Linux System Programming
Part -II

Agenda

1 UNIX / Linux Introduction – Kernel structure

2 File Management

3 I/O Handling

4 File Locking

5 Process Management

Agenda

Objectives

UNIX is a full-feature robust operating system. UNIX and its variants are predominantly used in high-end server environments. Learning UNIX systems programming through an intensive hands-on is a must for any application developer who is intended to work in a UNIX related projects.

The aim of this course is to introduce basic concepts of UNIX kernel architecture and kernel subsystems. This would allow the participants to develop software involving files, processes, memory management, signals, timers, file locking, daemon process and System V Inter process communications along with pipe, fifo and socket programming.

This comprehensive hands-on course provides the essential knowledge and skills of UNIX system programming and topics compatible with the UNIX variants like Linux, Solaris, HP-UX, etc.

UNIX Introduction

Introduction

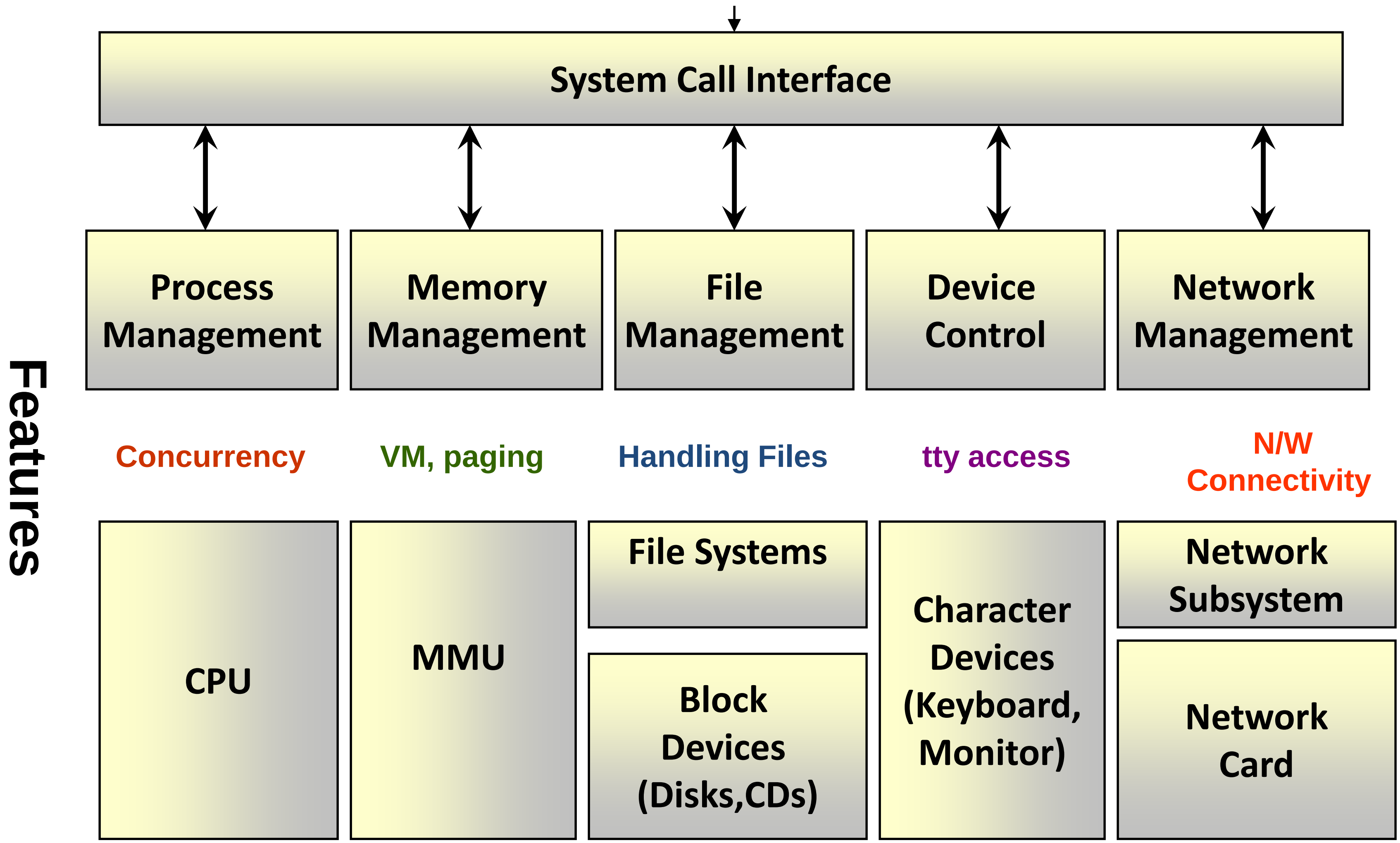
- Unix Features
- Unix Layered Approach
- Unix Kernel characteristics
- Types of Kernel
- Booting Procedure

- Why the name UNIX?
  - The multitasking operating system PDP-7 supported two simultaneous users and Brian Kernighan humorously called it **UNICS** for the uniplexed Information and computing system. The name was changed to **UNIX** in 1970.

Unix features

- Written in a high level language
- Provides a simple user interface
- Uses hierarchical file system
- Gives File protection
- Provides a simple and consistent interface to peripheral devices.
- Is a multi-user, multi-process system.

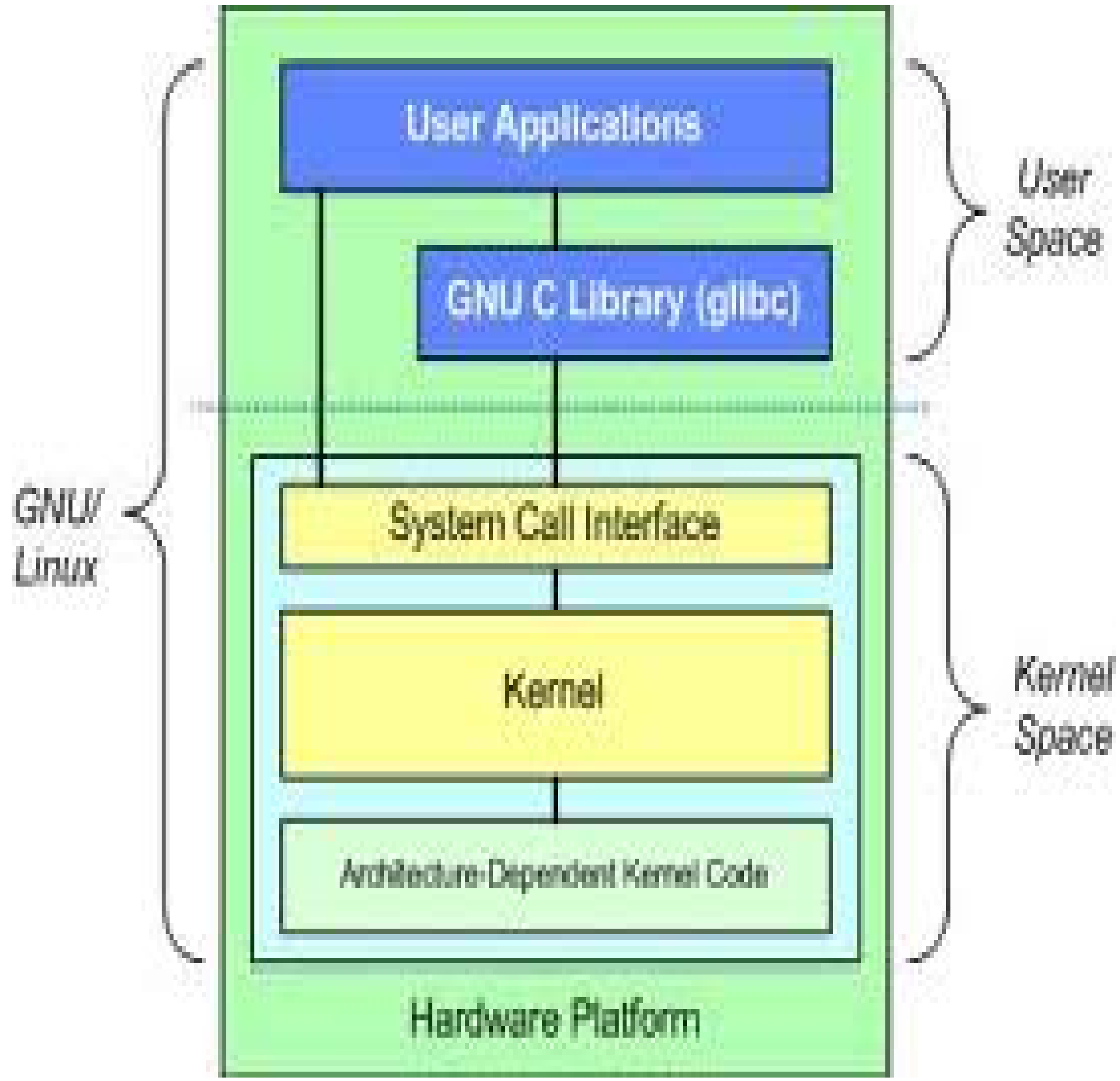
UNIX Kernel Generic Structure



Linux

- The **Linux kernel** is the operating system kernel
- It’s unix type of operating systems
- It is one of the most prominent examples of free and open source software.
- The Linux kernel is released under the GNU General Public License version 2

Linux Kernel Architecture



Kernel Functionality

- Provides following facilities:
  - Basic kernel services
  - System start up and shutdown
  - Memory management
  - Process control
  - File system
  - I/O System (Device Drivers)
  - IPC facilities
  - Networking service

Basic Kernel Services

- The kernel also provides following basic services:
  - Interrupt and trap handling
  - Separation between user and system space - System calls
  - Scheduling
  - Timer and clock handling
  - File descriptor management

Kernel Types

- Types of Kernel:
  Monolithic
  Micro Kernel

- Monolithic: Suitable for Desktop (UNIX, MS-Windows)
  Less run time overhead
  Less extensibility
  A set of primitives or system calls to implement all operating system services such as
  process management,
  concurrency, and memory management itself and
  one or more device drivers as modules

Examples : Unix kernel
Unix System V
Linux

- Micro Kernel:

  Low-level address space management, thread management, and inter-process communication (IPC)
  Suitable for Embedded (RTOS)

- Run time overhead

- Highly extensible
  Example : Qnx

Kernel

- The kernel code contains architecture dependent as well as architecture independent code
- Machine-dependent code deals with
  - Low-level system startup functions
  - Trap and fault handling
  - Low-level manipulation of runtime context of a process
  - Configuration and initialization of hardware devices
  - Runtime support for I/O devices

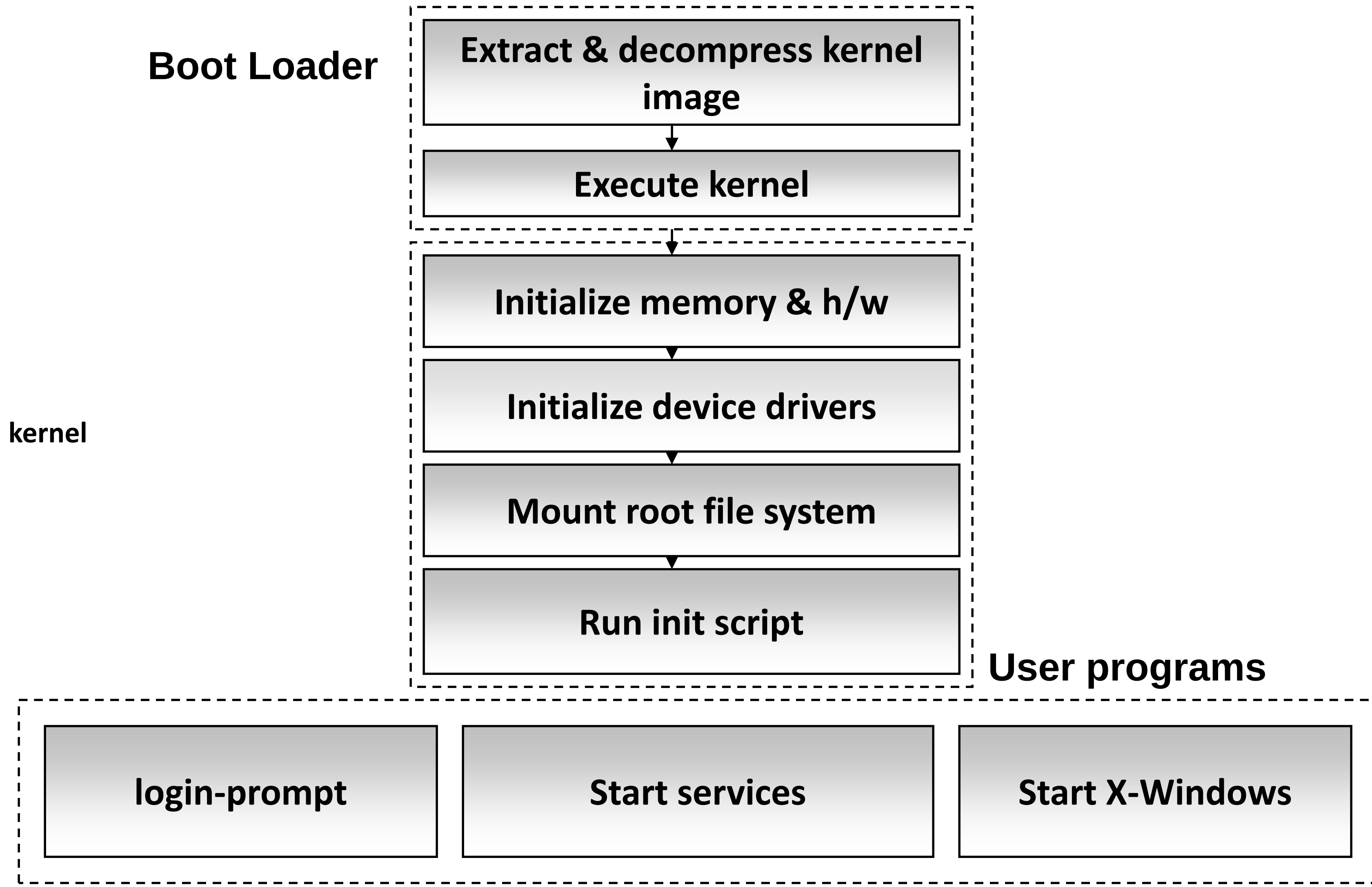
Typical Kernel Structure

- Machine-independent code deals with:
  - System call handling
  - The file system: files, directories, pathname translation, file locking, and I/O buffer management
  - Terminal handling support: the terminal-interface driver and terminal line disciplines
  - IPC facilities
  - Network communication support

Booting Procedure

- Power on
- Boot Loader / MBR
- Loaded into Physical Memory and uncompressed
- Execute Kernel
- Mount File Systems (/etc/fstab)
- Choose Run Level (/etc/inittab)
- Spawn init process
- Shell / login prompt

