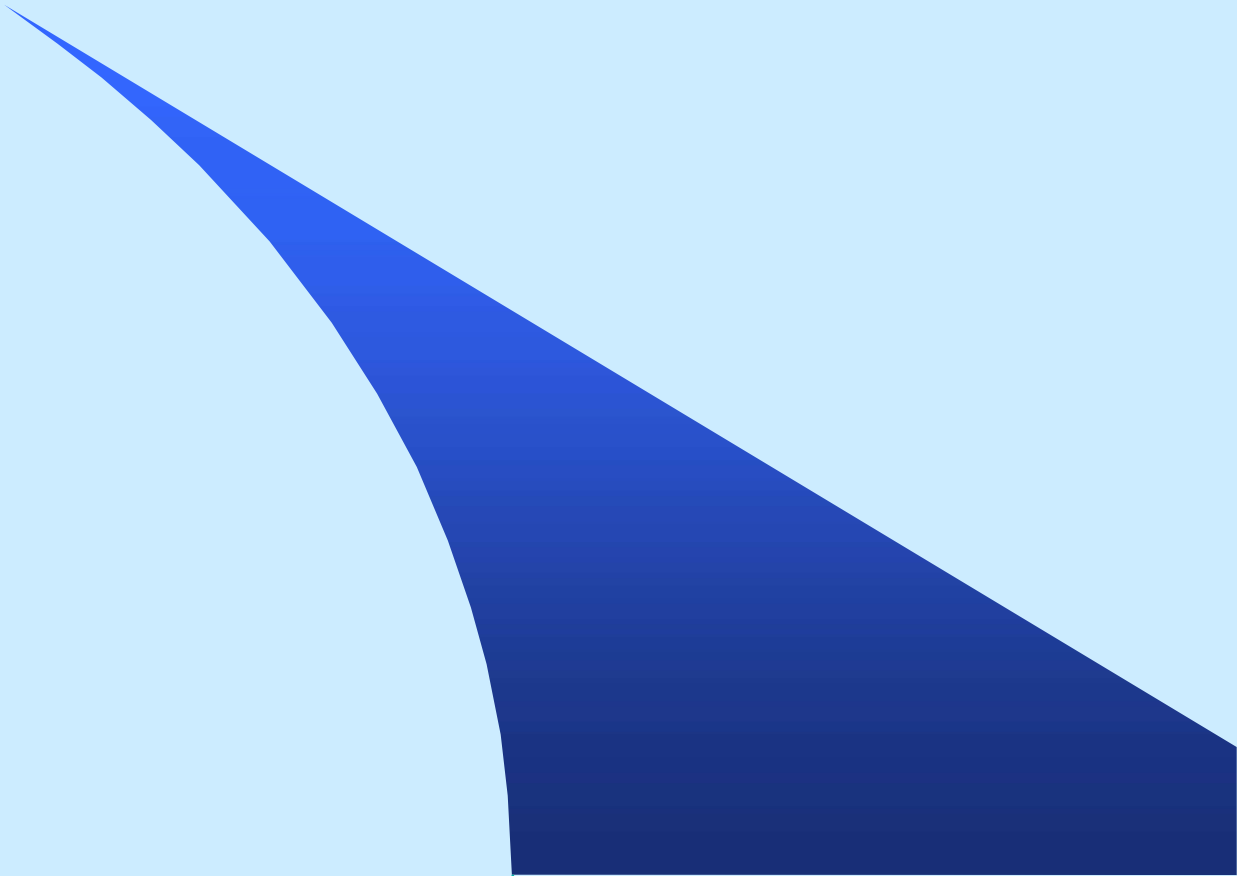
UNIX System Programming

Signals



Overview

1. Definition
2. Signal Types
3. Generating a Signal
4. Responding to a Signal
5. Common Uses of Signals
6. Timeout on a `read()`
7. POSIX Signal Functions
8. Interrupted System Calls
9. System Calls inside Handlers
10. More Information

1. Definition

- A signal is an *asynchronous* event which is delivered to a process.
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types `ctrl-C`, or the network hangs
- Sent from kernel (e.g. detects divide by zero (SIGFPE) or could be at the request of another process to send to another)

