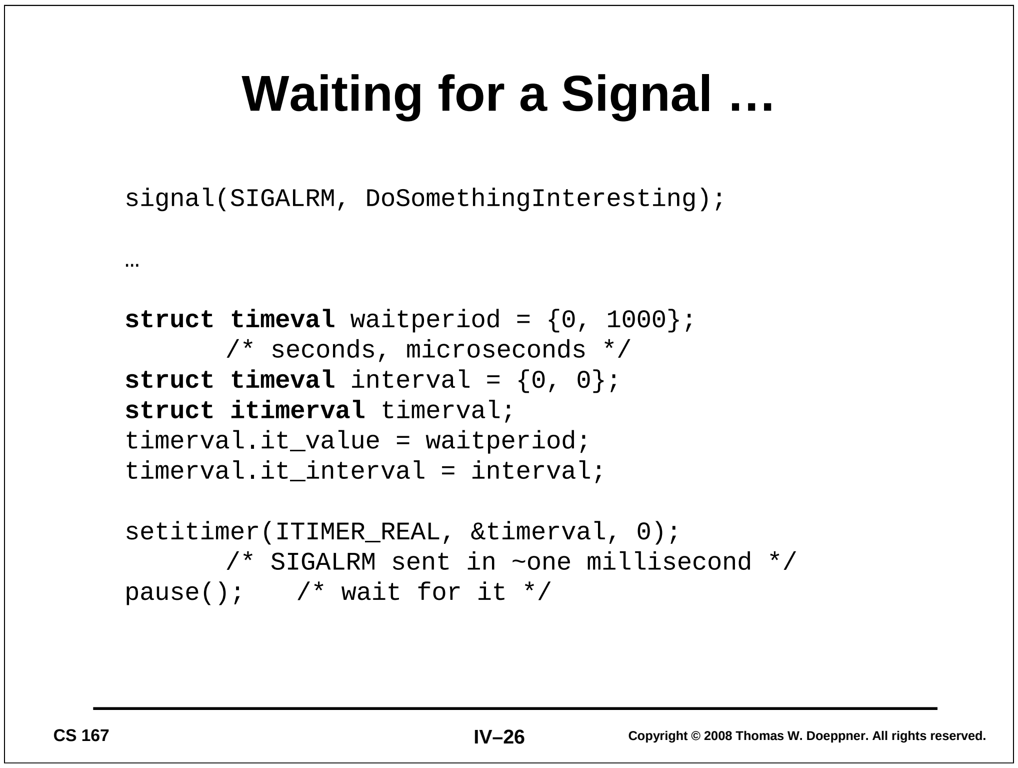
```
int sigaction(int sig, const struct sigaction *new,
              struct sigaction *old);
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int, siginfo_t *, void *);
   sigset_t sa_mask;
   int sa_flags;
};

int main() {
   struct sigaction act; void sighandler(int);
   sigemptyset(&act.sa_mask);
   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 starting in the next 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.



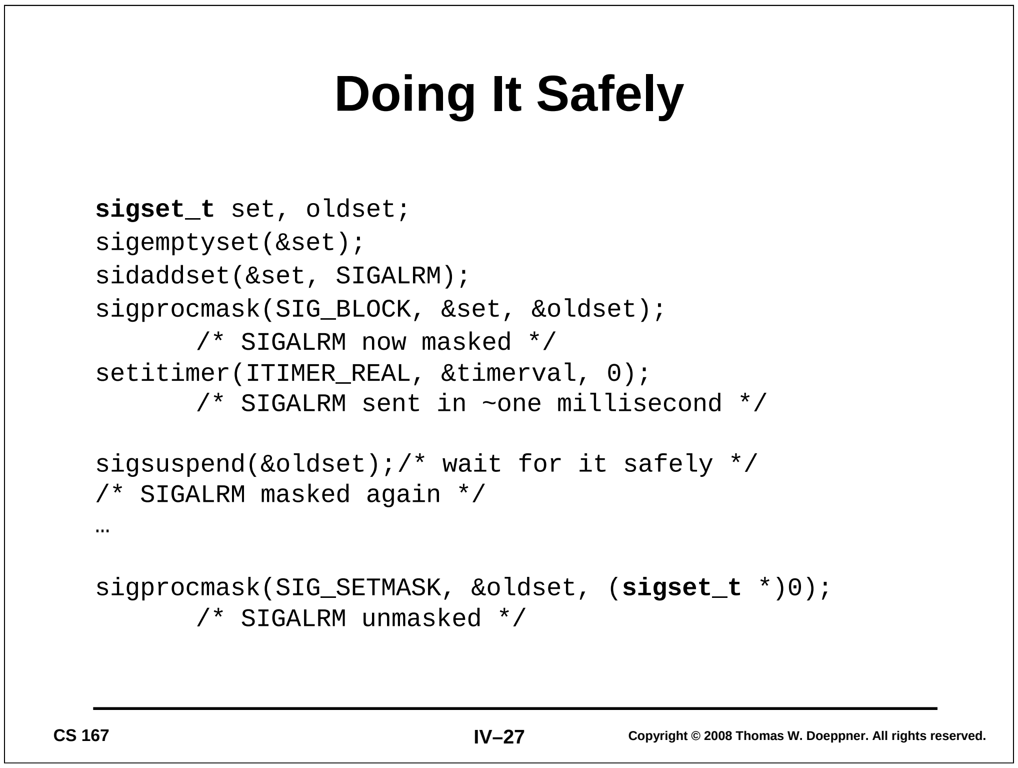
Waiting for a Signal …

