System Software

Agenda

1 Inter Process Communications

2 Signals

3 Multithreading

4 Timers and Resource Limits

5 Memory Management

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.

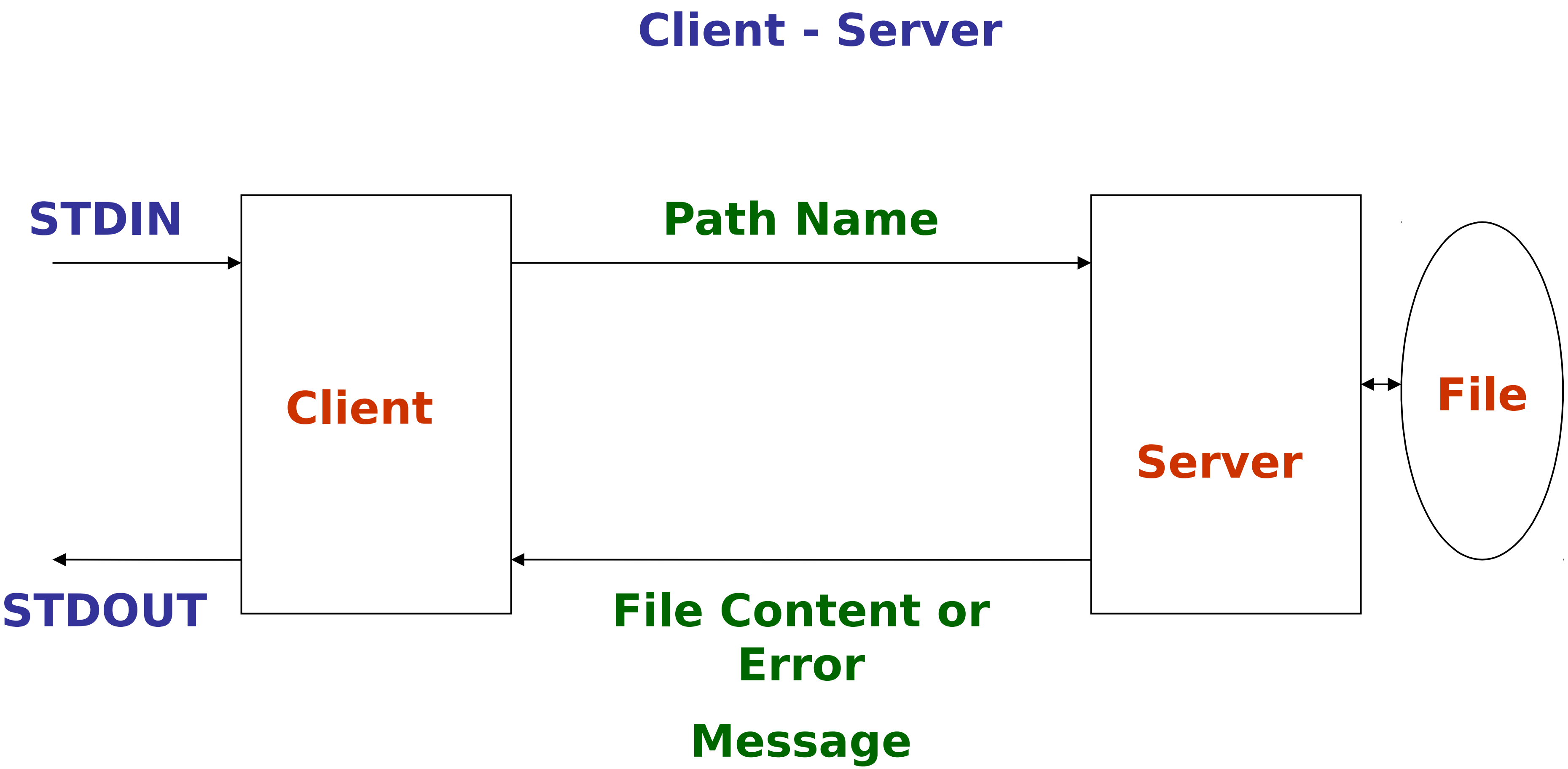
- The shared resources must also be synchronized from the concurrent access by many processes.

- 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

Pipe Example

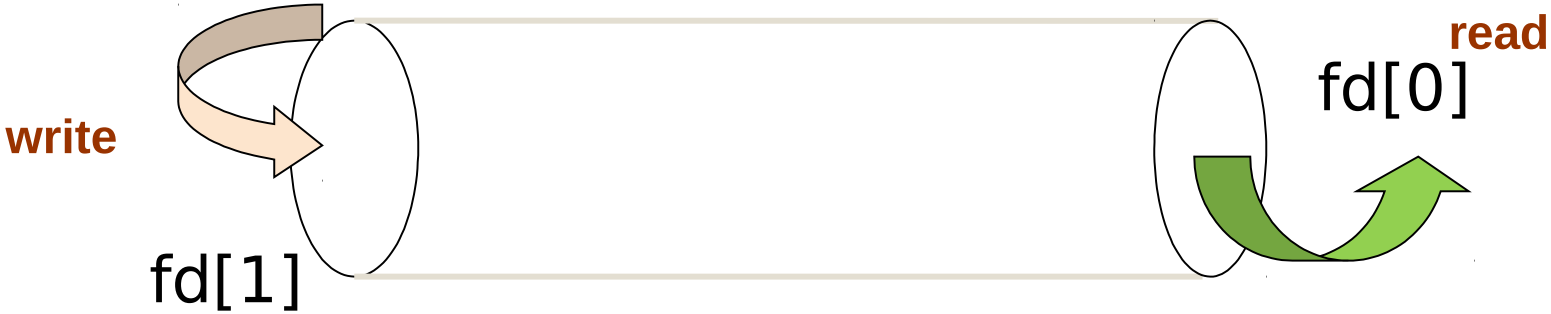


pipe (or unnamed pipe “|”)

- 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.

- 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 – half duplex

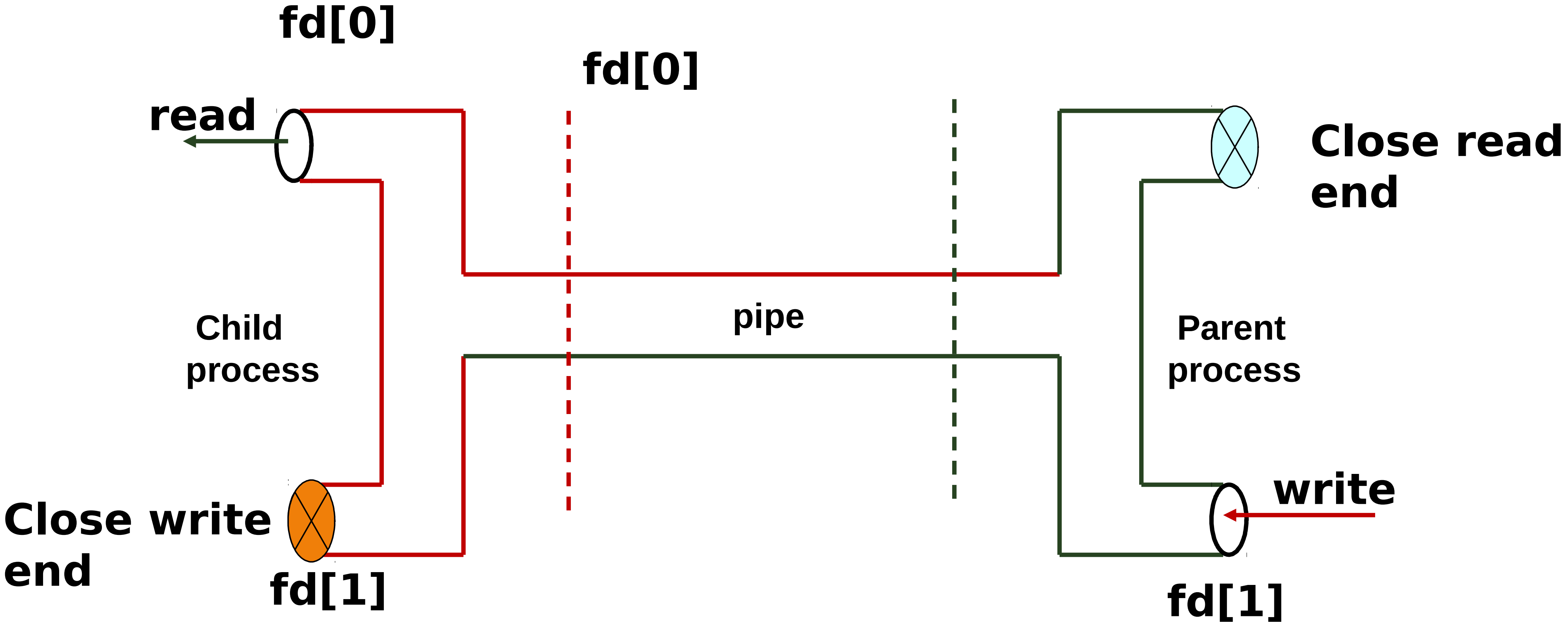



```
int fd[2];
pipe(fd);
    -returns with fd[0], fd[1];

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

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

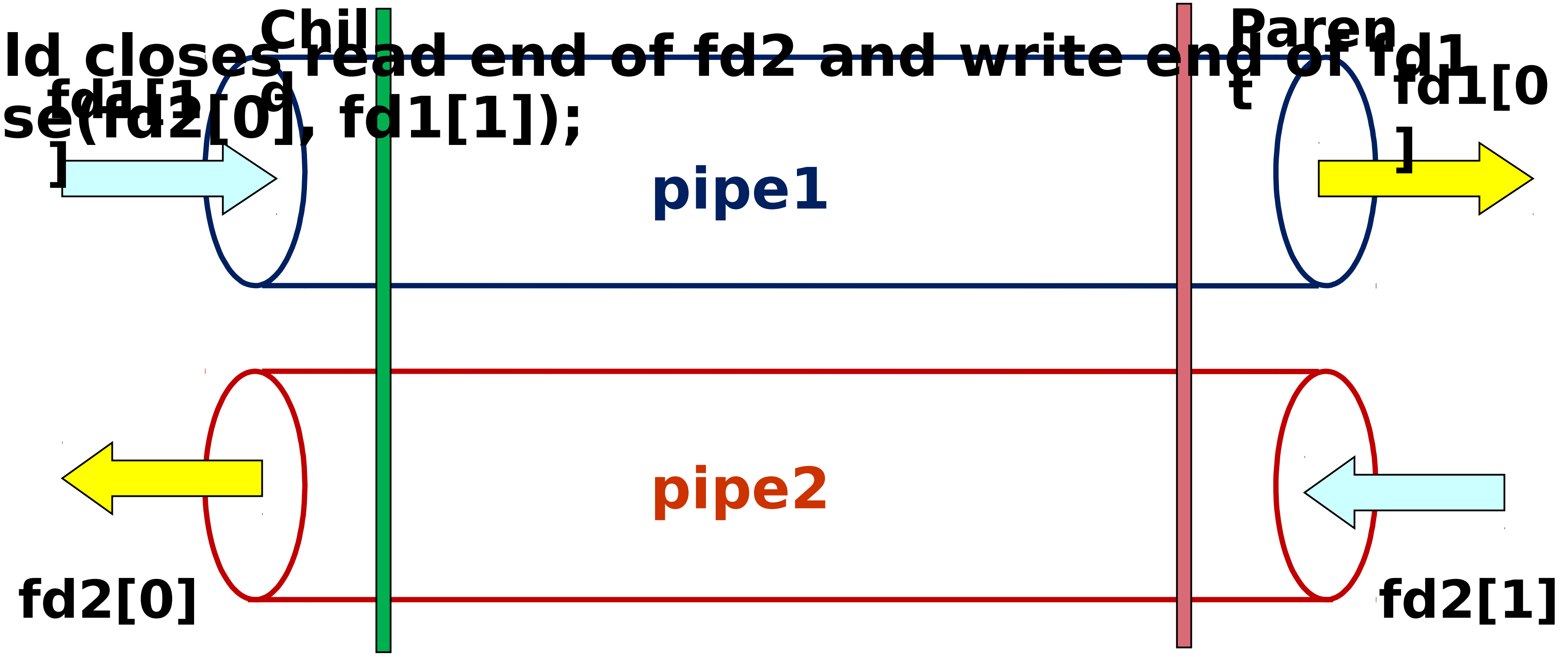
One way data transfer from related process

One-way communication from parent to child

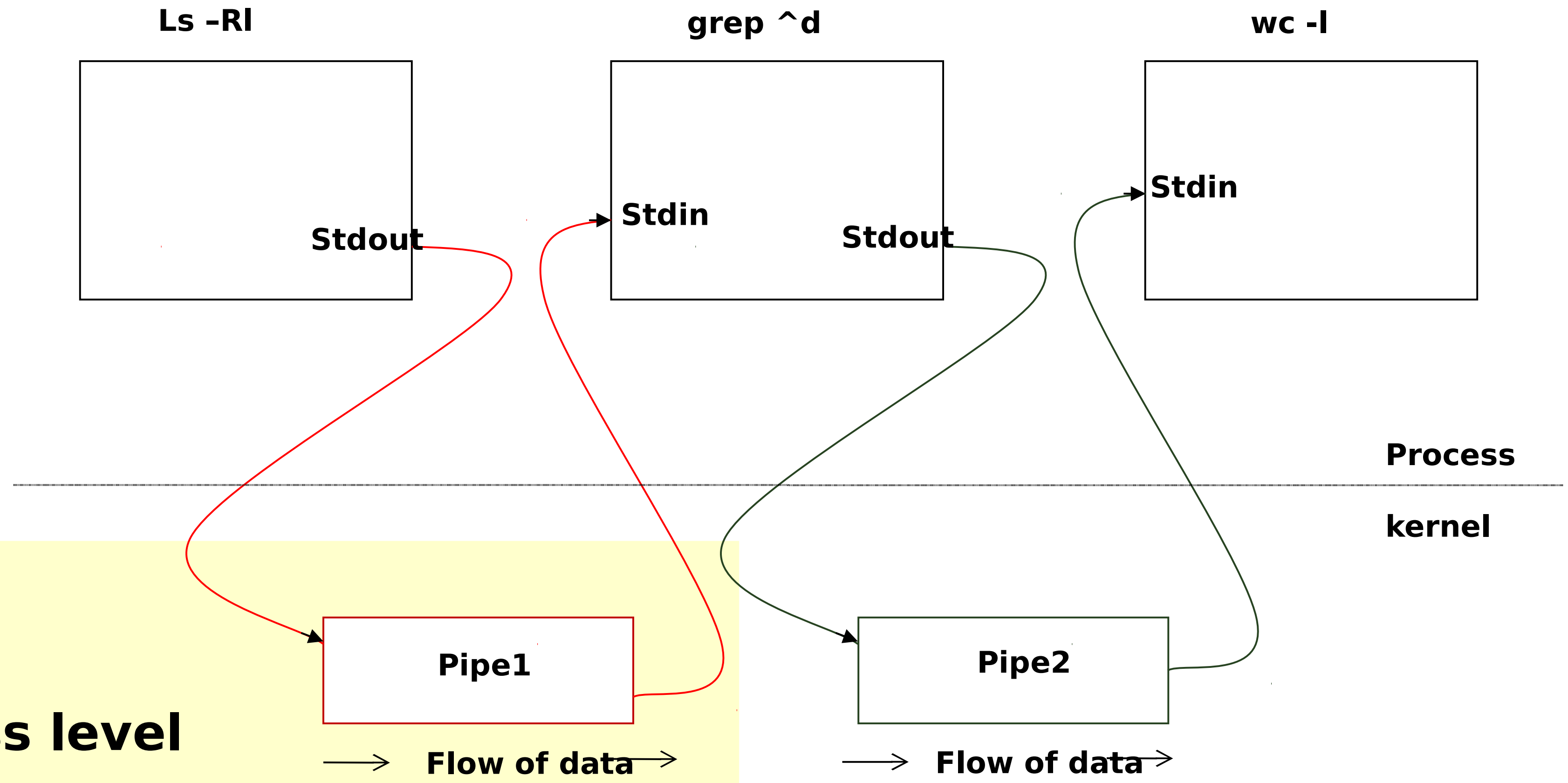


Two Way Communication

- 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]);



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 ( ) Library Function

- 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);
- 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

