Operating Systems and UNIX

Pushpak Bhattacharyya
Computer Science and Engineering Department
IIT Bombay.

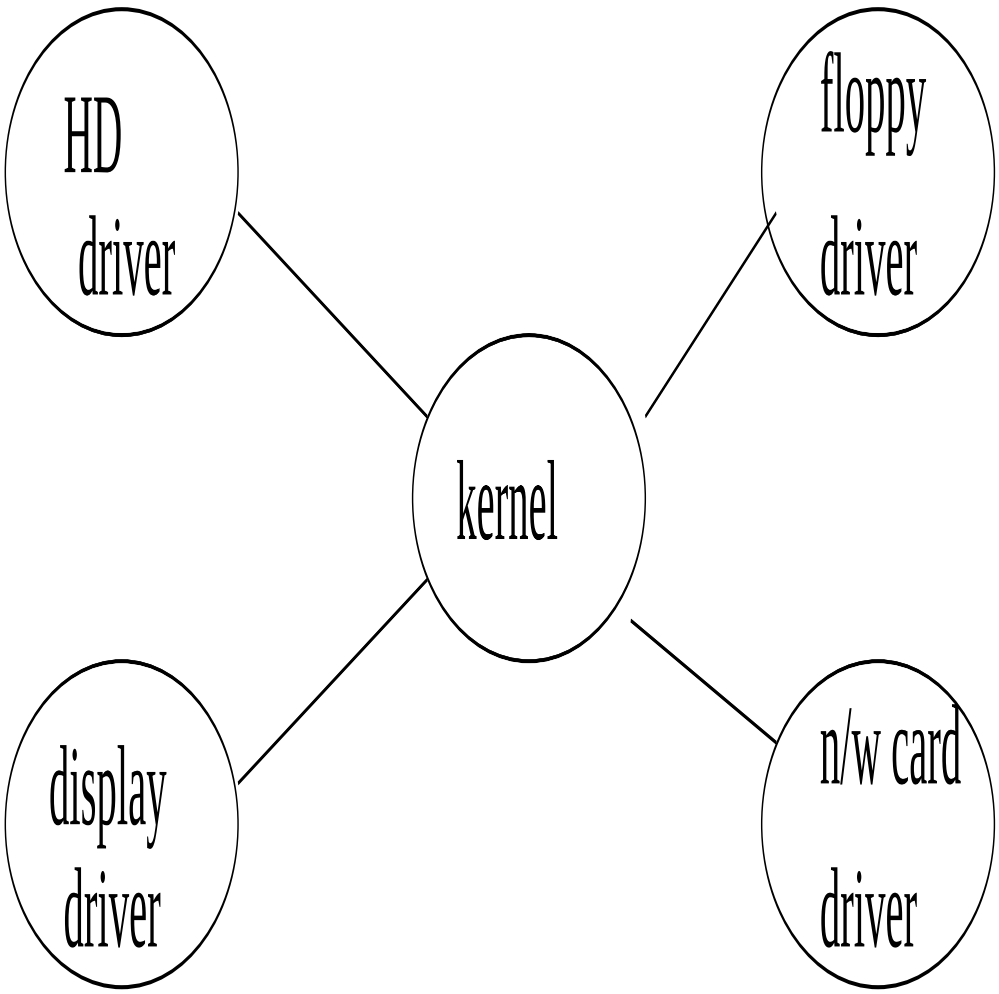
- Topics to be covered
  - Process management and UNIX.
  - Concurrency issues and UNIX.
  - Memory management and UNIX.
  - Devices and UNIX.
- Basic Assumption :
  - The model of the machine is Von Neumann.
  - single processor, single storage seperate from each other.

Significant Developments:

- Single dedicated system.
- Resident moniter.
- Multiprogramming
  arrange for
  1. mutual protection between programmes.
  2. allocation of sufficient memory.
- Time sharing: allocating CPU to users for fixed amounts of time

Process Management and UNIX

Devices and the OS



Device drivers provides *calls* or *functions* which enable kernel to obtain attention of the device drivers.

OS view of Code

OS is concerned with process.

A process is a programme in execution.

When a process executes, *system code*, *library code* and *user code* get executed.

Example

```
main ()
{
 int a,b,c; char arr[100];
 a=myfunction (i,j); //user function
 b=sqrt(a); //library function
 c=write(fd, arr, 10); // system function
}
```

- System code executes at a higher level of previledge. The processor switches to protected mode.
- Library code execution is through
  - complie time linking
  - dynamic linking
- User code executes through user context.

System Call Interface

```
Algorithm syscall
Input:- System call no.
Output:-result of system call
{
   Find entry in the system call table
         corresponding to system call no.;
   Determine no. of parameters to syscall;
   Copy parameters from user address
         space to u_area;
   Save current context for abortive
         return;
   Invoke sys call code in kernel;
   If (error during execution of sys call){
      Set register 0 in user saved register
         context to error no;
      Turn on carry bit in ps reg in user
         saved reg context;
   }
   else return successfully by placing
         value in reg0 and 1.
}
```

Sys Call Inteface

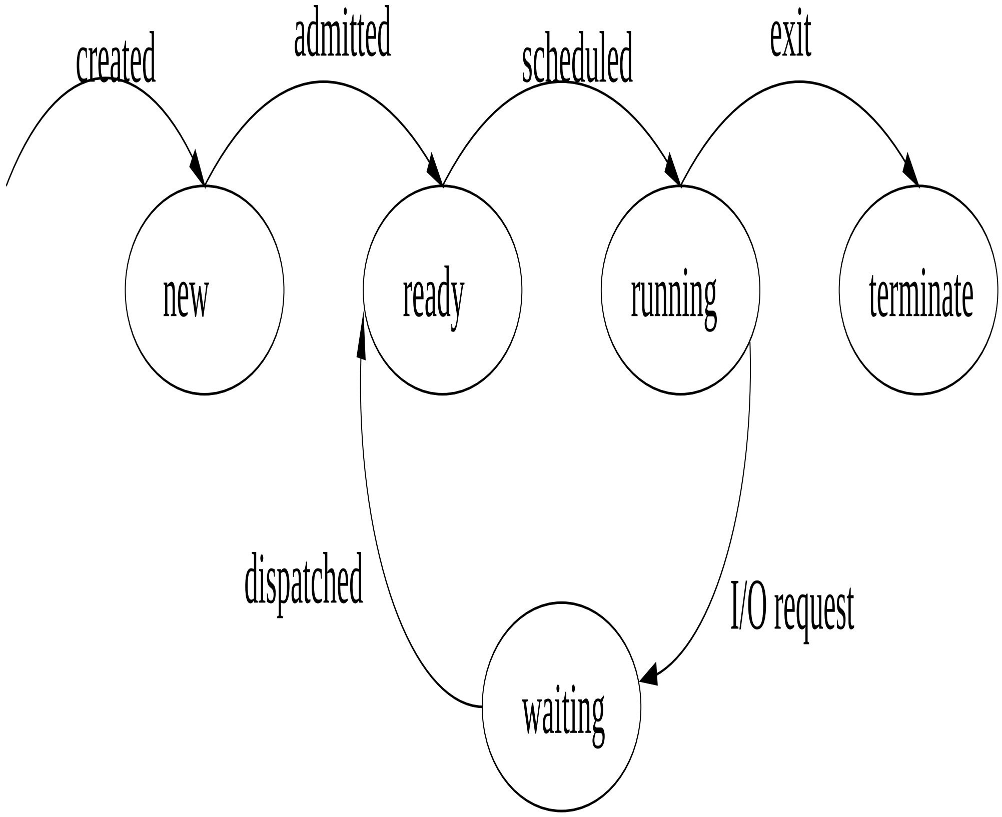
```
Char name[] = "file";
main()
{
  int fd;
  fd = creat(name,0666);
}

Portions of generated M68k Assembler code
# code for main
58: mov & 0x116, (%sp) # push 0666
5e: mov & ox 204,-(%sp) # push name
64: jsr 0x7a # call "Create"

# library code for creat
7a : movq & 0x8,%d 0 # move 8 to R0
7c: trap &0 x0 # operating sys. trap
7e: bcc &0x6 <86> # branch to
          addr. 86 if carry sit clear
80: jmp 0x13c # jump to addr 13c
86: rts       # return from subroutine
```

Things happening to a process

- Creation
- Deletion
- Active
- Suspended



PROCESS STATE DIAGRAM

Exercise:

- How to incorporate interrupt processing in the state diagram.

