File management & Trace utilities

Agenda

1 **Overview of Unix Architecture**

2 **File I/O**

3 **strace & ltrace**

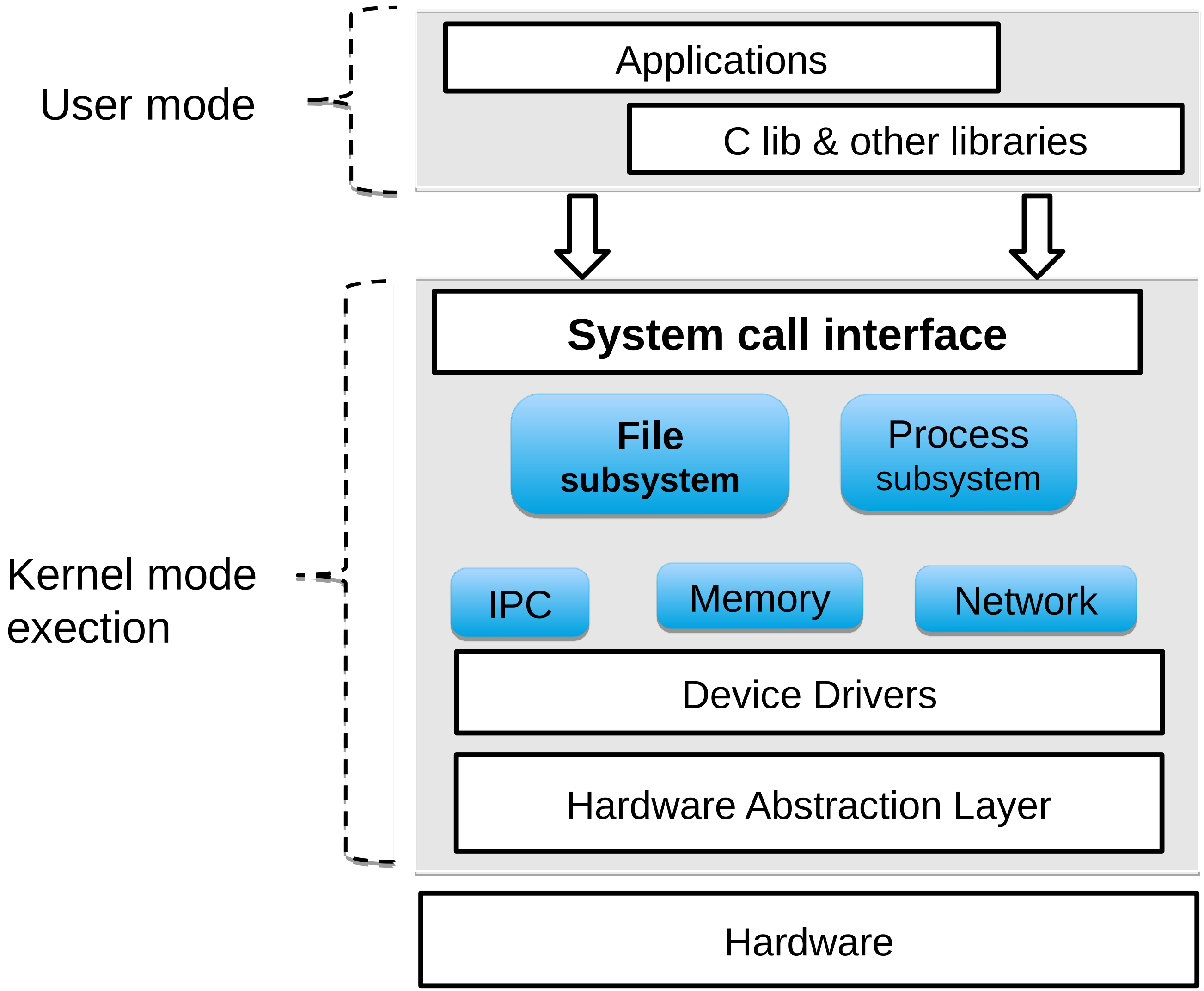
Objectives

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

- get overall view of architecture of UNIX operating system
- differentiate between user mode & kernel mode execution
- know what a system call is and its purpose
- program file I/O using file system calls
- use strace and ltrace to understand a program execution and identify system call / library call related errors

