# Chap 10. Signals

Hongik University

Eunsung Jung

*Disclaimer: The slides are borrowed from many sources!*

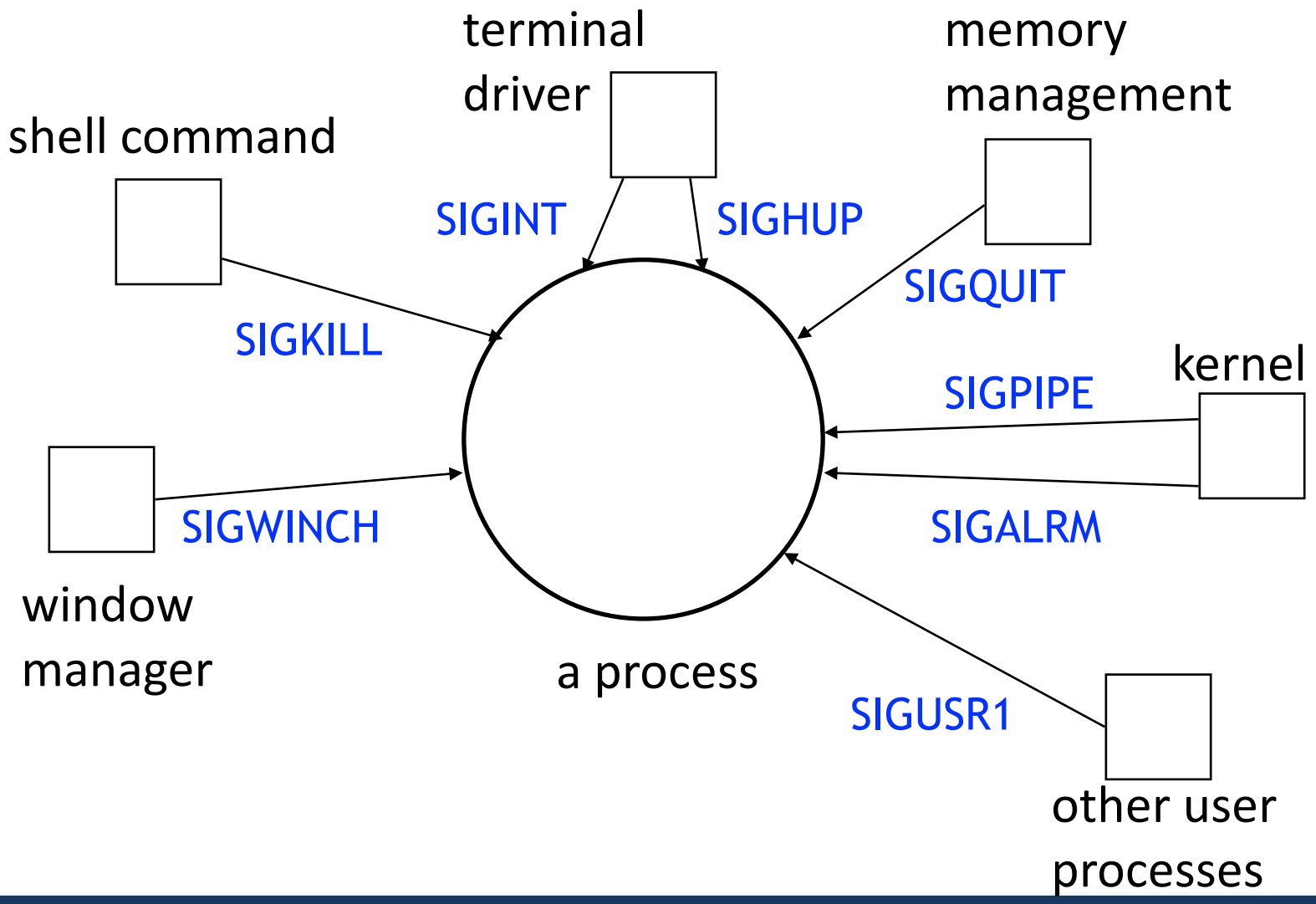
# Signals

- Signals are software interrupts from unexpected events
  - an illegal operation (e.g., divide by 0)
  - a power failure
  - an alarm clock
  - the death of a child process
  - a termination request from a user (Ctrl-C)
  - a suspend request from a user (Ctrl-Z)