Memory Management

- Things happening to memory
  - Allocation
  - Deallocation
  - Sharing
  - Protection
- Allocation
  - First fit
  - Best fit

- Allocation: Maintain free list and busy list.
- Deallocation: Shift from busy list to free list.
- Sharing: Share code pages and Make the code re-entrant.
- Protection: Indicate by read write bit on each page.

Files

A File is an abstraction of related data residing on the secondary storage.

Operations on files are:

- Creation
- Deletion
- Updation
- Sharing
- Protection

Files formats: *.exe, .obj, .rts, .doc, .txt, .gif,* etc.

file formats interpreted by the system.

The basic underlying information is in terms of bits and bytes.

```
main ()
{ int fd
  fd = open ("/home/ram/abc",RD-only)
}
```

Global file table for system in wide open files.

Inode table in the hard disk. Each *Inode* contains permission, owner, creation, time, modification time, addresses of data stocks etc.

For large files indirection of addresses.

Deletion of file

- Deletion of data blocks
- Delection of Inode

Sequence is an important question for crash recovery.

Inter process Communication

In UNIX, processes communicate by

- Signal and kill
- Shared memory
- Message Passing

For shared resources critical section. Consider the Railway Reservation program:

```
if (count<72) {
  seat[count]= allotted;
  count++,
}
```

Command Interpretation

Provides facilities like

- Background execution (&)
- Input output redirection (<)
- Piping (c1|c2)

Basic pseudocode of CI:

```
While (1) {
  get command ();
  check command ();
  if (fork() ==0){
       /* command code */
  }
  else {
       Interpret command();
  }
}
```

fork system call returns 0 to the child and process-id to the parent.

How many process are created by

```
for (i=0; i<3; i++)
        fork ();
```

i=0

parent, child1

i=1

parent, child1, child2, child11

i=2

parent, child1, child2, child3, child11, child12, child21, child111

To create just 3 processes

```
for (i=0; i<3; i++)
     if (fork () ==0) break;
```

UNIX SHELL internal

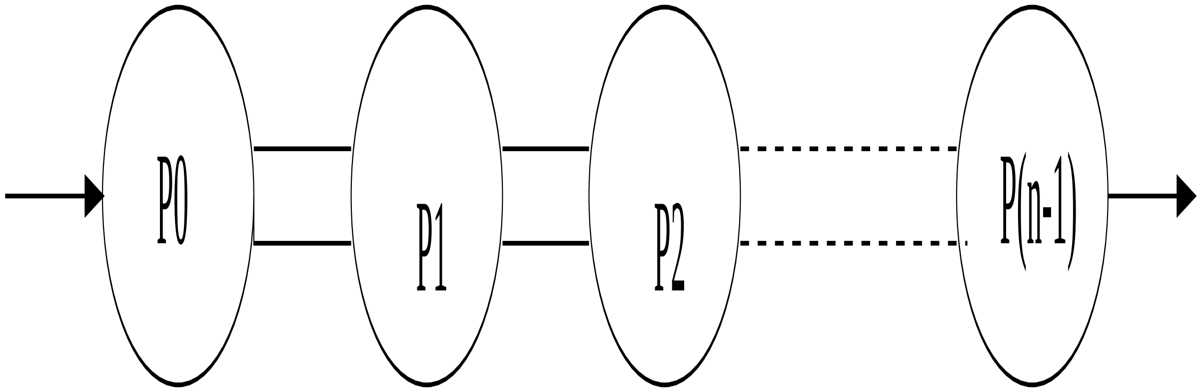
SHELL Code:

```
/* Read command line until EOF*/

   while (! EOF)\{
      /* command line contains & */
      if (/* command line contains & */
             amper =1;
      else   amper = 0;
      /* command execution arrangement */
      if (fork() = = 0){
          if (/* redirect output */){
             fd = create(newfile, fmask);
             close (stdout);
             dup(fd);
             close(fd);
             /*stdout is now redirected*/
      }
      if (/*piping */){
             pipe (fildes);
             if (fork() = = 0){
             {/*first component of command line*/
```

```
            close (stdout);
            dup(fildes[1]);
            close(fildes[1]);
            close(flides[0]);
            /* stdout goes to pipe */
            /*child process does command 1*/
            exec (command1)
      }
      /* 2nd command component of command line */
      close(stdin);
      dup(fildes[0])
      close(fildes[0])
      close(fildes[1])
      /* std input comes from pipe*/
   }
   exec (command2);
 }
/*parent continues; waits for child to exit*/
if (amper = = 0)
     retid = wait(&status);
}
```

An assignment on process creation and synchronisation



The task is to send an integer to the pipeline of processes. Each processes increments it by 1. The final result comes out as the initial value incremented by number of processes. The processes are ceated by the parent who does nothing else. The only process to interact with the user is the first child and the only process to output to the screen is the last child.

Process Schedulng

To increase the degree of multiprogramming as much as possible.

Multiprogramming is simultaneous residence of programs in the main memory.

Scheduler is a piece of code executed by the CPU to decide whom it should give itself to.

Scheduling algorithm should run very efficiently for good system performance.

Scheduling Data Structures

- **Process control block (PCB)**
  - Process State: Ready, running etc.
  - Program Counter: Next instruction to be executed
  - CPU registers
  - Scheduling Information: Time elapsed, time required, Priority etc.
  - Accounting info: Resources Used
  - Memory Info: Address and size of blocks used.
  - I/O Status: Info on files and devices operated.
- **Various Queues** like *Ready Queue* and *Device Queues.*

Exercise: Enumerate all possible changes in the PCB where a process goes from ready state to wait state and vice versa.

Various Concerns in Scheduling

CPU bound job - Spends more time on CPU.

I/O bound job - Spends more time on I/O.

Context switch time is 1 to 1000 microsecond.

CPU and I/O burst alternate.

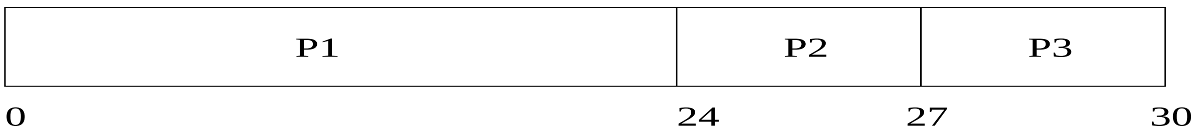
Two kinds of scheduling: premptive and non-preemptive scheduling.

Scheduling Algorithms

Scheduling Criteria are: *CPU utilization*, *Throughput*, *Turn Around Time*, *Wait Time* and *Response time.*

Scheduling Algorithms:
FCFS

Process	Burst time
p1	24
p2	3
p3	3



GANTT CHART

Other Algorithms

- Pre-emptive Shortest job first
- Non Pre-emptive Shortest job first or Shortest Remaining Time
- Priority Scheduling
- Round Robin

Predict time required with
$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.

Process	Arrival time	Burst time
p1	0	8
p2	1	4
p3	2	9
p4	3	5

Exercise

1. Run the following program from the terminal then redirect the output to a file. Explain the output

```
main ()
{
printf ("hello\n");
if (fork()= = 0)
    printf ("world\^n")
}
```

2. The *tee* command is used as follows (an example):
   tee a b c
   The command copies the standard input to standard output and also to the files a,b,c. Give a piece of c-code which implements this operation.

Round Robin Scheduling

FCFS + pre emption = Round Robin

Time quantum, q used for time sharing. H/W support required in the form of timer interrupt.

```
   Process : p1   p2   p3
 Burst Time: 24    3    3
             q= 4
```

Time quantum should not be too small, nor too large.

