LINUX Internals

DAY -1

What is Linux ?

- Linux is an operating system that was initially created as a hobby by a young student, Linus Torvalds, at the University of Helsinki in Finland.
- Linus began his work in 1991 when he released version 0.02 and worked steadily until 1994 when version 1.0 of the Linux Kernel was released.
- The kernel, at the heart of all Linux systems, is developed and released under the GNU General Public License (GPL) and its source code is freely available to everyone (http://www.kernel.org).

Linux Kernel Version

- The first official release of Linux 1.0 was in March 1994.
- Just a year later, Linux 1.2 was released.
- Linux 2.0 arrived in June 1996.
- Linux 2.2 in January 1999.
- Linux 2.4 in January 2001.
- Linux 2.6 in December 2003.

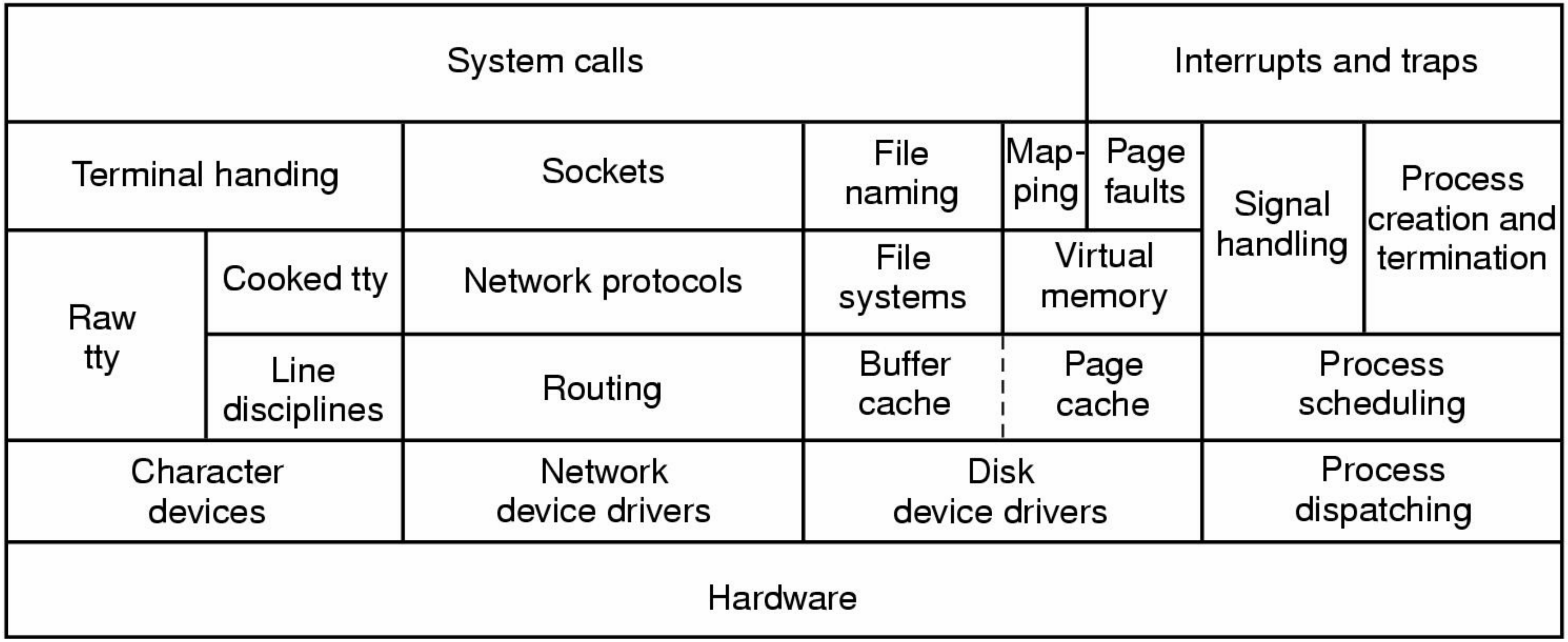
Linux - Why Popular ?

- Royalty-free – Open Source
- Strong networking support
- Standard (UNIX/POSIX) interface
- Growing number of embedded distributions
- Availability of support
- Modern OS (eg. memory management, kernel modules, etc.)

Linux Features

- Written in High Level Language ‘C’
- Monolithic
- Layered Approach
- Simple User Interface
- Hierarchical File System
- Dynamic Module Loading Support
- Pre-emptive Kernel

General Structure of Linux kernel



System calls

Interrupts and traps

Terminal handing

Sockets

File naming

Map-ping

Page faults

Signal handling

Process creation and termination

Raw tty

Cooked tty

Line disciplines

Network protocols

Routing

File systems

Virtual memory

Buffer cache

Page cache

Process scheduling

Character devices

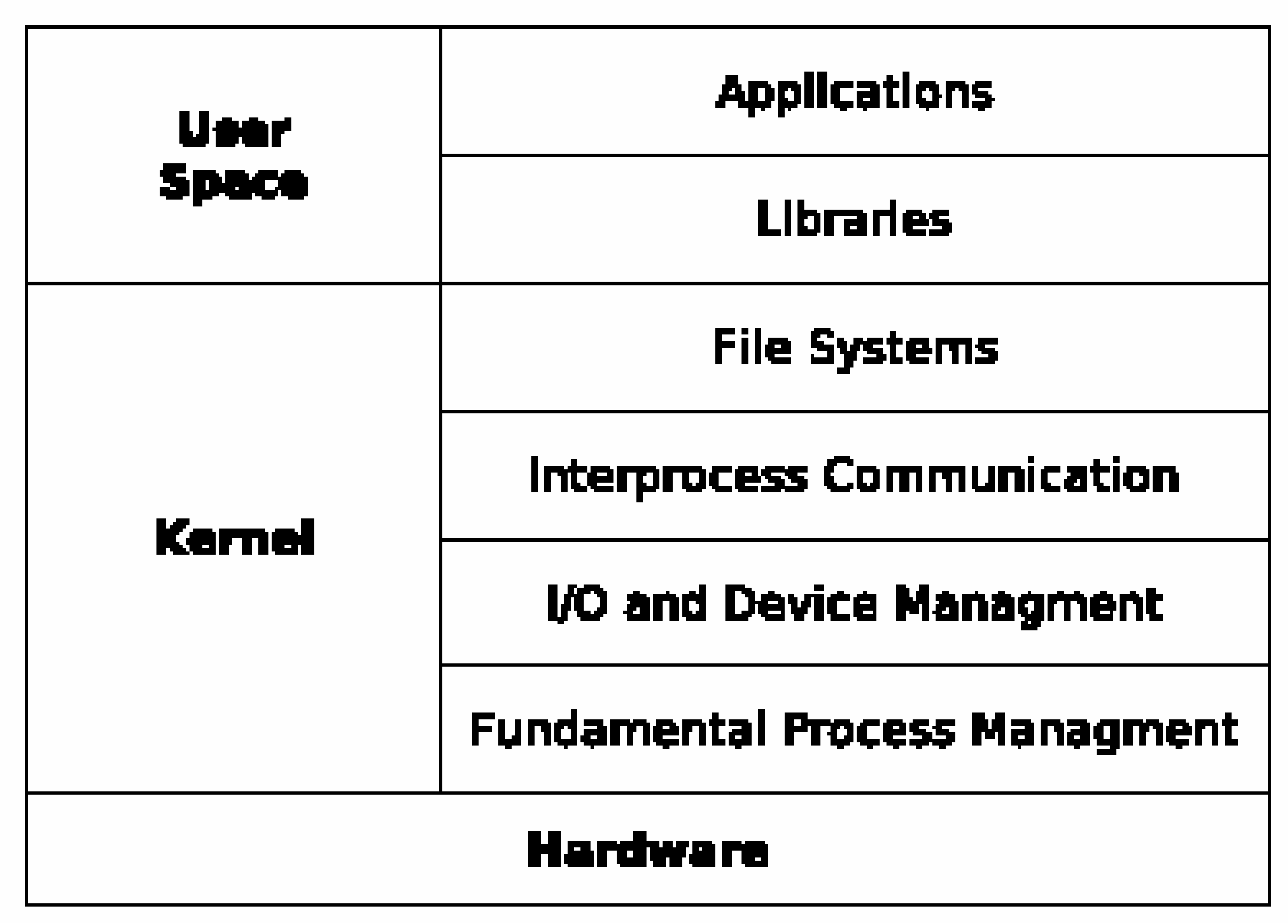
Network device drivers

Disk device drivers

Process dispatching

Hardware

Monolithic Kernel

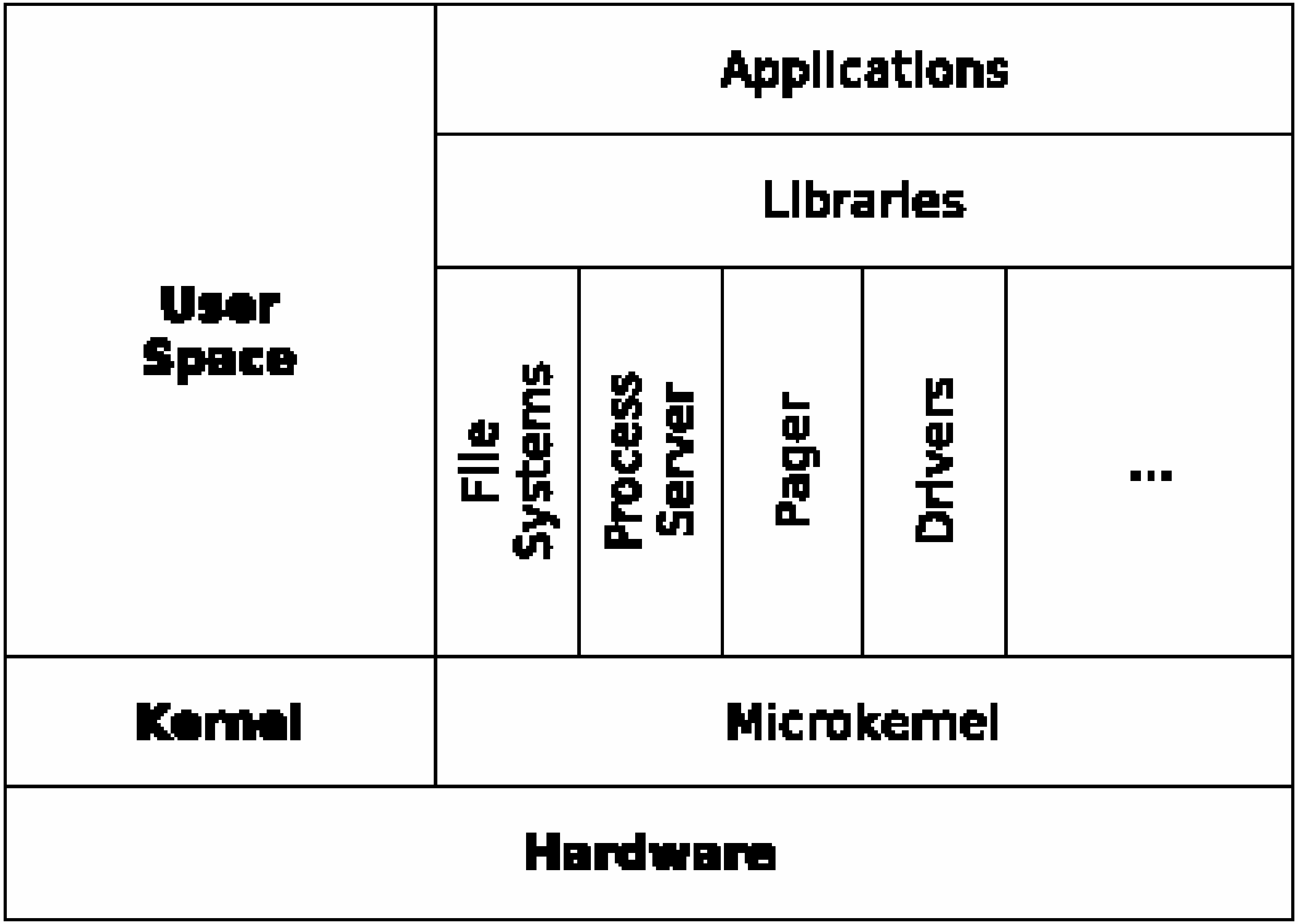


User Space	Applications
	Libraries
Kernel	File Systems
	Interprocess Communication
	I/O and Device Managment
	Fundamental Process Managment
Hardware	

Monolithic Kernel

- It runs every basic system service like process and memory management, interrupt handling and I/O communication, file system, etc. in kernel space
- It is constructed in a layered fashion, built up from the fundamental process management up to the interfaces to the rest of the operating system (libraries and on top of them the applications).
- The inclusion of all basic services in kernel space has three big drawbacks: the kernel size, lack of extensibility and the bad maintainability. Bug Fixing or the addition of new features means a recompilation of the whole kernel.

