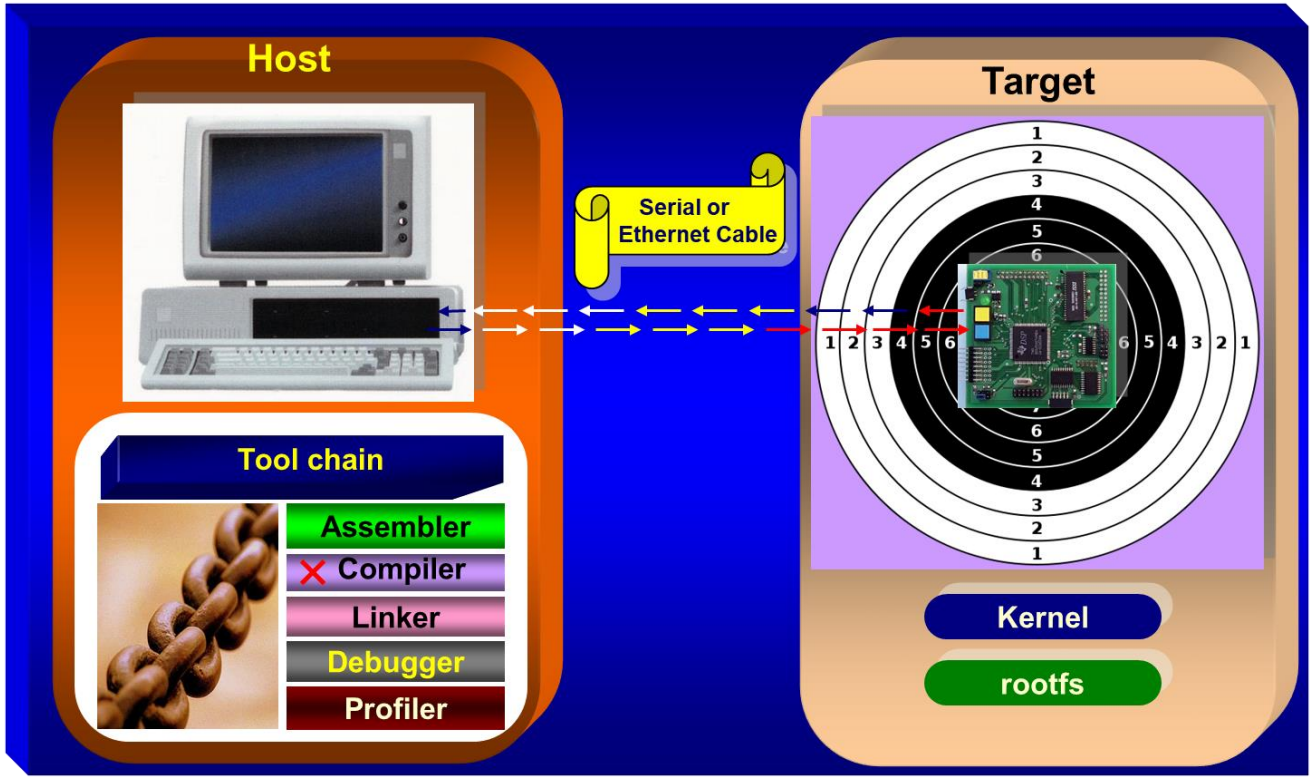
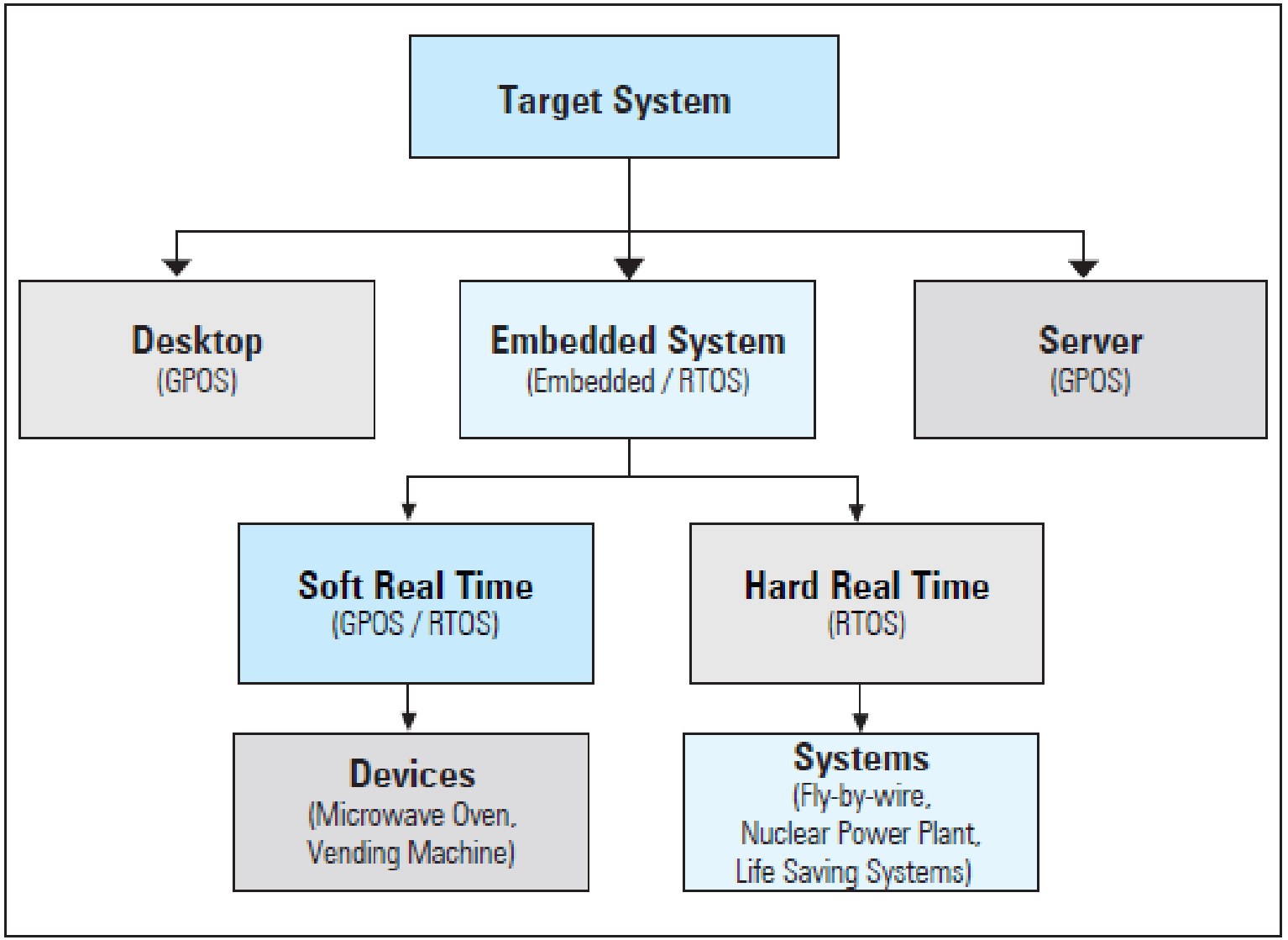
CS 513 Software System Recap