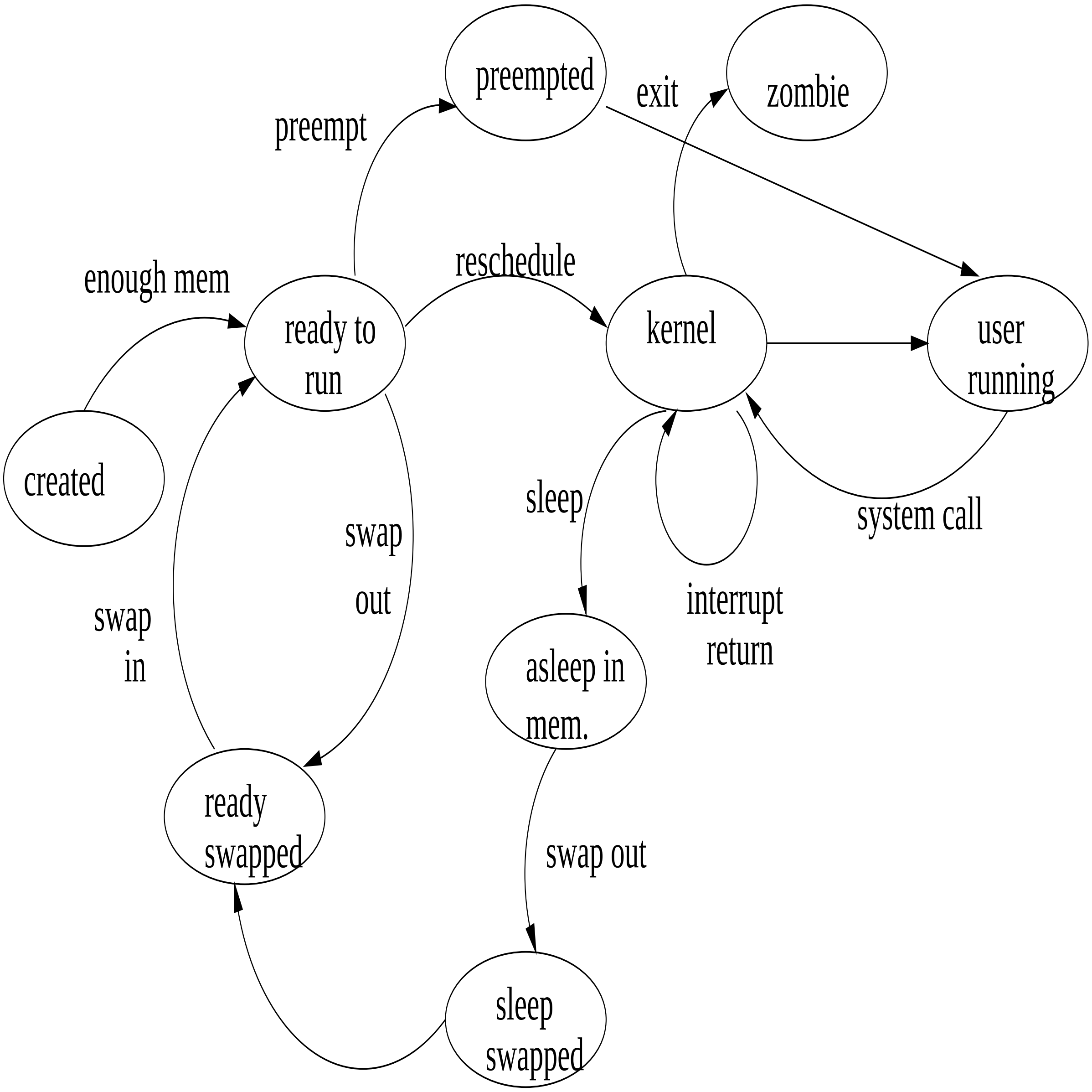
Multilevel Queue Scheduling

- System process
- Interactive Process
- Batch Process
- Student Process

Priority decreasing from top to bottom.

Multilevel queue scheduling with feedback is used in UNIX.

UNIX process state diagram



Basic Functions of UNIX Kernel

Unix Kernel has three basic jobs:

- Create User Processes and Schedule their execution
- Provide with sysem services
- Service hardware interrupts and exceptions

User Process Description

It consists of:

- Contents of its memory space, including the code and data areas
- Contents of CPU general purpose registers and program counter as well as any floating point registers or special status registers (hardware context).
- Other information such as open files, the current working directory and the status of any outstanding system call request (software context).

Process Creation and Deletion

Two fundamental processes are created during initial system boot strap procedure.

- Process O : Swapper Process
- Process 1 : Init process

New processes are created through *fork* and *exec* system calls.

At system startup the init process forks several daemons and gettys.

When a user logs in, many *execs* finally bring the command interpreter or shell into execution.

Processes get killed by the *exit* system call.

Process Definition Structures

- *u-area* contains all information about the process when it is active. Written out when the process is swapped
- *Process Table* contains information about inactive processes.
- Relevant header files are */user/include/sys/user.h* and */user/include/sys/proc.h*.

Sleep and Wakeup

Processes go to sleep because they have to wait for some event like

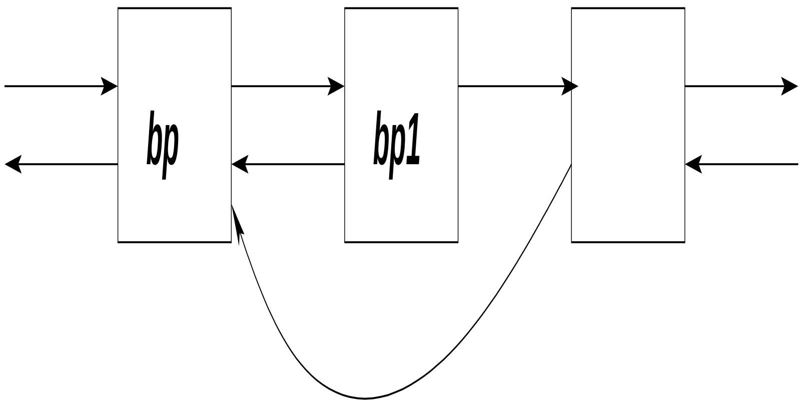
- I/O completion from a peripheral device
- a process to exit
- a resource to become avaliable

Processes *sleep* on an event and then *wakeup* because the event condition is no longer true.

Kernel Processes Non-Interruptable

```
 struct queue{

 } *bp, *bp1;
 bp1->forp=bp->forp;
 bp1->backp=bp
 bp->forp=bp1
/*possible context switch*/
 bp1->forp->backp=bp1
```



Fork Algorithm

```
input: none
output: child pid to parent and 0 to child

{
   Check for avaliable kernel resources;
   Get free process table slot and unique PID no;
   Check that the user is not running too many
     processes;
   Mark child state created;
   Copy date from parent proc. table slot to
     new child slot;
   Increament count on current directories node;
   Increment open file counts in file table;
   Make copy of parent context (u_area,data,stack);
   Push dummy system level context layer into child
     system level context;
  /*Dummy context contains data by which the
    child recognizes itself and starts running
    when scheduled*/
   if (executing process is parent){
      change child state to READY TO RUN;
      return (child-id);
   }
   else{
      Initialize the u-area timing fields;
      return(0);
   }
}
```

Process Synchronization

Processes need to be synchronized when they communicate and share data.

