Signals

Overview

- Definition
- Signal Types
- Generating a Signal
- Signal Disposition
- POSIX Signal Functions
- System Calls inside Handlers

Interrupt vs Exceptions vs signals

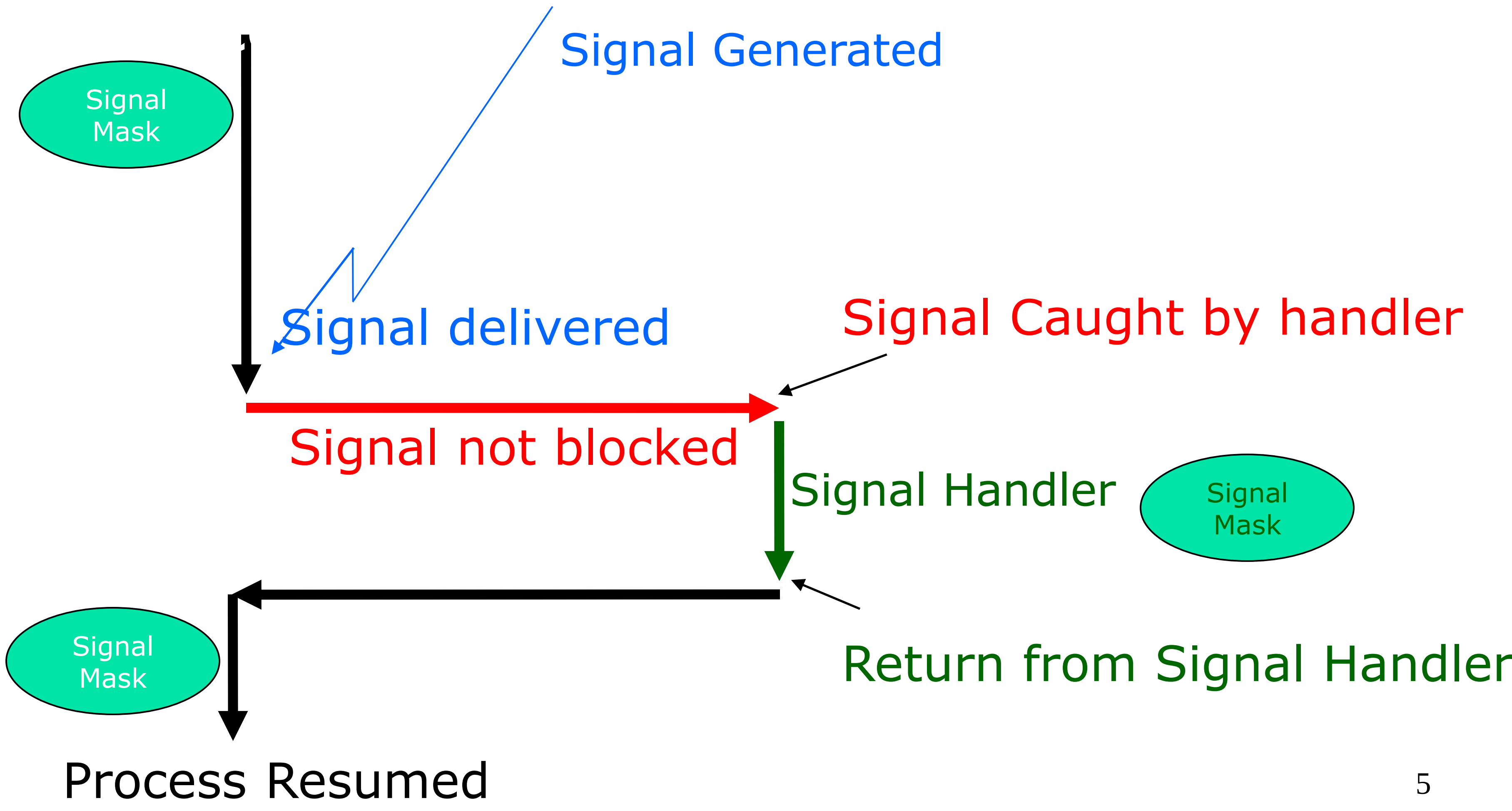
- Interrupts
  - Asynchronous
  - Generated by the hardware
- Exceptions
  - Synchronous
  - Generated by the processor
- Signals
  - Synchronous
  - Asynchronous

What is a Signal?

