

Unix Programming

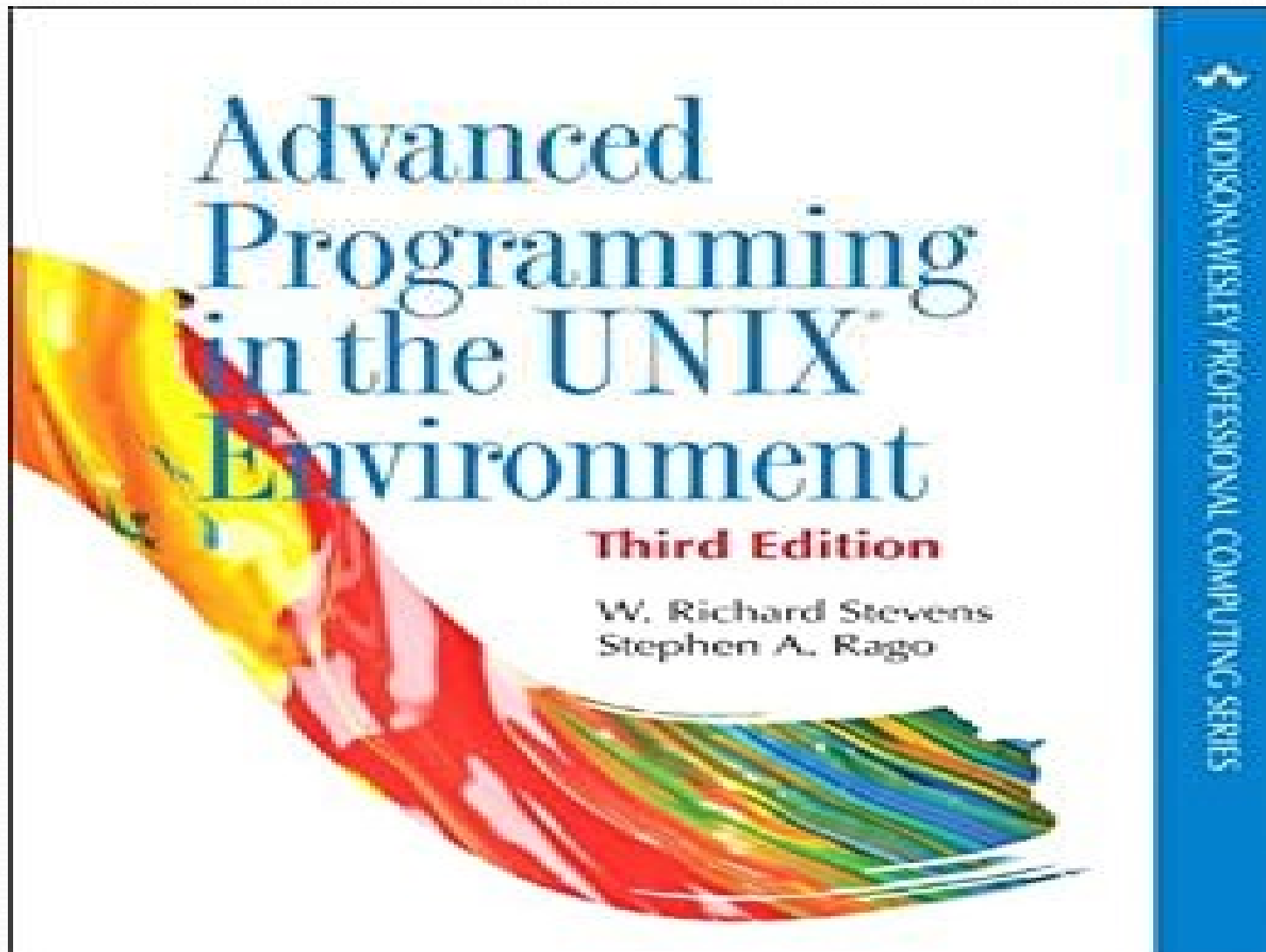
UNIT-VI

B.Tech(CSE)-V SEM

UNIT : VI Signals :

- Signal functions
- unreliable signals
- interrupted system calls
- kill and raise functions
- alarm, pause functions
- abort, sleep functions

Textbook:






Introduction:

- Signals are **software interrupts**.
- Signals provide a way of handling **asynchronous events**.
- **For eg:**
 - a user at a terminal typing the **interrupt** key to stop the program
 - the next program in a pipeline terminating prematurely.