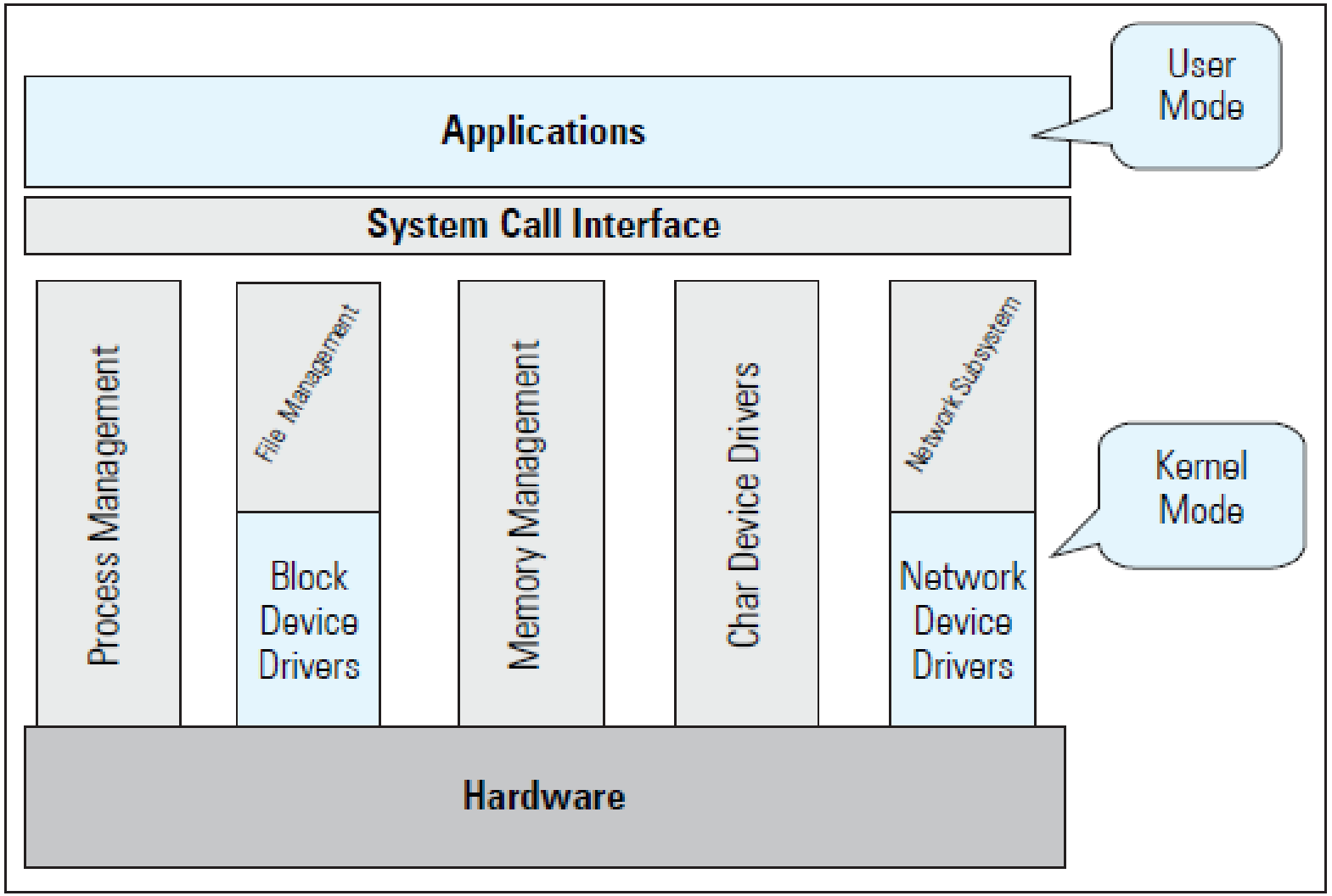


Prof. B Thangaraju

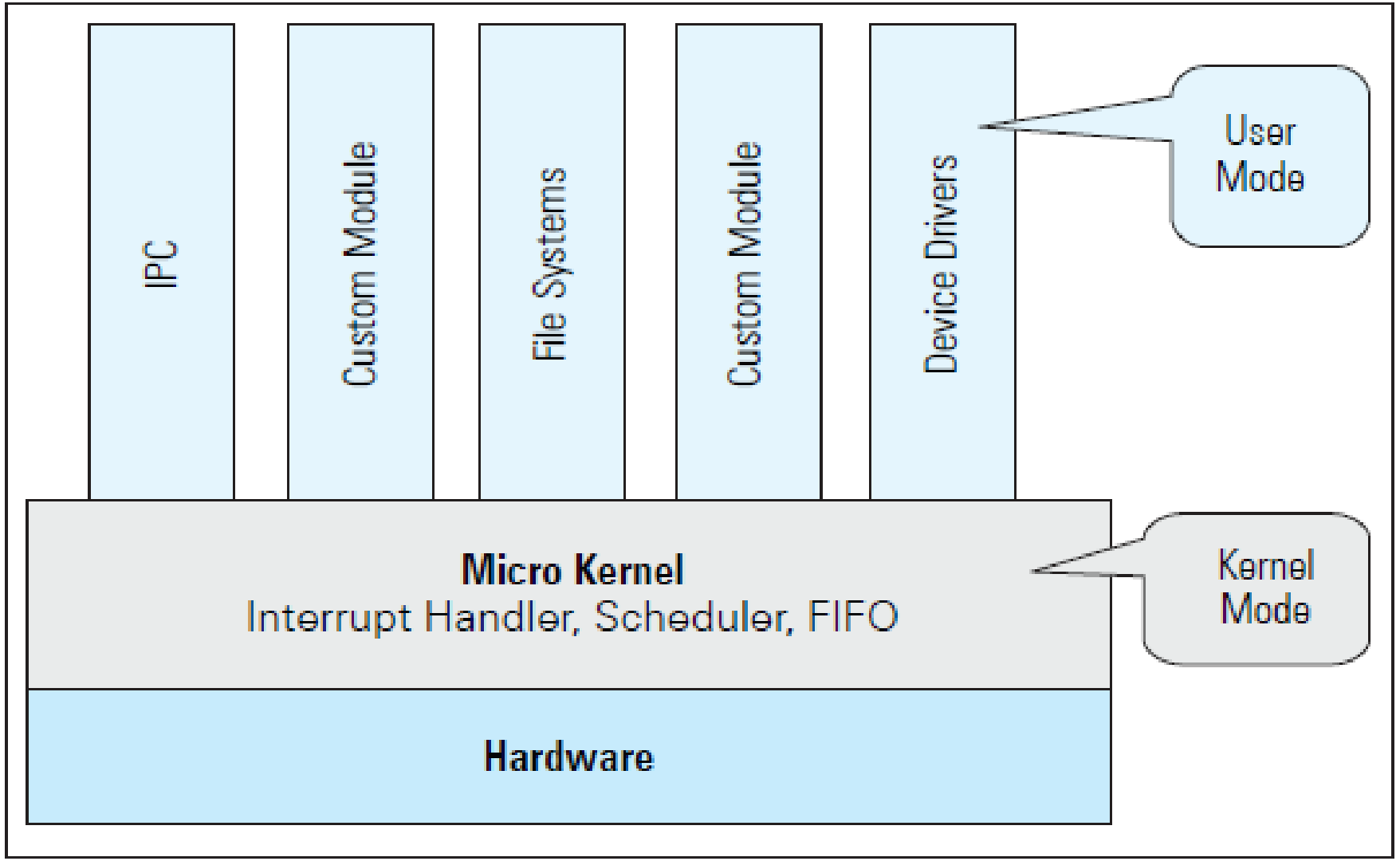
Types of System



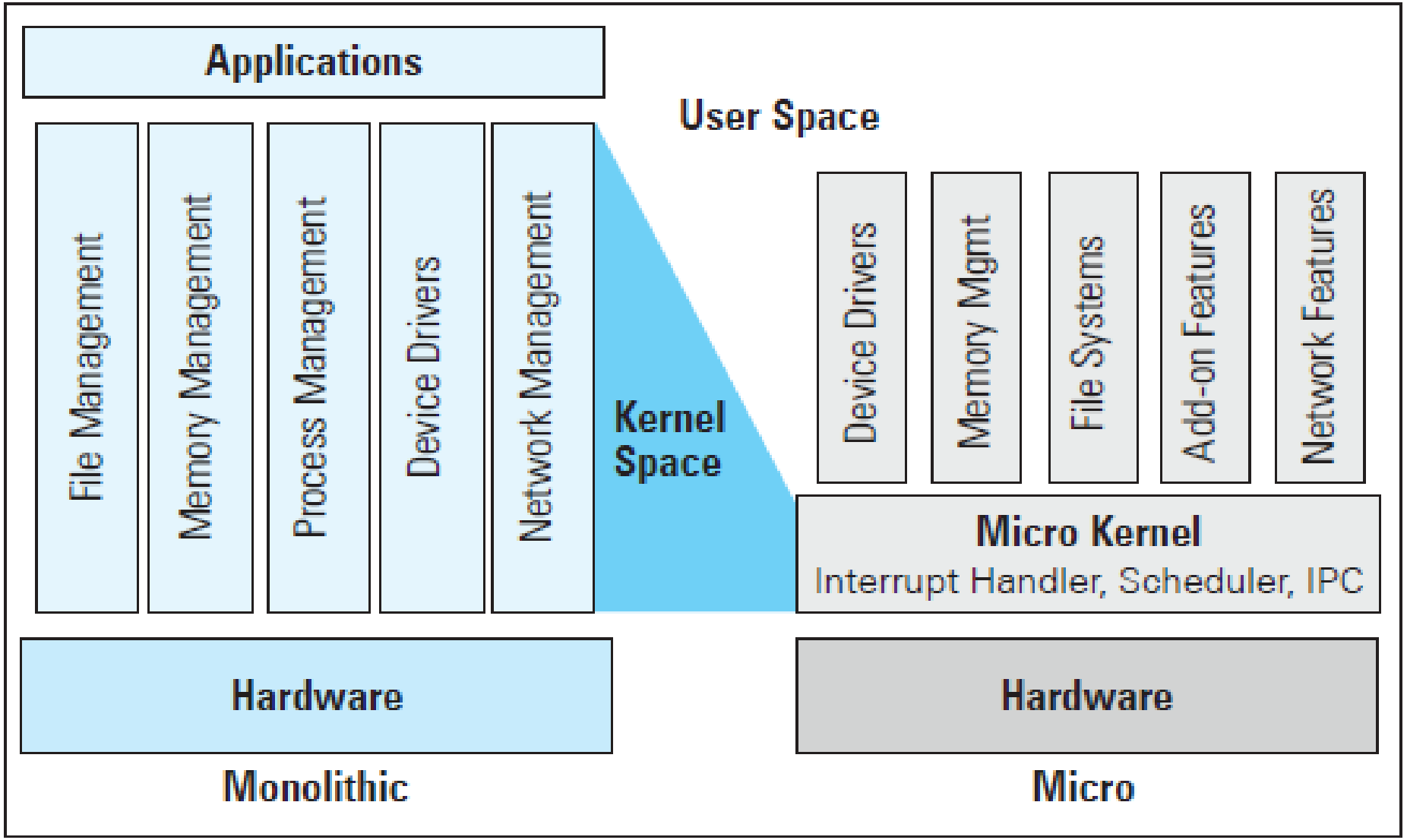
GPOS


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

