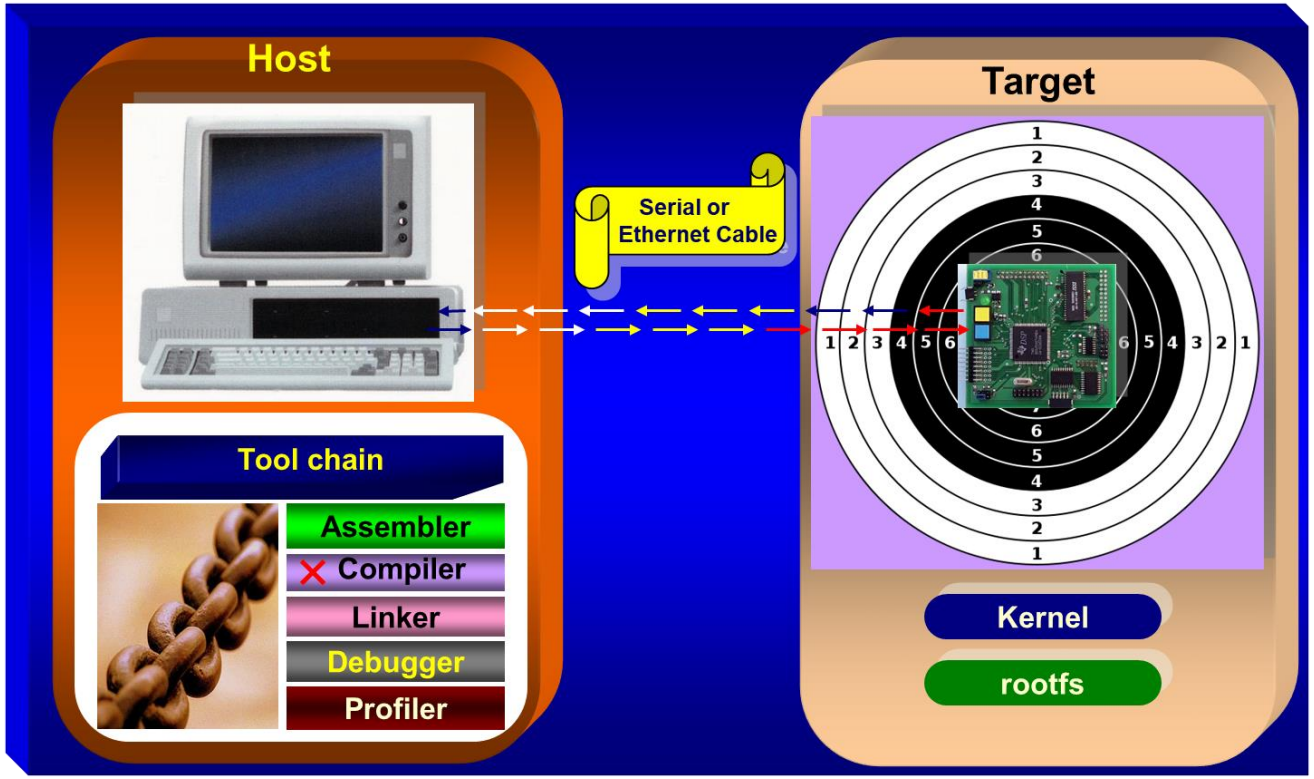
System Software



Prof. B Thangaraju

Agenda

1: Computer Architecture

- Basic structure of computer hardware and software
- Process, Memory and I/O systems: CPU, RAM, Virtual Memory, I/O devices
- Types of System - Server, Desktop, Embedded and Real Time
- Operating System Vs Kernel

2. Kernel Architecture

- Kernel Subsystems (computing resource management)
- Types of Kernel: Monolithic, Micro and Hybrid Architecture
- Monolithic - Server and Desktop
- Microkernel - Embedded and Real Time systems
- Hybrid - Handle both RT and Non-RT tasks

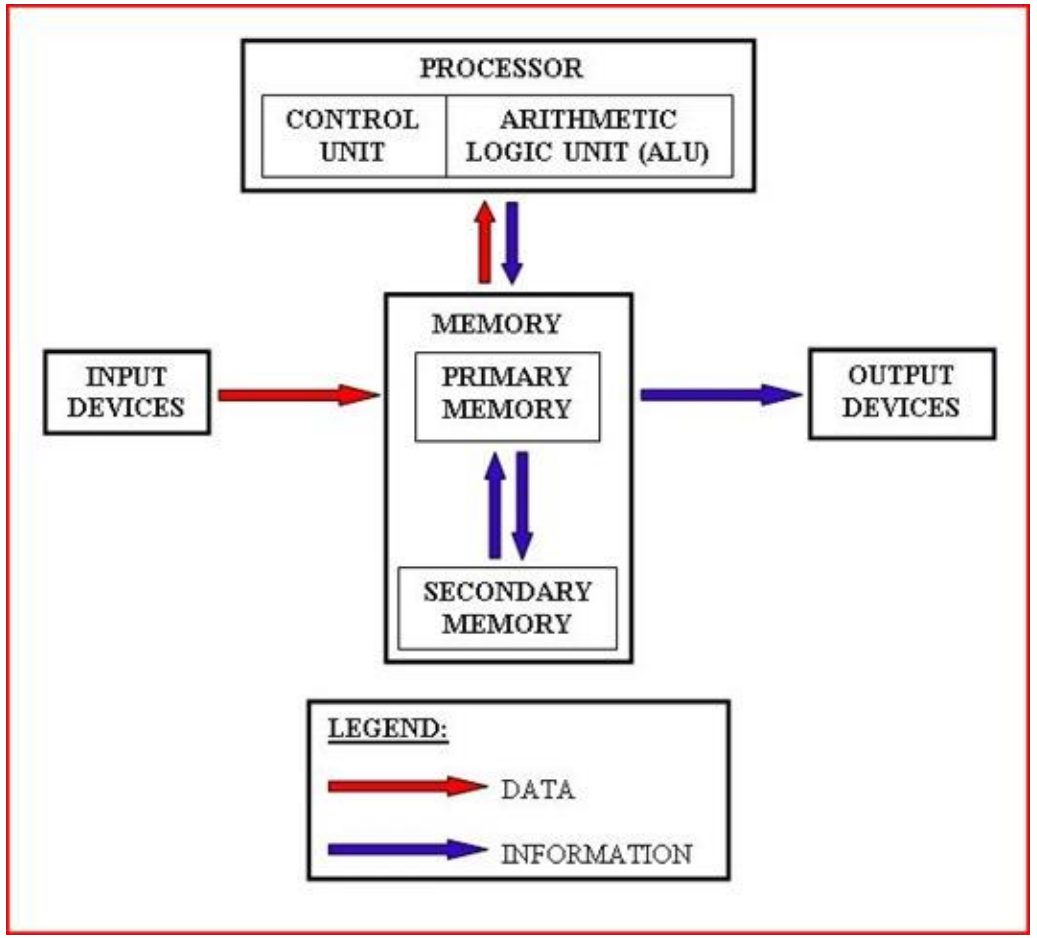
3. System Internals

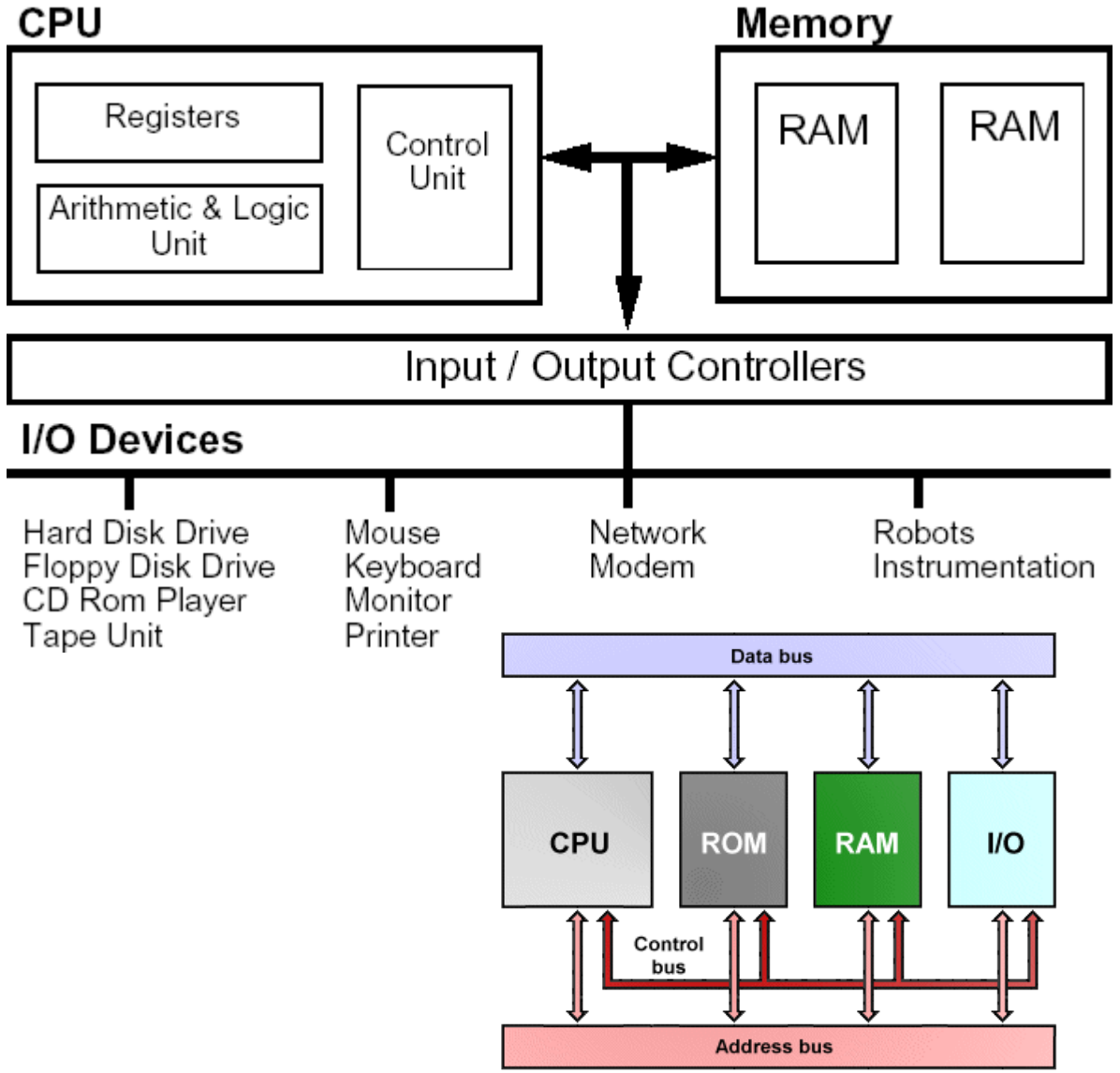
- Brief implementation of - process, file, memory and signal management
- Communication Mechanisms - pipe, FIFO, message Q, shared memory

4: Synchronization Mechanisms - File and process

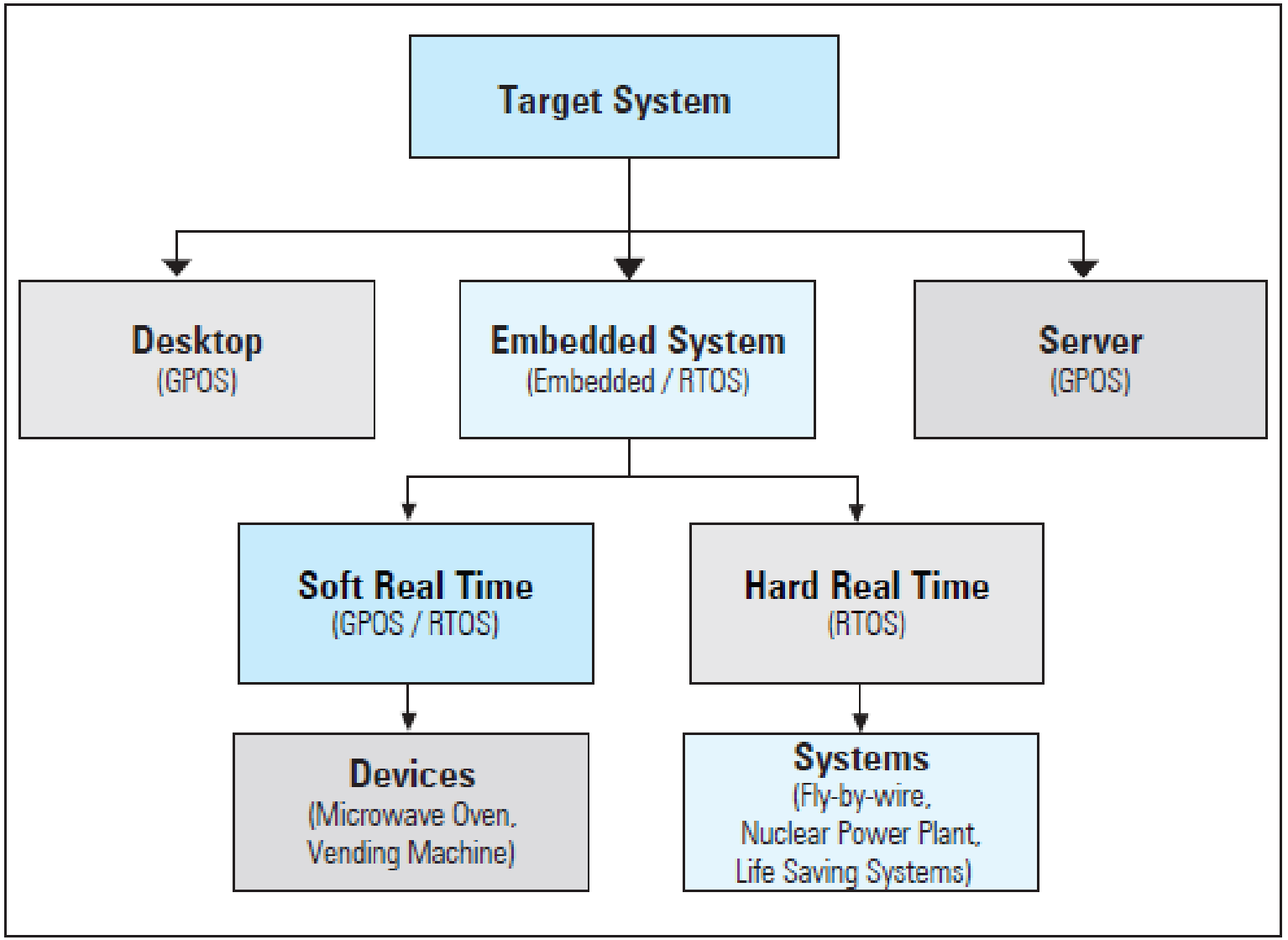
- Implementation of Soft Real Time Systems - as per POSIX standard
- Application Program Vs Kernel Module