Overview of Unix Architecture

UNIX Architecture – A programmer perspective



User mode & kernel mode execution

- Multiuser & multitasking operating systems such as UNIX have two modes of execution, namely, user mode and kernel mode
  - Such a mechanism is required to ensure reliability and integrity of the operating system
- An application in UNIX spends its execution either in user mode or kernel mode.
- Much of the execution of an application takes place in user mode, which is less privileged mode.
  - A number of restrictions are imposed on what an application can do while it is executing in user mode
  - For example, can't get access to data structures associated with the kernel or of the process itself
- Kernel mode execution is privileged mode
  - Through system calls, a process can get access to and also can manipulate data structures of its own or of the kernel
  - Operations performed in privileged mode is determined by the logic of the program.

User mode & kernel mode execution (Contd.).

- When a system call is invoked, the process has to make a transition from user mode to kernel mode
  - How the transition takes place depends on the processor
    - For x86 processors, software interrupt number 128 (i.e., 0x80) serves the purpose
  - On return from a system call, the process reverts back to user mode

- The command **time** provides the time spent in user & kernel mode by an application (passed as argument to the command)

```
$ time ls -l /dev >/dev/null

real    0m0.087s
user    0m0.070s
sys     0m0.020s
$
```

System Calls

- ***System Call*** is an interface or entry point to operating system.
- System call execution takes place in kernel mode.
- System calls are broadly classified into
  - File & file system related
  - Process & scheduler related
  - Signals
  - Inter-process communication
  - Socket related
  - Kernel related

File I/O

File Management Subsystem

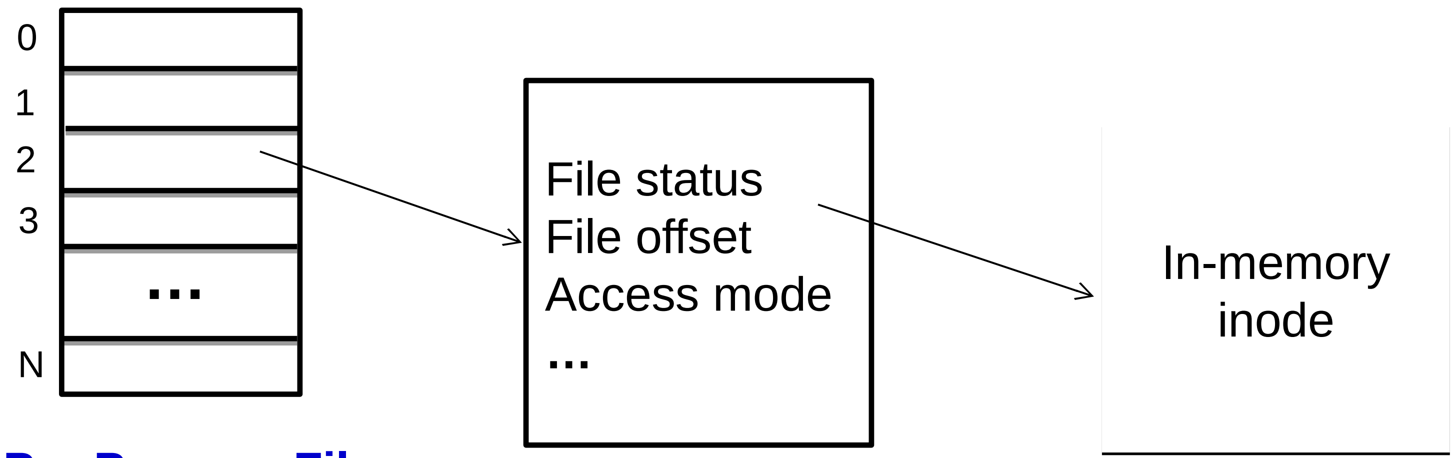
- Of the various subsystems an operating system has, file management subsystem is quite critical from user point of view because, it determines
  - the facilities available
  - ease of use