# Predefined Signals (1/2)

- 31 signals
- Every signal has a name.
  - begin with ‘SIG’
  - SIGABRT: abort signal from abort()
  - SIGALRM: alarm signal from alarm()
- Actions of the default signal handler
  - terminate the process and generate a core (dump)
  - ignores and discards the signal (ignore)
  - suspends the process (suspend)
  - resume the process
- Default signal handler can be overridden.

# Signal Sources



# Predefined Signals (2/2)

esjung — esjung@hpclab:/usr/include — ssh hpclab — 84×32

```
#define SIGTERM         15
#define SIGSTKFLT       16
#define SIGCHLD         17
#define SIGCONT         18
#define SIGSTOP         19
#define SIGTSTP         20
#define SIGTTIN         21
#define SIGTTOU         22
#define SIGURG          23
#define SIGXCPU         24
#define SIGXFSZ         25
#define SIGVTALRM       26
#define SIGPROF         27
#define SIGWINCH        28
#define SIGIO           29
#define SIGPOLL         SIGIO
/*
#define SIGLOST         29
*/
#define SIGPWR          30
#define SIGSYS          31
#define SIGUNUSED       31

/* These should not be considered constants from userland.  */
#define SIGRTMIN        32
#define SIGRTMAX        _NSIG

/*
 * SA_FLAGS values:
 *
* SA_ONSTACK indicates that a registered stack_t will be used.
                                                        65,1          34%
```

# Signal Generation

- Terminal-generated signals
  - CTRL-C → SIGINT
  - CTRL-Z → SIGSTP signal
- Hardware exceptions generate signals
  - divide by 0 → SIGFPE
  - invalid memory reference → SIGSEGV
- kill()
  - sends any signal to a process or process group
  - need to be owner or super-user
- Software conditions
  - SIGALRM: alarm clock expires
  - SIGPIPE: broken pipe
  - SIGURG: out-of-band network data

# Handling of Signals

- Disposition or action:
  Process has to tell the kernel "if and when this signal occurs, do the following."

- Ignore the signal:
  all signals can be ignored, except SIGKILL and SIGSTOP

- Let the default action apply:
  most are to terminate process

# Representative UNIX Signals (1/2)

- **SIGART**: generated by calling the abort function.
- **SIGALRM**: generated when a timer set with the alarm expires.
- **SIGCHLD**: whenever a process terminates or stops, the signal is sent to the parent.
- **SIGCONT**: this signal sent to a stopped process when it is continued.
- **SIGFPE**: signals an arithmetic exception, such as divide-by-0, floating point overflow, and so on
- **SIGILL**: indicates that the process has executed an illegal hardware instruction.
- **SIGINT**: generated by the terminal driver when we type the interrupt key and sent to all processes in the foreground process group.