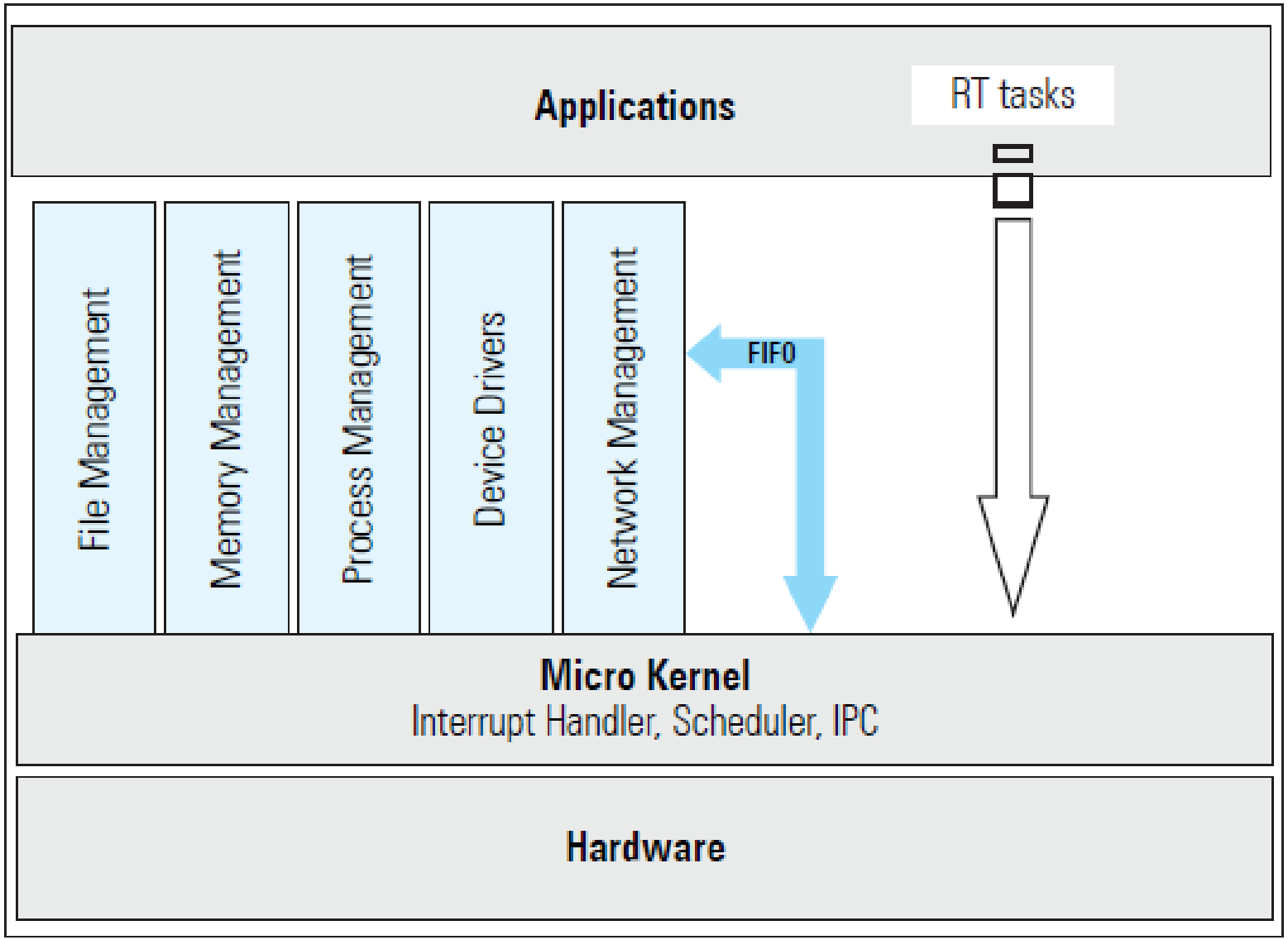
Micro Kernel Architecture



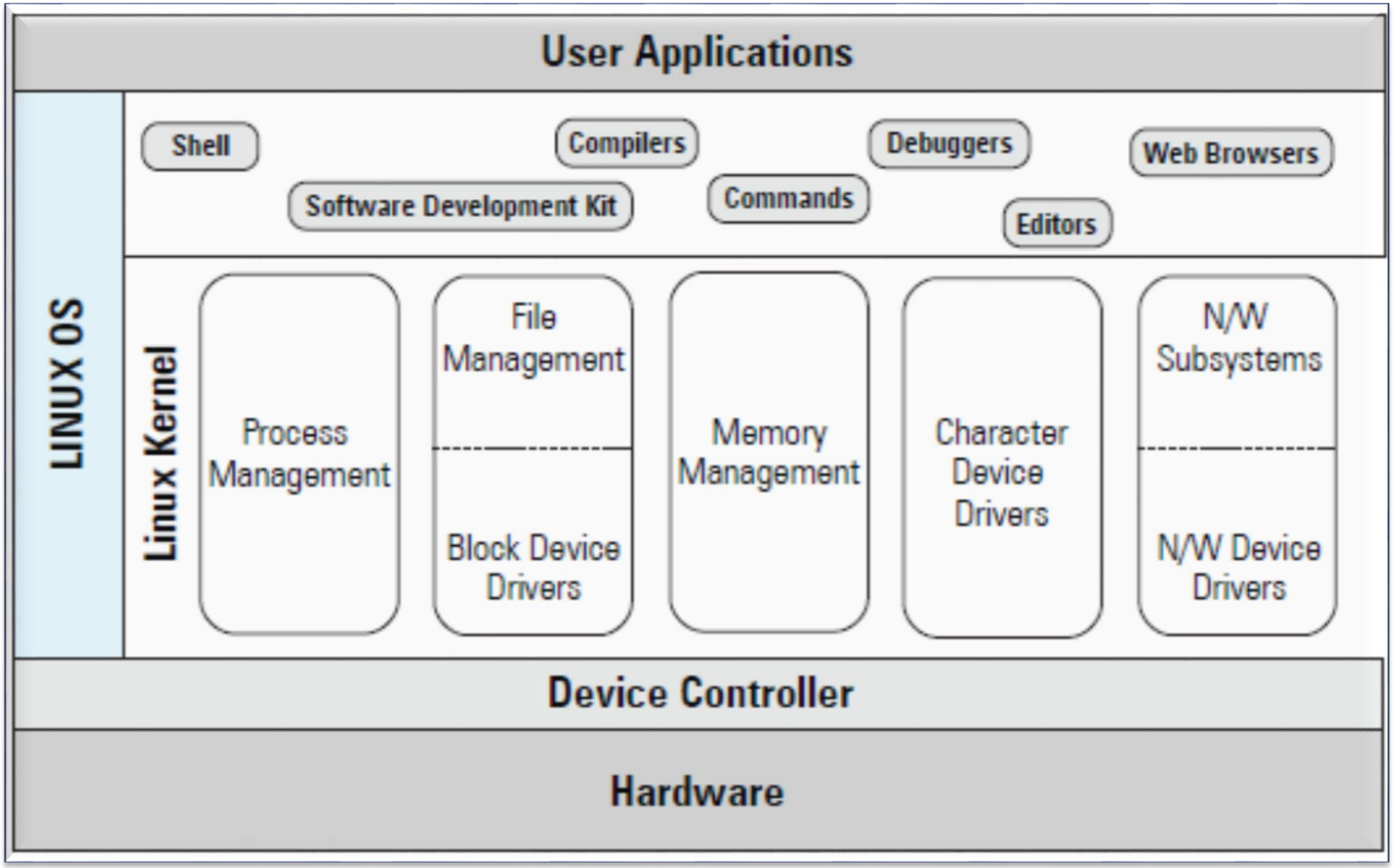
Monolithic Vs Micro Kernel



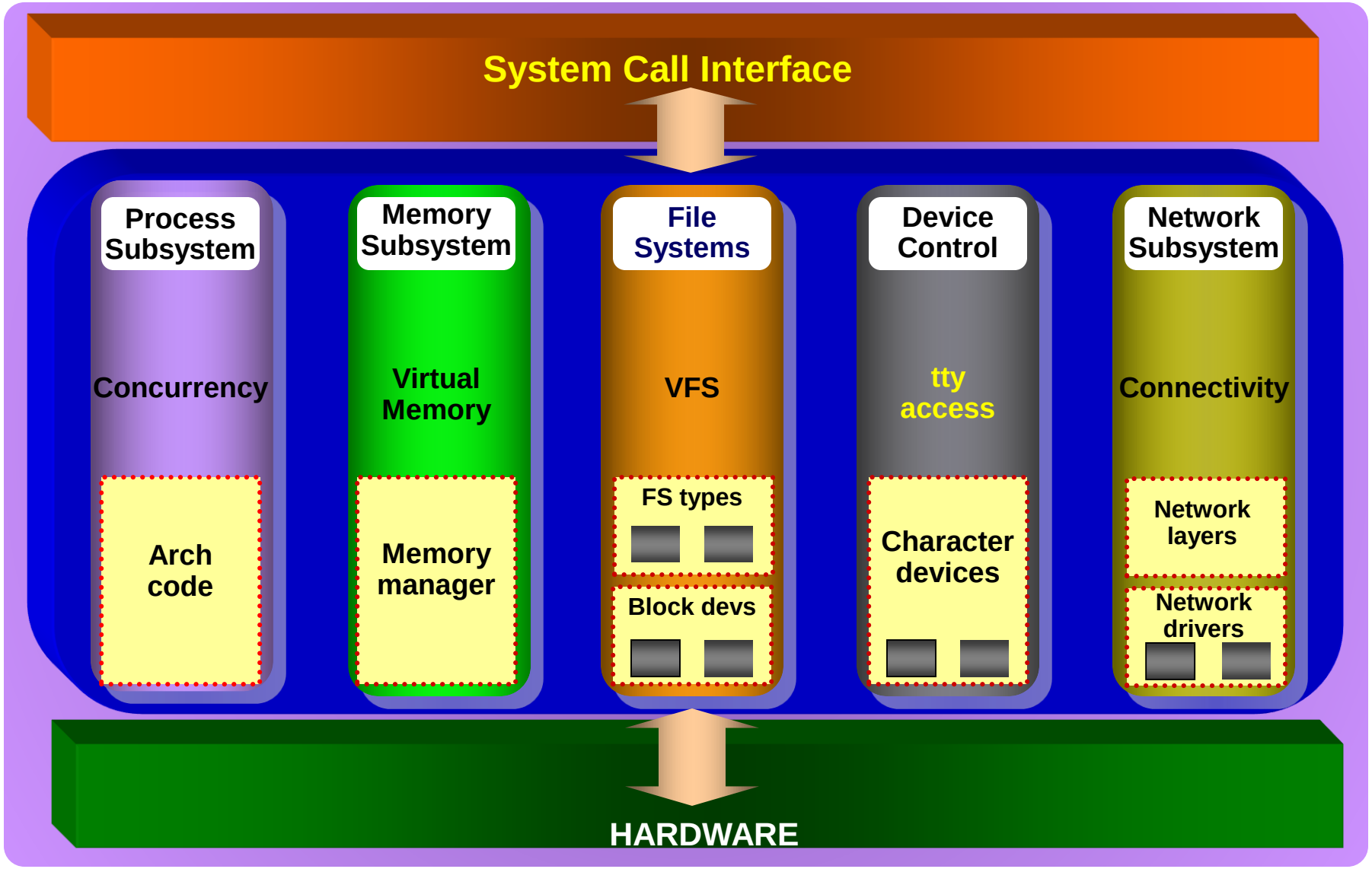
Hybrid Kernel Architecture