In UNIX this is achieved by

- Signals
- Shared Memory
- Message Passing

SIGNAL and SETJMP

```
#include <signal.h>
#include <setjmp.h>
jmp-buf sjbuf;
main ()
{
  int on_intr();
  signal (SIGINT,on_intr);
  sigsetjmp(sjbuf);
  /*save current stack position*/
  for (;;){
         /* main processing loop*/
  }
  ...
}
on_intr() /* reset if interrupted*/
{
  signal (SIGINT,onintr);
    /* reset for next interrept*/
  printf ("\n Interrupt\n");
  siglongjmp(sjbuf, 0);
    /*return to save state*/
}
```

Name	Description	Default section
SIGALRM	Alarm clock	Terminate
SIGBUS	Bus error	Term. with core
SIG CLD	Death of child proc.	Discarded
SIGFPF	FPF	Term. with core
SIGHUP	hangup	Terminate
SIGILL	illegal Instruction	Term. with core
SIGINT	CTRL - C	Terminate
SIGKILL	kill	Terminate
SIGPIPE	Write on to pipe (no one to read)	Terminate
SIGPWR	Power fail	Terminate
SIGQUIT	Quit (CTRL \)	Term. with core
SIGSEGV	segmentation violation	Term. with core
SIGSTOP	suspend (CTRL Z)	Stop process
SIGSYS	Bad arguement to system call	Term. with core
SIGTRAP	Trace trap	Term. with core
SIGUSRI	user defined signal 1	Terminate
SIGVSRZ	user defined signal 2	Terminate
SIGXCPU	CPU time lim. exceed	Terminate
SIGXFXZ	file size lim exceed	Terminate

Exercise: What does the following do?

```
#include <fcntl.h>
int fdrd, fdwrt; char c;
main(argc, argv)
{
  int argc; char *argv[];
  if (argc != 3) exit(1);
  if ((fdrd=open(argv[1],O_RDONLY))==-1)
     exit(1);
  if((fdwt=creat(argv[2],0666))==-1)
     exit(1);
  fork();
  rwdrt();
  exit(0)
}

rdwrt()
{
  for(;;){
    if (read(fdrd,&c,1) != 1) return;
    write(fdwrt,&c,1);
  }
}
```

Process Synchronization

Processes need to be synchronized whenever they share data and have speed difference.

Producer Consumer Problem: appears whenever two processes running asynchonously exchange data. One produces data that is consumed by the other process.

Example: Kernel and Disk Controller.

Buffer introduced to take care of speed difference.

```
Producer Code
-------------
while (1){
   produce(nextp);
   /*item is a local variable  nextp*/
   while (counter == n) loop();
   /* buffer full */
   buffer[in]=nextp;
   counter ++;
}
Consumer Code
-------------
while(1){
   while (counter== 0) loop();
   /* buffer empty*/
   nextc=buffer[out];
   counter--;
}
```

Shared variables between producer and consumer are *buffer* and *counter*.

Counter value indicates the number of items in buffer.

```
counter++                         counter--
---------                         ---------
load;reg_1<--counter     load;reg_2<--counter
incr;reg_1<--reg_1+1     decr;reg_2<--reg_2-1
store;counter<--reg_1    store;counter<--reg_2
```

When producer and consumer execute concurrently inconsistency arises in the value of the counter.

Producer executes

$$reg_1 \leftarrow counter; reg_1 \leftarrow reg_1 + 1$$

Consumer executes

$$reg_2 \leftarrow counter; reg_2 \leftarrow reg_2 - 1$$

Suppose original counter value = 7.

Producer executing store, counter=8
Consumer executing store, counter = 6

Thus the counter value may be 6, 7, 8 depending on the sequence of the execution leading to inconsistency.

Should be executed in Critical Section.

Critical Section Problem

Producer executing *counter++* and Consumer executing *counter–* are examples of **Critical Section Problem**.

A process executing in its CS will prevent other processes from executing their own CS.

Criteria for C.S Solution

1. Ensure mutual Exclusion
2. Prevent mutual blocking or ensure progress: The decision of which process should enter the C.S. is made by only the processes wanting to enter C.S. and no other process.
3. Ensure Bounded wait: A process waiting on the C.S. has to wait for only finite amount of time.

Process Structure

Pre amble (entry code)

Critical Section

Postamble (exit code)

Some attempts at solution, (two process)

P_i : while (turn == j) loop();

C.S.

turn = j

P_j : while (turn == i) loop();

C.S.

turn = i

this will not prevent mutual blocking. Suppose P_i executes postamble and immediately wants to enter C.S, it will be prevented from doing so.

Correct solution to 2 process C.S. problem:

Pi

```
    flag[i] = 1;
    turn = j;
    while (flag[j] == 1 && turn == j)
        loop();
    C.S.
    flag[i] = 0;
```

Pj

```
    flag[j] = 1
    turn = i;
    while (flag[i] == 1 && turn == i)
        loop();
    C.S.
    flag[j] = 0;
```

Mutual exclusion

Case 1: Pj executing in C.S, flag[j] = 1,
Pi will set turn = j and loop.

Case 2: both Pi & Pj want to enter C.S.
flag[i]=1=flag[j]. But turn = i or j but
not both. so only one process admitted.

Mutual blocking

Both wanting to enter C.S. will not loop at the while loop.
Suppose Pj is not executing C.S. flag[j] is 0. So Pi will not wait at the loop.

Bounded wait

Suppose Pi wants to enter C.S. This is indicated by flag[i] = 1. Pj after finishing once if it wants to enter again, will set turn=i and wait at the write loop.

But before it comes to *while*, if Pi sets turn=j then Pj will enter again. But after that Pi will definitely enter.

Dekker Algorithm

```
flag[i] = 1;
while (flag[j] == 1){
    if (turn == j){
        flag[i] = 0;
        while (turn == j) loop();
        flag[i] = 1;
    }
}
C.S.
turn = j;
flag[i] = 0;
```

Semaphore

Basic Definition

Atomic operation, P(S) : while (s <= 0) loop();

s = s - 1

V(S) : s = s + 1

Mutual exclusion

```
Pi                                   Pj
--                                   --
P(s)                                 P(s)
   C.S.                                C.S.
V(s)                                 V(s)
```

Initial value of s is 1.

Producer-consumer problem

n-cell buffer $\quad$ $0..n-1$ buffer

Semaphore	mutex	empty	full
initial value	1	n	0

(Initial value says there are initially n empty cell and 0 full cell)

```
Producer                    Consumer
--------                    --------
P(empty)                    P(full)
P(mutex)                    P(mutex)
  produce in                   consume from
  buffer                       buffer
V(mutex)                    V(mutex)
V(full)                     V(empty)
```

Synchronization

1. Readers-Writers Problem
2. Dining Phillosopher problem

Dining Phillosopher Problem

Semaphore chopstick[5];

Phillosopher[i] code

———————————-

```
think
P(chopstick[i])
P(chopstick[i+1 mod 5])
   eat
V(chopstick[i])
V(chopstick[i+1 mod 5])
   think
```

Synchronisation achieved. But Deadlock may occur when all phillosophers pick up stick. All execute P operation.

To solve deadlock problem in Dining phillosopher problem

Strategy 1: If a phillosopher lifts the left chopstick then both his neighbours lift right chopstick and vice versa. Starvation is possible in this strategy.

Stratergy 2: A phillosopher lifts a chopstick if the other is available.

Reader-Writer Problem

n readers and n writers.

Constraints

1. when one writer writes no reader or other writer can access the database.
2. Any number of readers can read.

semaphore mutex= 1, wrt= 1;

integer readcount= 0;

```
Writer                  Reader
-----                   ------
P(wrt);                 P(mutex)
   write;               readcount ++;
V(wrt);                 if (readcount == 1)
                    /* 1st reader */
                              P(wrt);
                        V(mutex)
                             read
                        P(mutex)
                        readcount --;
                        if (readcount == 0)
                          /* last reader */
                              V(wrt);
                        V (mutex);
```

Stadium Problem

A stadium has a capacity of 5000 in which a person can enter if there is room inside. Else he must wait and can enter only if somebody from inside leaves. Design a computer control system for this.

person wanting to enter presses *entry* switch
person wanting to exit press *exit* swich.

entry —> SIGINT
exit —> SIGQUIT

Two processes watching over a gate, P_entry and P_exit.

