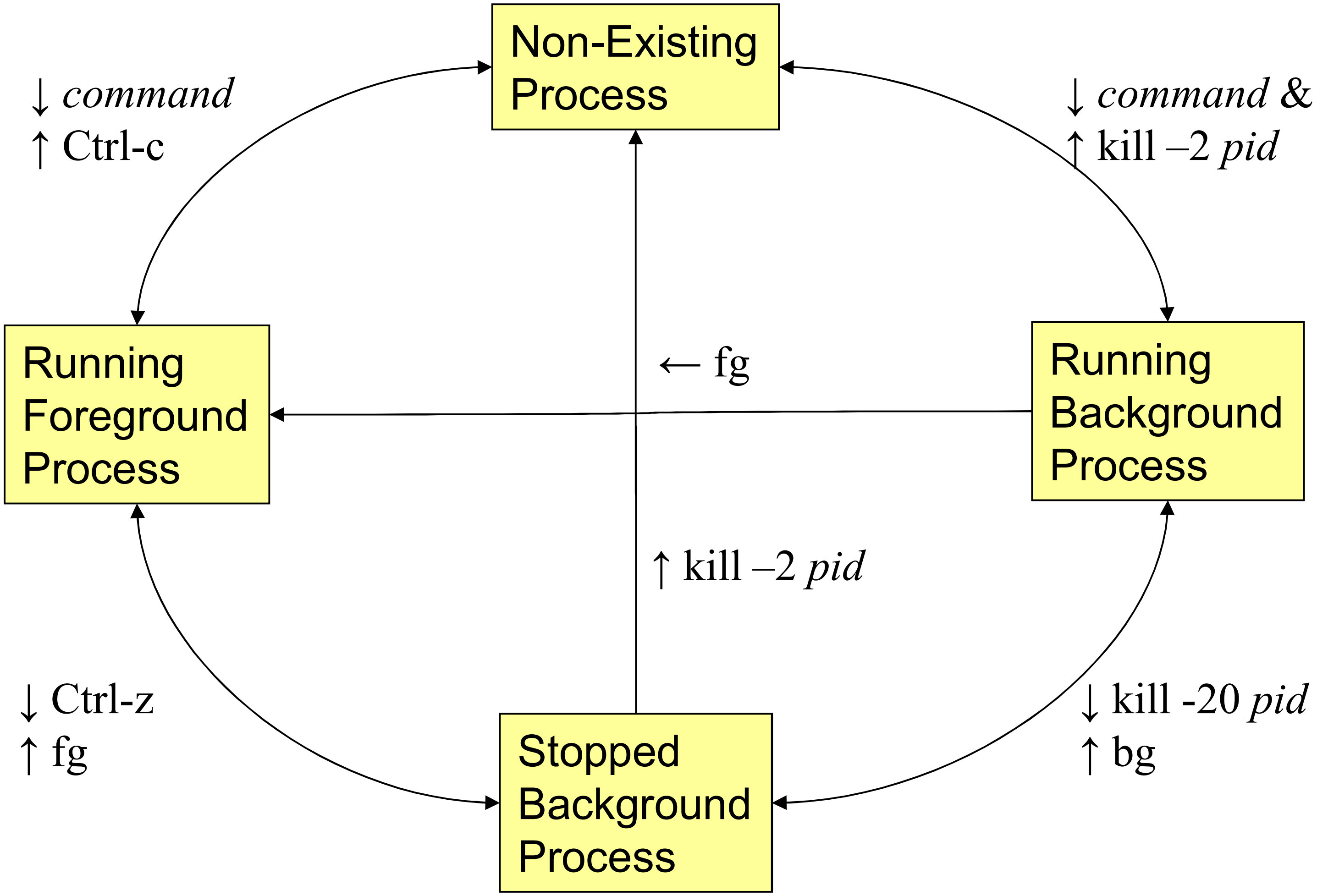
11-UNIX

The **signal** System Call

Chapter 20,24,25,26

UNIX Process Control



sleep – system command

Definition - delay for a specified amount of time

Form - sleep NUMBER[SUFFIX]...
sleep OPTION

Description - Pause for NUMBER seconds.

UNIX Process Control

[Demo of UNIX process control using **infloop.c**]

```
/**********************************/
/*               infloop.c               */
/**********************************/
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* print doing something, rest, and repeat again */
{
   for (;;) {
     printf("doing something ...\n") ;
     sleep(2);
   }
   return EXIT_SUCCESS;
}
```

Running *infloop*

```
[bielr@athena ClassExamples]> infloop
doing something ...
doing something ...
doing something ...
doing something ...
doing something ...
^C
[bielr@athena ClassExamples]>
```

Signals (LPI page 388)

A *signal* is a notification to a process that an event has occurred.

Signals can come from another process or the kernel. A process can also send a signal to itself.