```
root@localhost:~
[root@localhost ~]# mkfifo myfifo
[root@localhost ~]# ll myfifo
prw-r--r--. 1 root root 0 Aug 26 14:36 myfifo
[root@localhost ~]#
```

- **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 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);**

FIFO

- **Once a FIFO is created, you can use file's related system calls (open, read, write, select, close etc.,) 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 store data and you cannot use FIFO across network.**
- **Less secure than a pipe, since any process with valid access permission can access data.**
- **No message boundaries. Data is treated as a stream of bytes.**

FIFO Limitation

- **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).**

```
#include <unistd.h>
 main () {
  long PIPE_BUF, OPEN_MAX;
  PIPE_BUF = pathconf(".", _PC_PIPE_BUF);
  OPEN_MAX = sysconf (_SC_OPEN_MAX);
  printf ("Pipe_buf = %ld\t OPEN_MAX =
                     %ld\n", PIPE_BUF, OPEN_MAX);
 }
```

System V IPC

root@localhost:/home/raju

```
SVIPC(7)                    Linux Programmer's Manual                   SVIPC(7)

NAME
       svipc - System V interprocess communication mechanisms

SYNOPSIS
       #include <sys/msg.h>
       #include <sys/sem.h>
       #include <sys/shm.h>

DESCRIPTION
       This  manual  page refers to the Linux implementation of the System V inter
       process communication (IPC) mechanisms: message queues, semaphore sets,  and
       shared  memory  segments.   In  the  following,  the  word resource means an
 Manual page ipc(5) line 1 (press h for help or q to quit)
```

- **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 : MQ, SHM and SEM**
- **Persist till explicitly delete or reboot the system**

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.**

Key:

- **the first step is to create a shared unique identifier.**
- **The simplest form of the identifier is a number**
- **the system generates this number dynamically by using the *ftok* library function.**
- **Syntax: key_t ftok (const char *filename, int id);**

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.**

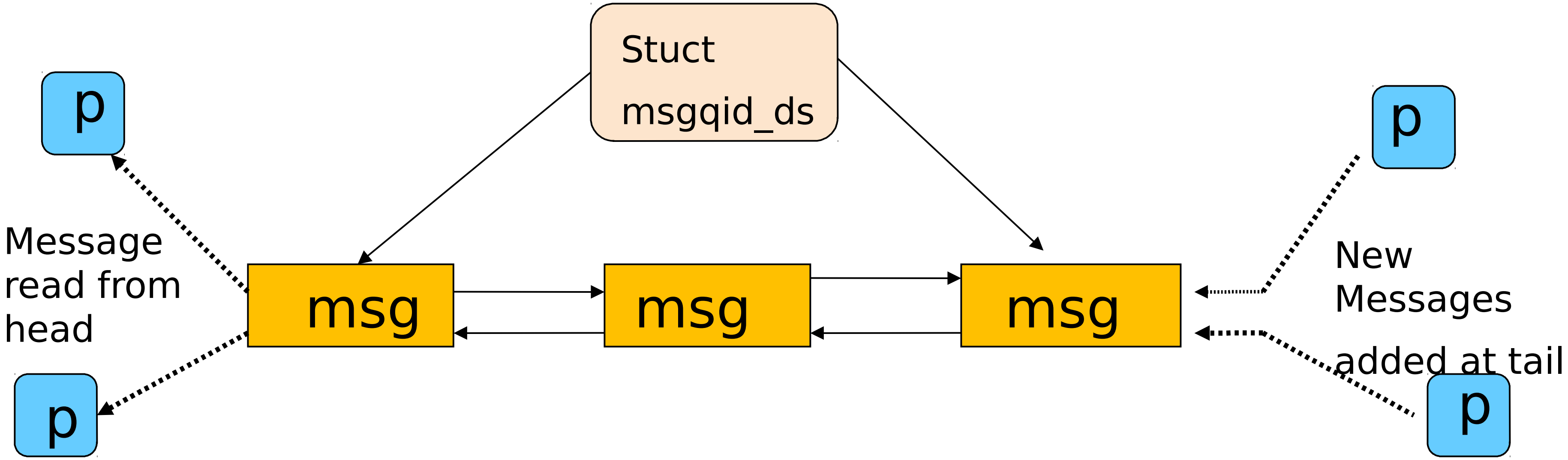
- **The syntax for the *control* function is:**

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 Q

- **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 Q

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

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

msgid
xxx

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

MQ System Calls

- **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);

- **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**

- **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**

MQ System Calls

- 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

- 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

MQ Limitations

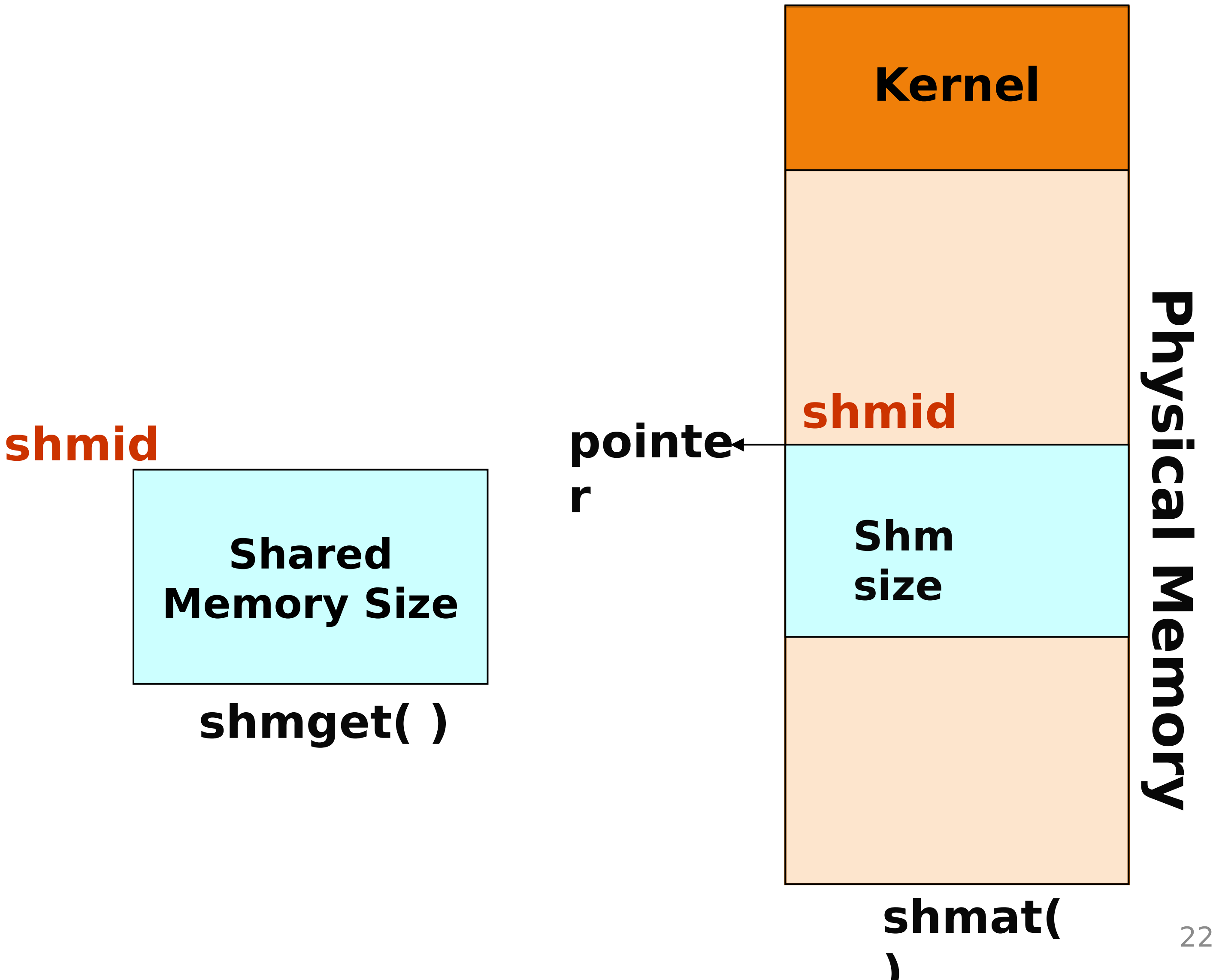
- **Message queues are effective if a small amount of data is transferred.**
- **Very expensive for large transfers.**
- **During message sending and receiving, the message is copied from user buffer into kernel buffer and vice versa**
- **So each message transfer involves two data copy operations, which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

root@localhost:/home/raju

```
[root@localhost raju]# ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

[root@localhost raju]#
```

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

- **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.**

- **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**

shm – system calls

- shmget system call is used to create a shared memory segment.
- The syntax:

  int shmget (key_t key, int size, int shmflg);

  key: the return value of ftok function.

  size: size of the shared memory.

  shmflg: IPC_CREAT|0744
- On success the shmget returns the shared memory ID or else it returns -1.

- 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.

shm – system calls

- 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

Example -

- Read:
  - printf (“SHM contents : %s \n”, data);
- Write:
  - printf (“”Enter a String : ”);
  - scanf (“ %[^\n]”,data);

- 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

shm –system calls

- 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);

- Data can either be read or written only. Append ?
- 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.

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;
  }
```

Decrementing Operations:

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

- 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

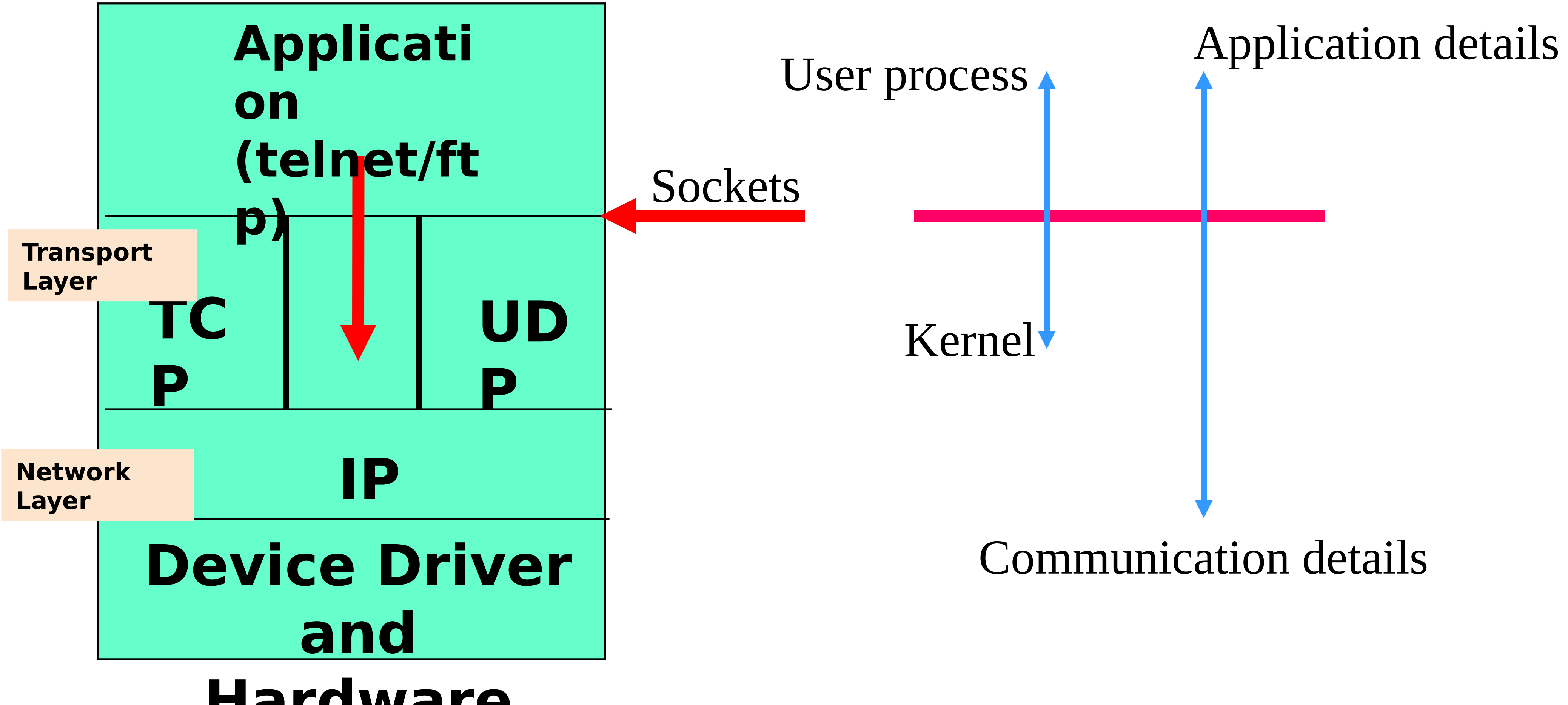
Semaphore Implementation

```
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);
```