Computer Architecture

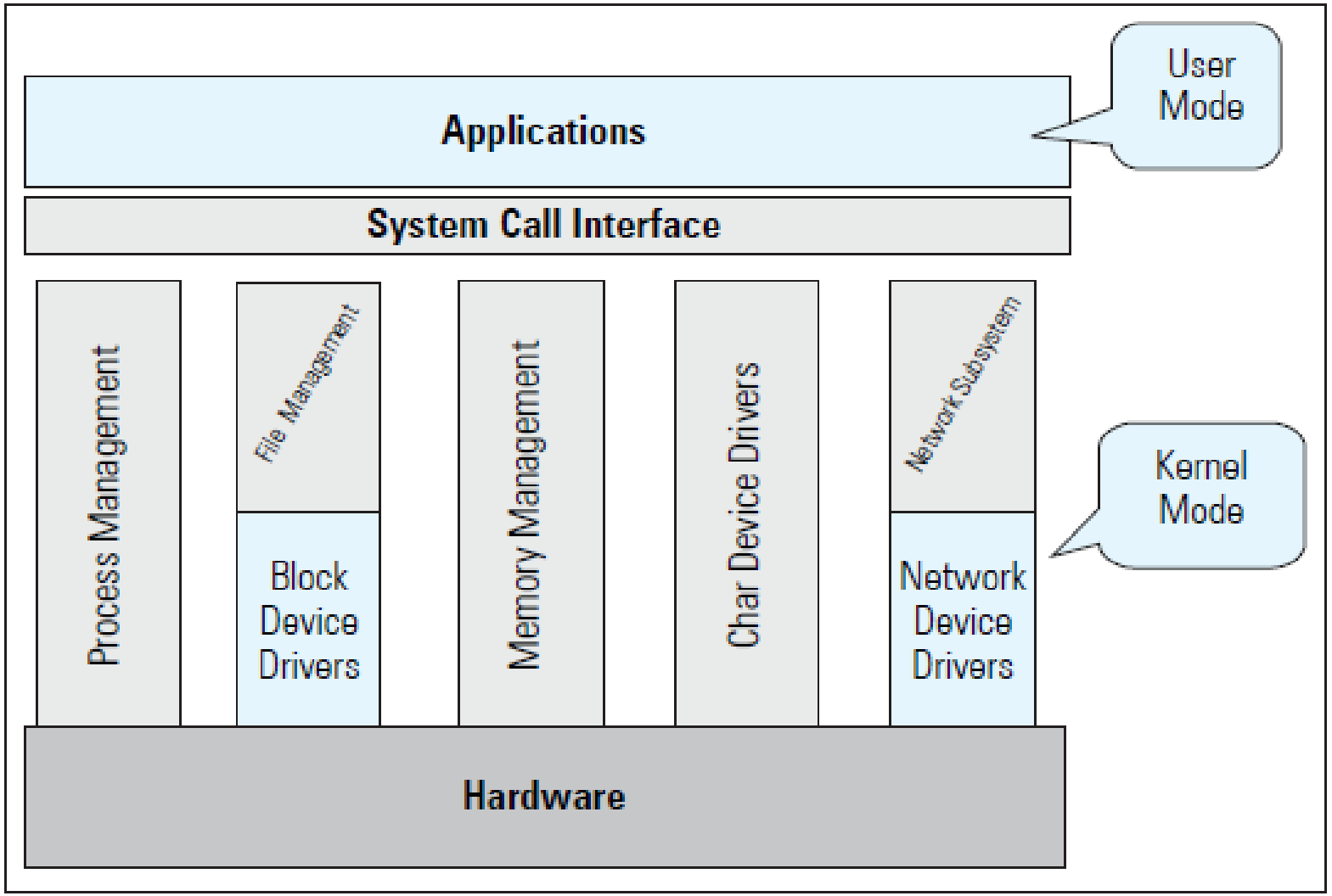




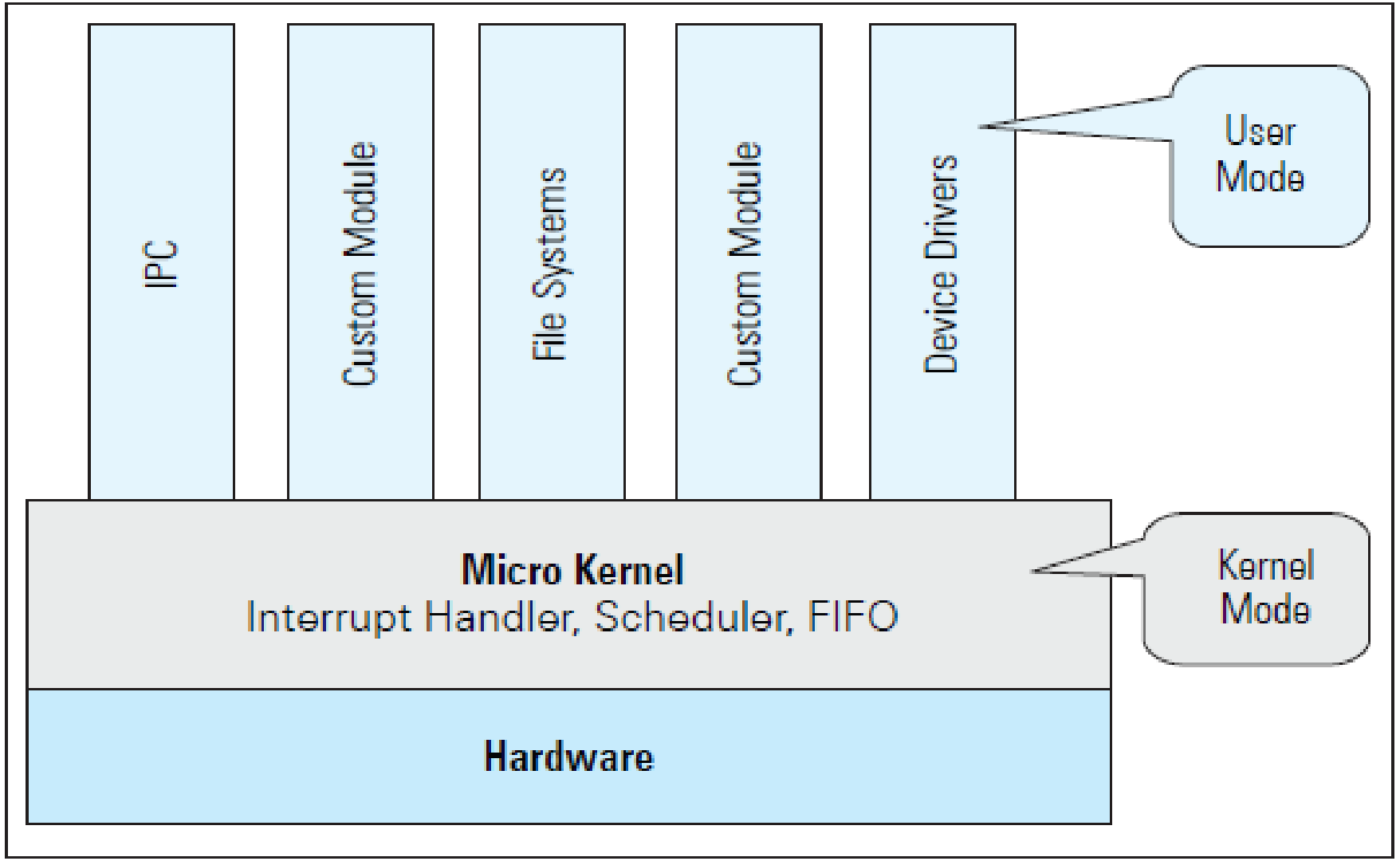
Types of System



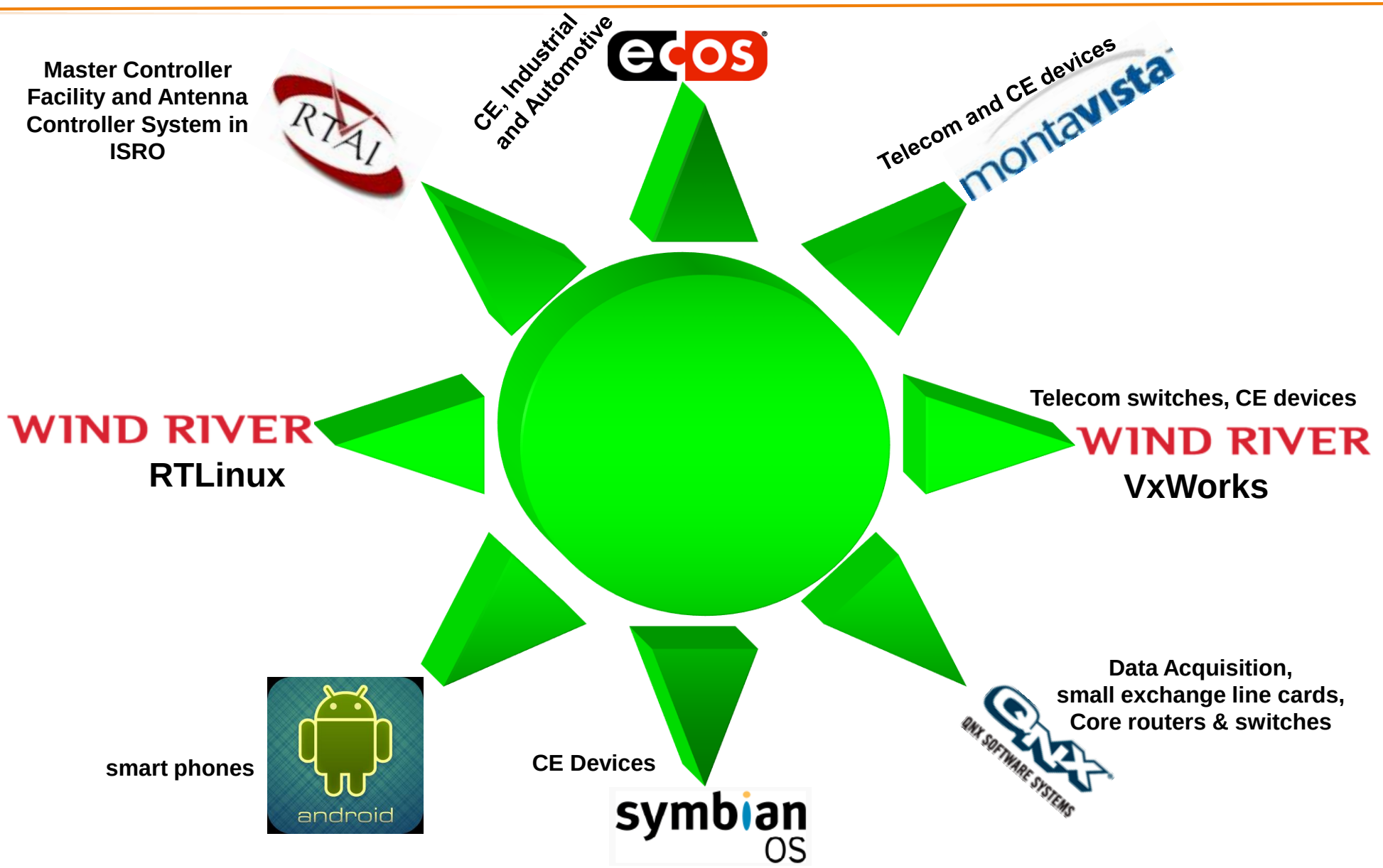
GPOS


Applications
User Mode
System Call Interface
Process Management
File Management
Block Device Drivers
Memory Management
Char Device Drivers
Network Subsystem
Network Device Drivers
Kernel Mode
Hardware