```
signal(SIGALRM, DoSomethingInteresting);

…

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.

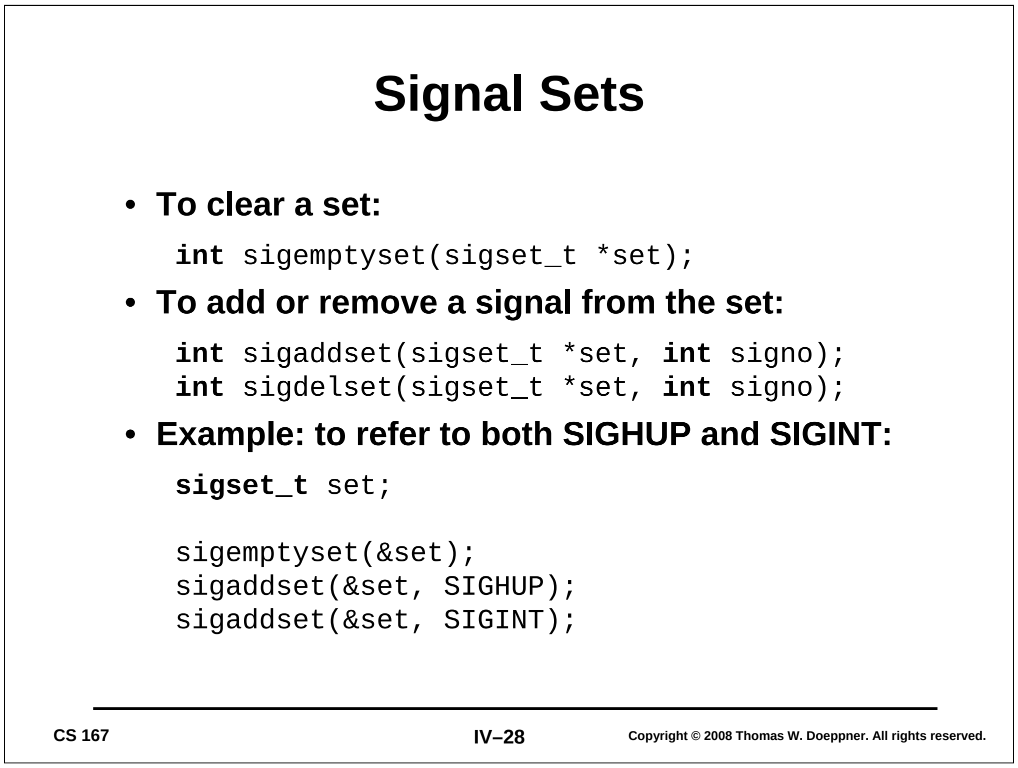


Doing It Safely

```
sigset_t set, oldset;
sigemptyset(&set);
sidaddset(&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 (*set*) 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);
```

