Process

Process

- The process is one of the fundamental abstraction in Linux operating system
- Running instance of a program is called process
- *“An address space with one or more threads executing within that address space, and the required system resources for those threads”* – IEEE Std 1003.1. 2004 Edition

Process Hierarchy / Family Tree

- All processes are descendants of the init process, whose PID is one.
- The kernel starts init in the last step of the boot process. The init process, in turn, reads the system *initscripts* and executes more programs, eventually completing the boot process.
- Every process on the system has exactly one parent. Likewise, every process has zero or more children. Processes that are all direct children of the same parent are called *siblings*.

Process Identification

- The system identifies processes by a unique process identification value or PID
- PID is represented by an opaque type pid_t and is of type int
- Default maximum value is only 32,768 (short int type)

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

pid_t getpid (void);

pid_t getppid (void);
```

Process Creation – fork()

- fork() creates a child process that is a copy of the current task
- Differs from the parent in its PID and PPID * (check the man page)
- The new process inherits several characteristics of the old process. Among the characteristics inherited are:
  - The environment.
  - All signal settings.
  - The set user ID and set group ID status.
  - The time left until an alarm clock signal.
  - The current working directory and the root directory.
  - The file creation mask as established with umask().

- fork() returns zero in the child process and non-zero (the child's process ID) in the parent process.

File sharing between parent and child

- File descriptors that are open in the parent are duplicated in the child is an important characteristic of fork().
- The parent and the child process share the same file descriptor table.
- By default, process opens three different files opened for standard input, standard output, and standard error. When a command is executed as a process, they are inherited

Fork ( ) System Call

- A new process running the same image as the current one can be created via the fork( ) system call:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

Copy-on-write

- Copy-on-write is a lazy optimization strategy designed to mitigate the overhead of duplicating resources.