Micro Kernel



User Space

Applications

Libraries

File Systems

Process Server

Pager

Drivers

...

Kernel

Microkernel

Hardware

Micro Kernel

- To overcome these limitations of extensibility and maintainability, the idea of micro kernels appeared at the end of the 1980's.

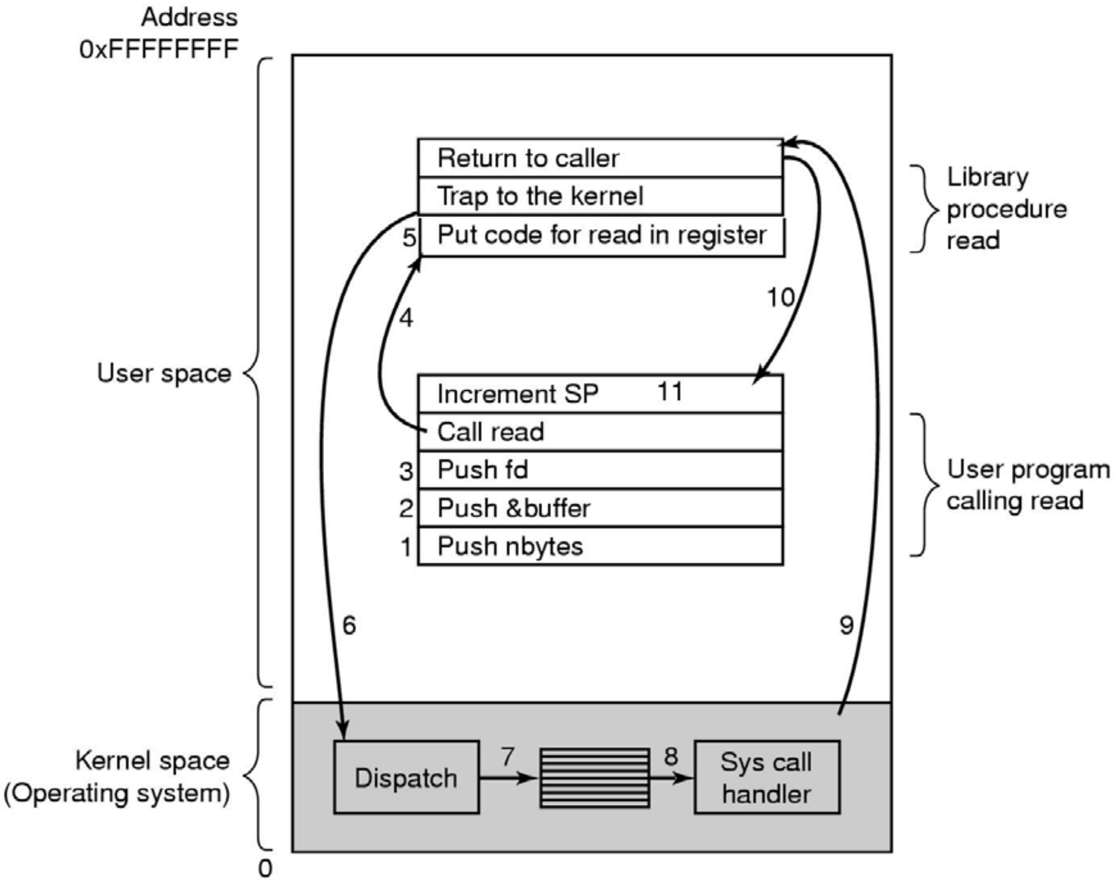
- The concept was to reduce the kernel to basic process communication and I/O control, and let the other system services reside in user space in form of normal processes (as so called servers).

- There is a server for managing memory issues, one server does process management, another one manages drivers, and so on.

System Calls

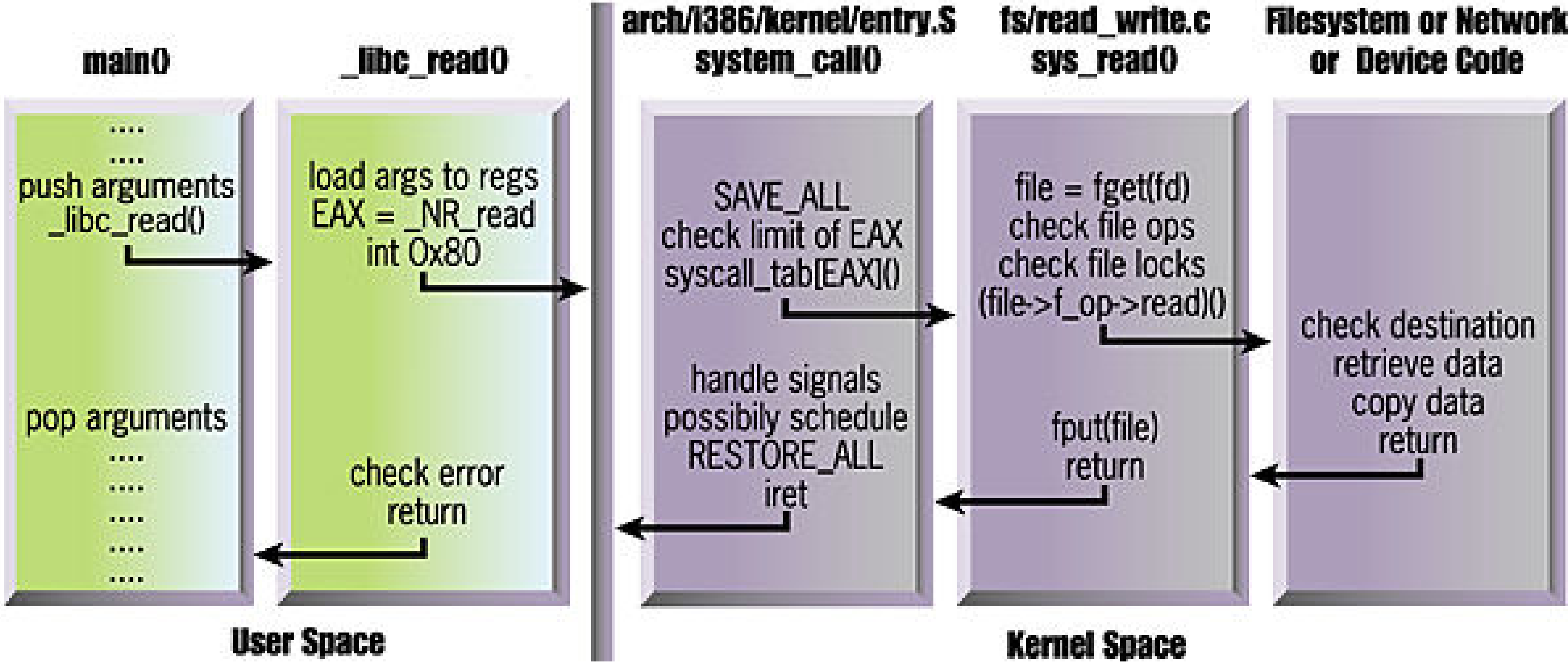
- $ vi /usr/src/linux/arch/i386/kernel/entry.S
- Library functions will call internally a system call. Every system call has a +ve integer number and will be executed in Kernel mode.
- For example - printf C library function calls internally write system call.
- Information about System Calls, refer man 2 <system call name>

Steps in Making a System Call



read (fd, buffer, nbytes);

Steps to Perform read ( )



main()

....
....
push arguments
_libc_read()

pop arguments
....
....
....
....
....

_libc_read()

load args to regs
EAX = _NR_read
int 0x80

check error
return

arch/i386/kernel/entry.S
system_call()

SAVE_ALL
check limit of EAX
syscall_tab[EAX]()

handle signals
possibily schedule
RESTORE_ALL
iret

fs/read_write.c
sys_read()

file = fget(fd)
check file ops
check file locks
(file->f_op->read)()

fput(file)
return

Filesystem or Network
or Device Code

check destination
retrieve data
copy data
return

User Space

Kernel Space

File Management

- The basic model of I/O system is a sequence of bytes (and there are no file format) that can be accessed either randomly, or sequentially.

- The I/O system is visible to a user process as a stream of bytes (I/O stream). A Linux process uses descriptors to refer I/O streams.