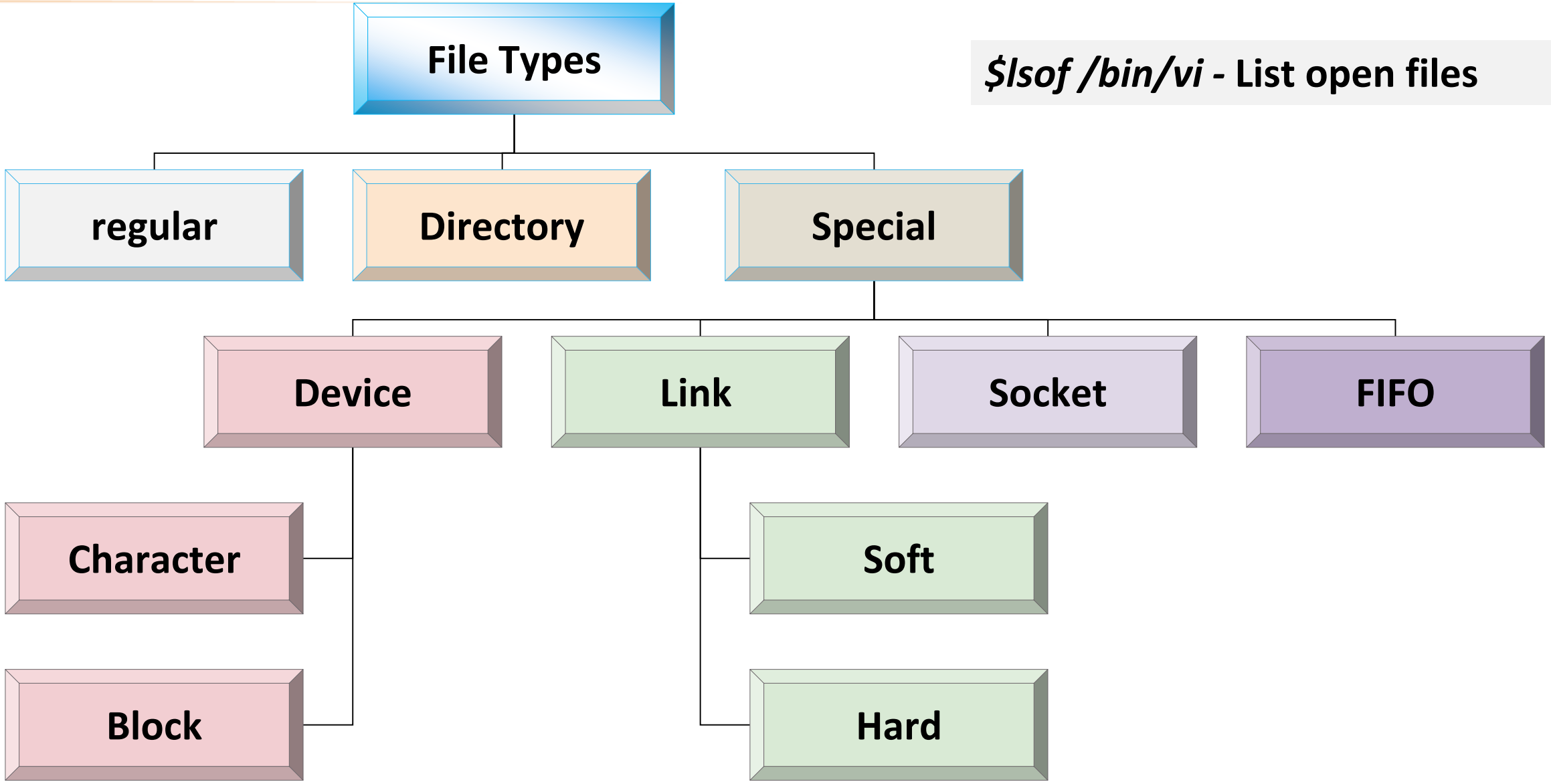
Introduction – Linux OS



Linux Kernel Architecture



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

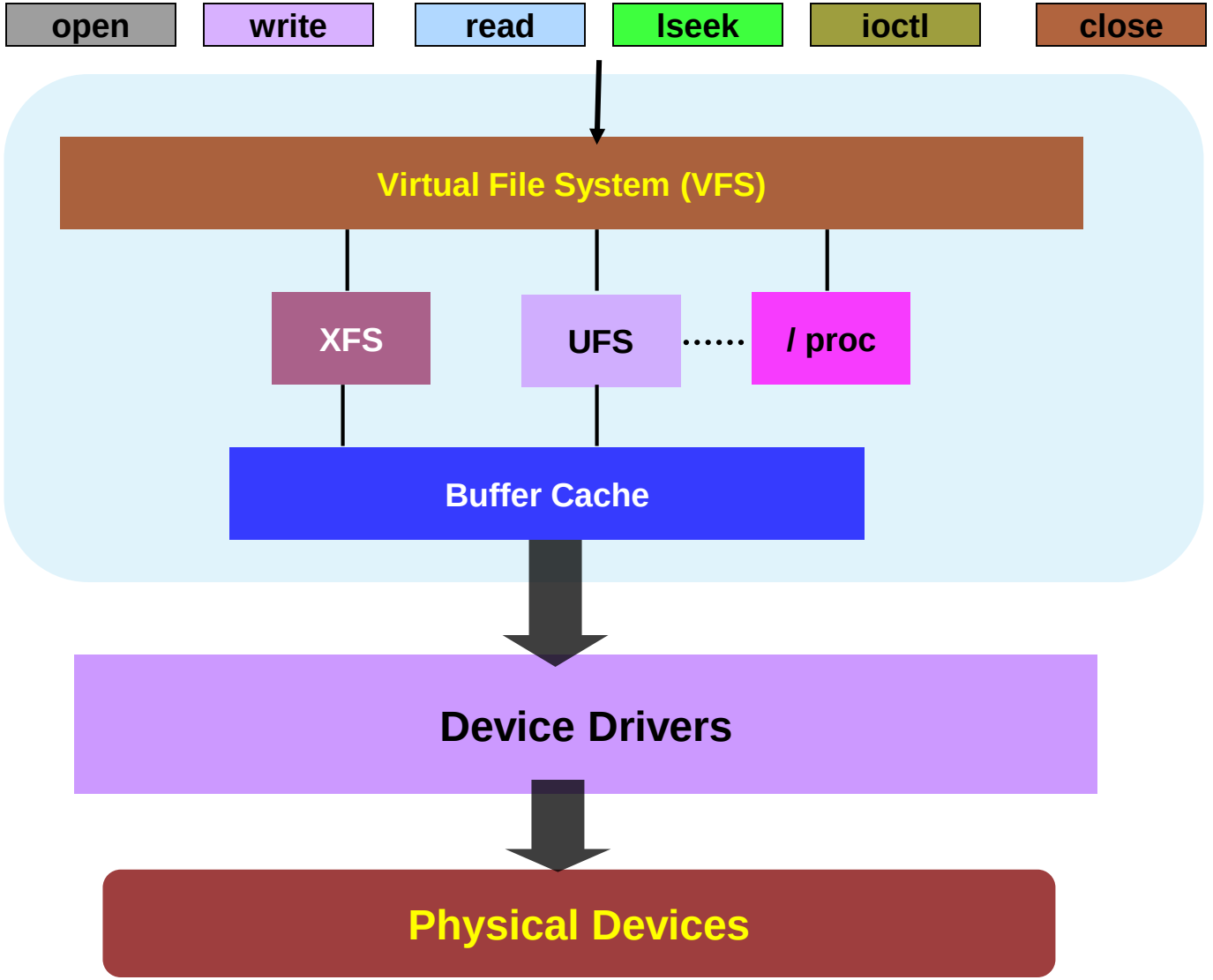
- FIFO (p)
- Socket (S)
- Link File
  - Soft Link (l)
  - Hard Link (inode numbers are same)
- Device File
  - Character (c)
    - Example: Monitor, Keyboard, Mouse, Tape
  - Block (b)
    - Example: Hard disk, CDROM, Floppy