Signals from the Kernel

- Types of events that cause the kernel to generate a signal:
    - Hardware exceptions
        - Problem with a machine instruction
        - Divide by zero
        - A reference to inaccessible memory
    - User-typed special characters
        - CTRL-C or CTRL-Z
    - Software event
        - Timer went off
        - Child of this process terminated
        - Terminal window was resized

Signals

Standard signals in Linux have:

- names.   Ex. SIGINT
- numbers. Ex. SIGINT has the number 2
- Linux has 31 standard signals
- Standard signals are sometimes called *Traditional* signals

- *Realtime signals* (LPI section 22.8)
  - Extension set to Standard signals
  - Allow signals to be queued
  - Allow data to accompany the signal

Signal Types (31 in POSIX) Table 20-1

Name	Description	Default Action
SIGINT	Interrupt character typed	terminate process
SIGQUIT	Quit character typed (^\)	create core image
SIGKILL	Sure kill	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
SIGCHLD	user-defined signal type	ignore

See `man 7 signal`

For more, see table 20-1 in our LPI book, page 396.

Process Control Implementation (1 of 2)

Exactly what happens when you:

Type **Ctrl-c**?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 2/SIGINT **signal**

Type **Ctrl-z**?

- Keyboard sends hardware interrupt
- Hardware interrupt is handled by OS
- OS sends a 20/SIGTSTP **signal**

Process Control Implementation (2 of 2)

Exactly what happens when you:

Issue a "**kill –*sig pid***" command?

- OS sends a *sig* **signal** to the process whose id is *pid*

Issue a **"fg" or "bg"** command?

- OS sends a 18/SIGCONT **signal**
- fg – foreground process, a shell command
- bg – background process, a shell command

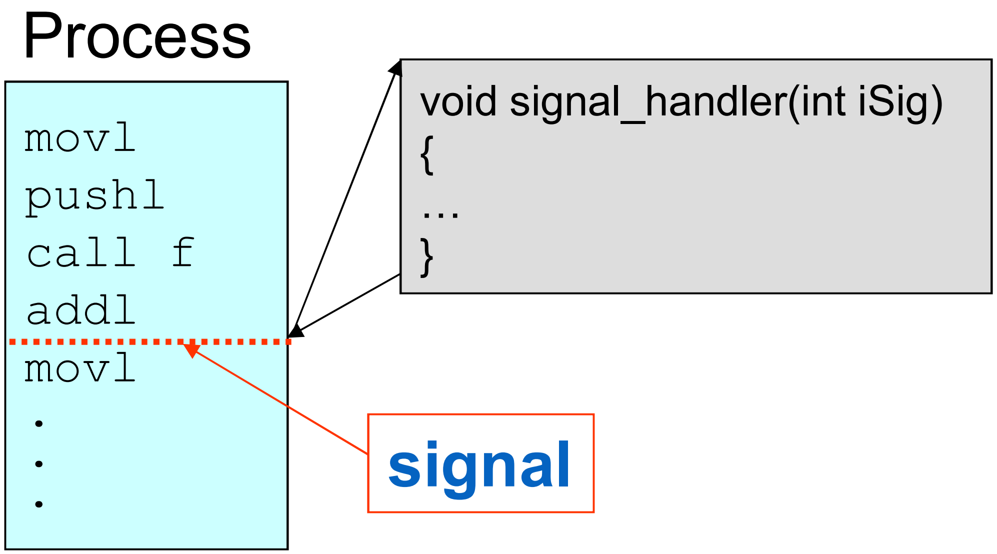
Definition of Signal

Signal: A signal is an *asynchronous* event which is delivered to a process.

Event gains attention of the OS

- OS stops the application process immediately, sending it a signal
- **Signal handler** executes to completion
- Application process resumes where it left off

e.g. user types
ctrl-C
at anytime.



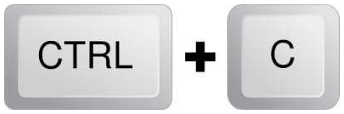
Process

```
movl
pushl
call  f
addl
movl
.
.
.
```

```
void signal_handler(int iSig)
{
…
}
```

Examples of Signals (1 of 2)

User types Ctrl-c



- Event gains attention of OS

- OS stops the application process immediately, sending it a 2/SIGINT signal

- Signal handler for 2/SIGINT signal executes to completion
  Default signal handler for 2/SIGINT signal exits process

Examples of Signals (2 of 2)