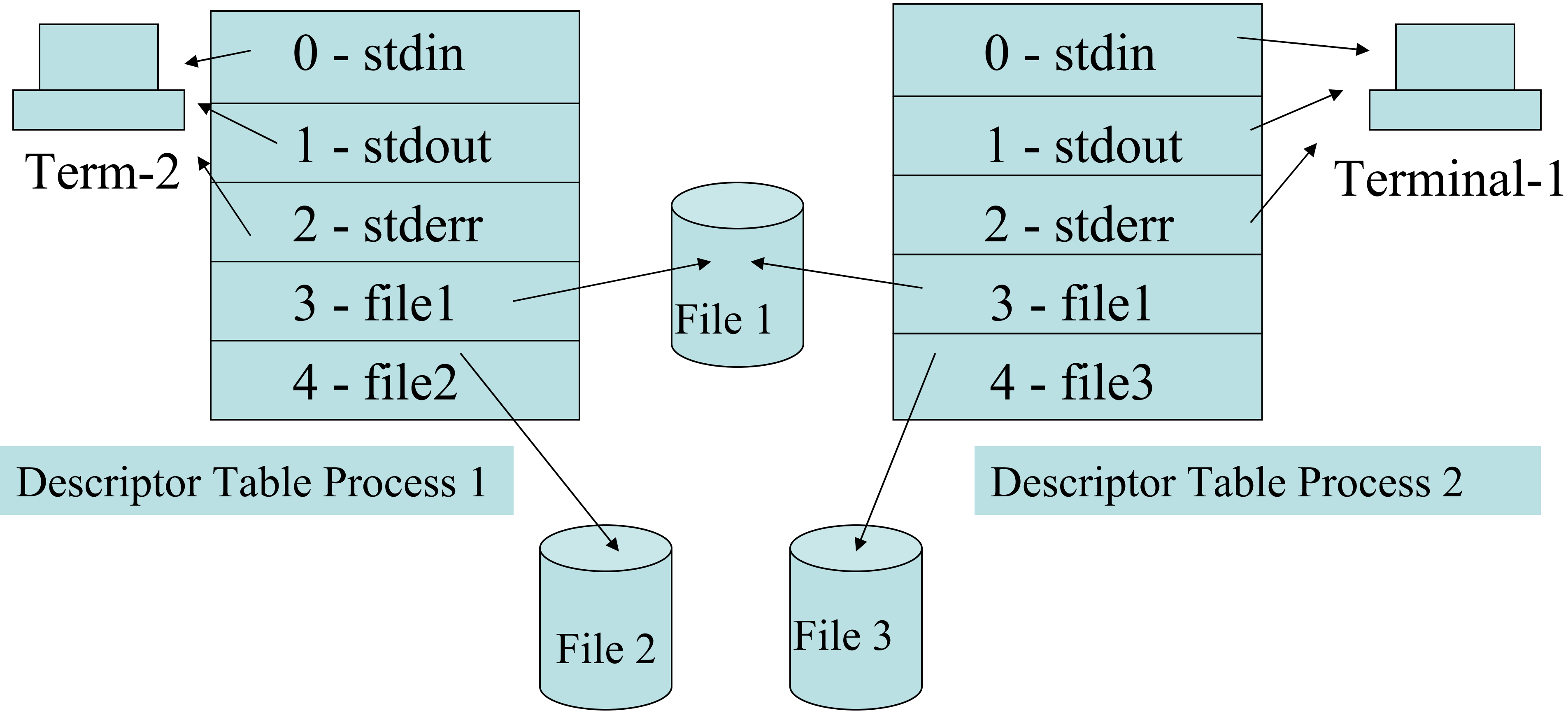
File Descriptor (fd)

0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file2

fd table

- The system calls related to the I/O system take a descriptor as an argument to handle a file.
- The descriptor is a positive integer number.
- If a file open is not successful, fd returns -1.
- Linux supports different types of files.

File Descriptor (fd)



Term-2

0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file2

Descriptor Table Process 1

File 1

File 2

File 3

0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file3

Terminal-1

Descriptor Table Process 2

File Management –File Systems

$***man fs*** lists the familiar file systems with brief description

The term file system refers to –

- Some code in the kernel that is activated in response to a program using file I/O system calls (such as open, read, write, close etc). In other words, file system facilitates file related system calls.

- A set of data structures (such as I-node table, mounted file systems table etc.) used to track the usage of a device.

File Systems

A file system enables storage of –

- names of ordinary files and directories
- the data contained in ordinary files and directories
- the names of device special files

File Systems - Creating

When a file system is created, Linux creates a number of blocks on that device. These blocks are

B	S	inode table	Data blocks

Boot block contains bootstrap code, which is used when the system is booting.

File Systems – Super block

- Each device also contains more than one copies of the super-block- as the super-block contains information that must be available to use the device.

- If the original super-block is corrupt, an alternate super-block can be used to mount the file system.

File Systems - Superblock

- The super-block contains info. such as:
  - a bitmap of blocks on the device, each bit specifies whether a block is free or in use.
  - the size of a data block
  - the count of entries in the I-node table
  - the date and time when the file system was last checked
  - the date and time when the file system was last backed up

File Systems – I-node table

- The I-node table contains an entry for each file stored in the file system. The total number of I-nodes in a file system determine the number of files that a file system can contain.

- When a file system is created, the I-node for the root directory of the file system is automatically created.

File Systems – I-node table

Each I-node contains following info:

- file owner UID and GID
- file type and access permissions
- date/time the file was created, last modified, last accessed
- size of the file
- number of hard links to the file
- Each I-node entry can track a very large file

Device Special Files

A device special file describes following characteristics of a device :

- Device name
- Device type (block device or character device)
- Major device number
- Minor device number

- Each file system must be mounted before it can be used. Normally, all file systems are mounted during system startup.

Device Special Files

- Each file is located on a file system. Each file system is created on a device, and associated with a device special file.

- Therefore, when you use a file, UNIX can find out which device special file is associated with that file and send your request to a particular device driver.

FS - Mounting

- The “dev” directory contains names of each device special file. Therefore each *device special file name* is also stored in a device.
- A file system is mounted typically under an empty directory. This directory is called the “mount point” for the file system.

FS- Mounting

- You can use mount command to find how many file systems are mounted, and what is the mount point for each file system :

```
$ mount
/dev/hda2 on / type ext2 (rw)
none on /proc type proc (rw)
```

FS– Internal Routines

- The file system contains a number of internal support routines that are used for accessing a file.
  - namei( )
  - iget() / iput( )
  - bread( )
  - bwrite( )
  - getblk( )

Buffer Cache

- The file system also maintains a *buffer cache*.
- The buffer cache is stored in physical memory (non-paged memory).
- The buffer cache is used to store any data that is read from or written to a **block-device** such as a hard-disk, floppy disk or CD-ROM.

Buffer Cache

If data is not present in buffer cache:

- the system allocates a free buffer in buffer cache
- reads the data from the disk
- stores the data in the buffer cache.

If there is no free buffer in the buffer cache:

- the system selects a used buffer
- writes it to the disk
- marks the buffer as free
- allocates it for the requesting process.

Buffer Cache

- While all this is going on, the requesting process is put to wait state.
- Once a free buffer is allocated and data is read from disk into buffer cache, the process is resumed.
- A process can use the *sync( )* system call to tell the system that any changes made by itself in the buffer cache must be written to the disk.

File I/O –System Calls

System calls for file I/O

– open	- To open or create a file
– read,write	- To perform file I/O
– lseek	- To seek to a location in the file
– close	- To close an open file
– dup,dup2	- To duplicate the file descriptors
– fcntl	- File control
– stat	- To obtain information about a file

Lab Exercise –Day 1

1.Write a program to copy the content of a file to another using read and write system calls.

2.Write a program to open a file in read only mode. Read line by line from the file. Display each line as it is read. Close the file when end-of-file is reached.

3.Write a program to read from the standard input and display on standard output.

4. Using lstat system call display the contents of inode no , block size of a file……..

5.Using lstat system call check the type of file.

LINUX Internals

DAY -2

Process Management

Introduction to Process

- A process can be thought of as a program in execution

- Process also include PC and all CPU registers as well as the process stacks containing temporary data

- During the lifetime of a process it will use many resources

Introduction to Process

- Os should keep track of the processes to ensure that the system resources are shared fairly.
- Most precious resource is CPU. Since UNIX is a multiprocessing OS, its main objective is to have maximum CPU utilization

Dual Modes

- In order to run Linux, the computer hardware must provide two modes of execution: User and Kernel.
- Each process has virtual address space, references to virtual memory are translated to physical memory locations using set of address translation maps
- When current process yields CPU to another process (a context switch), the kernel loads these registers with pointers to translation of new process

Per-process Objects

- There are two important per-process objects

  * uarea (user area) – is a data structure that contains information about a process of interest to the kernel, such as a table of files opened by the process, identification information, and saved values of the process registers when the process is not running

  * kernel stack – to keep track of its function call sequence when executing in the kernel

  Both u area and kernel stack, while being per-process entities in the process space, are owned by the kernel

Context of a Process

- The UNIX kernel is reentrant
- Kernel functions may execute either in process context or in system context
- User code runs in user mode and process context, and can access only the process space

Context of a Process

- System calls and exceptions are handled in kernel mode but in process context, and may access process and system space

- Interrupts and system wide tasks are handled in kernel mode and system context, and must only access system space

Process Structure

- In order to manage the processes in the system, each process is represented by a *task_struct data structure*

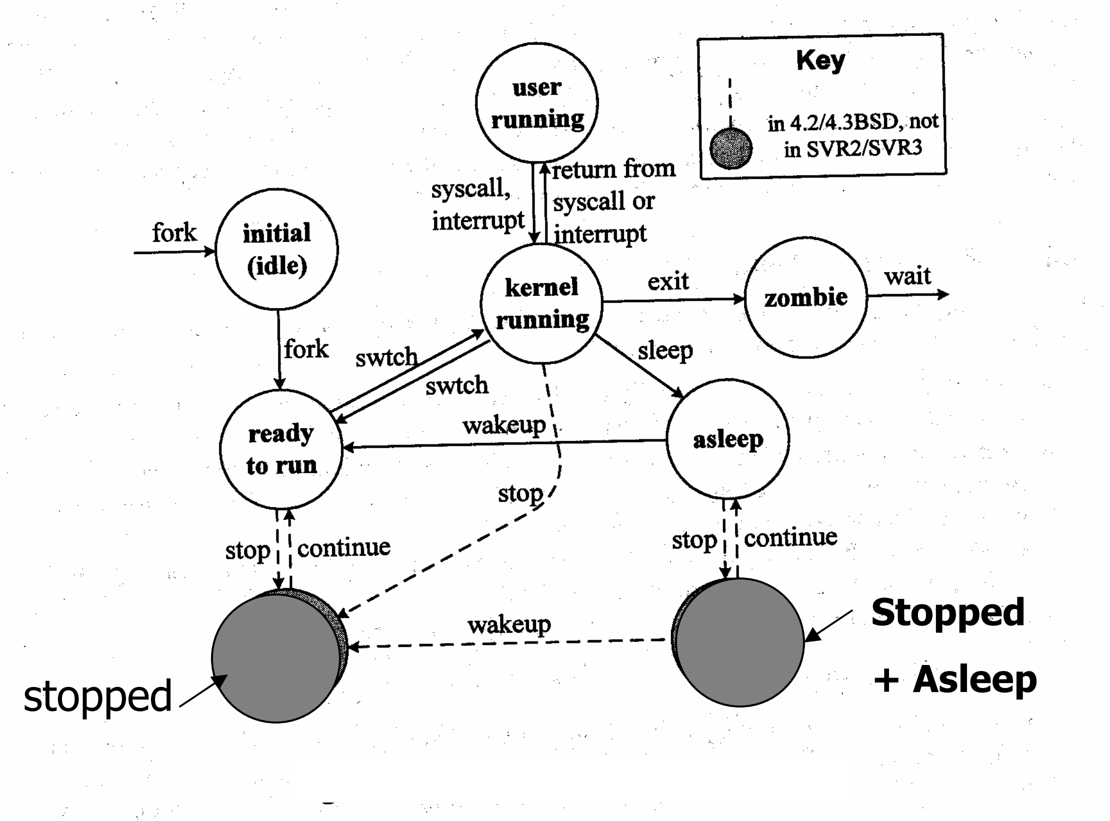
- The task vector is an array of pointers to every task_struct data structure in the system.