- C Library provides file I/O functions such as **fopen()** and **fread()**
- These functions in turn call file system calls, to service requests.
- Some system calls related to file subsystem
  - Basic file operations – open, read, write, close, creat,
  - Changing/querying file attributes – chmod, chown,
  - Directory operations - chdir, mkdir, rmdir,
  - Duplicate descriptor – dup, dup2

File Descriptors

- A process can operate on one or more files simultaneously
  - Beware of same file being operated by one or more processes simultaneously
  - The number of descriptors per process is system defined. On exceeding the limit, **“Too many files open”** error is displayed.
- Every process has a table of opened files, called *file descriptor table*.
- System calls such as open() and creat() return a *file descriptor*, corresponding to the named file specified as argument.
  - Subsequently, in the program, operations on the file using system calls are performed by specifying the descriptor, rather than file name
- File descriptor is
  - a non-negative integer number.
  - An index into the ***file descriptor*** table of the process
- The number of descriptors per process is system defined.
- Most file system calls take file descriptor as the first argument

File Descriptor table



Per Process File descriptor table

- Every process that gets created by shell has three descriptors pre-opened
  - descriptor 0 for standard input
  - descriptor 1 for standard output
  - descriptor 2 for standard error

- In programming, descriptors 0, 1 and 2 can be represented by the macros STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO respectively.

File Descriptor table (Contd.).

$ cat abc

0	stdin
1	stdout
2	stderr
3	abc
4	FREE
...	
n	FREE

$ cp in out

0	stdin
1	stdout
2	stderr
3	in
4	out
...	
n	FREE

Open/ create a file

```
#include <fcntl.h>
int  open(const char *filename, int  flag, mode_t mode);
int  creat(const char *filename,  mode_t mode);
```

creat() is equivalent to open() with the flag argument set to O_WRONLY | O_CREAT | O_TRUNC

On success, both return file descriptor, which is used for operations on the file.

Mode can be specified as octal value or using the macros S_IRWXU, S_IRUSR, S_IWUSR, etc

The permissions of the created file are **(mode & ~umask)**.

Open/ create a file (Contd.).

```
#include <fcntl.h>
#include <sys/stat.h>
int open(const char *path, int  flag, int  mode);
```

- Lets file be opened or created
- *flag* can be one or combination of O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT, O_EXCL
- *mode* is the permissions to be set for newly created file

  applicable only if O_CREATE is is specified in *flag* parameter.

Open/ create a file – example

```
int  fd;
fd = open("sample.dat", O_RDONLY);
             // open for reading
             // open fails if file des not exist

fd = open("sample.dat", O_WRONLY);
             // open for writing
             // overwrites existing file
             // open fails if file does not exist

fd = open("sample.dat", O_WRONLY | O_EXCL);
             // open file for writing,
             // only if file does not exist
```

Open/create a file – example (Contd.).

```
fd = open("sample.dat", O_WRONLY | O_CREAT, 0600);
             // create if file does not exist,
             // else overwrite

fd = open("sample.dat", O_TRUNC | O_WRONLY | O_CREAT,
             0600);
             // create if file does not exist,
             // else truncate and write
```

close a file descriptor

```
#include <unistd.h>
int  close(int fd);
```

Closes specified file descriptor.

On success, returns 0; otherwise returns -1.