Structure of Processes

/*a semaphore *Stadium* is used and initialized to 5000, all processes use it*/

P_entry

```
# include <setjmp.h>
# include <signal.h>
struct jmp_buf watch;
main()
{
  Signal (SIGINT, admit);
  sigsetjmp (watch);
  while (1)
    write (screen, "welcome\n" , 8);
}
admit(){
  P(stadium);
  write (screen,"admitting in\n",10);
  open_gate();
  siglongjmp(watch);
}
```

P_exit

```
# include <setjmp.h>
# include <signal.h>
stuct sjbuf watch;
main ()
{
  signal (SIGQUIT,show_out);
  sigsetjmp (watch);
  while (1){
write(screen,"see you\n", 8);
}
show_out(){
  write (screen,"exiting\n",9);
  open_gate();
  V(stadium);
  siglongjmp(watch);
}
```

UNIX Semaphore

1. **A set of non-negative integer values**

```
# include <sys/types.h>
# include <sys/ipc.h>

struct semid_ds{
   /*operation permission struct*/
   struct  ipc_perm sem_perm;
   /*ptr to first semaphore set*/
   struct  sem      *sem_base;
   /* #of semaphores in set */
   ushort           sem_nsems;
   /*time of last sem_op*/
   time_t          sem_otime;
   /*time of last change */
   time_t          sem_ctime;
};
```

```
struct sem {
  /* semaphore value, non negative*/
   ushort       semval;
   /* pid of last opn*/
   short       sempid;
  /*#waiting semval><cal*/
  ushort        semncnt;
  /*# waiting semval = 0*/
  ushort        semzcnt;
};
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/sem.h>
int semget(key_t key,int nsems,int semflag);
int semop (int semid, struct sembuf,
         *opsptr[][],unsigned int nops);
struct sembuf {
      ushort sem num; /*semphone no.*/
      short semop;    /*semphone opn.*/
      short sem_flag; /*opn flags*/
};
```

Case

1. semop +ve, value of *semop* added to semphore's current value. Release of resources.

2. semop 0, caller wants to wait until semaphore value becomes zero.

3. semop negative, caller wants to wait until the semaphore's value becomes $\geq$ absolute value of semop. Allocation of resources.

```
int semctl(int semid, int semnum, int cmd,
                                   union semun arg);
union semun {
   /*used for SETVAL only */
   int val;
   /*used for IPC-STA & IPC-SET */
   struct semid_ds *buff;
   /* used for IPC-GETALL and IPC-SETALL
   ushort *array;
} arg;
```

Bridge Crossing Problem

To synchronize the passing of cars crossing a bridge from left and right with the constraint that at any point of time only cars from a particular direction can go over the bridge.

```
P_left
------
P(mutexL);
leftcount ++;
if (leftcount == 1) P(wait);
V(mutext);
    pass;
P(mutext);
leftcount--;
if (leftcount == 0) V(wait);
V(mutext);
```

```
P_right
-------
P(mutexR);
rightcount++;
if (rightcount==1) P(wait);
V(mutexR);
     pass;
P(mutexR);
rightcount --;
if (rightcount == 0) V(wait);
V(mutexR);
```

Simple Interface to UNIX semaphore

```
/*create with initial value, or open*/
id = sem_create(key, initval);
/*open, must already exit*/
id = sem_open(key);
/*P operation down by 1*/
sem_wait(id);
/*V operation, up by 1*/
sem_signal(id);
/*wait if amount<0, signal if amount>0 */
sem_op(id, amount);
/*close*/
sem_close(id);
/*remove (delete)*/
sem_rm(id);
```

3-member semaphore set

first member [0] is actual semaphore value. [1] is used to know when all processes have finished with the semaphore. [2] is used as a lock variable.

```
# include <sys/tupes.h>
# include <sys/ipc.h>
# include <sys/sem.h>
# include <errno.h>
extern int errno;
/*initial value of process counter*/
# define BIGCOUNT 10,000
/*semaphore opn. arrays for semop calls*/
static struct sembuf op_lock[2]={
  2,0,0, /* wait for [2](lock) to equal 0*/
  2,1,SEM_UNDO /*then increment[2] to 1.
    This locks it. UNDO to relese lock.
    Processes exit before explicitly
    unlocking*/
};
```

```
static struct sembuf  op_endcreate[2]={
  1,-1, SEM-UNDO, /*decrement [1] (process
                 counter) with undo on exit*/
    /* undo to adjust process counter if
     process exits before emplicity
     calling  sem_close() */
  2, -1, SEM_UNDO /*then decrement[2] back
     to 0 */
};

static struct sembuf op_open [1] = {
  1,-1,SEM-UNDO /*decrement [1],proc counter
              with UNDO on exit */
};
```

```
static struct sembuf op_close[3] = {
  2,0, 0, /*wait for [2] (lock) to equal 0*/
  2,1, SEM-UNDO /* then increment [2]
  to 1,this locks it */
  1,1, SEM-UNDO /* then increment [1]
  proc counter*/
};

static struct sembuf op_unlock [1] = {
  /*decrement [2] (lock) back to 0 */
  2,-1, SEM-UNDO
};

static struct sembuf op_op[1]= {
  /*decrement or increment [0] with
   undo on exit */
  0,99, SEM-UNDO
  /* the 99 is set to actual amount
    to add or subtract (+ve or -ve)*/
};
```

```
int sem_create(key, initval)
key_t key;
int initval;
{
  register int id, semval;
  union semun{
       int val;
       struct semid_ds *buf;
       ushort  *array;
  }semctl_arg;
  again:
  if((id=semget(key,3,0666 | IPC_CREATE))<0)
     return (-1); /*permission problem or
                  table full */
  if (semop(id, &op_lock[0],2) < 0){
     if (errno == EINVAL ) goto again;
     err_sys ("Cannot lock");
  /* critical  section for changing
      process counter*/
  }
```

```
/* get the value of process counter.
 if it is 0 then no one has
 initialized the semaphore */

if ((semval == semctl(id,1,GETVAL,0))<0)
    err_sys ("cannot GETVAL");
if (semval == 0){
   semctl_arg.val = initval;
   if (semctl (id,0,SETVAL,semctl_arg)<0)
      err_sys ("cannot SETVAL[0]");
   semctl_arg.val = BIGCOUNT;
   if (semctl(id,1,SETVAL,semctl_arg)<0)
      err_sys ("cannot SETVAL[1]");
}
/* decrement process counter and then
release lock*/
if (semop (id, &op_endcreat[0],2)<0)
    err_sys ("cannot end creat");
return(id);
```

```
/*open an existing semaphore */
int sem_open(key)
key_t key;
{
  register int id;
  /*does not exist or table full */
  if ((id = semget(key,3,0))<0)
    return (-1);
  /* decrement the process counter */
  if (semop(id, &op_open[0], 1)<0)
    err_sys ("cannot open");
  return(id);
}

sem_rm(id) /* remove a semaphore*/
int id;
{
if (semctl(id,0,IPC_RMID,0)<0)
   err_sys ("cannot remove");
}
```

```
sem_close(id) /*close a semaphore*/
int id;
{
  register int semval;
  if (semop (id &op_close[0],3)<0)
     err_sys ("cannot semop");
  /*read the process counter to see if this
  is the last reference to semaphore*/
  if ((semval = semctl(id,1,GETVAL,0))<0)
     err_sys ("cannot GETVAL");
  if (semval>BIGCOUNT)
     err_sys("impossible BIGCOUNT value");
  else if (semval == BIGCOUNT)
     sem_rm (id);
  else if ((semop(id,&op_unlock[0],1)<0)
     err_sys ("cannot unlock"); /*unlock*/
}
```

```
sem_wait (id) /*P operation */
int id;
{
  sem_op (id,-1);
}
sem_signal (id) /*V operation*/
int id;
{
  sem_op (id,1)
}

/* general semaphore operation increment or
decrement by user specified amount*/
sem_op (id,value)
int id;
int value;
{
  if ((op_op[0].sem_op = value) == 0)
     err_sys ("cannot have value == 0");
  if (semop (id, &op_op[0], 1) <0);
     err_sys ("semop error");
}
```

Exercise

Sleeping Barber Problem.

A barber shop consist of a waiting room with n chairs and the barber room containing the barber chair. if there are no customers to be served the barber goes to sleep. If a customer enters the barbershop and all chairs are occopied then the customers leaves the shop. If the barber is busy but the chairs are avaliable, then the customer sits in one of the free chairs. if the barber is asleep the customer wakes up the barber.
Write a program to co-ordinate the barber and the customers.