- task_struct is quite large and complex

Process State

As a process executes it changes *state* according to its circumstances. Standard UNIX processes have the following states:

- Ready
- Running
- Wait
- stopped
- Zombie
- Exit



stopped

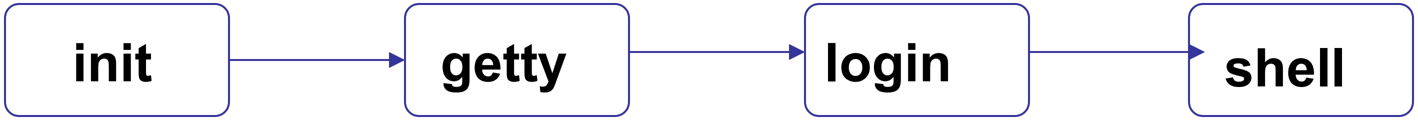
Stopped + Asleep

Identifiers

- Every process in the system has a process identifier.
- Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system
- ppid, pid, uid, gid, euid, egid

init - shell

- In Linux no process is independent of any other process
- Every process in the system, except the initial process has a parent process
- New processes are not created, they are copied, or rather *cloned* from previous processes



Times and Timers

- The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime
- Each clock tick, the kernel updates the amount of time that the current process has spent in system and in user mode
- UNIX also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire

Process Scheduling

- *scheduler* that must select the most deserving process to run out of all of the processes in the ready to run queue. The traditional UNIX scheduler uses preemptive round-robin scheduling
- Scheduling priorities have integer values between 0 and 140, with smaller numbers meaning higher priorities

Process Scheduling

- For the scheduler keeps information in the *task_struct* for each process
- *policy* This is the scheduling policy that will be applied to this process.
- *priority* This is the priority that the scheduler will give to this process.
- *rt_priority* Linux supports real time processes and these are scheduled to have a higher priority than all of the other non-real time processes in system.
- *counter* This is the amount of time that this process is allowed to run for.

Creating a new process

- *fork( )* system call creates a new process
- All statements after the *fork( )* system call in your program are executed by two processes
- If *fork ( )* returns 0 it is a child process else if > 0 it is parent process else (-1) error
- A parent process can use the *wait( )* system call to wait for the exit of any child process

Process Creation

- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address space
  - Child duplicate of parent
  - Child has a program loaded into it

exec – to run a program

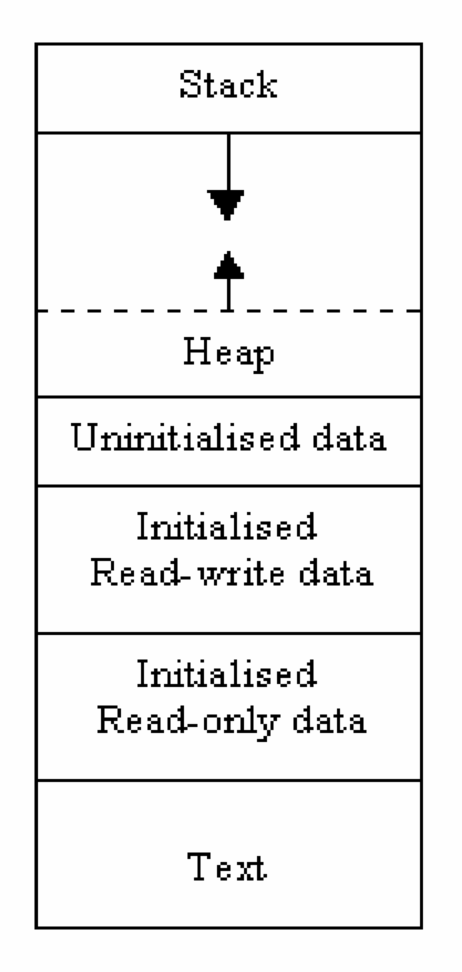
To run a new program in a process, use one of the "exec" family of calls

- pathname of the program to run
- name of the program
- each parameter to the program
- (char *)0 as the last parameter to specify end of parameter list

fork ( ) demo

Process Image

User context



Stack

Heap

Uninitialised data

Initialised
Read-write data

Initialised
Read-only data

Text

Kernel context

Kernel data

$ size a.out (man size)

```
 text    data     bss     dec     hex    filename
  920     268      24    1212     4bc     a.out
```

Signal Handling

Signals -Introduction

- **Signals are requests sent to a process, causing it to divert its execution to do something else**

- **Signals are software interrupts**

- **Signals provide a way of handling synchronous or asynchronous events depends on its nature**

Signals -Introduction

- **Each signal has an integer number**
- **Symbolic name is defined in the file /usr/include/bits/signum.h**
- **Refer $man 7 signal**
- **$kill –l**

List of Signals

```
[raju@linux62 raju]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS
```

Signal Generation

Kernel generates signals to process in response to various events. Some major sources of signals are:

- Exceptions
- other process
- Terminal interrupts
- Job control
- Quotas
- Notifications
- Alarms

Signal Handling

- **When the signal is sent to the process, the operating system stops the execution of the process, and "forces" it to call the signal handler function**

- **Each signal has a default signal handler, which is a function that gets called when the process receives that signal**

Signal Handling

- **Catch the Signal: instead of executing a default signal handler, the control should execute a given signal handler.**

- **Two signals SIGSTOP and SIGKILL cannot catch, they cause the process to terminate immediately. This is useful when debugging programs whose behavior depends on timing**.

Signal Vs Interrupt

- **Signals are very similar to hardware interrupts in their behavior**

- **The difference is that while interrupts are sent to the operating system by the hardware. For example: clock interrupt, page fault, device interrupt**

- **Signals are sent to the process by the operating system, or by other processes**

Signal System Call

- **signal (int signum, sighandler_t handler);**
- **The signal() system call is used to modify a default action of a specified signal**
- **signal() accepts a signal number and a pointer to a signal handler function**

Kill –System Call

- **Sending a given signal to the specified pid.**

- **int kill(pid_t pid, int sig);**

- **The kill ( ) system call is used to send a given signal to the specified process(es). It accepts two arguments, signal and pid**

Pre-defined Handlers

```
signal (SIGINT, (void *) our_handler);

kill (SIGINT, pid );
```

SIG_IGN

Causes the process to ignore the specified signal.

SIG_DFL

Causes the system to set the default signal handler for the given signal

Lab Exercise –Day 2

1. Write a C program to create a child process and let the two process update the same file ?

2. Create child process and let the child execute one of your C program?

3. Write a C program to create zombie and orphan process?

4. Using system calls wait for a specific child in a parent process?

5. Write a C program to handle divide by zero , Ctrl C and SIGSEGV.

6. Write a C program to send SIGINT from one process to other.

LINUX Internals

Day 3

LINUX Internals

User Level Threads

POSIX Threads

- Thread is a sequential flow of control through a program
- If a thread is created, it will execute a specified function

  Two type of threading:
  1. Single Threading
  2. Multi threading

POSIX Threads

All threads within a process share

1. process instructions
2. address space, data
3. Open files (example: file descriptors)
4. Signal Handlers
5. Current working directory, uid and gid

POSIX Threads

Each thread has its own

1. Thread id

2. set of registers, pc, sp

3. stack (for local variables and return addresses)

POSIX Threads

Advantages of Threads

1. It takes less time to create a new thread in a process

2. It takes less time to terminate a thread than a process

3. It takes less time to switch between two threads within the same process

4. Communication between threads are easier.

POSIX Threads

There are two broad categories of thread implementation:

1. User level Threads (ULT)

2. Kernel level threads (or kernel-supported threads or Light weight processes)

POSIX Threads

#include <pthread.h>

Thread management is done by the application and the kernel is not aware of the existence of threads

Thread library contains code for creating and destroying threads, passing messages and data between threads, for scheduling thread execution and for saving and restoring thread contexts

POSIX Threads

- This thread application are allocated to a single process managed by the kernel

- All the activity takes place in user space and within a single process. kernel is unaware of this activity; the kernel continues to schedule the process as a unit and assigns a single execution state to that process

POSIX Threads

- Advantages:

  Thread switching does not require kernel mode, Scheduling can be application specific and can run on any OS

- Disadvantage:

  When it executes a system call, not only is that thread is blocked, but all the threads within the process are blocked

Kernel Threads

- Kernel Level Threads

  Thread management is done by the kernel

  Advantage: If one thread in a process is blocked, kernel can schedule another thread of the same process.

  Disadvantage: Transfer of control from one thread to another within the same process requires a mode switch to the kernel

Threads –Comparison

- Comparison of creation and synchronization time of ULT, LWP & Process

	CREATION TIME (μ sec)	SYNCHRONIZATION TIME (μ sec)
ULT	52	66
LWP	350	390
Process	1700	200

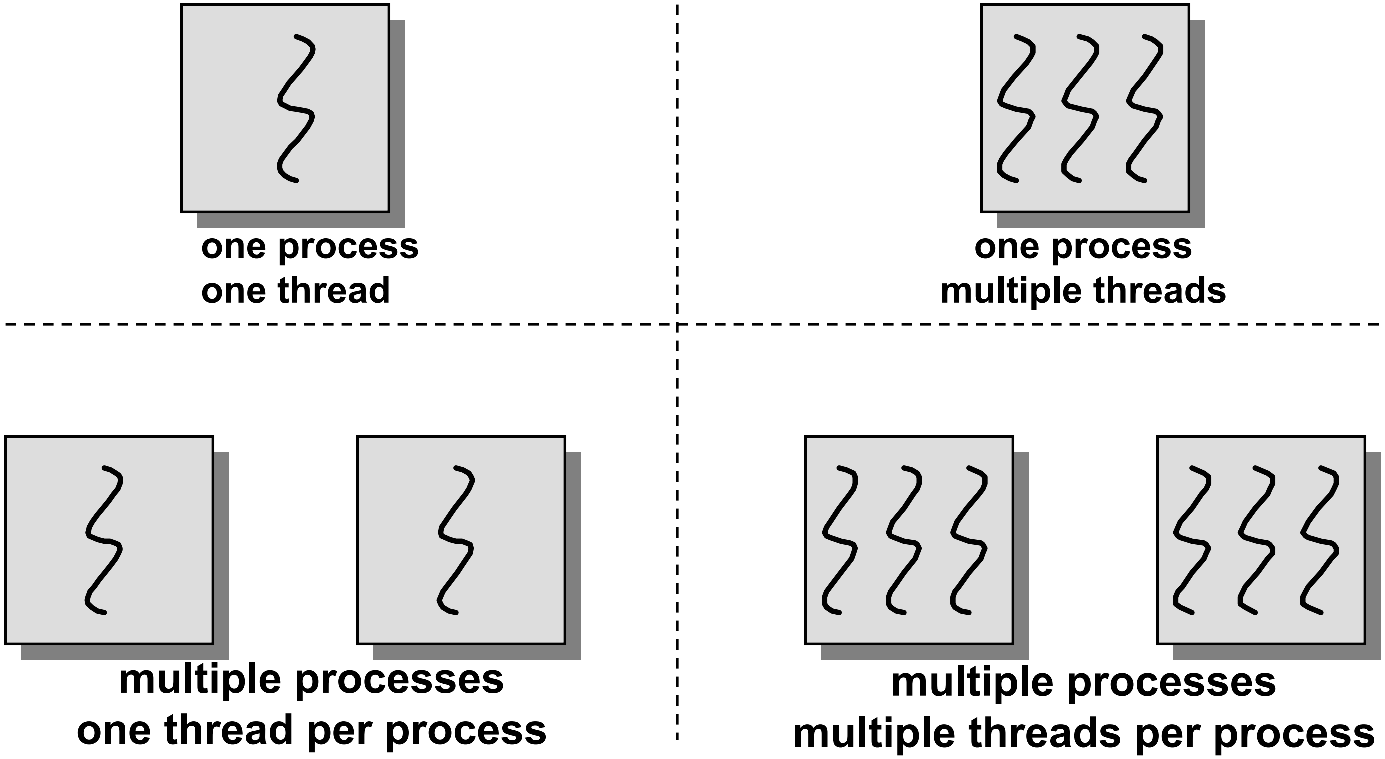
Threads -Applications

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- use fewer system resources
- Specific applications in uniprocessor machines