- 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 modem hangs

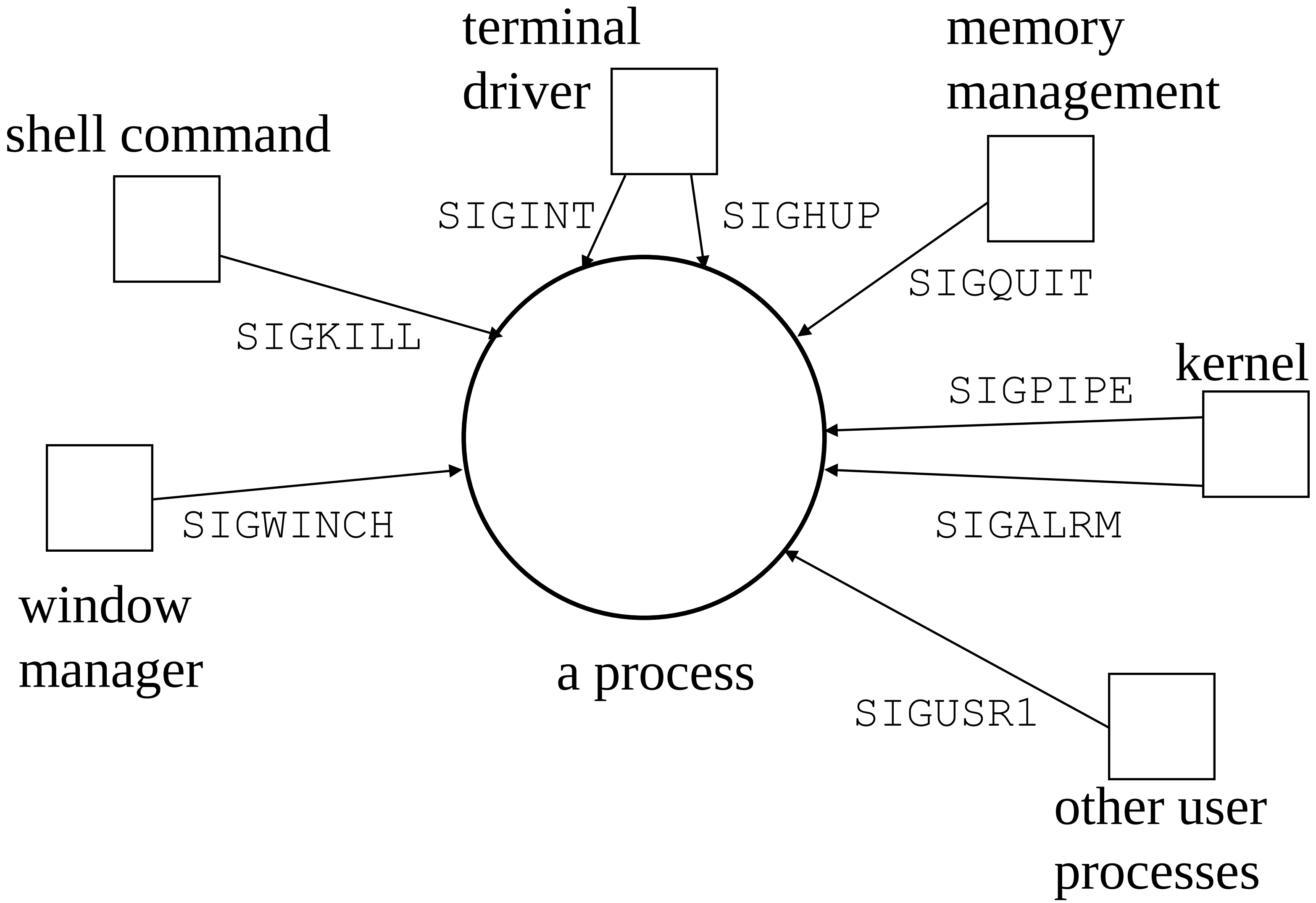
How Signals Work



Signal Handler

- A signal will suspend the execution of the program.
- A signal action must be registered before the signal's arrival.
- The signal handling procedure then invokes the registered function or action.
- The function that is called to handle a signal is known as a ***signal handler***