Process makes illegal memory reference

```
int *ptr;
*ptr = 20;
```

- Event gains attention of OS

- OS stops application process immediately, sending it a 11/**SIGSEGV** signal

- Signal handler for 11/SIGSEGV signal executes to completion

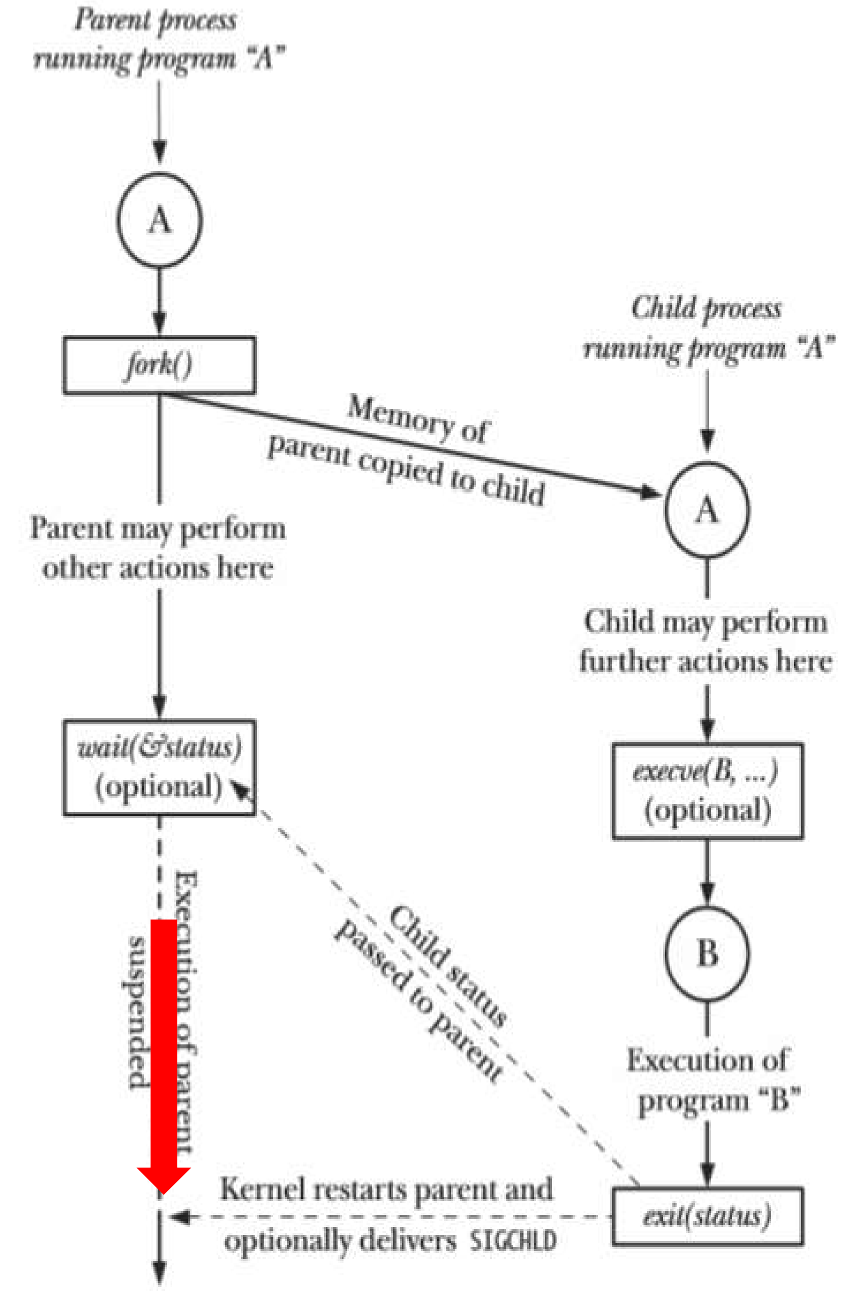
  Default signal handler for 11/SIGSEGV signal prints "segmentation fault" and exits process

PS: Since **ptr** is uninitialized, we would get a SEG_FAULT

If we said "ptr = 20", it would not get fault until you tried to access *ptr

Example of Signal:

fork, exit, wait, & *execve*



Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

Sending Signals via Keystrokes

Three signals can be sent from keyboard:

Ctrl-c → 2/SIGINT signal
Default handler exits process

Ctrl-z → 20/SIGTSTP signal
Default handler suspends process

Ctrl-\ → 3/SIGQUIT signal
Default handler exits process

Sending Signals via Commands

`kill -`*`signal pid`*

Send a signal of type **`signal`** *to the process with id* **`pid`**

Can specify either signal type name (-SIGINT) or number (-2)

No signal type name or number specified => sends 15/SIGTERM signal

Default 15/SIGTERM handler exits process

Examples:

```
kill –2 1234
kill -SIGINT 1234
```

Same as pressing Ctrl-c if process 1234 is running in foreground

signal call

```
#include <signal.h>

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

            Returns previous signal disposition on success,
            or SIG_ERR on error
```

- *sig* – the signal whose disposition we wish to change
- *handler* – the address of the function that should be called when this signal is delivered

Form:

```
void handler (int sig)
{
     /* code for the handler */
}
```

Sending Signals via Function Call

`raise()`

```
int raise(int iSig);
```

- Commands OS to send a signal of type **`iSig`** to current process, **itself**.
- Returns 0 to indicate success, non-0 to indicate failure

Example

```
int ret = raise(SIGINT);
        /* Process commits termination. */
        /* where SIGINT is the number    */
        /* of the signal to  send        */
```