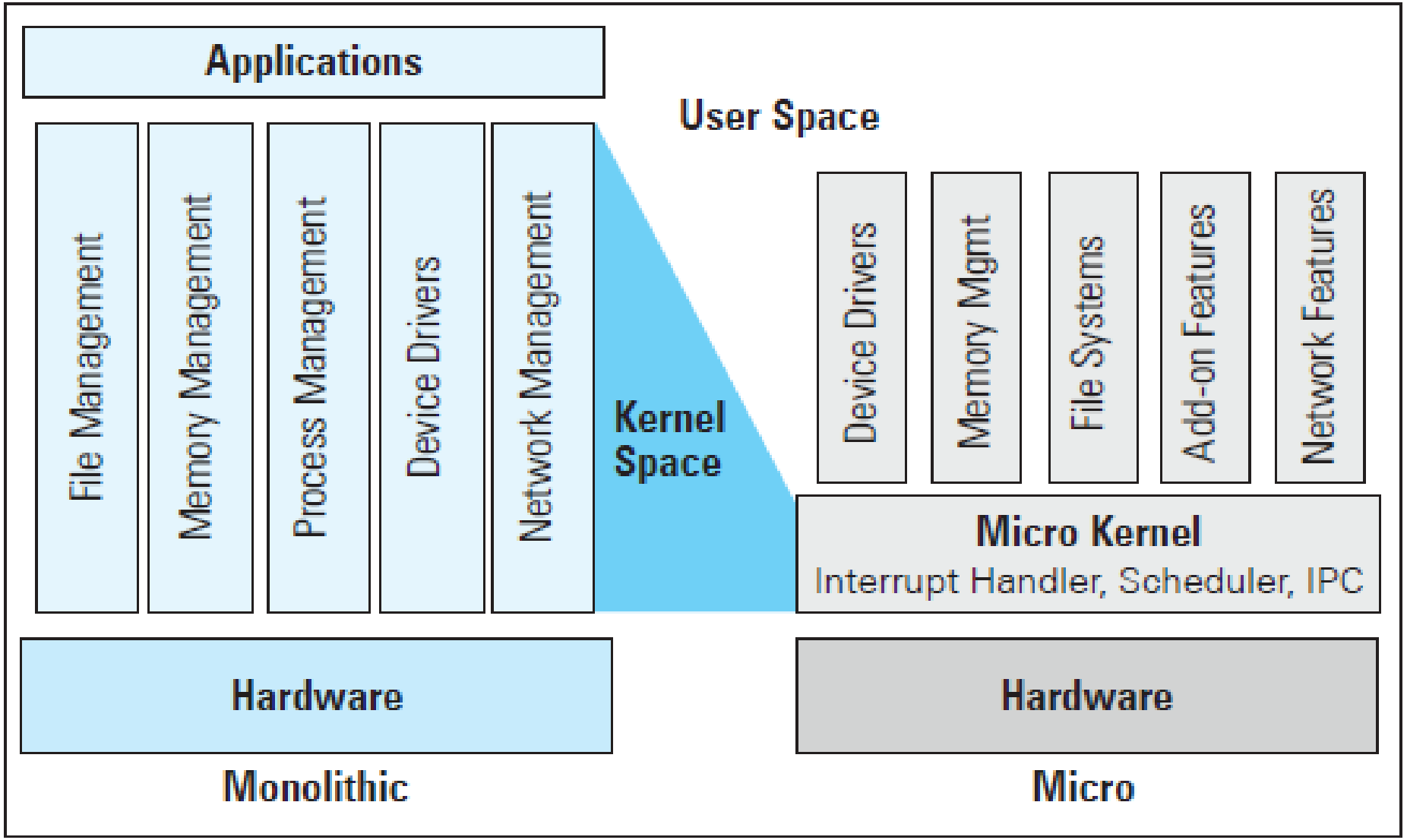
Micro Kernel Architecture



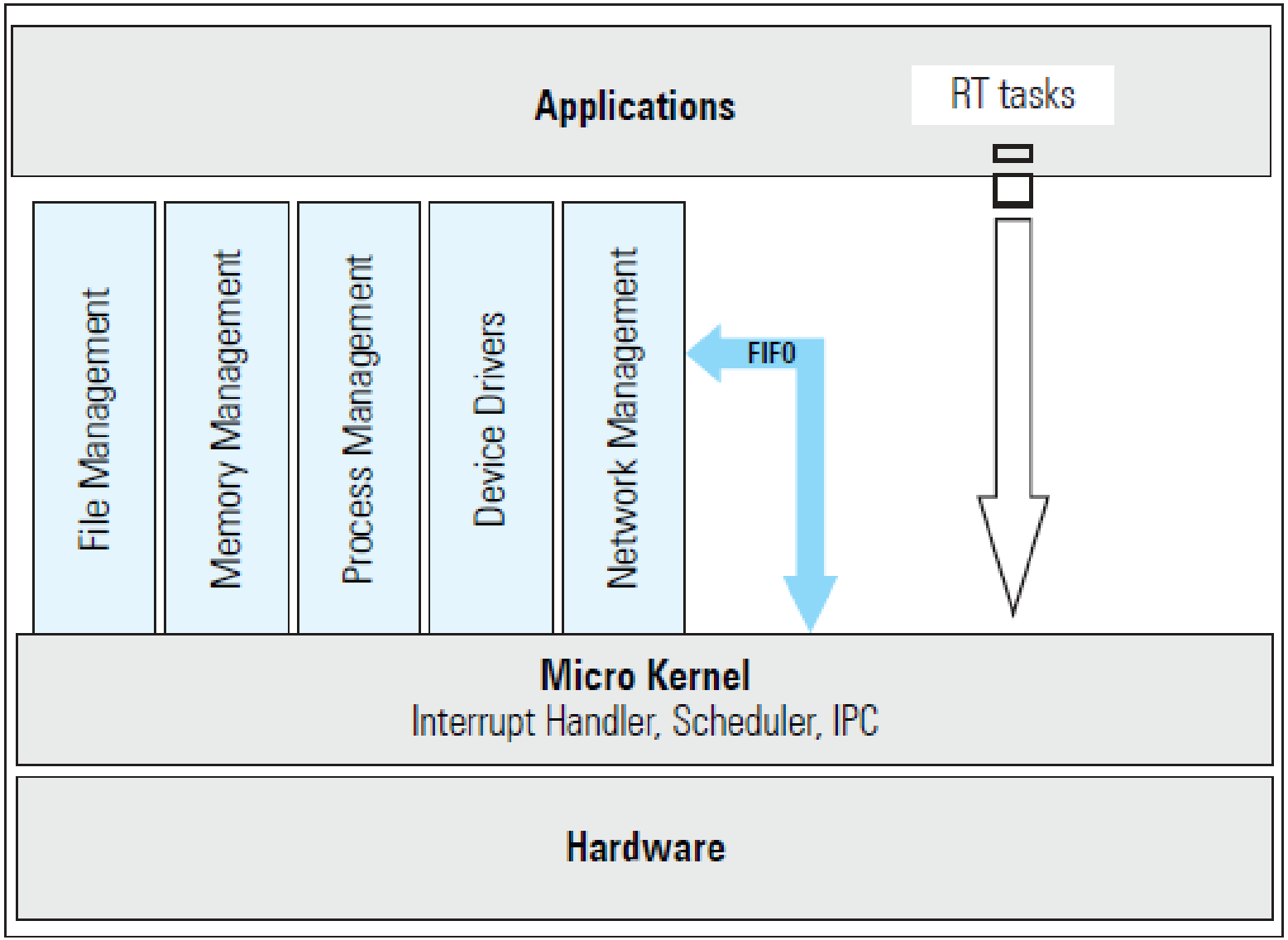
Few Popular RTOS



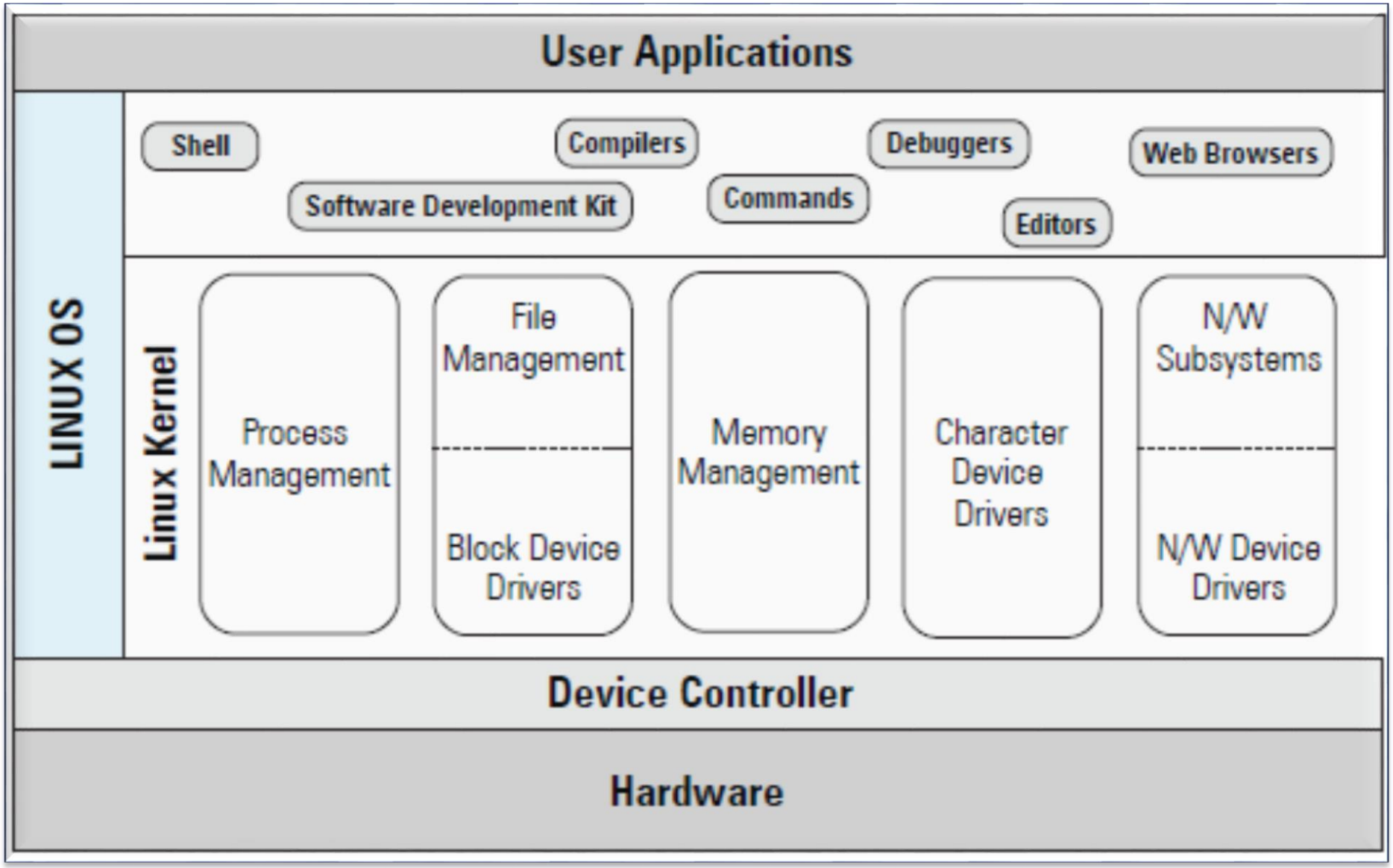
Monolithic Vs Micro Kernel



Hybrid Kernel Architecture



Introduction – Linux OS



List of Linux Distributions ...not completely

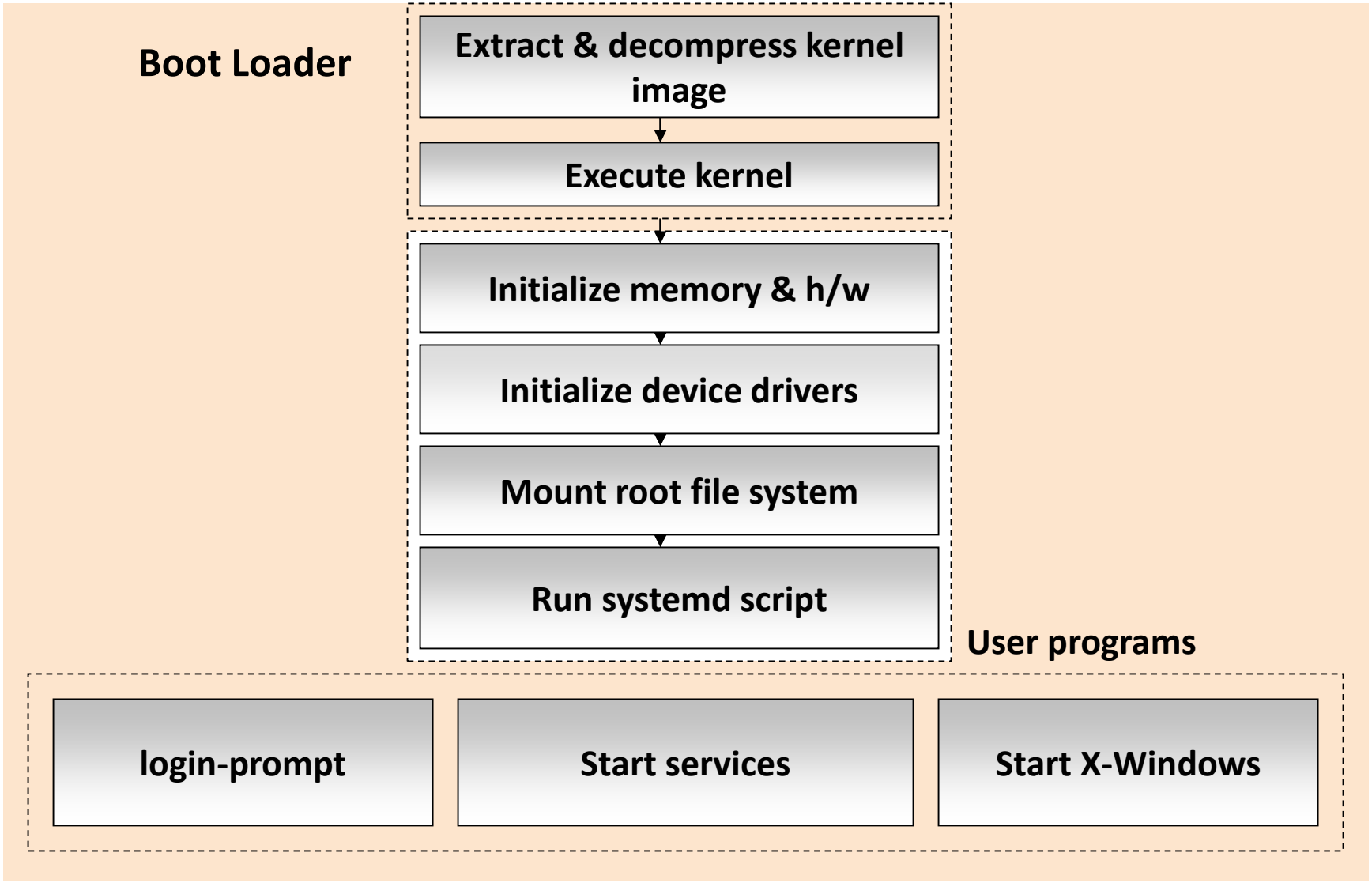
Kernel Source Code

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**

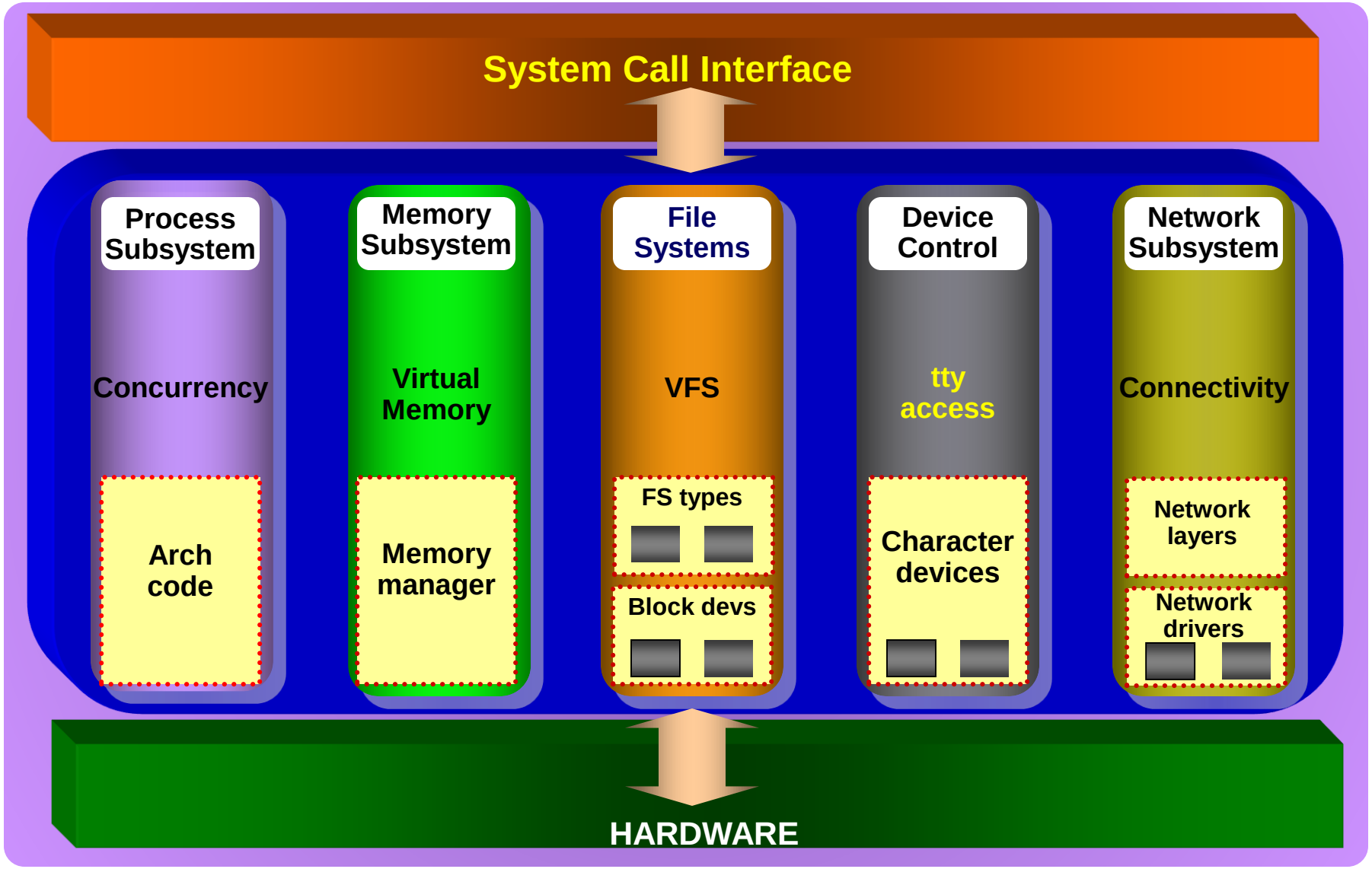
- **Machine-independent code deals with**
  - **System call handling**
  - **The file system: files, directories, pathname translation, file locking, and I/O buffer management**
  - **Terminal handling support: the terminal-interface driver and terminal line disciplines**
  - **IPC facilities**
  - **Network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