- After forking both parent and child share parent's original pages.
- Copy occurs only on write.
- Since many of the forks are followed by an exec, copying the parent's address space into the child's address space is often a complete waste of time.

wait ( ) System Call

- You can control the execution of child processes by calling wait() in the parent.
- wait() forces the parent to suspend execution until the child is finished.
- wait() returns the process ID of a child process that finished.
- If the child finishes before the parent gets around to calling wait(), then when wait() is called by the parent, it will return immediately with the child's process ID.

wait ( ) System Call

- The prototype for the wait() system call is:

  int wait(status)

  int *status;

- “status” is a pointer to an integer where the LINUX system stores the value returned by the child process. wait() returns the process ID of the process that ended.

waitpid System Call

- The **waitpid**() system call suspends execution of the calling process until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below
- Its prototype is

  pid_t waitpid (pid_t pid, int *stat_loc, int options);

  status is in stat_loc, where as options argument allows us to modify the behavior of waitpid
- The value of *pid* can be:
  - < -1 : wait for any child process whose process group ID is equal to the absolute value of *pid*.
  - -1 : meaning wait for any child process.
  - 0: meaning wait for any child process whose process group ID is equal to that of the calling process.
  - > 0 : meaning wait for the child whose process ID is equal to the value of *pid*.

Process Termination

- Explicitly by calling exit ( ) system call
- Implicitly by returning from main function of a process
- Involuntary termination
  - Signal or exception is neither handled nor ignored
  - Due to raising of exception during its kernel mode execution
  - When it receives the SIGABRT or other termination signal
- A process can terminate before its parent or after its parent
- Process termination is handled by do_exit( )

Exit system call

- exit() system call ends a process and returns a value to it parent.
- The prototype for the exit() system call is:

  void exit(status)

  int status;
- status is an integer between 0 and 255. This number is returned to the parent via wait() as the exit status of the process.
