MODULE 9 :
Operating Systems Structure and Application Services

Operating Systems Services - Review

- User Interface
- Program loading and execution
- Process start up and scheduling
- Address space separation
- Inter-process communication
- Concurrency control
- Dynamic memory allocation
- File system manipulation
- Input / Output Operations
- Networking
- Fault detection and recovery
- Auditing of OS data structures
- Protection of kernel data

Operating Systems Services – Loading and Running an Application

- We have looked briefly at the type of OS services an application (let's continue with the basic example of a text editor) typically requires - slide 12 of previous module.

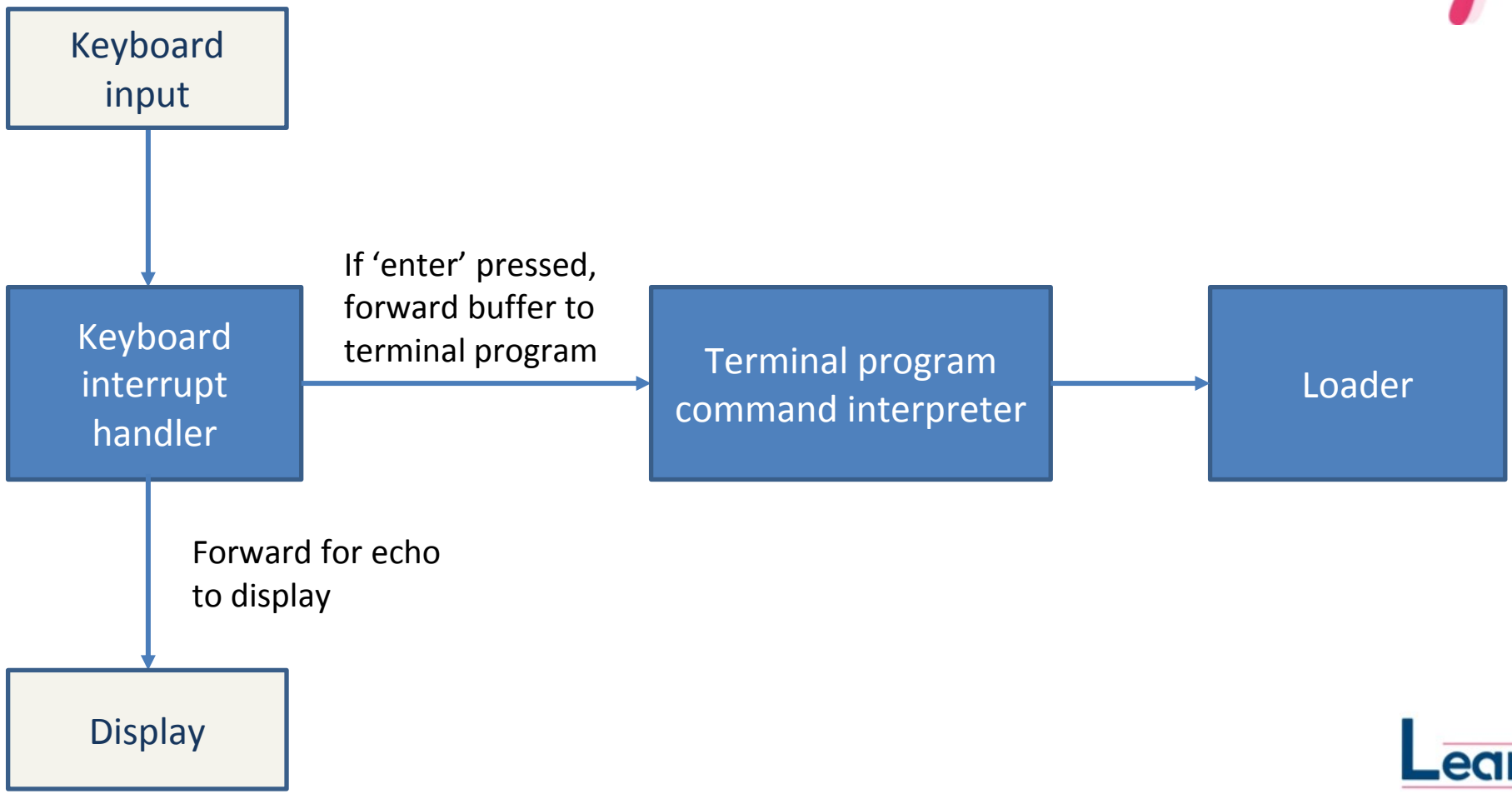
- There's a little bit that the OS needs to do before an application begins to run however:
  - ✔ It needs to have set up a command line/a GUI from where the command to load and run the application can be executed
  - ✔ The application code and data need to be loaded in an area in RAM

Loading and Running an Application (contd.)