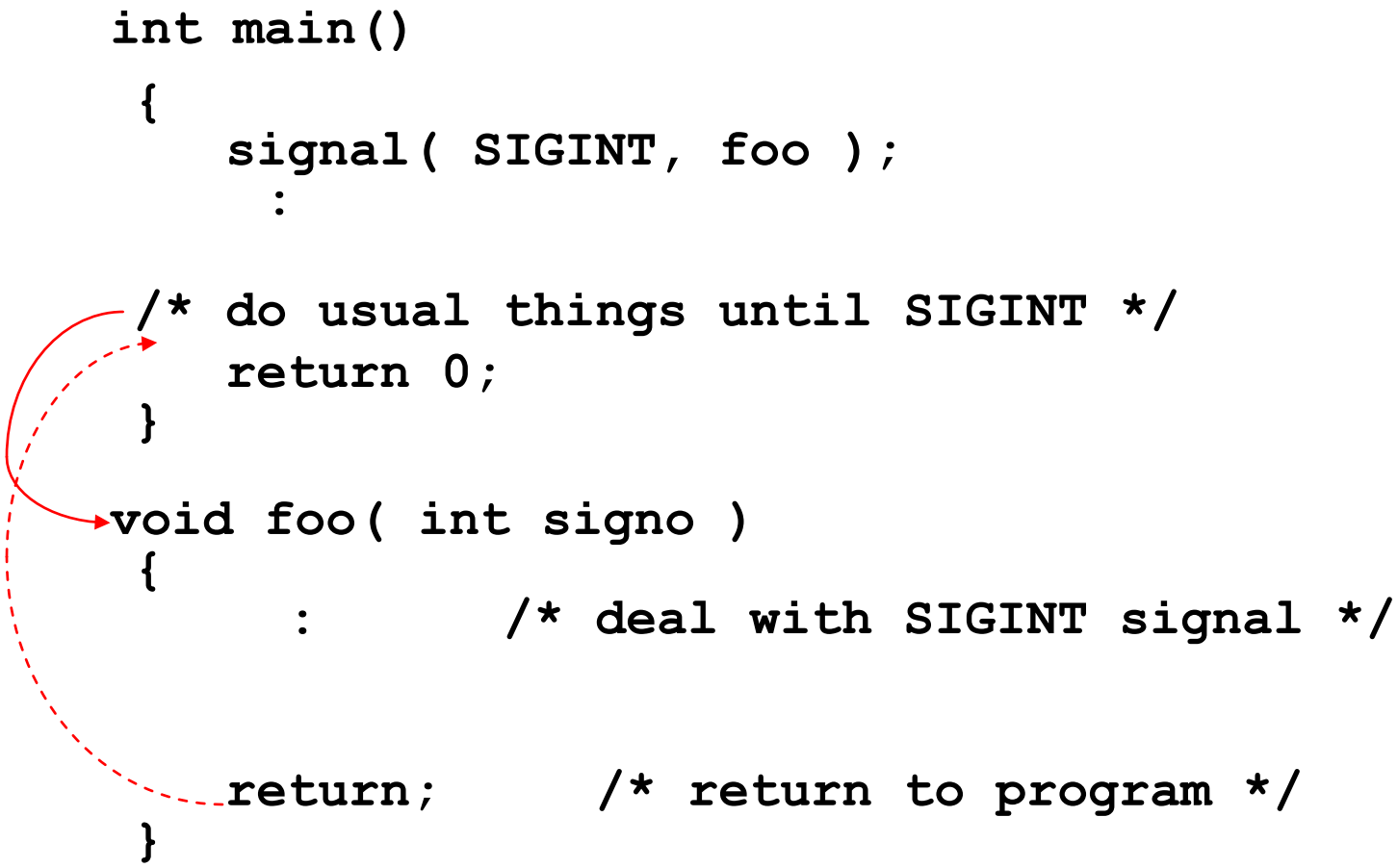
# Representative UNIX Signals (2/2)

- **SIGKILL**: can't be caught or ignored. a sure way to kill any process.
- **SIGPIPE**: if we write to a pipeline but the reader has terminated, SIGPIPE is generated.
- **SIGSEGV**: indicates that the process has made an invalid memory reference. (→ core dumped)
- **SIGTERM**: the termination signal sent by the kill(1) command by default.
- **SIGSTP**: Cntl-Z from the terminal driver which is sent to all processes in the foreground process group.
- **SIGUSR1**: user defined signal 1
- **SIGUSR2**: user defined signal 2

# signal()

- Signal Handler Registration
- **signal(int signo, void(*func)()))**
  - specify the action for a signal (*signo* → *func*)
  - *func*
    - SIG_IGN (ignore)
    - SIG_DFL (default)
    - user-defined function
  - Return: the previous *func*

# Example



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

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

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


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

# Example : alarm2.c (w/ handler) (1/2)

```
#include <stdio.h> // alarm2.c
#include <signal.h>

int alarmFlag=0;
void alarmHandler();

main( ) {
    signal(SIGALRM, alarmHandler);
    alarm(3);
    printf("Looping …\n");
    while(!alarmFlag) {
        pause( );
    }
    printf("Loop ends due to alarm signal \n");
}

void alarmHandler( ) {
    printf("An alarm clock signal was received\n");
    alarmFlag = 1;
}
```