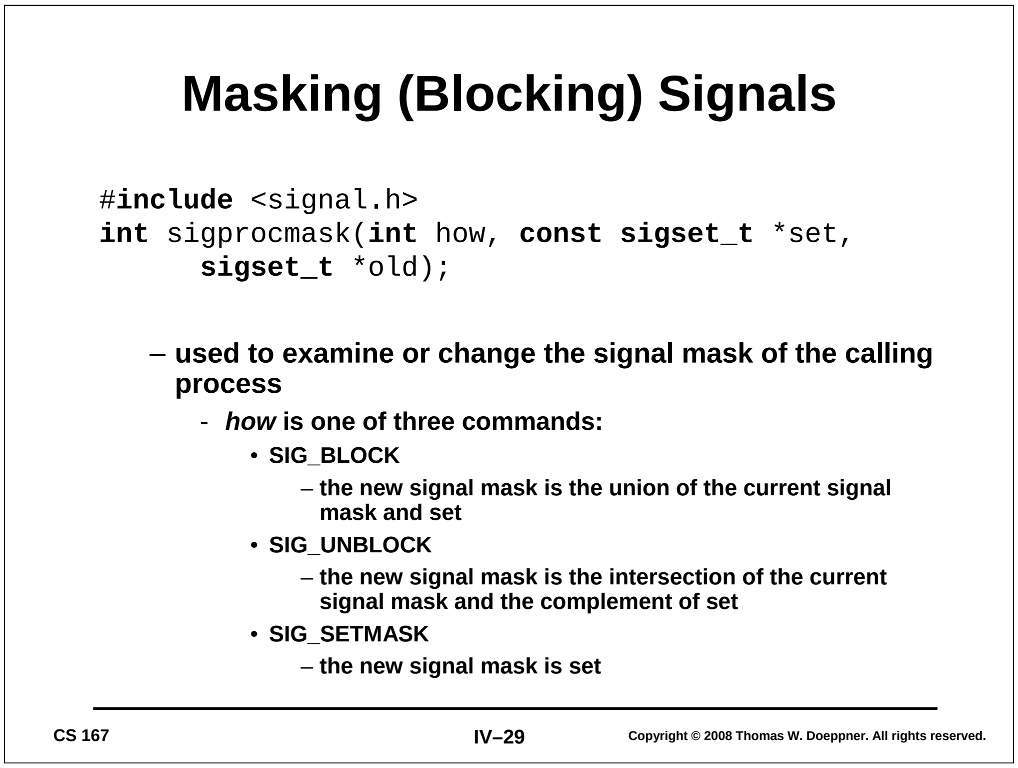
- **To add or remove a signal from the set:**

```
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*.

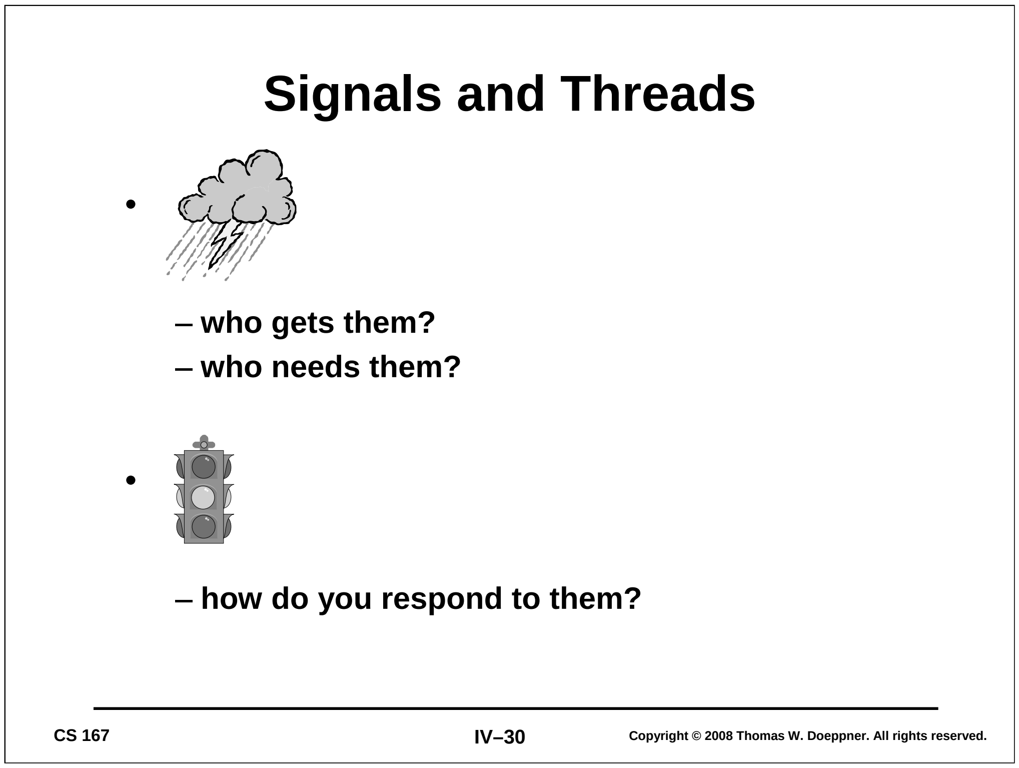


Masking (Blocking) Signals

```
#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.