Signals

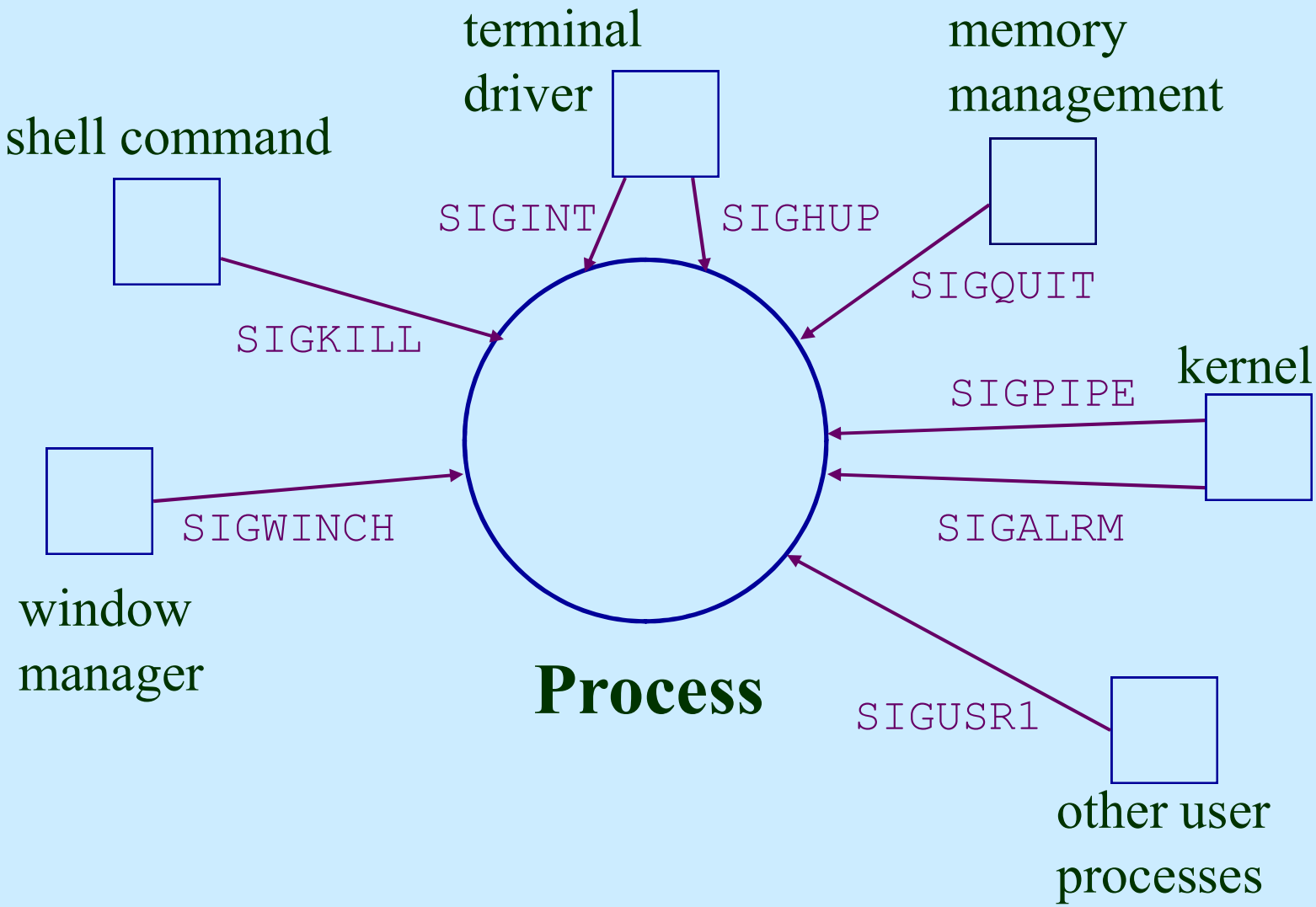
- Only information that a signal carries is its unique ID and that it arrived
- Each signal is mapped to a specific bit in the process signal status.
- Since this is a single bit, only one signal of a specific type can be “seen” by the targeted process. i.e. you can’t stack them !

2. Signal Types (31 in POSIX)

Name	*Description*	*Default Action*
`SIGINT`	Interrupt character typed	terminate process
`SIGQUIT`	Quit character typed (`^\`)	create core image
`SIGKILL`	`kill -9`	terminate process
`SIGSEGV`	Invalid memory reference	create core image
`SIGPIPE`	Write on pipe but no reader	terminate process
`SIGALRM`	`alarm()` clock 'rings'	terminate process
`SIGUSR1`	user-defined signal type	terminate process
`SIGUSR2`	user-defined signal type	terminate process

- See `man 7 signal`

Signal Sources



3. Generating a Signal

- You have probably already sent signals to applications without knowing it.
- have you pressed CTRL-C (or ^C) to stop a program from running?
- When you shut down your computer, the O/S sends a shutdown signal to your applications to make sure files are closed properly and there is no data loss (assuming your application can capture the signal).

Generating a Signal

- Use the UNIX command:

```
$ kill -KILL 4481
```

  - send a `SIGKILL` signal to pid 4481
  - check
    - **`ps -l`**
  - to make sure process died

- `kill` is not a good name; `send_signal` might be better.

kill()

- Send a signal to a process (or group of processes).