Applications

- A file server on a LAN
- Graphical User Interfaces (GUIs)
- web applications

Threads –OS implementation



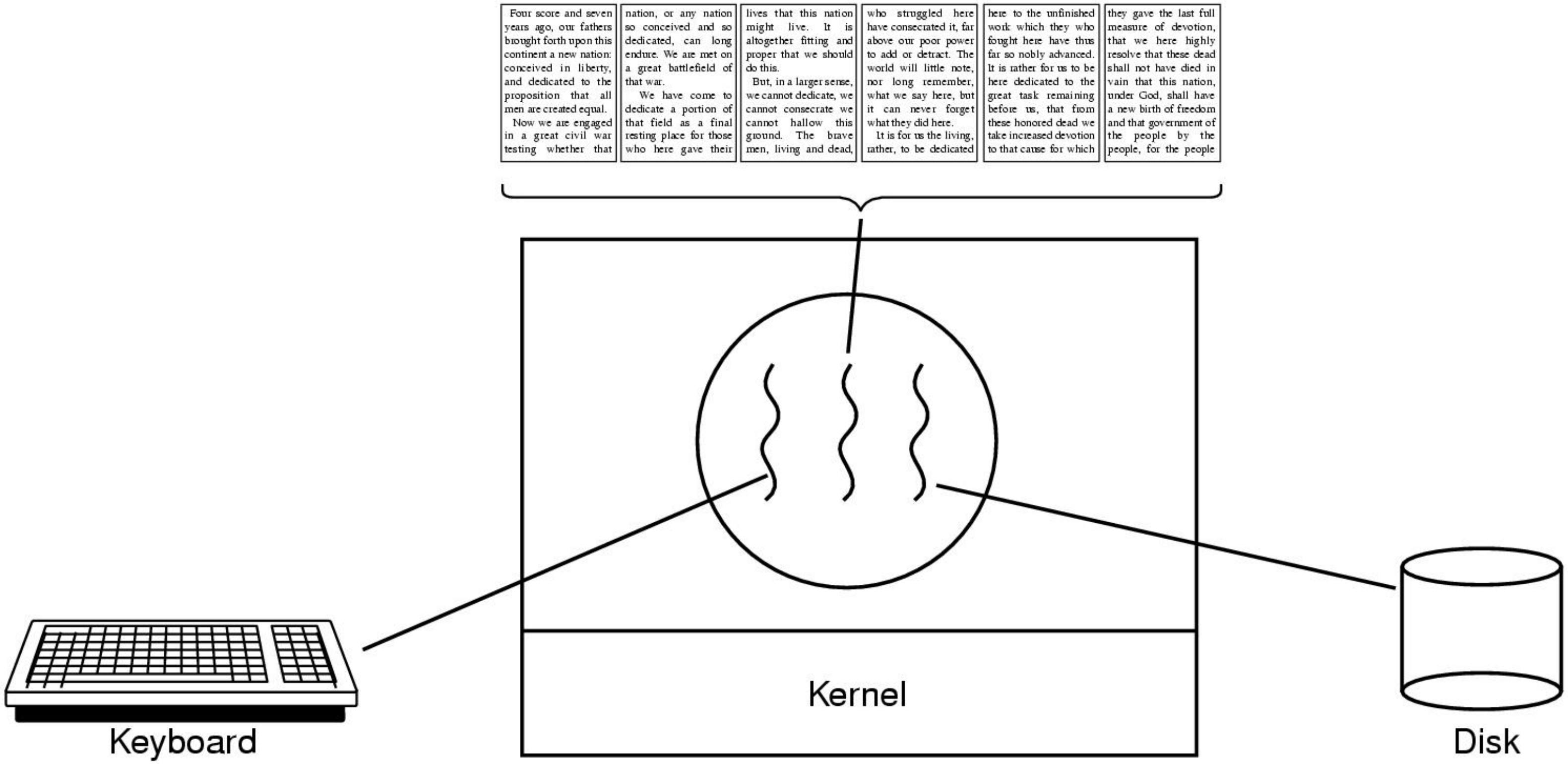
one process
one thread

one process
multiple threads

multiple processes
one thread per process

multiple processes
multiple threads per process

Thread Usage



Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.

It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people, for the people

A word processor with three threads

Hello Thread Example

```
#include <pthread.h>
void thread_function (void) {
      printf (“ Hello POSIX Thread\n”);
      }
main ( ) {
      pthread_t mythread;
      pthread_create ( &mythread, NULL, thread_function, NULL);
 }
```

```
$cc thread.c -lpthread
```

Primitive IPC

IPC - Introduction

- Traditionally this term described different ways of message passing between different processes
- A complex programming environment often multiple processes must communicate with each other and share some resources and information

IPC - Introduction

- Interprocess interactions have several distinct purposes:

* Data transfer * Sharing data

* Event notification * Resource sharing

* Process control

IPC Mechanisms

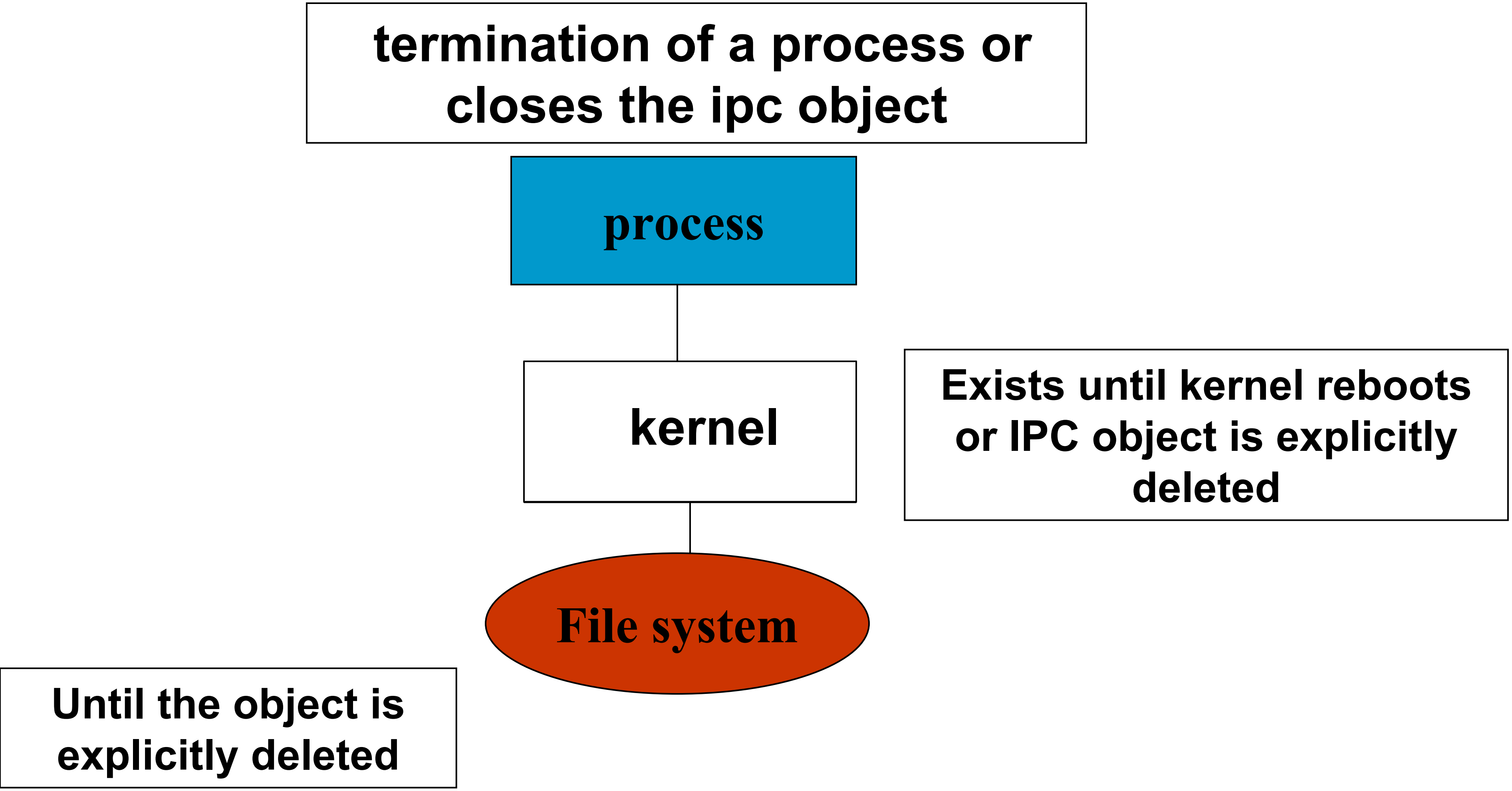
Primitive IPC

- Pipe
- FIFO

System V IPC

- Message Queues
- Shared Memory
- Semaphores

Persistence of IPC Objects



termination of a process or closes the ipc object

process

kernel

Exists until kernel reboots or IPC object is explicitly deleted

File system

Until the object is explicitly deleted

Unnamed Pipe

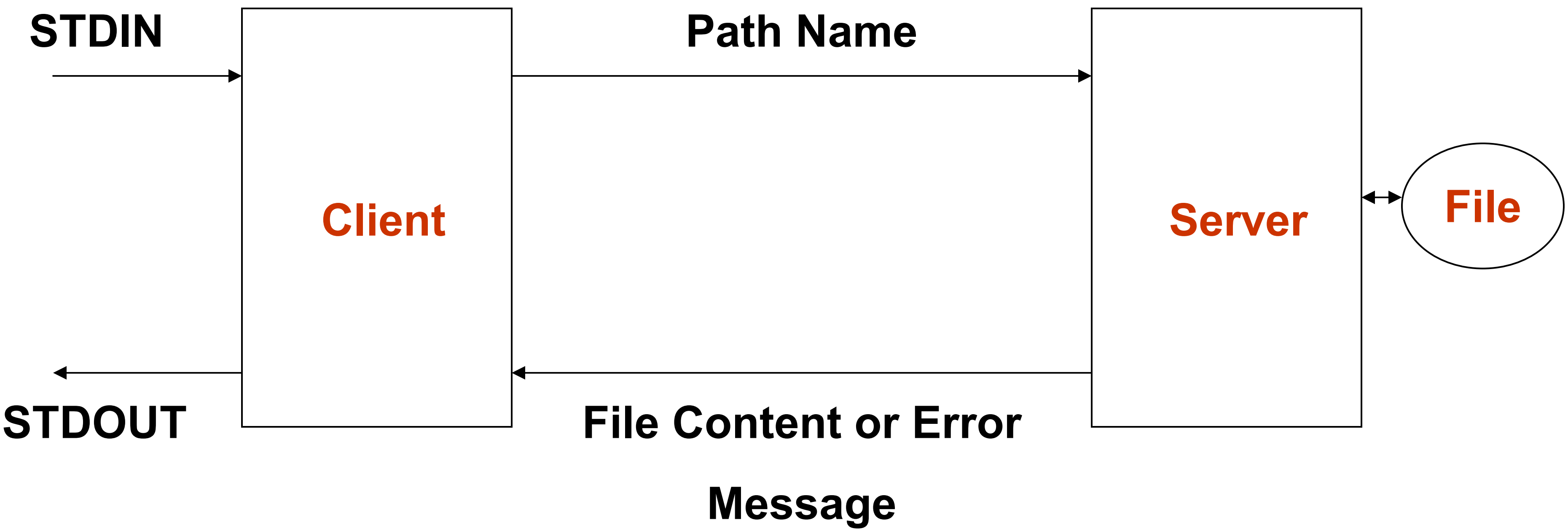
PIPE -Introduction

A pipe is a set of two file descriptors. A pipe allows two related processes to communicate by sending some data from one process to another process.