Booting Procedure

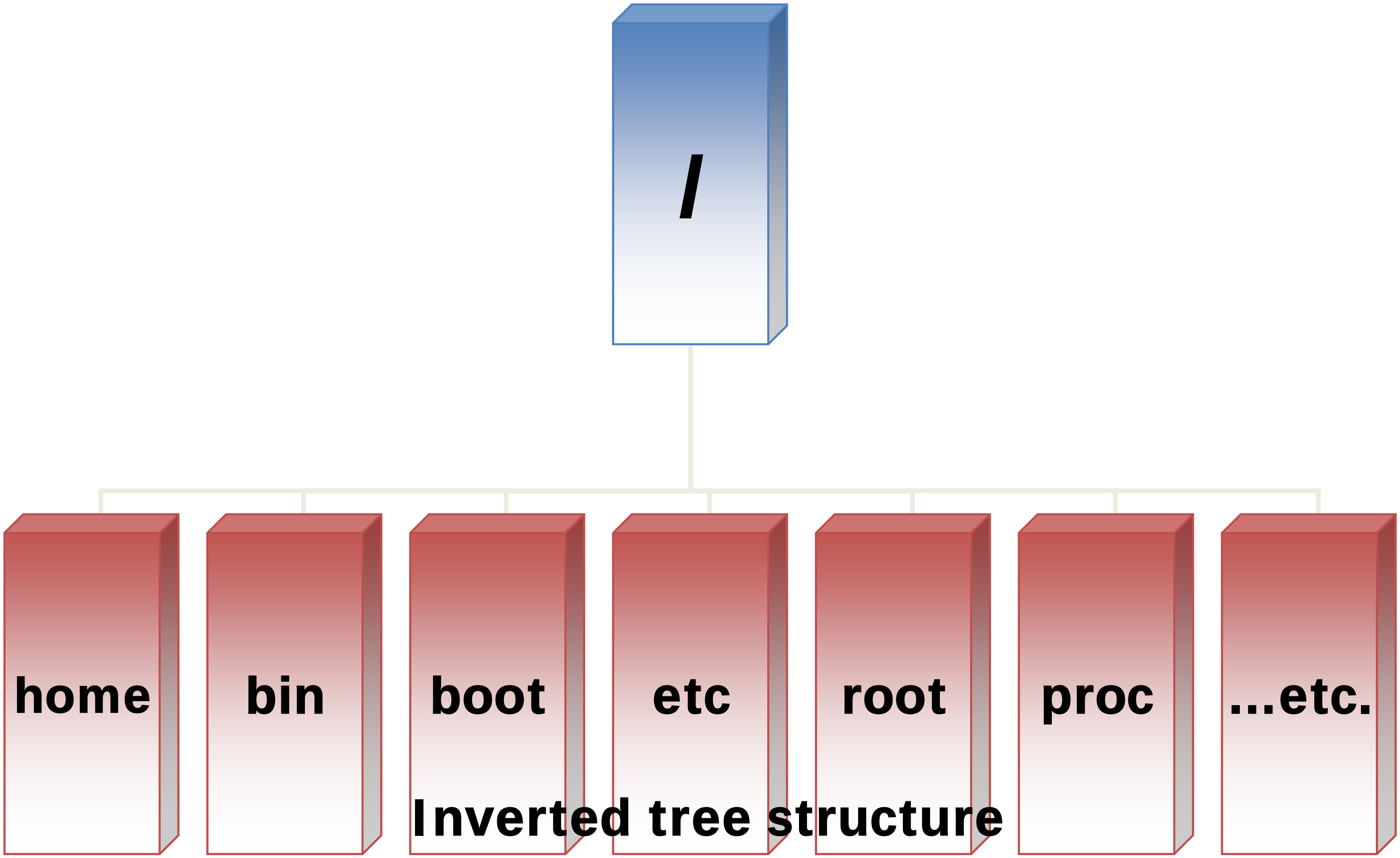


FILE MANAGEMENT

Introduction

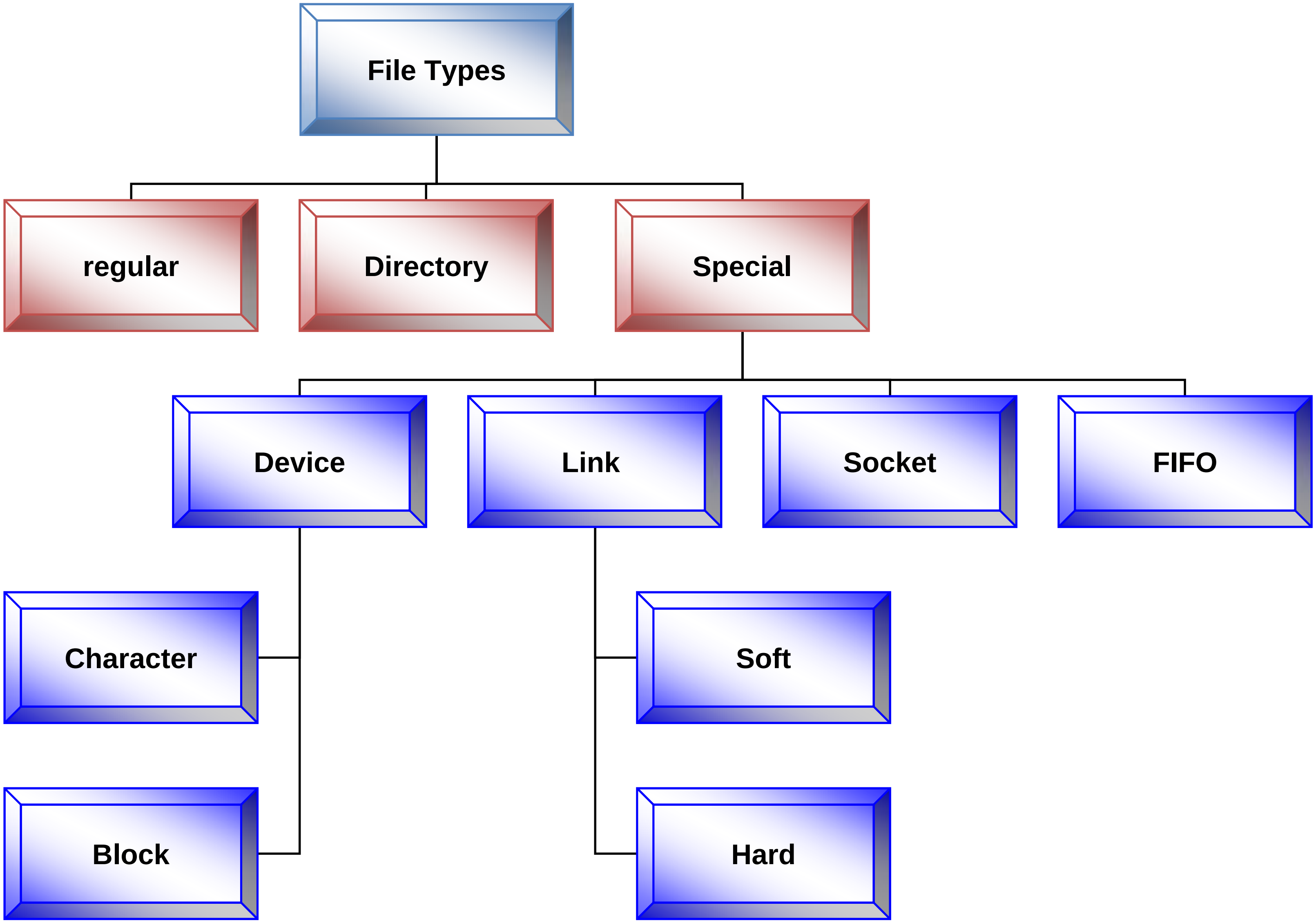
- Different Types of Files
- File and File System
- Types of File Systems
- Device Special Files
- Mounting and Unmounting Devices and File systems
- Buffer Cache

File Tree Structure



Inverted tree structure

Different types of files



File Types

- Identification of File types
  - ? r w - r- - r- -   ?-specifies a type of a file

- Regular (-)
- Directory (d)
- Special Files

Special Files

- Special Files
  - FIFO (p)
  - Socket (S)
  - Link File
    - Soft Link (l)
    - Hard Link (inode numbers are same)
  - Device File
    - Character (c)
      - Example: Monitor, Keyboard, Mouse, Tape
    - Block (b)
      - Example: Hard disk, CDROM, Floppy

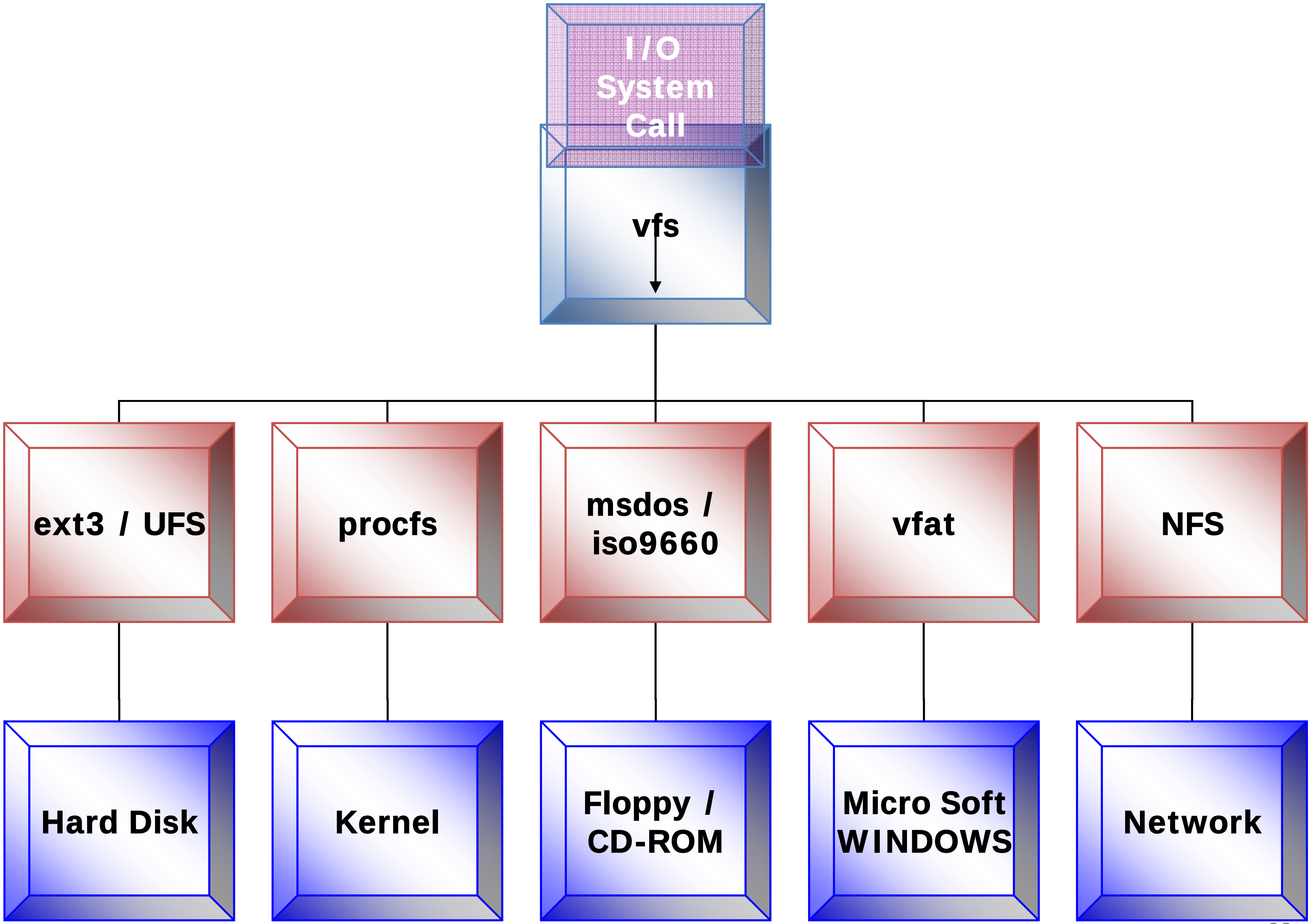
File System

- Facilitates persistent storage and data management
- Facilitates file related system calls.
- Different types of file system for different needs –depends on implementation
  - A logical file system appears as a single entity to the user process, but it may be composed of a number of physical file systems.

Types of File systems

- VFS
- ext3
- UFS
- proc
- msdos
- iso9660
- vfat

VFS


I / O
System
Call
vfs
ext3 / UFS
procfs
msdos /
iso9660
vfat
NFS
Hard Disk
Kernel
Floppy /
CD-ROM
Micro Soft
WINDOWS
Network

File Systems - Creating

When you create a file system, Linux creates a number of blocks on that device.

- Boot Block
- Super-block
- I-node table
- Data Blocks

- Unix also creates an entry for the "/" (root) directory in the I-node table, and allocates data block to store the contents of the "/" directory.

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 - Superblock

- Each device also contains more than one copy of the super-block.
- Unix maintains multiple copies of super-block, as the super-block contains information that must be available to use the device.
- If the original super-block is corrupted, an alternate super-block can be used to mount the file system.

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.

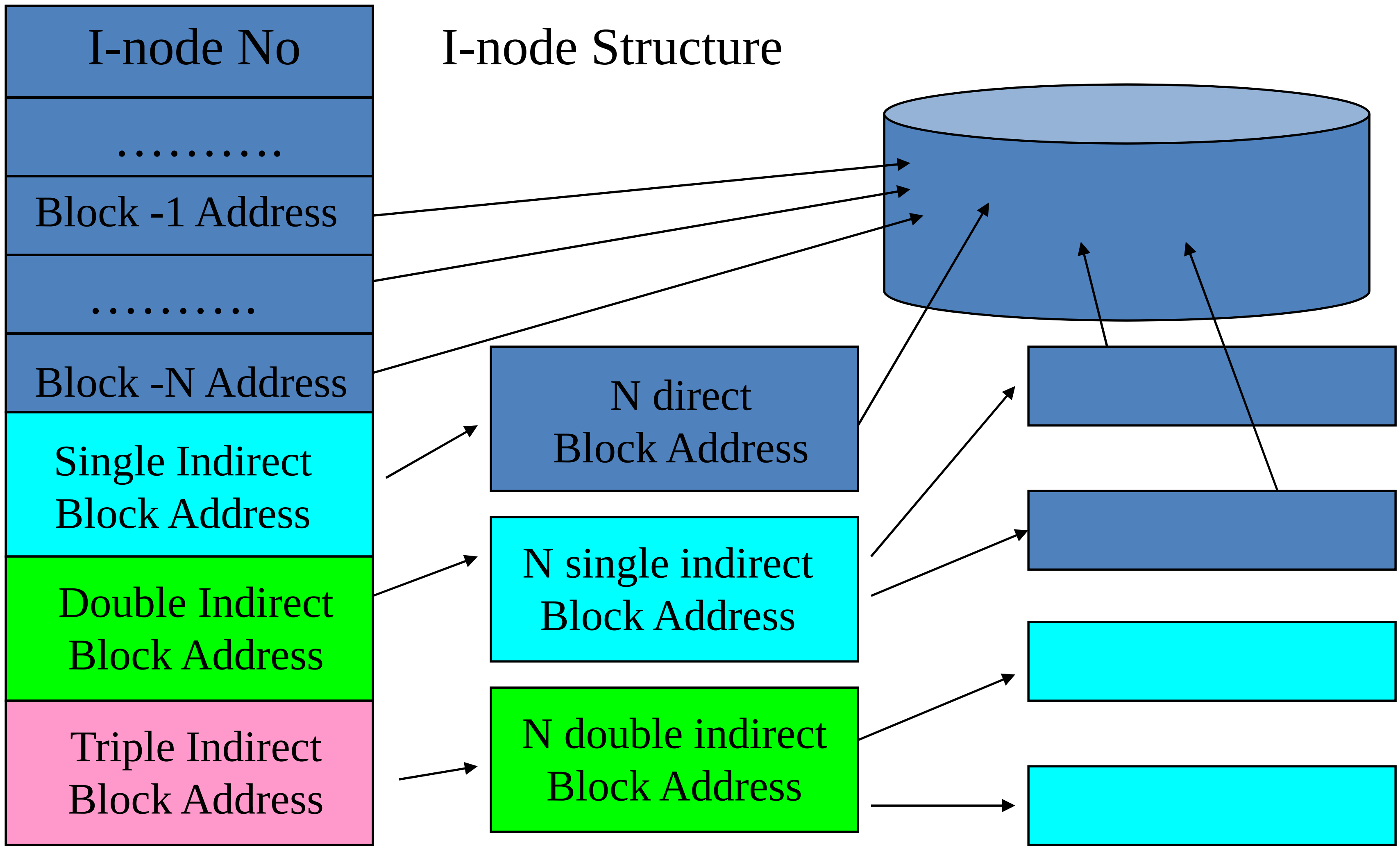
- When a file system is created, the I-node for the root directory of the file system is automatically created.

- Each I-node entry describes one file.

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
  - the size of the file
  - the number of hard links to the file
  - Each I-node entry can track a very large file

ext File System I-node



File system - Regular Files

- Ordinary file creation:
- The kernel allocates space in the hard disk. The text in the file is stored one character per byte of memory.
- The file holds these characters and nothing more. It does not contain any information about its beginning or ending.
- An inode entry is created on a section of the disk set aside for this purpose.