Algorithm

```
semaphore WaitforCust=0, mutex=1;
int nr_customer=0;//shared memory
BarberProcess    CustomerProcess
-------------    ---------------
P(WaitforCust)   P(mutex)
give_service()   if (nr_customer==n)
nr_customer--       //no chair
                     V(mutex);
                     go_back();
                 else
                     get_chair();
                     if (nr_customer==0)
                     //first customer
                       V(WaitforCust);
                     nr_customer++;
                     V(mutex);
```

Shared Memory

Another method of IPC and synchronization.

The following shows how to get shared memory for a process.

```
/* get shared memory*/
shmid = shmget(shmkey, MEMSIZE,0660)

if (shmid == -1)
   shmid = shmget (shmkey,MEMSIZE,
     0660 | IPC-CREATE)
if (shmid == -1)
  err_sys("shm call failed to creat
                  or open");
/*attach shared memory to address space*/
ptr_shm = shmat (shmid,0,0);
if (ptr_shm == -1)
   err_sys ("shmattach failed");
```

Shared Memory Example

a program to show how a process reads from and writes into a shared memory at two different addresses.

```
# include <sys/type.h>
# include <sys/ipc.h>
# include <sys/shm.h>
# define k 1024
# define SHMKEY 75
int shmid;
main()
{
   int i,*pint;
   char *addr1, *addr2;
   extern char *shmat();
   extern cleanup();
   for (i=0; i<20; i++)
     signal (i, cleanup);

   shmid = shmget (SHMKEY,128*k,
                    0777|IPC_CREATE)
   addr1 = shmat(shmid,0,0);
```

```
   addr2 = shmat(shmid,0,0);
   printf("addr1=0x%x, addr2=0x%x\r",
            addr1,addr2);
   pint = (int *) addr1;
   for (i=0; i<256; i++)
      *pint++ = i;

   pint= (int*) addr1;
   pint = 256;
   pint= (int*) addr2;

   for (i=0; i<256; i++)
     printf ("Index %d\t value %d/n",
             i,*pint ++);
   pause(); /*suspend execution*/
}
```

```
cleanup()
{
   shmctl(shmid,IPC_RMID,0);
   exit();
}

/* Another process: waits until the 1st
process writes 0 into the 1st location of
shared memory*/

# define SHMKEY 75
# define K 1024
int shmid;
main()
{
    int i,*pint,char *addr;
    extern char *shmat();
    shmid = shmget (SHMKEY, 64*K, 0777);
    addr = shmat (shmid,0,0);
    pint = (int *) addr;
    while (*pint = = 0);
    for (i=0; i<256; i++)
       printf(" %d\n",*pint++);
}
```

Shared Memory Exercise

Implement the Reader-Writer, Bridge Crossing and the Sleeping Barber problem using signals and shared memory.

Memory Management and UNIX

Memory Management Principles

Memory Management is concerned with the management of *primary memory*. Primary memory is the volatile memory which disappears when the power is put off. This is as opposed to the *secondery memory* which concerns hard disk, floppy, tape drive etc.

The goal of memory management is to *keep many processes in the primary memory to allow multiprogramming*.

Memory Management Functions

Keep track of the status of each location in primary memory, *i.e.* whether allocated or free.

Determine the allocation policy for the memory, *i.e*, deciding to whom it should go, how much, when and where.

Deallocation policy handles the reclamation of the memory.

A very important concern of memory management is to ensure clear separation between the user's view of memory and the actual physical memory.

Logical and Physical Memory

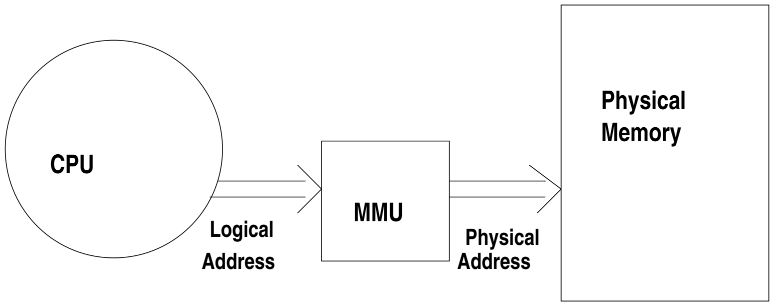


Figure: Logical and Physical Address

The user program believes that memory is one contiguous space, containing only this one program. In actuality the user program is scattered throughout physical memory, which also holds other programs.

The difference between the user's view of memory and the actual physical memory is reconciled by the *address translation*, or mapping, hardware.

The above figure shows that the address output by the CPU onto the address bus is the *logical address* which goes into the *Memory Management Unit (MMU)*. The output of the MMU which goes to the Physical store is the *Physical Address*.

Memory Management Evolution

Bare Machine: The whole machine is available to the user. User can control the use of memory as he wishes. However, no service in terms of interrupt handling, system call or error processing and job sequencing.

Resident Monitor: Memory is divided into two sections, one for the user and the other for the resident monitor of the OS.

There is a *fence register* between the areas to check the correctness of all the memory accesses.

Swapping: When the next user progam is to be executed, if there is not enough memory, current contents of the user memory is written out to the backing store and next user's memory is read in.

Evolution

Paging: Memory is divided into *fixed sized* blocks and the process is allocated memory in terms of number of blocks. This, however, suffers from *Internal fragmentation* which means partial usage of a unit of memory so that the rest of the memory unit is unusable.

Segmentation: Memory is divided into protions which reflect the structure in the user's program like *functions, data structures* etc. This, however, suffers from *external fragmentation* which means a situation in which the physical memory has many unused chunks scattered all over each too small to meet normal request.

Compaction is required to eliminate holes in the memory resulting from external fragmentation.

Fragmentation

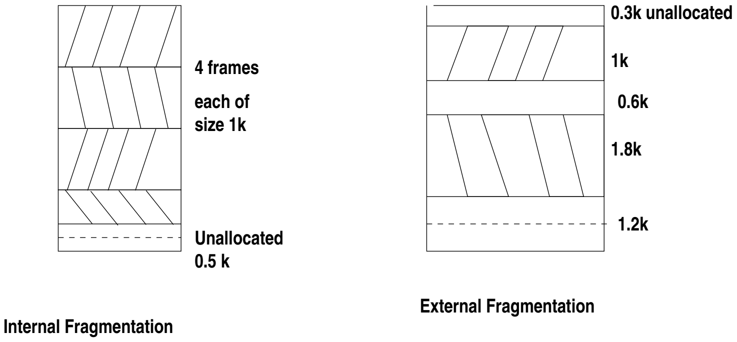


Figure: Fragmentation

In the above figure we see the examples of Internal and External Fragmentation. In the first case, the physical store has 4 frames each of 1K byte. A program occupies 3.5K bytes. The remaining 0.5K byte is free but unallocatable. This is *internal fragmentation.*

In the second case, the memory has program occupying 1K and 1.8K bytes. 0.3K, 0.6K and 1.2K are free. However, if another program needs more than 1.2K of contiguous storage, it cannot have it. This is *external fragmentation.*

Non-contiguous Memory Allocation

Contiguous memory allocation for processes is an unnecessary restriction,

Processes may be allocated memory anywhere in the physical memory, though the user perceives memory to be a continuous block available to himself only.

So address translation between logical address (LA) and physical address (PA) is done by Memory Management Unit (MMU).

Logical Address- Address output by the CPU to the MMU.

Physical Address- Address output by the MMU to the physical memory.

Paging

Informally, a *page* means a block of memory in the logical address space.

Paging permits a program to be allocated physical memory *wherever it is available.*

Every address generated by the cpu divided into two parts: a page number p and a page offset d.

The page number is used as an index into a page table. The page table contains the *base address* of each page in physical memory.

Base address combines with the page offset to define the physical memory address.

Physical memory is broken into fixed-size blocks called *frames*. Logical memory is also broken into blocks of the same size as *pages*.