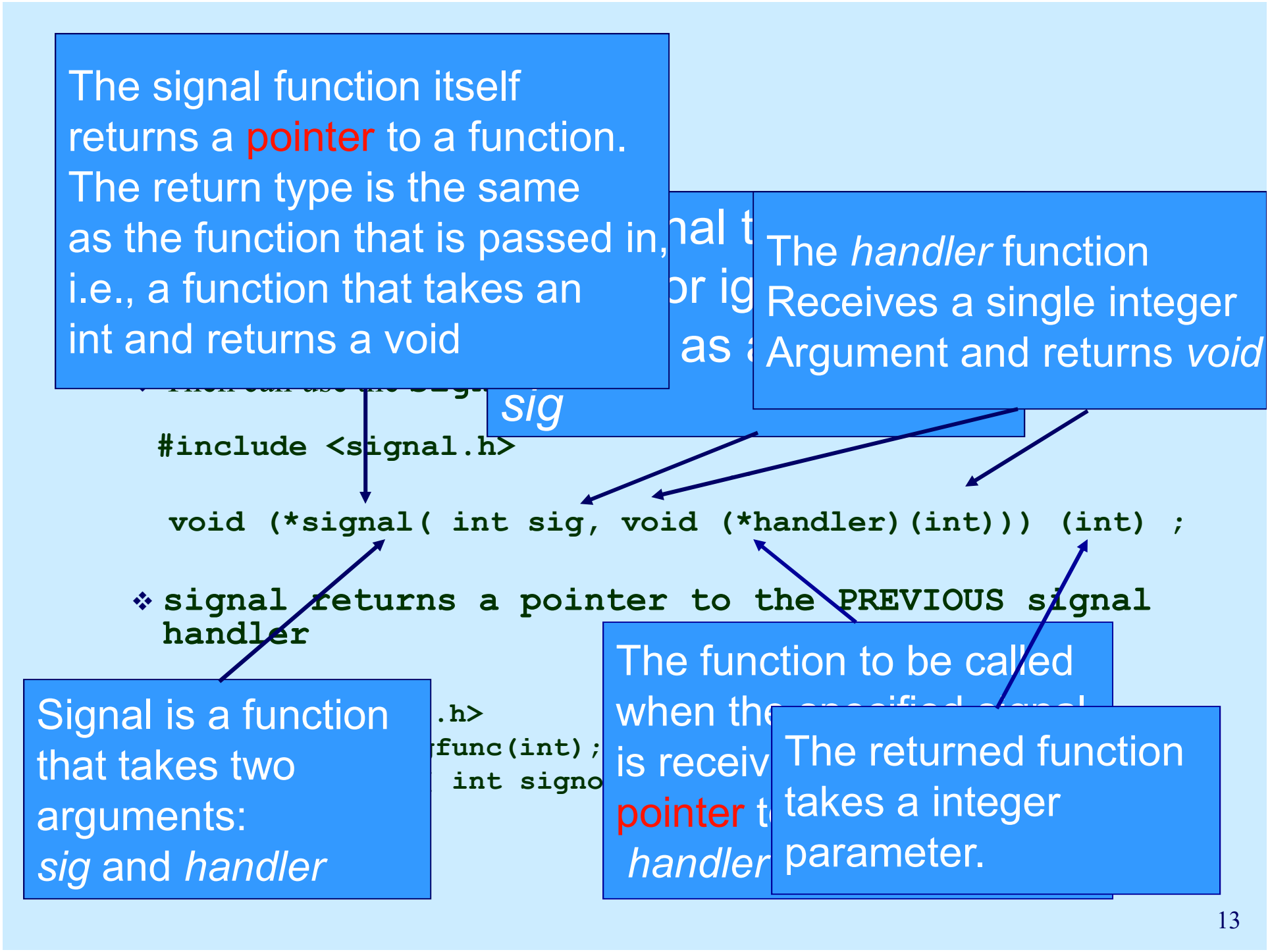
- ```
  #include <signal.h>

  int kill( pid_t pid, int signo );
  ```

- Return 0 if ok, -1 on error.

NOTE: the PID must be something you have the "right" to talk to.
  ```

Some pid Values

pid	*Meaning*
> 0	send signal to process `pid`
== 0	send signal to all processes whose process group ID equals the sender's pgid. e.g. parent kills all children

4. Responding to a Signal

- A process can:

  1) ignore/discard the signal (not possible with `SIGKILL` or `SIGSTOP`)

  2) execute a signal handler function, and then possibly resume execution or terminate

  3) carry out the default action for that signal

- The choice is called the process' *signal disposition*

signal(): library call

```
#include <signal.h>

   void (*signal( int signo, void (*func)(int) ))(int);

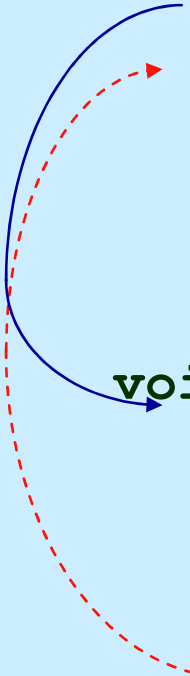
   typedef void Sigfunc(int);  /* my defn */

   Sigfunc *signal( int signo, Sigfunc *handler );
```

- signal returns **a pointer** to a function that returns an int (i.e. it returns a pointer to Sigfunc)
- Specify a signal handler function to deal with a signal type.
- Returns *previous* signal disposition if ok, `SIG_ERR` on error.


The signal function itself returns a pointer to a function. The return type is the same as the function that is passed in, i.e., a function that takes an int and returns a void
The *handler* function Receives a single integer Argument and returns *void*
sig
#include <signal.h>
void (*signal(int sig, void (*handler)(int))) (int) ;
signal returns a pointer to the PREVIOUS signal handler
Signal is a function that takes two arguments: *sig* and *handler*
The function to be called when the
is receiv
pointer t
handler
The returned function takes a integer parameter.

How to Program with Signals