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 (for example ‘2” for floppy, “3” for hard-disk)
  - Minor device number (such as “1” for “hda1”)

Device Special Files

- Switch Table - Unix kernel maintains a set of tables using the major device numbers.

- The switch table is used to find out which device driver should be invoked
- For example : fd –> file table –> inode table –> switch table –> device drivers

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 corresponding device driver.

Assessment

- The -l option to **ls** displays the file type, using the first character of each input line
- Difference between hard link and soft link

Linux Partition

- Linux uses more than one partition on the same disk, even when using the standard installation procedure
- when Linux didn't have journal file systems and power failures might have lead to disaster.
- The use of partitions remains for security and robustness
- a journal file system only provides data security in case of power failure and sudden disconnection of storage devices.

There are two kinds of major partitions on a Linux system

- ***data partition*: ** normal Linux system data, including the *root partition* containing all the data to start up and run the system;
- ***swap partition*: ** expansion of the computer's physical memory, extra memory on hard disk.

Linux Partition Contd.,

- Linux systems use **fdisk** at installation time to set the partition type.
- The standard Linux partitions have number 82 for swap and 83 for data, which can be journaled (ext3) or normal (ext2, on older systems).
- The **fdisk** utility has built-in help, should you forget these values.

Linux Partition Contd.,

- Linux supports a variety of other file system types, such as the relatively new Raiser file system, JFS, NFS, FATxx and many other file systems natively available on other (proprietary) operating systems.
- The standard root partition (indicated with a single forward slash, /) is about 100-500 MB, and contains the system configuration files, most basic commands and server programs, system libraries, some temporary space and the home directory of the administrative user.
- A standard installation requires about 250 MB for the root partition.

- Swap space (indicated with *swap*) is only accessible for the system itself, and is hidden from view during normal operation.

- Swap is the system that ensures, like on normal UNIX systems

Assessment

- On Linux system , did you seen the messages like *Out of memory, please close some applications first and try again? Never.. try to find out the reason ?*

Linux Partition Contd.,

- The kernel is on a separate partition as well in many distributions
- a */boot* partition, holding your kernel(s) and accompanying data files.
- The rest of the hard disk(s) is generally divided in data partitions, although it may be that all of the non-system critical data resides on one partition
- For example when you perform a standard workstation installation,When non-critical data is separated on different partitions, it usually happens following a set pattern:

Linux Partition Contd.,

- a partition for user programs (*/usr*)
- a partition containing the users' personal data (*/home*)
- a partition to store temporary data like print- and mail-queues (*/var*)
- a partition for third party and extra software (*/opt*)

Linux Partition Contd.,

- a partition with all data necessary to boot the machine
- a partition with configuration data and server programs
- one or more partitions containing the server data such as database tables, user mails, an ftp archive etc.
- a partition with user programs and applications
- one or more partitions for the user specific files (home directories)
- one or more swap partitions (virtual memory)
- Servers usually have more memory and thus more swap space.

Linux Partition Contd.,

- In Linux, **df** is the GNU version, and supports the -h or *human readable* option which greatly improves readability.
- The **df** command only displays information about active non-swap partitions.
- These can include partitions from other networked systems, like in the example below where the home directories are mounted from a file server on the network,

[root@localhost KARTHIKEYAN]# df

- Filesystem 1K-blocks Used Available Use% Mounted on
-
- /dev/mapper/VolGroup00-LogVol00
-
- 471016352 5534712 441169424 2% /
-
- /dev/hda1 101086 11038 84829 12% /boot
-
- tmpfs 1033212 0 1033212 0% /dev/shm
-
- /dev/sda1 3915748 831940 3083808 22% /media/KARTHIKEYAN
-

[root@localhost KARTHIKEYAN]# **df -h**

- Filesystem   Size Used Avail Use% Mounted on
-
- /dev/mapper/VolGroup00-LogVol00
-
-                       450G 5.3G 421G   2% /
-
- /dev/hda1   99M   11M   83M 12% /boot
-
- tmpfs   1009M   0 1009M   0% /dev/shm
-
- /dev/sda1   3.8G 813M 3.0G 22% /media/KARTHIKEYAN

File Systems - Mounting

- Each file system must be mounted before it can be used. Normally, all file systems are mounted during system startup depending on fstab entry
- **root** file system is mounted by default

- It is possible to mount a file system at any time using the "**mount**" command.

File Systems - Mounting

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

Screen shot of mount command

$ mount

/dev/hda2 on / type ext2 (rw)

none on /proc type proc (rw)

File Systems - Mounting

- The “dev” directory contains names of each device special file.
- Each file system has a “/” (root) directory. However, once a file system is mounted, it’s the root directory that is accessed through mount point.
- A file system is mounted typically under an empty directory. This directory is called the “mount point” for the file system.

File Systems – Mounting

- When a file system is mounted, the system reads the I-node table and the super-block into memory.
- The in-memory I-node table is used when a process tries to access a file.
- If kernel does not find an entry in this I-node table, it reads the I-node from the on-disk I-node table into in-memory I-node table.
- File system can be unmounted using **umount** cmd

UFS

- UNIX File System –general purpose disk based file system.

- Used in Solaris (ext3 for Linux)

- Boot block contains bootstrap code

Inode block (UFS)

- Inode stores info about
  - type of the file
  - id (uid, gid)
  - size of the file
  - read, execute, modify -attributes
  - Time stamps (creation, modification and access time)
  - number of links to the file
  - arrays of disk block addresses that hold the file’s data

Assessment

- How to check the mount points ?
- What is different between df and du ?
- The df command only display information about active non-swap partition is it **TRUE** or **FALSE**?

Super block –UFS

A super block holds

- information about the geometry
- layout of the file system
- Multiple copies of super block (replicate)

- Major information
  - Address of super block
  - Offset of the first inode blocks
  - Offset of the first data blocks
  - Last time written
  - Number of blocks
  - Number of data blocks

Creating Ext 2 File system

- **Creating an ext2 file system with the loop device**
- **mke2fs -c /dev/loop0 10000**
- **Creating a mount point and mounting the file system through the loop device**
- **mkdir /mnt/point1**
- $ **mount -t ext2 /dev/loop0 /mnt/point1**
- $ **ls /mnt/point1**

Contd.,

- At the time a new file is created, it gets a free inode.
- In that inode is the following information:
- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file
- Date and time of creation, last read and change.
- Date and time this information has been changed in the inode.
- Number of links to this file
- File size
- An address defining the actual location of the file data.

procfs

- To store necessary information about currently running processes and system limitations
- Size of /proc is zero.
- Using /proc we can see the memory area of the process
- Syntax: /proc/<ProcessID>/maps

Assessment

- Type the following command in your terminal, look at the memory area of the process
- Cat /proc/1/maps
- Like that check your running shell memory area
- Identify the starting and ending virtual address,Offset,major:minor numbers and inode number

Looking different memory region

- Cat /proc/iomem
- We can look at how different devices are mapped into the memory
- lspci –vvv
- Vmstat – virtual memory statistics

Assessment

- Open /proc file identify the running process ?
- How to find out CPU load balance ?
- What is the inode value of uptime ?

File Systems – 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.

- It reduces disk traffic and access time

File Systems – Buffer Cache (Contd.).

- 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.

File Systems – Buffer Cache (Contd.).

- 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 Systems – Buffer Cache (Contd.).

- 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.

File Systems – Buffer Cache (Contd.).

- 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.

I/O Handling

I/O System Overview

- The basic model of I/O system is a sequence of bytes that can be accessed either randomly, or sequentially.
- There are no file formats (sequential, indexed etc.) and no control blocks (such as a file control block) in a typical user process.
- The I/O system is visible to a user process as a stream of bytes (I/O stream). A Unix process uses **descriptors** (small unsigned integers) to refer to I/O streams.

I/O System Overview (Contd.).

- The system calls related to the Input-Output operations take a descriptor as an argument.
- Each process has a separate File Descriptor (FD) Table.
- Valid file descriptor ranges from 0 to a maximum descriptor number that is configurable.
- Kernel assigns descriptors for standard input (0), standard output (1) and standard error (2) of the FD table as part of process creation.
- Kernel always assign minimum possible value from the fd table to any new file descriptors.

fd table

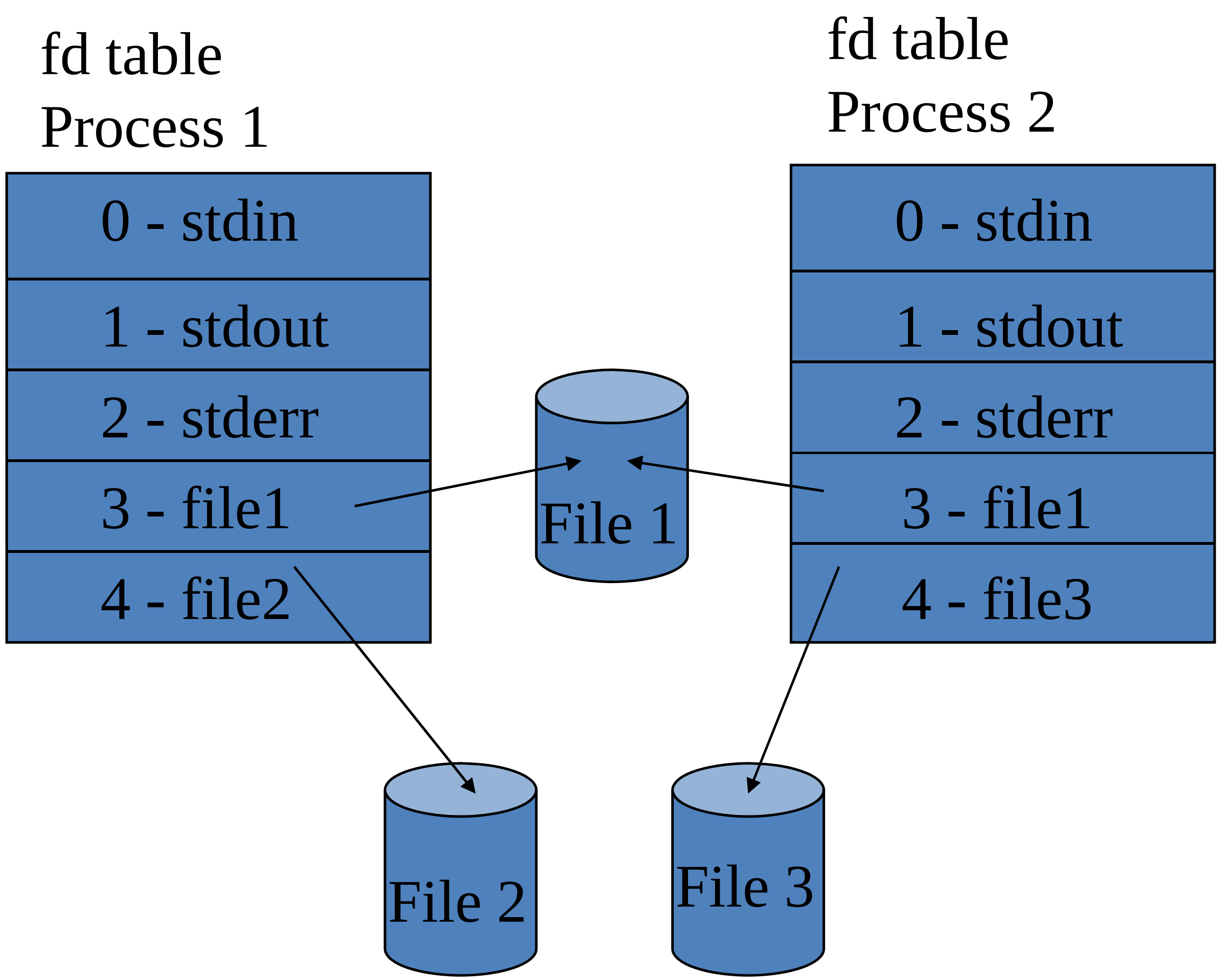
- The system calls related to the I/O system take a descriptor as an argument to handle a file.

- The descriptor is a positive integer number.

- If a file open is not successful, fd returns -1.

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

fd table for each process



Locate a file

- File descriptor table (fd, process specific)
- File table (offset, mode,permission, pointer to inode table)
- Inode Table (inode number, pointer to Data Block).
- Switch table (only for device special files)
- Data Block (where a file is stored)

Different Levels of Calls

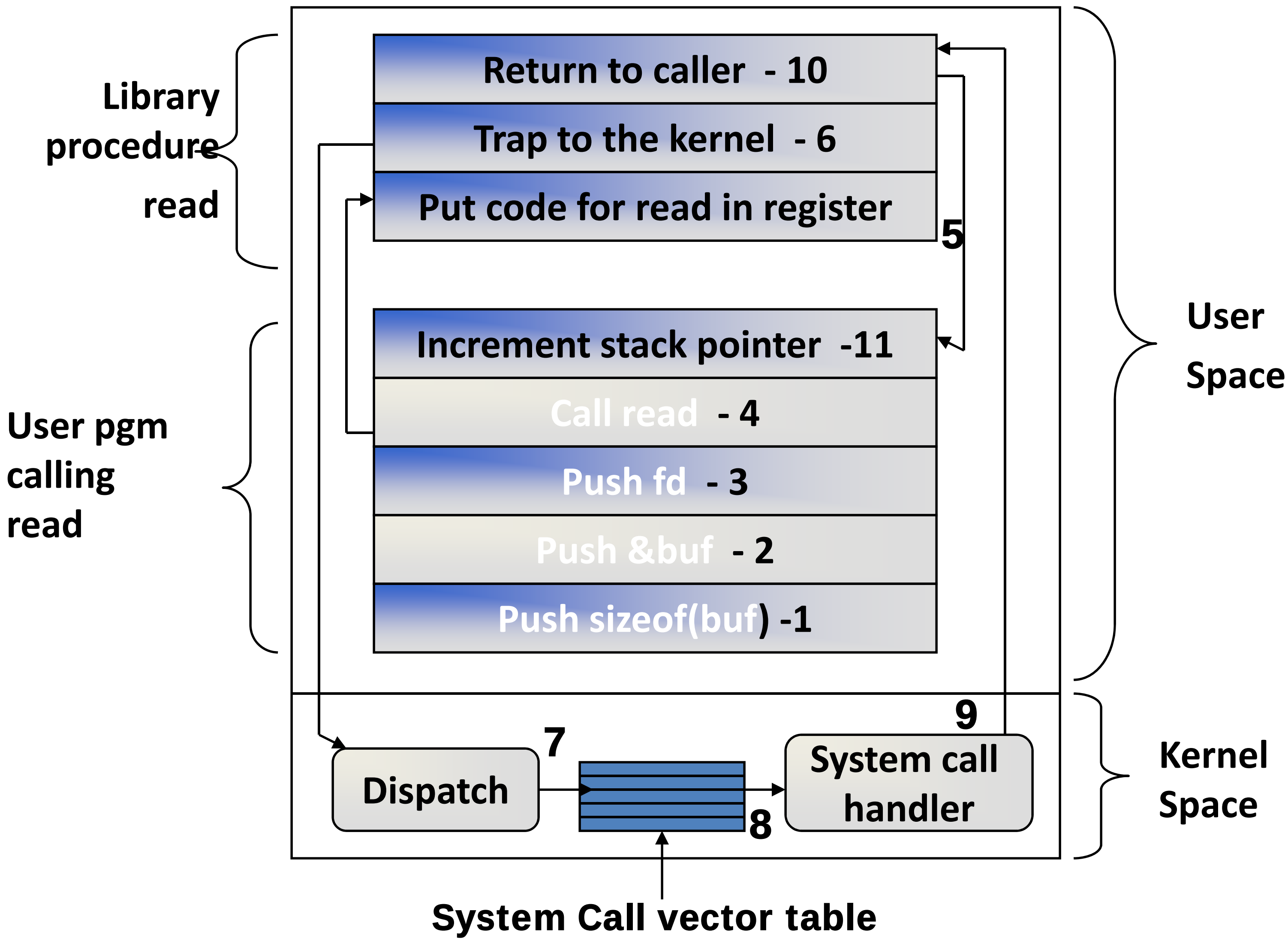
- Library Functions (Application Programs)
  – fopen, fwrite, fread, fclose
- System Calls (System Programs)
  – open, write, read, close
- Entry Points (Kernel Programs)
  – my_open, my_write, my_read, my_release

System Calls

- The system call code is physically located in the kernel. The kernel itself is stored in a separate area of memory - which is normally not accessible to the process.