Signals and Threads

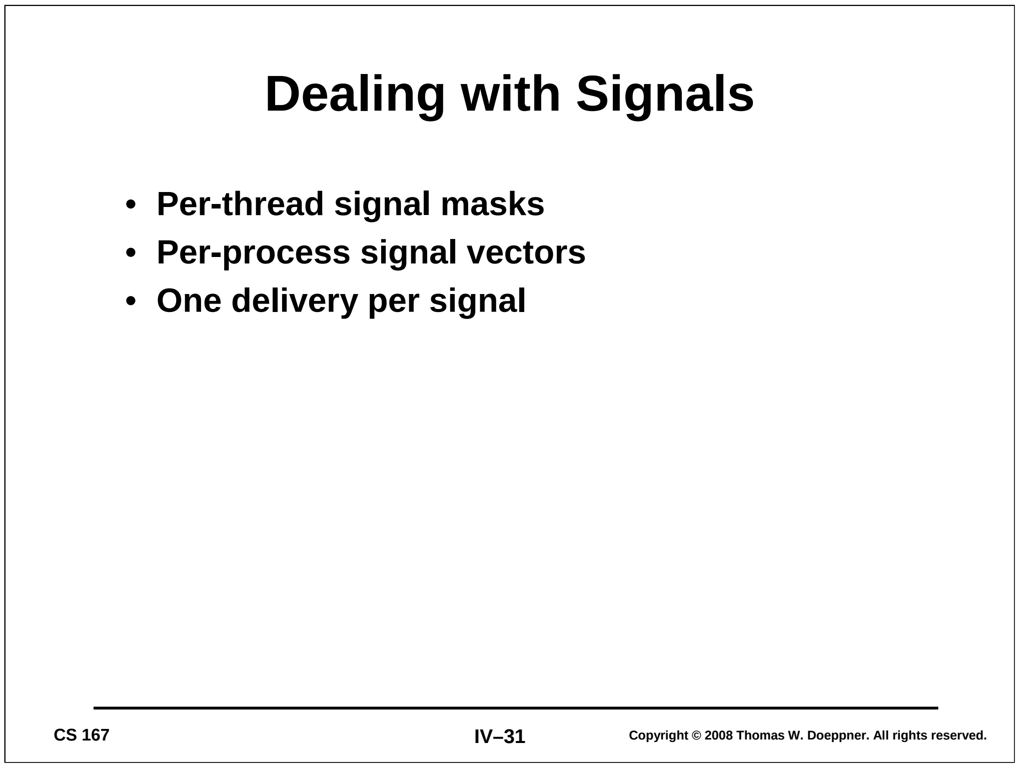
- – who gets them?
  – who needs them?
- – how do you respond to them?

CS 167 IV–30 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

Asynchronous signals were designed (like everything else) with single-threaded processes in mind. A signal is delivered to the process; if the signal is *caught*, the process stops whatever it is doing, deals with the signal, and then resumes normal processing. But what happens when a signal is delivered to a multithreaded process? Which thread or threads deal with it?

Asynchronous signals, by their very nature, are handled asynchronously. But one of the themes of multithreaded programming is that threads are a cure for asynchrony. Thus we should be able to use threads as a means of getting away from the “drop whatever you are doing and deal with me” approach to asynchronous signals.

Synchronous signals often are an indication that something has gone wrong: there really is no point continuing execution in this part of the program because you have already blown it. Traditional Unix approaches for dealing with this bad news are not terribly elegant (some, but not all, POSIX-threads implementations provide a more elegant means for dealing with such situations by means of an exception-handling package, similar to what’s available to C++ programmers).