Booting Procedure

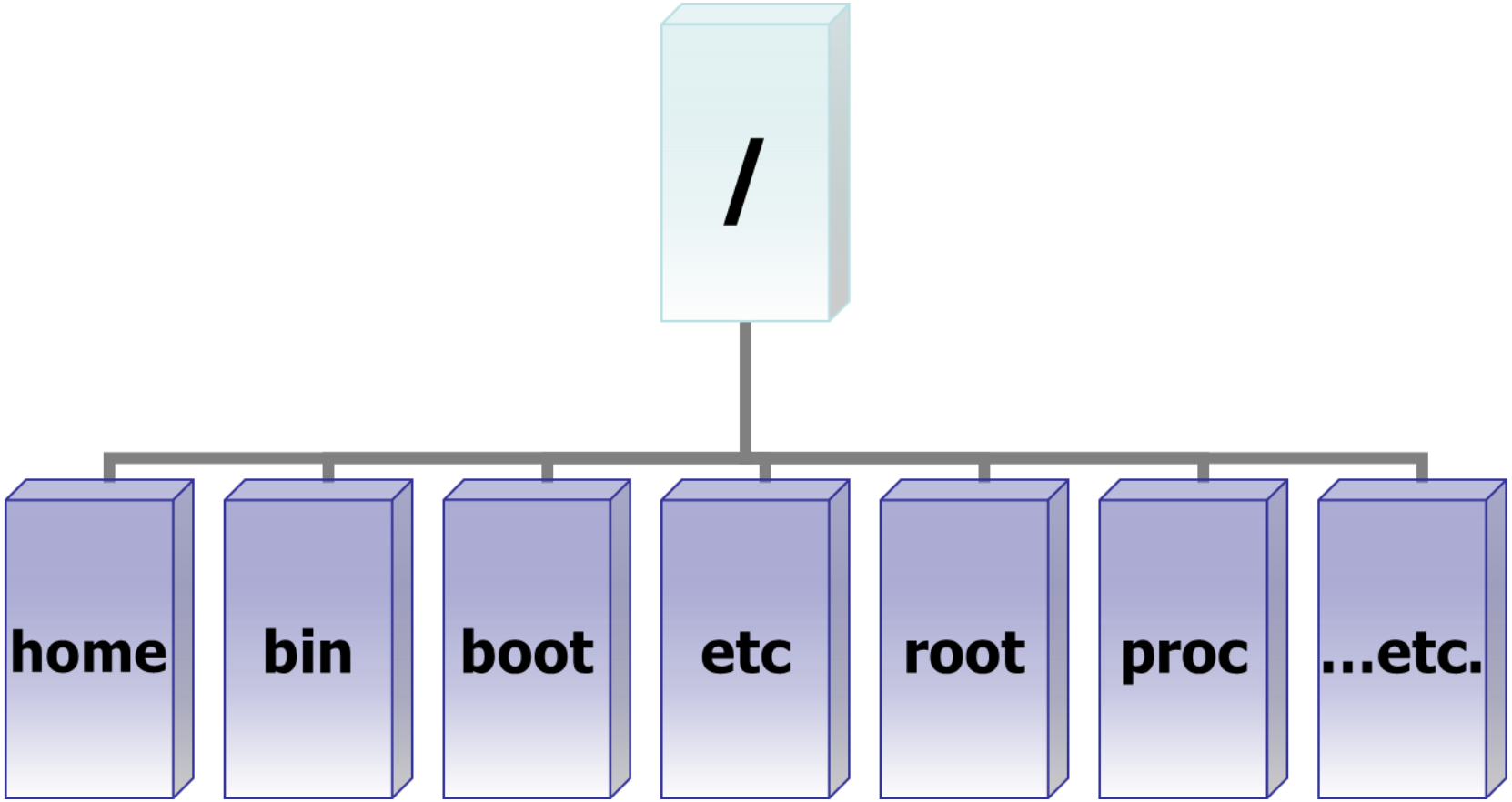


- Power on
- Boot Loader / MBR
- Loaded into Physical Memory and uncompressed
- Execute Kernel
- Mount File Systems (/etc/fstab)
- Choose Run Level (/etc/inittab)
- Spawn systemd process
- Shell / login prompt

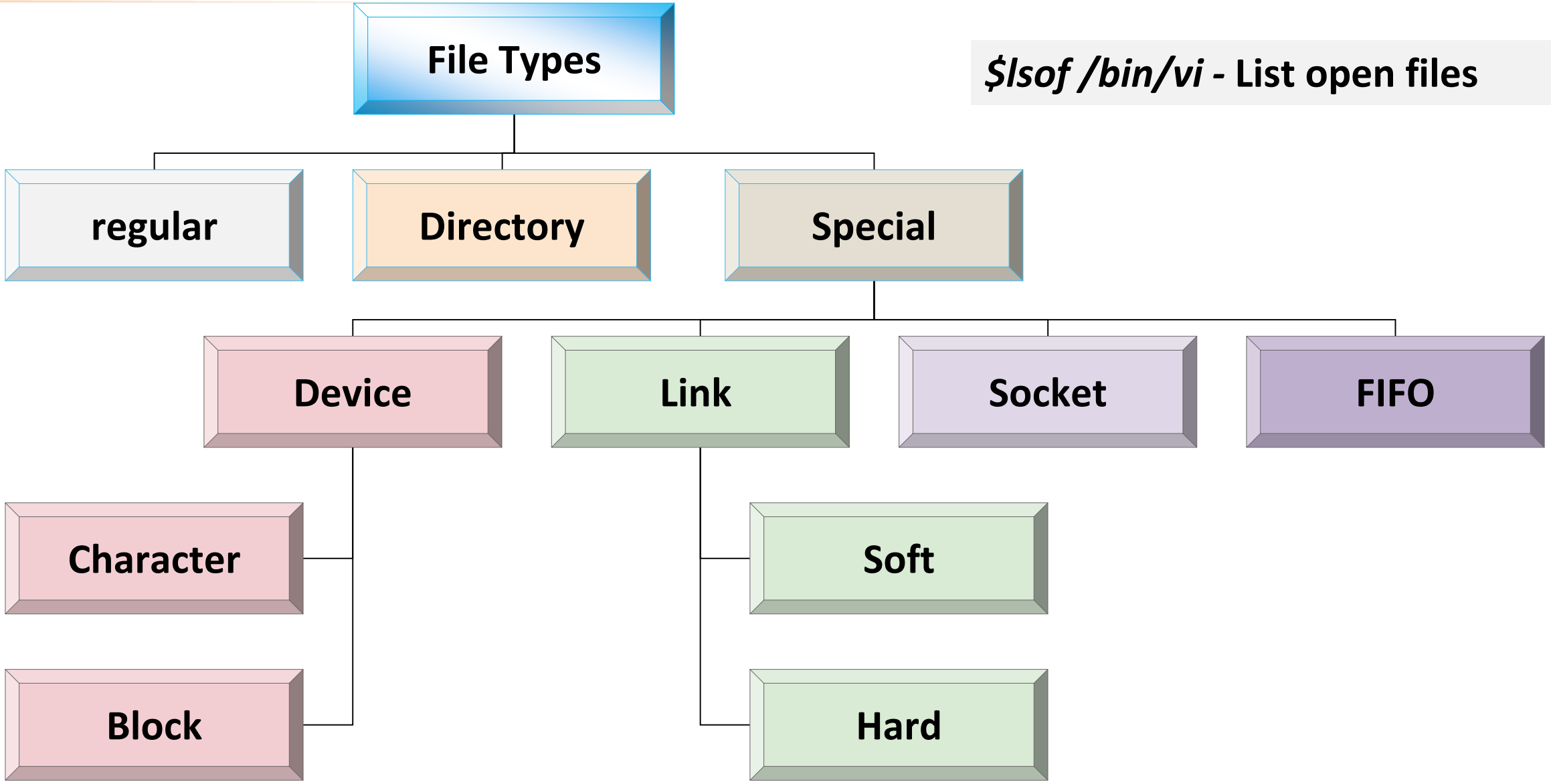
Linux Kernel Architecture



Inverted Tree Structure



File Types



$lsof /bin/vi - List open files

$ ls -l

- Identification of File types
  - ? r w - r- - r- -
  - ?-specifies a type of a file
- Regular (-)
- Directory (d)
- Special Files

Special Files

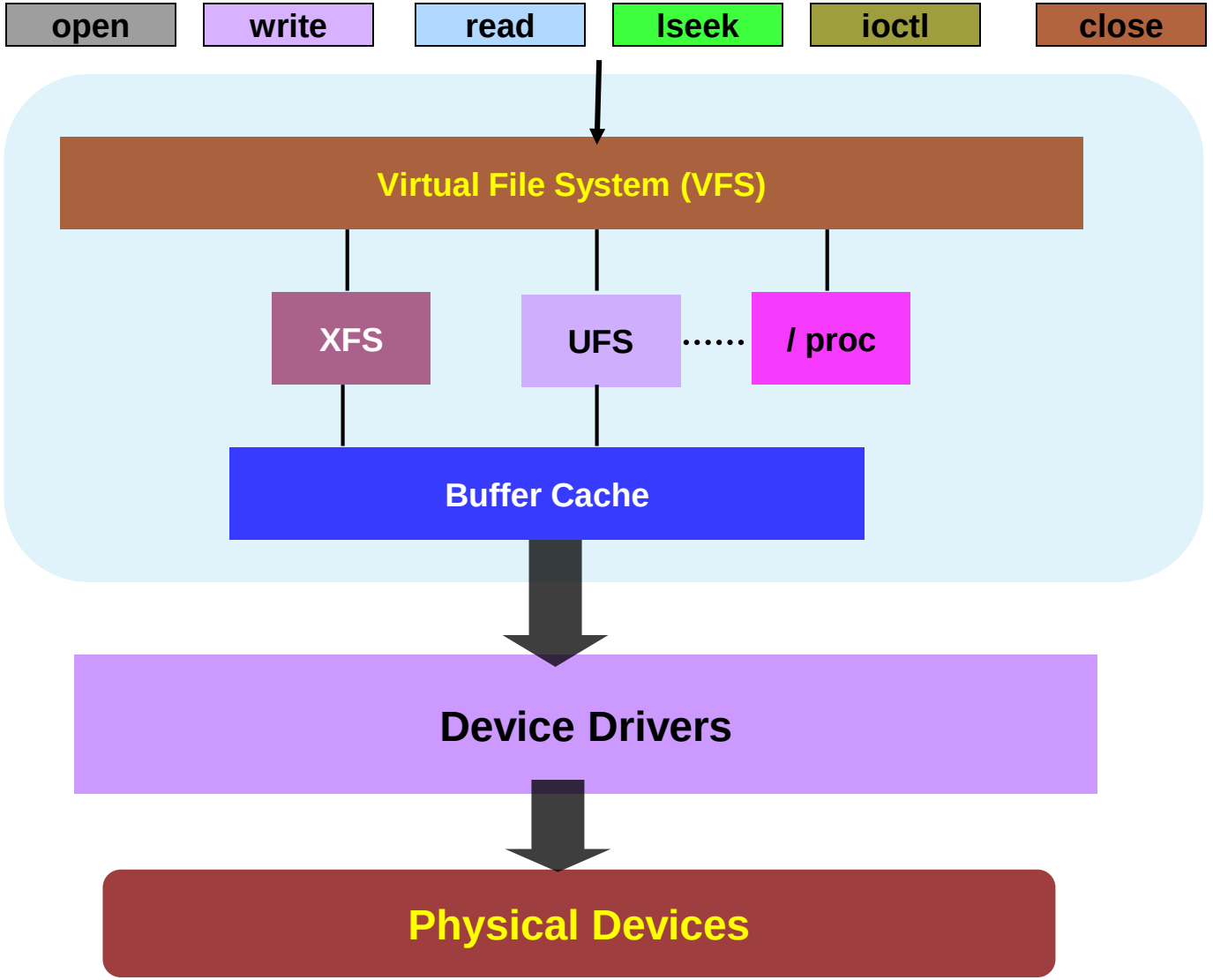
- FIFO (p)
- Socket (S)
- Link File
  - Soft Link (l)
  - Hard Link (inode numbers are same)
- Device File
  - Character (c)
    - Example: Monitor, Keyboard, Mouse, Tape
  - Block (b)
    - Example: Hard disk, CDROM, Floppy

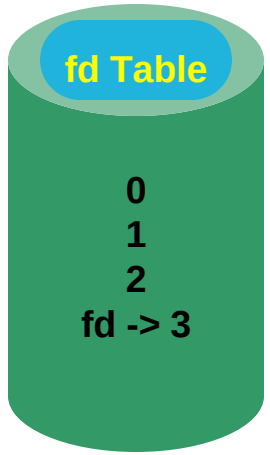
File System

- 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.

- VFS
- XFS
- ext4
- UFS
- proc
- msdos
- iso9660
- Vfat
- Aufs
- …….

File System Architecture





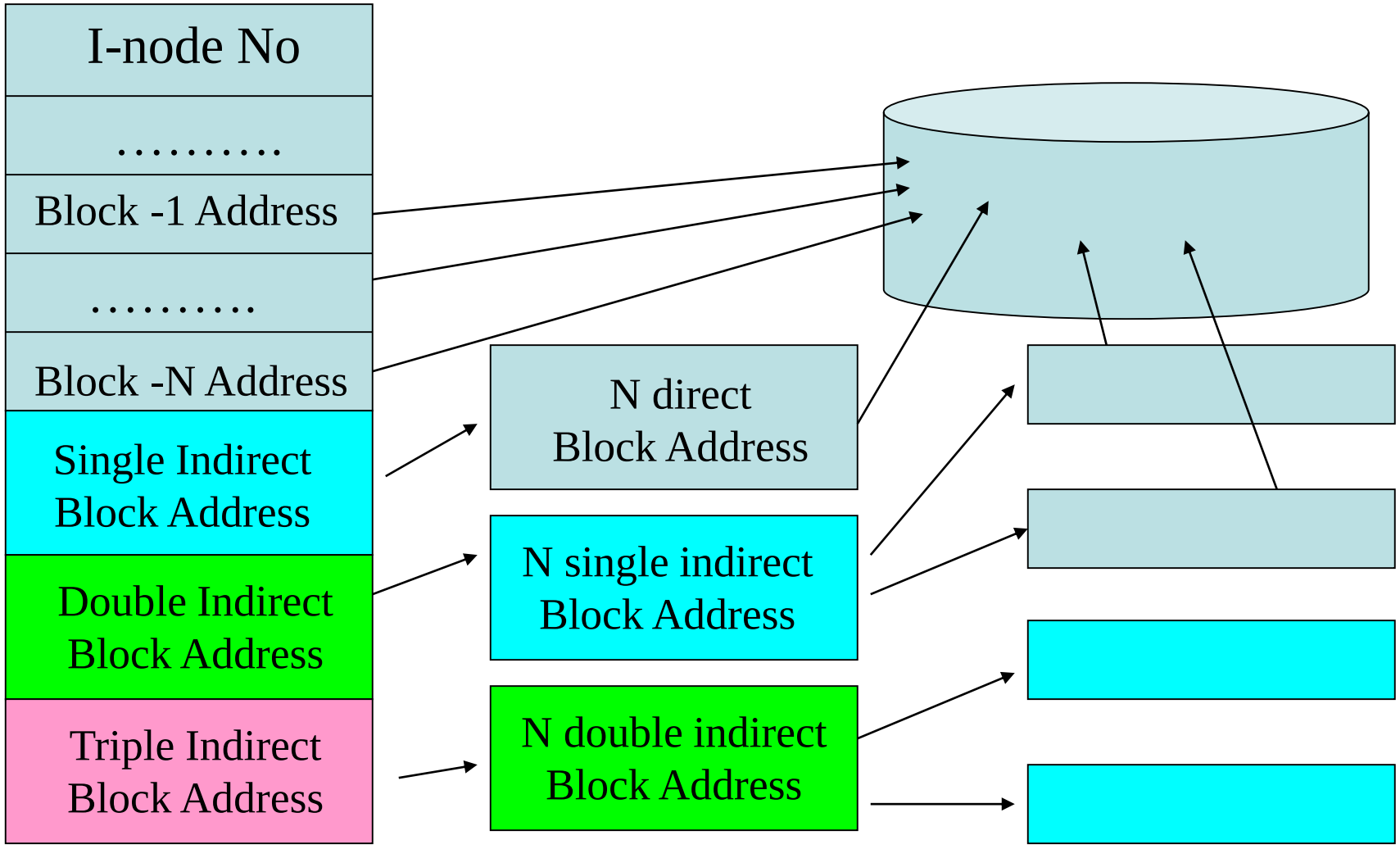
Proc File System

- 3 imp features
- Process
- System info
- System Limitations

UNIX Files

- Regular
- Directory
- Link (Soft and Hard)
- FIFO
- Socket
- Character Device
- Block Device

ext File System



- Each I-node entry can track a very large file.
- Assuming a block size of 4KB and 12 direct data blocks, a single I-node entry can track file that has a maximum size of:
  - 12 *4 KB +
  - (1024 * 4 KB) +
  - (1024*1024 * 4 KB) +
  - (1024*1024 * 1024 * 4 KB)
- File size can grown upto 16 terra bytes.

