Network Socket Programming - 1

BUPT/QMUL

2010-10-12

Review

- Basic network definitions
  - Terms for Network Devices
  - Terms for Network Performance Parameters
  - Ways to connect to the Internet
  - Terms for Network Types
- Layered architecture

Agenda

- Basic concepts in Network Programming
- Introduction to IP & TCP/UDP
- Introduction to Sockets

Basic Concepts in Network Programming

Basic Concepts in Network Programing

- *Introduction to Network Programming*
- Program Developing
- Basic Concepts
  - Process
  - File Descriptor
  - System Call
  - Signal

Introduction to Network Programming

- Network Programming encompasses various concepts, techniques and issues that are involved in writing programs which will communicate with other remote programs.
- Examples
  - Concepts – how to interact with the protocol stack
  - Techniques – what APIs (Application Programming Interface) to use …
  - Issues – how to handle reliability…

Introduction to NP - *classes*

- Protocol Implementation
  - TCP/IP
  - IPX/SPX
  - ...
- Hiding the complexities
  - Sockets
  - RPC : Remote Procedure Call
- Programming language
  - c, java, Perl, Virtual Basic, ...
- Applications for specific services
  - Mail server
  - Multimedia
  - Banking application
  - ...

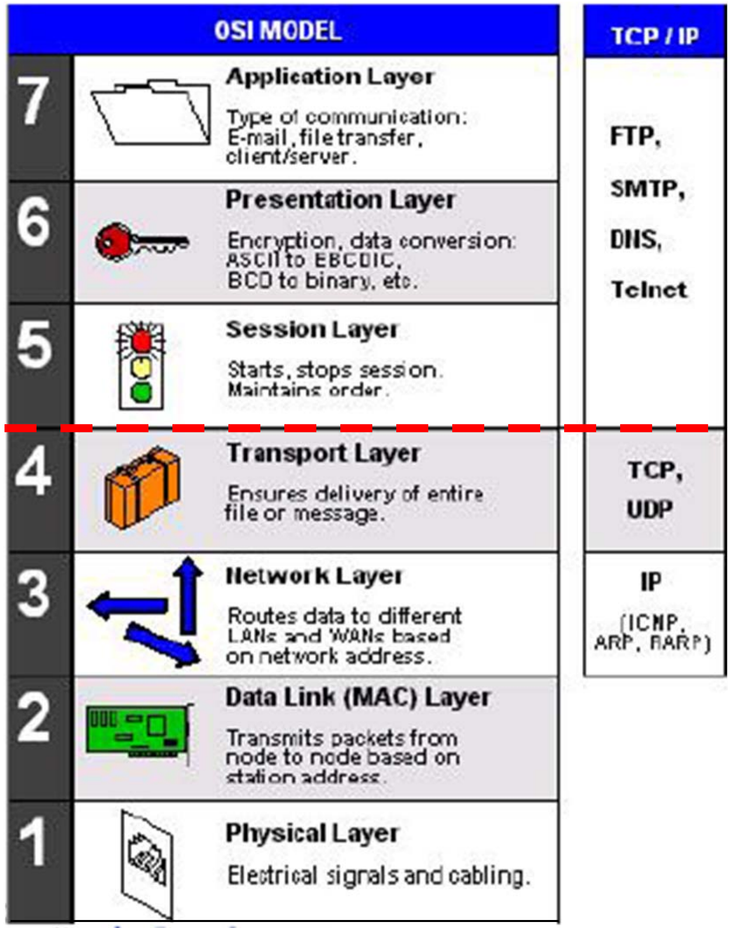
Introduction to NP - *importance*

- Network Programming is a rather wide field
- The concepts and techniques learnt can be helpful in numerous application areas
  - Distributed applications
  - Intelligent/Remotely-managed Devices

Introduction to NP
– *environments in this course*

- TCP/IP nodes on Ethernet
- LINUX as the Operating System
- C language for most sample programs and assignments