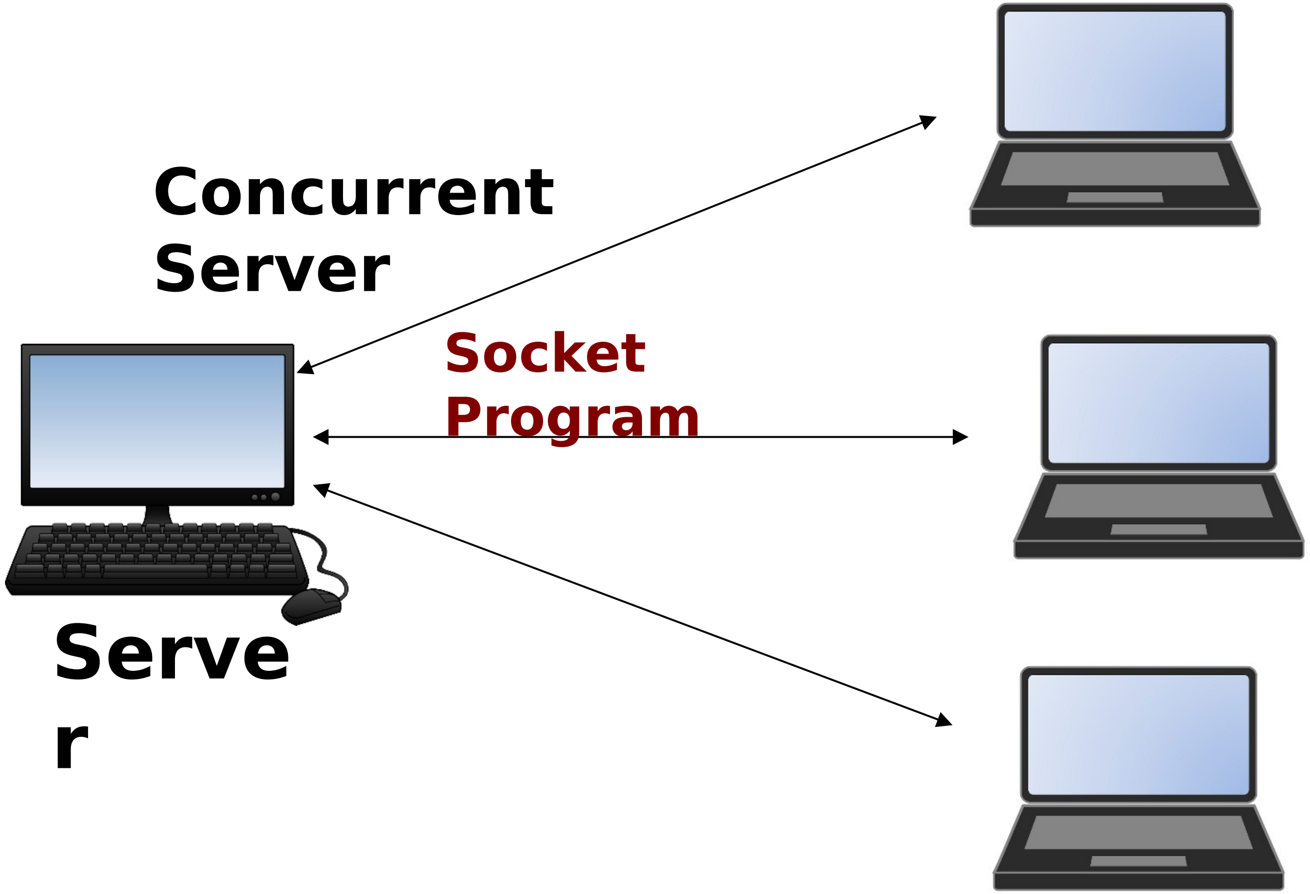
```
struct sembuf {
  short sem_num;  /* semaphore number: 0 means first */
  short sem_op;   /* semaphore operation: lock or unlock */
  short sem_flg;  /* operation flags : 0, SEM_UNDO, IPC_NOWAIT */
  };
struct sembuf buf = {0, -1, 0};  /* (-1 + previous value) */
semid = semget (key, 1, 0);

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

Socket Programming - TCP/IP Protocol Stack

A socket is used to communicate between different machines (different IP addresses). Socket of type SOCK_STREAM is full-duplex byte streams.



Socket Programming – Client Server Model



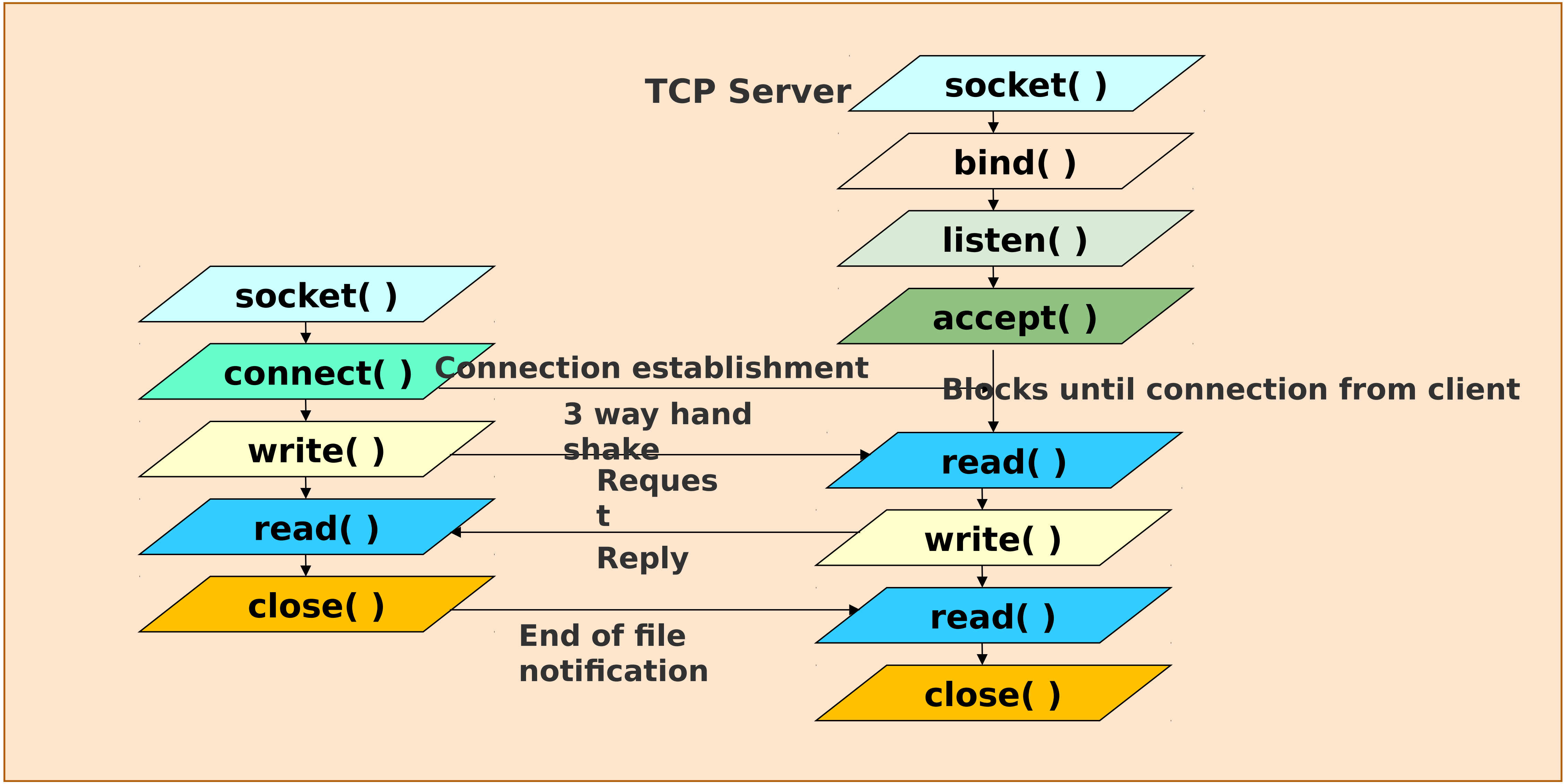
- A socket is a communication endpoint and represents abstract object that a process may use to send or receive messages.
- The two most prevalent communication APIs for Unix Systems are Berkeley Sockets and System V Transport Layer Interface(TLI)

socket ( ) system call

- The typical client - server relationship is not symmetrical
- Network Connection can be connection-oriented or connectionless
- More parameters must be specified for network connection, than for file I/O
- The Unix I/O system is stream oriented
- The network interface should support multiple communication protocol

- int socket (int domain,int type,int protocol);
- Domain (AF – Address Family)
  - AF_UNIX for UNIX domain
  - AF_INET for Internet domain
- Socket type
  - SOCK_STREAM for TCP (Connection Oriented)
  - SOCK_DGRAM for UDP (Connectionless)
- Protocol
  - Protocol number is used to identify an application. List of the protocol number and the corresponding applications can be seen at /etc/protocols.
- The socket system call returns a socket descriptor on success and -1 for failure.

Socket Functions



sock structure

- struct sockaddr_in {

```
    short int sin_family;
    unsigned short int sin_port;
   struct in_addr sin_addr;
 }
```

- sin_family - address family
- sin_port - port number
- sin_addr - internet address (IP addr)

- The in_addr structure used to define sin_addr is as under

```
struct in_addr {
     unsigned long s_addr;
/* refers to the four byte IP address */
 }
```

Little endian byte order : example: Intel series

High order byte | low order byte

Address A+1 00 01 64 C1 Address A

Increasing memory address

Big endian byte order : example: IBM 370, Motorola

High order byte | low order byte

Address A+1 C1 64 01 00 Address A

Increasing memory address

Byte ordering ex: 91,329 hex: 00 01 64 C1

Socket system calls

- Internet protocols use big endian byte ordering called network byte order
- The following functions allow conversions between the formats.

#include <netinet/in.h>

htons() – "Host to Network Short"

htonl() – "Host to Network Long"

ntohs() – "Network to Host Short"

ntohl() – "Network to Host Long"

- h stands for host n stands for network
- s stands for short l stands for long

- int bind (int sockfd, struct sockaddr *my_addr,int addrlen);
- sockfd - the socket file descriptor returned by socket().
- my_addr - a pointer to a struct sockaddr that contains information about IP address and port number.
- addrlen - set to sizeof (struct sockaddr)

Socket system calls

int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);

- **sockfd - the socket file descriptor returned by socket().**
- **serv_addr - is a struct sockaddr containing the destination port and IP address.**
- **addrlen - set to sizeof (struct sockaddr).**

int listen (int sockfd,int backlog);

- **sockfd - the socket file descriptor returned by socket().**
- **backlog - the number of connections allowed on the incoming queue.**
- **Backlog should never be zero as servers always expect connection from client.**
- **The listen function converts an unconnected socket into a passive socket,**
- **On successful execution of listen is indicating that the kernel should accept incoming connection requests directed to this socket.**

Socket system calls

- int accept (int sockfd, void *addr, int *addrlen);
- sockfd
  - the socket file descriptor returned by socket().
- addr
  - a pointer to a struct sockaddr_in. The information about the incoming connection like IP address and port number are stored.
- addrlen
  - a local integer variable that should be set to sizeof (struct sockaddr_in) before its address is passed to accept().

- Socket descriptor can be closed like file descriptor.

close (sockfd);

- Close system call prevents any more reads and writes to the socket. For attempting to read or write the socket on the remote end will receive an error.

Socket system calls

- **int shutdown (int sockfd, int how);**

- **sockfd - socket file descriptor of the socket to be shutdown.**
- **how – if it is**
  - **0 - Further receives are disallowed**
  - **1 - Further sends are disallowed**
  - **2 - Further sends and receives are disallowed.**

- **The shutdown system call gives more control (than close (sockfd) over how the socket descriptor can be closed.**

Socket Programming

SERVER

```
struct sockaddr_in serv, cli;

sd = socket (AF_INET,
  SOCK_STREAM, 0);

serv.sin_family = AF_INET;
serv.sin_addr.s_addr =
  INADDR_ANY;
serv.sin_port = htons (portno);

bind (sd, &serv, sizeof (serv));
listen (sd, 5);

nsd = accept (sd, &cli, &sizeof
  (cli));
read / write (nsd, ….);
```

CLIENT

```
struct sockaddr_in serv;
sd = socket (AF_INET,
  SOCK_STREM, 0);

serv.sin_family = AF_INET;
serv.sin_addr.s_addr = inet_addr
  ("ser ip");
serv.sin_port = htons (portno);

connect (sd, &server, sizeof
  (server));
read / write (sd, ….);
```

Iterative Vs Concurrent Server

- **One client request at a time.**

```
nsd = accept (sd, &cli,…);
while (1) {
   read/write(nsd, …);
}
```

Many clients requests can be serviced concurrently

```
while (1) {
   nsd =(accept (sd, &cli, ….);
     if (!fork( )) {
           close(sd);
           read/write(nsd, …..);
           exit();
     } else
           close(nsd);
}
```

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.

get **and** ***set*** **timer**

- **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);**

- **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);**

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 */
```

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**

```
[root@localhost ~]# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7892
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7892
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
[root@localhost ~]#
```

Hard and Soft Limits

-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.

- 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
  - 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).