Kilo Vs Kibi

KB Vs KiB			
Decimal		**Binary**	
Value	**Metric**	**Value**	**IEC***
10^3 (1000)	Kilo	2^{10} (1024)	Kibi
10^6	Mega	2^{20}	Mebi
10^9	Giga	2^{30}	Gibi
10^{12}	Tera	2^{40}	Tebi
10^{15}	Peta	2^{50}	Pebi
10^{18}	Exa	2^{60}	Exbi
10^{21}	Zetta	2^{70}	Zebi
10^{24}	Yotta	2^{80}	Yobi

*The **International Electrotechnical Commission** is a non-profit, non-governmental **international standards organization** that prepares and publishes International Standards for all electrical, electronic and related technologies – collectively known as "electrotechnology"

FS Comparison

Feature	EXT4	XFS	BTRFS
Architecture	Hashed B-tree	B+ tree	Extent based
Introduced	2006	1994	2009
Max volume size	1 Ebytes	8 Ebytes	16 Ebytes
Max file size	16 Tbytes	8 Ebytes	16 Ebytes
Max number of files	4 billion	2^{64}	2^{64}
Max file name size	255 bytes	255 bytes	255 bytes
Attributes	Yes	Yes	Yes
Transparent compression	No	No	Yes
Transparent encryption	Yes	No	Planned
Copy-on-Write (COW)	No	Planned	Yes
Snapshots	No	Planned	Yes

Linux supported File systems ($ man fs)

- **minix** – is the file system used in the Minix OS, the first to run under Linux
- **ext** – is an elaborate extension of the minix file system.
- **ext2** – is the high performance disk file system used by Linux for fixed disks as well as removable media.
- **ext3** – is a jounaling version of the ext2 file system.
- **Reiserfs** – is a journaling file system, designed by Hans Reiser.
- **XFS** – is a journaling file system, developed by SGI (Silicon Graphics Inc.)
- **JFS** – is a journaling file system, developed by IBM
- **msdos** – used by DOS and Windows.
- **umsdos** – extended DOS file system.
- **vfat** – extended DOS file system used by Microsoft Windows95 and Windows NT.
- **proc** – pseudo file system, which is used as an interface to kernel data structures. Its files do not take space.
- **iso9660** – CD-ROM file system type conforming to the ISO 9660 standard.
- **hpfs** – is the high performance File system. This file system is read only under Linux due to the lack of available documentation.
- **sysv** - is an implementation of the SystemV file system.
- **nfs** – is the network file system to access disks located on remote computers.
- **smb-** is a network file system that supports the SMB protocol, used by Windows and Windows NT
- **ncpfs** – is a network file system that supports the NCP protocol, used by Novell NetWare.

Comparison of files systems with various features you can get it from : http://en.wikipedia.org/wiki/Comparison_of_file_systems

XFS File System

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

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

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

- **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**

XFS File System

- Each device also contains more than one copy of the super-block.
- Linux 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.

- 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.

XFS File System

- 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

- 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.

Device Special File

- 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”)

- 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

$*mount* Command

- 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.

- Each file system must be mounted before it can be used. Normally, all file systems are mounted during system startup depending on */etc/fstab* entry
- (*blkid, lsblk, lsblk* –help)
- root file system is mounted by default
- It is possible to mount a file system at any time using the "mount" command.

$*mount* Command

- 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.

- 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

- 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/sda6 on / type ext4 (rw)
none on /proc type proc (rw)
```

procfs

- To store necessary information about currently running processes, system information and limitations. Size of /proc is zero. Created on the fly.