Address Calculation in Paging

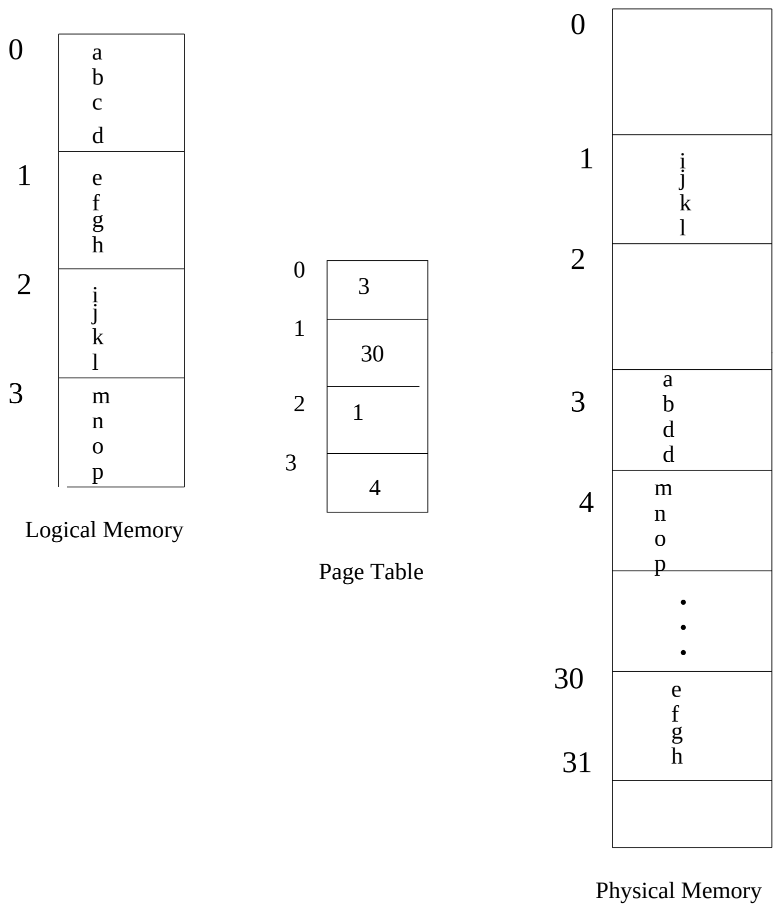


Figure-1: Paging

Consider the above figure. Here page size is 4 words and a physical memory has 32 words (8 pages).

In the logical memory there are 4 pages.

continued

Therefore the page table has 4 entries mapping pages in logical address space to physical address space.

The mapping is logical page 0 to physical page 3, 1 to 30, 2 to 1 and 3 to 4.

Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 3.

Thus logical address 0 maps to physical address 12 (= 3 x 4 + 0).

Logical address 3 (page 0, offset 3) maps to physical address
15 (= 3 x 4 + 3).

Features and Advantages of Paging

While doing *job scheduling*, the pages of the job are loaded into the allocated frames, and the frame numbers are put in the page table for this job.

Instructions to load or modify the page table registers are previlleged.

There is no *external fragmentation.* Any free frame can be allocated to a job that needs it.

Internal fragmentation is possible. In the worst case, a program would need n pages plus 1 word.

Small page size is desirable to minimize internal fragmentation. However, too small a page size makes the page table too big.

Advantages of Paging

Paging increases memory usage by having *code pages shared*

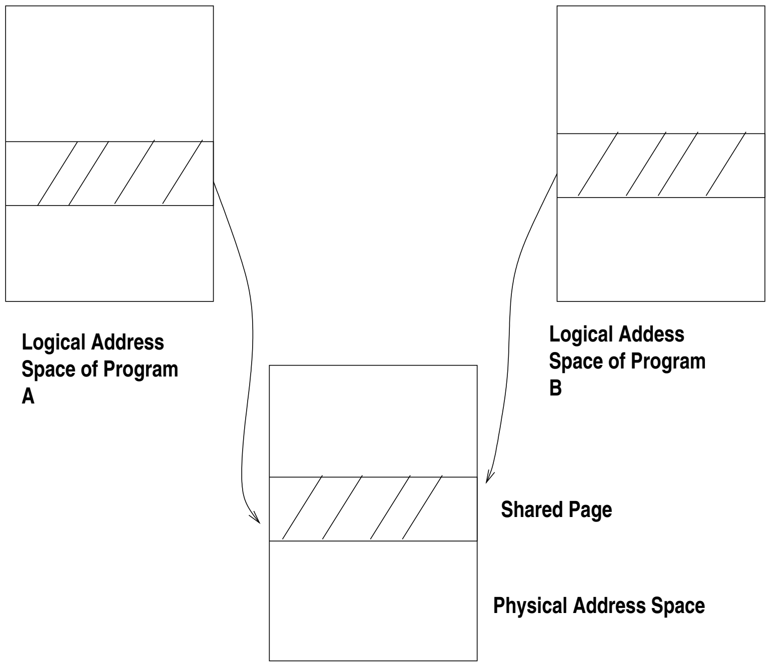


Figure: Page Sharing between Processes

Paging facilitates *protection* by associating *read/write* bits with page table entries.

Paging facilitates *virtual memory* by associating *valid/invalid* bits with page table entries.

Disadvantages of Paging

Two memory accesses needed to access a word (one for the page table, one for the word). Thus memory is slowed by a factor of two.

So we use a small hardware memory, called a *address translation buffer* which contains part of the page table, i.e., page numbers and their corresponding frame numbers.

If the page number is not in the address translation buffer, a memory reference to the page table in the memory must be made.

Effective Access Time

The percentage of times that a page number is found in the associative registers (the *hit ratio*) related to the number of associative registers.

An 80 percent hit ratio means that 80 percent of the time we find the desired page number in the associative registers.

The Effective Access Time (t_e) is the expected time required when address translation buffer is used.

$t_e = p.(t_s + t_a) + (1 - p).(t_s + 2t_a)$,
where t_s is the time to search the cache and t_a is the memory access time.

Virtual Memory

Virtual memory allows the execution of processes that may not be completely in memory.

So *user programs can be larger than physical memory.*

Motivation for having parts of programs in memory is that in real situations, the entire program is not required in main memory. Examples are:

i) Programs often have code to handle unusual error conditions. Since these errors seldom occur in practice, this code is almost never executed.
ii) Arrays, lists, and tables are generally allocated more memory than they will actually need.
iii) Certain options and features of a program may be rarely used, such as a text editor command to convert all characters in a range of lines to upper case.

Advantages of Virtual Memory

The ability to execute a program which is only partially in memory would have many benefits:

i) A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for a very large *virtual address space.*
ii) Since each user could take less physical memory, more users can be run at the same time, with a corresponding increase in cpu utilization and throughout but no increase in response time or turnaround time.
iii) Less I/O would be needed to load or swap each user into memory so each user would run faster.

Demand Paging

Demand Paging means bring a page of memory from the secondery storage to the primary memory only when it is required.

Demand paging uses *lazy swapping*. The lazy swapper never swaps a page into memory unless it is needed.

It decreases the swap time and the amount of physical memory needed allowing an increased degree of multiprogramming.

Pure demand paging refers to a situation in which the program begins executing with no page in the memory and immediately traps to the OS for its page. It goes on executing and trapping and this way brings memory pages as and when needed.

Page Fault and Replacement

If the program tries to access a page which is not in the memory, a *page fault* is said to occur.

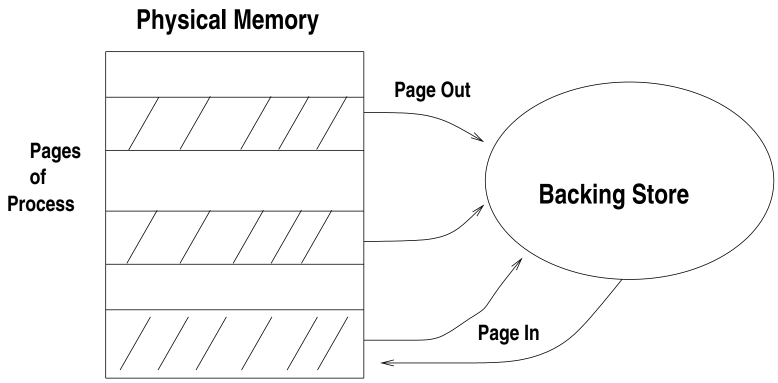


Figure: Page Fault Handling

The figure above shows how the pages of a program can go to the backing store and also can come into the primary memory whenever required.