- It is better to close a descriptor which is no longer required.
- Closing a file descriptor makes it reusable.
- When a process terminates, all file descriptors left open are automatically closed.

read operations

```
#include <unistd.h>
ssize_t  read(int  fd, void *buf, size_t  bufsize);
```

- Reads utmost *bufsize* number of bytes from file descriptor *fd,* and stores the content in *buf*
- Return value   +ve → number of bytes successfully read
                       0   → end of file
                      -1   → an error (*errno* is set appropriately)

- The return value which determines how many actually have been read.
  - this value can be less than or equal to the requested number of bytes
  - This value has to be noted to if the data read into ***buf*** is to be processed later. (e.g., writing the data into another file).

> Note: ***buf*** is a valid block of process memory and can hold utmost ***bufsize*** number of bytes. This must have got statically or dynamically allocated before calling **read()**.

write operations

```
#include <unistd.h>
ssize_t  write(int  fd, void *buf,
                 size_t  bufsize);
```

- writes *bufsize* number of bytes to file descriptor *fd,* from content in *buf*
- Return value +ve   number of bytes successfully written

  -1   an error (*errno* is set appropriately)

- The return value indicates bytes actually written.
  - this can be less than or equal to the requested no. of bytes

Example – file read & write

```
int  filecopy(int  infd, int outfd)
{
   char  buf[BUFSIZE];
   int   n;

   while (( n = read(infd, buf, BUFSIZE)) > 0)
   {
      write( outfd, buf, n );
   }
   return 0;
}
```

writing to a file

```
#include <fcntl.h>
main()
{
    int fd;
    fd = open("abc", O_WRONLY);
    close(fd);
}
```

Assuming the file **abc** does not exist , what is the difference between these two?

```
#include <fcntl.h>
main()
{
    int fd;
    fd = open("abc", O_WRONLY);
    write(fd, "hello", 5);
    close(fd);
}
```

writing to a file (Contd.).

Fix error if any, in the following code.

```
#define BUFSIZE  20
char    buf[BUFSIZE];
int     fdin, fdout;

fdin = open("temp1", O_RDONLY);
fdout = open("temp2", O_WRONLY);

while( read(fdin, buf, BUFSIZE) > 0)
{
      write(fdout, buf, BUFSIZE);
}

close(fdin);
close(fdout);
```

Duplicating descriptor

An existing file descriptor can be duplicated to another descriptor using either dup() or dup2() system call.

Once existing descriptor is successfully duplicated, the new as well as the old descriptor can be used for I/O operations and the effect is the same.

```
#include <unistd.h>
int  dup(int fd);
int  dup2(int oldfd, int newfd);
```

- dup() returns the lowest free file descriptor
- In case of dup2(), if successful, it returns **`newfd`**.
  - If **`newfd`** is a descriptor already in used, the descriptor is first closed, and subsequently duplicated. If **`oldfd`** and **`newfd`** are identical, the descriptor is not closed.

Duplicating descriptor – example

```
char buf[50];
int  fd1, fd2;
fd1 = open("sample.dat", O_RDONLY);
fd2 = dup(fd1);

n = read(fd1, buf, 10);
          // read 10 bytes from the file
write(STDOUT_FILENO, buf, n);


n = read(fd2, buf, BUFSIZE);
          // read 10 bytes from the file
write(STDOUT_FILENO, buf, n);
```

Random seek

lseek() allows to alter the file pointer to a specific position.

```
#include <unistd.h>
off_t  lseek(int fd, off_t offset, int whence);
```

offset	whence	Position of file pointer after the call (new offset)	Remarks
N	SEEK_SET	N	N should be positive
N	SEEK_CUR	Current offset + N	if the new offset is negative, the call fails and returns -1
N	SEEK_END	Current file size + N	if the new offset is negative, the call fails and returns -1

Random seek – example

Assuming a file has records of 100 bytes length each, the following code snippet reads first 20 bytes of each record.

```
#define READSIZE 20
char    buf[READSIZE];
int     fd;
fd = open("temp1", O_RDONLY);

while( 1 )
{
   read(fd, buf, READSIZE);
   // TODO: process data
     lseek(fd, 80, SEEK_CUR); // skip to next
  record
}
```

How can the same be done using SEEK_SET?