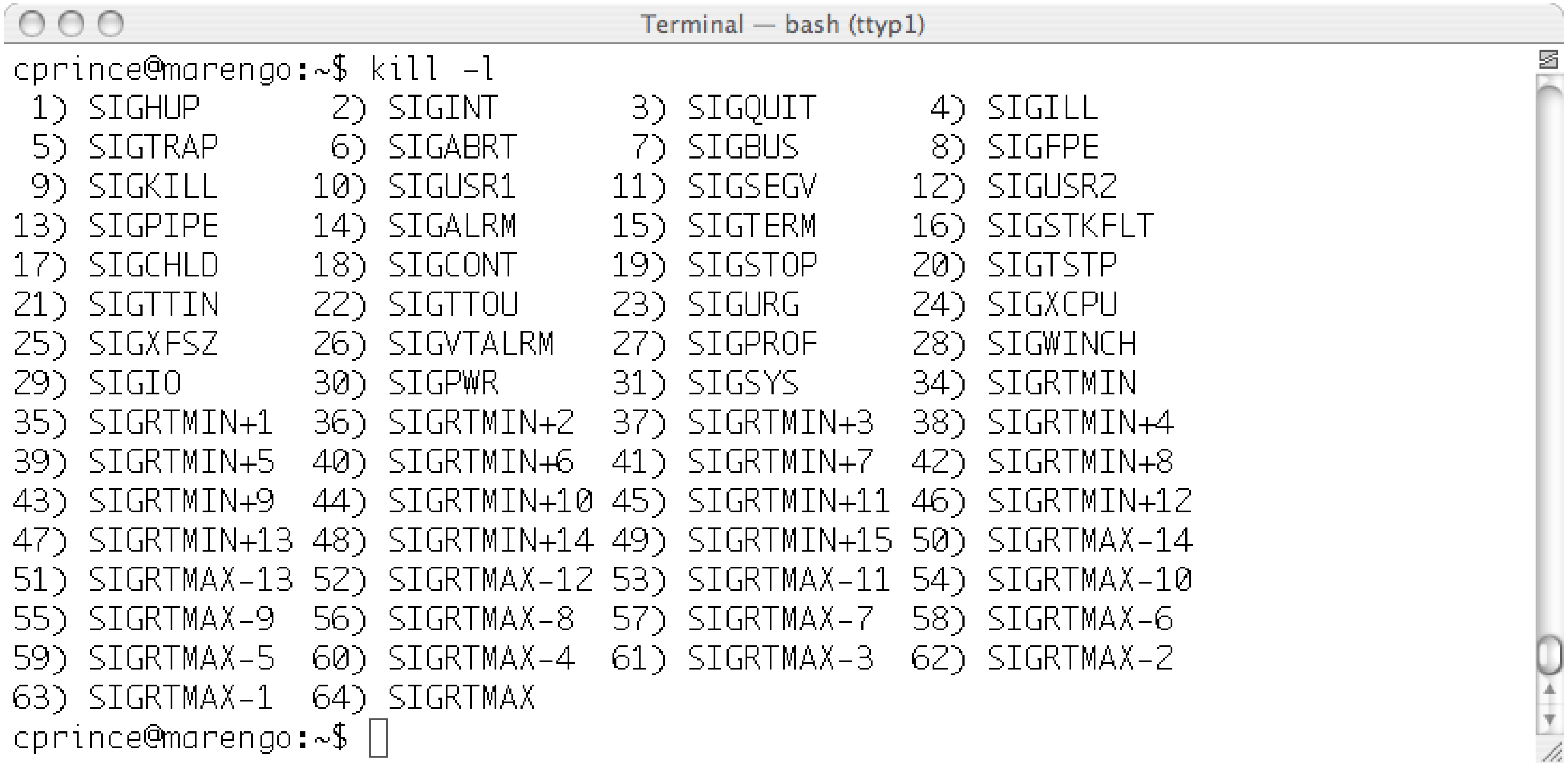
Signal Sources



Signal Types

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

Signals



```
cprince@marengo:~$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
cprince@marengo:~$ 
```