```
int main()
    {
    signal( SIGQUIT, foo );
      :

    /* do usual things until SIGQUIT */
    return 0;
    }



void foo( int signo )
    {
      :          /* deal with SIGQUIT signal */


    return;      /* return to program */
    }
```



sig_examp.c – Signal Handler

```
void sig_usr( int signo )
/* argument is signal number */
{
  if( signo == SIGINT )
     printf("Received ^C SIGINT\n");
  else if( signo == SIGQUIT )
      printf("Received ^\ SIGQUIT\n");
  else {
      printf("Error: received signal %d\n",signo);
      exit(1); }
  return;
}
```

sig_examp.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sig_usr( int signo );    /* handles two signals */

int main()
   {
   int i = 0, done = 0;
   if( signal( SIGINT,sig_usr ) == SIG_ERR )
       printf( "Cannot catch ^C SIGINT\n" );

   if( signal( SIGQUIT,sig_usr ) == SIG_ERR )
      printf("Cannot catch ^\ SIGQUIT\n");

   while(!done)
       {
       printf( "%2d\n", i );
       pause();
       /* pause until signal handler
                * has processed signal */
       i++;
       }
    return 0;
   }
```

Usage

```
$ ./sig_example
 0
^CReceived SIGINT
 1
^CReceived SIGINT
 2
^\Received SIGQUIT
 3
^Z
[1]+  Stopped     ./sig_example
$ kill -9 %1
[1]+  Stopped ./sig_example
```

Special Sigfunc * Values

- *Value* — *Meaning*

`SIG_IGN`	Ignore / discard the signal.
`SIG_DFL`	Use default action to handle signal.
`SIG_ERR`	Returned by `signal()` as an error.

Multiple Signals

- If many signals of the *same* type are waiting to be handled (e.g. two `SIGQUIT`s), then most UNIXs will only deliver one of them.
  - **the others are thrown away - i.e. pending signals are not queued**
  - **for each signal type, just have a single bit indicating whether or not the signal has occured**

- If many signals of *different* types are waiting to be handled (e.g. a `SIGQUIT`, `SIGSEGV`, `SIGUSR1`), they are not delivered in any fixed order.

pause()

- Suspend the calling process until a signal is caught.
- ```
  #include <unistd.h>
  int pause(void);
  ```
- Returns -1 with `errno` assigned `EINTR`.
- `pause()` only returns after a signal handler has returned.
  ```

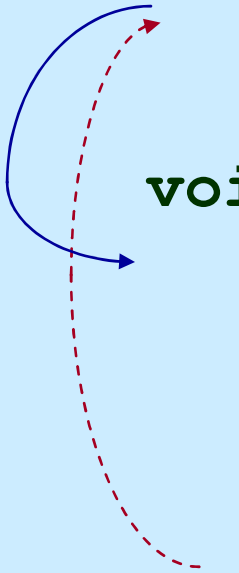
The Reset Problem

- In many UNIXs, the signal disposition in a process is reset to its default action immediately after the signal has been delivered.
  - This is a concern with Solaris but not with Linux (loki)

- Must call `signal()` again to reinstall the signal handler function.

Reset Problem Example

```
int main()
    {
    signal(SIGQUIT, foo);
     :
    /* do usual things until SIGQUIT */
    }

void foo(int signo)
    {
    signal(SIGQUIT, foo); /* reinstall */
          :
    return;
    }
```



Reset Problem

```
      :
void ouch( int sig )
        {
        printf( "OUCH! - I got signal %d\n", sig );
        signal(SIGQUIT, ouch);
        }
int main()
        {
        int done = 0;
        signal( SIGQUIT, ouch );
        while(!done)
                {
                printf("Hello World!\n");
                sleep(1);
                }
        }
```

To keep catching the signal with this function, must call the *signal* system call again (for some versions of Unix)

Problem: from the time that the interrupt function starts to just before the signal handler is re-established the signal will not be handled.

If another SIGQUIT signal is received during this time, default behavior will be done, i.e., program will terminate.

Re-installation may be too slow!

- There is a (very) small time period in `ouch()` when a new `SIGQUIT` signal will cause the default action to be carried out -- process termination.

- With `signal()` there is no answer to this problem.
  - POSIX signal functions solve it (and some other later UNIXs)

Another Example

```
#iinclude <stdio.h>
#include <signal.h>
#include <unistd.h>
int beeps = 0;                          // GLOBAL VARIABLE !!!
static void handler( int signo )
    {
    printf( "BEEP\n" );
    if(++beeps < 5 )
            {
            signal( SIGALRM, handler );  /* may not be needed */
            alarm(1); /* send a SIGALRM in 1 sec */
            }
    else
            {
            printf("BOOM!\n");
            exit(0);
            }
    return;
    }
```