Resource Limitation

- 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"

```
        int getrlimit(int resource, struct rlimit *rlim);
        int setrlimit(int resource, const struct rlimit *rlim);
        int prlimit(pid_t pid, int resource, const struct rlimit *new_limit,
struct rlimit *old_limit);

 struct rlimit {
                rlim_t rlim_cur;  /* Soft limit */
                rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
            };
```

Resource Usage

```
struct rusage {
        struct timeval ru_utime; /* user CPU
time used */
        struct timeval ru_stime; /* system C
time used */
        long   ru_maxrss;        /* maximum
resident set size */
        long   ru_ixrss;         /* integral shared
memory size */
        long   ru_idrss;         /* integral
unshared data size */
        long   ru_isrss;         /* integral
unshared stack size */
        long   ru_minflt;        /* page reclaims
(soft page faults) */
        long   ru_majflt;        /* page faults
(hard page faults) */
        long   ru_nswap;         /* swaps */
        long   ru_inblock;       /* block input
```

int getrusage(int who, struct rusage *usag

getrusage() returns resource usage meas

for who, which can be one of the following

RUSAGE_SELF

RUSAGE_CHILDREN

RUSAGE_THREAD

sysconf - get configuration information at run time:

long sysconf(int name);

Example: ret= sysconf(_SC_CLK_TCK);

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

Multi Threading

Thread is a sequential flow of control through a program.
If a process is defined as a program in execution then a thread is defined as a function in execution.
If a thread is created, it will execute a specified function.
Two type of threading: 1. Single Threading and 2. Multi threading

The created threads within a process share

1. instructions of a process
2. process address space and data
3. open file descriptors
4. Signal Handlers
5. pwd, uid and gid

The created threads maintain it's own

1. thread identification number (tid);
2. pc, sp, set of registers
3. stack
4. priority of the threads
5. scheduling policy

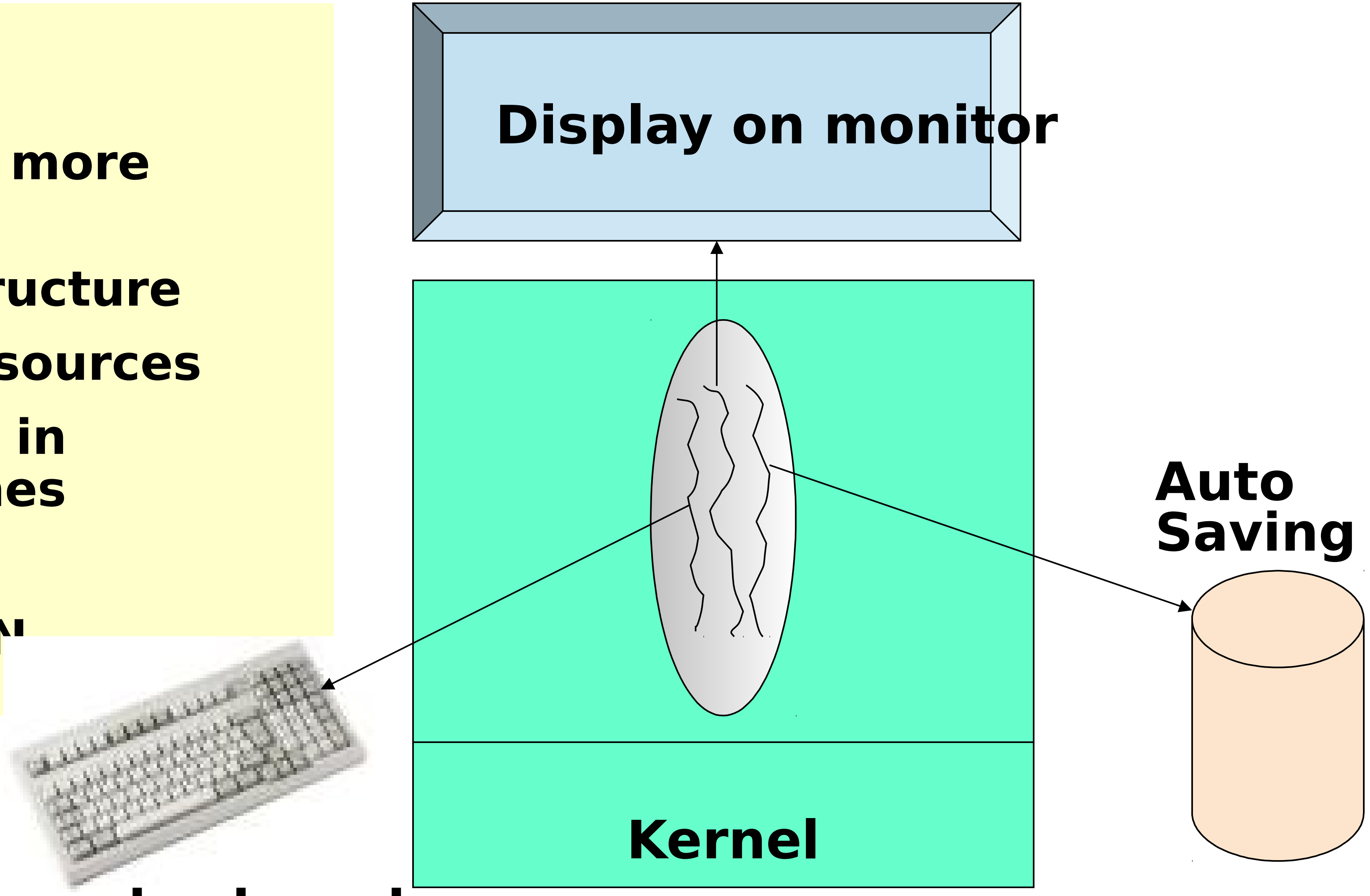
Advantages of Threads:
Takes less time for creation of a new thread, termination of a thread and communication between threads are easier.

Advantages

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

Applications

- **A file server on a LAN**
- **GUI**
- **web applications**



Input from keyboard

Thread Creation

```
#include <pthread.h>
void thread_func(void) { printf(" Thread id is %d", pthread_self()); }
main ( ) {
pthread_t mythread;  pthread_create ( &mythread, NULL, (void *) thread_func, NULL);
 }
```

This needs to be compiled as follows...$gcc pthread.c –lpthread

pthread_t is type-defined as unsigned long int. It takes the thread address as the first argument, the second argument is used to set the attributes for the thread-like stack size, scheduling policy, priority; if NULL is specified, then it takes default values for the attributes.

The third argument is the function that the thread should execute when created. The fourth argument is the argument for the thread function. If that function has a single argument to be passed, we can specify it here. If it has more than one argument, then we have to use a structure and declare all the arguments and pass the address of the structure.

Linux Signals

root@localhost:~

```
[root@localhost ~]# kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
[root@localhost ~]#
```

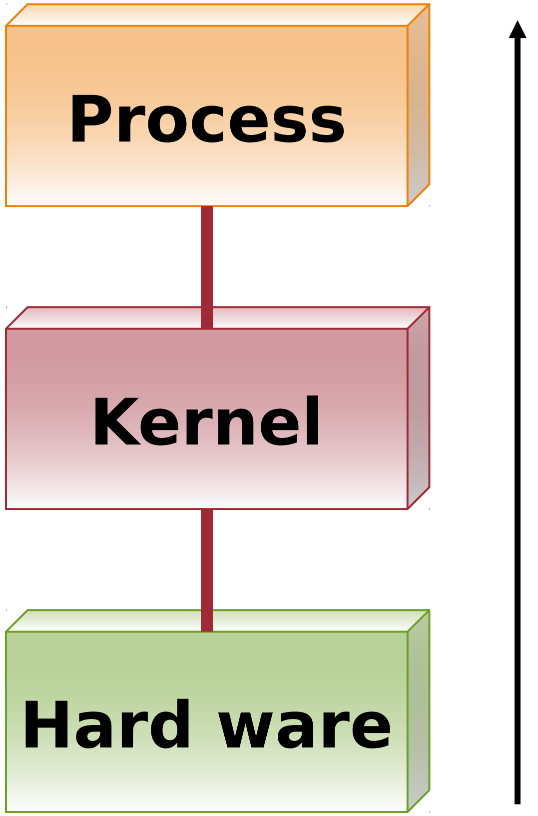
Introduction

- **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.**

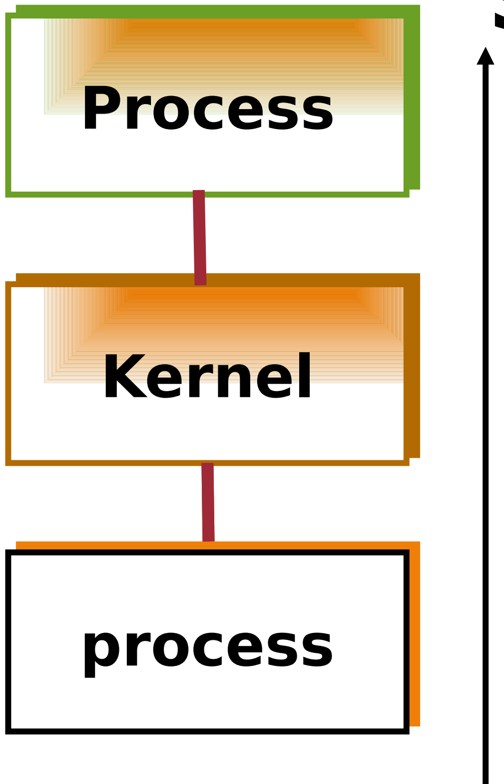
Signal Vs Interrupt

Interrupt

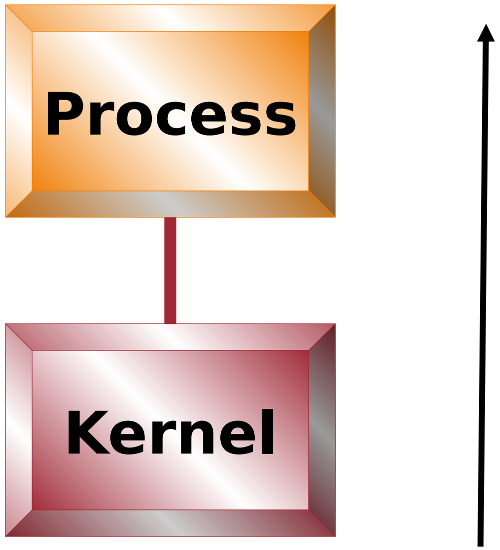


I/O operation (ex: mouse click)

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**

signal System Call

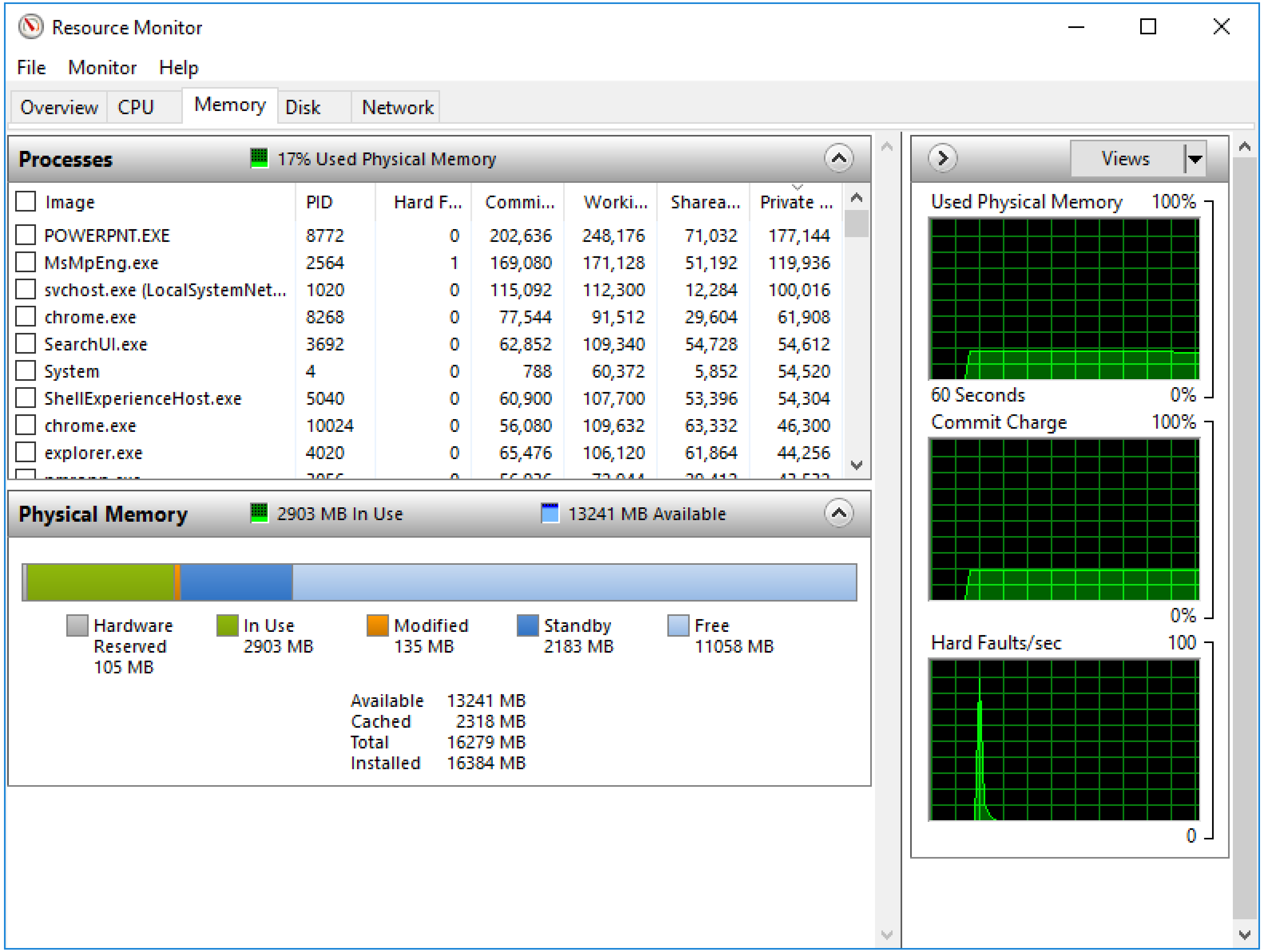
- sigaction ( ) is ame as signal( ) but it has lot of control over a given signal.
- The syntax of sigaction is:

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.

- 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

emory Management



Virtual Memory

- **Memory management: one of the most important kernel subsystems**
- **Virtual Memory: Programmer need not to worry about the size of RAM (Large address space)**
- **Static allocation: internal Fragmentation**
- **Dynamic allocation: External Fragmentation**
- **Avoid Fragmentation: Thrashing -overhead**

- **Large address space: virtual memory is many times larger than the physical memory in a system.**
- **For a 32 bit OS, the virtual memory size will be 2 to the power of 32 i.e 4GB. But the RAM size may be much smaller.**
- **Each process has a separate virtual address space.**
- **Each process space is protected from other processes.**
- **It supports shared virtual memory, i.e more than one process can share a shared page.**
- **Uses paging technique.**

Page Table

- Hard ware support (MMU, TLB) is required.
- Fair share allocation
- static allocation
- Minimize internal fragmentation
- identified by a PFN (Page Frame Number)
- virtual address is split into two parts namely an offset and a virtual page frame number.