Introduction to NP
– *environments in this course*

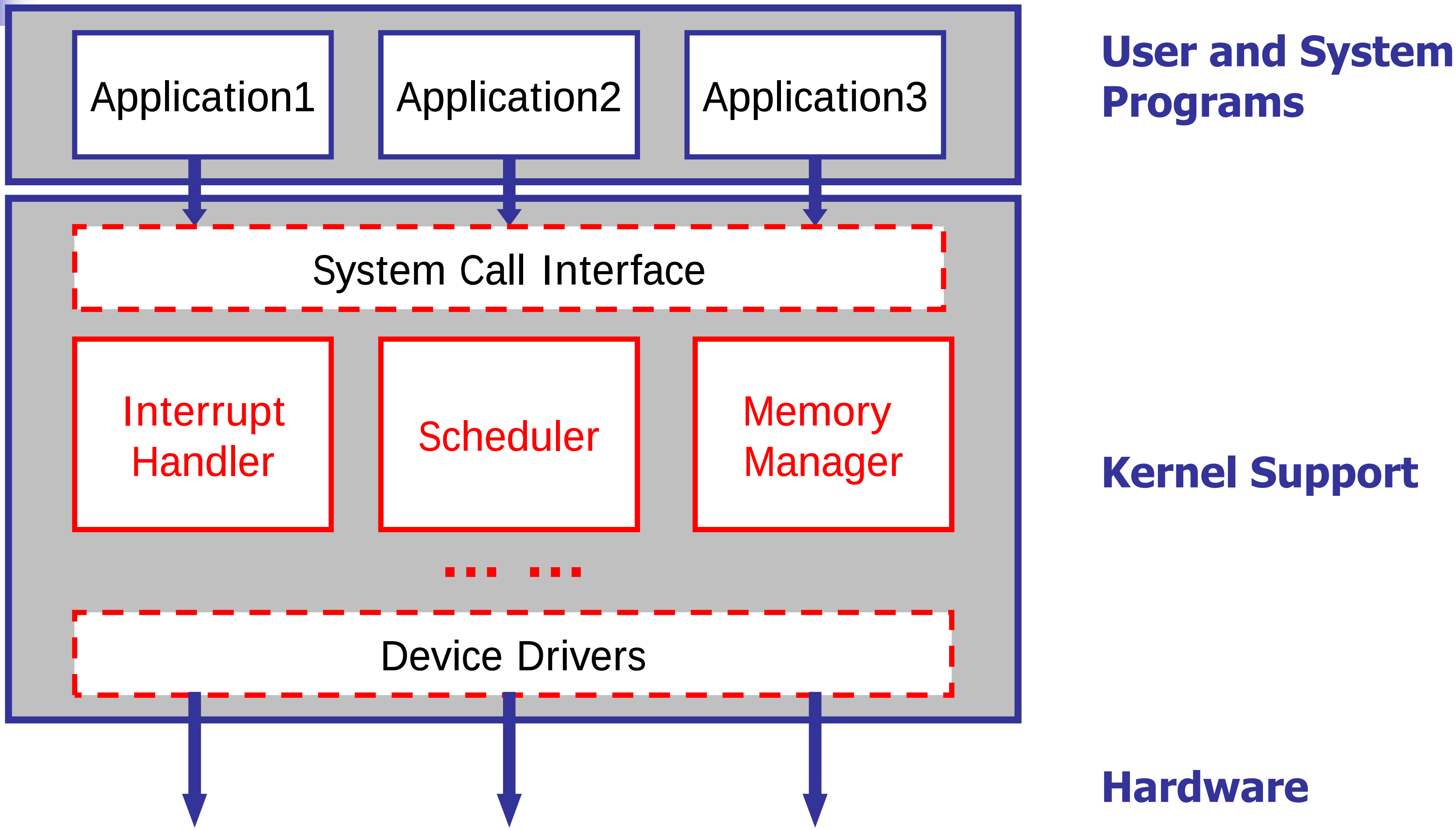


User and System Programs

Kernel Support

Hardware

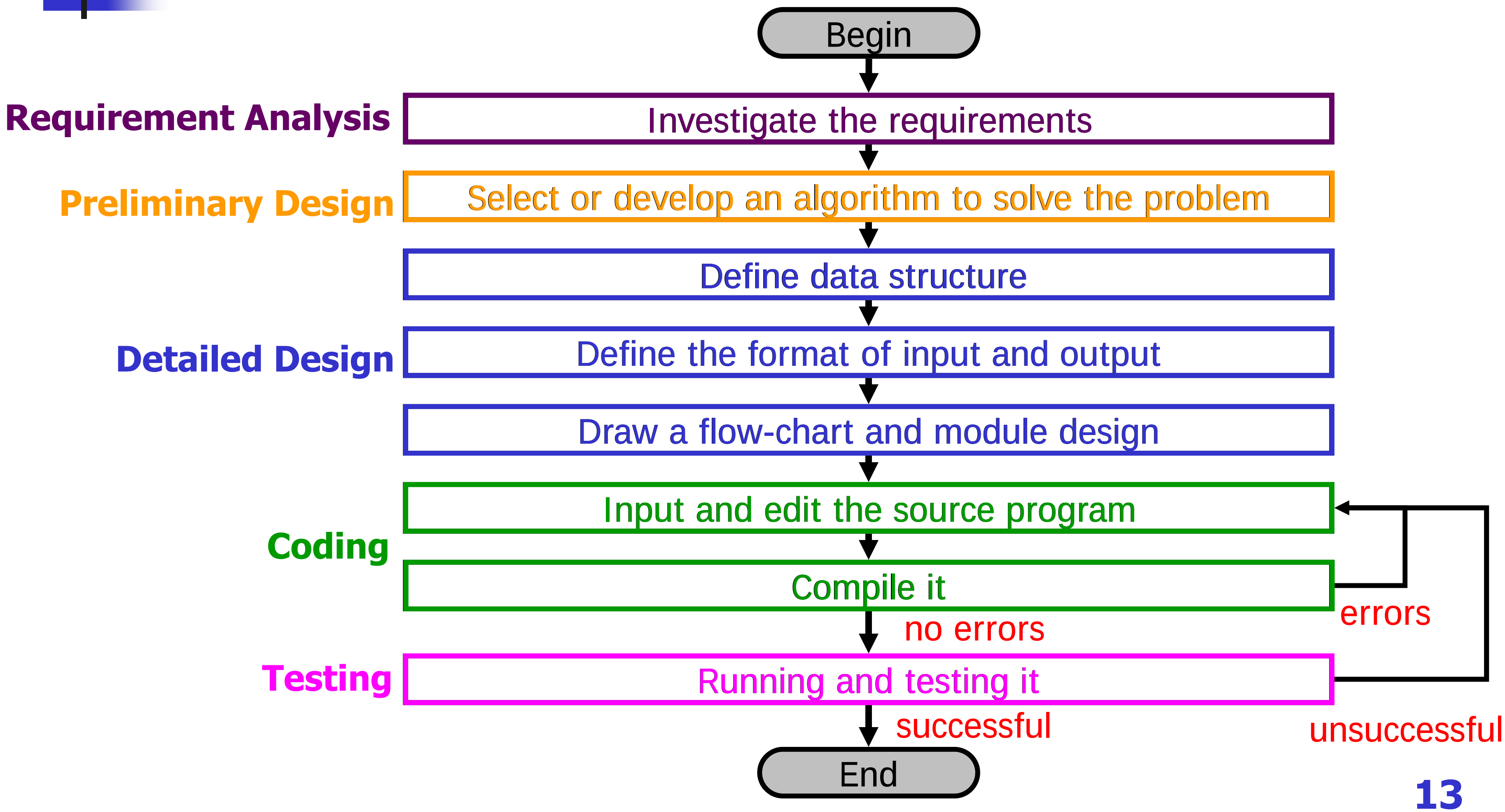
Introduction to NP
– *environments in this course*



Basic Concepts in NP

- Introduction to Network Programming
- *Program Developing*
- Basic Concepts
  - Process
  - File Descriptor
  - System Call
  - Signal

Program Developing - *Phases*



Program Developing - *skills*

- Programming style
  - ident, remarks, variable names,
  - http://www.chinalinuxpub.com/bbs/archive/index.php/t-436.html
- Editor
  - vi, a very powerful full screen editor
  - pico, an utility with Linux
  - http://network.ustc.edu.cn/doc-program/linux/Linux-manual.html
- Related Linux/Unix command
  - http://tech.sina.com.cn/2000-04-25/46/1.html
- Backup your program is important!