```
int main( void )
    {
    int i = 0, done = 0;
    if( signal( SIGALRM, handler ) == SIG_ERR )
        printf("Cannot catch SIGALRM\n" );
    alarm(1); /* send a SIGSALRM in 1 sec */
    while( !done )
        {
        printf( "%d: ", i );
        pause(); /* wait for any signal  */
        i++;
        }
    return 0;
    }
```

5. Common Uses of Signals

5.1 Ignore a Signal

5.2 Clean up and Terminate

5.3 Report Status

5.4 Turn Debugging on/off

5.5 Restore Previous Handler

5.1 Ignore a Signal

```
      :
int main()
{
   signal(SIGINT, SIG_IGN);
   signal(SIGQUIT, SIG_IGN);
      :
   /* do work without interruptions */
}
```

- Cannot ignore/handle `SIGKILL` or `SIGSTOP`
- Should check for `SIG_ERR`

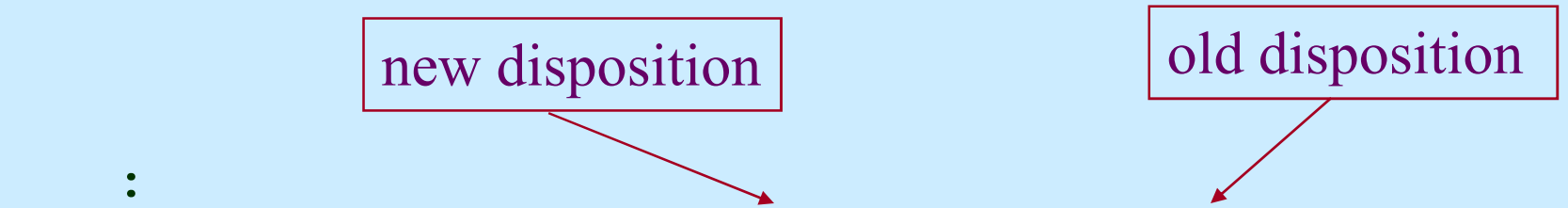
5.2 Clean up and Terminate

```
    :
/* global variables */
int my_children_pids;
    :
void clean_up(int signo);

int main()
{
  signal(SIGQUIT, clean_up);
    :
}


void clean_up(int signo)
{
  unlink("/tmp/work-file");
  kill(my_children_pids, SIGTERM);
  wait((int *)0);
  printf("Program terminated\n");
  exit(1);
}
```

Problems

- If a program is run in the background then the interrupt and quit signals (`SIGINT`, `SIGQUIT`) are automatically ignored.
- Your code should not override these changes:
  - check if the signal dispositions are `SIG_IGN`

Checking the Disposition




```
    :
if( signal(SIGINT, SIG_IGN ) != SIG_IGN )
    signal(SIGINT, clean_up);

if( signal(SIGQUIT, SIG_IGN ) != SIG_IGN )
    signal(SIGQUIT, clean_up);
    :
```

- *Note*: cannot check the signal disposition without changing it (sigaction that we will look at later is different)

5.3 Report Status

```
   :
  void print_status(int signo);
  int count;   /* global */

  int main()
  { signal(SIGUSR1, print_status);
      :
    for( count=0; count < BIG_NUM; count++ )
      {
      /* read block from tape */
      /* write block to disk */
      }
    ...
  }

void print_status(int signo)
  {
    signal(SIGUSR1, print_status); // depends
    printf("%d blocks copied\n", count);
    return;
  }
```

- **Must use global variables for status information**

5.4 Turn Debugging on/off

```
  :
void toggle_debug(int signo);
int debug = 0;  /* initialize here */

int main()
{
  signal(SIGUSR2, toggle_debug);
  /* do work */
  if (debug == 1)
     printf("...");
  ...
}


void toggle_debug(int signo)
{
  signal(SIGUSR2, toggle_debug); // depends
  debug = ((debug == 1) ? 0 : 1);
  return;
}
```

5.5 Restore Previous Handler

```
    :
Sigfunc *old_hand;

/* set action for SIGTERM;
   save old handler */
old_hand = signal(SIGTERM, foobar);

/* do work */

/* restore old handler */
signal(SIGTERM, old_hand);
    :
```

6. Implementing a read() Timeout

- Put an upper limit on an operation that might block forever
  - e.g. `read()`

- 6.1 `alarm()`
  6.2 Bad `read()` Timeout
  6.3 `setjmp()` and `longjmp()`
  6.4 Better `read()` Timeout

6.1 alarm()

- Set an alarm timer that will ‘ring’ after a specified number of seconds
  - a `SIGALRM` signal is generated

- ```
  #include <unistd.h>
  long alarm(long secs);
  ```

- Returns 0 or number of seconds until previously set alarm would have ‘rung’.
  ```

Some Tricky Aspects

- A process can have at most one alarm timer running at once.
- If `alarm()` is called when there is an existing alarm set then it returns the number of seconds remaining for the old alarm, and sets the timer to the new alarm value.
  - What do we do with the "old alarm value"?
- An `alarm(0)` call causes the previous alarm to be cancelled.