What your interrupt character is?

```
$ stty -a
speed 9600 baud; 0 rows; 0 columns;
lflags: icanon isig iexten echo echoe -echok echoke -echonl echoctl
        -echoprt -altwerase -noflsh -tostop -flusho pendin -
nokerninfo
        -extproc
iflags: -istrip icrnl -inlcr -igncr ixon -ixoff ixany imaxbel -ignbrk
        brkint -inpck -ignpar -parmrk
oflags: opost onlcr -oxtabs
cflags: cread cs8 -parenb -parodd hupcl -clocal -cstopb -crtscts -
dsrflow
        -dtrflow -mdmbuf
cchars: discard = ^C; dsusp = ^Y; eof = ^D; eol = <undef>;
        eol2 = <undef>; erase = ^H; intr = ^C; kill = ^U; lnext = ^V;
        min = 1; quit = ^\; reprint = ^R; start = ^Q; status = ^T;
        stop = ^S; susp = ^Z; time = 0; werase = ^W;
$
```

Generate Signal

- Command line
    - Kill
    - keys
- From process
    - Kill()
    - Raise()
    - Alarm()

Signal Disposition

- Signal Disposition
  - Default handler
  - Our own signal handler
    - Signal()
    - Sigaction
  - Ignore the signal

Note:

- Cannot ignore/handle `SIGKILL` or `SIGSTOP`

Controlling Signal

- Masking & Suspending Signals
  - Sigprocmask()
  - Sigsuspend()
- Pause()

Generating a Signal

- Use the UNIX command:

```
$ kill <signal name> <pid>
```

```
$ kill SIGKILL 4481
```

send a SIGKILL signal to pid 4481

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.

Kill()

pid	***Meaning***
> 0	send signal to process pid
== 0	send signal to all processes whose process group ID equals the sender's pgid.

Raise()

- Sends a signal to the calling process or thread

```
int raise(int signo)
```

pause()

- Suspend the calling process until a signal is caught.

```
#include <unistd.h>
int pause(void);
```

- Returns -1 with errno assigned EINTR. (Linux assigns it ERESTARTNOHAND).
- pause() only returns after a signal handler has returned.

Signal Example: Signals from another process - 1

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

int main() {
    int pid;

    printf("Enter pid of process to send signal to: ");
    scanf("%d", &pid);

    kill(pid, SIGUSR1);
}
```

Signal Disposition

Signal Disposition

- Signal Disposition
  - Default handler
  - Our own signal handler
    - Signal()
    - Sigaction
  - Ignore the signal

Note:

- Cannot ignore/handle `SIGKILL` or `SIGSTOP`

- How to register our own signal handler?
- How to ignore a signal handler?
- How to restore default handler?
- How to restore previous signal handler?

Signal()

-

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

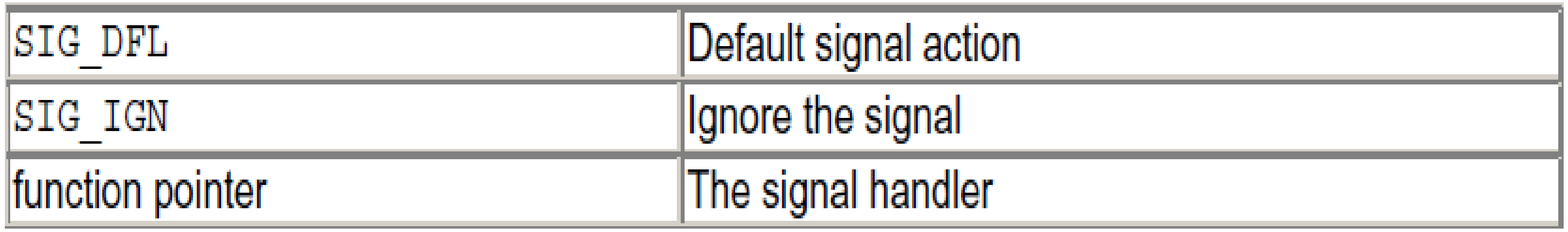
- In Linux:

```
typedef void (*sighandler_t)(int);

sighandler_t signal(int signo, sighandler_t handler);
```

- Signal returns a pointer to a function that returns an int

Argument “func” may be of

- The argument func allows the caller to register the action that is required for the given signal.
- There are three possible values for the argument func.



SIG_DFL	Default signal action
SIG_IGN	Ignore the signal
function pointer	The signal handler

Handling

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

signal t

nt or ig

en as a

sig

The *handler* function
Receives a single integer
Argument and returns *void*

```
#include <signal.h>

void (*signal( int sig, void (*handler)(int))) (int) ;
```

- **signal returns a pointer to the PREVIOUS signal handler**

```
.h>
igfunc(int
1( int sig
```

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.

Example

```
#include<signal.h>
void handler(int signo)
{
        printf("got signal %d \n", signo);
}

int main()
{       /*Registering Our own Handler for SIGINT*/
        signal(SIGINT,handler);
         pause();
        printf("END\n");
return 0;
}
```

Registering Multiple Signal Handlers

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

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

int main()
  {
  int i = 0;
  if( signal( SIGUSR1,sig_usr ) == SIG_ERR )
     printf( "Cannot catch SIGUSR1\n" );
  if( signal( SIGUSR2,sig_usr ) == SIG_ERR )
     printf("Cannot catch SIGUSR2\n");
   }
          :
```

continued

```
void sig_usr( int signo )
/* argument is signal number */
{
  if( signo == SIGUSR1 )
        printf(“Received SIGUSR1\n”);
  else if( signo == SIGUSR2 )
        printf(“Received SIGUSR2\n”);
  else
        printf(“Error: received signal
                   %d\n”, signo);

  return;
}
```

Multiple Signals

- If many signals of the *same* type are waiting to be handled (e.g. two SIGINTs), then most UNIXs will only deliver one of them.
  - the others are thrown away
- If many signals of *different* types are waiting to be handled (e.g. a SIGINT, SIGSEGV, SIGUSR1), they are not delivered in any fixed order.

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

Reset Problem Example

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

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

Reset Problem

To keep catching the signal with this function, must call the *signal* system call again.

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

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 SIGINT 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 foo() when a new SIGINT signal will cause the default action to be carried out -- process termination.

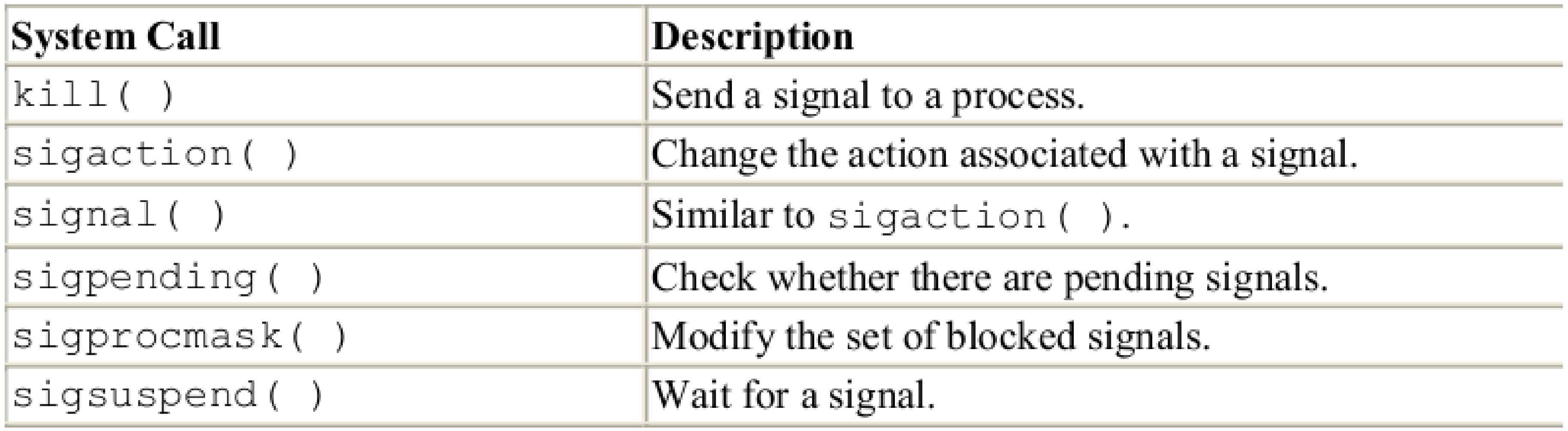
- With signal() there is no answer to this problem.
  - POSIX signal functions solve it (and some other later UNIXs)