Random seek – example (Contd.).

System call	Read/write position before call	Read/write position after call	File size after call
fd = open(“xyz”, O_RDWR)	-	0	500 (assume)
read(fd, buf, 100)	0	100	500
lseek(fd, 10, SEEK_CUR)	100	110	500
read(fd, buf, 100)	110	210	500
lseek(fd, -100, SEEK_CUR)	210	110	500
read(fd, buf, 10)	110	120	500
lseek(fd, -100, SEEK_CUR)	120	20	500
lseek(fd, -100, SEEK_CUR)	20	20 (error, no change in offset)	500
lseek(fd, 200, SEEK_END)	20	700	500 No change in file size
write(fd, buf, 10)	700	710	710 File size change only after a write

Points to Note and Guidelines

1. Points to Note:
   a) **`read()`** can return less than the number of bytes requested.
   b) **`creat()`** opens a file only for writing
   c) Doing an **`lseek()`** farther from the end of file, does not automatically increase the file size, unless a **`write()`** follows the **`lseek()`.**

2. DOs
   a) Always check the return value of system calls
   b) Make it a practice to always close unused file descriptors

strace & ltrace

Tracing a Process

- Process tracing can be done in two ways
  - From command line using strace or ltrace commands
  - Debugging
- strace command executes a program given as an argument to its completion, analyzes the execution and provides various details such as
  - System calls executed and the status of each system call
  - Frequency of system calls which got executed and the time consumed
- strace is an effective tool for debugging
  - Without adding debug code, it is possible to identify which system calls have failed
- strace can not be used to identify failure of library function calls.
- strace can also be used to investigate how a program works

Using strace

strace [options] command [args]

Some often used options are

-c	get statistics of systems calls made, time consumed
-v	verbose, i.e., include full information for system calls
-T	show time spent in each system call
-o outfile	write output of strace into the file outfile

Using strace (Contd.).

```
$ strace date
execve("/bin/date", ["date"], [/* 26 vars */]) = 0
...
clock_gettime(CLOCK_REALTIME, {1326350239, 654703000}) = 0
open("/etc/localtime", O_RDONLY)        = 3
…
read(3, "TZif\
  0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\4\0\0\0\4\0"..., 4096) =
  109
close(3)                                = 0
munmap(0xb7dbf000, 4096)                = 0
stat64("/etc/localtime", {st_mode=S_IFREG|0644, st_size=109, ...})
  = 0
stat64("/etc/localtime", {st_mode=S_IFREG|0644, st_size=109, ...})
  = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) =
  0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
  -1, 0) = 0xb7dbf000
write(1, "Thu Jan 12 12:07:19 IST 2012\n", 29) = 29
exit_group(0)                           = ?
```

Using strace (Contd.).

```
$ strace -c date
Fri Apr 13 18:48:09 IST 2012
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
 32.73    0.000109         109         1           write
 18.92    0.000063           5        14         7 open
 12.31    0.000041           3        12           old_mmap
 10.51    0.000035           6         6           read
  ...
  2.10    0.000007           1         7           close
  1.50    0.000005           1         4           brk
  0.30    0.000001           1         1           gettimeofday
  ...
  0.30    0.000001           1         1         1
 clock_gettime
------ ----------- ----------- --------- --------- ------------
100.00    0.000333                    68         8 total
```

ltrace

- **ltrace** tool is similar to strace but useful to trace calls to shared libraries.