- Translate a process virtual address into physical address since processor use only virtual address space
- The size of a page table is normally size of a page
- if it is 4kb, each page address size is 4byte, so 1024 page entries in a page table.
- holds info about
  - Whether valid page table or not?
  - PFN
  - access control information
- Stored in TLB (Translation Look-

Memory Mapping

- **Executable image spilt into equal sized small parts (normally a page size)**
- **Virtual memory assigns virtual address to a each part**
- **Linking of an executable image into a process virtual memory**

Swapping

- **Swap space is in hard disk partition**
- **If a page is waiting for certain event to occur, swap it.**
- **Use physical memory space efficiently**

- **Demand Paging - don't load all the pages of a process into memory**
- **Load only necessary pages initially**
- **if a required page is not found, generate page fault then the page fault handler brings the corresponding page into memory.**

Kernel Data Structure

- **Source Code: /usr/src/linux-4.12/mm**

/usr/src/linux-4.12/include/linux

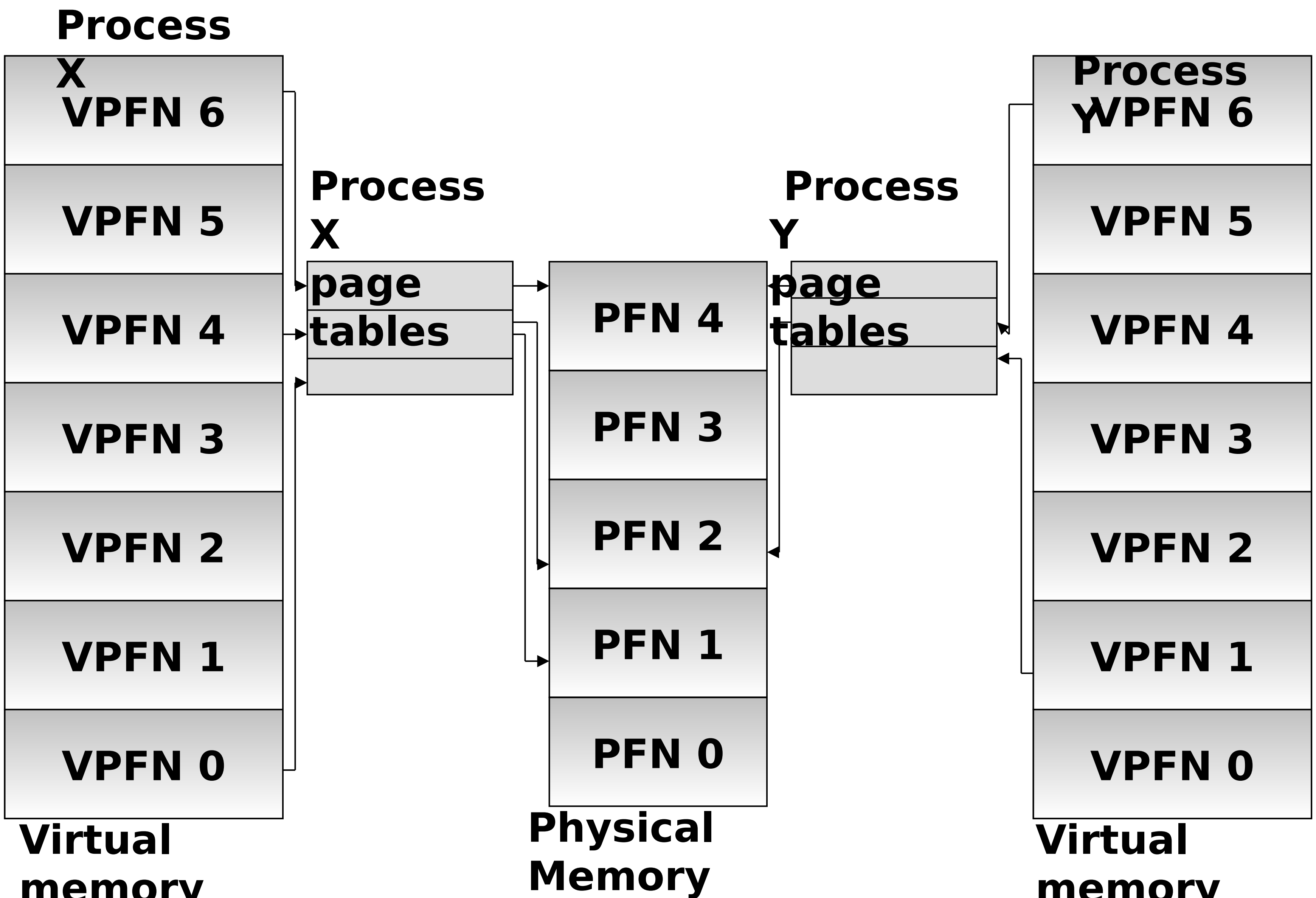
- **virtual memory is represented by an mm_struct data structure**
- **it has pointers to vm_area_struct data structure**
  - **created when an executable image is mapped with the process virtual address**
  - **has starting and end points of virtual memory**
  - **represents a process's image like text, data and stack portion**
  - **has control access info**

mm_types.h

struct **mm_struct** {
struct **vm_area_struct** *mmap; /* list of VMAs */
....

struct **vm_area_struct** {
/* The first cache line has the info for VMA tree walking. */
unsigned long vm_start;
unsigned long vm_end;
/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next, *vm_prev;