- Therefore, the first thing that is required to execute a system call is to change to Kernel Mode - so that the kernel memory can be accessed - this is what the “int 0x80” instruction in system call wrapper function (on Intel) does.

read(fd, &buf, sizeof(buf))



System Calls

– creat / open
– read, write
– lseek
– close, unlink
– dup / dup2
– fcntl
– stat
– select
– sync

Opening a file

int creat (char *file_name, mode_t mode)

int open(char *file_name, int flags);

int open (char *file_name, int flags,
mode_t mode);

Ex: fd = open("temp", O_RDWR|O_CREAT, 0744);

Duplicate a fd

dup or dup2 copies the oldfd into the newfd.

int new_fd = dup (old_fd);

int dup2 (int new_fd, int old_fd);

new_fd and old_fd shares: locks, file position and flags.

write

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

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

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

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

On success return with:

- Number of bytes read
- 0 - indicates end of the file
- -1 - on error.

Random Access

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

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

File control

- fcntl is used to manipulate file descriptors. It performs given operations to the specified file descriptor.

```
int fcntl (int fd, int command);
int fcntl (int fd, int command, long arg);
int fcntl (int fd, int command, &flock struct);
```

- The fcntl function use for
  - duplicating a file descriptor
  - reading the file descriptor flag
  - File locking

Get File Status

int stat (“file_name”, struct stat *);

int fstat (fd, struct stat *);

int lstat (“file_name”, struct stat *);

select ( ) system call

- *Select* is a system call used to handle more than one file descriptor in an efficient manner.
- int select (int n, fd_set *readfds,

  fd_set *writefds, fd_set *exceptfds,

  struct timeval *timeout);

fd set

- fd_set is the file descriptor set, which is an arrays of file descriptors.
- FD_CLR (int fd, fd_set *myset);
- FD_ISSET (int fd, fd_set *myset);
- FD_SET (int fd, fd_set *myset);
- FD_ZERO (fd_set *myset);

Remove a file

write modifications to the disk file

void sync (void);

Close a fd

int close (fd);

int unlink("file_name");

equivalent to $rm file_name;

Internal Routines

- UNIX contains a number of internal routines that are used for accessing a file.

Some of these routines are:

- namei() (convert a “file_name” into an inode)
- iget() (reads an I-node)
- iput() (writes an I-node)
- bread() (read a block from buffer cache/disk)
- bwrite() (write a block from buffer cache to disk)
- getblk() (get a free block in the buffer cache)

File Locking

Introduction

- Share data in a file
- Concurrent access
- Race condition
- Dead Lock
- Synchronization
- File locking

File Lock

- File locking is a mechanism which allows only one process to access a file at any specific time.
- By using file locking mechanism, many processes can read/write a single file in a safer way.

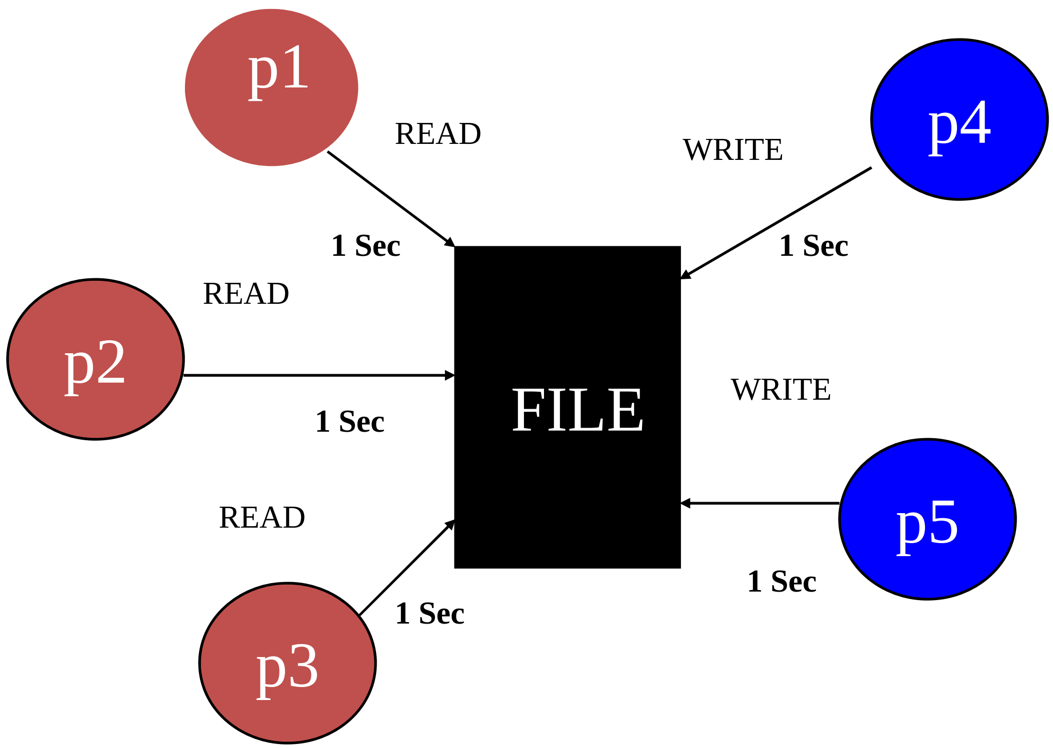
Case study :

- We will take the following case to understand why file locking is required.
- Mr.Anandh opens and reads a file which contains account related information.
- Mr.Sathish" also opens the file and reads the information in it.
- Now "Mr.Anandh" changes the account balance of a record in its copy, and writes it back to the file.
- Then Mr.Sathish which has no way of knowing that the file is changed since its last read, has the old original value.
- It then changes the account balance of the same record, and writes back into the file.
- Now the file will have only the changes done by Mr.Sathish.
- To avoid such issues locking is used to ensure "serialization".

Types of File Locking

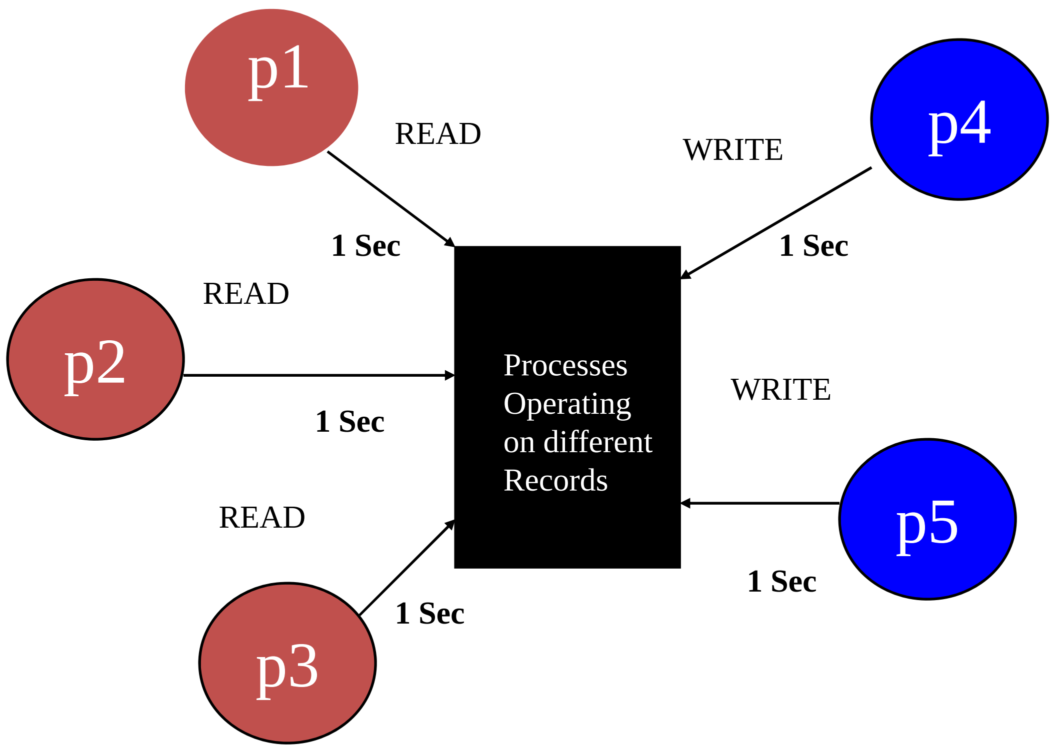
- Two Types
  - Mandatory locking
    - Lock an entire file
  - Advisory (or) Record locking
    - Lock to specific byte range
    - Granularity
    - Improve Performance

Mandatory Locking



Total Time = 1 + 1 +1= 3 sec

Record Locking



Total Time = 1 sec

flock structure

```
struct flock
{
    short  l_type;  /* lock type: read, write or unlock */
    short  l_whence;
    off_t           l_start;
    off_t           l_len;
    pid_t           l_pid;
};
```

Read lock: allows many readers but not a single writer

Write lock: allows only one writer but not a single reader

Locking

- Lock or unlock is performed by fcntl function.
- int fcntl (int fd, int cmd, struct flock &);
- Command may be:
  – F_SETLKW
  – F_SETLK
  – F_GETLK

Rules :

- Remount the root filesystem with “mand” option using the mount command as shown below.
- This will enable mandatory locking at the file system level.
- Note: You need to be root to execute the below command.
- # **mount -oremount,mand /**
- Enable the Set-Group-ID and disable the Group-Execute-Bit
- **chmod g+s g-x filename**

Pseudo code –Write Lock

```
struct flock lock;
   lock.l_type      = F_WRLCK;
   lock.l_whence = SEEK_SET;
   lock.l_start      = nth record;
   lock.l_len                  = sizeof (record);
   lock.l_pid                  = getpid( );

  fcntl ( fd, F_SETLKW, &lock );
   …….critical section……
   lock.l_type = F_UNLCK;

  fcntl ( fd, F_SETLK, &lock );
```

Assessment

- Why we need to assign set group id permission while doing the mandatory lock ?
- Write a C program to demonstrate mandatory lock and advisory lock in same terminal

Process Management

Introduction

- Process is a program in execution.
- Processes carry out tasks in a system
- A process includes program counter (PC), CPU registers and process stacks, which contains temporary data.
- Unix is a multiprocessing system
- The unix kernel is reentrant

Processes : Introduction

- A process uses many resources like memory space, CPU, files, etc., during its lifetime.
- Kernel should keep track of the processes and the usage of system resources.
- Kernel should distributes resources among processes fairly.
- Most important resource is CPU. In a multiprocessing environment, to attain an ideal performance of a system, the CPU utilization should be maximum.

Mode and space

- In order to run unix, the computer hardware must provide two modes of execution
  - kernel mode
  - user mode
- Some computers have more than two execution modes
  - eg: Intel processor. It has four modes of execution.

- Each process has virtual address space , references to virtual memory are translated to physical memory locations using set of address translation maps.

Context Switch

- Execution control is changing from one process to another.
- When a current process either completes its execution or is waiting for a certain event to occur, the kernel saves the process context and removes the process from the running state.
- Kernel loads next runnable process's registers with pointers for execution.
- Kernel space: a fixed part of virtual address space of each process . It maps the kernel text and data structures.

Per process object

- two important per-process objects are managed by the kernel in which are in process address space.
  - 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. It is protected from user mode.
  - kernel stack – to keep track of its function call sequence when executing in the kernel

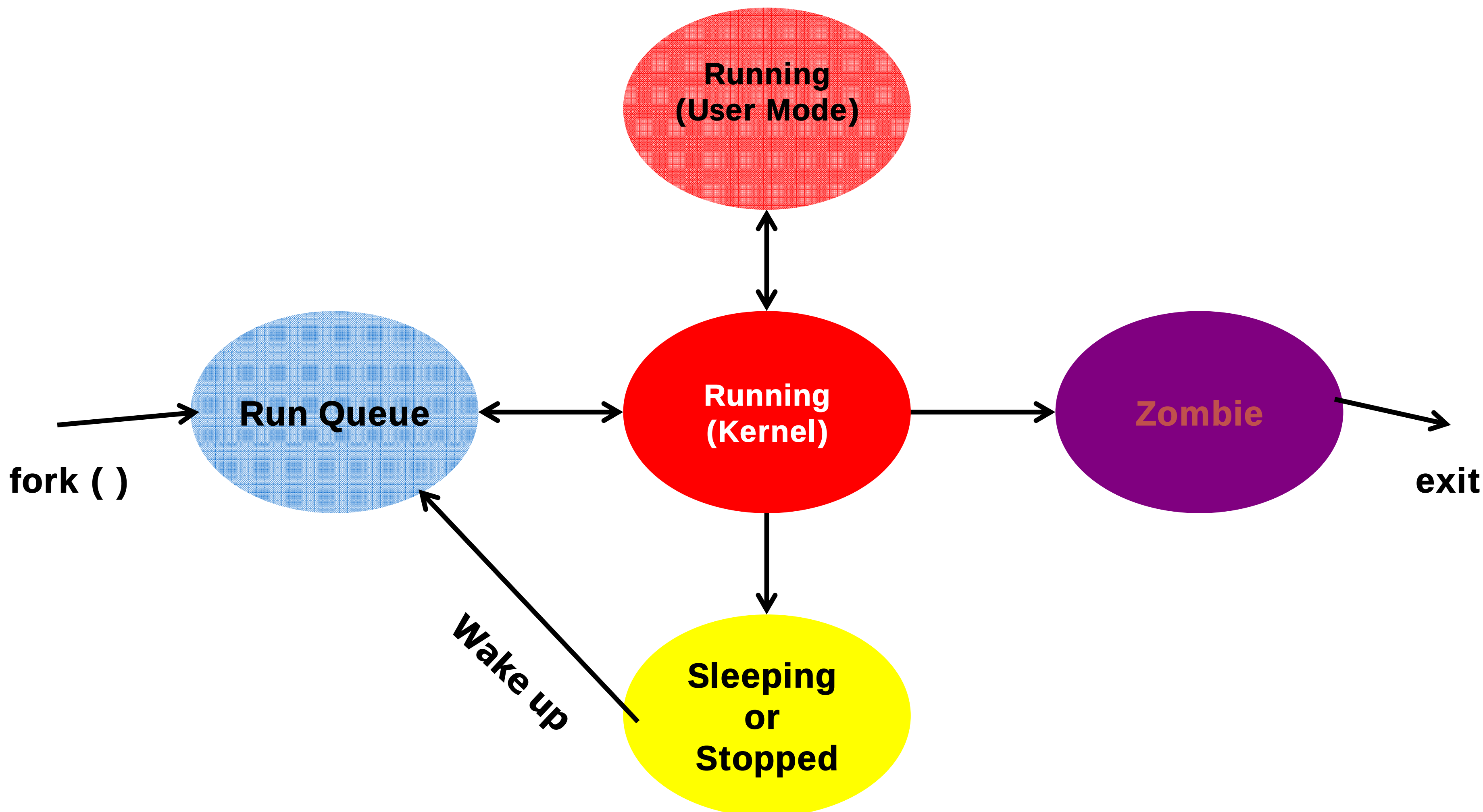
Execution context

- 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
- System calls and signals 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

- Every process is represented by a task_struct data structure.
- This structure is quite large and complex.
- When ever a new process is created a new task_struct structure is created by the kernel and the complete process information is maintained by the structure.
- When a process is terminated, the corresponding structure is removed.
- Uses doubly linked list data structure.
- Solaris uses proc structure to manage processes.