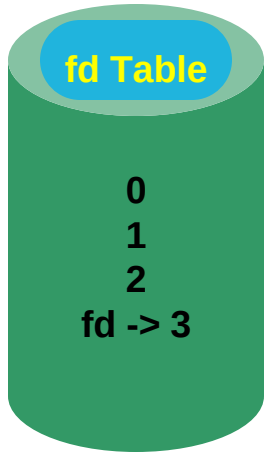
File System

- 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





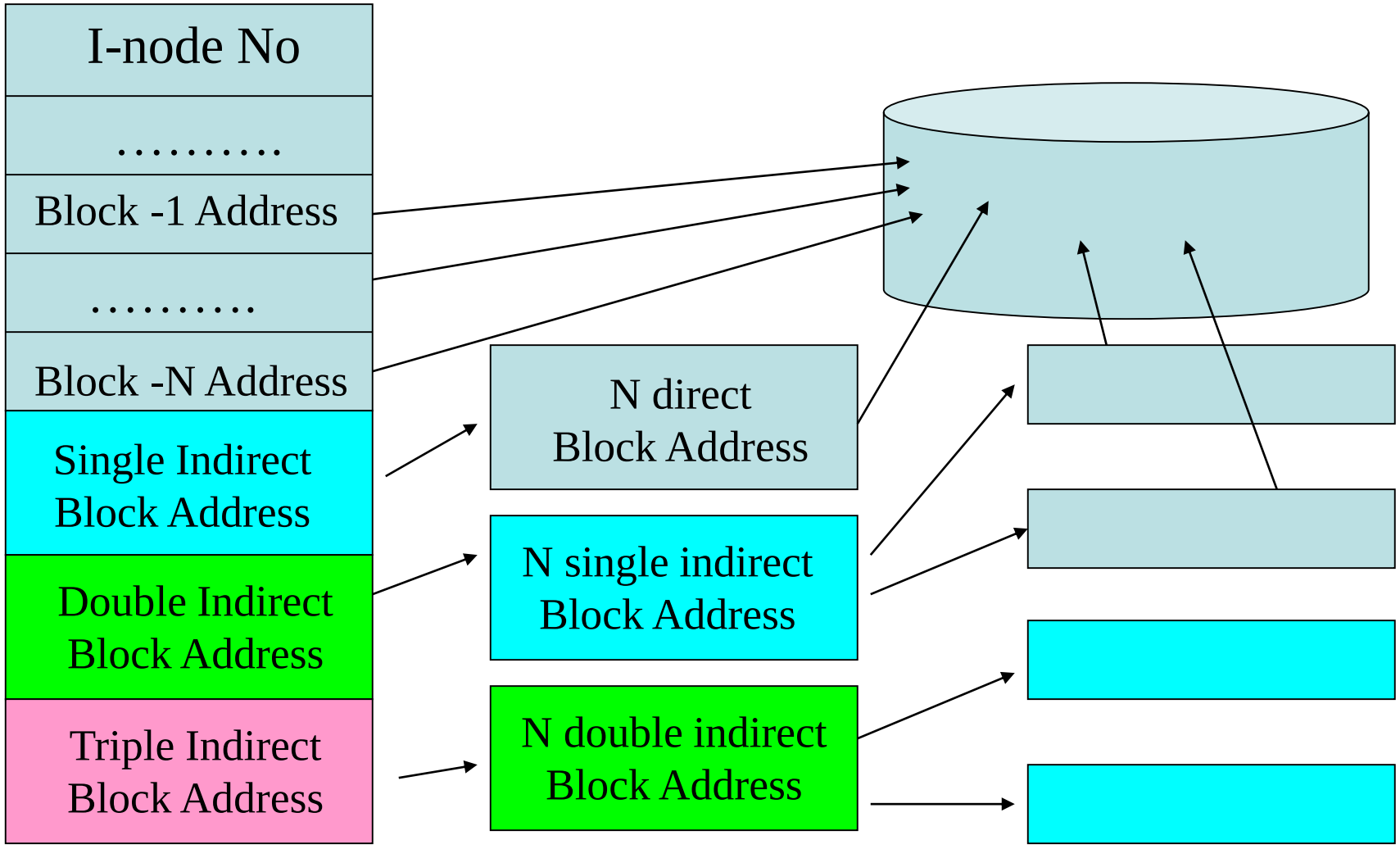
Proc File System

- 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.

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**

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.
- 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.

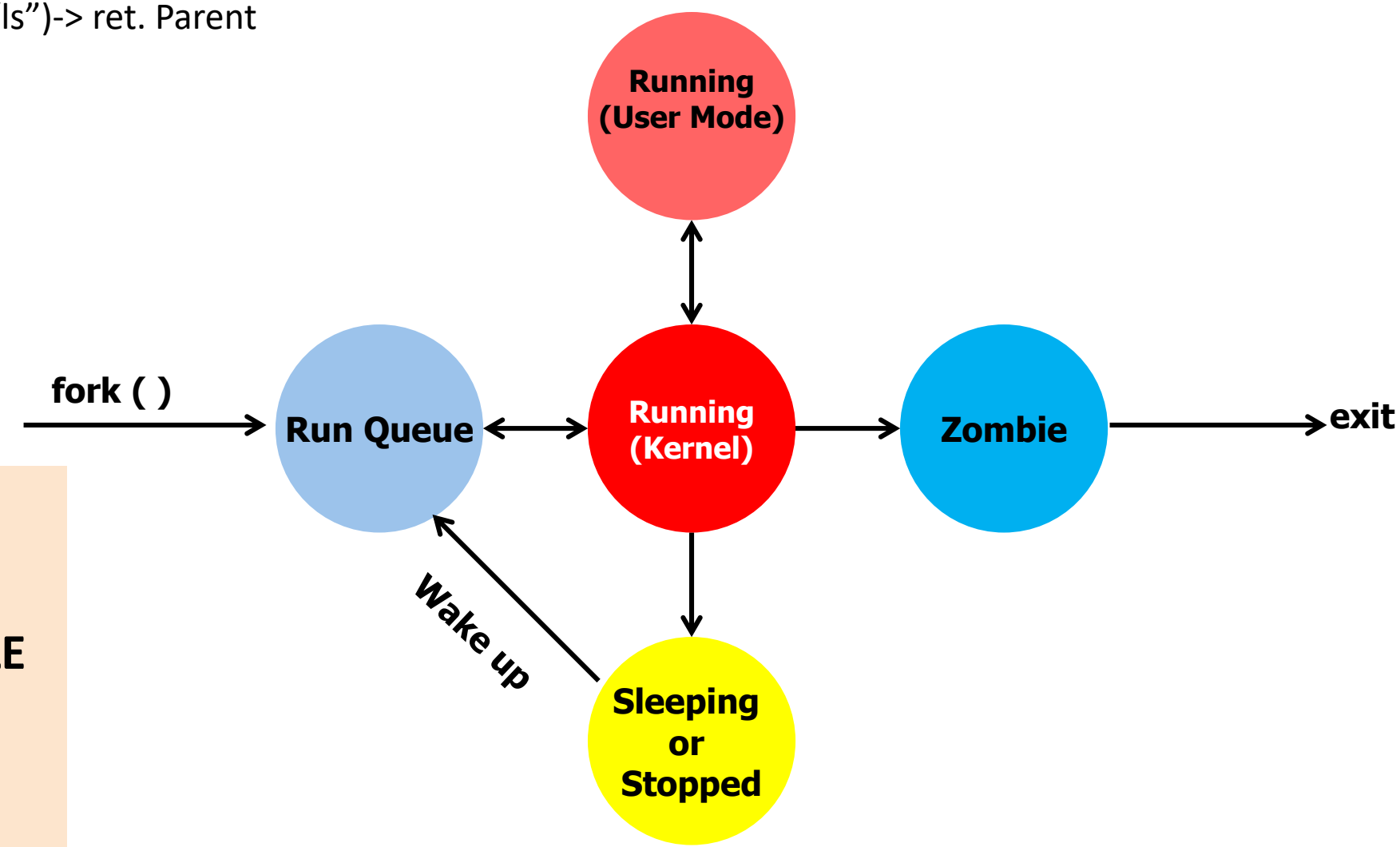
Execution Context

- 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.

