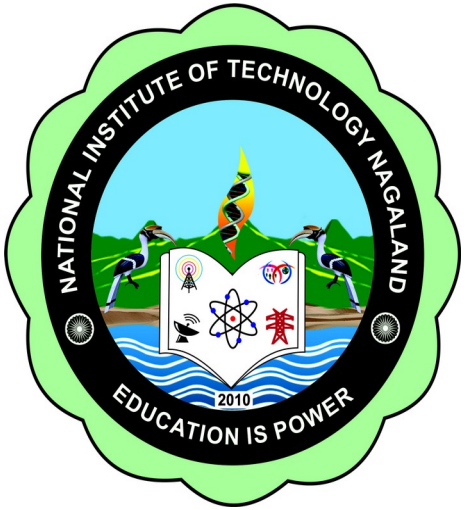



# UNIX System Programming

## Signals & Resource Limits

Module 3

# Agenda

1. Signals
2. Resource Limits

# Objectives

At the end of this module you will be able to:

- understand the purpose of signals
- handle signals in applications
- understand POSIX signal handling
- know resource limits defined by UNIX
- alter resource limits
- test the impact of resource limits on processes

# Signals

# Purpose of Signals

- Signal is a mechanism to indicate occurrence of an event to a process
- Signal is a software interrupt or notification.
- Signals help to manage asynchronous events, such as
  - Occurrence of an event (timer expires, alarm, etc.,)
  - a user quota exceeds (file size, no of processes etc.,)
  - an I/O device is ready
  - encountering an illegal instruction
  - a terminal interrupt like Ctrl-C or Ctrl-Z.
  - some other process send ( kill -9 pid)
- Each signal has a unique number and represented by macros, which are defined in **`<signal.h>`**.

# Signals

| Signal | Value | Action* | Comment |
|---|---|---|---|
| SIGHUP | 1 | Term | Hang up |
| SIGINT | 2 | Term | Interrupt from keyboard (Ctrl + C) |
| SIGQUIT | 3 | Core | Quit from keyboard (Ctrl + \) |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort(3) |
| SIGKILL* | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm(2) |
| SIGTERM | 15 | Term | Termination signal |
| SIGCHLD | 17 | Ignore | Child stopped or terminated |
| SIGSTOP* | 19 | Term | Stop process |

* Comments in the next slide

# Signals (Contd.).

- The default action for a signal can be one of the following, depending on the signal (refer to the previous slide)
  - Term  application terminates
  - Core  application terminates with core dump
  - Ignore  application ignores the signal by default
- Note that **`SIGSTOP`** & **`SIGKILL`** can not be caught or ignored

# Signals (Contd.).

- A process can receive signals from different sources
  - **from operating system**
    - Due to hardware exceptions which trigger signals (e.g., division with zero, invalid memory reference)
  - **from another process**
    - When the other process makes kill() system call
  - **from itself**
    - alarm(), which triggers SIGALRM

- The difference between an exception and a signal is that exceptions are synchronous whereas signals are asynchronous.
  - Some exceptions trigger signals. (e.g., floating point exception generates SIGFPE)

# Sending Signal to a Process

- One process can send signal to another process using **`kill`** command.

```
$  kill  -s signumber  processid
```

or

```
$  kill  -signame  processid
```

```
$ sleep  1000&
[1] 17719
$ kill –s SIGKILL 17719
```

```
$ sleep 1000&
[1] 17814
$ kill –s 9 17814
```

**`$ kill  –TERM  2340`** request for termination of process with PID 2340

# Sending Signals from program

- One process can send signal to another using **`kill()`** system call.

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

sends **`signum`** signal to the **`pid`** specified

- One process can send signal to itself using **`raise()`**

```
#include <signal.h>
int raise(int signum);
```

sends **`signum`** signal to itself

# signal function

```
#include <signal.h>

typedef void (*sighandler_t)(int);

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

- Handler argument can be one of the following
  - **programmer defined handler**  to perform specific action
  - **SIG_IGN**  to ignore the signal
  - **SIG_DFL**  to perform default action (in most cases terminate process)

- returns previous value of the signal handler
- This can be saved, to restore to the old value if signal handler is to be restored

# signal handler

```
#include <signal.h>
void sighandler(int sig)
{
   printf("received signal = %d\n", sig);
}

int main (void)
{
   signal (SIGINT, sighandler);
   while(1)
   {
   }
}
```

Comment the **`signal()`** call, and run the program.
Try Ctrl+C and Ctrl+\ while the program is in execution.

Uncomment the **`signal()`** call, and run the program.
Try Ctrl+C and Ctrl+\ while the program is in execution.

What is the difference noticed?

# Inheriting signal handlers by child processes

```
#include <signal.h>
void sighandler(int sig)
{
    printf("pid %d received signal
             = %d\n", getpid(), sig);
}
int main (void)
{
    printf("parent pid = %d\n",
        getpid());
    signal (SIGINT, sighandler);
    if (fork() == 0)   {
       printf("child pid =%d\n",
            getpid());
    }
    while(1)
    {
    }
}
```

**Does a child process inherit signal handler?**

**Run the test program.**

# Handling SIGCHLD to Avoid Zombies