- Also supports tracing of system calls, where as strace does not allow tracing of library calls.

ltrace [options] command [args]

Some useful options

-c □ count time and calls for each library function & report summary

-o file □ write output to the named file

-S □ display system calls as well as library functions

-L □ don't display library functions

Using ltrace

```
$ ltrace date
__libc_start_main(0x8049550, 1, 0xbfffa7e4, 0x804e5d8, 0x804e620
  <unfinished ...>
getenv("_POSIX2_VERSION")                    = NULL
getenv("POSIXLY_CORRECT")                    = NULL
setlocale(6, "")                             = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")                      = "coreutils"
__cxa_atexit(0x804d740, 0, 0, 0x804e7a3, 0xbfffa7e4) = 0
getopt_long(1, 0xbfffa7e4, "Rd:f:r:s:ul::", 0x804e8e0, NULL) = -1
dcgettext(0, 0x804e7fc, 5, 0x804e8e0, 0)     = 0x804e7fc
clock_gettime(0, 0xbfffa6d8, 0, 0, 5)        = 0
localtime(0xbfffa670)                        = 0xe48320
nl_langinfo(131180, 0x4f9e953e, 382212, 0xbfffa6d8, 0xbfffa7e4) = 0xb7515375
realloc(NULL, 200)                           = 0x99e2728
```

Using ltrace (Contd.).

```
strftime("Mon", 1024, "%a", 0xe48320)          = 3
memcpy(0x99e2728, "Mon", 3)                    = 0x99e2728
strftime("Apr", 1024, "%b", 0xe48320)          = 3
memcpy(0x99e272c, "Apr", 3)                    = 0x99e272c
memcpy(0x99e2730, "30", 2)                     = 0x99e2730
memcpy(0x99e2733, "19", 2)                     = 0x99e2733
memcpy(0x99e2736, "05", 2)                     = 0x99e2736
memcpy(0x99e2739, "58", 2)                     = 0x99e2739
strlen("IST")                                  = 3
memcpy(0x99e273c, "IST", 3)                    = 0x99e273c
memcpy(0x99e2740, "2012", 4)                   = 0x99e2740
puts("Mon Apr 30 19:05:58 IST 2012"Mon Apr 30 19:05:58 IST 2012 )      = 29
free(0x99e2728)                                = <void>
exit(0 <unfinished ...>
__fpending(0xe43ca0, 0xb75e85d0, 1, 0, 0xb7515375) = 0
+++ exited (status 0) +++
$
```

Using ltrace (Contd.).

```
$ ltrace -c date
Mon Apr 30 19:09:34 IST 2012
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 24.01    0.000396         396         1 dcgettext
 16.31    0.000269         269         1 setlocale
 12.55    0.000207          25         8 memcpy
 12.43    0.000205         205         1 puts
  9.28    0.000153         153         1 localtime
  3.94    0.000065          65         1 clock_gettime
  3.58    0.000059          29         2 strftime
  ...
  1.52    0.000025          25         1 free
  1.52    0.000025          25         1 realloc
  ...
  1.52    0.000025          25         1 nl_langinfo
------ ----------- ----------- --------- --------------------
100.00    0.001649                    26 total
```

Summary

In this module, we discussed

- The architecture of UNIX operating system from a programmer perspective
- The difference between user mode & kernel mode execution
- The purpose of system calls
- How to program file I/O using file system calls
- How to use strace and ltrace

Review Questions

1. What will be the return value of open(), if a file gets opened successfully?

2. Given the following read call

   ```
   read(0, buf, 50);
   ```

   what can be the possible return values of the read()?

3. Given the following read call

   ```
   read(fd, buf, 50);
   ```

   what assumptions can be made?

4. If a file is to be written, but should prompt whether to overwrite or not, what is the way file should be opened?

5. In what way creat() is different from open()?

References

1) W.Richard Stevens and Stephen A.Rago, Ed 2., New Delhi: Pearson Education, 2009.

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

3) Rochkind, Advanced Unix Programming, Ed 2. New Delhi: Pearson Education, 2008.

4) Arnold Robbins, Linux Programming by Example, New Delhi: Pearson Education, 2008.

Thank You