Process states



Linux process states

- TASK_RUNNING // runnable ,running state
- TASK_INTERRUPTIBLE // sleeping or block or waiting
- TASK_UNINTERRUPTIBLE // does not wakeup runnable process
- TASK_STOPPED //execution has stopped
- TASK_ZOMBIE // Zombie state (parent not received signal)

Solaris Process States

- SIDL
- SRUN
- SONPROC
- SSLEEP
- SSTOP
- SZOMB

Identifiers

- Every process in the system has a process identifier.
  - The process identifier is not an index into the task vector, it is simply a number.
  - Each process also has User and Group identifiers, these are used to control the process access to the files and devices in the system
  - eg: ppid ,pid,uid, gid, euid,egid

Scheduling

- The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime.
- This clock is the combination of software and hardware setup.
- It is independent of CPU frequency.
- A clock tick unit is Jiffy. System‘s interactive response depends on the clock frequency.
- For example: the jiffy value may be 10ms (100Hz) or 1ms (1000Hz) depending on implementation

Scheduling

- Each clock tick, the kernel updates the amount of time that the current process has spent in system and in user mode.

- Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire. These timers can be single-shot or periodic timers.

Process Scheduling

- The job of a scheduler is to select the most deserving process to run out of all of the runnable processes in the run queue.
- Implement fair scheduling to avoid starvation
- Implement suitable scheduling policy
- Updates state of the processes in every clock tick (jiffy)

Process scheduling

- policy - FIFO, Round Robin, Shortest Job First, FILO, Priority based etc.
- priority - higher priority process will be allowed to run.
- Pre-emptive and Non-preemptive scheduling
- rt_priority – many UNIX variants support real time scheduling priority range.

Priority range

- Scheduling priorities (in a typical UNIX system) have integer values between 0 and 127, with smaller numbers meaning higher priorities.

- For Solaris: 0 to 169

- For Linux: 0 to 139

Process Scheduling -Linux

- The Linux kernel implements two separate priority ranges.
- The first is the nice value, a number from -20 to 19 with a default of zero. Larger nice values correspond to a lower priority.
- A process with a nice value of -20 receives the maximum time slice, whereas a process with a nice value of 19 receives the minimum time slice.
- Time slice: minimum -10ms, default -150ms and maximum – 300ms

Process Scheduling -Linux

- The second range is the real-time priority
- By default, it ranges from zero to 99.
- All real time processes are at a higher priority than normal processes.
- Linux implements real-time priorities in accordance with POSIX.

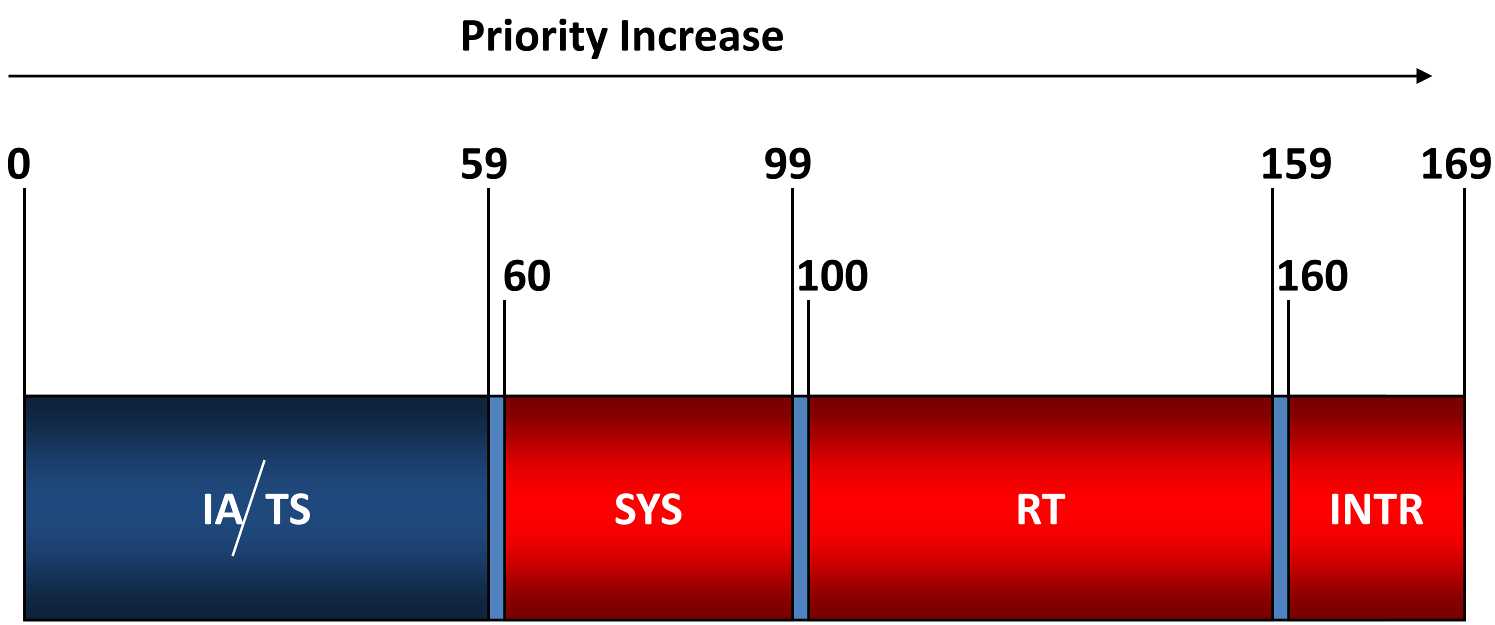
Process Scheduling -Linux

- Linux provides two real-time scheduling policies, SCHED_FF and SCHED_RR.
- The normal non real-time scheduling policy is SCHED_OTHER.
- SCHED_FIFO implements without time slices- so it can run until it blocks or explicitly yields the processor.
- SCHED_RR is identical to SCHED_FIFO except that each process can only run until it exhausts a predetermined time slice.

Scheduler system calls

- nice() Set a process's nice value
- sched_setscheduler() Set a process's scheduling policy
- sched_getscheduler() Get a process's scheduling policy
- sched_setparam() Set a process's real-time priority
- sched_getparam() Get a process's real-time priority
- sched_get_priority_max() Get the maximum real-time priority
- sched_get_priority_min() Get the minimum real-time priority
- sched_rr_get_interval() Get a process's timeslice value

Solaris Priority range



IA-Interactive class

TS-Time Sharing class

SYS-System class

RT-Real Time class

INTR-Interrupt class

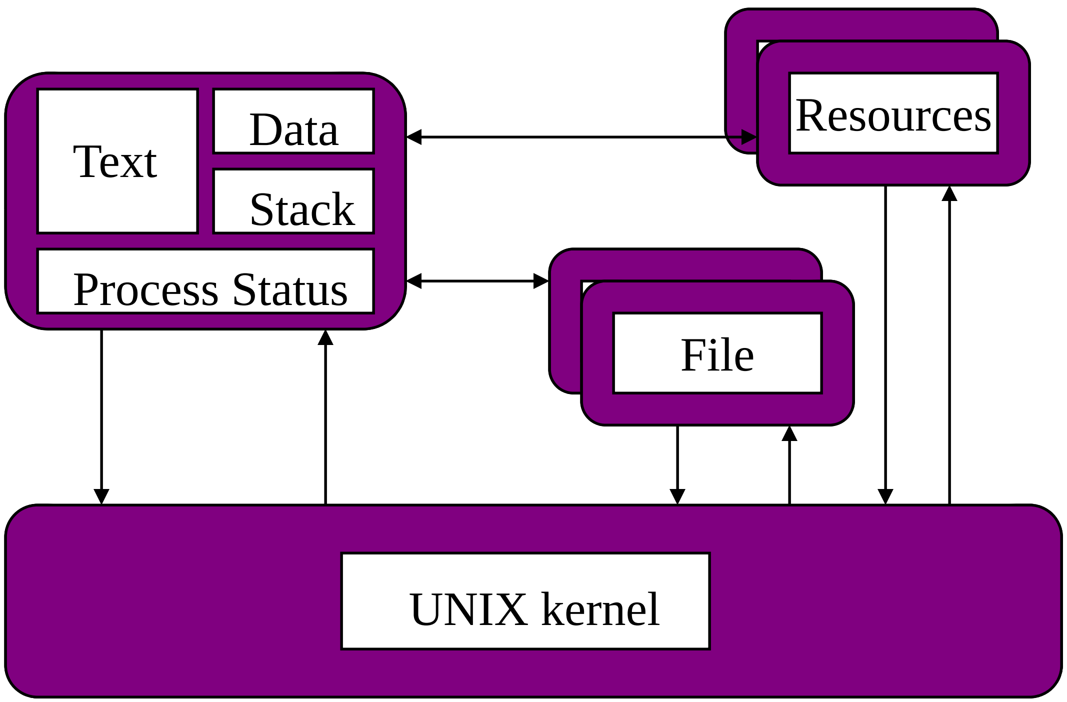
Solaris Priorities range

- 170 global priorities, numerically larger priority values correspond to higher priorities
- The priority range is split into three scheduling classes
  - TS (Round Robin, 0-59)
  - IA (is an enhanced TS) (Round Robin, 0-59)
  - SYS (FIFO, 60 -99)
  - RT (Round Robin, 100-159)
  - Interrupt (160 -169)

Executing programs

- A Command is normally executed in a shell
- When you enter a command, the shell searches the corresponding command's executable image (use PATH environment variable) and loads the image then executes.
- For execution, the shell uses fork and creates a new child process and the child process's image is replaced with the command's executable image.
- After completion of execution of the child process it gives the exit status to the parent process, i.e, shell.

A Unix Process



links

- In a UNIX system 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 *cloned* from previous processes.
- Every task_struct representing a process keeps pointers to its parent process and as well as to its own child processes
- $pstree – process tree structure shows the process dependency.

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
- 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.

fork ( )

- pid_t fork (void); creates a new process
- All statements after the **fork()** system call in a program are executed by two processes - the original process that used **fork()**, plus the new process that is created by **fork( )**

```
main ( ) {
        printf (“ Hello fork %d\n, fork ( ) ”);
        }
```

- Hello fork: 0
- Hello fork: x (> 0);
- Hello fork: -1

Parent and child

```
if (!fork) {
        /* Child Code  */
}
else {
      /* parent code */
        wait (0); /* or */
        waitpid(pid, ….);
}
```

Zombie state and Orphan process

- When a child process exits, it has to give the exit status to the parent process.
- If the parent process is busy or suspended then the child process will not be able to terminate.
- Such state is called Zombie.
- if parent exits before child, the child will become an orphan process and the init process (grand parent) will take care of the child process.

Copy on Write (COW)

- UNIX uses the COW technique for economical use of the memory page.
- The parent and child process shares the parent space and is not copied to the child process. The memory pages are write protected.
- if parent or child wants to modify the pages, then kernel copies the parent pages to the child process.
- Advantage: Kernel can defer or prevent copying of a parent process address space.

execl

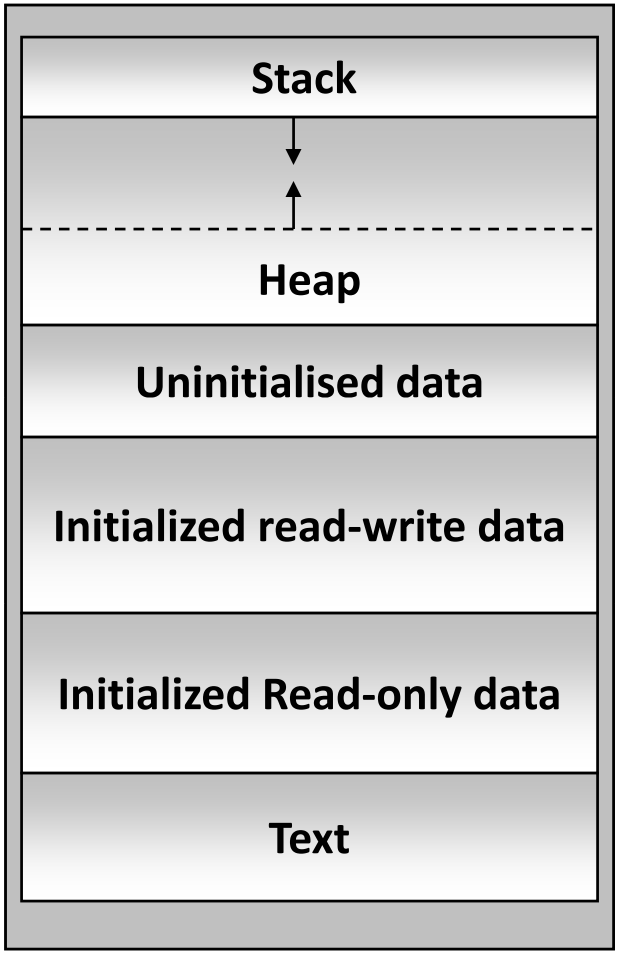
- To run a new program in a process, you use one of the “exec” family of calls (such as “execl”) and specify following:
  - the pathname of the program to run
  - the name of the program
  - each parameter to the program
  - (char *)0 or NULL as the last parameter to specify end of parameter list

exec family

- int execl (const char *path, const char *arg, …..);
- int execlp (const char *file, const char *arg);
- int execle (const char *path, const char *arg, ……., char *const envp[ ]);
- int execv (const char *path, char *const argv[ ]);
- int execvp (const char *file, char *const argv[ ]);

- All the above library functions call internally execve system call.
- int execve (const char *filename, char *const argv [ ] , char *const evnp [ ]);

An executable image



$ size a.out (man size)

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

Text portion

User Context consists portions accessible to the process while running in user mode.

• The *text* portion of a process contains the actual machine instructions that are executed by the hardware.

•When a program is executed by the OS, the text portion is read into memory from its disk file, unless the OS supports shared text and a copy of program is already being executed.

Data portion

- The data portion contains the program's data. It is possible for this to be divided into 3 pieces

- Initialized read only data contains elements that are initialized by the program and are read only while the process is executing.

- Initialized read write data contains data elements that are initialized by the program and may have their values modified during execution of the process.

Stack portion

- Un-initialized *data* contains data elements that are not initialized by the program but are set to zero before execution starts .

- The *heap* is used while a process is running to allocate more data space dynamically to the process.

- The *stack* is used dynamically while the process is running to contain the stack frames that are used by many programming languages.

Kernel context

- The stack frames contain the return address linkage for each function call and also the data elements required by a function.
- A gap is shown between heap and stack to indicate that many OS leave some room between these 2 portions, so that both can grow dynamically.
- The ***kernel context*** of a process is maintained and accessible only to the kernel. This area contains info that the kernel needs to keep track of the process and to stop and restart the process while other processes are allowed to execute.

Assessment

- Write a program to demonstrate process related system call including fork(),getpid(),getppid()
- Write a program terminate state of process
- write a program create a zombie process,now extend program to solve the problem of zombie process creation
- Write a program to implement ls –l in a process through execv function.

Timers
Resource Limits
Log Messages

Time Zone

- To display the current time zone:
  execute `date`
- Different types of time zone
- Files are in /usr/share/zoneinfo/Asia/Calcutta
- To change: cp xyz /etc/localtime

Alarm

- unsigned int alarm (unsigned int seconds);
- It is used to set an alarm for delivering SIGALARM signal.
- On success it returns zero.

Interval Timers

- Three interval timers.
  - ITIMER_REAL
    - Decrements in real time, once timer is expired, it raises SIGALRM signal.
  - ITIMER_VIRTUAL
    - Timer decrement is only based on execution of the process, upon expiration SIGVTALRM is delivered.
  - ITIMER_PROF
    - Used to profile the time spent by the process in user as well as kernel mode. SIGPROF is delivered when the timer is expired.

getitimer

- get value of an interval timer
- **int getitimer (int which, struct itimerval *val);**
- On success it returns zero and the timer value is stored in the itimverval structure.
- Example: **ret = getitimer (ITIMER_REAL, val);**

setitimer

- Set value for a interval timer
- **int setitimer (int interval_timers, const struct itimerval *val, struct itimvferval *old_value);**
- On success it returns zero.
- Example**: ret = setitimer(ITIMER_REAL, &value, 0);**

TSC (x86)

- System can provide very high resolution time measurements through the time-stamp counter which counts the number of instructions since boot.