Sending Signals via Function Call

`kill()`

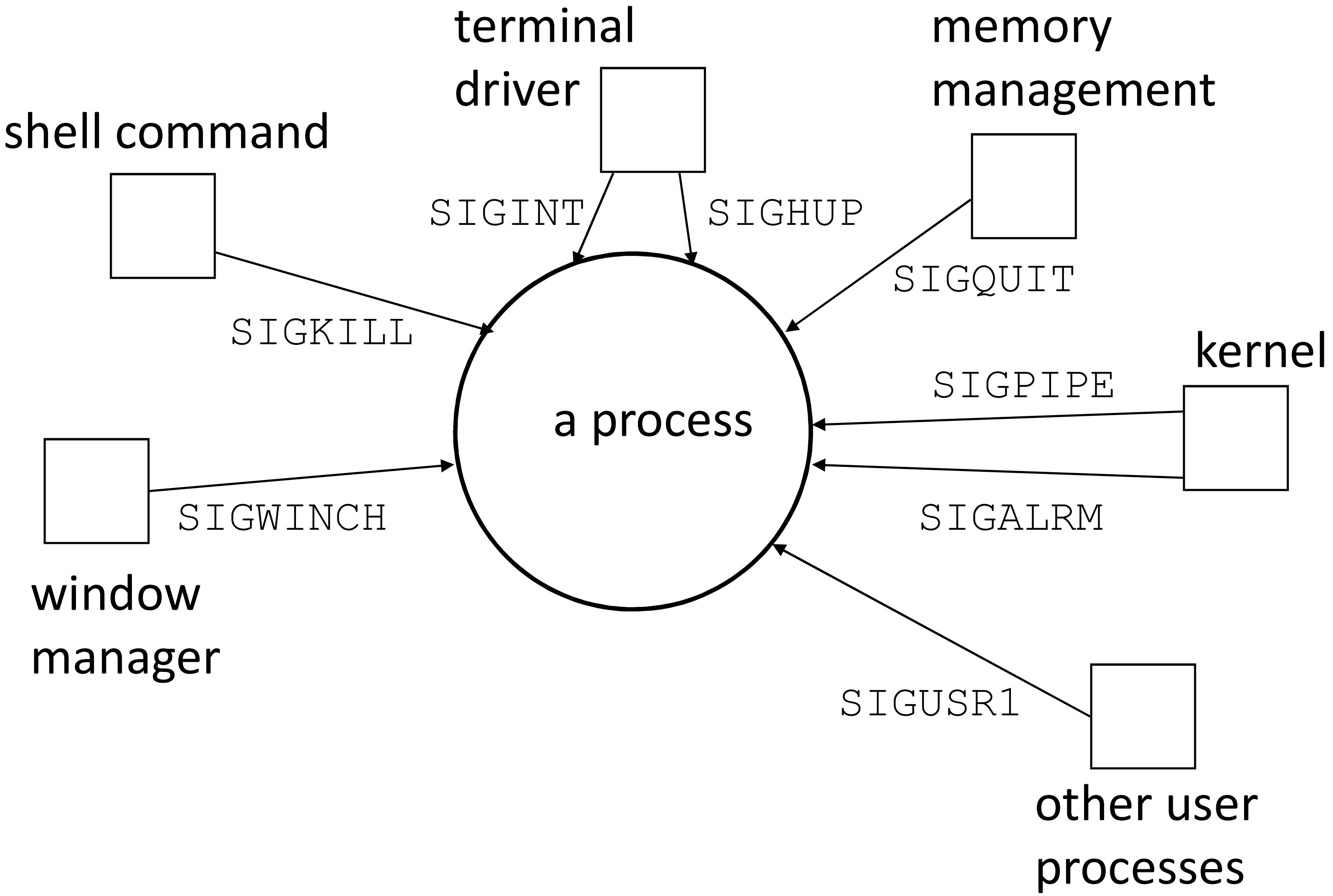
```
int kill(pid_t iPid, int iSig);
```

- Sends a `iSig` signal to the process whose id is `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process

Example:

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT);    /* Process sends itself a
                          SIGINT signal */
```

Signal Sources



Responding to a Signal

A process can:

- ignore/discard the signal
  - not possible with `SIGKILL` or `SIGSTOP`
- execute a **signal handler** function, and then possibly resume execution or terminate
- carry out the default action for that signal

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

Signal delivery & handler execution

(LPI Page 399)

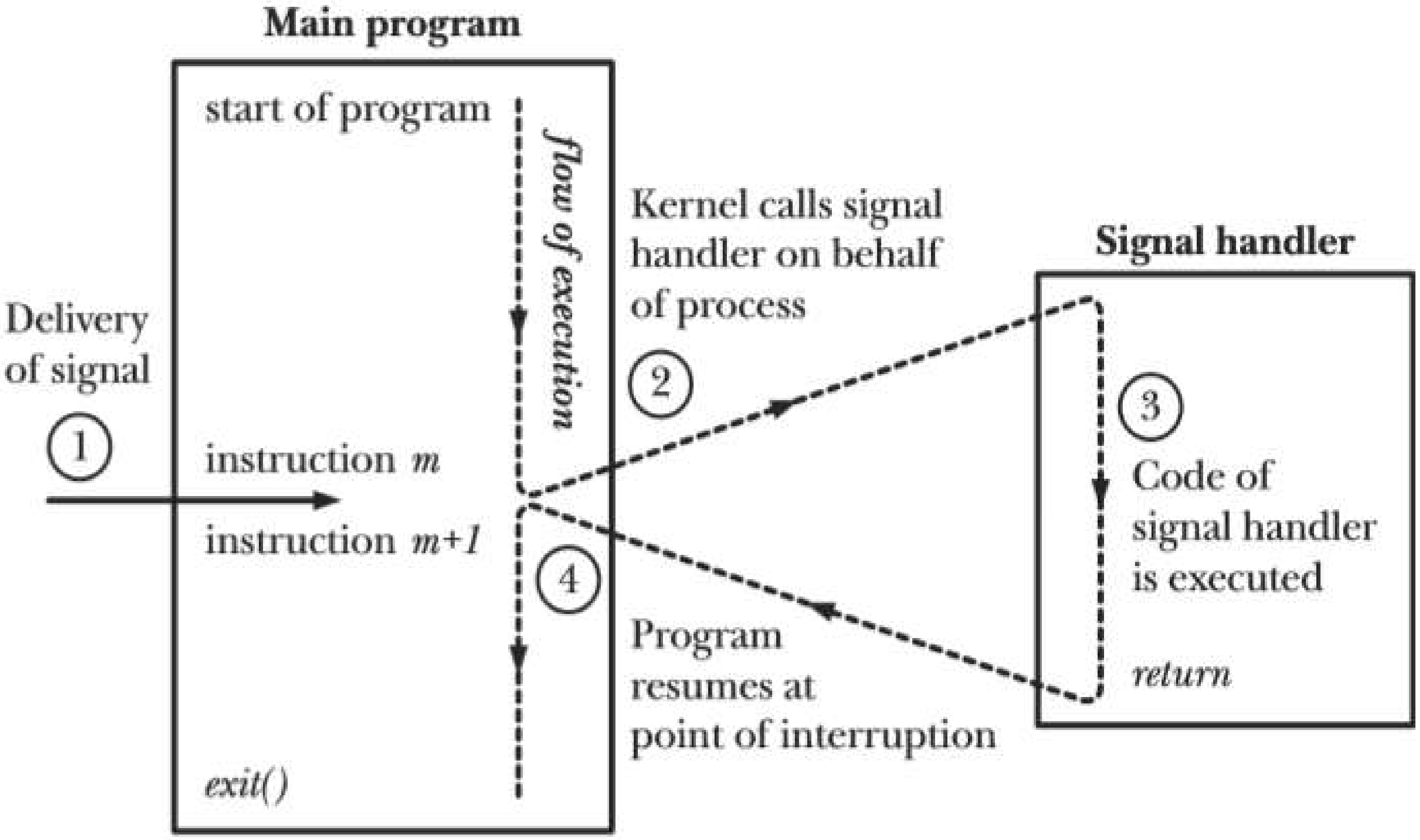

Main program
start of program
flow of execution
Delivery of signal
1
instruction m
instruction m+1
exit()
Kernel calls signal handler on behalf of process
2
Signal handler
3
Code of signal handler is executed
return
4
Program resumes at point of interruption

Figure 20-1: Signal delivery and handler execution

Example: ouch.c (LPI page 399)

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

static void sigHandler(int sig) {
    printf("Ouch!\n");  /* UNSAFE (see Section 21.1.2) */
}
int main(int argc, char *argv[]) {
    int j;
    if (signal(SIGINT, sigHandler) == SIG_ERR)
        fprintf(stderr, "signal");
    for (j = 0; ; j++) {
        printf("%d\n", j);
        sleep(3);                       /* Loop slowly... */
    }
    return EXIT_SUCCESS:
}
```

Running the *ouch* program

```
[bielr@athena ClassExamples]> ouch
0
1
^C Ouch!
2
3
^C Ouch!
4
5
6
^C Ouch!
7
8
^\Quit (core dumped)
[bielr@athena ClassExamples]>
```

Chapter 21.
Signals: Signal Handlers

Async-Signal-Safe Function (1 of 2)

An Async-Signal-Safe function is one in which the implementation guarantees to be safe when calling from the signal handler.

A function is Async-Signal-Safe because it is **not interruptible** by a signal handler.

```
Note: /* UNSAFE (see Section 21.1.2)
         Page 422  */
```

Async-Signal-Safe Function (2 of 2)

For example:

Suppose a program is in the middle of a call to printf(3) and a signal occurs whose handler itself calls printf().

In this case, the output of the two printf() statements would be intertwined. To avoid this, the handler should not call printf() itself when printf() might be interrupted by a signal.

Question: Then, what function(s) would we use instead?

Async Signal Safe Functions

(Table 21-1/Page 426, UPPER half of table)