- **What exactly is a command line?**
  - ✔ It is an input from the keyboard given to a terminal program, supplied in a line format.
  - ✔ The terminal program, generally a kernel thread, processes the input and calls the corresponding command from a command list that has earlier been set up as part of command line initialization.
  - ✔ The command is a load-and-run-application command in this case.

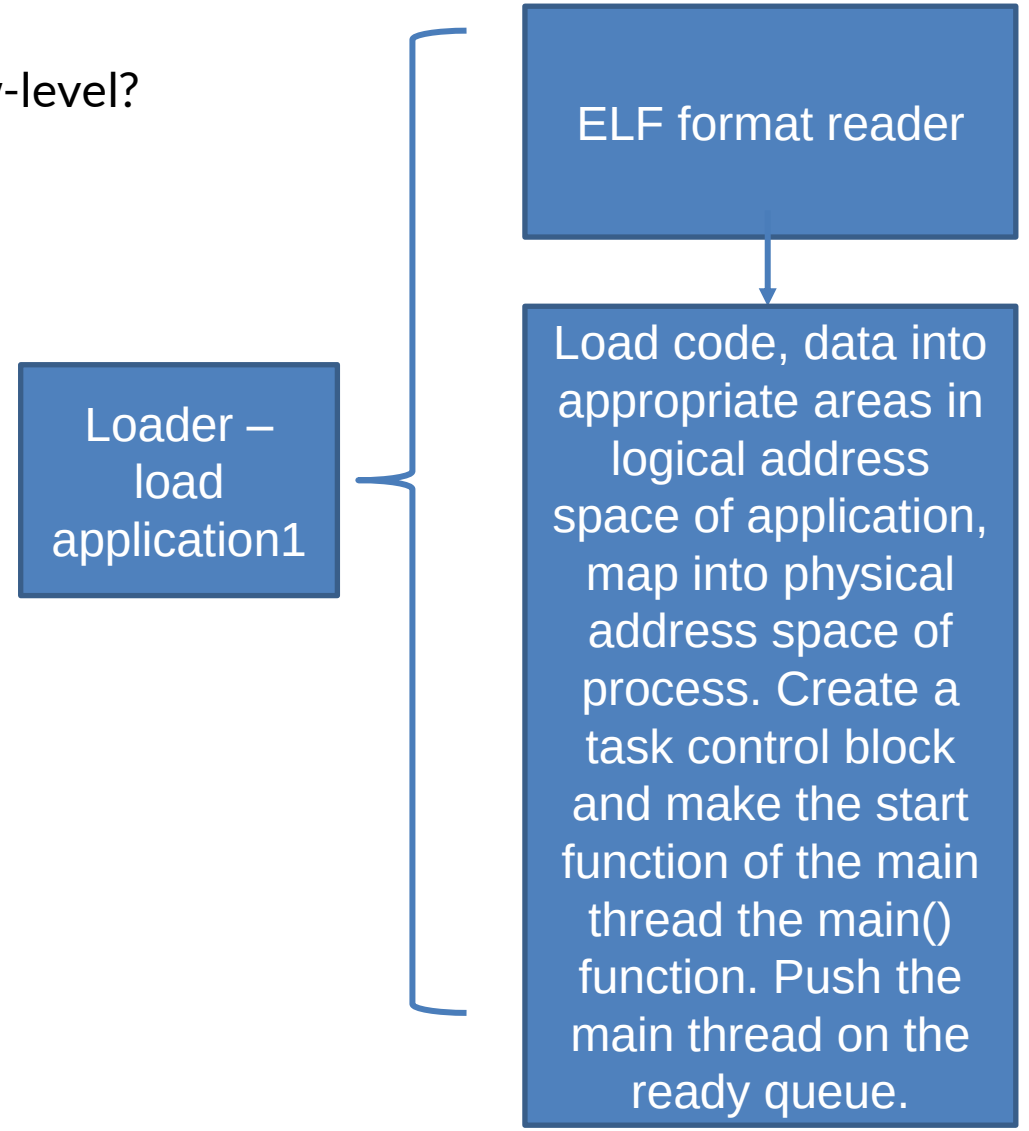
Loading and Running an Application – Command Interpreter

How does this work at a low-level?



Loading and Running an Application – Loader and Application Execution

How does this work at a low-level?