To measure Time Stamp Counter (TSC)

```
# include <sys/time.h>

unsigned long long rdtsc ( )
 {
   unsigned long long dst;
   _ _asm_ _    _ _volatile_ _   ("rdtsc":"=A" (dst));
   return dst;
 }
```

Time measurement

```
main ( )
 {
   long long int start, end;

   start = rdtsc();

   /* Give your job;  */

   end = rdtsc();

   printf (" Difference is : %llu\n", end - start);

}
   /*  This is the most accurate way of time measurement */
```

Resource Limits

- The OS imposes limits for certain system resources it can use.
- Applicable to a specific process.
- The “ulimit” shell built-in can be used to set/query the status.
- “ulimit –a” returns the user limit values

Details of the Resource

-c → Maximum size of “core” files created.

-f → Maximum size of the files created.

-l → Maximum amount of memory that can be locked using mlock() system call.

-n → Maximum number of open file descriptors.

-s → Maximum stack size allowed per process.

-u → Maximum number of processes available to a single user.

Hard Limits

- Each resource has two limits –Hard and Soft
- Hard Limits
  - Absolute limit for a particular resource. It can be a fixed value or “unlimited”
  - Only superuser can set hard limit.
- “ulimit” command has –H or –S option to set hard/soft limits. Default is soft limit.
- Hard limit cannot be increased once it is set.

Soft Limits

- Soft Limits
  - User-definable parameter for a particular resource.
  - Can have a value of 0 till <hard limit> value.
  - Any user can set soft limit.

- Limits are inherited (the new values are applicable to the descendent processes).

getrlimit / setrlimit

- getrlimit()/setrlimit() are system-call interfaces for getting and setting resource limits.
- Syntax
  - getrlimit(<resource>, &r)
  - setrlimit (<resource>, &r)
  - where r is of type “struct rlimit”

```
struct rlimit {
        rlim_t rlim_cur;   /* Soft limit */
        rlim_t rlim_max;   /* Hard limit */
};
```

  - <resource> can be any of the following:

<resource> Details

- <resource> can be any of the following:
  - RLIMIT_FSIZE → Maximum size of the file.
  - RLIMIT_MEMLOCK → Maximum amount of memory that can be locked.
  - RLIMIT_NOFILE → Maximum number of open file descriptors.
  - RLIMIT_STACK → Maximum stack size allowed.
  - RLIMIT_NPROC → Maximum number of process available to a single user.

sysconf

- To print configuration information and system limits during runtime.

- `long sysconf (int query);`

- Example: `ret= sysconf(_SC_CLK_TCK);`

- On success it returns the value of the system limits.

syslog

- Used by applications to “log” error & status messages to a log file (ex: /var/log/messages).
- Managed by the “system logger” (syslogd) daemon.
- Syslogd can log messages to:
  – file (locally or across the network)
  – pipe
  – terminal/console

- **`$man syslogd`** gives detail

syslog ( ) Library Function

- Open a connection from the program to “system logger”

void **openlog**(const char *ident, int option, int facility);

  - ident is for identification (eg: program name)
  - option is for flag to control the operation of logging (eg: LOG_CONS, LOG_PID etc.)
  - facility is type of program logging the message (eg: LOG_KERN, LOG_MAIL, LOG_FTP etc.)

syslog ( ) Library Function (Contd.).

- Do the actual logging of messages

void syslog(int priority, const char *format, ...)
priority may be LOG_ALERT, LOG_CRIT, LOG_ERR etc.)
format → similar to printf formats (eg: %6d, %s etc.)

- where the messages get logged depends on the priority and syslogd configuration

- Close the connection (any open file descriptors being used to “log”). Syntax: void closelog(void);

syslog -- Example

```
#include <syslog.h>
{
….
char *ident = “Syslog demo";
int option = LOG_CONS | LOG_PID;
int facility = LOG_USER;
int priority = LOG_USER | LOG_ERR;
int i = 1;

openlog(ident, option, facility);
syslog(priority, “Demo no: %3d”, i);
closelog();
…
}
```

The output in /var/log/messages will be similar to:

May 14 04:07:40 mickey syslog demo[15218]: Demo no: 1

syslog – related commands/ functions

- int setlogmask(int mask)
  - Sets a log priority mask.
  - Any calls by the process to syslog() with a priority not set in mask will be rejected

- syslog.conf (Mostly present in /etc)
  - syslog configuration file ($man syslog.conf gives more details)

Assessment

- List the alarm() and sleep() system calls behaviors
- what is log file ?
- Which utility is used to make automate rotation of a log?

Signaling Mechanisms

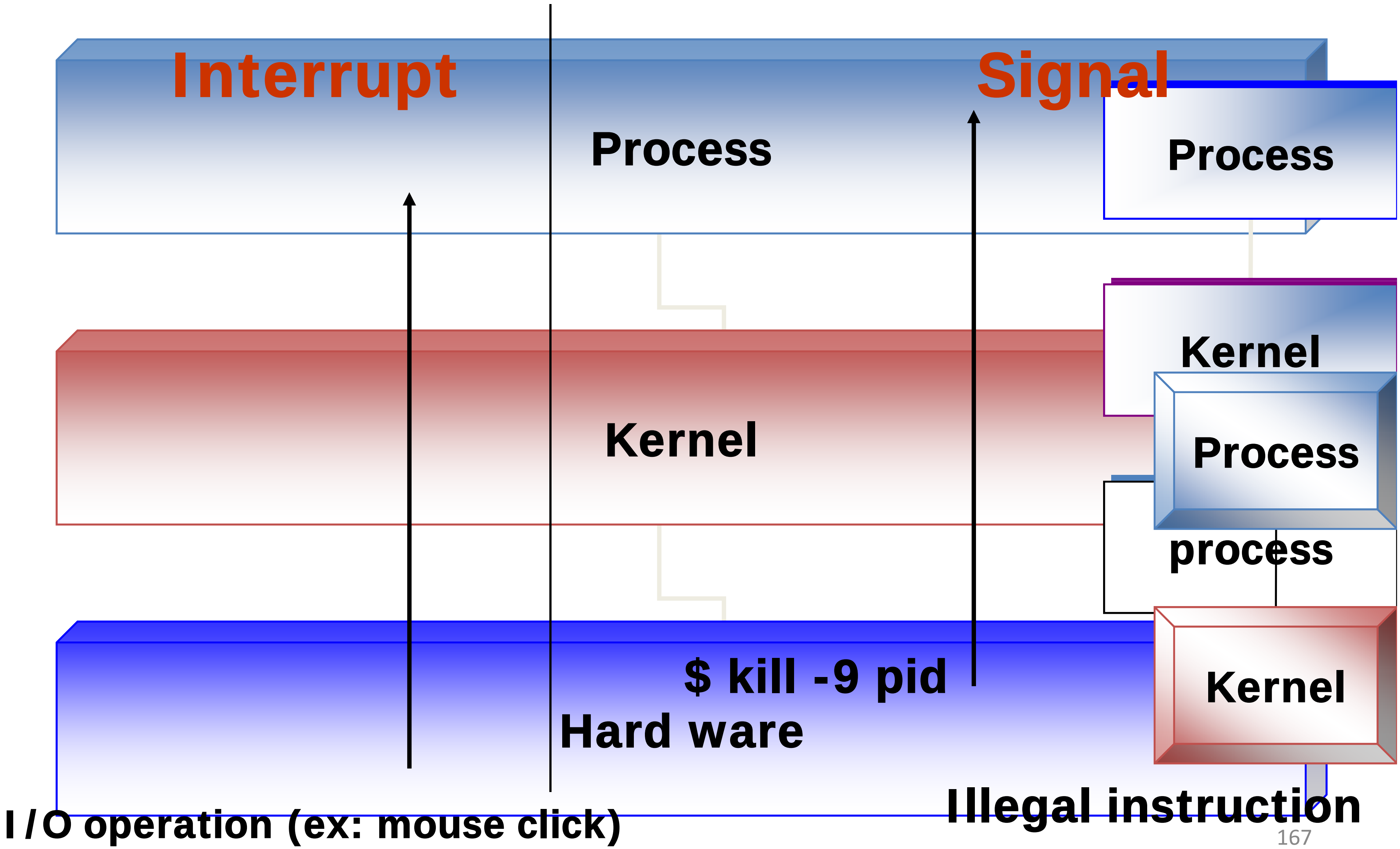
Introduction

- Signals are a fundamental method for inter process communication and are used in everything from network servers to media players.
- A signal is generated when
  - an event occurs (timer expires, alarm, etc.,)
  - a user quota exceeds (file size, no of processes etc.,)
  - an I/O device is ready
  - encountering an illegal instruction
  - a terminal interrupt like Ctrl-C or Ctrl-Z.
  - some other process send (kill -9 pid)

Introduction (Contd.).

- Each signal starts with macro SIGxxx.
- Each signal may also specifies with its integer number
- For help: $ kill –l , $ man 7 signal
- When a signal is sent to a process, kernel stops the execution and "forces" it to call the signal handler.
- When a process executes a signal handler, if some other signal arrives the new signal is blocked until the handler returns.

Signal Vs Interrupt



Receiving a Signal

- How a process receives a signal, when it is
  - executing in user mode
  - executing in kernel mode
  - not running
  - in interruptible sleep state
  - in uninterruptible sleep state

Handling a Signal

- When a signal occurs, a process could
  - Catch the signal
  - Ignore the signal
  - Execute a default signal handler

- Two signals that cannot be caught or ignored
  - SIGSTOP
  - SIGKILL

signal system call

- signal system call is used to catch, ignore or set the default action of a specified signal.
- int signal (int signum, (void *) handler);
- It takes two arguments: a signal number and a pointer to a user-defined signal handler.
- Two reserved predefined signal handlers are :
  - SIG_IGN
  - SIG_DFL.

kill system call

- Kill system call is used to send a given signal to a specific process
- int kill (pid_t process_id, int signal_number);
- it accepts two arguments, process ID and signal number
- If the pid is positive, the signal is sent to a particular process.
- If the pid is negative, the signal is sent to the process whose group ID matches the absolute value of pid.

sigaction system call

- Same as signal( ) but it has lot of control over a given signal.
- The syntax of sigaction is:

int sigaction (int signum, const struct sigaction *act,

struct sigaction *oldact);

  - **signum**, is a specified signal
  - **act** is used to set the new action of the signal signum;
  - **oldact** is used to store the previous action, usually NULL.

Primitive Inter Process Communications

Introduction

- In a multiprocessing environment, often many processes are in need to communicate with each other and share some of the resources.

- The shared resources must also be synchronized from the concurrent access by many processes.

- IPC mechanisms have many distinct purposes: for example

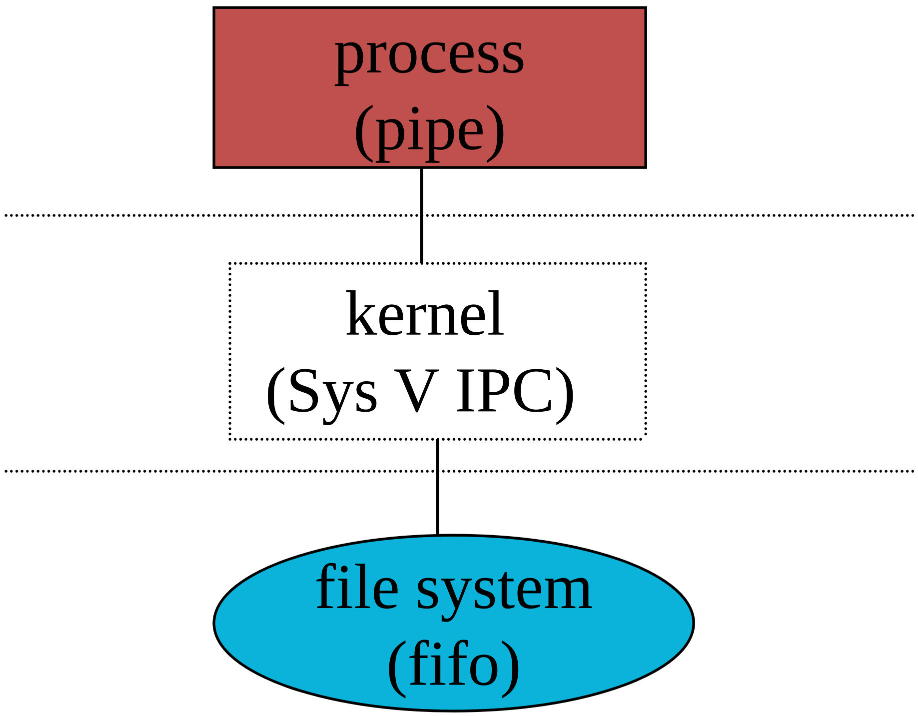
* Data transfer
* Sharing data
* Event notification
* Resource sharing
* Process control

IPC Mechanisms

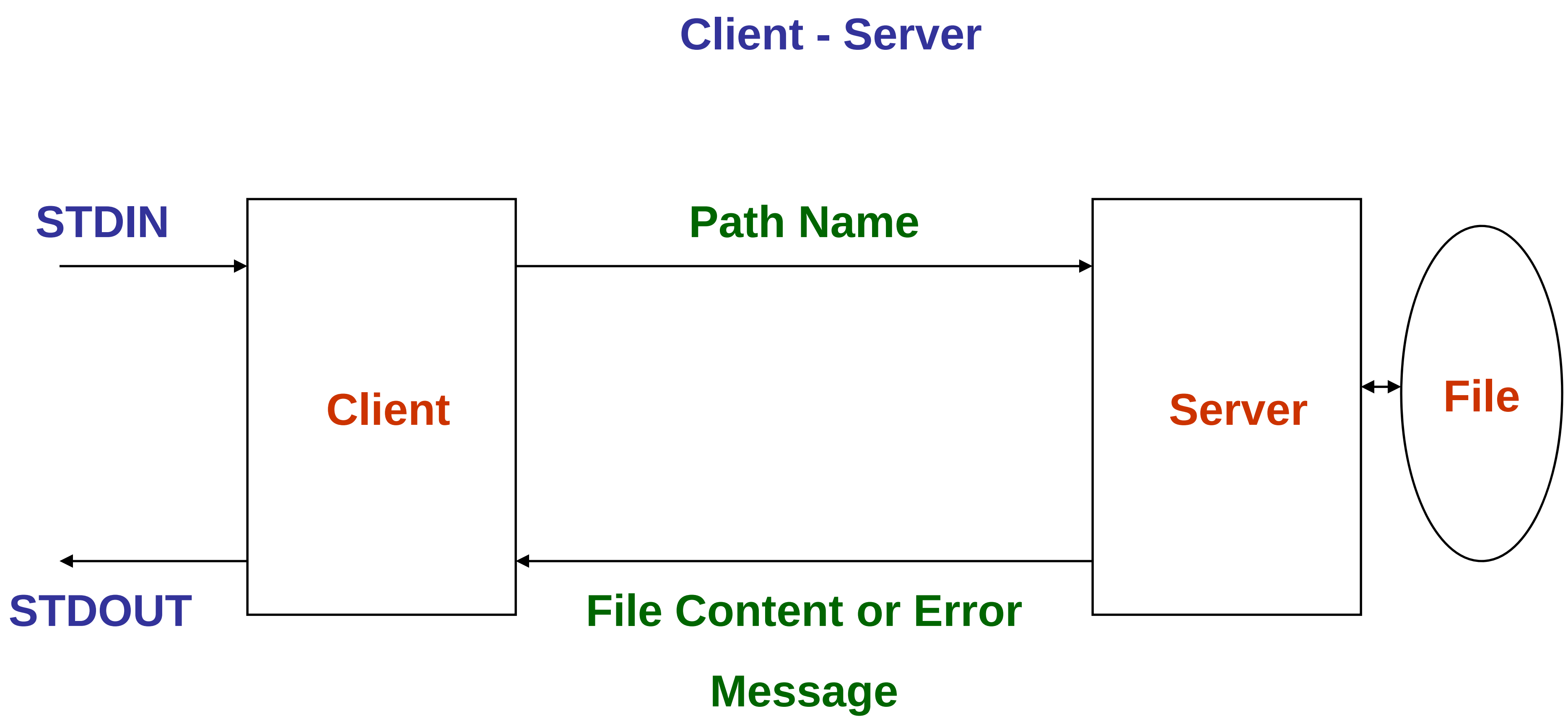
- Primitive
  - Unnamed pipe
  - Named pipe (FIFO)
- System V IPC
  - Message queues
  - Shared memory
  - Semaphores
- Socket Programming

Persistence of IPC Objects

Persistence: once an IPC object is created how long it will be in the system.