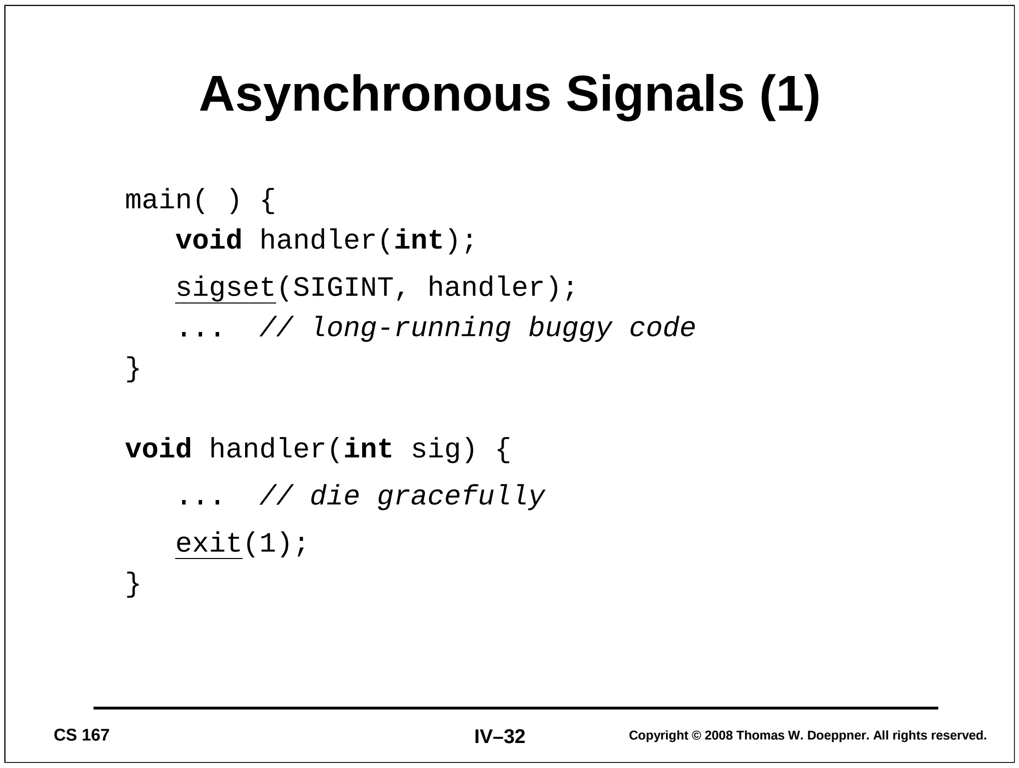
Dealing with Signals

- **Per-thread signal masks**
- **Per-process signal vectors**
- **One delivery per signal**

The standard Unix model has a process-wide signal mask and a vector indicating what is to be done in response to each kind of signal. When a signal is delivered to a process, an indication is made that this signal is *pending*. If the signal is unmasked, then the vector is examined to determine the response: to suspend the process, to resume the process, to terminate the process, to ignore the signal entirely, or to invoke a signal handler.

A number of issues arise in translating this model into a multithreaded-process model. First of all, if we invoke a signal handler, which thread or threads should execute the handler? What seems to be closest to the spirit of the original signal semantics is that exactly one thread should execute the handler. Which one? The consensus is that it really does not matter, just as long as exactly one thread executes the signal handler. But what about the signal mask? Since one sets masks depending on a thread's local behavior, it makes sense for each thread to have its own private signal mask. Thus a signal is delivered to any one thread that has the signal unmasked (if more than one thread has the signal unmasked, a thread is chosen randomly to handle the signal). If all threads have the signal masked, then the signal remains pending until some thread unmasks it.