6.2 Bad read() Timeout

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

#define MAXLINE  512

void sig_alrm( int signo );

int main()
    {
    int n;
    char line[MAXLINE];
    if( signal(SIGALRM, sig_alrm) == SIG_ERR )
          {
          printf("signal(SIGALRM) error\n");
          exit(1);
          }
        . . .
```

```
        alarm(10);
        n = read( 0, line, MAXLINE );
        alarm(0);

        if( n < 0 )  /* read error */
              fprintf( stderr, "\nread error\n" );
        else
              write( 1, line, n );
        return 0;
        }

void sig_alrm(int signo)
   /* do nothing, just handle signal */
   {
   return;
   }
```

Problems

- The code assumes that the `read()` call terminates with an error after being interrupted

- Race Conditon: The kernel may take longer than 10 seconds to start the `read()` after the `alarm()` call.
  - the alarm may 'ring' before the `read()` starts
  - then the `read()` is not being timed; may block forever
  - Two ways two solve this:
    - **`setjmp`**
    - **`sigprocmask`** and **`sigsuspend`**

6.3 setjmp() and longjmp()

- In C we cannot use `goto` to jump to a label in another function
  - use `setjmp()` and `longjmp()` for those 'long jumps'

- Only uses which are good style:
  - error handling which requires a deeply nested function to recover to a higher level (e.g. back to `main()`)
  - coding timeouts with signals

Nonlocal Jumps: setjmp() & longjmp()

- **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - **» controlled way to break the procedure call/return discipline**
  - **» Useful for error recovery and signal recovery**
- **setjmp(jmp_buf j)**
  - **» called before longjmp()**
  - **» identified return site for subsequent longjmp()**
  - » On first call, returns a zero
  - **» subsequent calls return the value from the int 2nd parameter of longjump()**
  - **» Called once, returns one or more times**
- **Implementation:**
  - **» remember where you are by storing the current register context, stack pointer and PC value in jmp_buf**
  - **» returns 0 the first time it is called.**

Prototypes

- **#include <setjmp.h>**
  int setjmp(jmp_buf env);
  void longjmp(jmp_buf env, int val);

- **Returns 0 if called directly, non-zero if returning from a call to longjmp().**
- **In the setjmp() call, env is initialized to information about the current state of the stack.**
- **The longjmp() call causes the stack to be reset to its env value.**
- **Execution restarts after the setjmp() call, but this time setjmp() returns val.**

Example

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
void handler(int sig);
jmp_buf buf;

int main()
        {
        int done = 0;
        signal(SIGQUIT, handler);
        if( !setjmp(buf) )
                printf("starting\n");
        else
                printf("restarting\n");
        while( !done )
                {
                sleep(1);
                printf("processing...\n");
                }
        }
```

```
void handler(int sig)
        {
         // may not be needed
        signal(SIGQUIT, handler);
        longjmp( buf, 1 );
        }
```

Note: setjmp() returns a zero on first call (FALSE). It returns the value of the 2nd param of longjump() on subsequent invocations.

Output

```
$ ./jumper
    starting
    processing...
    ^\restarting
    processing...
    ^\restarting
    processing...
     processing...
     ^z
    Stopped (user)
$ kill -9 %1
[1]  Killed      ./jumper
```

Another Example: sleep1()

```
#include <signal.h>
#include <unistd.h>

void sig_alrm( int signo )
     {
     return; /* return to wake up pause */
     }
unsigned int sleep1( unsigned int nsecs )
     {
     if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
            return (nsecs);
     alarm( nsecs );          /* starts timer */
     pause();                 /* next caught signal wakes */
     return( alarm( 0 ) );    /* turn off timer, return unslept
                               * time */
     }
```

Comments on sleep(1)

- **Alarm erases "old" set alarms set by process**
  - **» Look at return value from the previous alarm() call**
    - **If less than new alarm() - wait until old alarm() expires**
    - **If larger than new alarm()- reset old alarm() with remaining seconds when done with new alarm()**
- **Race condition**
  - **» between first call to alarm and the call to pause**
  - **=> never get out of pause (fix via setjmp/longjmp or sigprocmask/sigsuspend)**

sleep2(): Fixes race condition

```
static void jmp_buf env_alrm;
void sig_alrm( int signo )
     {
     longjmp( env_alrm, 1 );
     }
unsigned int sleep2( unsigned int nsecs )
     {
     if( signal( SIGALRM, sig_alrm ) == SIG_ERR )
            return (nsecs);
     if( !setjmp( env_alrm))   /* returns 0 (FALSE) on 1st call */
            {
            alarm( nsecs );      /* starts timer */
            pause();             /* next caught signal wakes */
            }
     return( alarm( 0 ) );
     }
```

Sleep1 and Sleep2

- Sleep2 fixes race condition. Even if the pause is never executed.
  - **» A SIGALRM causes sleep2() to return**
  - **» Avoids entering pause() via longjmp()**

- There is one more problem
  - **» SIGALRM could interrupt some other signal handler and subsequently abort it by executing the longjmp**