Signal concepts:

- Every signal has a name. They all begin with the three characters **SIG**.
- For example:
 - **SIGABRT** is the abort signal that is generated when a process calls the abort function.
 - **SIGALRM** is the alarm signal that is generated when the timer set by the alarm function goes off.
- Signal names are all defined by **positive integer constants (the signal number)** in the header `<signal.h>`.
- No signal has a signal number of **0**. The **kill** function uses the signal number of **0** for a special case. POSIX.1 calls this value the **null signal**.



Conditions (situations) that generate signals:

- Numerous conditions can generate a signal:
 - The terminal-generated signals occur when users press certain terminal keys. Pressing the DELETE key or Control-C.
 - Hardware exceptions generate signals. For example, divide by 0 and invalid memory reference.
 - The `kill(2)` function allows a process to send any signal to another process or process group, with limitations: we have to be the owner of the process that we're sending the signal to, or we have to be the super user.
 - The `kill(1)` command allows us to send signals to other processes. This command is often used to terminate a runaway background process.



Conditions that generate signals:

- Software conditions can generate signals when a process should be made aware of various events.
- For example:
 - **SIGURG**: generated when out-of-band data arrives over a network connection).
 - **SIGPIPE**: generated when a process writes to a pipe that has no reader).
 - **SIGALRM**: generated when an alarm clock set by the process expires).

Signal dispositions:

- We can tell the kernel to do one of three things when a signal occurs.
- This is called the **disposition of the signal**, or the **action associated with a signal**.
 - **Ignore the signal**. Most signals can be ignored, but two signals can never be ignored: **SIGKILL** and **SIGSTOP**.
 - **Catch the signal**. To do this, we tell the kernel to call a function of ours whenever the signal occurs. In our function, we can do whatever we want to handle the condition.
 - **Let the default action apply**. Every signal has a default action. The default action for most signals is to terminate the process.

Signal function:

- The simplest interface to the signal features of the UNIX System is the signal function.

```
#include <signal.h>
void (*signal(int signo, void (*func)(int)))(int);
/* Returns: previous disposition of signal if OK, SIG_ERR on error */
```

- **Arguments:**

- The ***signo*** argument is the name of the signal.
- The value of ***func*** one of the following:
 - the constant **SIG_IGN**, which tells the system ignore the signal;
 - the constant **SIG_DFL**, which sets the action associated with the signal to its default value;
 - the address of a function to be called when the signal occurs, which arranges to "catch" the signal. This function is called either the **signal handler** or the **signal-catching function**.

Unreliable signals:

- In earlier versions of the UNIX System, signals were unreliable, which means that signals could get lost: a signal could occur and the process would never know about it.
- One problem with these early versions was that the action for a signal was reset to its default each time the signal occurred.
- The code that was described usually looked like:

```
int sig_int(); /* my signal handling function */
...
    signal(SIGINT, sig_int); /* establish handler */
...
sig_int()
{
    signal(SIGINT, sig_int); /* reestablish handler for next time */
    ...
    /* process the signal ... */
}
```



Unreliable signals:

- The problem with this code fragment is that there is a window of time (after the signal has occurred, but before the call to signal in the signal handler) when the interrupt signal could occur another time.
- This second signal would cause the default action to occur, which terminates the process. This is one of those conditions that works correctly most of the time, causing us to think that it is correct, when it isn't.
- Another problem with these earlier systems was that the process was unable to turn a signal off when it didn't want the signal to occur. All the process could do was ignore the signal.

Interrupted System Calls:

- In earlier UNIX systems, if a process caught a signal while the process was blocked in a "slow" system call, the system call was interrupted.
- The system call returned an error and `errno` was set to `EINTR`.
- This was done under the assumption that since a signal occurred and the process caught it, there is a good chance that something has happened that should **wake up the blocked system call**.
- We have to differentiate between a system call and a function. It is a system call within the kernel that is interrupted when a signal is caught.