- By convention, when a process exits with a status of zero that means it didn't encounter any problems; when a process exit with a non-zero status that means it did have problems.

Zombie Process

- When the child process terminates an association with its parent survives until the parent in turn either terminates normally or calls wait.
- The child process entry is still in the system, because is exit code needs to be stored in case the parent subsequently calls wait. This way child becomes defunct or a zombie process.

exec System Calls

- Unlike the other system calls and subroutines,

  a successful exec system call does not return.

- Instead, control is given to the executable binary file named as the first argument.

- When that file is made into a process, that process replaces the process that executed the exec system call -- a new process is not created.

exec System Calls

The LINUX system calls that transform a executable binary file into a process are the "exec“ family of system calls. The prototypes for these calls are:

- int execl(file_name, arg0 [, arg1, ..., argn], NULL)
  char *file_name, *arg0, *arg1, ..., *argn;
- int execv(file_name, argv)
  char *file_name, *argv[];
- int execle(file_name, arg0 [, arg1, ..., argn], NULL, envp)
  char *file_name, *arg0, *arg1, ..., *argn, *envp[];
- int execve(file_name, argv, envp)
  char *file_name, *argv[], *envp[];
- int execlp(file_name, arg0 [, arg1, ..., argn], NULL)
  char *file_name, *arg0, *arg1, ..., *argn;
- int execvp(file_name, argv)
  char *file_name, *argv[];

- Letters added to the end of exec indicate the type of arguments:
  - l argn is specified as a list of arguments.
  - v argv is specified as a vector (array of character pointers).
  - e environment is specified as an array of character pointers.
  - p user's PATH is searched for command, and command can be a shell program

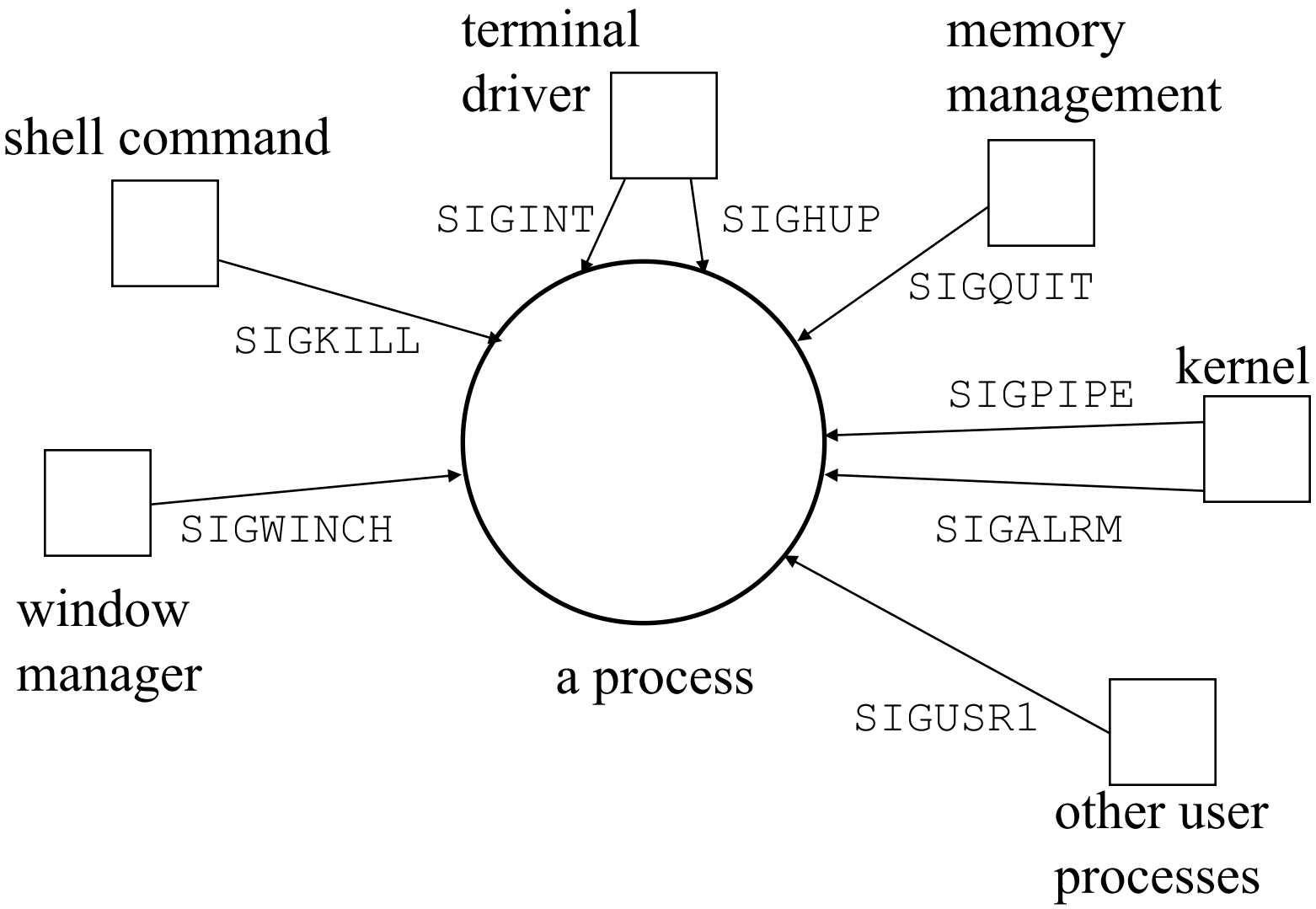
Thank you

Signals

Signal

- Asynchronous event
  - These can happen a litter later after the event is generated.

Signal Sources



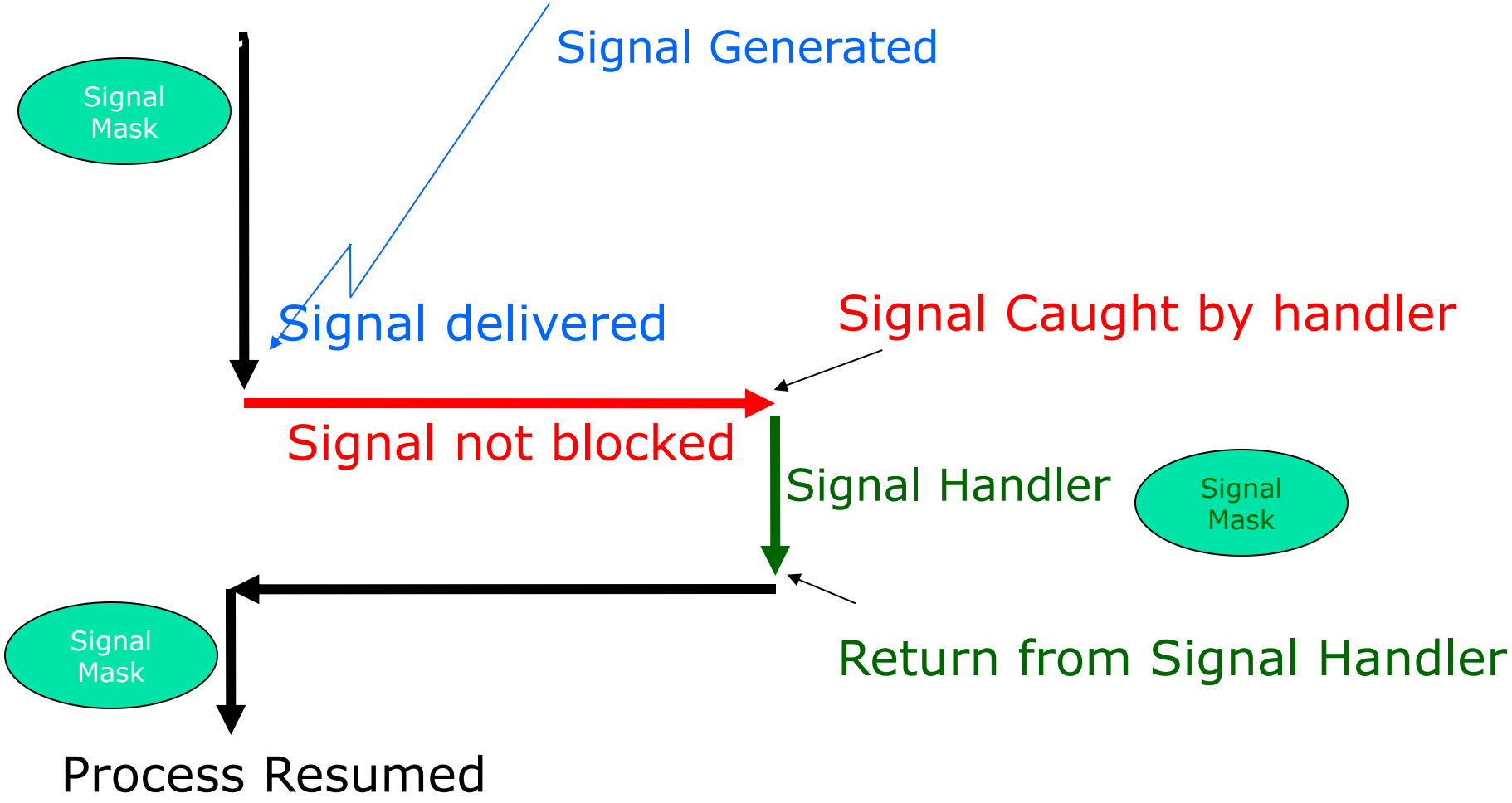
Signal sources – outline

- Signals can be sent to process by
  - another process
    - Using ***Kill*** system call
  - Kernel

Signal generation & reception

- How a process receives a signal
  - Kernel will set the respective bit in process entry table
- When process will handle a signal?
  - When process returns from kernel mode to user mode

How Signals Work



List of signals

- The command ‘kill –l’ lists all the signals that are available.