6.4 Better read() Timeout

```
#include <stdio.h>
#include <unistd.h>
#include <setjmp.h>
#include <signal.h>

#define MAXLINE  512
void sig_alrm( int signo );
jmp_buf env_alrm;

int main()
{ int n;
  char line[MAXLINE];

 if( signal(SIGALRM, sig_alrm) == SIG_ERR)
     {
     printf("signal(SIGALRM) error\n");
     exit(1);
     }
```

```
    if( setjmp(env_alrm) != 0 )
       {
       printf("\nread() too slow\n");
       exit(2);
       }

    alarm(10);
    n = read(0, line, MAXLINE);
    alarm(0);

    if( n < 0 )  /* read error */
      printf("\nread error\n" );
    else
      write( 1, line, n );
    return 0;
  }


void sig_alrm(int signo)
  /* interrupt the read() and jump to setjmp() call with value 1 */
  {
    longjmp(env_alrm, 1);
  }
```

Caveat: Non-local Jumps

From the UNIX **`man`** pages:

WARNINGS

If longjmp() or siglongjmp() are called even though env was never primed by a call to setjmp() or sigsetjmp(), or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

A Problem Remains!

- If the program has several signal handlers then:
  - execution might be inside one when an alarm ‘rings’
  - the `longjmp()` call will jump to the `setjmp()` location, and abort the other signal handler -- might lose / corrupt data

Stop here Winter 2023

7. POSIX Signal Functions

- The POSIX signal functions can control signals in more ways:
  - can *block signals* for a while, and deliver them later (good for coding critical sections)
  - can *switch off the resetting* of the signal disposition when a handler is called (no reset problem)

- The POSIX signal system, uses signal sets, to deal with pending signals that might otherwise be missed while a signal is being processed

7.1 Signal Sets

- The signal set stores collections of signal types.
- Sets are used by signal functions to define which signal types are to be processed.
- POSIX contains several functions for creating, changing and examining signal sets.

Prototypes

- ```
  #include <signal.h>

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

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

  int sigismember( const sigset_t *set,int signo );
  ```
  ```

Cont'd

- ***sigemptyset*** **- initializes signal set pointed by set so that all signals are excluded**
- ***sigfillset*** **- all signals are included**
  - `One of sigemptyset or sigfillset must be called to initialize a signal set`
- ***sigaddset*** **- add a single signal (signo) to set**
- ***sigdelset*** **- remove signo from set**
- ***sigismember*** **- is signo from a member of set**

7.2 sigprocmask()

- A process uses a signal set to create a mask which defines the signals it is blocking from delivery.
  - good for critical sections where you want to block certain signals.

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

- ***how*** **- indicates how mask is modified**

how Meanings

- *Value* *Meaning*

`SIG_BLOCK`	`set` signals are added to mask
`SIG_UNBLOCK`	`set` signals are removed from mask
`SIG_SETMASK`	`set` becomes new mask

A Critical Code Region

```
sigset_t newmask, oldmask;

sigemptyset( &newmask );
sigaddset( &newmask, SIGQUIT );

/* block SIGQUIT; save old mask */
sigprocmask( SIG_BLOCK, &newmask, &oldmask );

/* critical region of code */

/* reset mask which unblocks SIGQUIT */
sigprocmask( SIG_SETMASK, &oldmask, NULL );
```

7.3 sigaction()

- **`#include <signal.h>`**

  `int sigaction(int signo, const struct sigaction *act,`
  `struct sigaction *oldact );`

- Supercedes (more powerful than) `signal()`
  - `sigaction()` can be used to code a non-resetting `signal()`
- ***signo* is the signal you want to perform an action on**
- ***act* is the action**
- ***oact* is the old action (can be set to NULL, if uninteresting)**
- **Cannot handle SIGSTOP and SIGKILL**

sigaction() Structure

```
struct sigaction
    {
    void       (*sa_handler)( int );
                  /* action to be taken or SIG_IGN, SIG_DFL */
    sigset_t   sa_mask; /* additional signals to be blocked */
    int        sa_flags; /* modifies action of the signal */
    }
```

- sa_flags –
  - SA_RESETHAND: reset handler to default upon return
  - SA_RESTART: if process is interruptible, then it is restarted

sigaction() Behavior

- A *signo* signal causes the *sa_handler* signal handler to be called.
- While *sa_handler* executes, the signals in *sa_mask* are blocked. Any additional *signo* signals are also blocked.
- *sa_handler* remains installed until it is changed by another `sigaction()` call. No reset problem.

Signal Raising

```
struct sigaction
        {
        void (*) (int) sa_handler
        sigset_t sa_mask
        int sa_flags
        }
```

```
#include <signal.h>
#include <stdio.h>
void ouch( int signo )
  { printf( "OUCH! signo = %d\n", signo ); }

int main()
  {
  int done = 0;
  struct sigaction act;
  act.sa_handler = ouch;
  sigemptyset( &act.sa_mask );
  act.sa_flags = 0;
  sigaction( SIGQUIT, &act, 0 );

                                llo World!\n");