```
/dev/sda1 on /boot type xfs
/dev/sda2 on / type xfs
proc on /proc type proc
[root@localhost proc]# ls
1       1210  1452  218   239   2460  27    42   536  63          cmdline     keys         self
10      1211  15    2209  2390  2461  270   43   538  662         consoles    key-users    slabinfo
10702   1231  16    2213  2393  2471  2702  459  540  7           cpuinfo     kmsg         softirqs
11      13    1669  2282  2396  2479  2703  460  541  743         crypto      kpagecount   stat
11281   1332  1689  2301  240   2484  271   461  542  8           devices     kpageflags   swaps
11306   1334  17    2305  2405  2487  28    462  543  8222        diskstats   loadavg      sys
11331   1340  18    2309  241   2491  29    463  545  8458        dma         locks        sysrq-trigger
11355   14    1812  2328  2416  2510  3     466  546  9           driver      mdstat       sysvipc
11421   1412  1854  2337  242   2514  30    467  552  9044        execdomains meminfo      timer_list
11426   1413  19    2346  2421  2527  31    468  553  9081        fb          misc         timer_stats
11432   1414  1957  2350  2426  26    350   469  556  9086        filesystems modules      tty
1193    1419  2     2353  2430  265   370   471  560  93          fs          mounts       uptime
1196    1420  20    2368  2434  266   372   502  562  9338        interrupts  mtrr         version
1199    1426  2076  2372  2437  2663  373   512  567  acpi        iomem       net          vmallocinfo
12      1427  21    2376  2442  267   382   515  570  asound      ioports     pagetypeinfo vmstat
1200    1428  2136  238   2446  268   3835  526  575  buddyinfo   irq         partitions   zoneinfo
1203    1429  2143  2381  245   269   39    529  593  bus         kallsyms    sched_debug
1209    1451  2144  2387  2455  2699  41    535  6    cgroups     kcore       scsi
```

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

- 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.

- 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

- overview
- fd table
- System Calls
- Opening a file
- Duplicating a file descriptor
- Random Access
- File control
- Get file status
- *select* system call

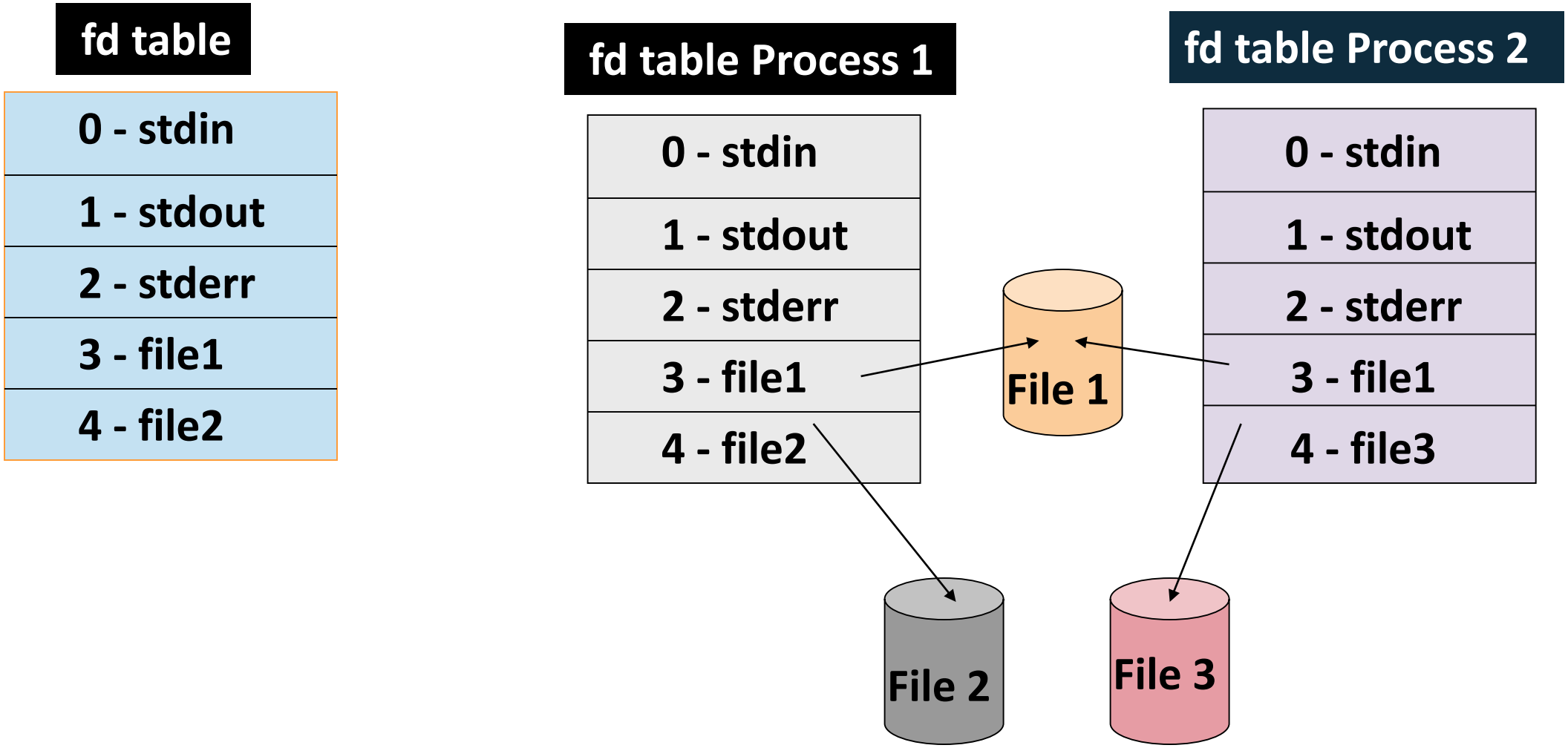
Overview –I/O

- 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.

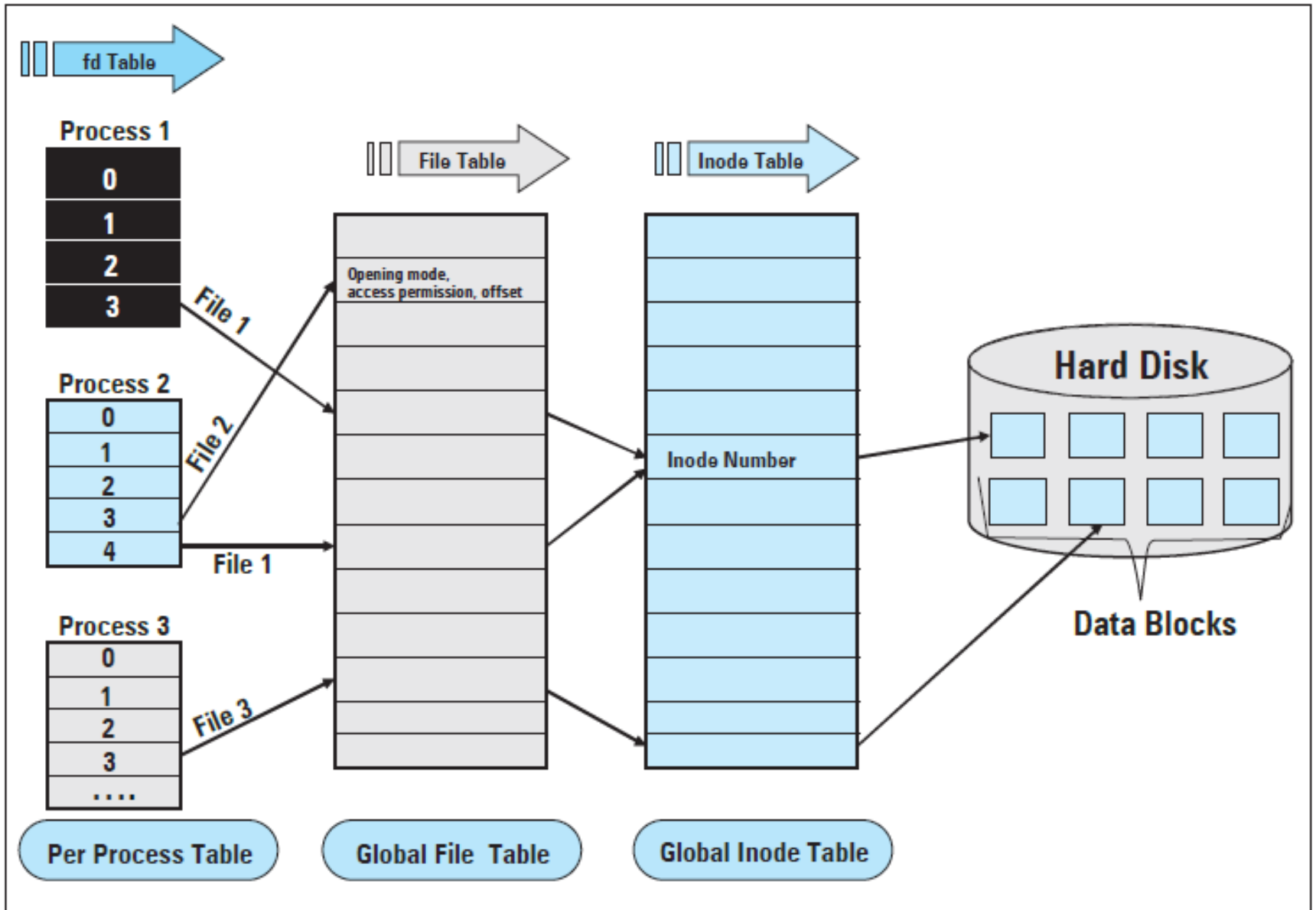
- The system calls related to the Input-Output operations take a descriptor as an argument.
- Each process has a separate File Descriptor (FD) Table.
- Valid file descriptor ranges from 0 to a maximum descriptor number that is configurable. (ulimit, /proc/sys/fs/file-max)
- Kernel assigns descriptors 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.

- 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.

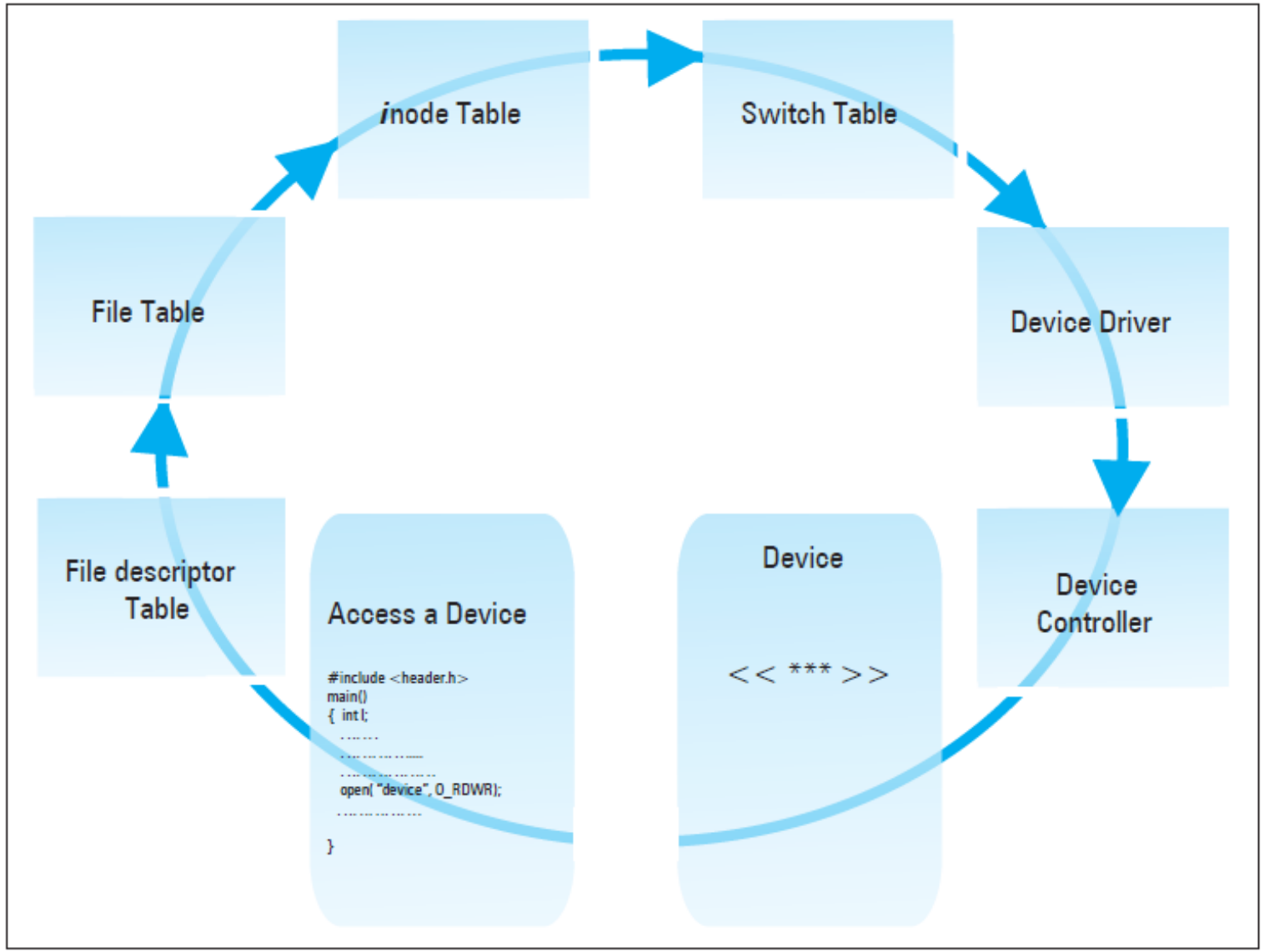
fd Table


fd table
0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file2
fd table Process 1
0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file2
fd table Process 2
0 - stdin
1 - stdout
2 - stderr
3 - file1
4 - file3
File 1
File 2
File 3

Interaction between fd and Data Block



User Process Accessing Device Special File



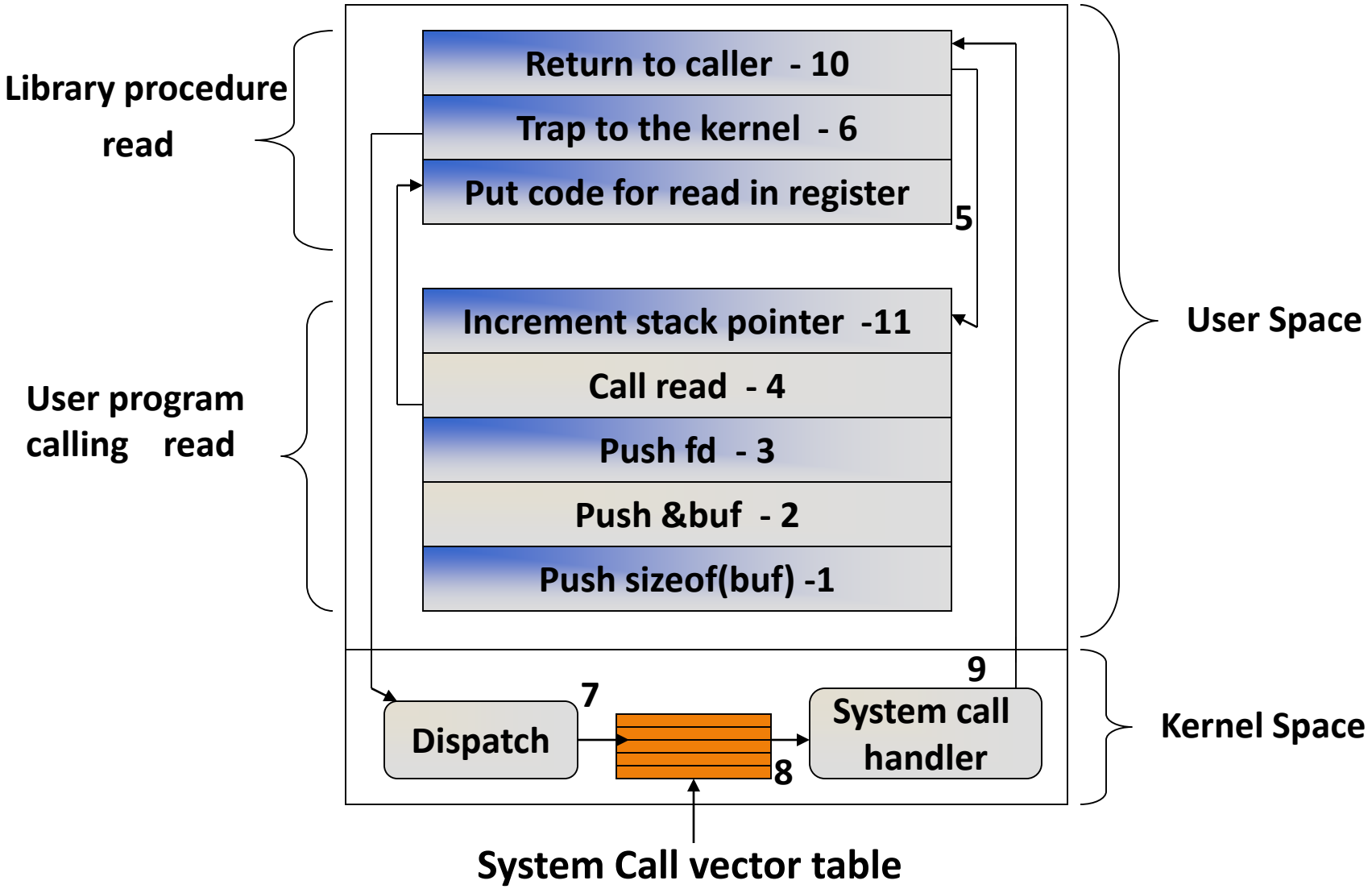
System Calls

- 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)

- 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

- 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, &buff, sizeof(buf))



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

System Calls

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);

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.

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 or -1 on error.

System Calls

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 or -1 on error.

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 or -1 on error.

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

whence: SEEK_SET, SEEK_CUR or SEEK_END – from the end of file

- On success the system call returns with any one of the following value:
  - Offset value or 0 or -1

System Calls

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

int fstat (fd, struct stat *);

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

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 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);

I/O subroutine

write modifications to the disk file - void sync (void);

Close a fd - int close (fd);

int unlink("file_name") - equivalent to $rm file_name;

Linux uses internal routines for accessing a file. For example

- 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)

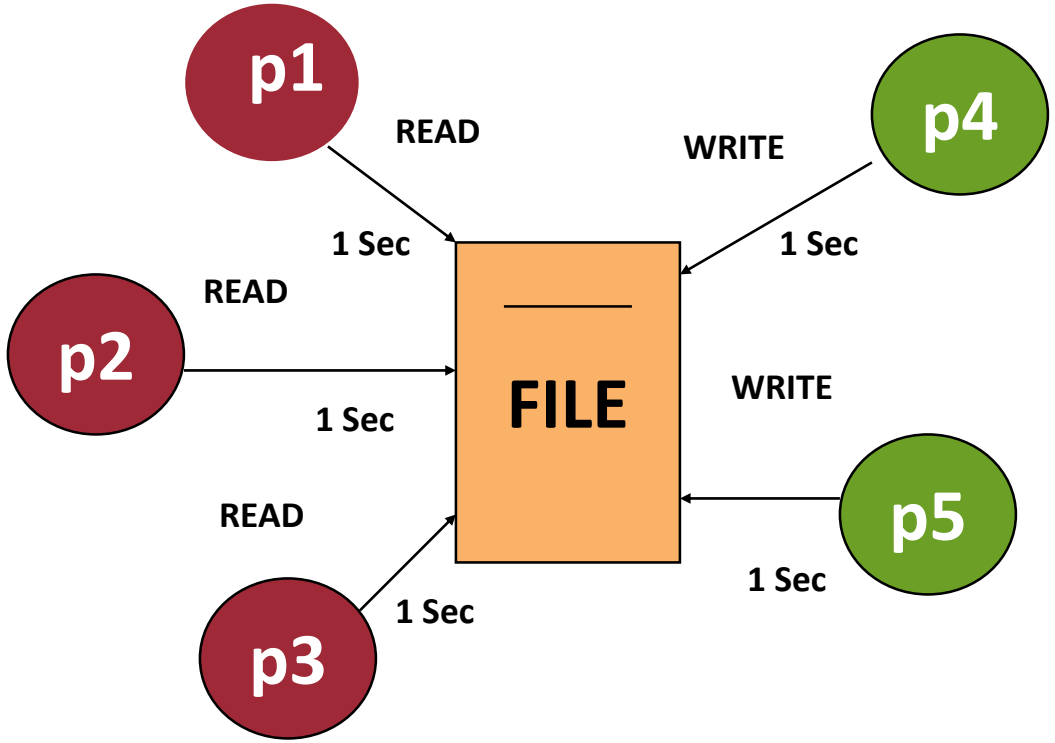
File Locking

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

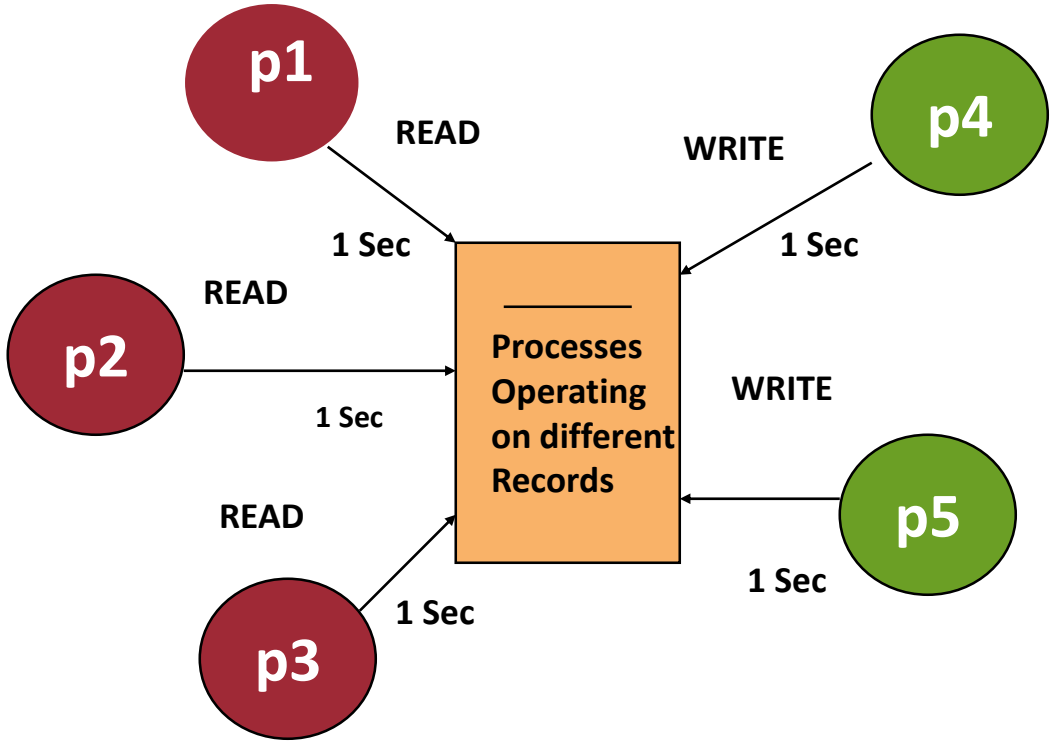
Types of Locking

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

Mandatory Vs Record Locking



Total Time = 1 + 1 +1= 3 sec



Total Time = 1 sec

flock Structure

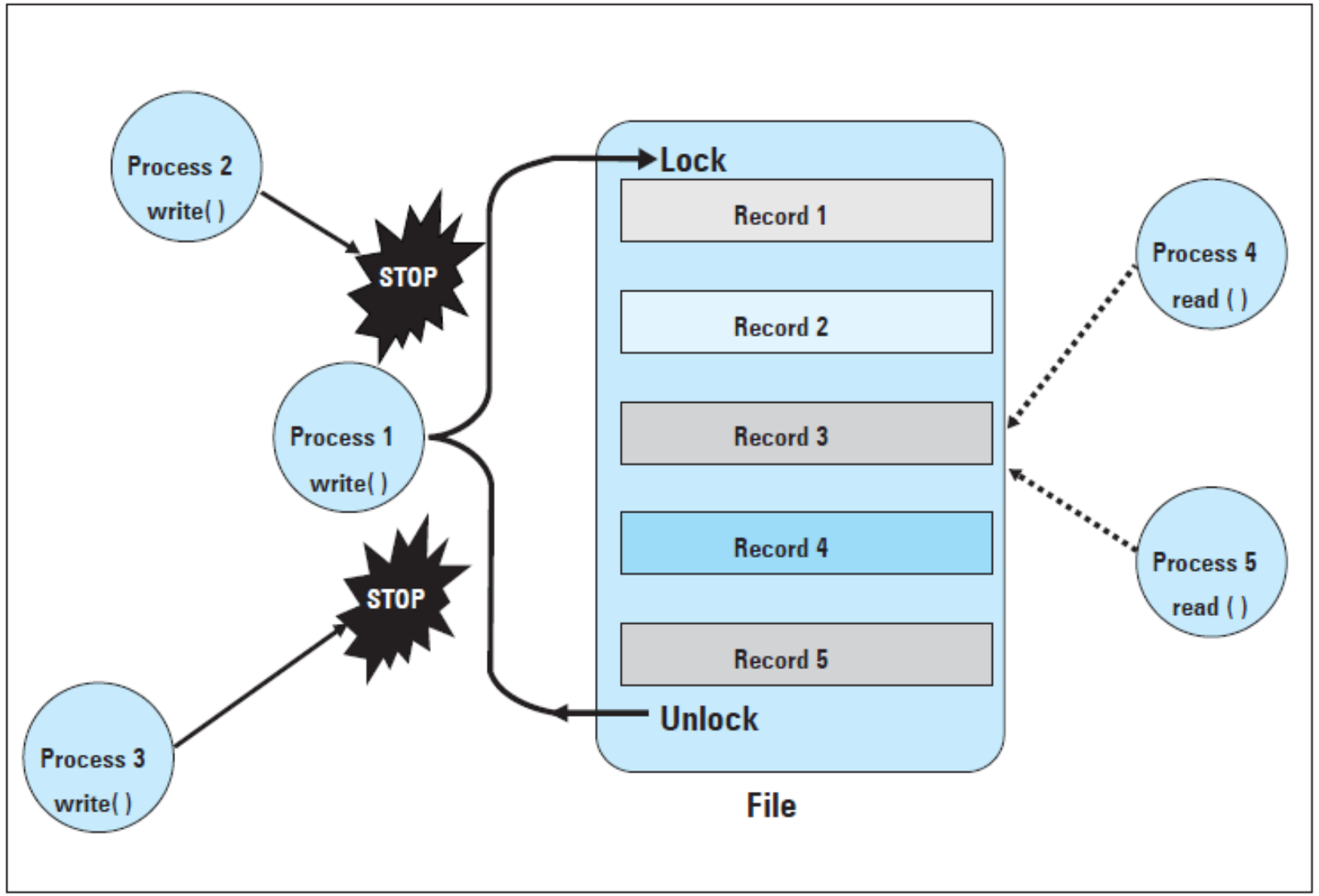
```
struct flock {
   short        l_type;
   /* lock type: read, write or unlock */
   short        l_whence;
   off_t        l_start;
   off_t        l_len;
   pid_t        l_pid;
};
```

Read lock: allows many readers but not a single writer

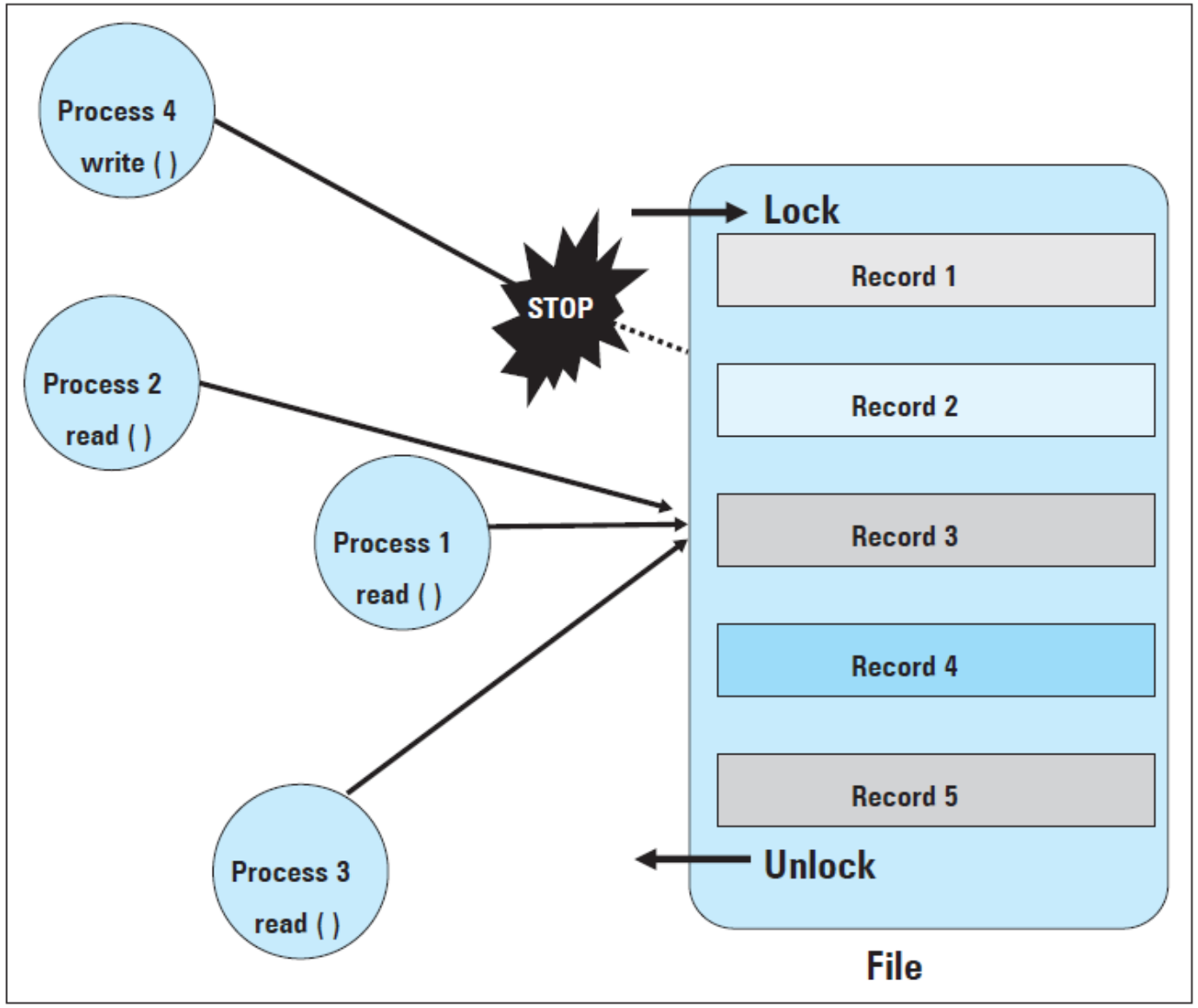
Write lock: allows only one writer but not a single reader

- 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

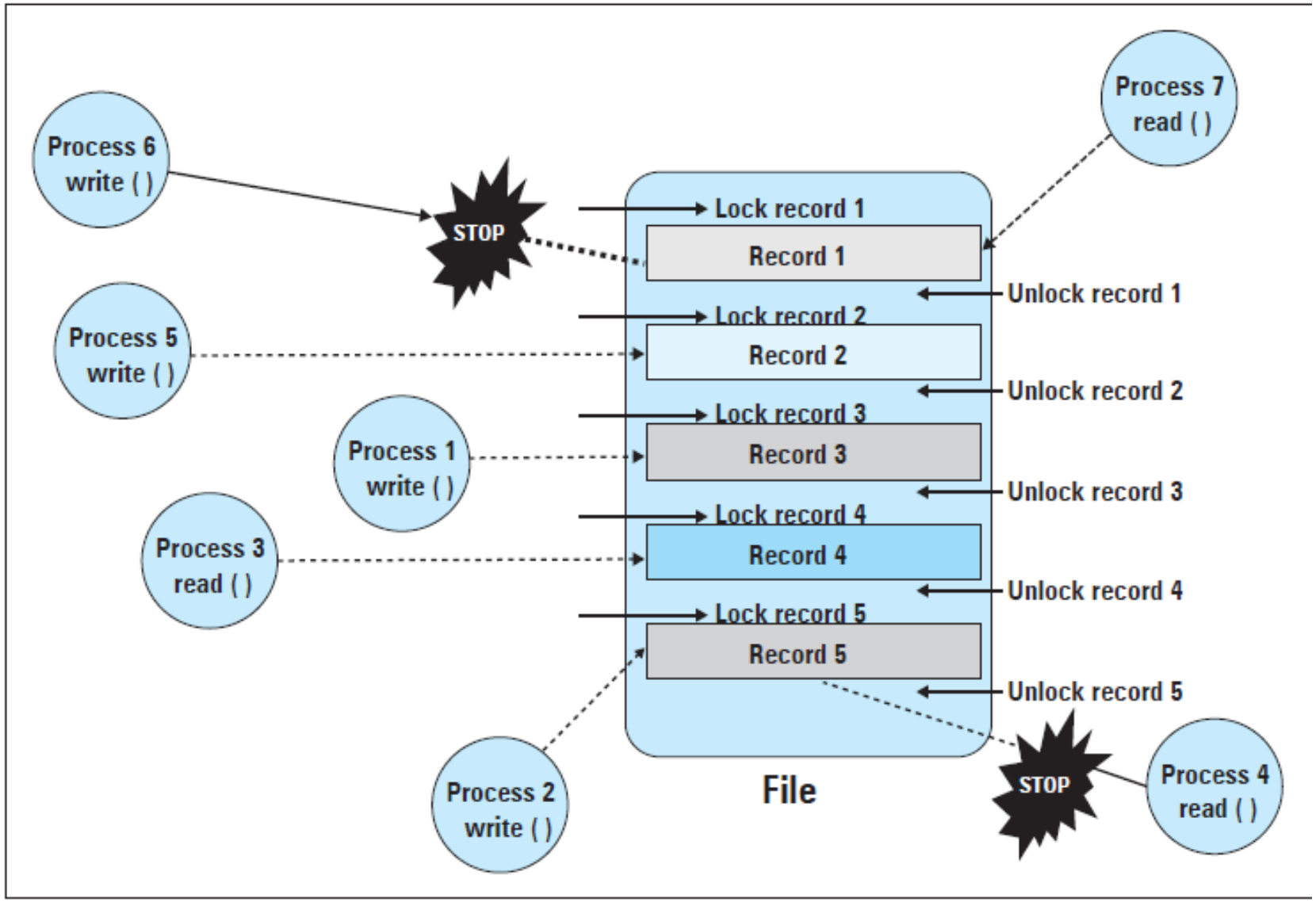
Non-Shared Lock –Mandatory File Locking



Shared Lock – Mandatory File Locking



Shared Lock – Advisory or Record Locking



Sketch of File Locking Implementation

fcntl Implementation