Restore Previous Handler

```
Sighandler_t *old_hand;

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

/* do work */

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

Sigaction()

Signal API



System Call	Description
`kill( )`	Send a signal to a process.
`sigaction( )`	Change the action associated with a signal.
`signal( )`	Similar to `sigaction( )`.
`sigpending( )`	Check whether there are pending signals.
`sigprocmask( )`	Modify the set of blocked signals.
`sigsuspend( )`	Wait for a signal.

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)

POSIX Signal Functions

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

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.
- signal sets can be used as masks that enable or disable collections of signals.

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 );
```

Emptying a Signal set

- The function sigemptyset() is used to initialize a signal set to the state of "no signal members."
- The function sigemptyset() accepts a pointer to the set to initialize.

```
sigset_t my_sigs; /* Signal set declaration*/
sigemptyset(&my_signals); /* Clear set */
```

- This example initializes the signal set my_sigs to contain no signals

Filling a Signal Set

- The function sigfillset() fills a signal set with all possible signals.

```
sigset_t   all_sigs;
sigfillset(&all_sigs);
```

- The signal set all_sigs is initialized to contain every possible signal.

Adding Signal to Signal Set

- sigaddset() is used to add new signal to a signal set.
- This function is often used to add a new signal after the set has been emptied.

```
#include <signal.h>
int sigaddset(sigset_t *set, int signum);
```

Example

```
sigset_t two_sigs;
sigemptyset(&two_sigs);          /* Initialize as empty */
sigaddset(&two_sigs,SIGINT);  /* Add SIGINT to set */
sigaddset(&two_sigs,SIGPIPE); /* Add SIGPIPE to set */
```

- The function sigemptyset() initializes the set two_sigs.
- The signals SIGINT and SIGPIPE are then added by calling the function sigaddset().

Removing Signals from signalset

- Sigdelset() is used fro deleting a signal from signalset.
- This function is often used after using sigfillset() to remove one or more signals from the set.

```
#include <signal.h>
int sigdelset(sigset_t *set,int signum);
```

Example for sigdelset()

```
sigset_t sig_msk;
sigfillset(&sig_msk); /* Initialize with all sigs */
sigdelset(&sig_msk,SIGINT);//Del SIGINT from set
```

- The sig_msk set is filled with all possible signals by calling sigfillset(). Function sigdelset() is then used to remove SIGINT from this set.
- The resulting signal set sig_msk includes all signals except SIGINT

Testing for Signals in signalset

- sigismember() is used to test if the signal is a meber of given signalset.

```
#include <signal.h>
int sigismember(const sigset_t *set,int signum);
```

- Returns
  - “1” if “signum” is a member of given signalset ”set”.
  - “0” if “signum” is not a meber of given “set”.

Example for sigismember()

```
sigset_t myset;
sigemptyset(&myset); /* Clear the set */
sigaddset(&myset,SIGINT); /* Add SIGINT to set */
if (sigismember(&myset,SIGINT)) //Test for SIGINT
    puts("HAS SIGINT");
if (sigismember(&myset,SIGPIPE))//Test fr SIGPIPE
    puts("HAS SIGPIPE");
```

- The message HAS SIGINT will be displayed, but since the SIGPIPE signal is not a member of the set, the message HAS SIGPIPE will not be shown

sigaction()

- Supercedes (more powerful than) signal()
  - sigaction() can be used to code a non-resetting signal()

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

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

sigaction Structure

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

sa_flags –

- SIG_DFL reset handler to default upon return
- SA_SIGINFO denotes extra information is passed to handler (.i.e. specifies the use of the “second” handler in the structure.

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 more signo signals are also blocked.
- sa_handler remains installed until it is changed by another sigaction() call. No reset problem.