The processes must co-operate and assume the role of a reader or a writer with respect to a specific pipe.
Data is passed in order.
Data doesn’t get lost in middle.
Zero buffering capacity

PIPE : Example

Client - Server



pipe –System Call

- A process (parent process) creates a pipe using the pipe() system call and passes an array of two integers (file descriptors).

- Another process can use the pipe, provided it has access to the above file descriptors. This is possible if another process is the child of the first process, and it was created after the pipe was created.

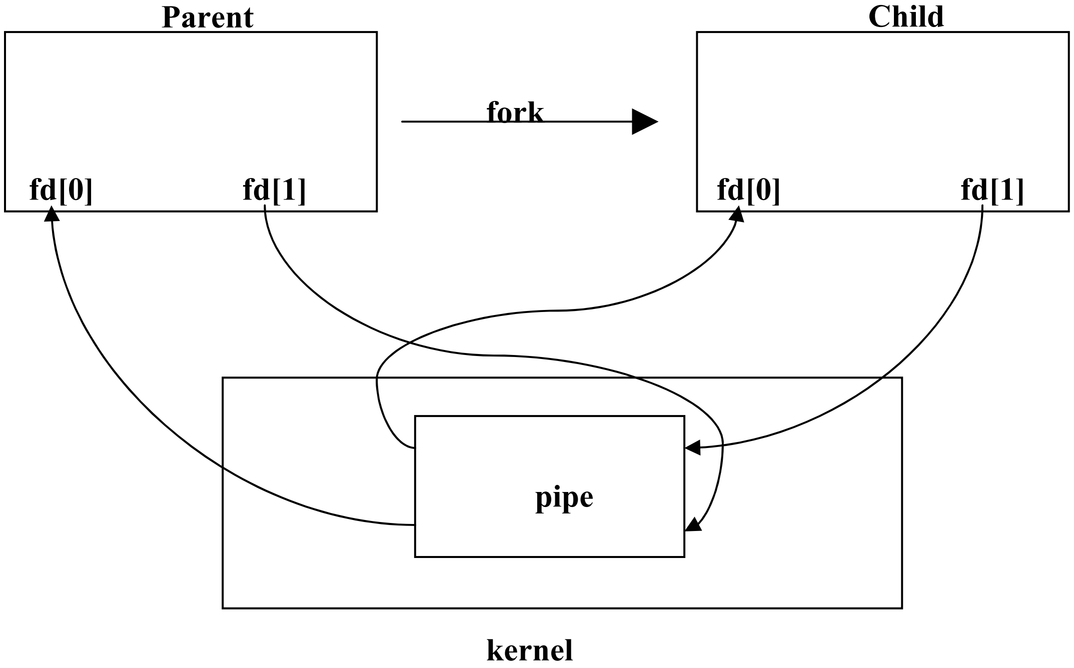
- int fd[2];   pipe (fd); fd[0] and fd[1]

PIPE –unidirectional flow

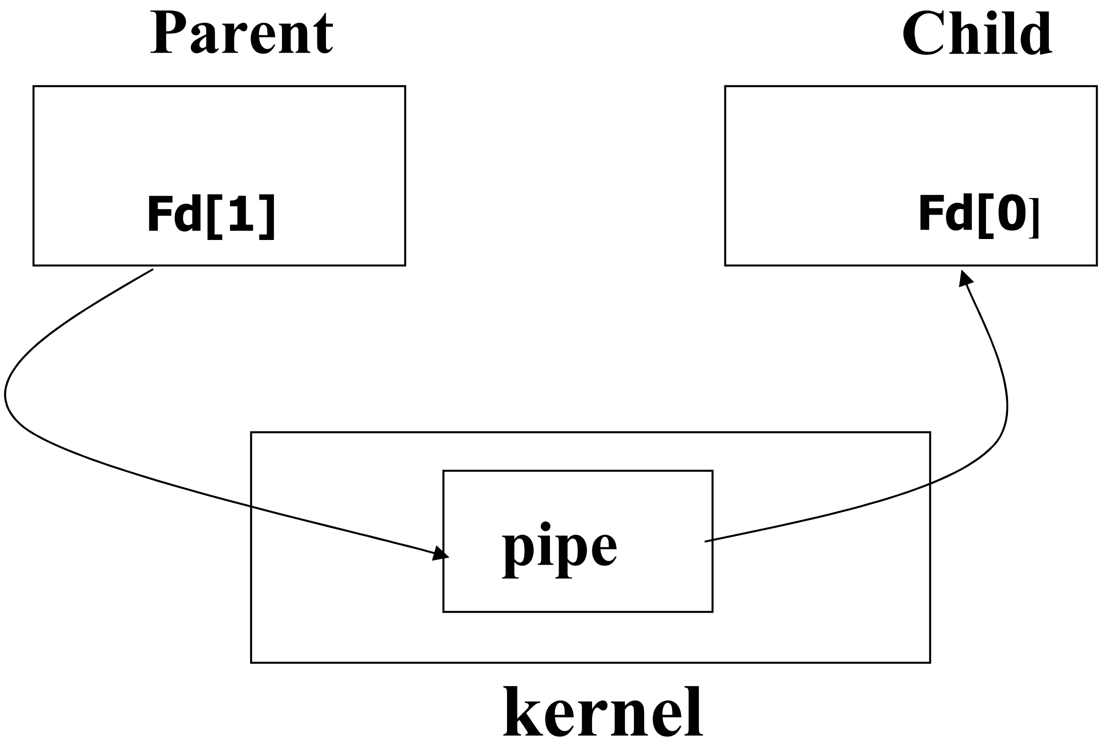
- Create pipe (fd[0] and fd[1])
- Parent closes read end of pipe (fd[0])
- Child closes write end of pipe (fd[1]).
- The read and write system calls are blocking calls . Meaning that the reading process will wait when there is no data in the pipe and the writing process will wait when there is no reading process.

PIPE - visualization

Pipe in a single process after fork



PIPE - visualization



Pipe from parent to child

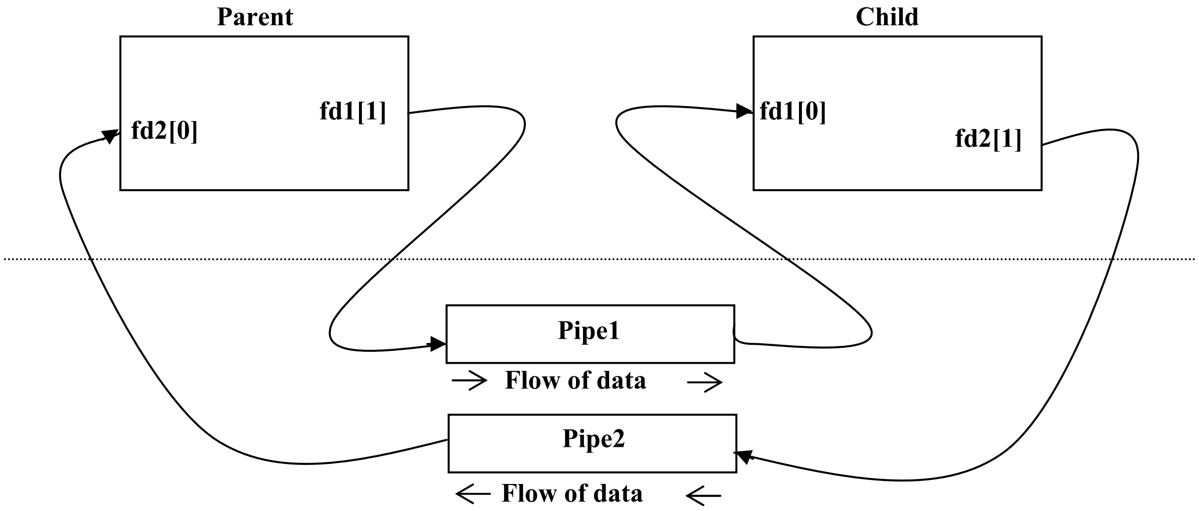
PIPE – bi directional flow

- Create pipe1 (fd1[0] and fd1[1]
  create pipe2 (fd2[0] and fd2[1])

- parent closes read end of pipe1 (fd1[0])
- parent closes write end of pipe2 (fd2[1])

- child closes write end of pipe1 (fd1[1])
- child closes read end of pipe2 (fd2[0])

PIPE – bi directional flow



FIFO – Named Pipe

FIFO - Introduction

Pipes were the first widely used form of IPC, available both within programs and from the shell

The problem with pipes is that they are usable only between processes that have a common ancestor (i.e., a parent-child relationship)

If we want to communicate between unrelated processes, we have to use FIFO – the named pipe

FIFO Basic concepts

- Named pipe works much like a regular pipe, but does have some noticeable differences
- Named pipes exist as a device special file in the file system
- Processes of different ancestry can share data through a named pipe
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use

Creating FIFO

Methods of creating named pipes from the shell

$mknod MYFIFO p (OR) $mkfifo MYFIFO

To create a FIFO within a C program, we can make use of the mknod() system call:

int mknod (char *pathname, mode_t mode, dev_t dev);

- ex: *mknod ("/tmp/MYFIFO", S_IFIFO*|*0666, 0);*

FIFO

- Once a named pipe is created each process has to open the named pipe using the open() system call

- One process can open the named pipe for reading, the other can open it for writing etc. To read from named pipe, process can use read() system call and to write to it, it can use the write() system call

FIFO -Limitation

- The system-imposed limits on pipes and FIFOs are:

OPEN_MAX : The maximum no. of descriptors opened at any time by a process

PIPE_BUF : The max. amount of data that can be written to a pipe or FIFO atomically

FIFO - Limitations

- Half duplex; cannot use across network
- They are less secure than pipes, since any process with right privileges can access them
- While reading data is removed from the pipe, pipe cannot be used for broadcast data to multiple receivers
- Data in a pipe is treated as a byte stream and has no knowledge about message boundaries

Lab Exercise –Day 3

1. Write a C program to perform communication between first child and second child of the parent using pipes.
2. Establish bi- directional communication between parent and child using pipes
3. Write a C program to implement ls -l|wc -l using pipes.
4. Using FIFOS establish communication between unrelated processes.