```
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 );
```

flock Implementation

```
flock ( fd, LOCK_EX );

……critical section……

flock ( fd, LOCK_UN );
```

```
flock ( fd, LOCK_SH );

……critical section……

flock ( fd, LOCK_UN );
```

Process Management

- mode and space
- Context switch
- Per process objects
- Execution Context
- Process structure
- Process states
- Process scheduling
- Process Creation - fork
- execl family of Library functions

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.
- Linux is a multiprocessing system
- The Linux kernel is reentrant

- 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.

Context Switch

- 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.

- 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.

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

- 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.

Task_struct Structure - /usr/src/linux-4.12/include/linux/sched.h

```
struct task_struct {
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;

        int on_rq;

        int prio, static_prio, normal_prio;
        unsigned int rt_priority;
        const struct sched_class *sched_class;
        struct sched_entity se;
        struct sched_rt_entity rt;
```

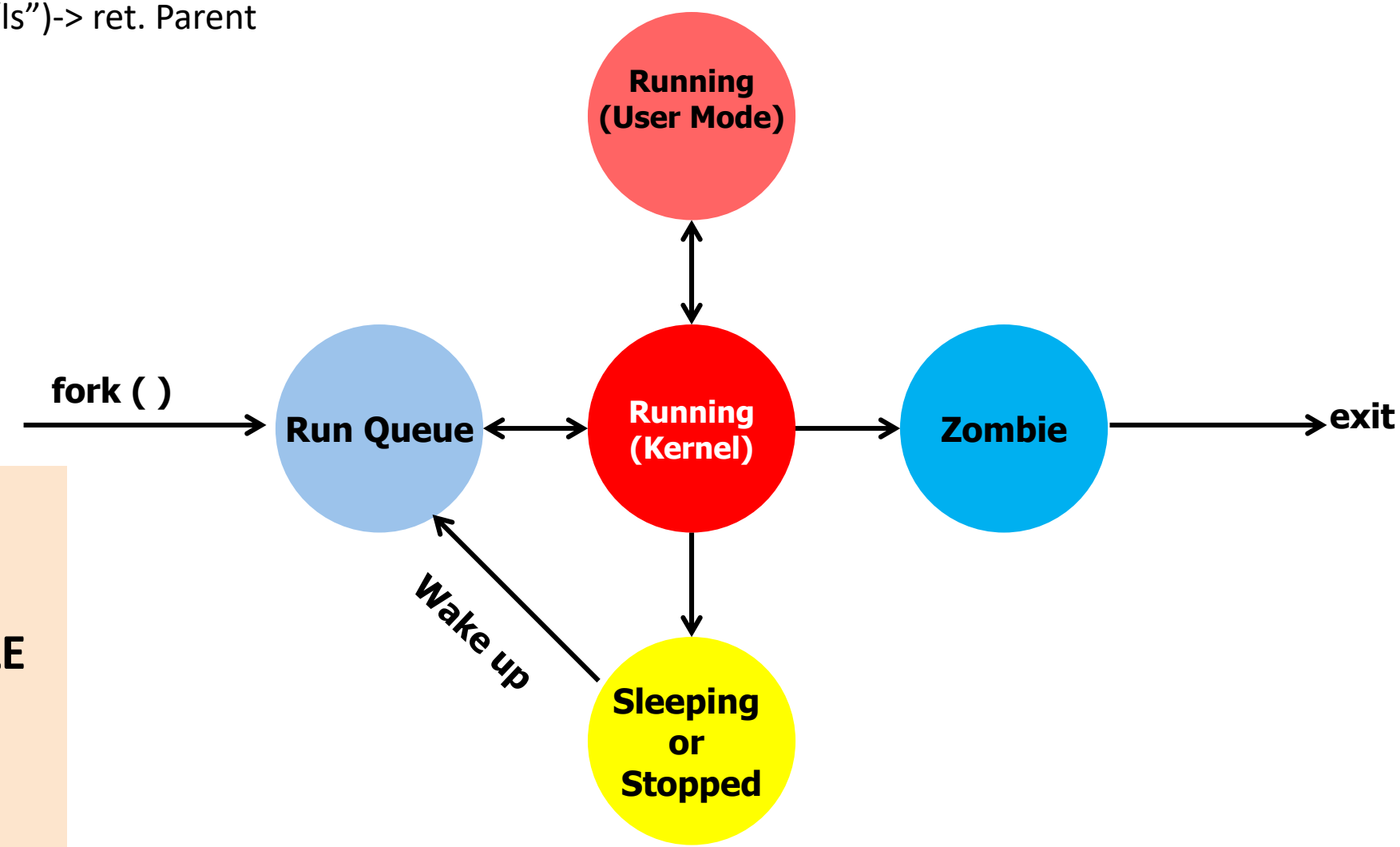
```
unsigned int policy;
int nr_cpus_allowed;
cpumask_t cpus_allowed;

struct sched_info sched_info;
struct list_head tasks;