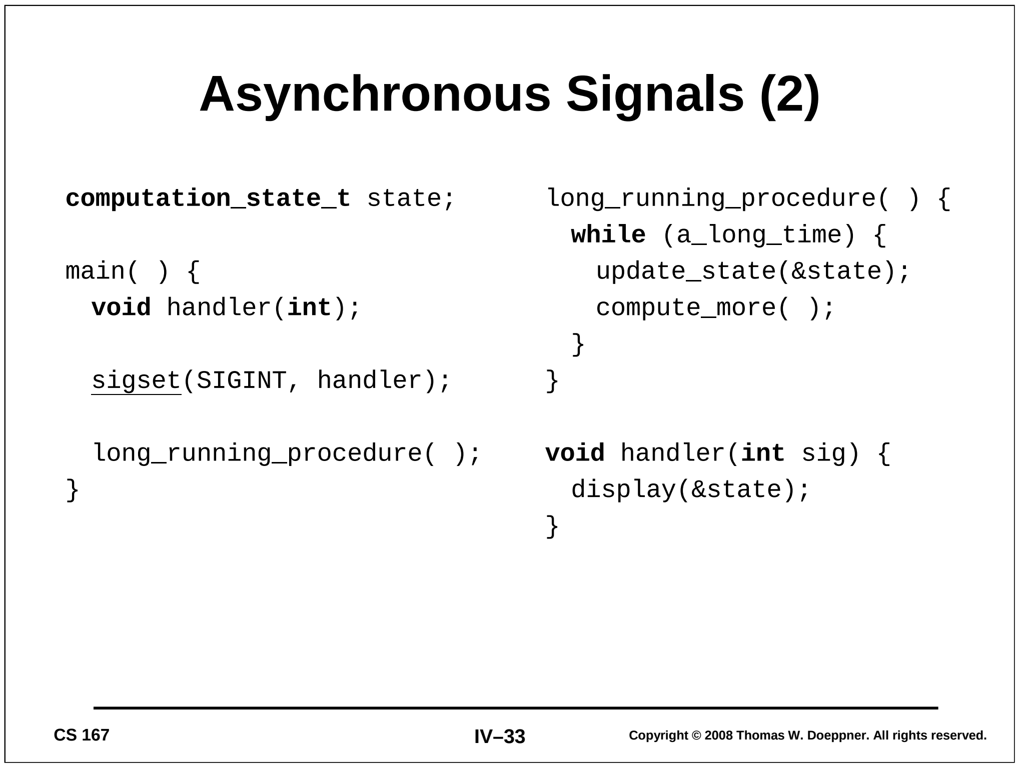
A related issue is the vector indicating the response to each signal. Should there be one such vector per thread? If so, what if one thread specifies process termination in response to a signal, while another thread supplies a handler? For reasons such as this, it was decided that, even for multithreaded processes, there would continue to be a single, process-wide signal-disposition vector.



Asynchronous Signals (1)