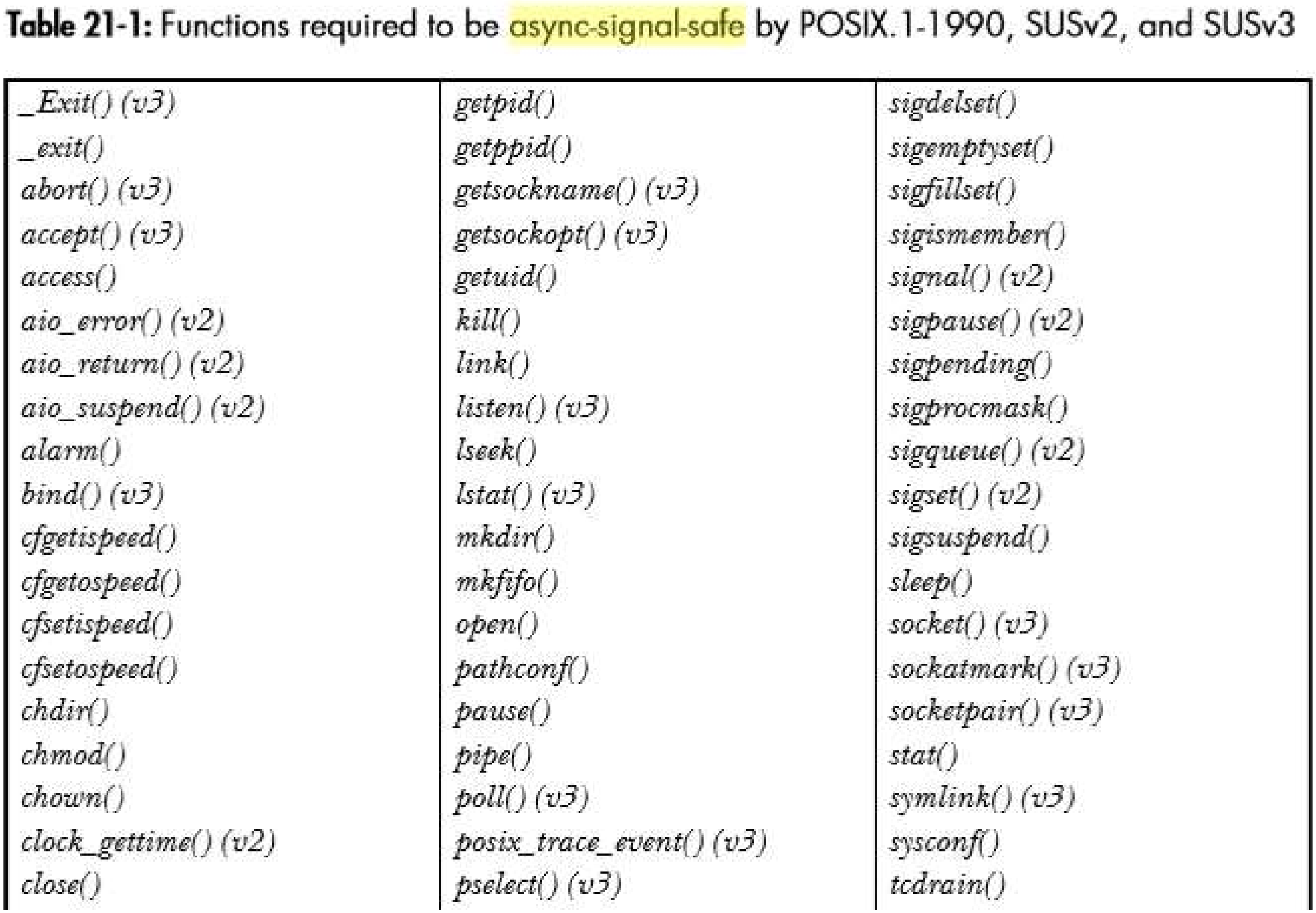


Table 21-1: Functions required to be async-signal-safe by POSIX.1-1990, SUSv2, and SUSv3

_Exit() (v3)
_exit()
abort() (v3)
accept() (v3)
access()
aio_error() (v2)
aio_return() (v2)
aio_suspend() (v2)
alarm()
bind() (v3)
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
clock_gettime() (v2)
close()

getpid()
getppid()
getsockname() (v3)
getsockopt() (v3)
getuid()
kill()
link()
listen() (v3)
lseek()
lstat() (v3)
mkdir()
mkfifo()
open()
pathconf()
pause()
pipe()
poll() (v3)
posix_trace_event() (v3)
pselect() (v3)

sigdelset()
sigemptyset()
sigfillset()
sigismember()
signal() (v2)
sigpause() (v2)
sigpending()
sigprocmask()
sigqueue() (v2)
sigset() (v2)
sigsuspend()
sleep()
socket() (v3)
sockatmark() (v3)
socketpair() (v3)
stat()
symlink() (v3)
sysconf()
tcdrain()

Special Sigfunc* Values used in signal() function

Value ***Meaning***

`SIG_IGN` Ignore / discard the signal.

Example: To ignore a ctrl-c command from the command line.

`SIG_DFL` Use default action to handle signal.

Example: To reset system so that SIGINT causes a termination at any place in our program.

`SIG_ERR` Returned by **`signal`**`()` as an error.

Handling Multiple Signals

If many signals of the *same* type are waiting to be handled (e.g. two `SIGINT`s), then most UNIXs will only deliver one of them. (Signals are not queue). 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.

pause()

Suspend the calling process until a signal is caught.

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

Returns -1 with **`errno`** assigned **`EINTR`**.
(Linux assigns it `E`**`RESTARTNOHAND`**).