# Example : alarm2.c (w/ handler) (2/2)

- Execution

```
[esjung@hpclab opensource]$ gcc -o alarm2 alarm2.c
[esjung@hpclab opensource]$ ./alarm2
Looping …
An alarm clock signal was received
Loop ends due to alarm signal
[esjung@hpclab opensource]$
```

# SIGCHLD

- Whenever a process terminates or stops, the signal is sent to the parent.

- When a child process is killed, it sends SGICHILD signal to its parent process

# Example: timelimit.c (1/3)

**`$ timelimit N command`** // perform "command" within N seconds

```
#include <stdio.h> // timelimit.c
#include <signal.h>

int delay;
void childHandler( );

main(int argc, char *argv[])
{
    int pid;

    sscanf(argv[1], "%d", &delay);
    signal(SIGCHLD,childHandler);

    pid = fork();
    if (pid == 0) { // child
        execvp(argv[2], &argv[2]);
        perror("Limit");
    } else {        // parent
        sleep(delay);
        printf("Child %d exceeded limit and is being killed\n", pid);
        kill(pid, SIGINT);
   }
}
```

# Example: timelimit.c (2/3)

```
childHandler( ) /* Executed if the child dies before the parent */
{
    int childPid, childStatus;
    childPid = wait(&childStatus);
    printf("Child %d terminated within %d seconds\n", childPid, delay);
    exit(0);
}
```

# Example : timelimit.c (3/3)

- Execution

```
[esjung@hpclab opensource]$ ./timelimit 5 ls
7Process  alarm2.c  doatexit.c  lab3     pipe2.c  rdfile    timelimit    wtfile
alarm2    a.out     lab2        midterm  pipe.c   shfile.c  timilimit.c
Child 170561 terminated within 5 seconds
[esjung@hpclab opensource]$ ./timelimit 5 sleep 3
Child 170563 terminated within 5 seconds
[esjung@hpclab opensource]$ ./timelimit 5 sleep 100
Child 170573 exceeded limit and is being killed
[esjung@hpclab opensource]$
```

# 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.
  - 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 early System V UNIX

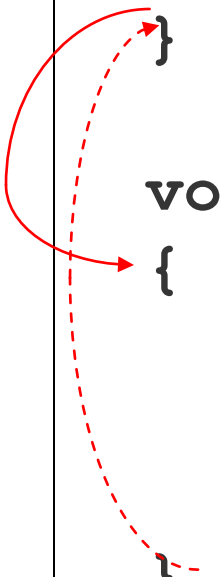
- In Linux (and many other UNIXs), the signal disposition in a process is reset to its default action immediately after the signal has been delivered.

- 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

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

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

**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.

- POSIX, BSD signal functions solve it (and some other later UNIXs)

# Modification in BSD 4.x signal environment

- Persistent Handlers
  - Signal handlers remain installed even after the signal occurs and do not need to be explicitly reinstalled.
- Masking
  - Signals are blocked for the duration of a signal handler (*i.e.* recursive signals are not normally allowed).
  - A "signal mask" can be set to block most signals during critical regions.
- Signal handlers normally remain installed during and after signal delivery.

# kill(), raise()

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
                        Both return: 0 if OK, 1 on error
```

- **kill** - sends a signal to a process or a group of process

- **raise** - function allows a process to send a signal to itself

# kill()

- pid means
  - pid > 0 : signal to the process whose process ID is pid
  - pid == 0 : signal to the processes whose process group ID equals that of sender
  - *pid < 0 : signal to the processes whose process group ID equals abs. of pid*
  - *pid == -1 : unspecified (used as a broadcast signal in SVR4, 4.3 + BSD)*

- Permission to send signals
  - The super-user can send a signal to any process.
  - The real or effective user ID of the sender has to equal the real or effective user ID of the receiver.

# alarm()

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
     Returns: 0 or number of seconds until previously set alarm
```

- alarm() sets a timer to expire at a specified time in future.
  - when timer expires, SIGALRM signal is generated,
  - default action of the signal is to terminate the process.
- Only one alarm clock per process
  - previously registered alarm clock is replaced by the new value.
- if alarm(0), a previous unexpired alarm is cancelled.