```
void sighandler(int sig)
{
    printf("pid %d received signal = %d\n", getpid(), sig);
}
int main (void)
{
    printf("pid = %d\n", getpid());
    signal (SIGCHLD, sighandler);
    if (fork() == 0) { printf("child pid =%d\n", getpid()); }
    while(1)
    {
    }
}
```

**NOTE:**
**Just catching of `SIGCHLD` does not ensure that all zombie processes are handled by the parent process.**

```
$ ./a.out&
32051
pid = 32051
SIGCHLD = 17
$ child pid =32052
$
$ kill -9 32052
$ pid 32051  received signal = 17
$
$ ps -S
  PID TTY      STAT   TIME COMMAND
32051 pts/1    R      0:20 ./a.out
32052 pts/1    Z      0:18 [a.out <defunct>]
$
```

# Handling SIGCHLD to Avoid Zombies (Contd.).

```
#include <signal.h>
void sighandler(int sig)
{
    int wstatus;
    printf("pid %d received signal
          = %d\n", getpid(), sig);
    wait(&wstatus);
}

int main (void)
{
    printf("pid = %d\n", getpid());
    signal (SIGCHLD, sighandler);
    fork();
    while(1)
    {
    }
}
```

```
$ ./a.out&
32803
pid = 32803
$
$ ps
32803 pts/1    00:00:06 a.out
32804 pts/1    00:00:06 a.out
$
$ kill -9 32804
$ pid 32803  received signal = 17
$
$ ps
32803 pts/1    00:00:06 a.out
$
```

# alarm ()  and pause()

- **`alarm()`** is used to set an alarm for delivering SIGALARM signal.

```
unsigned int alarm (unsigned int seconds);
```

  - On success  it returns the number of seconds remaining until previously set alarm due; zero if no alarm is scheduled.
  - **`alarm()`** is not a blocking call.
  - To cancel an existing alarm, pass value zero (0) as argument.

- **`pause()`** is a C Library function, which suspends calling process till a signal is caught and the signal handler is returned.

```
int pause (void);
```

  the function returns -1 and sets **`errno`** to **`EINTR`**

# alarm ()  and pause() – example

```
int main (void)
{
   printf("setting alarm(5)\n");
   alarm(5);
   printf("alarm set\n");
   printf("process paused\n");
   pause();
   return 0;
}
```

Note that signal handler is not defined.

Program output

```
$ ./a.out
setting alarm(5)
alarm set
process paused
Alarm clock
$
```

# Alarm with Signal Handler

```
#include <signal.h>
#define ALARM_TIME    5
int sec = 0;
void sighandler(int sig)
{
    printf("received signal =
         %d\n", sig);
    sec += ALARM_TIME;
    printf("%d seconds elapsed\n",
              sec);
    alarm(ALARM_TIME);
}
```

```
int main (void)
{
   signal (SIGALRM,
                sighandler);
    printf("setting alarm\n");
    alarm(ALARM_TIME);
    printf("alarm set\n");
    while(1)
    {
        pause();
    }
}
```

# Alarm with Signal Handler (Contd.).

**Program output with signal handler defined**

```
$ ./a.out
setting alarm
alarm set
received signal = 14
5 seconds elapsed
received signal = 14
10 seconds elapsed
received signal = 14
15 seconds elapsed
received signal = 14
20 seconds elapsed
...
```

**Output continues infinitely as shown above**

# POSIX Style Signal Handling

- POSIX signal handling has some advantages
  - Reliable signal handling
  - Large number of signals can be handled
  - Allows signals to be blocked and unblocked
  - Possible to define signal sets

- Signal handler is defined using **`sigaction()`**

- Why to prefer **`sigaction()`**
  - depending on the variant of Unix, semantics of **`signal()`** can be C library function or System V or BSD, resulting in portability issues.

# sigaction

```
#include <signal.h>
int sigaction( int signum,
                const struct sigaction *act,
               struct sigaction *oldact);
```

- **signum**, is a specified signal
- **act** is used to set the new action of the signal signum;
- **oldact** is used to store the previous action, usually NULL.

# Handling Signal – POSIX way

```
#include <signal.h>
void sighandler(int sig)
{
    printf("received signal = %d\n", sig);
}