LINUX Internals

Day 4

System V IPC

Sys V IPC - Introduction

Pipes and FIFOs do not satisfy the IPC requirements of many applications

- **System V IPC provided three mechanisms namely**

  message queues

  shared memory

  Semaphores,

Sys V IPC - Introduction

- **The IPCs objects are created in the kernel level**
- **Process can acquire the resource by making a shmget, semget or msgget system call**
- **Several control commands can be issued by control system call (shmctl, semctl or msgctl)**

Sys V IPC - Introduction

- The ipc_perm structure contains the common attributes of the resources (the key, creator and owner IDs and permission)

- Each IPC resource must be explicitly deallocated by the IPC_RMID command from the command line or using (- ctl) statement to delete the entry from the kernel otherwise, it will persist until reboot the system

Sys V IPC - Introduction

- To get information about the IPCs entries: use $ipcs command from the shell

- To get more theoretical information about IPCs refer

  $man 5,2 ipc,

  $man 8 ipcrm, ipcs

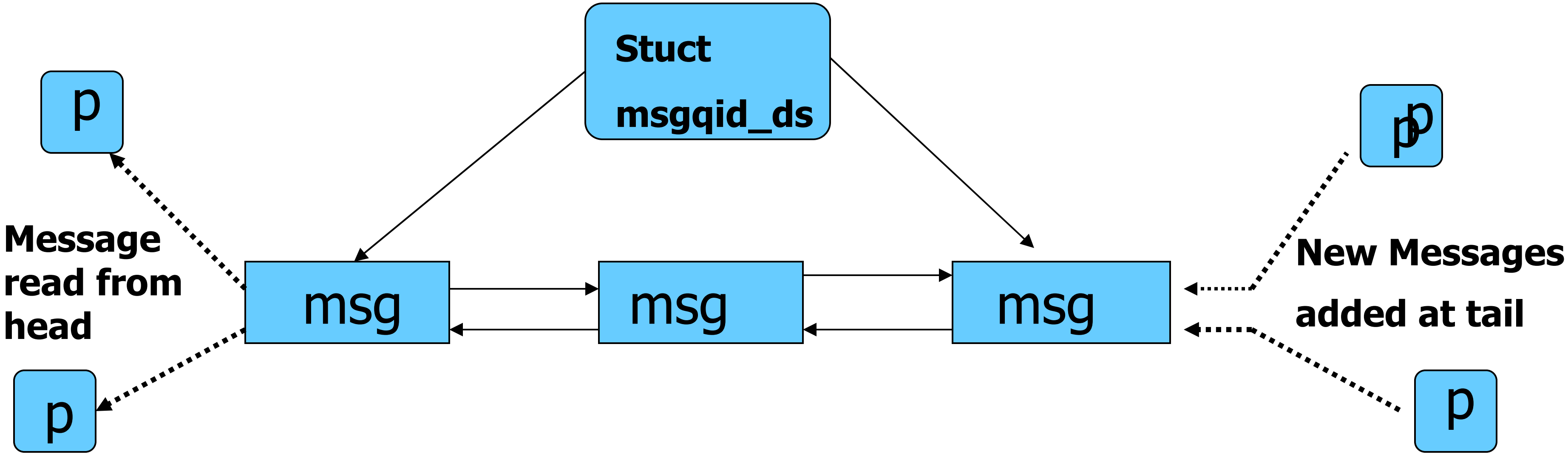
Message Queues

Message queues

msqid xxx

mtype x_1	msg text
mtype x_2	msg text
mtype x_3	msg text
mtype x_4	msg text
mtype x_5	msg text
---------	
mtype x_n	msg text

MQ - Internal



Stuct

msgqid_ds

Message read from head

New Messages

added at tail

Message Queue id

- msgget ( ) system call will create a message queue and it returns to the message queue id

  msqid = int msgget (key_t key, int msgflg);

- key is the system wide unique identifier

  key_t ftok (const char *path, int id);

- msgflg is the permission of the queue and it should be OR’d with IPC_CREAT flag for IPC objects

Message Q structure

- Each message is made of two parts, Which is defined in template structure struct msgbuf, as defined in <sys/msg.h>

```
    struct msgbuf {
                    long mtype;
                    char mtext[1];
            };
```

- We can use our own structure but the first member of the structure should be long int

mq - sending

- Message send system call is used to pass the message to the queue

- **msgsnd (int msqid, const void *msgp, size_t msgsz, int msgflg);**

- msgflg allows user to set optional parameters either zero or IPC_NOWAIT

mq - receiving

msgrcv ( ) system call is used to retrieve the message from the queue

```
int msgrcv (int msqid, void *msgp, size_t msgsz,
                                    long mtype, int msgflg);
```

If mtype is

0 - retrive the next message in the queue

+ve - get the mesg with an mtype equal to the specified msgtyp

mq - pseudo code

- key = ftok (“.“, ‘a‘);
- msqid = msgget (key, IPC_CREAT|0666);
- msgsnd (msqid, &struct, sizeof (struct), 0);
- msgrcv (msqid, &struct, sizeof (struct), mtype, 0);
- msgctl (msgid, IPC_RMID, NULL);

- $ipcrm msg msqid

mq - Limitations

```
$ipcs –lq

------ Messages: Limits --------

        max queues system wide = 16
        max size of message (bytes) = 8192
        default max size of queue (bytes) = 16384
```

Shared Memory

Shared Memory

- Shared memory, as the name implies, allows two or more processes, which have the appropriate permissions, to read and/or write to the same area of memory
- The distinguishing features of shared memory as an IPC mechanism are speed, flexibility and ease of use
- Example of a shared memory: editors or word processors in multi user environment

Shared Memory

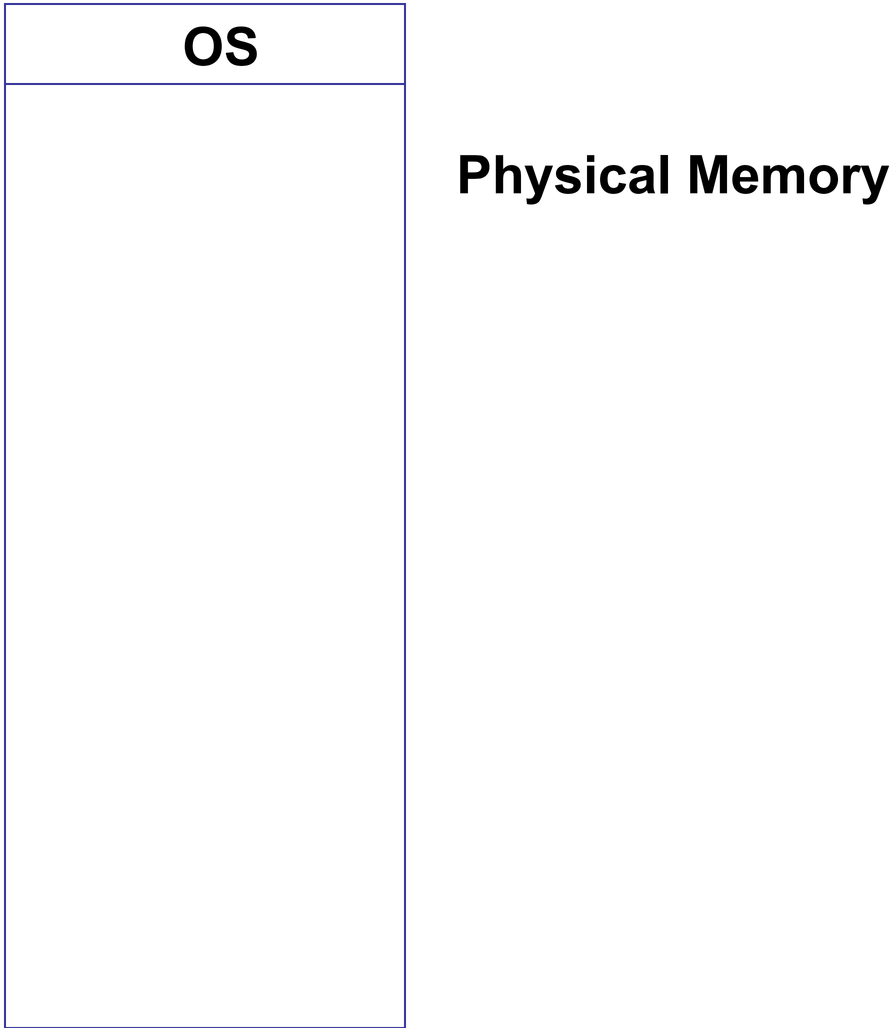
- Shared memory is a much faster method of communication than either semaphores or message queues
- Data does not need to be copied to a kernel buffer and back again. Accessing shared memory takes as much time as a normal memory access
- Using shared memory is quite easy. After a shared memory segment is set up, it is manipulated exactly like any other memory area

Shared Memory

The steps involved to create shared memory are

- – Creating shared memory
- – Connecting to the memory & obtaining a pointer to the memory
- – Reading/Writing & changing access mode to the memory
- – Detaching from memory
- – Deleting the shared segment