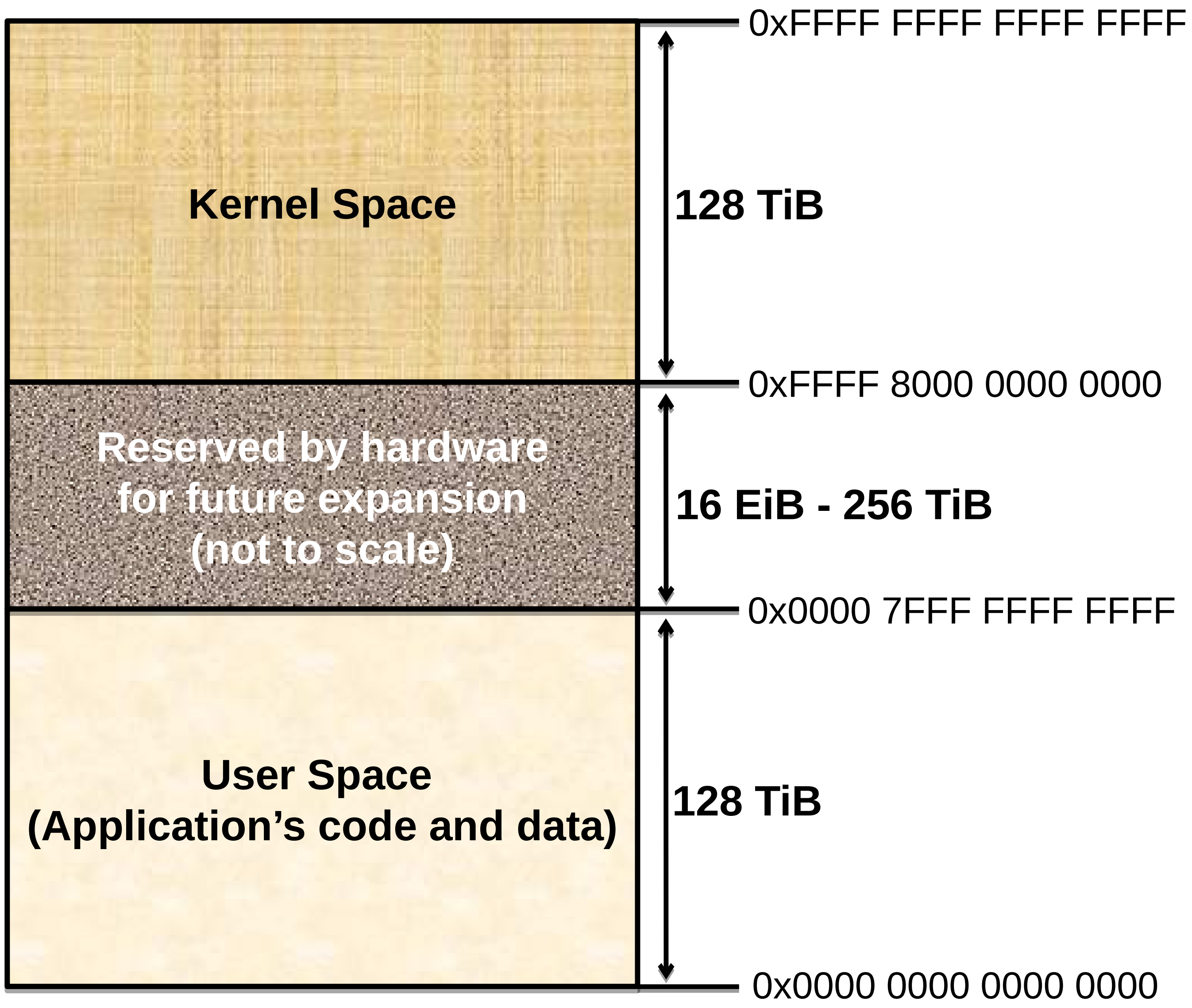
Virtual to Physical Memory Translation



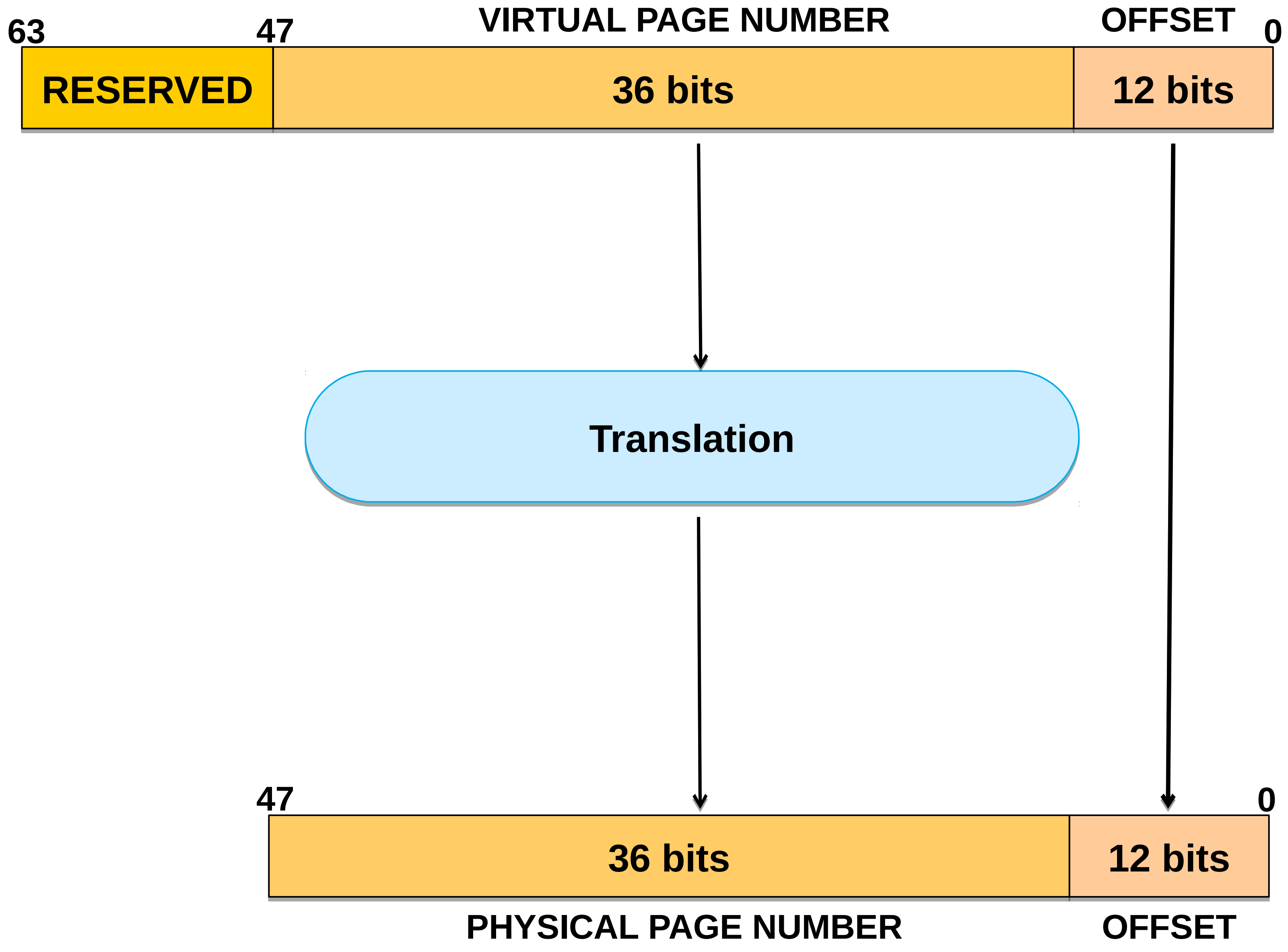
x86-64 Virtual Memory Layout

$pmap –x <pid>

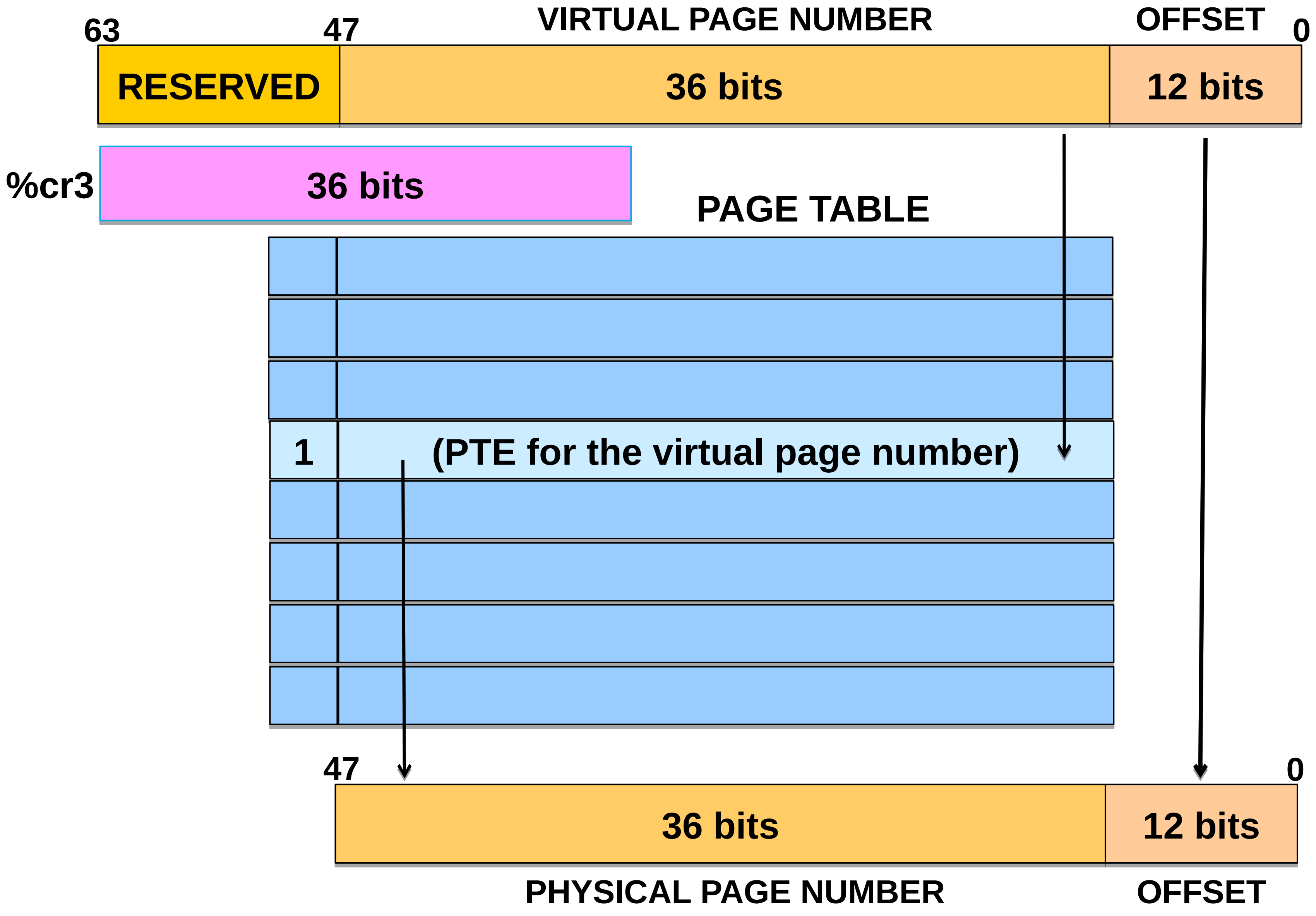
Prefix	Name			
Kibi	binary kilo	1 kibibyte (KiB)	2^{10} bytes	1024 B
Mebi	binary mega	1 Mebibyte (MiB)	2^{20} bytes	1024 KiB
Gibi	binary giga	1 Gibibyte (GiB)	2^{30} bytes	1024 MiB
Tebi	binary tera	1 Tebibyte (TiB)	2^{40} bytes	1024 GiB
Pebi	binary peta	1 Pebibyte (PiB)	2^{50} bytes	1024 TiB
Exbi	binary exa	1 Exbibyte (EiB)	2^{60} bytes	1024 PiB



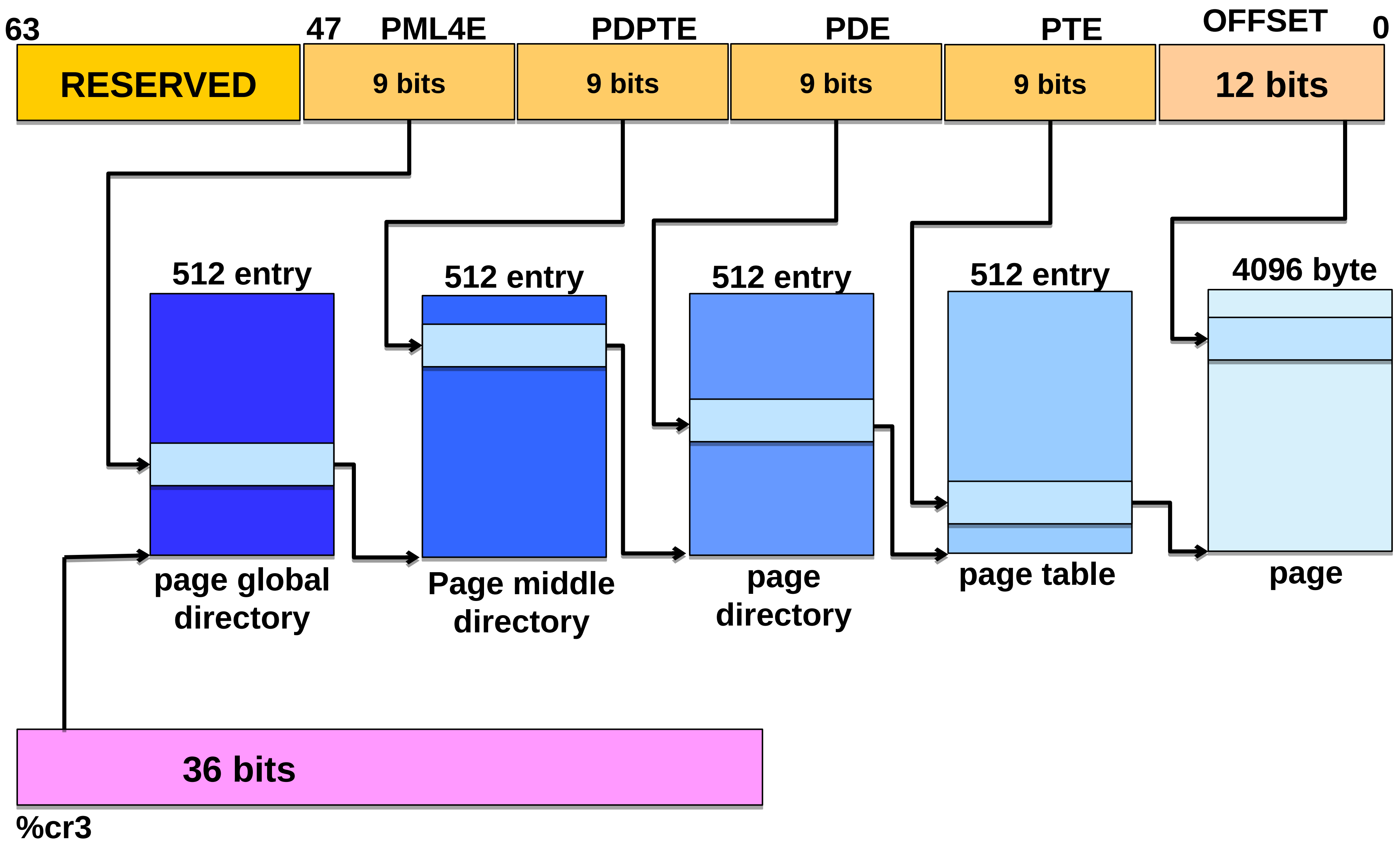
Virtual to Physical Transition



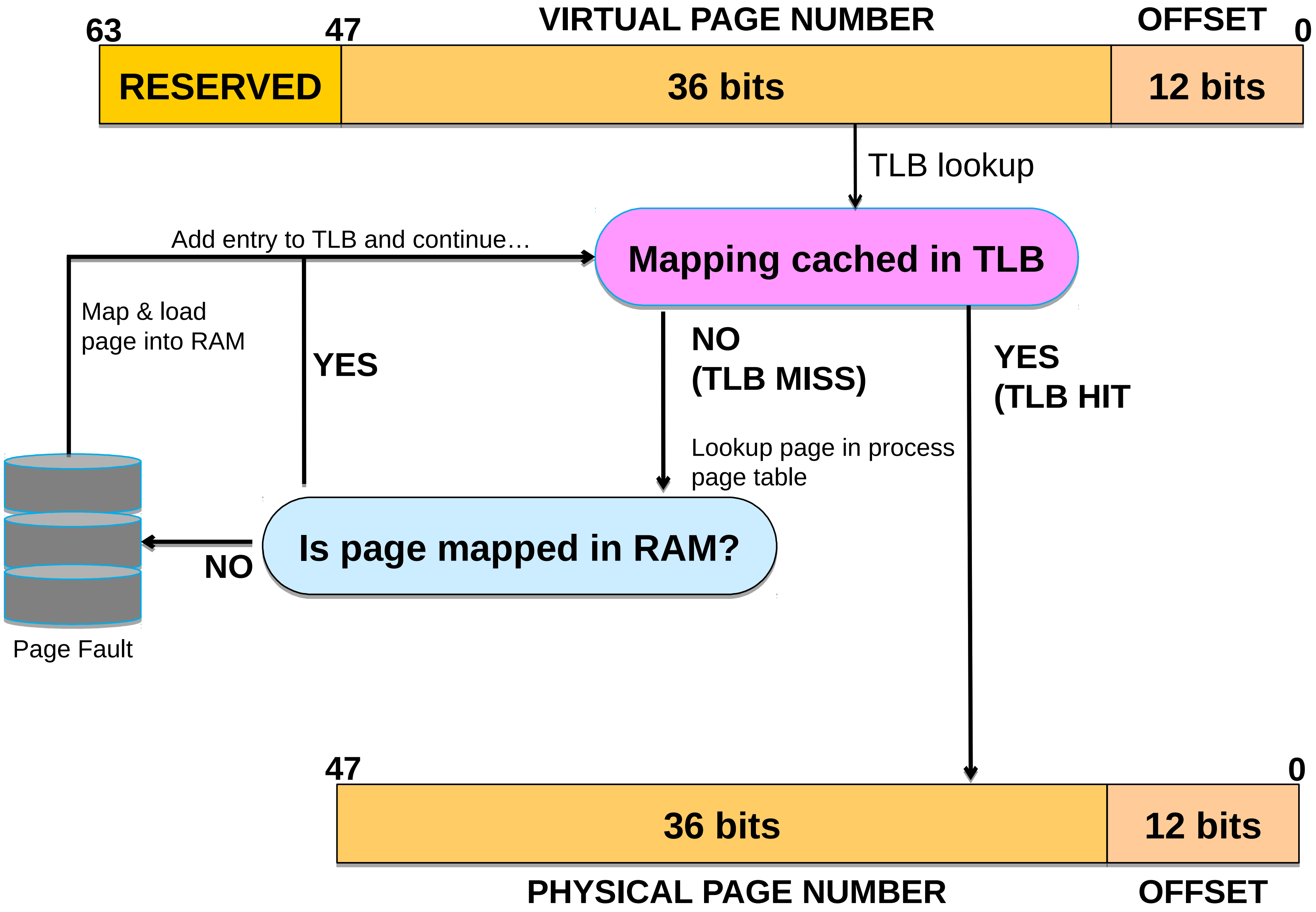
Memory Architecture – Page Table



Page Table Hierarchy on x86-64

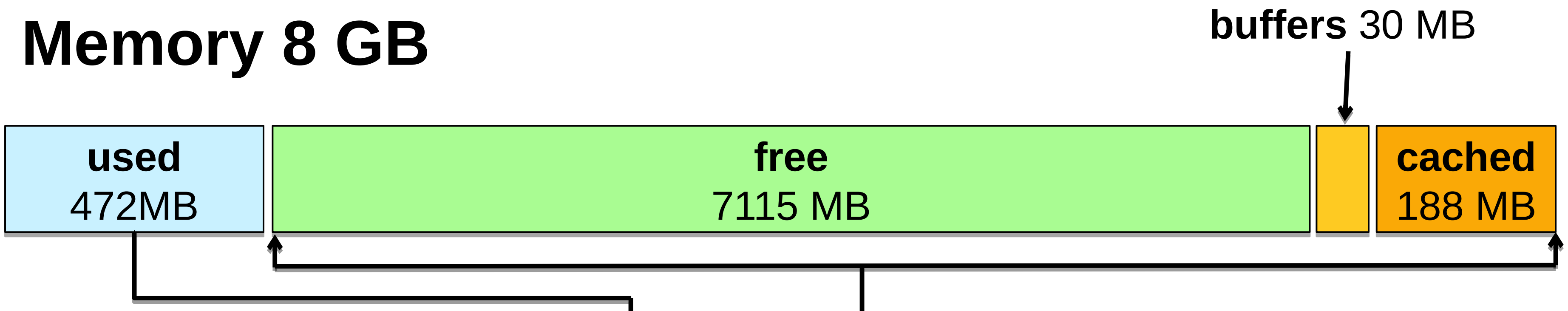


Memory Architecture



$ free -m

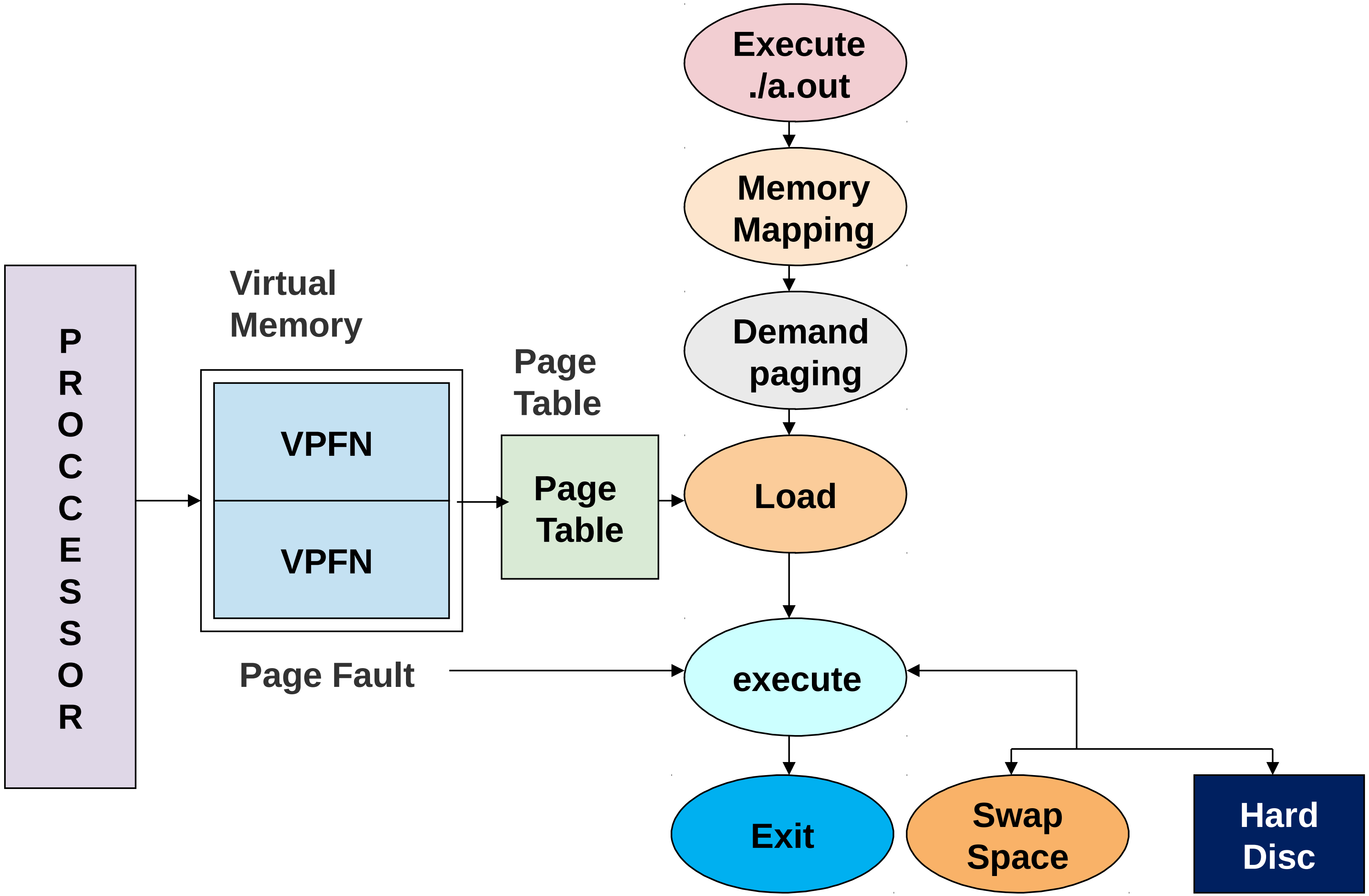
Memory 8 GB



	total	used	free	shared	buffers	cached
Mem:	7806	691	7115	0	30	188
-/+ buffers/cache:		472	7334			
Swap:	1999	0	1999			