What Does an Application Want from the OS?

Let's look at the text editor slide from the previous module in more detail:

- A text editor wants to read/write from a file on disk – so what type of information would it need to pass to the OS to get a specific file? (We'll look at just basic stuff here)
  - ✔ Obviously, the filename and directory path
  - ✔ A flag to open a new file if it's a new file that the editor wants to open
  - ✔ Permissions to create the file with

- What type of information would it pass to do a read/write?
  - ✔ Filename and directory path
  - ✔ File offset to start the read/write at

How does it get what it wants? (System call internals)

The call to open on Linux (or other Unix-like systems) would look something like:

int open(const char **pathname*, **int** *flags*, **mode_t** *mode*);

- The call returns a file descriptor id that is specific to the process. This id is referenced by subsequent application calls to read/write for the same file.
- Internal to the kernel, inside the open system call handler, this id is associated with a kernel file specific data structure (struct file) that along with the file offset has associations with physical page-related structures that help with reads/writes to/from the page cache. Dirtying the page cache triggers the flush out of the physical page to disk.

What happens when you execute a System Call?

- **What is a system call?**
  - ✔ A system call is, under the wraps, a system call/trap instruction that is part of processor instruction set architecture
  - ✔ It takes arguments that indicate the filename(in the case of the text editor or a device path for other device operations) whose read/write/control the application is interested in, and the type of call (open/read/write/control in the case of a file operation or various other values for other system calls)
  - ✔ These arguments passed in by way of the system call library will be pushed into appropriate registers within the library function and passed via the registers to the actual system call instruction

What happens when you execute a System Call?(contd.)

- **What is the purpose of a system call?**
  - ✔ To provide a generic hardware-agnostic interface for the application to call to access an OS service
  - ✔ It switches the processor to kernel mode, allowing privileged data access to kernel memory

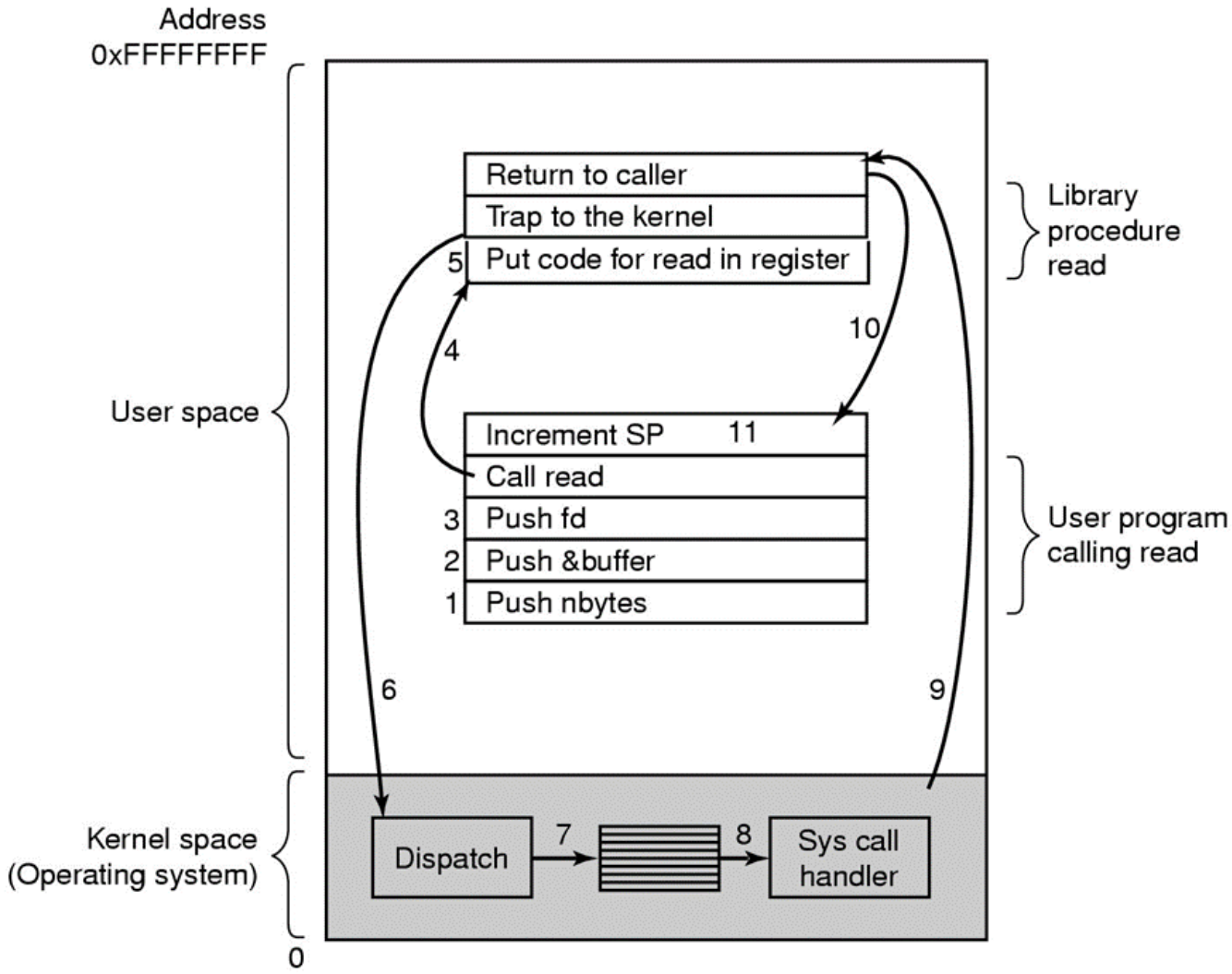
Why do we need to switch from user to Kernel Mode?