  }
```

Set the signal handler to be the function `ouch`

We can manipulate sets of signals to be blocked while *signo* is processed.

No flags are needed here. Possible flags include: SA_RESETHAND

This call sets the signal handler for the SIGQUIT (Ctrl-\) signal

Signal Raising

- This function will continually capture the Ctrl-\ (SIGQUIT) signal
- Default behavior is **not** restored after signal is caught.
- To terminate the program, must type Ctrl-c (SIGINT), or Ctrl-z (SIGSTOP) followed by kill %1

sigexPOS.c

```
/* include files as before */
#include <signal.h>
int main(void)
{
   /* struct to deal with action on signal set */
   static struct sigaction act;
   void catchquit(int); /* user signal handler */
   /* set up action to take on receipt of SIGQUIT */
   act.sa_handler = catchquit;
   /* create full set of signals */
   sigfillset(&act.sa_mask);
   /* before sigaction, SIGQUIT will terminate */
   /* now, SIGQUIT will cause catchquit to execute */
   sigaction( SIGQUIT, &act, NULL );
```

```
   /* remainder of main program */
   printf("sleep call #1\n");
   sleep(3);
  printf("sleep call #2\n");
  sleep(3);
  printf("sleep call #3\n");
  sleep(3);
  printf("sleep call #4\n");
  sleep(3);
  printf("Exiting \n");
  exit (0);
   }
/* simple signal handler */
   void catchquit(int signo)
      {
      printf("\nCATCHQUIT: signo=%d\n", signo);
      printf("\nCATCHQUIT: returning\n\n");
      }
```

```
$ ./sleeper
Sleep call #1
Sleep call #2
^\
CATCHQUIT: signo = 3
CATCHQUIT: returning
Sleep call #3
Sleep call #4
Exiting
$
```

Signals - Ignoring signals

- Other than SIGKILL and SIGSTOP, signals can be ignored:

- Instead of in the previous program:

```
  act.sa_handler = catchquit /* or whatever */
We use:
  act.sa_handler = SIG_IGN;
The ^\ key  will be ignored
```

Restoring previous action

- The third parameter to sigaction, oact, can be used:

```
/* save old action */
sigaction( SIGTERM, NULL, &oact );

/* set new action */
act.sa_handler = SIG_IGN;

sigaction( SIGTERM, &act, NULL );

/* restore old action */
sigaction( SIGTERM, &oact, NULL );
```

7.4 Other POSIX Functions

- `sigpending()` — examine blocked signals
- `sigsetjmp()`
  `siglongjmp()` — jump functions for use in signal handlers which handle masks correctly
- `sigsuspend()` — atomically reset mask and sleep

[sig]longjmp & [sig]setjmp

NOTES (longjmp, sigjmp)

POSIX does not specify whether longjmp will restore the signal context. If you want to save and restore signal masks, use siglongjmp.

NOTES (setjmp, sigjmp)

POSIX does not specify whether setjmp will save the signal context. (In SYSV it will not. In BSD4.3 it will, and there is a function _setjmp that will not.) If you want to save signal masks, use sigsetjmp.

Example

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

sigjmp_buf buf;

void handler(int sig)
        {
        siglongjmp(buf, 1);
        }

main()
{
  int done = 0;
  signal(SIGQUIT, handler);

  if( sigsetjmp(buf, 1) == 0 )
        printf("starting\n");
  else
        printf("restarting\n");
  …
```

```
  …
  while(!done)
        {
        sleep(5);
        printf("  waiting...\n");
        }
}
```

```
> ./longjump
starting
  waiting...
  waiting...          <-- Control-\
restarting
  waiting...
  waiting...
  waiting...
restarting            <-- Control-\
  waiting...
restarting            <-- Control-\
  waiting...
  waiting...
```

8. Interrupted System Calls

- When a system call (e.g. `read()`) is interrupted by a signal, a signal handler is called, returns, and then what?

- On many UNIXs, *slow* system function calls do not resume. Instead they return an error and `errno` is assigned `EINTR`.

Slow System Functions

- Slow system functions carry out I/O on things that can possibly block the caller forever:
  - pipes, terminal drivers, networks
  - some IPC functions
  - `pause()`, some uses of `ioctl()`

Non-slow System Functions

- Most system functions are non-slow, including ones that do *disk* I/O
  - e.g. `read()` of a disk file
  - `read()` is sometimes a slow function, sometimes not
- Some UNIXs restart non-slow system functions after the handler has finished.
- Some UNIXs only call the handler after the non-slow system function call has finished.

9. System Calls inside Handlers

- If a system function is called inside a signal handler then it may interact with an interrupted call to the same function in the main code.
  - e.g. `malloc()`
- This is not a problem if the function is *reentrant*
  - a process can contain multiple calls to these functions at the same time
  - e.g. `read()`, `write()`, `fork()`, many more

Non-reentrant Functions

- A functions may be non-reentrant (only one call to it at once) for a number of reasons:
  - it uses a static data structure
  - it manipulates the heap: `malloc()`, `free()`, etc.
  - it uses the standard I/O library
    - e,g, `scanf()`, `printf()`
    - the library uses global data structures in a non-reentrant way