PIPE: Example



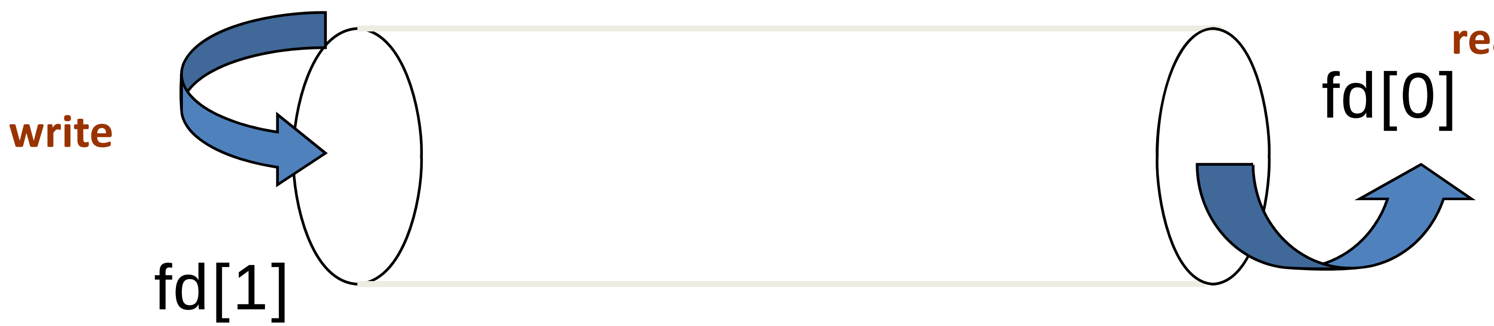
Unnamed pipe or pipe

- On command line pipe is represented as “|”
- It can be used in the shell to link two or more commands
    - For example ls –Rl | wc
- Two ends of a pipe is represented as a set of two descriptors.
- A pipe is used to communicate between **related processes.**

Pipe

- Half duplex
- Data is passed in order.
- Pipe uses circular buffer and it has zero buffering capacity
- The read and write system calls are blocking calls.

Pipe –one way communication



Creation of a pipe

- int fd[2];
- pipe(fd);

  -returns with fd[0], fd[1];

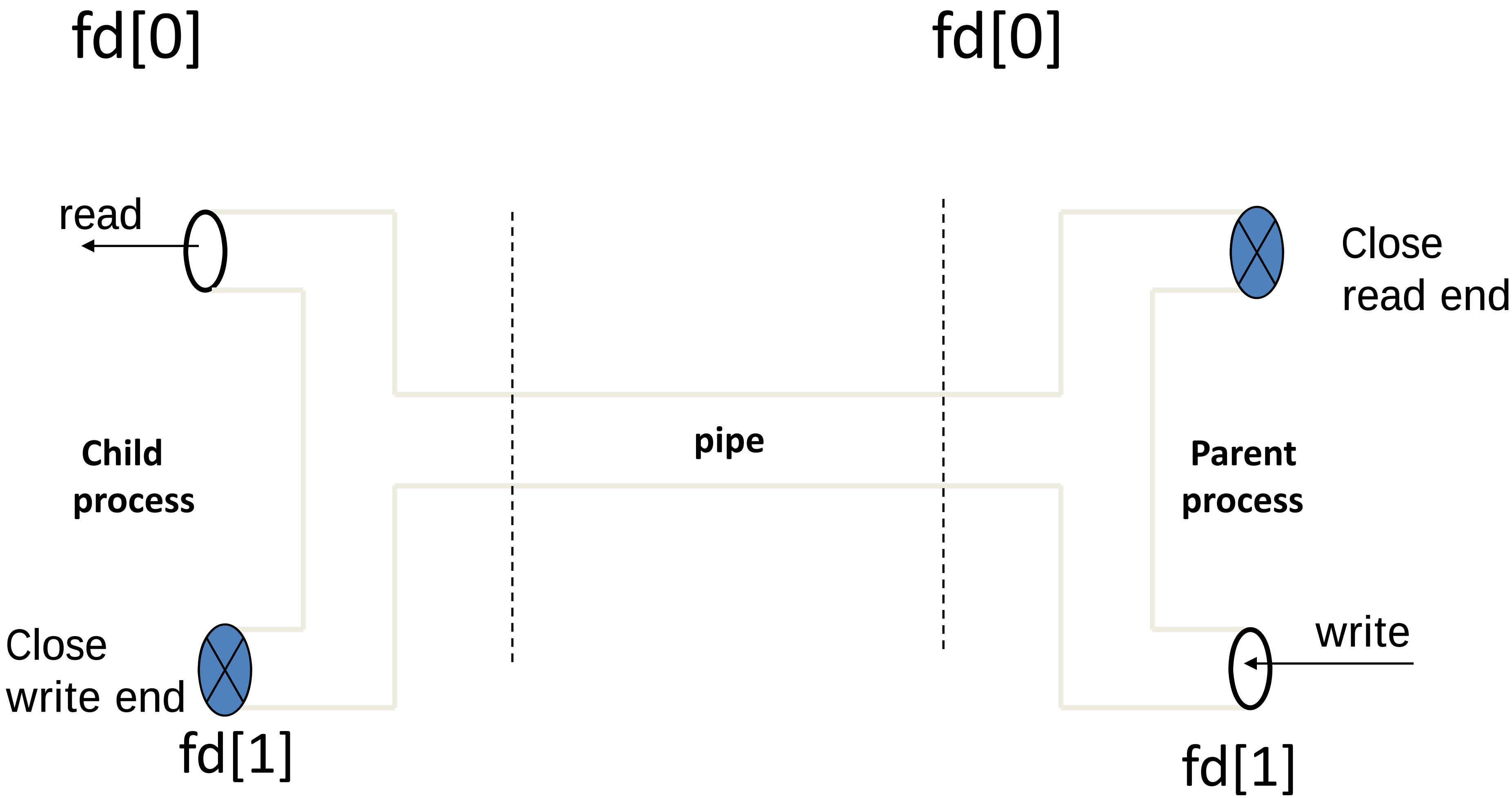
- write(fd[1], ...);
- read(fd[0], ...);

One way communication

- Create a pipe.
- Call fork.
- Parent can send data and child can read the data or vice versa.
- Unused ends (descriptors) should be closed.

One way communication using Pipe

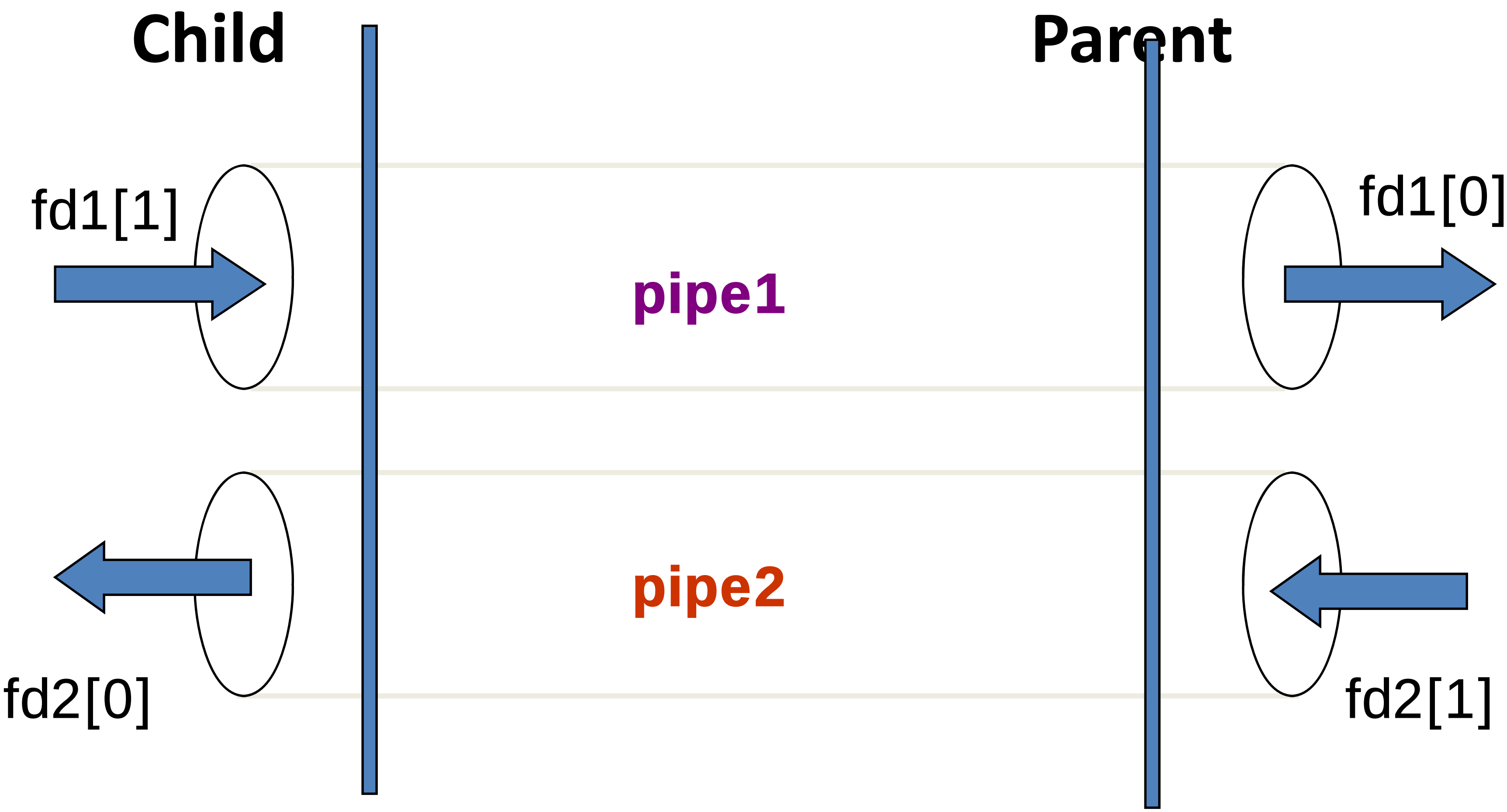
One-way communication from parent to child



Two way communication

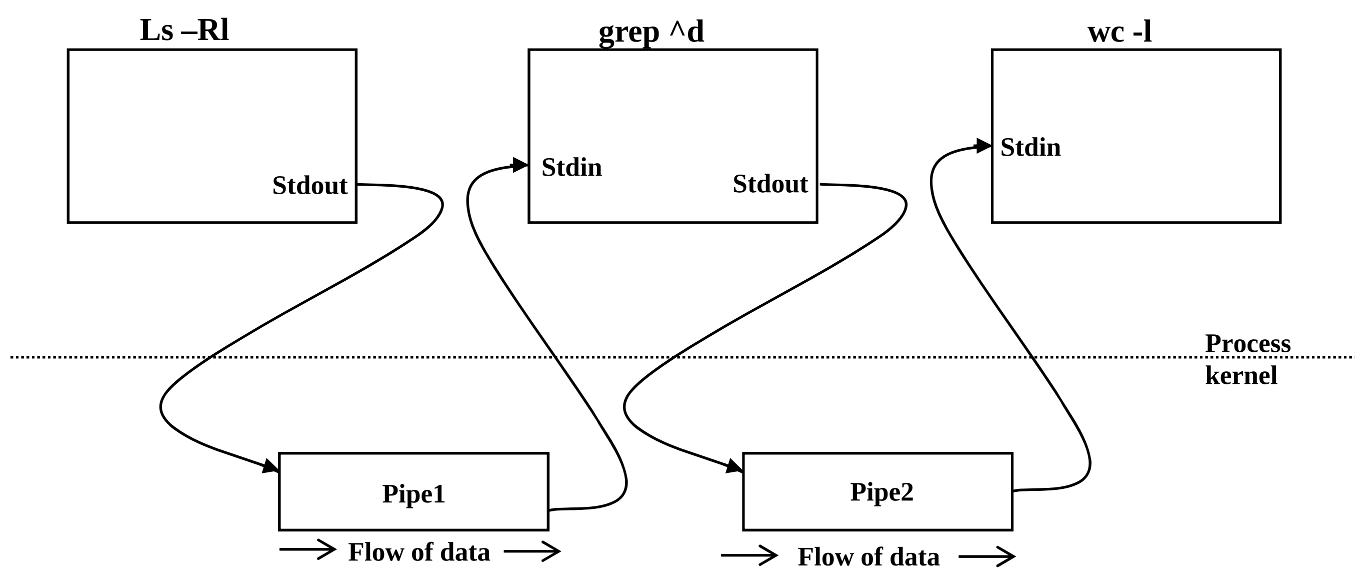
- Create two pipes say fd1, fd2.
- Four descriptors for each process (fd1[0], fd1[1], fd2[0], fd2[1]).
- Parent closes read end of fd1 and write end of fd2
    - close(fd1[0], fd2[1]);
- child closes read end of fd2 and write end of fd1
    - close(fd2[0], fd1[1]);

Two Way Communication (Contd.).



Usage of Two Pipes

- Execution of command: **$ ls –Rl | grep ^d | wc -l**



pipe

Advantages:

- Simplest form of IPC
- Persistence in process level
- Can be used in shell

Disadvantages:

- Cannot be used to communicate between unrelated processes

popen

- The popen library function opens a process by creating a pipe, execute fork and invoke a shell.
- FILE *popen (const char *executable, const char *mode);
- On success returns file pointer else NULL (if fork or pipe system calls failed).
- int pclose (FILE *STREAM);

pread / pwrite

- Used to read or write to a file at a specified offset value.
- Same as read/write except the starting position is from the given offset.
- On success the system calls return number of bytes read or written.
- If 0 returns
  - pwrite: nothing has been written
  - pread: end of file

FIFO - Introduction

- FIFO works much like a pipe
  - Half duplex, data passed in FIFO order, circular buffer and zero buffering capacity.
- FIFO is created on a file system as a device special file
- It can be used to communicate between unrelated processes
- It can be reused.
- Persist till the file is deleted.

FIFO Creation

- FIFO can be created in a shell by using mknod or mkfifo command.
  - mknod myfifo p
  - mkfifo a=rw myfifo
- In a C program mknod system call or mkfifo library function can be used.
  - int mkfifo (char *file_name, mode_t mode);
  - int mknod (char *file_name, mode_t mode, dev_t dev);
    - mknod(“./MYFIFO", S_IFIFO|0666, 0);

Using FIFO

- Once a FIFO is created either from a shell or through a program, file’s related system calls (open, read, write, select, close etc.,) are used to access the FIFO.

- For example: Process 1 may open a FIFO in write only mode and write some data.

- Process 2 may open the FIFO in read only mode, read the data and display on the monitor.

FIFO - Disadvantages

- Data cannot be broadcast to multiple receivers.
- If there are multiple receivers, there is no way to direct to a specific reader or vice versa.
- Cannot be used across network
- Less secure than a pipe, since any process with valid access permission can access data.
- Cannot store data
- No message boundaries. Data is treated as a stream of bytes.

System Limitations

- System imposed limits on pipes

1. Maximum number of files can be open within a process is determined by OPEN_MAX macro.

2. Maximum amount of data that can be written to a pipe of FIFO atomically is determined by PIPE_BUF macro (size of a circular buffer).