- All logical to physical address mappings (page table entries) contain a protection bit.
- This is constantly (during the execution of each instruction) compared with the current execution mode (in an MSR in PPC, CPL register in x86).
- Only if these match (either both are kernel/protected mode or the execution mode is kernel and the PTE indicates user-space memory) will instructions execute without error. If these don't match (execution mode is user/unprotected and PTE is protected), a protection violation exception is reported.

Why do we need to switch from user to Kernel Mode? (contd.)

- The reason the kernel data space is kept protected is obvious – you don't want errant application code to be writing to kernel space in user mode
- When you execute a system call/trap instruction, the current execution mode switches to kernel mode so that kernel mode data can be accessed.
- When you return from the exception by way of a return from interrupt instruction, the execution mode switches back to user/unprotected mode

System Call – A Control Flow Sequence



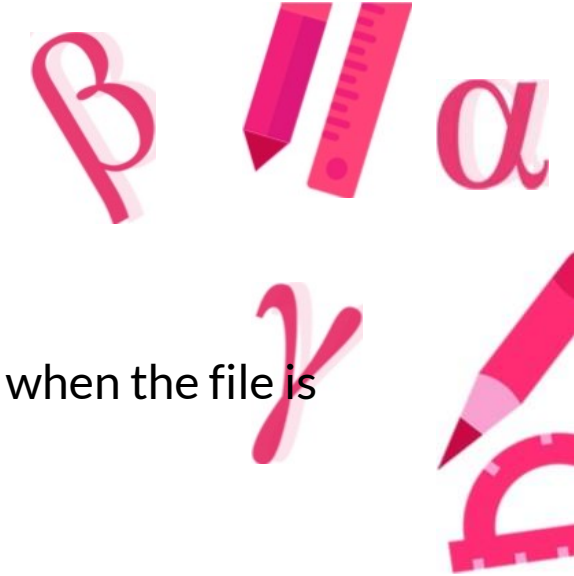
A Sample List of System Calls

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
1	write	sys_write	fs/read_write.c
2	open	sys_open	fs/open.c
3	close	sys_close	fs/open.c
4	stat	sys_newstat	fs/stat.c
5	fstat	sys_newfstat	fs/stat.c
6	lstat	sys_newlstat	fs/stat.c
7	Poll	sys_poll	fs/select.c
8	Lseek	sys_lseek	fs/read_write.c
9	Mmap	sys_mmap	arch/x86/kernel/sys_x86_64.c
10	Mprotect	sys_mprotect	mm/mprotect.c
11	Munmap	sys_munmap	mm/mmap.c
12	Brk	sys_brk	mm/mmap.c

A Sample List of System Calls (contd.)