`pause()` only returns after a signal handler has finished.

pause() Example

(1 of 3)

```
#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");
  while(1) {
     printf("%2d\n", i );
     pause();
      /* pause until signal handler has processed signal */
      i++;
  }           /* end of while loop */
  return 0;
}             /* end of main */
```

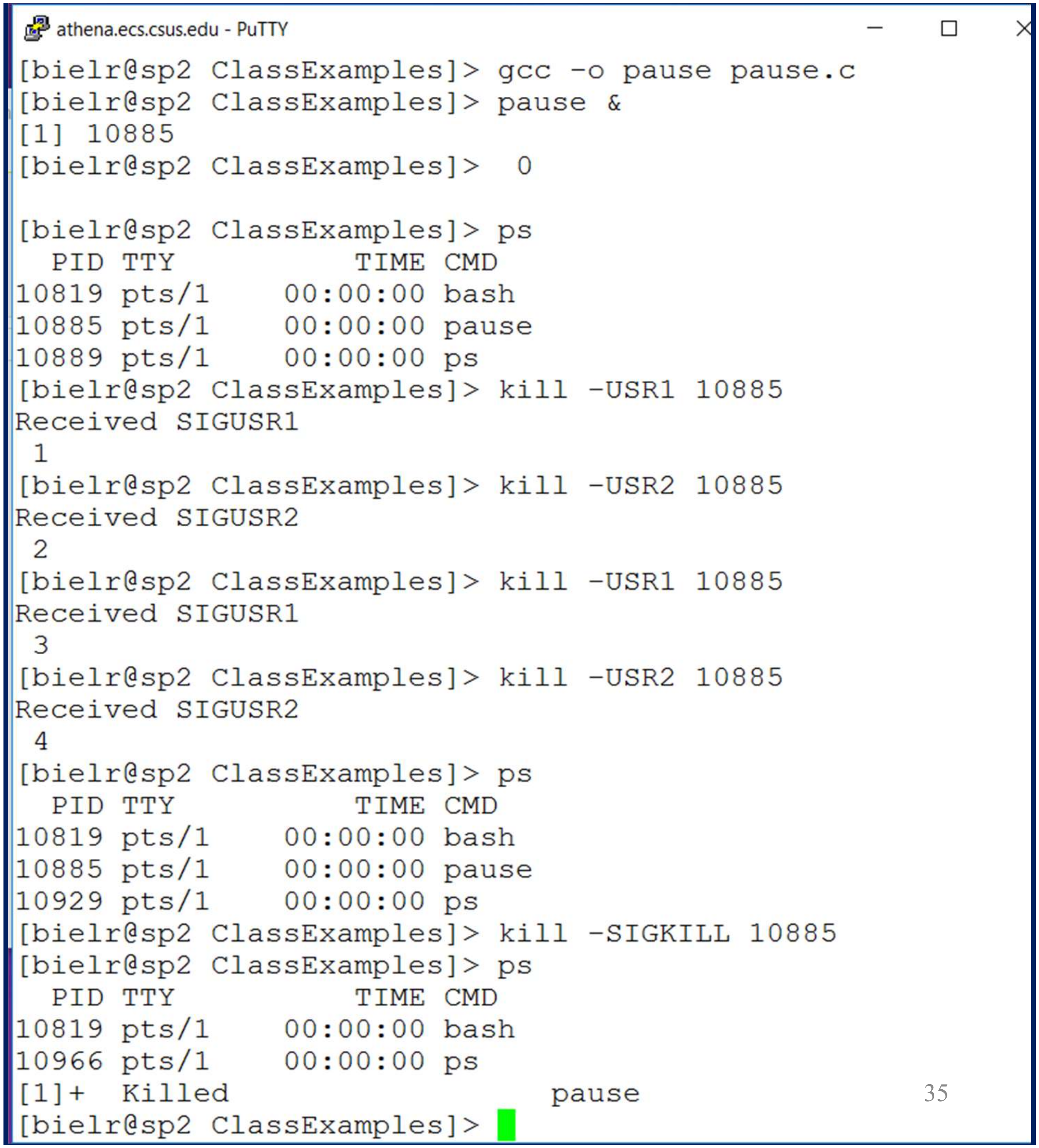
pause() – Example (2 of 3)

```
/* argument is signal number */
void sig_usr( int signo )
 {
    if( signo == SIGUSR1 )
       printf("Received SIGUSR1\n");
    else if( signo == SIGUSR2 )
       printf("Received SIGUSR2\n");
    return;
 }
```

Note: Executing a program with an “&” puts the program in the background.

pause() – Example

(3 of 3)



```
[bielr@sp2 ClassExamples]> gcc -o pause pause.c
[bielr@sp2 ClassExamples]> pause &
[1] 10885
[bielr@sp2 ClassExamples]>  0

[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10885 pts/1    00:00:00 pause
10889 pts/1    00:00:00 ps
[bielr@sp2 ClassExamples]> kill -USR1 10885
Received SIGUSR1
 1
[bielr@sp2 ClassExamples]> kill -USR2 10885
Received SIGUSR2
 2
[bielr@sp2 ClassExamples]> kill -USR1 10885
Received SIGUSR1
 3
[bielr@sp2 ClassExamples]> kill -USR2 10885
Received SIGUSR2
 4
[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10885 pts/1    00:00:00 pause
10929 pts/1    00:00:00 ps
[bielr@sp2 ClassExamples]> kill -SIGKILL 10885
[bielr@sp2 ClassExamples]> ps
  PID TTY          TIME CMD
10819 pts/1    00:00:00 bash
10966 pts/1    00:00:00 ps
[1]+  Killed                  pause
[bielr@sp2 ClassExamples]> 
```