```
venu@venuGopal:~$ 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
venu@venuGopal:~$
```

Signals

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

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;
$
```

Sending a signal : ctrl+c

- A program that prints “Hello world” every sec.
- ctrl+c is the signal SIGINT number 2 in the list is sent, to kill a program through keyboard

```
#include<stdio.h>
int main()
{
        while(1)
        {
                printf("Hello world\n") ;
                sleep(1) ;
        }
        return 0 ;
}
```

Output:

```
root@boss[jyo]#./first
Hello world
Hello world
Hello world
Hello world

root@boss[jyo]#
```

Sending a signal through a process : ctrl+c

- Consider a program with parent and child process.
- Child will be in a while one loop printing “Hello world” after every sec
- Parent process sends the signal using **‘kill’** system call.
- Syntax:
  - **int kill(pid_t** *pid*, **int** *sig*);
  - Example:
    - Kill(6358, SIGINT) ;

Kill - syntax

- UNIX command:

  $ kill <signal name> <pid>

  $ kill SIGKILL 4481

  send a SIGKILL signal to pid 4481

- System call

Return 0 if ok,
-1 on error

```
#include <signal.h>
int kill( pid_t pid, int signo );
```

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

```
#include<stdio.h>
#include<sys/types.h>
int main()
{
    pid_t cpid ;
    int status ;

    cpid = fork() ;
    if(cpid == 0)    // child process
    {
        printf("Child process pid is %d\n",
    getpid()) ;
        while(1)  {
                printf("Hello world\n") ;
                sleep(1) ;
        }
    }
    else  //parent process
    {
        printf("Parent process pid is %d\n",
    getpid()) ;
        system("ps -al") ;
        printf("Sending SIGINT to child
    process....\n") ;
        kill(cpid, SIGINT) ;
        wait(&status) ;
        system("ps -al") ;
    }

    return 0 ;
}
```

```
root@boss[jyo]#./second
Child process pid is 6383
Hello world
Parent process pid is 6382
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R     0  6382  5954  0  76   0 -   391 -      pts/4    00:00:00 second
1 S     0  6383  6382  0  76   0 -   391 -      pts/4    00:00:00 second
0 R     0  6384  6382  0  76   0 -  1055 -      pts/4    00:00:00 ps
Sending SIGINT to child process....
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R     0  6382  5954  0  77   0 -   391 -      pts/4    00:00:00 second
0 R     0  6385  6382  0  77   0 -  1056 -      pts/4    00:00:00 ps
root@boss[jyo]#
```

Parent process

Child process

Parent process

Signal Handling

- Kernel handles the signals in context of a process
- Three ways in handling the signals
  - Ignores the signal
  - Execute the default action (exit is the default action for most of the signals)
  - Execute a user defined signal handler.