# pause()

```
#include <unistd.h>

int pause (void);
                                Returns: -1 with errno set to EINTR
```

- suspends the calling process until a signal is caught.
- returns only if a signal handler is executed and that handler returns.
  - If signal handler is not registered, just quit
  - If signal handler is registered, return after the handler is processed.

# abort()

```
#include <stdlib.h>

void abort(void);
            This function never returns
```

- Causes abnormal program termination.
- This function sends the SIGABRT signal to the process.
- SIGABRT signal handler to perform any cleanup that it wants to do, before the process terminated.

# sleep()

```
#include <signal.h>

unsigned int sleep(unsigned int seconds) ;
                          Returns: 0 or number of unslept seconds
```

- This function causes the calling process to be suspended until either
  - The amount of wall clock time specified by second has elapsed (returns 0)
  - A signal is caught by the process and the signal handler returns (returns the number of unslept seconds)

# Process group

- Group of related processes that should receive a common signal for certain events
- Identified by Process group ID
- Example
  - grp = setpgrp();
    - Initialize process group id, the same as its pid
    - After fork(), child process will have parent's process group id

# setpgrp()

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

main(){
    register int i;
    setpgrp();
    for(i=0;i<10;i++){
        if(fork()==0){
            /* child process */
            if(i&1)
                setpgrp();
            printf("pid = %d pgrp= %d\n", getpid(),getpgrp());
            pause();
        }
    }
    kill(0,SIGINT);
}
```

# Results

```
[esjung@hpclab opensource]$ ./a.out
pid = 100672 pgrp= 100671
pid = 100673 pgrp= 100673
pid = 100674 pgrp= 100671
pid = 100676 pgrp= 100671
pid = 100675 pgrp= 100675
pid = 100677 pgrp= 100677
pid = 100678 pgrp= 100671
pid = 100679 pgrp= 100679
pid = 100680 pgrp= 100671
```

# Treatment of Signals

- Sending a signal to a process
  - issig() : check for receipt of signals
- Handling a signal
  - psig() : handle signals after recognizing their existence

# How to Send a Signal to Process

- Send a Signal
  - sets a bit in the signal field in process table entry.
  - process can remember different type of signal.
  - process can not remember how many signals it receives of particular type.
  - u area contains an array of signal-handler fields.

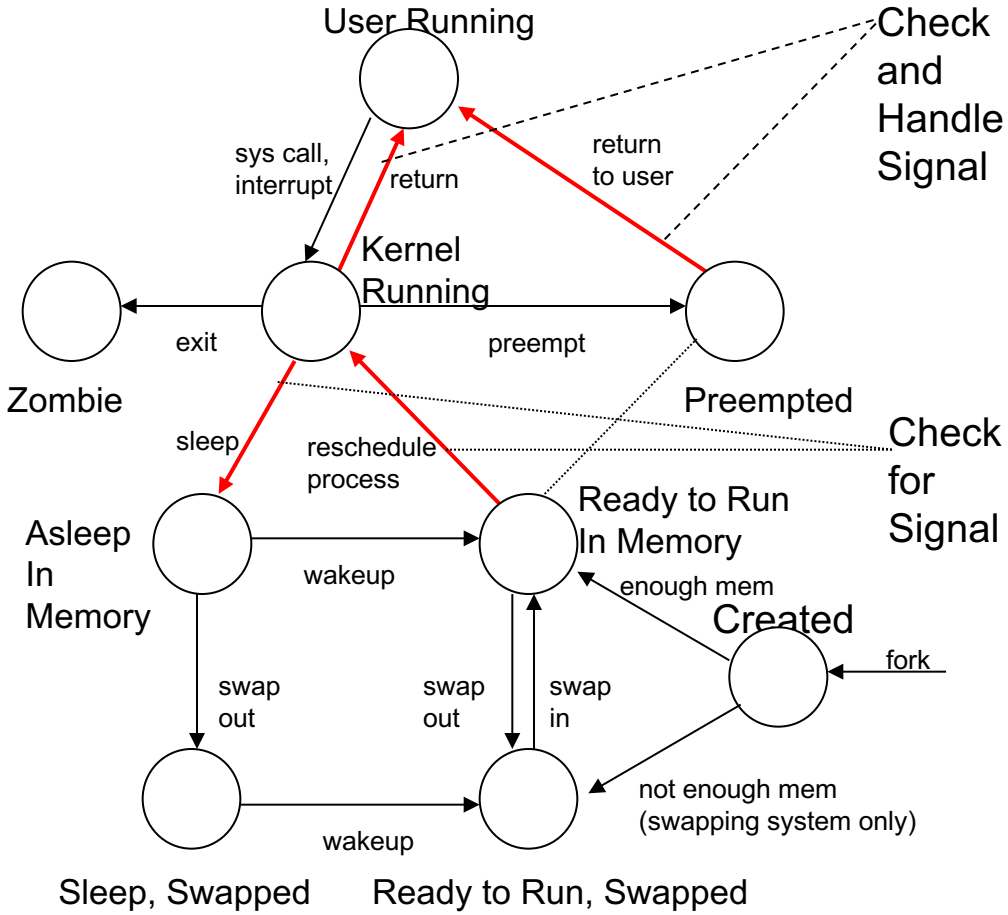
    kernel stores the address of the user-function in the field.

- Check for Signal when
  - about to return from kernel mode to user mode
  - enters or leaves the sleep state at a suitably low scheduling priority.

- Handle Signals
  - only when returns from kernel mode to user mode

# How to Send a Signal to Process(Cont.)

- no effect on a process running in the kernel mode
- a process never executes in user mode before handling outstanding signals



# Algorithm for issig