Program Developing – *C Compiler in Linux*

- cc
  - Example: % cc test1.c -o test
  - test1.c : program to be compiled
  - -o : specify the name for running program

- gcc
  - Example: % gcc test1.c –o test

Program Developing – *debugger in Linux*

- gdb [options] [executable-file [core file or process-id]]
- Example: % gdb test1
- gdb Command list
  - file : load the program for debugging
  - kill : stop the program for debugging
  - list : list the source code of the program for debugging
  - break : set a break point in the source program
  - run : run the program to be debugged
  - next : execute a single line of the program, but not go into it
  - step : execute a single line of the program, but go into it
  - quit : quit the gdb to shell
  - print : display the value of a variable
  - make : make a run-able program without quiting gdb
  - c : Continue running your program (e.g. at a breakpoint)
  - bt (backtrace) : display the program stack
- Reference web: http://www.chinalinuxpub.com/doc/pro/gdb.html

Basic Concepts in NP

- Introduction to Network Programming
- Program Developing
- ***Basic Concepts***
  - Process
  - System Call
  - File Descriptor
  - Signal

Basic Concepts - *definition*

Process	▪ A process is an instance of a program that is being executed by the operating system.
System Call	▪ Linux/Unix kernel provides a limited number (typically between 60 and 200) of direct entry points through which an active process can obtain services from the Kernel.
File Descriptor	▪ A file descriptor is a small integer used to identify a file that has been opened for I/O operation.
Signal	▪ A signal is a notification to a process that an event has occurred.