Signal F

```
int main()
  {
  struct sigaction act;

  act.sa_handler = ouch;

  sigemptyset( &act.sa_mask );

  act.sa_flags = 0;

                 SIGINT, &act,

                 'Hello
     sleep(1);
  }
  }
```

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

Set the signal handler to be the function `ouch`

**No flags are needed here. Possible flags include:
SA_NOCLDSTOP
SA_RESETHAND**

We can manipulate sets of signals..

This call sets the signal handler for the SIGINT (ctrl-C) signal

sigaction() Demo

```
#include<signal.h>
int count = 0;

void handler(int sig)
{
        ++count;
        printf("Got signal %d count %d\n", sig, count);
}

int main()
{
        struct sigaction sa_old, sa_new;
        /*Filling the sigaction structure*/
        sa_new.sa_handler = handler;
        sigemptyset(&sa_new.sa_mask);
        sa_new.sa_flags = 0;
```

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

sigaction() Demo

```
    //Registering Our own Handler for SIGINT
     sigaction(SIGINT, &sa_new, &sa_old);
     while(count < 2) {
          printf("waiting for signal\n");
          sleep(4);
     }
     //Restoring the old handler back
     sigaction(SIGINT, &sa_old, &sa_new);
     printf("END\n");
return 0;
}
```

Sigaction()

- The sigaction() function allows you to query the current signal action without modifying the current action for the indicated signal.

```
struct sigaction sa_old;         /* Queried signal set */
sigaction(SIGINT,0,&sa_old); /* Query SIGINT */
if ( sa_old.sa_handler == SIG_DFL )
     puts("SIG_DFL");            /* System Default */
else if ( sa_old.sa_handler == SIG_IGN )
     puts("SIG_IGN");            /* Ignore signal */
else                             /* Function Pointer */
  printf("sa_handler = 0x%08lX;\n",(long)sa_old.sa_handler);
```

Controlling Signals

- Blocking Signals
  - Sigprocmask()
- Obtaining Pending Signals
  - Sigpending()
    - To know about the pending signals
  - Sigsuspend()
    - reliable way to unblock that signal and allow the signal to be raised after noticing that the signal is in pending.

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

Blocking signals in Critical Code Region

```
sigset_t blk, svmask;

sigemptyset( &blk );
sigaddset( &blk, SIGINT );

/* block SIGINT; save old mask */
sigprocmask( SIG_BLOCK, &blk, &svmask );

/* critical region of code  SIGINT is blocked here*/

/* reset mask which unblocks SIGINT */
sigprocmask( SIG_SETMASK, &svmask, NULL );
```

Signals - Ignoring signals

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

- Instead of in the previous program:

```
  act.sa_handler = catchint /* or whatever */
We use:
  act.sa_handler = SIG_IGN;
The ^C 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 );
```

Obtaining Pending Signals

- Sigpending() is used
  - When signals are blocked by the sigprocmask(), they become pending signals, rather than being lost.

```
#include <signal.h>
int sigpending(sigset_t *set);
Returns     "0" on Success
            "1" on Failure
```

Example for pending signals

```
sigset_t pendg;          /* Pending signal set */
sigpending(&pendg); /* Inquire of pending signals */
if ( sigismember(&pendg,SIGPIPE) ) {
        puts("SIGPIPE is pending.");
}
```

- The set of pending signals is copied to the set provided in argument set.
- Example illustrates that signal SIGPIPE is blocked and how to test if the same signal is pending.
- It is useful when a program is in a critical code loop and needs to test for a pending signal.

Sigsuspend()

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- It temporarily applies the signal mask supplied in argument mask and then waits for the signal to be raised.
- If the mask permits the signal you know to be pending, the signal action will take place immediately.
- Once the signal action is carried out, the original signal mask is re-established

Example for sigsuspend()

```
sigset_t pendg; /* Pending signal set */
sigset_t notpipe; /* All but SIGPIPE */
sigfillset(&notpipe); /* Set to all signals */
sigdelset(&notpipe,SIGPIPE); /*Remove SIGPIPE */
sigpending(&pendg);//Query which signals are pending
/* Is SIGPIPE pending? */
if ( sigismember(&pendg,SIGPIPE) ) {
    sigsuspend(&notpipe); /* Yes, allow SIGPIPE to be
    }                                   raised */
```

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

Thank You