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

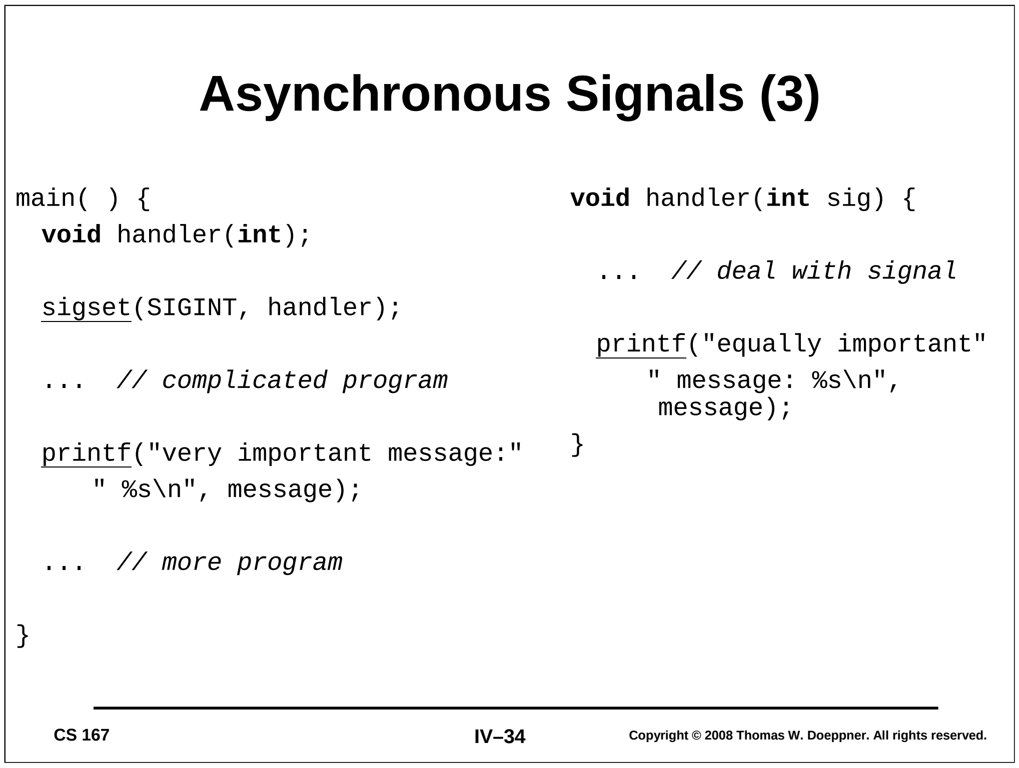


Asynchronous Signals (2)

```
computation_state_t state;

main( ) {
  void handler(int);

  sigset(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.



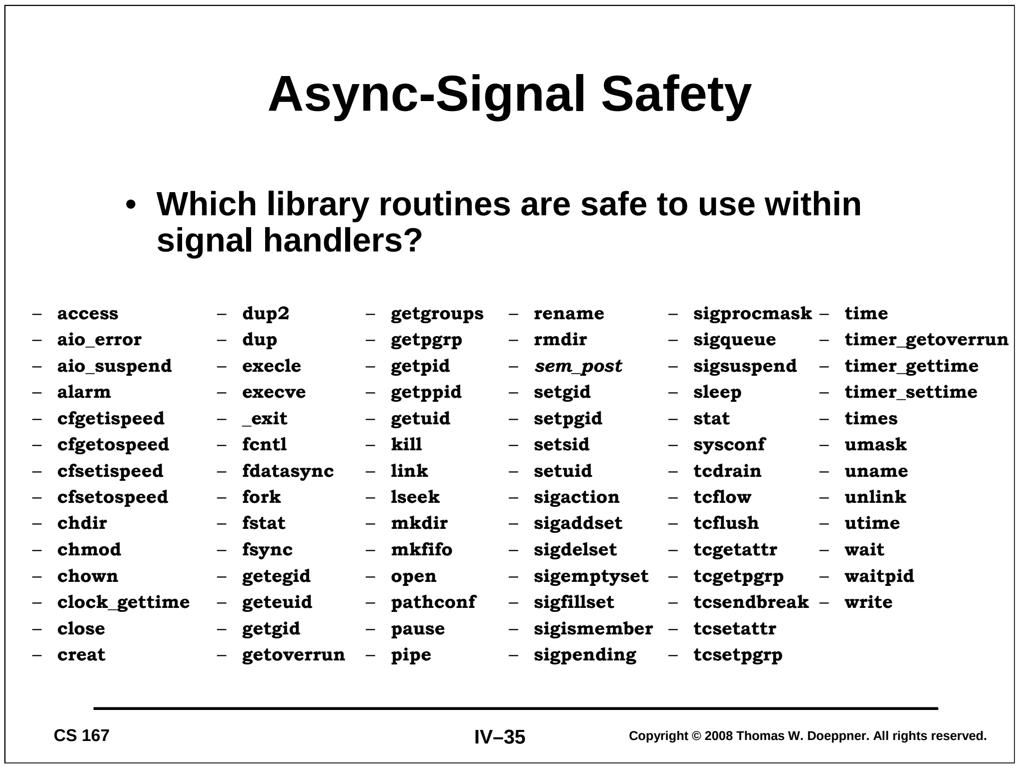
Asynchronous Signals (3)

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

  sigset(SIGINT, handler);

  ...  // complicated program

  printf("very important message:"
      " %s\n", message);

  ...  // more program

}
```

```
void handler(int sig) {

  ...  // deal with signal

  printf("equally important"
      " message: %s\n",
       message);
}
```

In this example, the signal handler calls *printf*. This is usually not a problem, except when the mainline code is interrupted while in the middle of *printf*—the *printf* called from the signal handler might destructively interfere with the *printf* called from the mainline code.

Similarly, it's dangerous (fatally so) to call *malloc* and *free* from within signal handlers: the handler might have interrupted another call to *malloc* or *free* and the result could be a corrupted heap.