%rax	Name	Entry point	Implementation
23	select	sys_select	fs/select.c
24	sched_yield	sys_sched_yield	kernel/sched/core.c
25	mremap	sys_mremap	mm/mmap.c
26	msync	sys_msync	mm/msync.c
27	mincore	sys_mincore	mm/mincore.c
28	madvise	sys_madvise	mm/madvise.c
29	shmget	sys_shmget	ipc/shm.c
30	shmat	sys_shmat	ipc/shm.c
31	shmctl	sys_shmctl	ipc/shm.c
44	sendto	sys_sendto	net/socket.c
45	recvfrom	sys_recvfrom	net/socket.c
46	sendmsg	sys_sendmsg	net/socket.c
47	recvmsg	sys_recvmsg	net/socket.c

File Descriptor



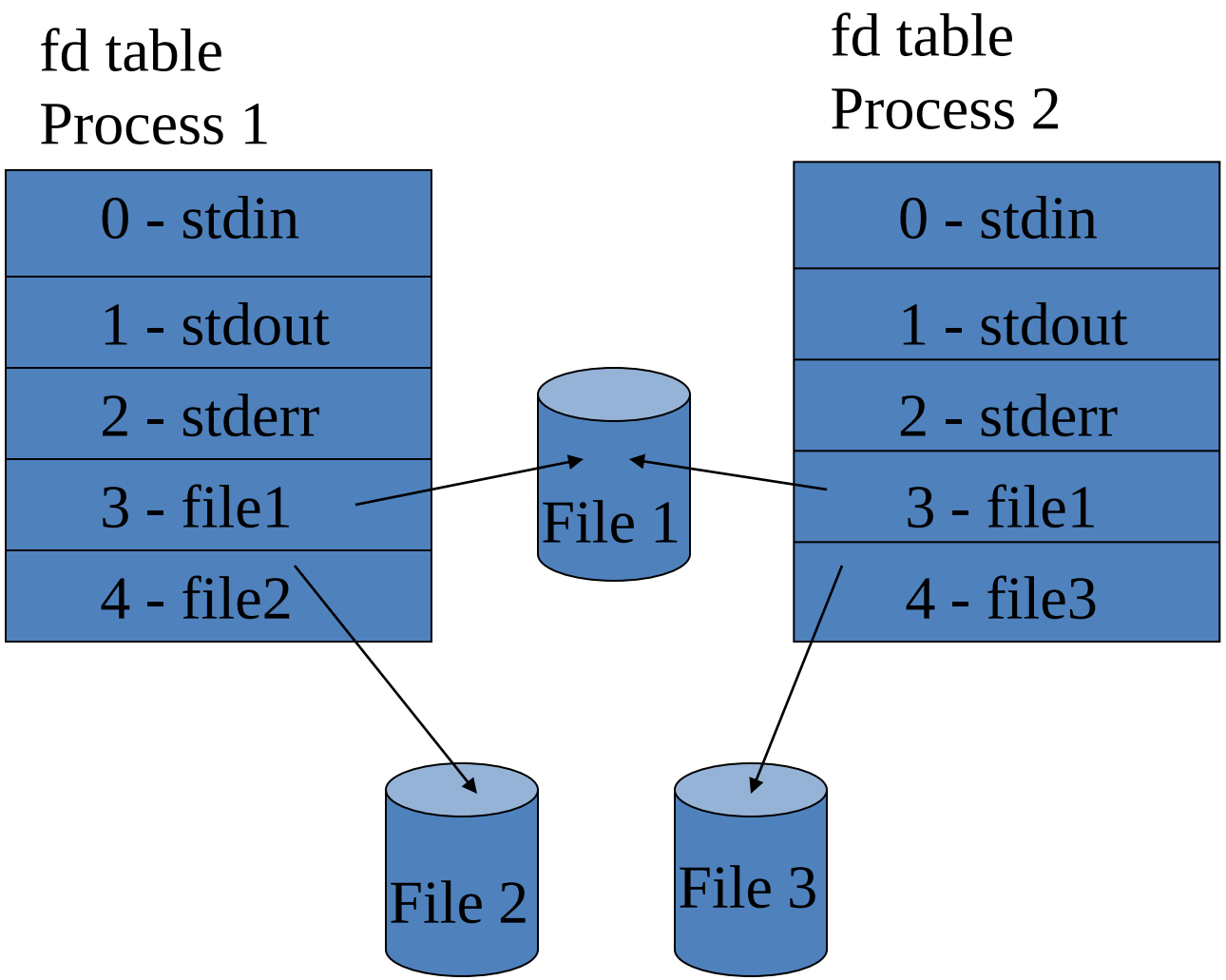
- File Descriptor is lowest available unique integer number given to a file when the file is opened.
- With file Descriptor is similar to file pointer.
- File operations can be performed using File Descriptor.
- When a file descriptor is closed file cannot be accessed.
- Kernel maintains separate file descriptor table for each process.