int main (void)
{
    struct sigaction action = { sighandler };
    sigaction(SIGINT, &action, NULL);
   while(1)
   {
   }
}
```

# Resource Limits

# Resource Limits

- The OS imposes limits for certain system resources it can use.
- Applicable to a specific process.
- The “**`ulimit`**” shell built-in can be used to set/query the status.
- “**`ulimit -a`**” returns the user limit values

# Resource Limits (Contd.).

- The command **`ulimit`** takes the following options

  -f  Maximum size of the files created.
  -l  Maximum amount of memory that can be locked using **`mlock()`** system call.
  -n  Maximum number of open file descriptors.
  -s  Maximum stack size allowed per process.
  -u  Maximum number of processes available to a single user.

# Resource Limits (Contd.).

- Resource limits are of two types
  - Hard limit
    - Absolute limit for a particular resource. It can be a fixed value or “unlimited”
    - Only super user can set hard limit.
    - Hard limit once set, cannot be increased.
  - Soft limit
    - User-definable parameter for a particular resource.
    - Can have a value from 0 till <hard limit> value.
    - Any user can set soft limit.
- Limits are inherited (the new values are applicable to the descendent processes).

# getrlimit/setrlimit

- **`getrlimit()/setrlimit()`** lets an application get/set resource limits

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit(int res, struct rlimit *reslimit);
int setrlimit(int res, const struct rlimit * reslimit);

struct rlimit {
  rlim_t rlim_cur;   /* Soft limit */
  rlim_t rlim_max;   /* Hard limit */
};
```

Note: soft limit value must be less than hard limit value

# getrlimit/setrlimit (Contd.).

Resource that can be set with limits

- RLIMIT_FSIZE  Maximum size of the file.
- RLIMIT_MEMLOCK  Maximum amount of memory that can be locked.
- RLIMIT_NOFILE  Maximum number of open file descriptors.
- RLIMIT_STACK  Maximum stack size allowed.
- RLIMIT_NPROC  Maximum number of process available to a single user.

# setrlimit() - example

```
#include <fcntl.h>
#include <sys/resource.h>
main(int argc, char *argv[])
{
  int fd, k;
  char *buf = "hello ";
  struct rlimit rlim = {30,100 };
  setrlimit(RLIMIT_FSIZE, &rlim);
  fd = open(argv[1], O_CREAT | O_TRUNC | O_WRONLY, 0600);
  for (k=0; k<20; k++)   {
      printf("attempting to write  %d bytes\n",
  strlen(buf));
      write(fd, buf, strlen(buf));
      printf("successfully written %d bytes\n",
  strlen(buf));
  }
  close(fd);
}
```

# setrlimit() – example (Contd.).

Run the previous program and observe that the program terminates after writing 30 bytes.

```
$ ./a.out out
attempting to write  6 bytes
successfully written 6 bytes
attempting to write  6 bytes
successfully written 6 bytes
attempting to write  6 bytes
successfully written 6 bytes
attempting to write  6 bytes
successfully written 6 bytes
attempting to write  6 bytes
successfully written 6 bytes
attempting to write  6 bytes
File size limit exceeded
$
```

# Hands-on and Assignments

# Exercises

1) Define a signal handler that catches SIGINT, SIGTERM and SIGQUIT, prints the signal it has received.

   Test the program that it functions as desired.

2) Write a program which meets the following requirements.
   a) Parent process registers a signal handler for SIGINT, which prints process ID and the signal which the handler received.
   b) Parent process creates two child processes both run in infinite loop.
   c) Parent process sends SIGINT to both child processes
   d) Parent process sends SIGINT to itself

   Run the program and observe the behaviour.

   Note: Use POSIX sigaction to register signal handler.

# Exercises (Contd.).

3) Write a program to check if resource limits set by a process are inherited by its child process.

4) Write a program to limit the number of files that can be opened by a process to not more than 10 files and test if that works. Can this be increased subsequently to 15 and open 5 more files?

# Summary

In this module, we discussed:

- the purpose of signals
- how to program signals
- POSIX signal handling
- various resource limits in UNIX
- how to alter resource limits
- impact of resource limits on processes

# Review Questions

a) Is signal handling synchronous or asynchronous?

b) For which signals, signal handler can not be defined?

c) What are the advantages of POSIX signal handling?

d) What should be done to ensure that a terminated process does not end up as a zombie?

e) For each of the following identify whether it is a blocking call or not.

i) pause ii) alarm

f) Does a process inherit signal handlers defined by its parent?

g) What is the significance of resource limits?

h) Does a process get terminated if it exceeds resource limit?

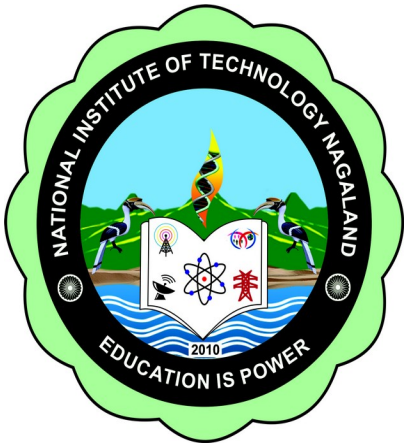
# References


1) W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment. Ed 2, US: Addison-Wesley, 2009.

2) Kay A. Robbins and Steven Robbins, UNIX Systems Programming, New Delhi: Pearson Education, 2009.

3) Rochkind, Advanced Unix Programming, Ed 2. US: Addison-Wesley, 2004.

4) Arnold Robbins, Linux Programming by Example, New Delhi: Prentice Hall, 2008.


NATIONAL INSTITUTE OF TECHNOLOGY NAGALAND
2010
EDUCATION IS POWER

# Thank You