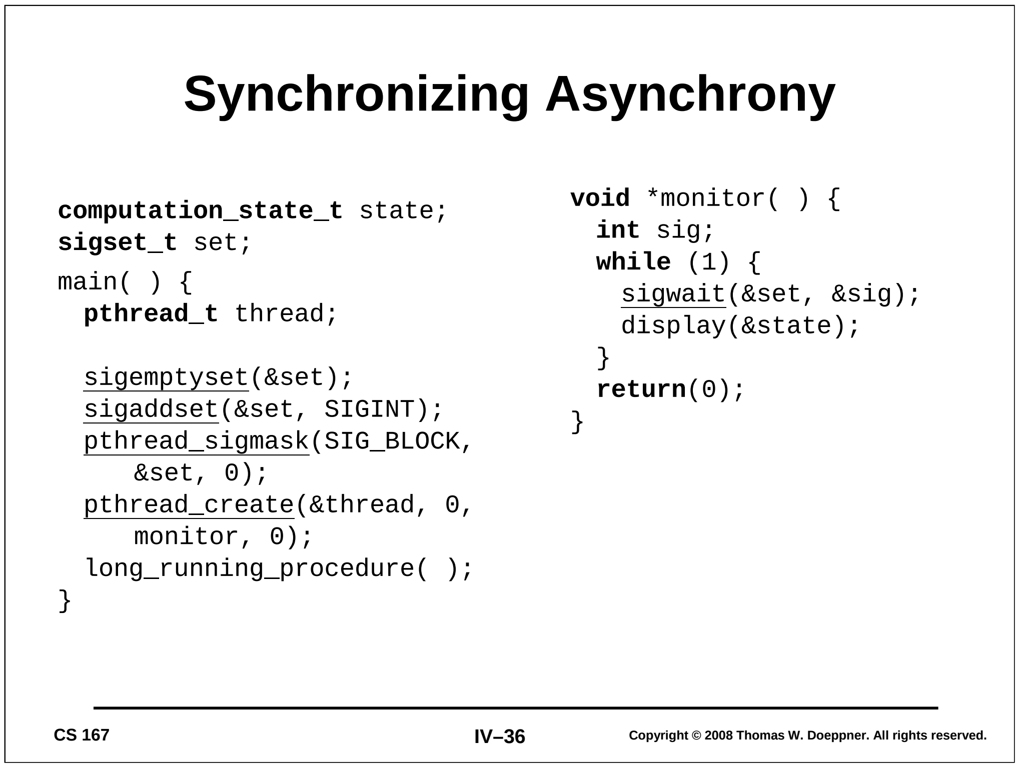
Async-Signal Safety

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

– **access**	– **dup2**	– **getgroups**	– **rename**	– **sigprocmask**	– **time**
– **aio_error**	– **dup**	– **getpgrp**	– **rmdir**	– **sigqueue**	– **timer_getoverrun**
– **aio_suspend**	– **execle**	– **getpid**	– ***sem_post***	– **sigsuspend**	– **timer_gettime**
– **alarm**	– **execve**	– **getppid**	– **setgid**	– **sleep**	– **timer_settime**
– **cfgetispeed**	– **_exit**	– **getuid**	– **setpgid**	– **stat**	– **times**
– **cfgetospeed**	– **fcntl**	– **kill**	– **setsid**	– **sysconf**	– **umask**
– **cfsetispeed**	– **fdatasync**	– **link**	– **setuid**	– **tcdrain**	– **uname**
– **cfsetospeed**	– **fork**	– **lseek**	– **sigaction**	– **tcflow**	– **unlink**
– **chdir**	– **fstat**	– **mkdir**	– **sigaddset**	– **tcflush**	– **utime**
– **chmod**	– **fsync**	– **mkfifo**	– **sigdelset**	– **tcgetattr**	– **wait**
– **chown**	– **getegid**	– **open**	– **sigemptyset**	– **tcgetpgrp**	– **waitpid**
– **clock_gettime**	– **geteuid**	– **pathconf**	– **sigfillset**	– **tcsendbreak**	– **write**
– **close**	– **getgid**	– **pause**	– **sigismember**	– **tcsetattr**	
– **creat**	– **getoverrun**	– **pipe**	– **sigpending**	– **tcsetpgrp**	

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 defines a number of these. Of the new routines introduced with multithreaded programming, only one is async-signal-safe—*sem_post*.

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, almost everything is async-signal-safe in Solaris. (Quick exercise: how might this be done?)



Synchronizing Asynchrony

```
computation_state_t state;
sigset_t set;
main( ) {
  pthread_t thread;

  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  pthread_sigmask(SIG_BLOCK,
     &set, 0);
  pthread_create(&thread, 0,
     monitor, 0);
  long_running_procedure( );
}
```

```
void *monitor( ) {
  int sig;
  while (1) {
    sigwait(&set, &sig);
    display(&state);
  }
  return(0);
}
```

In this slide we go back to the earlier problem and use a different technique for dealing with the signal. Rather than have the thread performing the long-running computation be interrupted by the signal, we dedicate a thread to dealing with the signal. We make use of a new signal-handling routine, *sigwait*. This routine puts its caller to sleep until one of the signals specified in its argument occurs, at which point the call returns. As is done here, *sigwait* is normally called with the signals of interest masked off; *sigwait* responds to signals even if they are masked. (Note also that a new thread inherits the signal mask of its creator.) (The system call *sigprocmask* is identical in effect to *pthread_sigmask* and could be used here instead.)

Among the advantages of this approach is that there are no concerns about async-signal safety since a signal handler is never invoked. The signal-handling thread waits for signals synchronously—it is not interrupted. Thus it is safe for it to use even mutexes, condition variables, and semaphores from inside of the display routine. Another advantage is that, if this program is run on a multiprocessor, the “signal handling” can run in parallel with the mainline code, which could not happen with the previous approach.