fd table

FD	File
0	Std Input
1	Std Output
2	Std Error
3	file1
4	file2

- For each process three file descriptors are reserved and they are 0,1 and 2.
- The system calls related to the I/O system take a descriptor as an argument to handle a file.
- If a file open is not successful, fd returns -1.

fd table for each process



File control System Calls

- creat / open
- read, write
- lseek
- close, unlink
- dup / dup2
- fcntl
- select

creat()

- creat system call is used to create a ordinary text file and the syntax is
- int creat (char *file_name, mode_t mode)
- Where,

  file_name is name of the file to be created.

  mode is file permissions in octal form.

creat()

- example : to create a file called sample with file permission 744 the system call is
- creat(“sample”,07444)

  If file is created successfully return value will be a file descriptor or else -1 is retuned.
- create may fail due to

  - File with same name might exist.

  - Space may not be available to create new file.

open()

open system call is used to open file for reading or writing or both and return value of open is file descriptor.

int open (char *file_name, int flags)

Where,

file_name is name of the file to be created.

flags are file opening modes and flags can be.

- O_RDONLY to open the file in read only mode.
- O_WRONLY to open the file in write only mode.
- O_RDWR to open the file in both read & write mode.

Include fcntl.h header file for open system call.

open()

Example: to open a file sample for writing the system call is

open(“sample”,O_WRONLY);

If file is opened successfully return value will be a file descriptor or else -1 is retuned.

open may fail due to

- File may not exist.
- the user might not have the permission to open the file in particular mode.

Example Code

In this example a file is created and opened for read and write.

```
#include<stdio.h>
#include<fcntl.h>
int main()
        {
        int i;
        creat("sample",0777);
        i=open("sample",O_RDWR);
        printf("value of i is %d\n",i);
        }
```

Output is : value of i is 3

open()

Open system call can be used to create a file and the open the file for reading or writing or both. Syntax for creating a file and opening the file is

open("sample", O_CREAT|O_RDONLY, 0777)

In this example a file is created and opened for read and write only using open system call .

```
#include<stdio.h>
#include<fcntl.h>
int main()
        {
        int i;
        i=open("sample",O_CREAT|O_RDWR,0777);
        printf("value of i is %d\n",i);
        }
```

Output is : value of i is 3

write()

Write system call is used to write into the file. Before writing into the file should be opened.

int write (int fd, const void *buf, int count);

fd - file descriptor (return value of open system call)

buf - pointer to the string to be written in file

count - number of character to be written into the file

It writes count bytes to the file from the buf.

On success return with:

Number of bytes written

0 indicates nothing was written

-1 on error.

read()

read system call is used to read content of a file. Before reading file should be opened.

int read (int fd, const void *buf, int count);

fd - file descriptor (return value of open system call).

buf - address of a variable where data is to be read.

count - number of character to be written into the file.

It reads count bytes from the file and store the data into the buf.

On success return with:

Number of bytes written

0 indicates nothing was written

-1 on error

Random Access lseek()

lseek system call is used to change the cursor position in a file.

int lseek (int fd, long int offset, int whence);

fd – file descriptor (return value of open system call).

offset – number of bytes to be moved

Whence -

SEEK_SET - from the beginning

SEEK_CUR – from the current position

SEEK_END – from the end of file

On success the system call returns with any one of the following value:

Offset value

0

-1

Lseek()

Example1 : to set the cursor at first position system call is

lseek(fd,0,SEEK_SET)

Example2 : to set the cursor at fifth position from the start of the file, system call is

lseek(fd,5,SEEK_SET)

Example3 : to move the cursor position at 8th byte from current cursor position, system call is

lseek(fd,8,SEEK_CUR)

Example3 : to move the cursor position at 8th byte from current cursor position, system call is

lseek(fd,8,SEEK_END)

Example Code

In this example 10 bytes are written in file1 and read back

```
#include<stdio.h>
#include<fcntl.h>
int main()
{
int fd1;
char buf1[11]={0};
fd1=open("file1",O_RDWR);
write(fd1,"hello world",10);
lseek(fd1,0,SEEK_SET);      //set the cursor
position
read(fd1,buf,10);
printf("data read from file1 is %s\n", buf);
close(fd1);
}
```

Example Code

In this example 10 bytes of file1 is copied into file2

```
#include<stdio.h>
#include<fcntl.h>
int main()
{
int fd1,fd2;
char buf[11]={0};
fd1=open("file1",O_RDONLY);
fd2=open("file2",O_WRONLY);
read(fd1,buf,10);
write(fd2,buf,10);
close(fd1);
close(fd2);
}
```

Example Code

In this example file1 is copied into file2

```
#include<stdio.h>
#include<fcntl.h>
int main()
{
int fd1,fd2,i=2;
char buf;
fd1=open("file1",O_RDONLY);
fd2=open("file2",O_WRONLY);
while(read(fd1,&buf,1)!=0)
        {
        write(fd2,&buf,1);
        }
close(fd1);
close(fd2);
}
```

Designing an Operating System?