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***

Using signal call

- Signal call
  - To register a user defined signal handler
  - To ignore a signal
  - To handle the default behaviour
  - Syntax:

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

- Three ways to handle a signal
  - Signal(*signal_number,* SIG_IGN*)*
  - *Signal(signal_number*, SIG_DFL)
  - Signal(*signal_number*, signal_handler)

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.

Func	Action
SIG_DFL	Default signal handler
SIG_IGN	Ignore the signal
Function pointer	Registered signal handler

Signals can't be trapped

Cannot ignore/handle `SIGKILL` or `SIGSTOP`

signal(): library call

- Specify a signal handler function to deal with a signal type.

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

- Returns *previous* signal disposition if ok, SIG_ERR on error.

Actual Prototype

- The actual prototype, listed in the “man” page is a bit perplexing but is an expansion of the Sigfunc type:

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

- In Linux:

  typedef void (*sighandler_t)(int);

  sig_handler_t signal(int signo, sighandler_t handler);

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

Signal registering and handling - part - I

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

void signal_handler(int signum)
{
        printf("Caught the signal %d\n",signum) ;
}
int main()
{
        (void) signal(SIGINT, signal_handler) ;

        while(1)
        {
                printf("Hello world\n") ;
                sleep(1) ;
        }
        return 0 ;
}
```

Signal handler

Registering the signal handler for SIGINT

```
root@boss[jyo]#./third
Hello world
Hello world
Hello world
Hello world
I caught the signal %2
Hello world
Hello world
Hello world
I caught the signal %2
Hello world
Hello world
Hello world

[1]+  Stopped                 ./third
root@boss[jyo]#
```

Signal registering and handling – part - II

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

void signal_handler(int signum)
{
        printf("Caught the signal %d\n",signum) ;
        signal(SIGINT,SIG_DFL) ;
}
int main()
{
        (void) signal(SIGINT, signal_handler) ;

        while(1)
        {
                printf("Hello world\n") ;
                sleep(1) ;
        }
        return 0 ;
}
```

Signal handler

Reverting to the default behavior

Registering the signal handler for SIGINT

```
root@boss[jyo]#./fourth
Hello world
Hello world
Hello world
Hello world
I caught the signal %2
Hello world
Hello world

root@boss[jyo]#
```

Signal registering and handling: Sending the signal to itself

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

void signal_handler(int signum)
{
        printf("Caught the signal %d\n",signum) ;
        signal(SIGINT,SIG_DFL) ;
        printf("Sending the SIGINT to myself...\n") ;
        kill(getpid(), SIGINT) ;
}
int main()
{
        (void) signal(SIGINT, signal_handler) ;
        while(1)
        {
                printf("Hello world\n") ;
                sleep(1) ;
        }
        return 0 ;
}
```

Signal handler

Reverting to the default behavior

Sending the signal to itself

Registering the signal handler for SIGINT

```
root@boss[jyo]#./fifth
Hello world
Hello world
Hello world
Hello world
I caught the signal %2
Sending the SIGINT to myself...

root@boss[jyo]#
```

Signal registering and handling : Through a process

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

void signal_handler(int signum)
{
        printf("Caught the signal
%d\n",signum) ;
        signal(SIGINT,SIG_DFL) ;
}
```

```
cdachyd@cdachyd:~/Desktop$ ./sixth
Hello world
Caught the signal 2
Hello world
Hello world
Hello world
Hello world
Hello world
cdachyd@cdachyd:~/Desktop$
```

```
int main()
{
    (void) signal(SIGINT, signal_handler) ;
    pid_t pid ;
    pid = fork() ;
    if(pid == 0)          // child process
    {
        while(1)
        {
            printf("Hello world\n") ;
             sleep(1) ;
        }
    }
    else                  // parent process
    {
        kill(pid,SIGINT) ;
        sleep(5) ;
        kill(pid,SIGINT) ;
    }
        return 0 ;
}
```