Interrupted System Calls:

- The system calls are divided into two categories:
 - the "slow" system calls and
 - all the others.
- The slow system calls are those that can block forever.
- Included in this category are:
 - Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
 - Writes that can block the caller forever if the data can't be accepted immediately by these same file types
 - Opens that block until some condition occurs on certain file types (such as an open of a terminal device that waits until an attached modem answers the phone)
 - The pause function (which by definition puts the calling process to sleep until a signal is caught) and the wait function
 - Certain ioctl operations
 - Some of the interprocess communication functions

Interrupted System Calls:

- The notable exception to these slow system calls is anything related to disk I/O.
- One condition that is handled by interrupted system calls, is when a process initiates a read from a terminal device and the user at the terminal walks away from the terminal for an extended period.
- In this example, the process could be blocked for hours or days and would remain so unless the system was taken down.
- The problem with interrupted system calls is that we now have to handle the error return explicitly.

Interrupted System Calls:

- To prevent applications from having to handle interrupted system calls, 4.2BSD introduced the automatic restarting of certain interrupted system calls.
- The system calls that were automatically restarted are **ioctl**, **read**, **readv**, **write**, **writen**, **wait**, and **waitpid**.
- The first five of these functions are interrupted by a signal only if they are operating on a **slow device**; **wait** and **waitpid** are always interrupted when a signal is caught.

kill and raise functions:

- The **kill** function sends a signal to a process or a group of processes.
- The **raise** function allows a process to send a signal to itself.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
/* Both return: 0 if OK, -1 on error */
```

- The call **raise(signo);** is equivalent to the call **kill(getpid(), signo);**

kill and raise functions:

- There are four different conditions for the *pid* argument to **kill**.

Condition	Description
<i>pid</i> > 0	The signal is sent to the process whose process ID is <i>pid</i> .
<i>pid</i> == 0	The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.
<i>pid</i> < 0	The signal is sent to all processes whose process group ID equals the absolute value of <i>pid</i> and for which the sender has permission to send the signal.
<i>pid</i> == 1	The signal is sent to all processes on the system for which the sender has permission to send the signal.

kill and raise functions:

- A process needs permission to send a signal to another process.
- The **superuser** can send a signal to any process.
- For other users, the basic rule is that the **real or effective user ID of the sender has to equal the real or effective user ID of the receiver**.
- POSIX.1 defines **signal number 0 as the null signal**.
- If the ***signo*** argument is 0, then the normal error checking is performed by kill, but no signal is sent. This is often used to determine if a specific process still exists.
- If we send the process the null signal and it doesn't exist, **kill returns 1 and errno is set to ESRCH**.

alarm and pause functions:

- The **alarm** function allows us to set a timer that will expire at a specified time in the future.
- When the timer expires, the **SIGALRM** signal is generated.
- If we ignore or don't catch this signal, its default action is to terminate the process.

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

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

Returns: 0 or number of seconds until previously set alarm



alarm function:

- The ***seconds*** value is the number of clock seconds in the future when the signal should be generated.
- There is only one of these alarm clocks per process.
- When we call alarm, if a previously registered alarm clock for the process has not yet expired, the number of seconds left for that alarm clock is returned as the value of this function.
- That previously registered alarm clock is replaced by the new value.
- If a previously registered alarm clock for the process has not yet expired and if the *seconds* value is 0, the previous alarm clock is canceled.
- The number of seconds left for that previous alarm clock is still returned as the value of the function.

pause function:

- The **pause** function suspends the calling process until a signal is caught.

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

Returns: 1 with errno set to EINTR

- The only time **pause** returns is if a signal handler is executed and that handler returns. In that case, **pause returns 1 with errno set to EINTR.**

abort function:

- The **abort** function causes abnormal program termination.

```
#include <stdlib.h>
void abort(void);
```

This function never returns

- This function sends the **SIGABRT** signal to the caller. (Processes should not ignore this signal.)
- **abort** overrides the blocking or ignoring of the signal by the process.
- The intent of letting the process catch the **SIGABRT** is to allow it to perform any cleanup that it wants to do before the process terminates.
- If the process doesn't terminate itself, when the signal handler returns, **abort** terminates the process.

sleep function:

```
#include <unistd.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:

1. The amount of wall clock time specified by *seconds* has elapsed.
2. A signal is caught by the process and the signal handler returns.

➤ In **case 1**, the return value is 0.

➤ When sleep returns early, because of some signal being caught (**case 2**), the return value is the number of **unslept** seconds (the requested time minus the actual time slept).