```
algorithm issig          /* test for receipt of signals */
input : none
output : true, if process receives signals that it does not ignore
          false otherwise
{
   while(received signal field in process table entry not 0)
   {
        find a signal number sent to the process;
        if ( signal is death of child )
        {
                if (ignoring death of child signals)
                        free process table entries of zombie children;
                else if (catching death of child signals)
                        return(true);
        }
        else if (not ignoring signal)
                return(true);
        turn off signal bit in received signal field in process table;
   }
   return(false);
}
```

# Algorithm for Handling Signal

1. determines signal type
2. turns off signal bit in the process table entry
3. if receives a signal to ignore
   - continues as if the signal has never occurred.
4. If signal handling function is set to its default value,
   - kernel dumps core image for signals that imply something is wrong with process and exit
   - kernel does not dump core for signals that do not imply error.
5. If receives a signal to catch,
   - accesses the user saved register context, and find the program counter and stack pointer
   - clears the signal handler field in u area( *undesirable side-effects* )
   - kernel creates a new stack frame and writes a program counter and stack pointer from user saved register context
   - kernel changes the user register context: program counter to the address of signal catcher function, and stack pointer to account for the growth of the user stack.

# Algorithm for psig

```
algorithm psig          /* handle signals after recognizing their
   existence */
input : none
output : none
{
   get signal number set in process table entry;
   clear signal number in process table entry;
   if (user had called signal sys call to ignore this signal)
        return;          /*done*/
   if (user specified function to handle this signal)
   {
        get user virtual address of signal catcher stored in u area;
        /*the next statement has undesirable side-effects*/
        clear u area entry that stored address of signal catcher;
        modify user level context;
                artificially create user stack frame to mimic
                call to signal catcher function;
        modify system level context;
                write address of signal catcher into program
                counter filed of user saved register context;
        return;
   }
```

# Algorithm for psig (cont.)

```
    if (signal is type that system should dump core image of
    process)
    {
          create file named "core" in current directory;
          write contents of user level context to file "core";
    }
    invoke exit algorithm immediately;
}
```