Tracing processes

- strace command
  - trace system calls and signals
  - strace runs until the given command exits
  - It is a useful tool for diagnostic, instructional and debugging
- ptrace system call
  - Process trace

strace

#strace -c -e trace=file mkfifo -m 0744 myfifo

execve("/usr/bin/mkfifo", ["mkfifo", "-m", "0744", "myfifo"]) = 0

```
% time   seconds    us/call   calls   syscall
------ ----------- ----------- --------- --------- ----------------
 47.62   0.000020       20        1      mknod
 33.33   0.000014        4        4      open
 11.90   0.000005        5        1      chmod
  7.14   0.000003        1        3      fstat
------ ----------- ----------- --------- --------- ----------------
100.00   0.000042                 9
```

ptrace

- Used to observe and control execution of another process
- Can even change a process' core image and registers (complete control)
- Main uses
  - Debugging (breakpoints in gdb)
  - System call traces (strace, truss etc.)
- Most programs will NOT face the need to use the ptrace() system call

Daemon Process

Introduction

- Daemon process starts during system startup
- They frequently spawn other process to handle services requests
    - Mostly started by initialization script /etc/rc
- Waits for an event to occur
- perform some specified task on periodic basis (cron job)
- perform the requested service and wait
    - Example print server

Characteristics

- executed at the background process
- Orphan process
- No controlling terminal
- run with super user privileges
- process group leaders
- session leaders

Daemon -program

```
int  init_daemon ( void ) {
     if ( ! fork ( ) ) {
               setsid ( ) ;
               chdir ( " / " ) ;
               umask ( 0 ) ;
      /* Specify Your Job  */
               return ( 0 );
     }
     else
               exit ( 0 );
}
```

System V IPC

Introduction

- Pipe and FIFO do not satisfy many requirements of many applications.
- Sys V IPC is implemented as a single unit
- System V IPC Provides three mechanisms namely
  - Message Queues,
  - Shared Memory
  - Semaphores.
- Persist till explicitly delete or reboot the system

Common Attributes

- Each IPC objects has the following attributes.
  - key
  - id
  - Owner
  - Permission
  - Size
    - Message queue – used-bytes, number of messages
    - Shared memory – size, number of attach, status
    - Semaphore – number of semaphores in a set
- The ipc_perm structure holds the common attributes of the resources.

System Limitations

```
$ ipcs -l
------ 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 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

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

Get a key

- If we wish to communicate between different processes using an IPC resource, the first step is to create a shared unique identifier.
- The system generates a number dynamically for a given mechanism by using the *ftok* library function.
- But apart from the creator, other processes that want to communicate with the creator process should agree to the key value.
- Syntax: key_t ftok (const char *filename, int id);

Get an id

- The syntax for a *get* function is:

int xxxget (key_t key, int xxxflg);

(xxx may be msg or shm or sem)

- If successful, returns to an identifier; otherwise -1 for error.

- The key can be generated in three different ways
  - from the *ftok* library function
  - by choosing some static positive integer value
  - by using the IPC_PRIVATE macro
- flags commonly used with this function are IPC_CREAT and IPC_EXCL.

Control a object

- The syntax for the *control* function is:

int xxxctl (int xxxid, int cmd, struct xxxid_ds *buffer);
(xxx may be msg or shm or sem);

- If successful, the *xxxctl* function returns zero, otherwise it returns -1.
- The command argument may be
  - IPC_STAT
  - IPC_SET
  - IPC_RMID

Message Queues - Introduction

- Message queue overcomes FIFO limitation like storing data and setting message boundaries.
- Create a message queue
- Send message (s) to the queue
- Any process who has permission to access the queue can retrieve message (s).
- remove the message queue.

Message queues

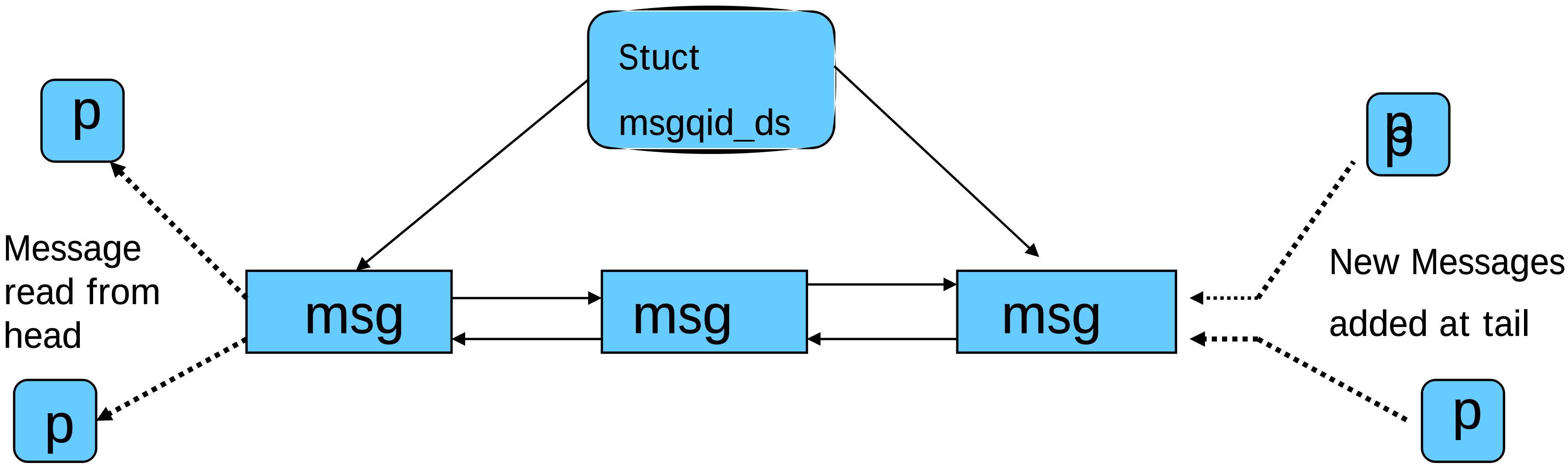
```
struct msgbuf {
            long mtype;
              char mtext [1];
}; Standard structure
```

```
struct My_msgQ {
            long mtype;
              char mtext [1024];
              void *   xyz;
}; Our own structure
```

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

Messages in a queue



msqid_ds

```
struct msqid_ds
{
  struct ipc_perm msg_perm;
  __time_t msg_stime;
      __time_t msg_rtime;
  __time_t msg_ctime;
  unsigned long int __msg_cbytes;
  msgqnum_t msg_qnum;
  msglen_t msg_qbytes;
  __pid_t msg_lspid;
  __pid_t msg_lrpid;
};
```

msgget

- **int msgget (key_t key, int msgflg);**
- The first argument key can be passed from the return value of the ftok function or made IPC_PRIVATE.
- To create a message queue, IPC_CREAT ORed with access permission is set for the msgflg argument.
- Ex: msgid = msgget (key, IPC_CREAT | 0744);
  msgid = msgget (key, 0);

msgsnd

- The syntax of the function is:

int msgsnd (int msqid, structu msgbuf *msgp, size_t msgsz, int msgflg);

- Arguments:
  – message queue ID
  – address of the structure.
  – size of the message text
  – message flag
    - 0 or IPC_NOWAIT

msgrcv

- syntax of the function is:

ssize_t msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);

- msgtype argument is used to retrieve a particular message.
  - 0 -retrieve in FIFO order
  - +ve - retrieve the the exact value of the message type
  - –ve - first message or <= to the absolute value.

- on success, msgrcv returns with the number of bytes actually copied into the message text

Destroying a message queue

- There are many ways:
- From command line, using one of the ways
  - $ ipcrm msg msqid
  - $ ipcrm –q msqid
  - $ ipcrm –Q msgkey
- Using system call
  - msgctl (msgid, IPC_RMID, 0);

Message queue : 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 (msqid, IPC_RMID, NULL);
- $ipcrm msg msqid

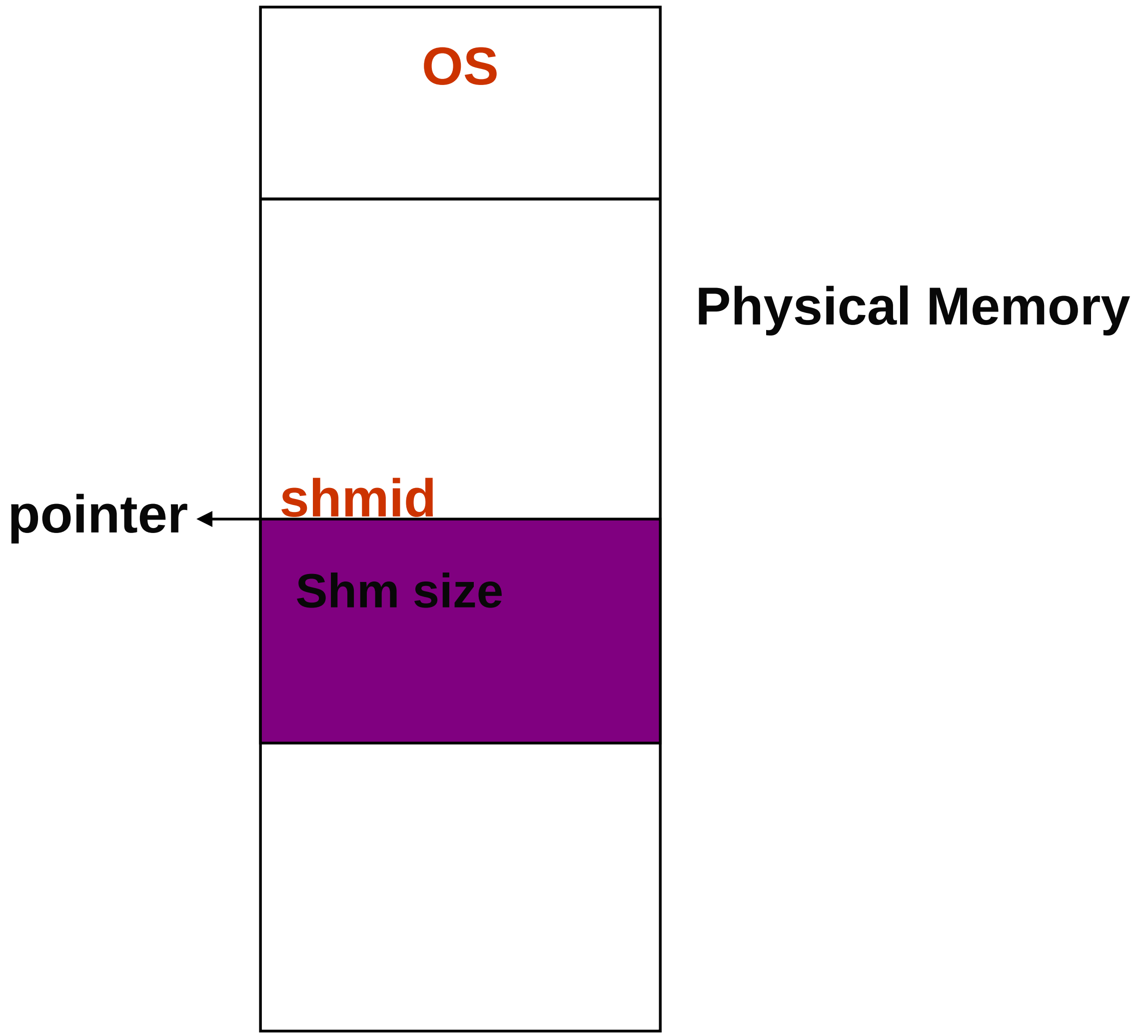
Limitations

- Message queues are effective if a small amount of data is transferred.
- Very expensive for large transfers.
- During message sending and receiving, the message is copied from user buffer into kernel buffer and vice versa
- So each message transfer involves two data copy operations, which results in poor performance of a system.
- A message in a queue can not be reused

Shared Memory - Introduction

- Very flexible and ease of use.
- Fastest IPC mechanisms
- shared memory is used to provide access to
  - Global variable
  - Shared libraries
  - Word processors
  - Multi-player gaming environment
  - Http daemons
  - Other programs written in languages like Perl, C etc.,

Shared Memory – shmat ( )



Why go for Shared Memory

- Shared memory is a much faster method of communication than either semaphores or message queues.
- does not require an intermediate kernel buffer
- 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 - Data structures

- The data structures used in shared memory are:
  - shmid_ds
  - ipc_perm
  - shminfo
  - shm_info
  - shmid_kernel

ipc_perm structure

```
struct ipc_perm
 {
  __key_t __key;               - Key
  __uid_t uid;                - Owner's user ID
  __gid_t gid;               - Owner's group ID
  __uid_t cuid;             - Creator's user ID
  __gid_t cgid;            - Creator's group ID
  unsigned short int mode;   - r/w permission unsigned
   short int __seq;  - Sequence number
};
```

shmid_ds

```
 struct shmid_ds
{
 struct ipc_perm shm_perm;
 size_t shm_segsz;
 __time_t shm_atime;
 __time_t shm_dtime;
 __time_t shm_ctime;
 __pid_t shm_cpid;
 __pid_t shm_lpid;
 shmatt_t shm_nattch;
};
```

Steps to access Shared Memory

- The steps involved 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

shmat

- used to attach the created shared memory segment onto a process address space.
- **`void *shmat(int shmid,void *shmaddr,int shmflg)`**
- Example: **`data=shmat(shmid,(void *)0,0);`**
- A pointer is returned on the successful execution of the system call and the process can read or write to the segment using the pointer.

Reading/ writing to SM

- Reading or writing to a shared memory is the easiest part.
- The data is written on to the shared memory as we do it with normal memory using the pointers
- Eg. Read:
- printf(“SHM contents : %s \n”, data);
- Write:
- prinf(“”Enter a String : ”);
- scanf(“ %[^\n]”,data);

Shmdt & shmctl

- The detachment of an attached shared memory segment is done by shmdt to pass the address of the pointer as an argument.

- Syntax: int shmdt(void *shmaddr);

- To remove shared memory call:

int shmctl(shmid,IPC_RMID,NULL);

- These functions return –1 on error and 0 on successful execution.

Shared Memory – pseudo code

- shmid = shmget (key, 1024, IPC_CREAT|0744);
- 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);

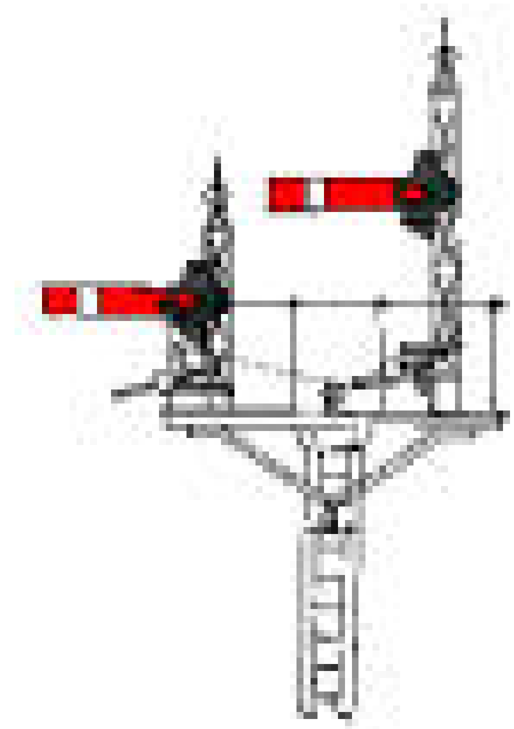
Limitations

- Data can either be read or written only. Append is not allowed
- Race condition
  - Since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes. Since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent the race condition in the shared memory.

Semaphores

Synchronization Tool
An Integer Number
P ( ) And V ( ) Operators
Avoid Busy Waiting

Types of Semaphore



Used in :

shared memory segment

message queue

file

Semaphores (Contd.).

- 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

System V IPC – 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
 };

union semun arg;

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: lock or unlock */
     short sem_flg;  /* operation flags : 0, SEM_UNDO, IPC_NOWAIT */
  };
struct sembuf buf = {0, -1, 0};  /* (-1 + previous value) */
semid = semget (key, 1, 0);

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

Assessment

- Write 2 program that will communicate via shared memory and semaphore. Data will be exchanged via memory and semaphore will be used to synchronize and notify each process when operations such as memory loaded and memory read have been performed

Assessment

- IPC
- Create two process A and B and implement IPC between them using message queue as below
- Process A – Hello, can you hear me
- Process B – Yes, Loud & clear
- Process A – I too can hear you
- Process B – Ok bye!
- Process A – Bye Bye.

Bibliography

1. Stevens. W R. Unix Network Programming Volume I & II
2. Chan, Terrance. Unix System Programming Using C++
3. **Linux System Programming** Talking Directly to the Kernel and C Library by Robert Love

References :

http://linuxdocs.org/HOWTOs/DB2-HOWTO/kernel24.html

Thank You