Process States

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



- TASK_RUNNING
- TASK_INTERRUPTIBLE
- TASK_UNINTERRUPTIBLE
- TASK_STOPPED
- TASK_ZOMBIE

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, ….);
}
```

Importance of “\n”.

Process tree structure

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.

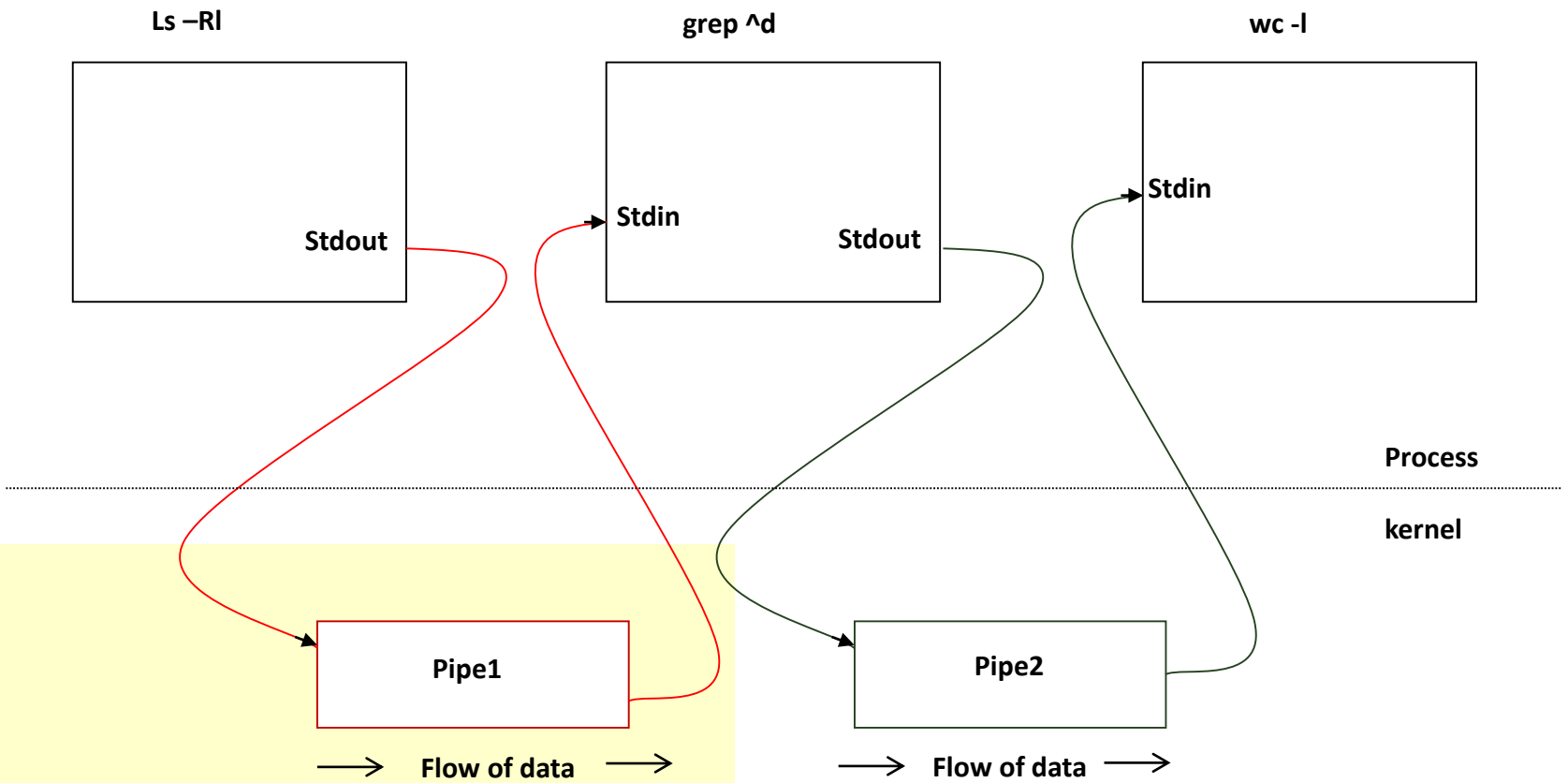
Inter Process Communications (IPC) Mechanisms

- 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

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

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**

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

```
int main()
{
  int fd1[2],fd2[2];

  pipe(fd1);
  pipe(fd2);

       if(!fork()) {
         dup2(fd1[1],1);
         close(fd1[0]);   //closing read end of pipe
         close(fd2[0]);
         close(fd2[1]);
         execlp("ls","ls","-Rl",(char *)NULL);
       }
       else {
```

```
            if(!fork()) {
                dup2(fd1[0],0); //read end of first pipe
                dup2(fd2[1],1); //duplicated the write end of 2nd pipe
                close(fd1[1]); //closing the write end;
                close(fd2[0]); //closing read end of 2nd pipe
                execlp("grep","grep","^d",(char *)NULL);
                }
              else {
                 dup2(fd2[0],0);
                 close(fd2[1]);
                 close(fd1[0]);
                 close(fd1[1]);
                 execlp("wc","wc","-l",(char *)NULL);
                }
        }
}
```

Semaphore

- Synchronization Tool
- An Integer Number
- P ( ) And V ( ) Operators
- Avoid Busy Waiting
- Types of Semaphore

Used in :

- shared memory segment
- message queue
- file

p and v operations

Incrementing Operations:

```
int v (int i)
 {
  i = i + 1; (unlock)
  return i;
 }
```

- 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

Decrementing Operations:

```
int p (int i)  {
  if (i > 0)
   then
    i--; (lock)
   else
    wait till i > 0;
   return i;
  }
```

- 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 Implementation

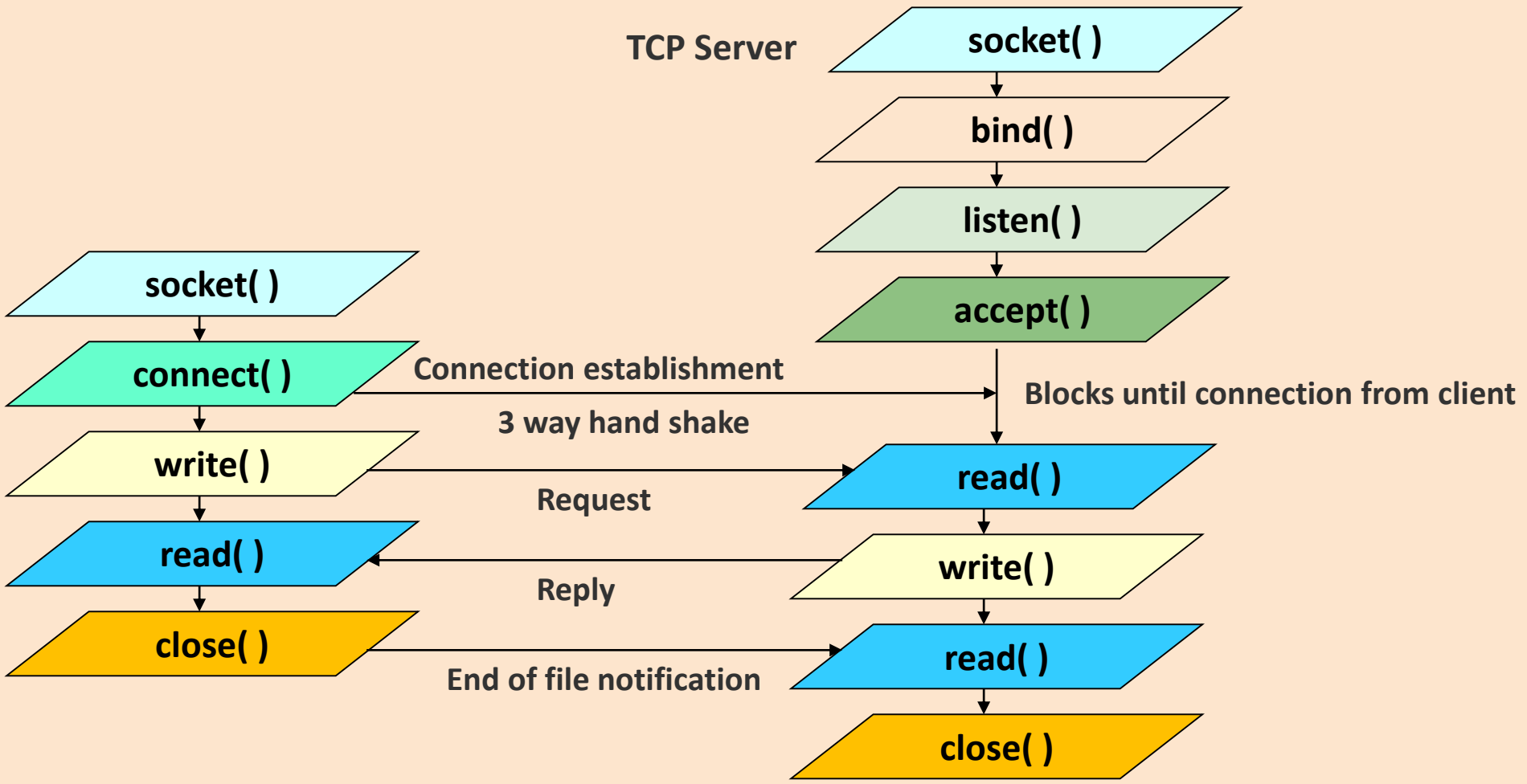
```
union semun {
  int val;      // value for SETVAL
  struct semid_ds *buf;  // buffer for
   IPC_STAT, IPC_SET
  unsigned short int *array;  // array for
   GETALL, SETALL
 };

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);
```

```
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 */
```

Socket Functions



Alarm and Timers

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

- **Three interval timers.**
  - **ITIMER_REAL**
    - This timer counts down in real (i.e., wall clock) time. At each expiration, a ***SIGALRM*** signal is generated.
  - **ITIMER_VIRTUAL**
    - This timer counts down against the user-mode CPU time consumed by the process. (The measurement includes CPU time consumed by all threads in the process.) At each expiration, a ***SIGVTALRM*** signal is generated.
  - **ITIMER_PROF**
    - This timer counts down against the total (i.e., both user and system) CPU time consumed by the process. At each expiration, a ***SIGPROF*** signal is generated.

Time Stamp Counter

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;
 }
```

```
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 */
```

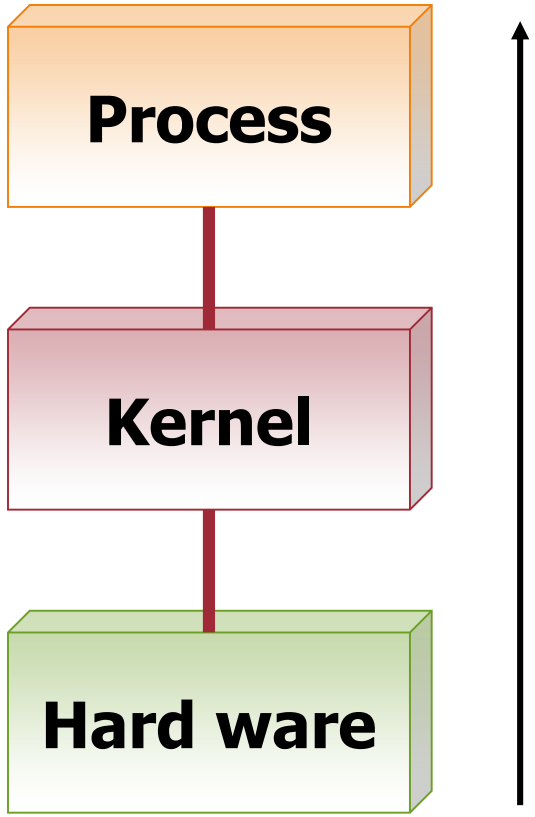
Signals Introduction

- 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)

- 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.

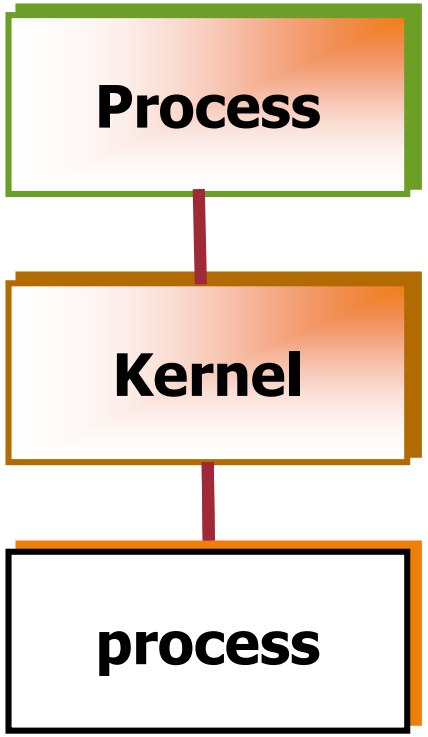
Signal Vs Interrupt

Interrupt

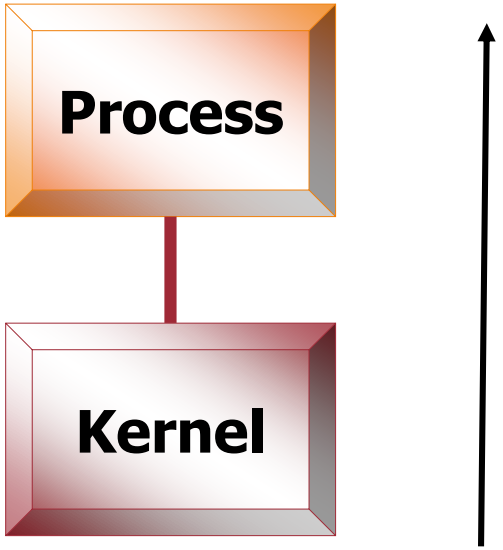


I/O operation (ex: mouse click)

Signal



$ kill -9 pid



Illegal instruction

signal System Call

- 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

- 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 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