Let’s consider the absolutely mandatory parts of an Operating System first:

- The scheduler
- The Memory Manager
- The Interrupt/Exception handlers

Some basic IO and debug mechanisms are mandatory as well, but let’s focus on the above parts for now.

How do you go about designing an Operating System(contd.)?

1. **First question you have got to ask is about the type of applications that are expected to run on this OS.**

You have to look at:

- ✔ The nature of the applications from a latency point of view. How big a delay is acceptable from an application data processing point of view? What is the expected throughput of application tasks? Are all applications equally time critical? Is starvation of lower priority tasks acceptable? The answers to these questions will decide the scheduling scheme of the OS.

How do you go about designing an Operating System (contd.)?

2. **Second question you have to ask is how fault tolerant the system should be.**

You have to look at:

- ✔ If the fault scenarios are in the software domain, you need kernel audit tasks auditing ready queues, software watchdogs looking for starvation and similar kernel-level checks for memory management data structure inconsistencies.

How do you go about designing an Operating System (contd.)?

- If CPU resets are the favoured recovery mechanisms in some scenarios, then migration of the tasks on the offending CPU's ready queue need to be migrated to suitably less busy CPUs etc.

- Software upgrades are a very critical part of highly fault tolerant systems. Active and passive storage partitions need to be supported and boot/kernel file consistency checks need to be used, and fallback mechanisms need to be in place in case of errors. In-field patching of code might also need to be supported.

How do you go about designing an Operating System (contd.)?

3. **Third question is what type of operating environment you are going to run the OS and application:**

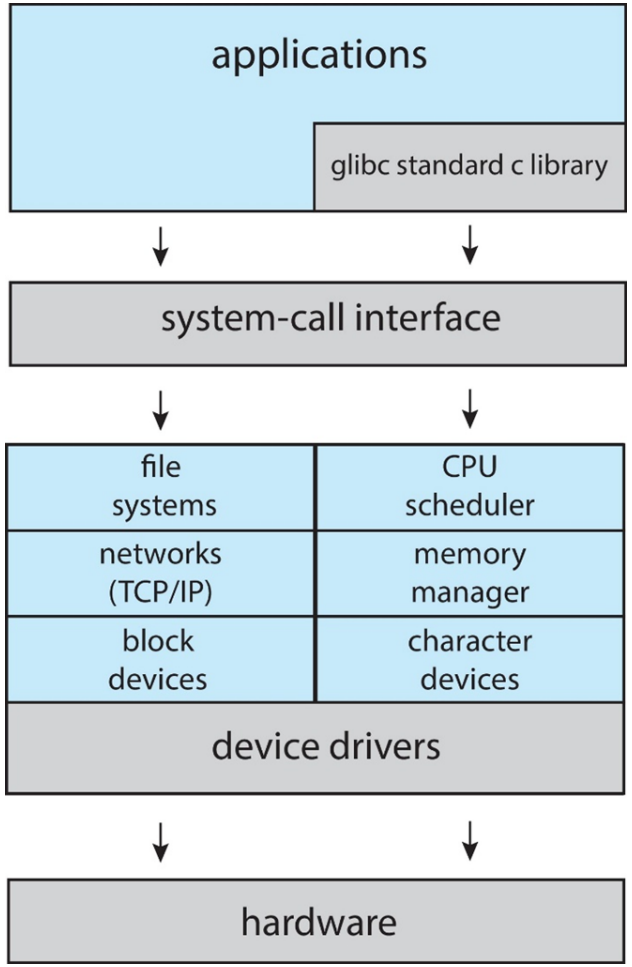
- ✔ Most embedded systems where the software execution environment is more controlled - definite set of applications requiring very predictable execution on proprietary hardware – you will opt for simplicity in OS design and not worry about flexibility as much.
- ✔ For instance, you might opt to always execute at kernel level, ignoring the extra protection of user-kernel space separation.
- ✔ Modules will get loaded into areas defined at build time.
- ✔ There will be no address space related context switch expenses when the scheduler switches between tasks.
- ✔ On more general purpose systems, the reverse design assumptions hold and you might opt for flexibility and protection over simplicity and time saving.