Mem: used = 472 + 30 + 188 = 690

Walk Through Program Execution



Thank You

Overview
of
LINUX Device Drivers

Agenda

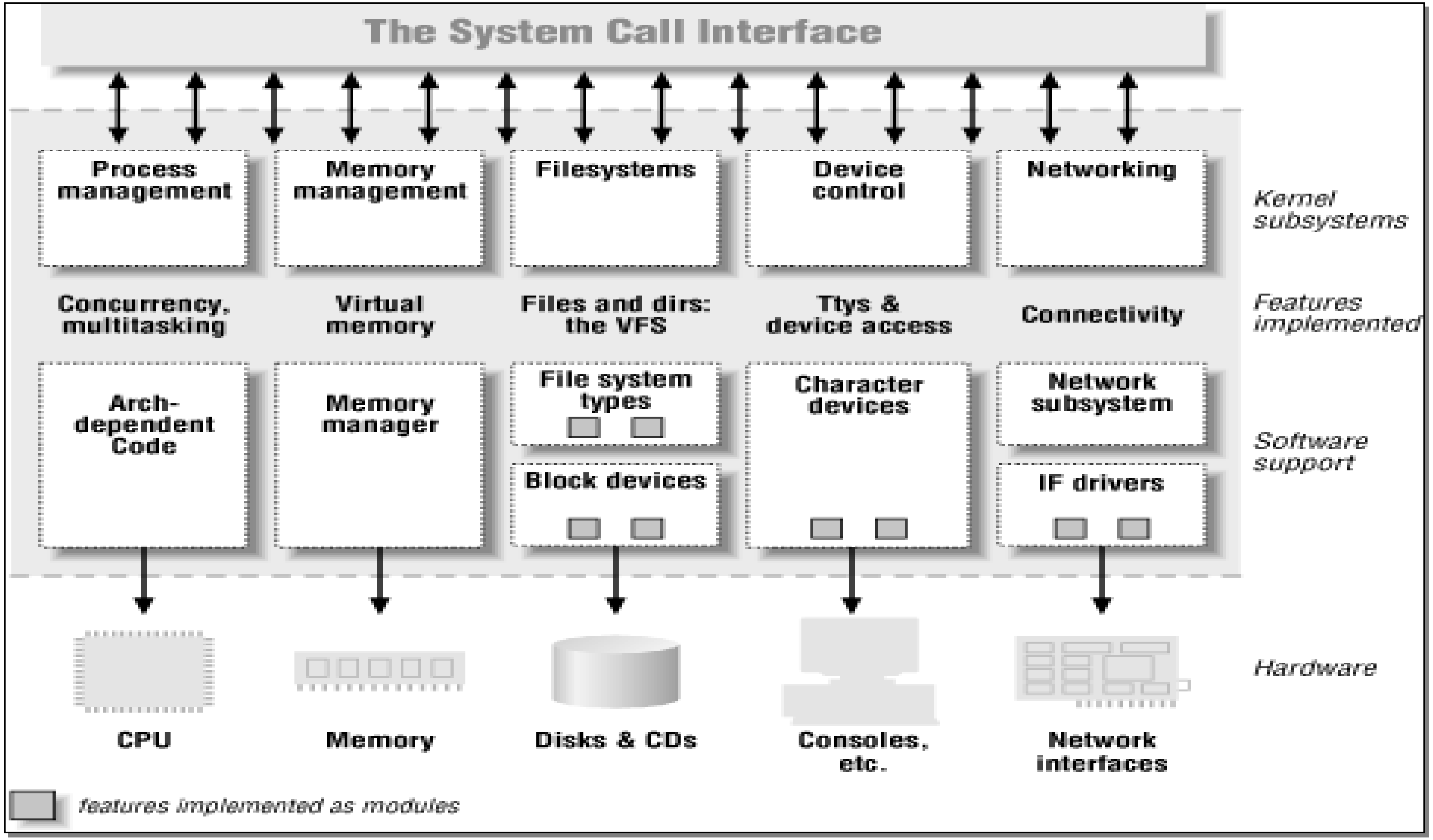
Introduction to Device Drivers

- Brief Introduction of Linux OS and Its Internals
- Classes of Devices
- Device Driver Classifications
- Kernel Modules Vs Applications
- Major and Minor numbers
- Device Driver Entry Points
- Developing and Compiling a Simple Module
- Loading and Removing a Module

Character Device Driver

- I/O Memory Allocation
- File Structure
- File Operations Structure
- Transfer of data to / from Device Driver
- Design of a Simple Character Device Driver

Linux Kernel Architecture



Device Driver Introduction

A device driver is a collection of functions that accept general requests for I/O operations and manipulate the device to perform the requested operations.

Under UNIX every device is managed by device driver functions.

The group of functions is compiled and attached as part of the UNIX kernel.

Device Driver Introduction

Device Drivers are the translators of the unix operating system kernel. They stand between the kernel and peripherals, translating requests for work into activity by the hardware.

A device driver is a “C’ program that controls a device.

Drivers are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface.

Device Driver Classifications

Statically linked driver: whose object code is linked with the kernel. The code of such device driver is physically contained in the kernel and therefore loaded in memory when the system boots.

Dynamically linked driver: whose object code is NOT linked with the kernel. The code of such device driver is NOT contained in the kernel, and the device driver is loaded and unloaded as and when required.

Dynamically Linked Driver

The device driver programming interface is such that drivers can be built separately from the rest of the kernel, and "plugged in" at runtime when needed.

Each piece of code that can be added to the kernel at runtime is called a module. For example: device drivers, file systems.

Each module is made up of object code that can be dynamically linked to the running kernel or removed from the running kernel by the appropriate command.

Fundamental Concept

Write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs.

The driver should deal with making the hardware available, leaving all the issues about how to use the hardware to the applications.

The kernel is non preemptive. Kernel functions do not get context switched. Thus, you need not do lots of messy locking on variables.

But, if one of the driver routines goes into an infinite loop, the machine will hang. If the routine takes too long, the machine will appear to pause.

None of the libc functions are available, but some of them are duplicated.

Kernel Modules Vs Applications

Applications performs a single task from beginning to end, A module registers itself in order to serve future requests.

Application call functions resolves external references using appropriate lib functions during link stage, whereas modules are linked only to the kernel, and only functions it can call are exported by kernel.

Since no library is linked to modules, source file should never include usual header files.

Segmentation fault is harmless during application development but kernel fault is fatal at least to the current process, if not for the whole system.

Kernel Modules Vs Applications

User activities are performed by means of a set of standardized calls that are independent of the specific driver.

Mapping those calls to device-specific operations that act on real hardware is then the role of the device driver.

However, unlike a C program, you do not link a device driver into an executable program as it does not have a main() function - meaning that a device driver does not have a single entry point.

Namespace Pollution

Kernel programmer must be aware of and avoid namespace pollution.

The programmer is to find unique names for new symbols.

Namespace collisions can lead module loading failures.

The best approach for preventing namespace pollution is to declare all your global variables as *static.*

Major and Minor Numbers

A driver never actually knows the name of the device being opened, just the device number—and users can play on this indifference to names by aliasing new names to a single device for their own convenience.

If you create two special files with the same major/minor pair, the devices are one and the same, and there is no way to differentiate between them.

Major and Minor Numbers

The major number identifies the driver associated with the device. It is a small integer that serves as the index into a static array of char drivers.

The kernel uses the major number at *open* time to dispatch execution to the appropriate driver.

The minor number is used only by the driver specified by the major number. It is common for a driver to control several devices, the minor number provides a way for the driver to differentiate among them.

Major and Minor Numbers

```
$ ls -l /dev/hda*
brw-rw----    1 root    disk      3,   0 Mar 24  2001 /dev/hda
brw-rw----    1 root    disk      3,   1 Mar 24  2001 /dev/hda1
brw-rw----    1 root    disk      3,   2 Mar 24  2001 /dev/hda2
```

```
$ ls –l /dev/lp*
crw-rw----    1 root     lp       6,   0 Mar 24  2001 lp0
crw-rw----    1 root     lp       6,   1 Mar 24  2001 lp1
crw-rw----    1 root     lp       6,   2 Mar 24  2001 lp2
```

Current Process Information

#include <linux/sched.h>

One of the most important header files.

struct task_struct *current;

The current process.

for ex: current - > pid; current - > comm; etc,.

The process ID and command name for the current process.

MODULE_AUTHOR (“ Linus Torvald ”);
Puts the author’s name into the object file.

MODULE_DESCRIPTION (“simple character device driver”);
Puts a description of the module into the object file.

$modinfo –a module.o : To check author name.

$modinfo –d module.o : To check module description.

Dynamic Memory Allocation

- obtains a memory area using *kmalloc* and releases it using *k*free.

  - behave like *malloc* except takes an additional argument, the priority.

  - usually, a flag of GFP_KERNEL or GFP_ATOMIC or GFP_USER.

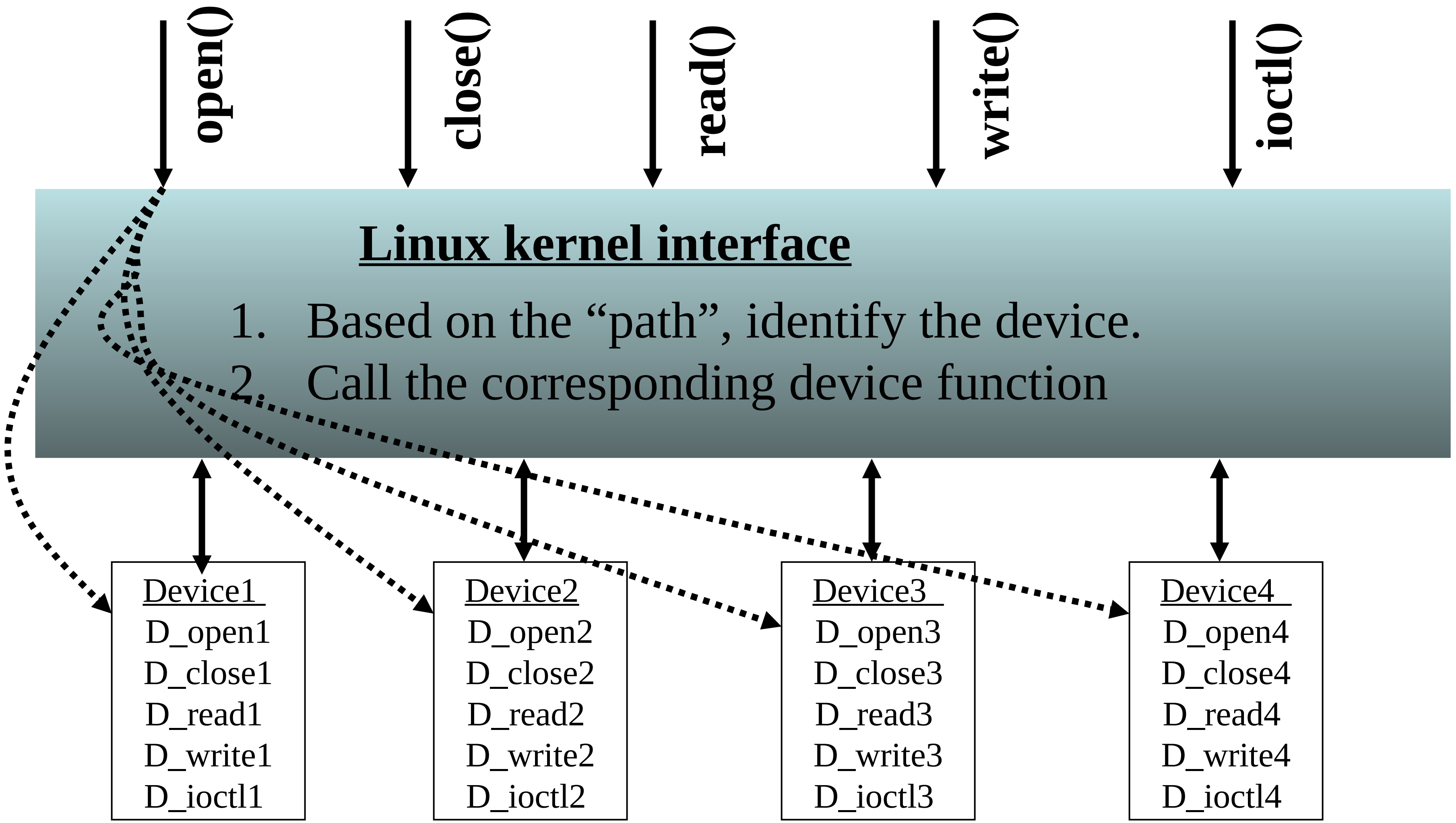
```
void *kmalloc (unsigned int size, int priority);
void kfree (void *obj);
```

Driver Entry Points

A device driver has a collection of functions known as entry points. Some of the entry points are :

open()
release()
llseek()
read() (used for character device drivers only)
write() (used for character device drivers only)
strategy() (used for block device drivers only)
ioctl ()
poll()

Input-Output Interface



Developing and Compiling a Simple Module

```
#include <linux/kernel.h>
int my_init (void)
  {
     printk ("Module Insertion Successful\n");
     return 0;
  }

void my_cleanup (void)
 {
      printk ("Module unloading Successful \n");
 }
module_init (my_init);
module_exit (my_cleanup);
```

Need root permission to load or remove the module.