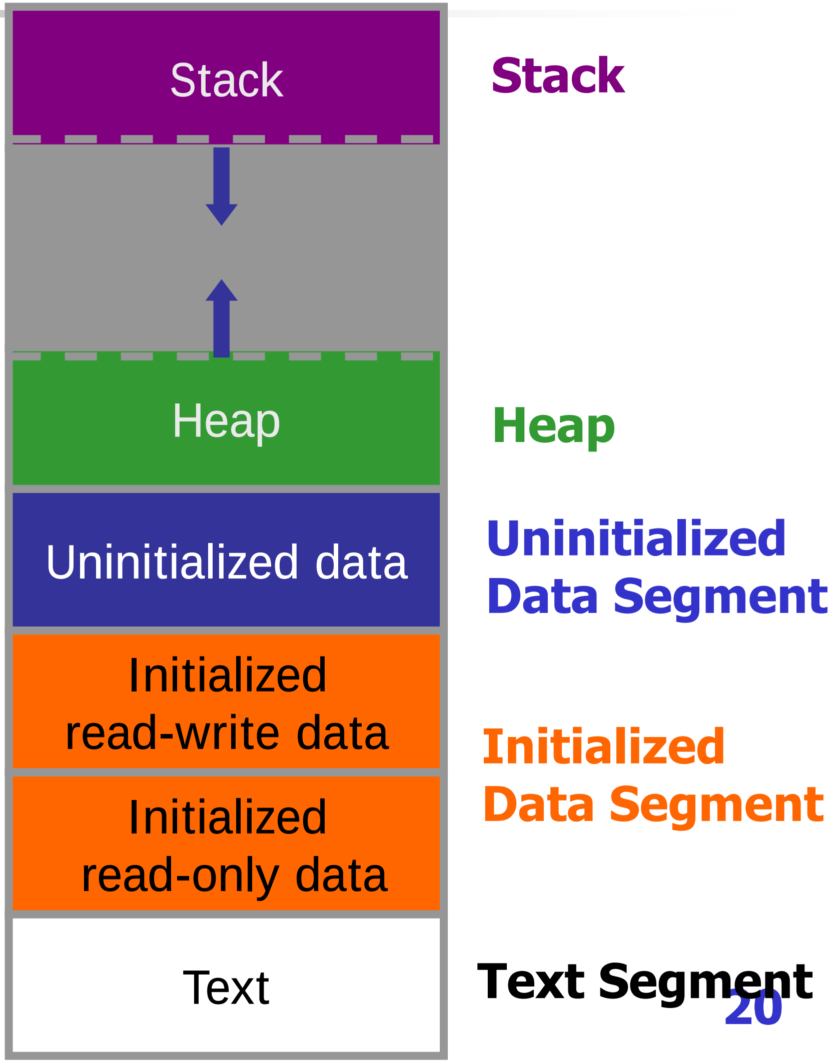
Basic concepts - *process*

- One of the most basic abstractions in Unix (the other one is File)
- process ≠ program
  - Program：a file containing instructions to be executed, static
  - Process: an instance of a program in execution, live entity
  - One program can have multiple processes
  - One process can invoke multiple programs
- Alias: task, job

Basic concepts - *process*

- Process is the basic unit for resource allocation in operating system

- Process in memory
  - Text: program code
  - Stack: temporary data (local variable, function parameters, return addresses)
  - Data: global variables
  - Heap: dynamic allocated memory, malloc()



Basic concepts – *process*

- PID (Process ID): Every process has a unique PID. The PID is an integer, typically in the range 0 through 32,767.
- PPID (Parent PID): Every process has a parent process ID.
- Special process
  - PID = 1: init process
  - PID = 0: special kernel process (e.g., idle/swapper process)
  - PID = 2: special kernel process (e.g., page daemon process)

- Linux command
  - ps –ef
  - To see every process on the system

```
[root@localhost ~]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Aug06  ?     00:00:00 init [5]
root         2     1  0 Aug06  ?     00:00:02 [migration/0]
root         3     1  0 Aug06  ?     00:00:00 [ksoftirqd/0]
```

Basic concepts – *process*

- Related system calls
  - fork(): to create a child process
  - getpid(): to obtain the PID of a process
  - getppid(): to obtain the PPID(Parent Process ID) of a process
  - exec(): often used after fork() to load another process
    - execl(), execv(), execle(), execve(), execlp(), execvp()
  - exit(): to terminate a process and release all the resources

Basic concepts – *process*

- Simple sample program of fork() – fork1.c

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

int main(void) {
        pid_t pid;
        pid=fork();
        printf("fork returned %d\n",pid);
        exit(0);
}
```

```
$ gcc fork1.c –o fork1
$ ./fork1
fork returned 0
fork returned 22770
```

Basic concepts – *process*

- Complete sample program of fork() – fork2.c

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

int main (void) {

        pid_t pid;

        printf("Original program, pid=%d\n", getpid());
        pid = fork();
        if (pid == 0) {
                printf("In child process, pid=%d, ppid=%d\n",
                        getpid(), getppid());
        } else {
                printf("In parent, pid=%d, fork returned=%d\n",
                        getpid(), pid);
        }
}
```

```
Original program, pid=987
In child process, pid=988, ppid=987
In parent, pid=987, fork returned=988
```

Basic concepts – *process*

- Sample program of exec() – exec1.c

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

int main (void) {

        char *arg[] = { "/bin/ls", 0 };

        /* fork, and exec within child process */
        if (fork() == 0) {
                printf("In child process:\n");
                execv(arg[0], arg);
                printf("I will never be called\n");
        }
        printf("Execution continues in parent process\n");
}
```