How do you go about designing an Operating System(contd.)?

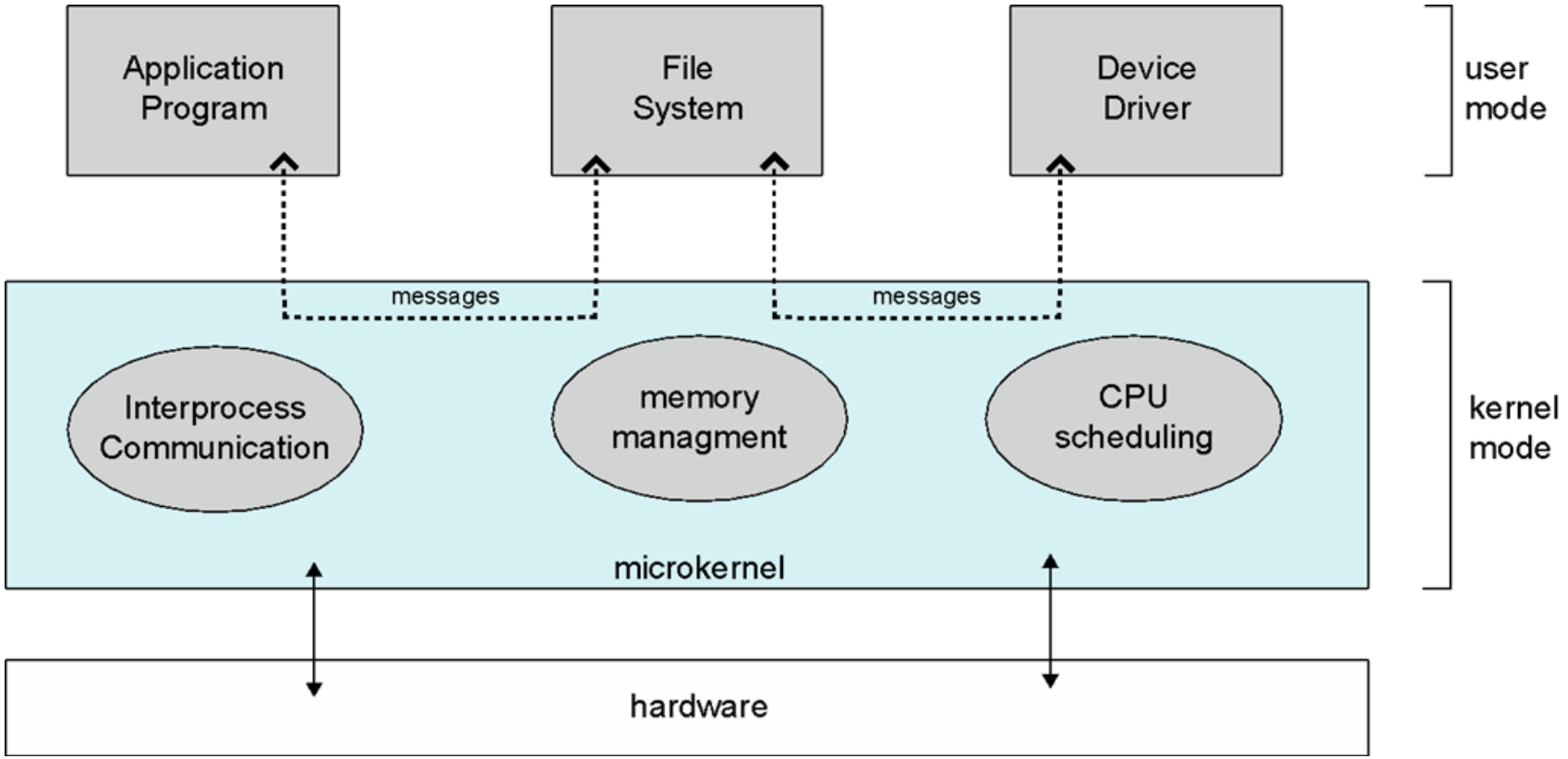
4. **What type of modularity does the OS need to support?**

- ✔ Can we afford some parts of the kernel to be in user space?
- ✔ For instance, can the File System and some Device Drivers be in user space and can these services be accessed by message passing rather than system calls?
- ✔ We will have to compromise a bit on performance or develop specialised fast messages specifically for these OS services.
- ✔ It will gives us flexibility in terms of easier debugging in user space and less time spent in implementation because user space development is an easier option.

Linux – Monolithic Design



QNX– Microkernel Design



Thank You!