$insmod ./char.o

Module Insertion Successful.

$rmmod char

Module unloading Successful.

Output of printk is shown because, after loading the module it can link to the kernel and can access the kernel's public symbols (functions and variables).

$lsmod

whether the module is loaded or not can be checked by

$cat /proc/modules (or) lsmod

Module	Size	No.of User
nfsd	69696	8
------	------	---
usbcore	49664	1

$cat /proc/devices

Character devices:

1 mem

----

10 misc

Block devices:

1 ramdisk

2 fd

Switch Table

The Unix kernel maintains a set of data structures known as : Switch Table

These data structures are used to locate and invoke the entry points of a device driver.

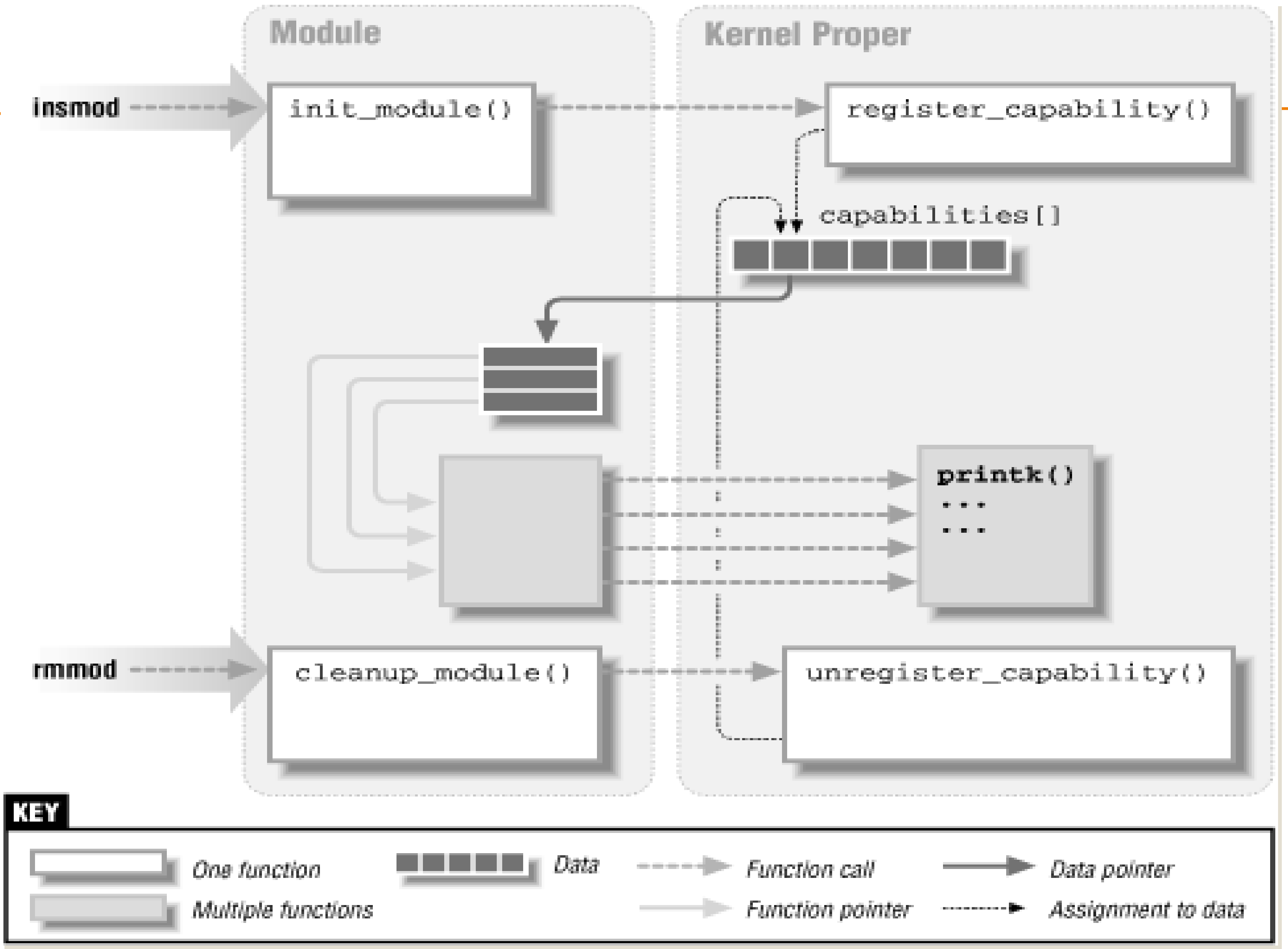
Each switch table is an array of structures. Each structure contains a number of function pointers.

Locating Driver Entry Points

To decide which element of the switch table should be used, the system uses the "major device number".

Then the system decides which entry point of the element should be used.

This depends upon the system call used by the process. If the process used "open" system call, then the "*open*" entry point is used and so on...


Module
Kernel Proper
insmod
init_module()
register_capability()
capabilities[]
printk()
...
...
rmmod
cleanup_module()
unregister_capability()
KEY
One function
Multiple functions
Data
Function call
Function pointer
Data pointer
Assignment to data

Device Registration

The init ( ) is called when the device driver is loaded in memory.

Registers other entry points of the driver with the kernel.

Registers the resources like major number, IRQ, IO ports etc., which are needed by the driver with the kernel.

```
int init_module ( )   {
    register device;
    }
    int register_chrdev (unsigned int major, const char
*name, struct file_operations *fops);
```

Some of the numbers such as 60 – 63, 120 - 127 and 240 – 254 are not used for the standard devices,

cleanup_module ( )

```
void cleanup_module ( )  {
   unregister the device and release the major number;
}
int unregister_chrdev (unsigned int major, const char
   *name);
```

struct file, defined in <linux/fs.h>, is the most important data structure used in device drivers.

The file structure represents an *open file*. It is not specific to device drivers, every open file in the system has an associated struct file in kernel space.

File Structure

It is created by the kernel on *open* and is passed to any function that operates on the file, until the last *close*. After all instances of the file are closed, the kernel releases the data structure.

```
struct file {
        struct file_operations      *f_op;
        unsigned int                f_flags;
        mode_t                      f_mode;
        loff_t                      f_pos;
        void                        *private_data;
        ……………………………………..
};
```

File Structure

mode_t f_mode

The file mode identifies the file as either readable or writable (or both).

loff_t f_pos

The current reading or writing position.

unsigned int f_flags

A driver needs to check the flag for non blocking operation. The flags are defined in the header <linux/fcntl.h>.

File Structure

struct file_operations *f_op

The operations associated with the file.

void *private_data

The driver is free to make its own use of the field or to ignore it. The driver can use the field to point to allocated data, but then must free memory in the *release* method.

file_operations Structure

An open device is identified internally by a file structure, and the kernel uses the file_operations structure to access the driver's functions. The structure, defined in <linux/fs.h>, is an array of function pointers.

Each file is associated with its own set of functions The operations are mostly in charge of implementing the system calls such as *open*, *read*, llseek, and so on.

The file_operations structure has been slowly getting bigger as new functionality is added to the kernel.

file_operations Structure

```
struct file_operations fops = {
    lseek,
    read,
    write,
    readdir
     select / poll
    ioctl
    open,
    flush
    release
};
```

file_operations Structure

The following list shows what operations appear in struct file_operations for the 2.4 series of kernels. The return value of each operation is 0 for success or a negative error code to signal an error.

loff_t (*llseek) (struct file *, loff_t, int);

The llseek method is used to change the current read/write position in a file. The loff_t is a "long offset".

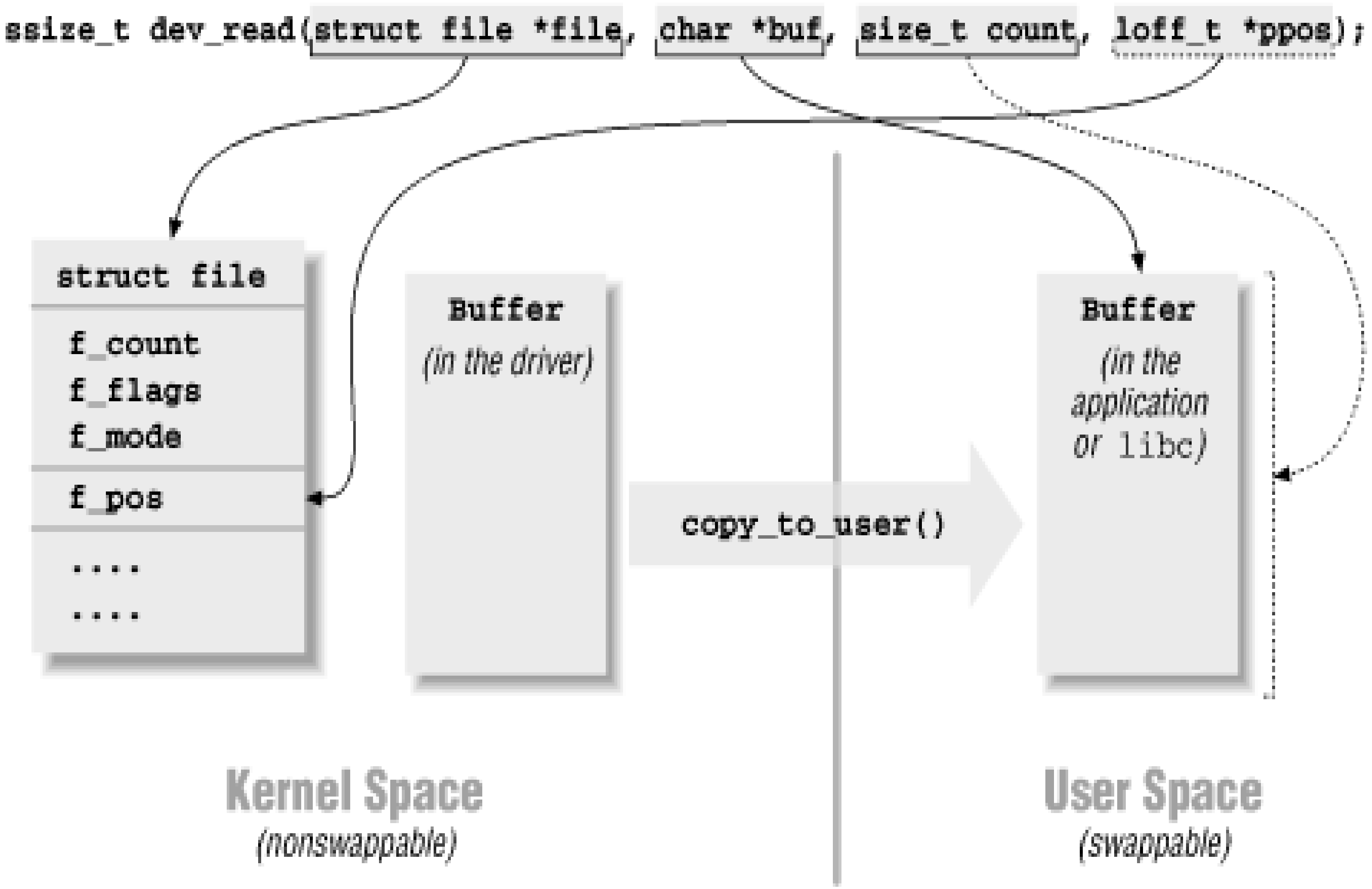
ssize_t (*read) (struct file *, char *, size_t, loff_t *);

used to retrieve data from the device. A non-negative return value represents the number of bytes successfully read.

ssize_t (*write) (struct file *, const char *, size_t, loff_t *);

sends data to the device. The return value, if non-negative, represents the number of bytes successfully written.

read for example



file_operations Structure

int (*readdir) (struct file *, void *, filldir_t);

This field should be NULL for device files; it is used for reading directories, and is only useful to file systems.

unsigned int (*poll) (struct file *, struct poll_table_struct *);

The *poll* method is the back end of two system calls, *poll* and *select*, both used to inquire if a device is readable or writable. Either system call can block until a device becomes readable or writable.

int (*ioctl) (struct inode *, struct file *, unsigned int cmd, unsigned long arg);

The *ioctl* system call offers a way to issue device-specific commands (like Formatting a track of a floppy disk, which is neither reading nor writing).

file_operations Structure

int (*mmap) (struct file *, struct vm_area_struct *);

mmap is used to request a mapping of device memory to a process's address space.

int (*open) (struct inode *, struct file *);

When a device file is open, this entry point is called.

int (*flush) (struct file *);

The _flush_ operation is invoked when a process closes its copy of a file descriptor for a device; Currently, _flush_ is used only in the network file system (NFS) code.

int (*release) (struct inode *, struct file *);

This operation is invoked when the file structure is being released. Like _open_, _release_ can be missing.*

Driver Kernel Communication

Any device operation is typically initiated by a process.

For example a process might use the read() system call to read 10 bytes for a file opened earlier, and store these 10 bytes in a local buffer.

The read() system call now finds out which device is associated with this request, the type of the device and the major device number associated with the device.

The read() system call then invokes the read() entry point of the device and supplies the address of the user (process) buffer to the device driver.

However, the user buffer is located in the process address space. Whereas the device driver executes in the kernel address space.

Transfer of Data To/From Drivers

To copy from user buffer to kernel buffer vice versa can be done by the following functions.

unsigned long copy_to_user(void *to, const void *from, unsigned long count);

unsigned long copy_from_user(void *to, const void *from, unsigned long count);

Device File Creation

The command to create a device node on a filesystem is **mknod;**

Superuser privileges are required for this operation. The command takes three arguments in addition to the name of the file being created. For example, the command

mknod /dev/dev1 c 254 0

creates a char device (c) whose major number is 254 and whose minor number is 0.

Note that once created by mknod, the special device file remains unless it is explicitly deleted.