```
[shiyan@localhost examples-for-ia]$ ./exec1
In child process:
Execution continues in parent process
[shiyan@localhost examples-for-ia]$ exec1  exec1.c      fork1
fork1.c  fork2  fork2.c
```

Basic concepts – *file descriptor*

- A file descriptor (an integer) is returned when a file is opened or created, and is used as an argument when later the file is read or written

- File descriptors are assigned by the kernel when the following system calls are successful
  - open
  - creat
  - dup
  - pipe
  - fcntl

Basic concepts – *file descriptor*

- There are two methods available under Unix for doing I/O (Input and Output)

method	Unix system calls for I/O	standard I/O library
concept	Working with file descriptors	Working with stream
header file	<unistd.h>	<stdio.h>
examples	open, read, write, lseek, …	printf, putc, getc, …

Basic concepts – *file descriptor*

- Related system calls
  - open
    - Be used to open or create a file
    - int open (char *pathname, int oflag, int mode);
  - close
    - Be used to close a file
    - int close (int filedes);
  - read
    - Be used to reading the data from an opened file
    - int read (int filedes, char *buff, unsigned int nbytes);
  - write
    - Be used to writing data into an opened file
    - int write (int filedes, char *buff, unsigned int nbytes);

File descriptor or -1

0 or -1

Bytes of data that are read, 0 or -1

Bytes of data that are written or -1

Name of the file

O_RDONLY
O_WRONLY
O_RDWD
…
(<fcntl.h>)

Only used when creating a file to indicate the access authority

Basic concepts – *file descriptor*

- Related system calls
  - lseek
    - Be used to locate in a file
    - long lseek (int filedes, long offset, int whence);
  - dup
    - Be used to duplicate a file descrip
    - int dup (int filedes);
    - int dup2 (int filedes, int filedes2);
  - fcntl
    - Be used to change the properties already open
    - int fcntl (int filedes, int cmd, int arg);

SEEK_SET
SEEK_CUR
SEEK_END

The new offset in the file or -1

The new file descriptor or -1

Depending on cmd or -1

Indicate the different functions of fcntl():
F_DUPFD,
F_GETFD/F_SETFD,
F_GETFL/F_SETFL,
F_GETOWN/F_SETOWN,
F_GETLK/F_SETLK/F_SETLKW

```
dup(filedes); <--> fcntl(filedes,F_DUPFD,0);
```

Basic concepts – *file descriptor*

- Sample program of lseek() – lseek1.c

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

char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";
#define FILE_MODE 0644

int main(void)
{
        int fd;

        if ((fd=creat("file.hole",FILE_MODE))<0)
        {       printf("creat error\n");
                exit(1);}
```

Basic concepts – *file descriptor*

- Sample program of lseek() – lseek1.c

```
        if (write(fd,buf1,10)!=10)
        {       printf("buf1 write error\n");
                exit(1);}
         /*offset now = 10 */
        if (lseek(fd,40,SEEK_SET)==-1)
        {       printf("lseek error\n");
                exit(1);}
         /*offset now = 40 */
        if (write(fd,buf2,10)!=10)
        {       printf("buf2 write error\n");
                exit(1);}
         /*offset now = 50 */
        exit(0);
}
```

Basic concepts – *file descriptor*

- Sample program of lseek() – lseek1.c

```
[shiyan@localhost examples-for-ia]$ ls -l file.hole
-rw-r--r--  1 shiyan shiyan 50 10ÔÂ  7 10:57 file.hole
[shiyan@localhost examples-for-ia]$ od -c file.hole
0000000   a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040  \0  \0  \0  \0  \0  \0  \0  \0   A   B   C   D   E   F   G   H
0000060   I   J
0000062
```

- od command
  - be used to display the content of the file
  - -c: display in character format
- When using lseek, the offset of the file can be larger than the length of the file. So, the next writing operation will extend the file and a hole will be made inside the file.