struct mm_struct *mm, *active_mm;
----------------
----------------
----------------
```

Process States

$ls -> fork() -> child process-> exec("ls")-> ret. Parent



- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE

Scheduling

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**

- **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**

Process 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.

- 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.

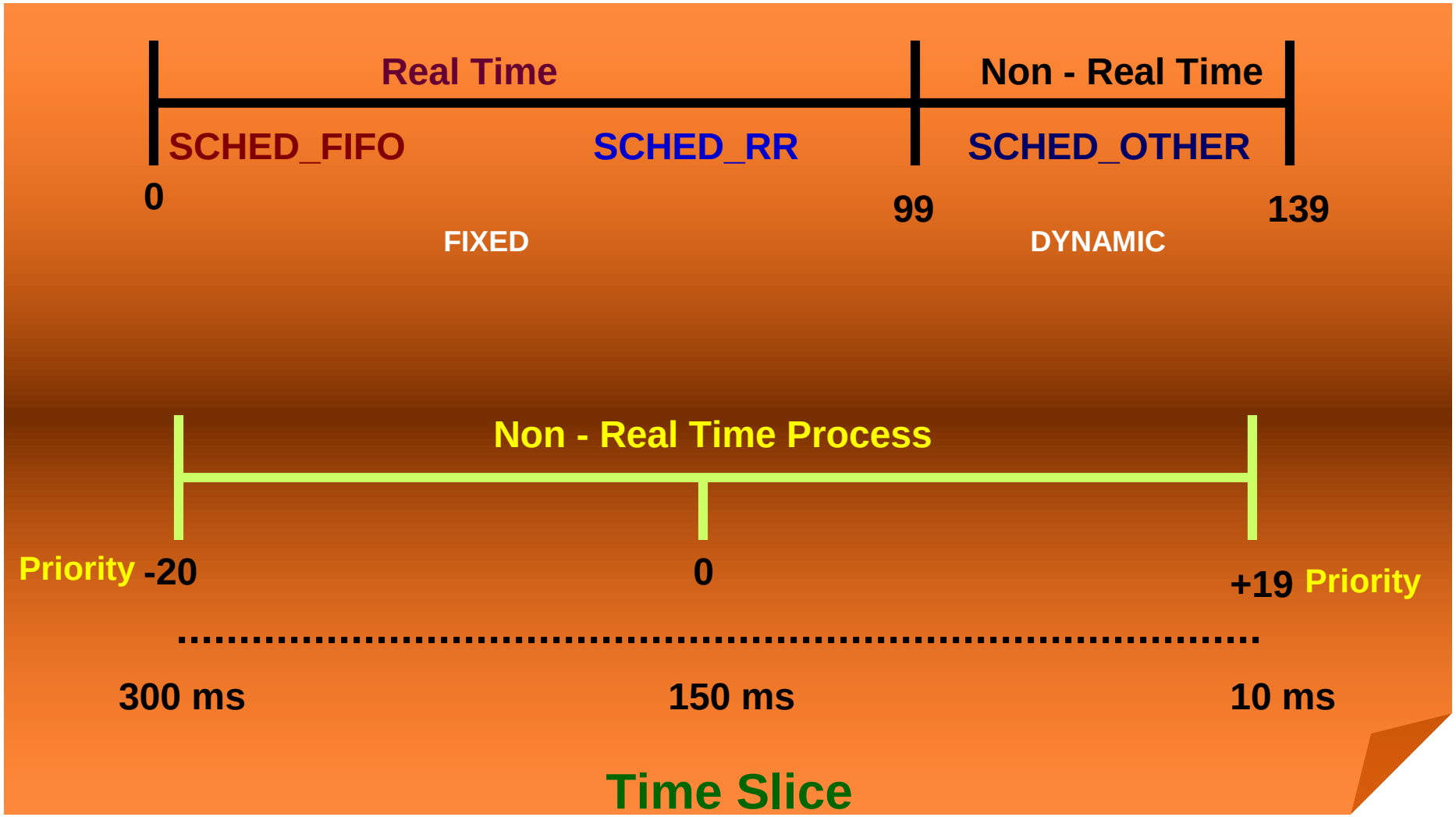
- The Linux kernel implements two separate priority ranges.
- The first is the nice value, a number from -20 to 19 with a default of zero. Larger nice values correspond to a lower priority.
- A process with a nice value of -20 receives the maximum time slice, whereas a process with a nice value of 19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

Process Scheduling

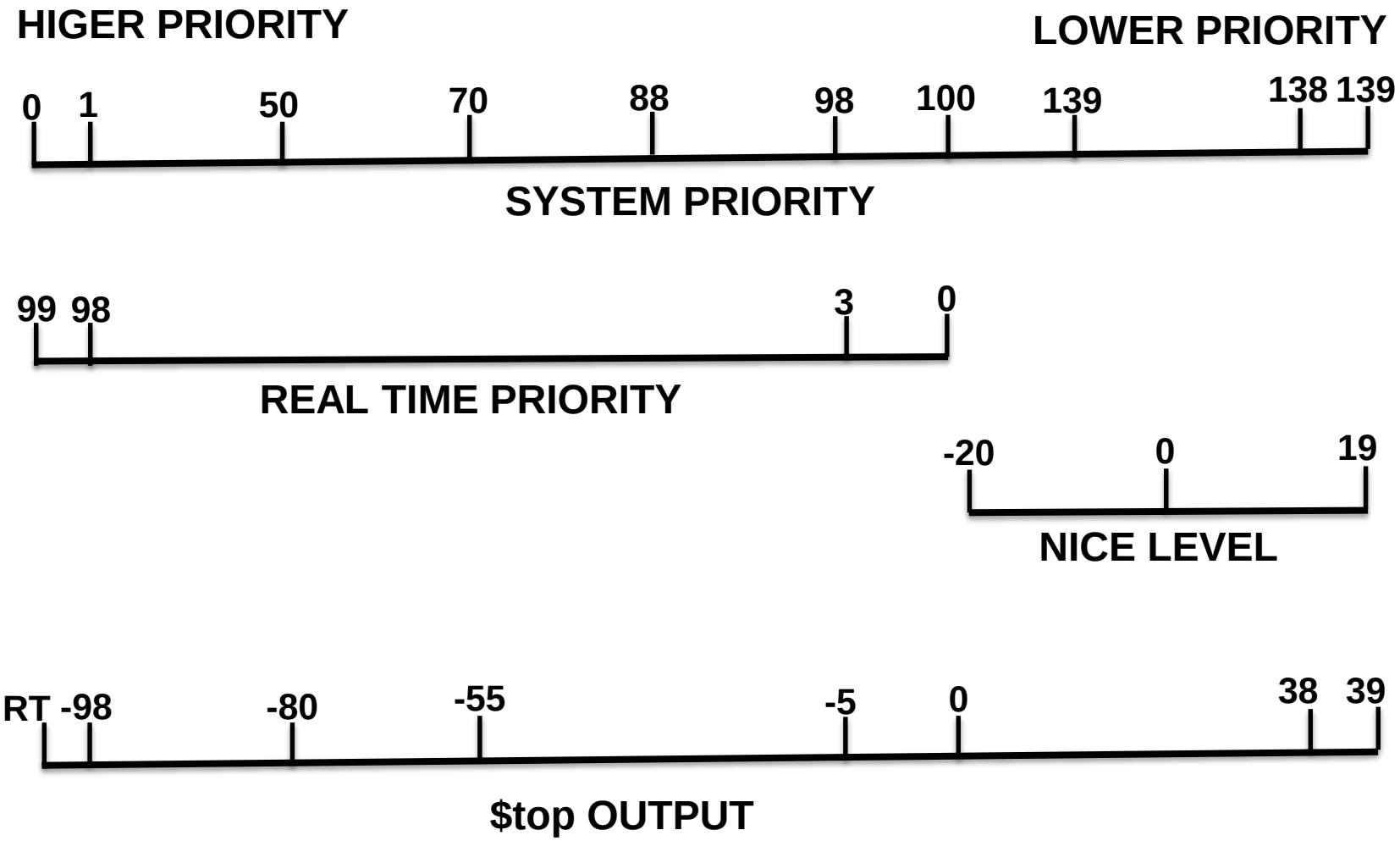
- The second range is the real-time priority
- By default, it ranges from zero to 99.
- All real time processes are at a higher priority than normal processes.
- Linux implements real-time priorities in accordance with POSIX.

- Linux provides two real-time scheduling policies, SCHED_FIFO 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.

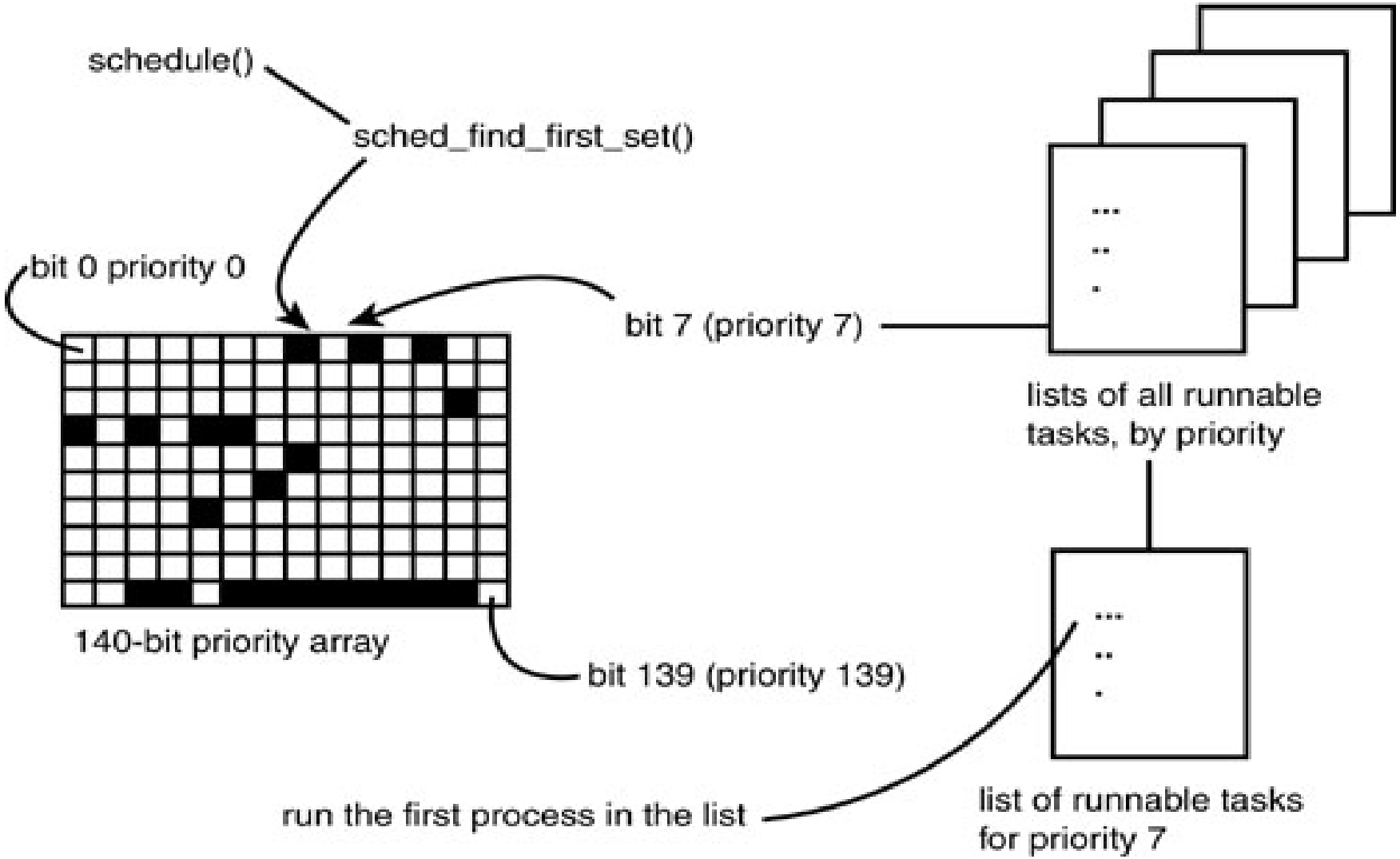
Linux Scheduling Priority



Linux Scheduling Priority



O(1) Scheduler



Ref: Linux Kernel Development by Robert Love

nice and *chrt* –Manipulate Scheduling Priorities

Command	Priority	nice
➢ $nice -19 ./a.out	39	19
➢ $nice -10 ./a.out	30	10

$nice -20 ./a.out

$renice –n 5 –p <pid>

xyz (process ID) old priority 19, new priority 5

$renice –n -20 –p <pid>

Old priority 5, new priority -20

In top command: PR – 0; NI = -20

Command	Priority	nice
○ $chrt -r –p -1 <pid>	< NOT VALID COMMEND>	
○ $chrt -r –p 0 <pid>	20	0
○ $chrt -r –p 1 <pid>	-2	0
○ $chrt -r –p 5 <pid>	-6	0
○ $chrt -r –p 50 <pid>	-51	0
○ $chrt -r –p 90 <pid>	-91	0
○ $chrt -r –p 98 <pid>	-99	0
○ $chrt -r –p 99 <pid>	RT	0
○ $chrt -r –p 100 <pid>	< NOT VALID COMMEND>	

Process Scheduling Commands

- $nice -19 ./***.sh; top ;
- r -> pid to renice; renice value: 10
- $man chrt (1)
- $chrt –m
  - Show minimum and maximum valid priorities.
- $chrt -p <pid>
  - To display scheduling attributes of an existing task
- $chrt -r -p 99 <pid>
  - Set scheduling policy to SCHED_RR

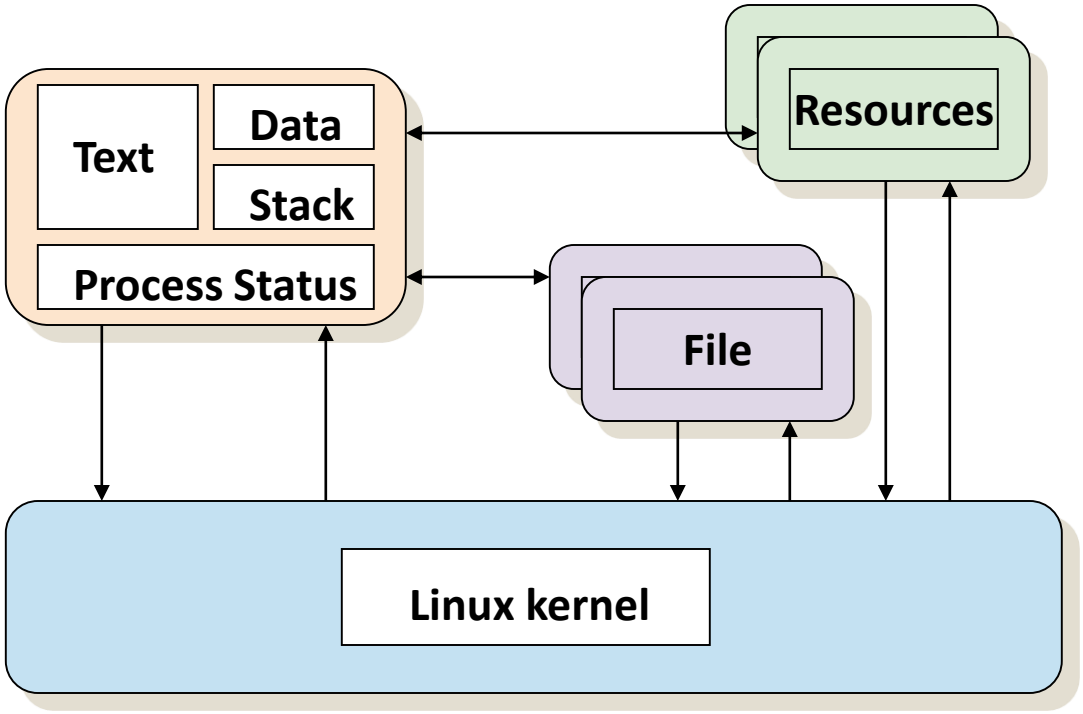
- $ top -p <pid>
- $chrt -f -p 99 <pid>
  - Set scheduling policy to SCHED_FIFO
- $chrt -o -p 0 <pid>
  - Set scheduling policy to SCHED_OTHER
- man 2 sched_setscheduler

System Calls for Scheduling

- 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

Process Creation

- 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.



Process Tree Structure

- In a Linux 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.

- 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.

Parent and Child

- 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

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

COW, Zombie and Orphan Process

- Instead of copying the address space of the parent, Linux uses the COW technique for economical use of the memory page.
- The parent space is not copied, it can be shared by both the parent and the child process but the memory pages are marked as 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.

- 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.

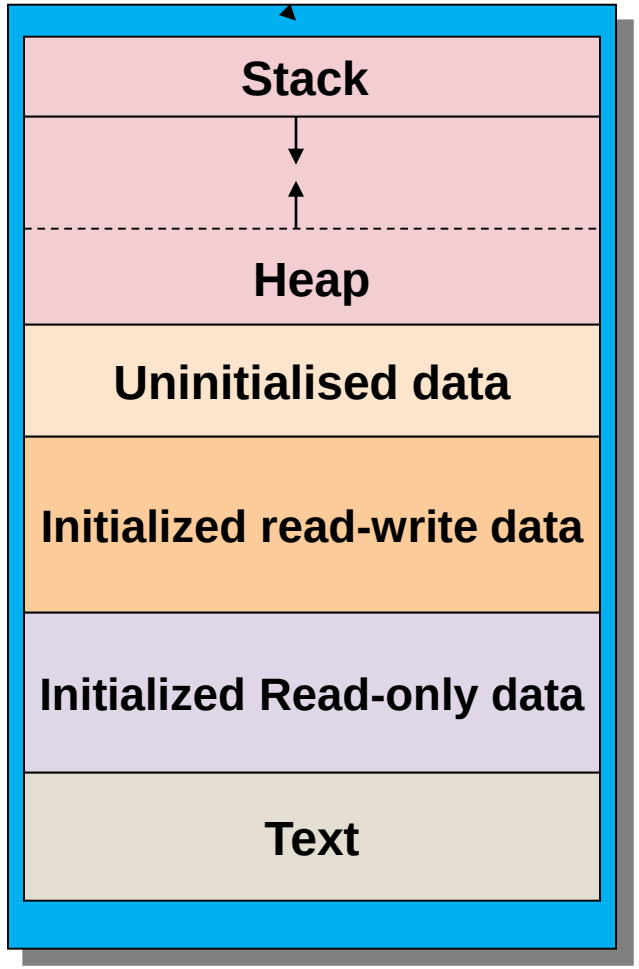
exec Family of Functions

- 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

- 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



- 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.

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

```
$ size a.out  (man size )
   text    data     bss     dec     hex filename
    920     268      24    1212     4bc a.out
```

Stack Portion

- 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.

- 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.

Daemon Process

- 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

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

Daemon Process Creation

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

Process Types

- Parent
- Child
- Orphan
- Daemon

Process States

- Running (R)
- Stopped (T)
- Sleep
  - Interruptible (S)
  - Uninterruptible (D)
- Zombie (Z)