kill and raise functions

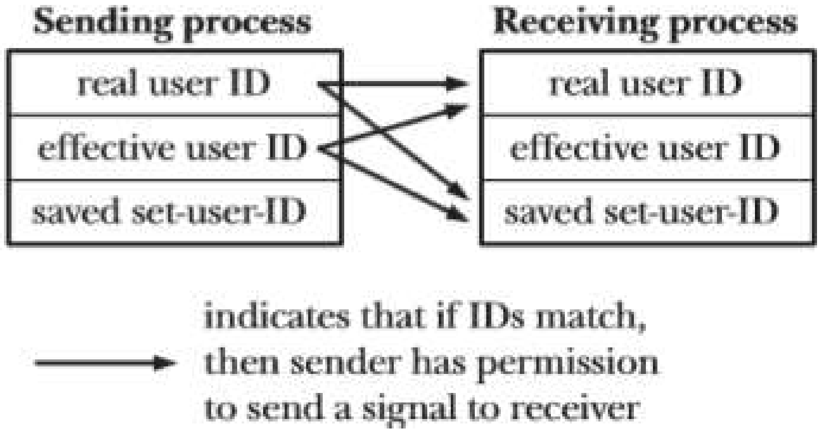
int kill(pid_t pid, int sig);

int raise(int sig);

kill sends a signal to a process or group of processes

- pid > 0 – send to process with PID = pid
- pid = 0 – send to all processes with PGID = PGID of caller
- pid < 0 – send to all processes with PGID = |pid| (to a group)
- pid = -1 – send to all processes which it has permission to send a signal

To send signals to other processes,
Real user ID/EUID (Effective User Id)
must match



indicates that if IDs match, then sender has permission to send a signal to receiver

raise allows a process to send a signal to itself

Signal Sets

Multiple signals are represented using a data structure called a *signal set*, provided by the system data type *sigset_t*.

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>

/* Initialize signal set to contain no member */
        int sigemptyset( sigset_t *set );

/* Initialize signal set to contain all signal */
        int sigfillset( sigset_t *set );

/* Add individual signal */
        int sigaddset( sigset_t *set, int signo );

/* Remove individual signal */

        int sigdelset( sigset_t *set, int signo );

/* Check to see if a signal is a member of a set */
        int sigismember( const sigset_t *set, int signo );
```

The Signal Mask

For each process, the kernel maintains a *signal mask* – a set of signals whose delivery to the process is currently blocked.

If a signal that is blocked is sent to a process,

delivery of that signal is delayed

until it is unblocked by being removed from the process signal mask

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 */
```

Note: SIGKILL and SIGSTOP cannot be blocked by sigprocmask.

“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, SIGINT );

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

/* critical region of code */
/* where a signal would interfere */
/* with task */

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

Signal - Review

- A signal is a notification that some kind of event has occurred.
- Send to a process by kernel, another process, or by itself.
- Different kind of signals. Each has unique id and purpose.
- Signal is typically asynchronous (unpredictable).
- Signal can be ignored, terminate a process, stop a process, or restart of stopped process. Also, see table 20-1.
- Signal can be also be ignored and handled by a programmer’s handler (catcher) function.
    - Later: Recommend to use sigaction() – more flexible/portable

sigaction() system call

- An alternative to *signal()*
- Used to change the action taken by a process on receipt of a specific signal.
- More complex than *signal()* but offers greater flexibility

(See signal(7) for an overview of signals.)

sigaction() System Call

```
#include <signal.h>

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

The arguments are explained on the next slide.

Note: why there is no const in front of struct sigaction *oldact ?

Because the call is going to modify it, so no constant!

sigaction() arguments

- *sig* – identifies the signal whose disposition we want to retrieve or change.

- *act* – pointer to a structure specifying a new disposition for the signal

- *oact* – pointer to a structure of the same type & is used to return information about the signal's previous disposition

sigaction Structure

```
struct sigaction
{
        void  (*sa_handler)( int );             / * address of handler */

        sigset_t        sa_mask;                /* signal blocked during invocation */

        int  sa_flags;                          /* flags control invocation  */
                                                /* (see page 417) */

        void  (*sa_sigaction)( int, siginfo_t *, void * );
                                                /* not for application use */
}
```

sa_flags – (typically has a 0 value)

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.

- sa_handler remains installed until it is changed by another sigaction() call.

Signal Raising

```
int main() {
    struct sigaction act;
    act.sa_handler = ouch;
    sigemptyset( &act.sa_mask );
    act.sa_flags = 0;
    sigaction( SIGINT, &act, 0 );
    while(1) {
        printf("Hello World!\n");
        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
SA_RESTART
SA_NODEFER

We can manipulate sets of signals..

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

```
#include <signal.h>        //the code (that works)
#include <stdlib.h>
#include <stdio.h>

static void ouch(int sig) {
   printf("Ouch!\n");  /* UNSAFE (see Section 21.1.2) */
}

int main(void)
{
        struct sigaction act;
        act.sa_handler = ouch;
        sigemptyset( &act.sa_mask );
        act.sa_flags = 0;
        sigaction( SIGINT, &act, 0 );
         while(1) {
              printf("Hello World!\n");
              sleep(1);
        }
}
```

Signals - Ignoring signals

Other than SIGKILL and SIGSTOP, signals can be ignored:

Instead of in the previous program:

```
    act.sa_handler = ouch; /* 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 );
```

11-UNIX

The **signal** System Call

The End