Basic concepts – *file descriptor*

- Sample program of read() and write() – readwrite1.c

```
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
        char quit='.';
        char buf[10];
        int fd;
        if((fd = open("out.out",O_RDWR | O_CREAT,0))==-1)
                printf("Error in opening\n");
        while(buf[0]!=quit)
        {
                read(0,buf,1);
                write(fd,buf,1);
                write(1,buf,1);
        }
        close(fd);
}
```

Basic concepts – *system call*

- System call is the only method for the user space to access the kernel

Basic concepts – *signal*

- Signals are some time called “Software interrupts”
- Signals can be sent from one process to another or from kernel to a process
- Header file: <signal.h>
- The names of the signals begin with SIG
  - SIGALRM: alarm clock timeout
  - SIGINT: Interrupt character (Ctrl-C) is typed

Basic concepts – *signal*

Five conditions that generate signals

1 Kill system call

- the system call ***kill*** allows a process to send a signal to another process or to itself

2 Kill command

- the command ***kill*** is also used to send a signal
- often used to terminate a background process out of control

3 Certain terminal characters

- e.g., the interrupt character (typically control-C or Delete) terminates a process that is running - it generates a SIGINT signal

4 Certain hardware conditions

- the hardware detects these conditions and then notifies the kernel. E.g., invalid storage access - SIGEGV

5 Certain software conditions

- the kernel notices these conditions and generates the signal. E.g., SIGALRM

Basic concepts – *signal*

What can a process do with a signal?

1 **Catch the signal**

- A process can provide a function that is called whenever a specific type of signal occurs. This function is called **handler**.

2 **Ignore the signal**

- A process can choose to ignore a signal
- Two signals that can not be ignored: SIGKILL, SIGSTOP

3 **Execute the default action**

- A process can allow the default to happen
- default actions of most signals are to terminate the process

Basic concepts – *signal*

- signal()

```
#include <signal.h>
void (*signal (int signo, void (*func)(int))) (int);
```

Argument 1:Signal name

e.g., SIGINT, SIGALRM

Argument 2:actions

- SIG_IGN (ignore the signal)
- SIG_DFL (execute default action)
- Address of the function handling the signal (catch the signal)

...dling

The signal function itself returns a pointer to a function. The return type is the same as the function that is passed in, i.e., a function that takes an int and returns a void

The signal to be caught or ignored is given as argument *sig*

The *handler* function Receives a single integer Argument and returns *void*

```
#include <signal.h>

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

- **signal returns a pointer to the PREVIOUS signal handler**

Signal is a function that takes two arguments: *sig* and *handler*

The function to be called when the specified signal is received is given as a pointer to the function *handler*

The returned function takes a integer parameter.

Basic concepts – *signal*

- Sample program of signal() – signal1.c

```
#include <signal.h>
void signalRoutine(int);

int main(void)
{
        printf("signal processing demo program\n");
        while(1)
        {
                signal(SIGINT,signalRoutine);
        }

}

void signalRoutine(int dummy)
{
        printf("Signal routine called[%d]\n",dummy);
        exit(0);
}
```

Basic concepts – *signal*

- Sample program of signal() – signal1.c

```
[shiyan@localhost examples-for-ia]$ ./signal1
signal processing demo program
Signal routine called[2]
[shiyan@localhost examples-for-ia]$
```

Introduction to IP & TCP/UDP

Please review the notes of course
"Telecommunication and Internet Fundamentals"

Introduction to Sockets

- you can prepare this part in our textbook given by Chapter 21 - the Socket Interface

Reference books

- W. Richard Stevens, ***Advanced Programming in the UNIX Environments***. 中译本: 尤晋元译, 机械工业出版社.
- W. Richard Stevens, ***UNIX Network Programming, Volume 1***. 中译本: 施振川等译, 清华大学出版社.
- Robert Love, ***Linux Kernel Development***. 中译本: 陈莉君 康华 张波 译, 机械工业出版社.

Labs

- Run all the sample programs today and answer the mentioned questions.
- Learn about the O_APPEND flag used in open() .
- Question:

1. How to use lseek() to append a file to another file?
2. When using O_APPEND to open a file for reading and writing, can we read the file from any location using lseek()? Can we update the data in any part of the file using lseek()?

- Please write a program to verify your answer.

Abbreviations

API	Application Programming Interface
IP	Internet Protocol
IPX	Internetwork Packet Exchange
Perl	Practical Extraction and Report Language
PID	Process Identifier
PPID	Parent Process Identifier
RPC	Remote Process Call
SPX	Sequenced Packet Exchange
TCP	Transport Control Protocol
UDP	User Datagram Protocol