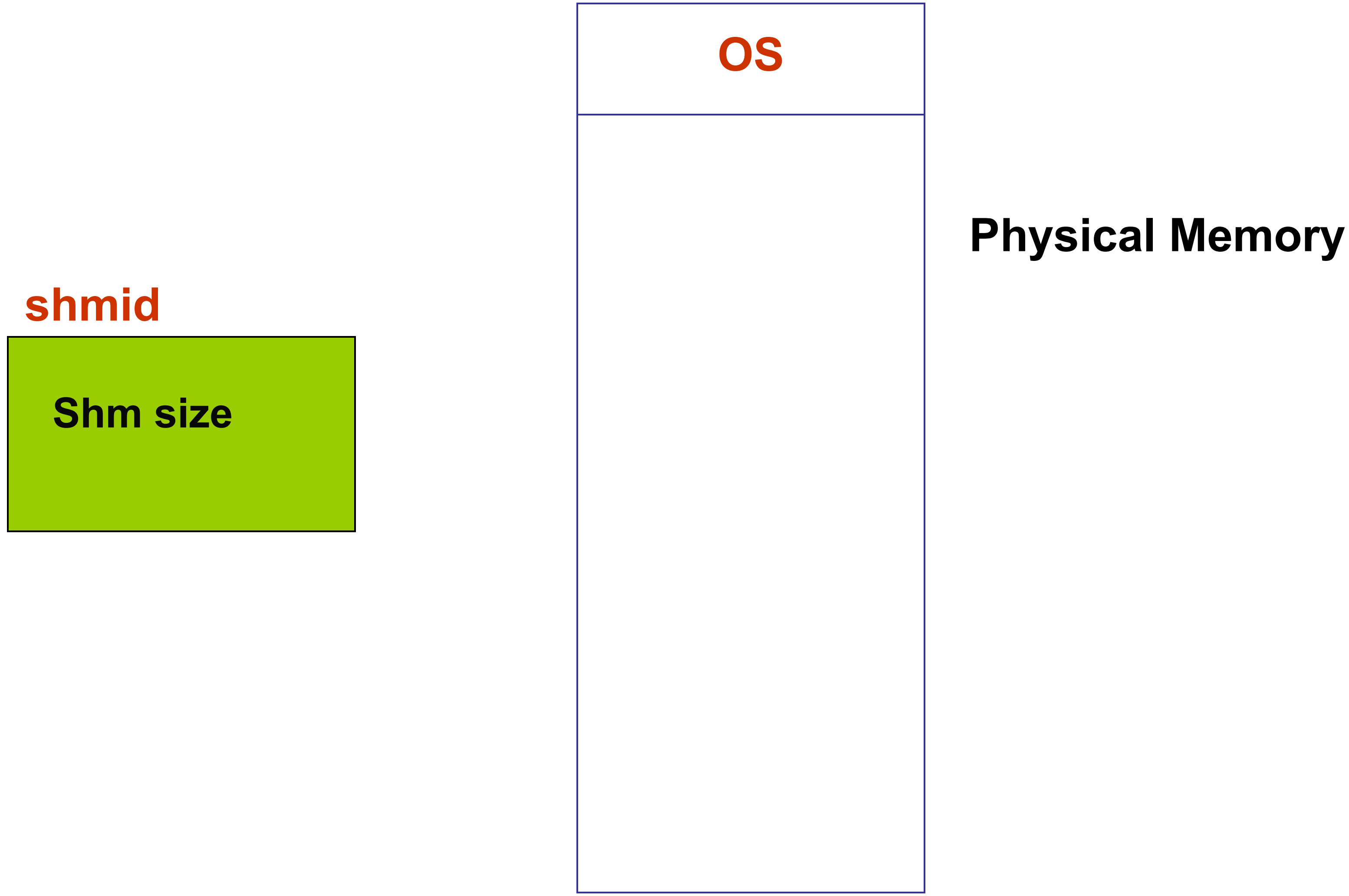
Shared Memory



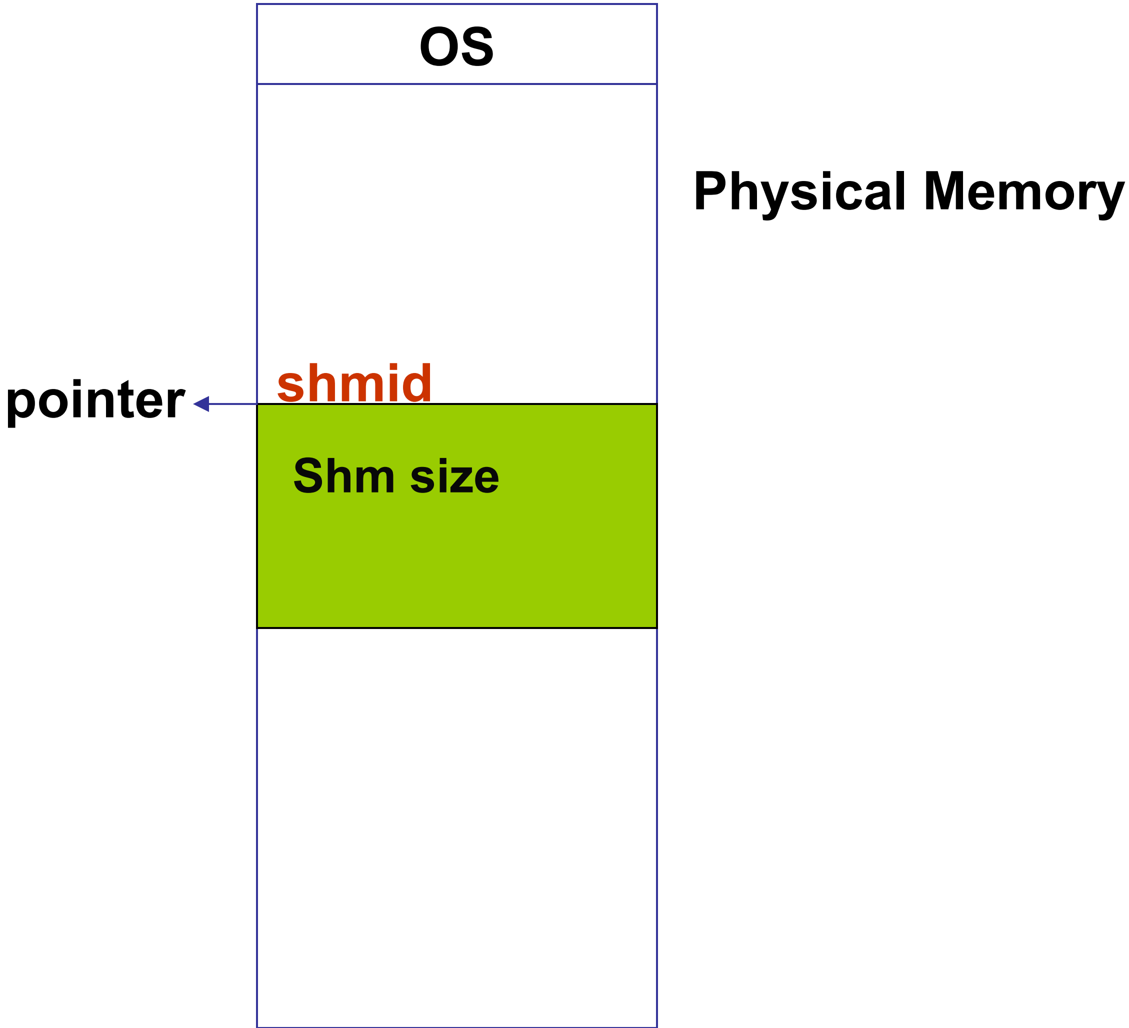
OS

Physical Memory

Shared Memory



Shared Memory



shm – pseudo code

```
shmid = shmget (key, SHM_SIZE, 0644 | IPC_CREAT);

void *shmat (int shmid, void *shmaddr, int shmflg);
        if the shm is read only pass SHM_RDONLY else 0

(void *)data = shmat (shmid, (void *)0, 0);
int shmdt (void *shmaddr);
int shmctl (shmid, IPC_RMID, NULL);
```

shm - Limitations

- **$ipcs –lm**

```
------ Shared Memory Limits --------
    max number of segments = 4096
    max seg size (kbytes) = 32768
    max total shared memory (kbytes) = 8388608
    min seg size (bytes) = 1
```

Semaphore

Semaphores

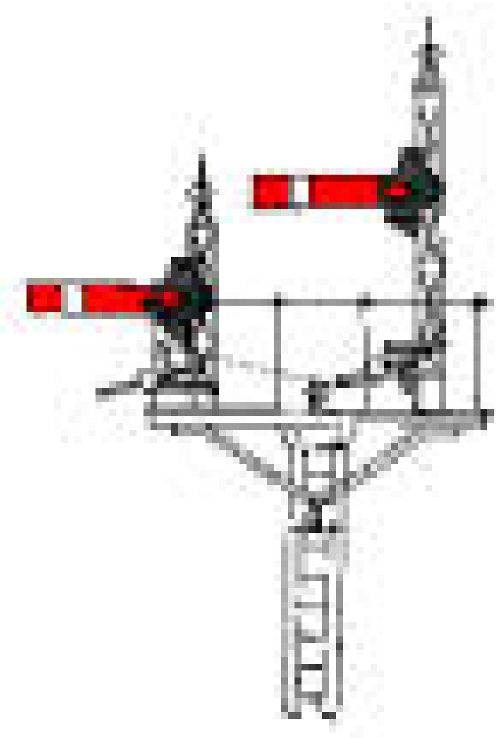
Synchronization Tool

An Integer Number

P ( ) And V ( ) Operators

Avoid Busy Waiting

Types of Semaphore



Semaphores

- If a process wants to use the shared object, it will “lock” it by asking the semaphore to decrement the counter
- Depending upon the current value of the counter, the semaphore will either be able to carry out this operation, or will have to wait until the operation becomes possible
- The current value of counter is >0, the decrement operation will be possible. Otherwise, the process will have to wait

Semaphores

- System V semaphore provides a semaphore set - that can include a number of semaphores. It is up to user to decide the number of semaphores in the set

- Each semaphore in the set can be a binary or a counting semaphore. Each semaphore can be used to control access to one resource - by changing the value of semaphore count

Semaphore - Initialization

```
union semun {
   int val;                                  // value for SETVAL
   struct semid_ds *buf; // buffer for IPC_STAT, IPC_SET
   unsigned short int *array; // array for GETALL, SETALL
   struct seminfo *__buf;       //buffer for IPC_INFO
 };
union semun arg;
```

Semaphore - Initialization

```
semid = semget (key, 1, IPC_CREAT | 0644);
```

```
arg.val = 1;
```

1 for binary

else > 1 for Counting Semaphore

```
semctl (semid, 0, SETVAL, arg);
```

Semaphore - Implementation

```
struct sembuf {
  short sem_num;  /* semaphore number: 0 means first */
  short sem_op;     /* semaphore operation */
  short sem_flg;     /* operation flags */
};

struct sembuf buf = {0, -1, 0};  /* (-1 + previous value) */
```

Semaphore - Implementation

```
semid = semget (key, 1, 0);
semop (semid, &buf, 1); /* locked */

-----Critical section--------

buf.sem_op = 1;
semop (semid, &buf, 1); /* unlocked */
```

Semaphore Limitations

```
$ipcs -ls
```

```
 ------ Semaphore Limits --------
    max number of arrays = 128
    max semaphores per array = 250
    max semaphores system wide = 32000
    max ops per semop call = 32
    semaphore max value = 32767
```

Lab Exercise –Day 4

1. Write a C program to retrieve the specific message from the message queue.
2. Write a C program to display the old message in the shared memory and update the new message whenever new process acquire the shared memory.
3. Perform file locking on a file using semaphores.

Summary

- Linux Kernel
- System Calls
- File Management
- Process Management
- Signals
- User Level Threads
- IPC Primitives: Pipe and FIFO
- System V IPC: MQ, Shm and Sem

1. The design of the UNIX operating System by M. J. Bach

2. UNIX Internals by Uresh Vahalia

3. Advanced Programming in the UNIX Environment by
W. Richard Stevens

4. Unix Network programming by W. Richard Stevens

5. Understanding the Linux Kernel by D. P. Bovet,M. Cesati

6. http://www.ecst.csuchico.edu/~beej/guide/

LINUX Internals

DAY 5

Advanced topics in Embedded Linux

- Overview of Embedded Linux
- Embedded Linux Application
- Comparison
- Advantages
- Hands on exercises
- Projects
- Q&A

Overview of embedded Linux

- **Embedded Linux** is the designation for Linux-based operating systems that are used as
  - Cell phones
  - Personal digital assitants
  - Media player handsets
  - Consumer electronic devices etc.
- Other embedded applications too…
  - Networking equipment
  - Machine control
  - Industrial automation
  - Navigation equipment

Comparison

- Embedded Linux can be characterized as different from desktop and server versions of Linux
- Limited resources in terms of RAM, secondary storage
- Frequently involves flashdisk based memories instead of hard disks
- Embedded Linux is tailor made specific to required application and target hardware
- Optimizations are done in order to make the OS Real Time OS (RTOS)

Advantages of Embedded Linux

- the source code can be modified and redistributed
- relatively small footprint (a typical installation may require less than two megabytes of memory)
- no royalty or licensing costs
- mature and stable
- large support base

References

- The Embedded Linux Consortium http://www.linux-foundation.org/en/ELC/ELCPS
- The Linux Kernel FAQs
  - http://www.faqs.org