Paging in and out are costly in terms of time as they require scheduling disk input output operations.

Page Fault Handling

The paging hardware, in translating the address through the page table notices that the invalid bit is set, causing a trap to the operating system for an invalid address error.

The operating system determines where the desired page is on the backing store and loads it into main memory. Then it restarts the instruction.

However, sometimes it may so happen that there are no free frames on the free frame list to accommodate the desired page.

This situation calls for *Page Replacement*.

Page Replacing Algorithm

1. Find the location of the desired page on the backing store.

2. Find a free frame.

3. If there is a free frame use it.

4. Otherwise use a page-replacement algorithm to select a victim frame.

5. Write the victim page to the backing store and change the page and frame table accordingly.

6. Read the desired page into the (newly) free frame, changing the page and frame tables.

7. Restart the user process.

Strategy for Victim Page Selection

When page replacement is required, we must select which frames are to be used.

In general, we want the one with the lowest page fault rate.

An algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.

The string of memory references is called a *reference string.*

Reference strings can be generated artificially (by a random number generator, for example) or by tracing a given system and recording the address of each memory reference.

First In First Out

A *First In First Out (FIFO)* replacement algorithm associates with each page the time when it was brought into memory.

When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory and replace the page at the head of the queue.

A freshly brought page is inserted at the end of the queue.

The FIFO page replacement algorithm is easy to understand and program. However, its performance is not always good.

In particular it may give reise to an anomalous situation called *Belady's Anomaly*.

Belady's Anomaly

Consider the reference string
`1,2,3,4,1,2,5,1,2,3,4,5`.

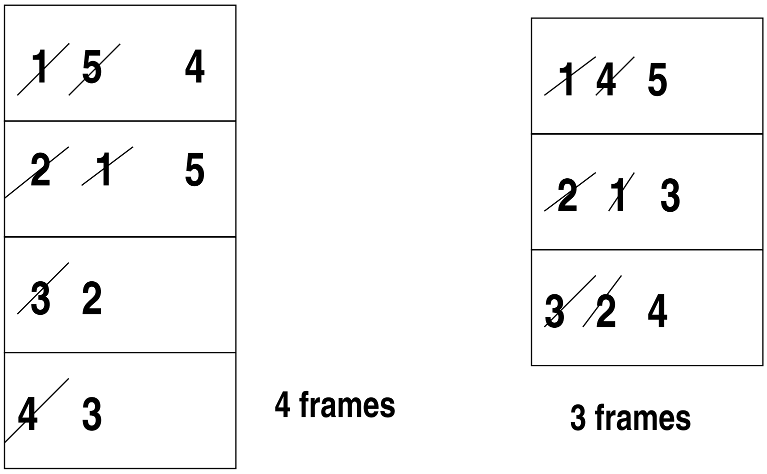


Figure: Belady's Anomaly

Initially there are no pages in the memory. With 4 frames there are 4 page faults at first. Then each crossed number in the above figure shows a fault. Thus there are 10 page faults with 4 frames.

Similarly there are 9 page faults with 3 frames.

Thus the number of faults for four frames (10) is greater than the number of faults for three frames (9). This is known as *Belady's anomaly*.

Optimal Replacement

An optimal page replacement algorithm simply states:

Replace that page which will not be used for the longest period of time.

Using this page replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Also it does not suffer from Belady's Anomaly.

Unfortunately, the optimal page replacement algorithm is difficult to implement, since it requires future knowledge of the reference string.

As a result, the optimal algorithm is used mainly for comparison studies.

LRU Strategy

Replace that page which has *not been used for the longest period of time.*

LRU page replacement is implemented by keeping a *stack* of page numbers. Whenever a page is referenced, remove it from the stack and put on the top.

Dirty bit is an important form of hardware assistance provided for page replacement. It indicates if a page has been modified.

Each page or frame has a dirty bit associated with it in the hardware. The dirty bit for a page is set by the hardware whenever a word in the page is written into.

While replacing, write a page to the backing store only if its dirty bit is set.

This reduces I/O time by half.

Thrashing

As the number of frames allocated to each process decreases, the page fault rate (and effective access time) increase, slowing process execution.

There is a minimum number of frames which must be allocated. This minimum number is defined by the instruction set architecture. Otherwise *Thrashing* may occur.

A process is *thrashing* if it is spending more time in paging than executing.

A process has a number of pages in active use. Then it wants another page. If there are no free frames, an active page may be swapped out and the new page brought in. But then it will fault again.

Thrashing severely degrades the system performance.

UNIX Memory Handling

This is done through *Regions*.

A region is a contiguous area of a process's address space such as text, data and stack.

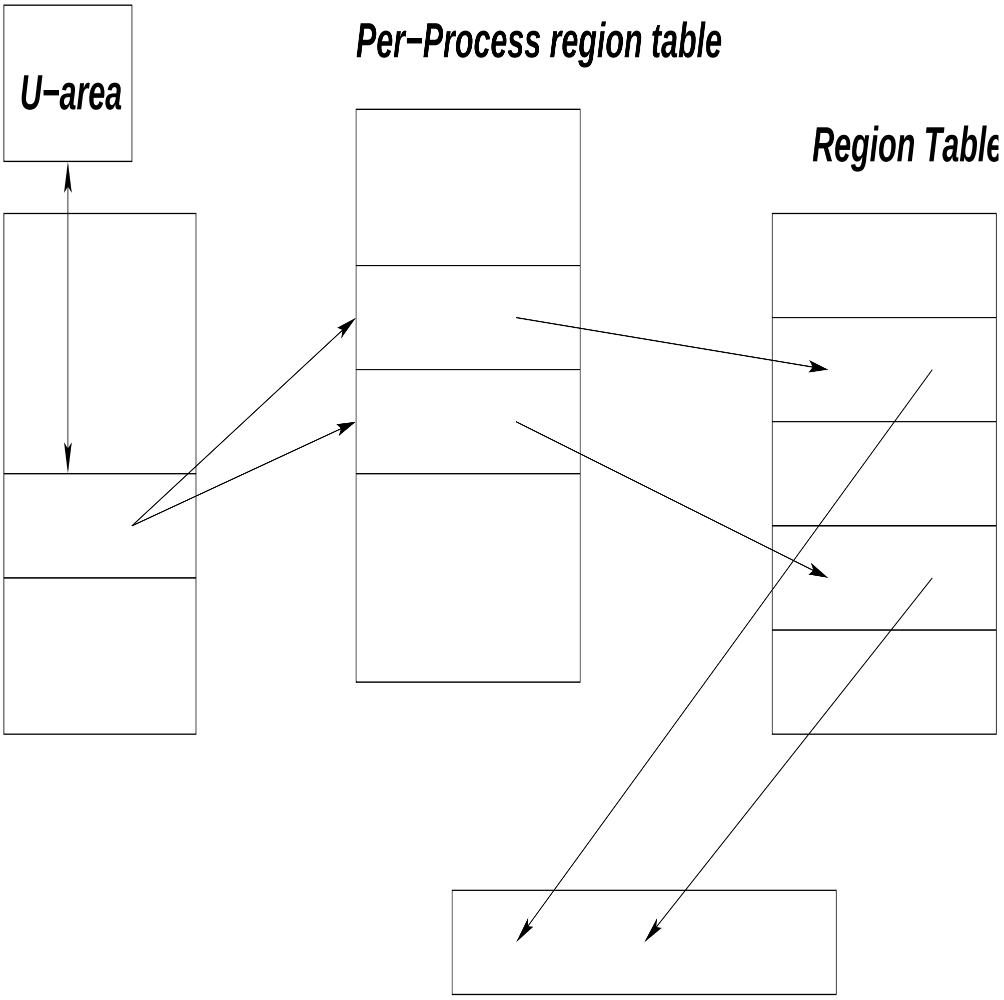
Region table entries describe attributes of the region, e.g., *private* or *shared* and also where the *data* of the region is.

exec system call replaces old regions.

fork duplicates the address space of the old process.

exit frees the regions the process has used.

UNIX Memory Data Structures



Regions

Regions are contiguous locations in a portion of virtual address space.

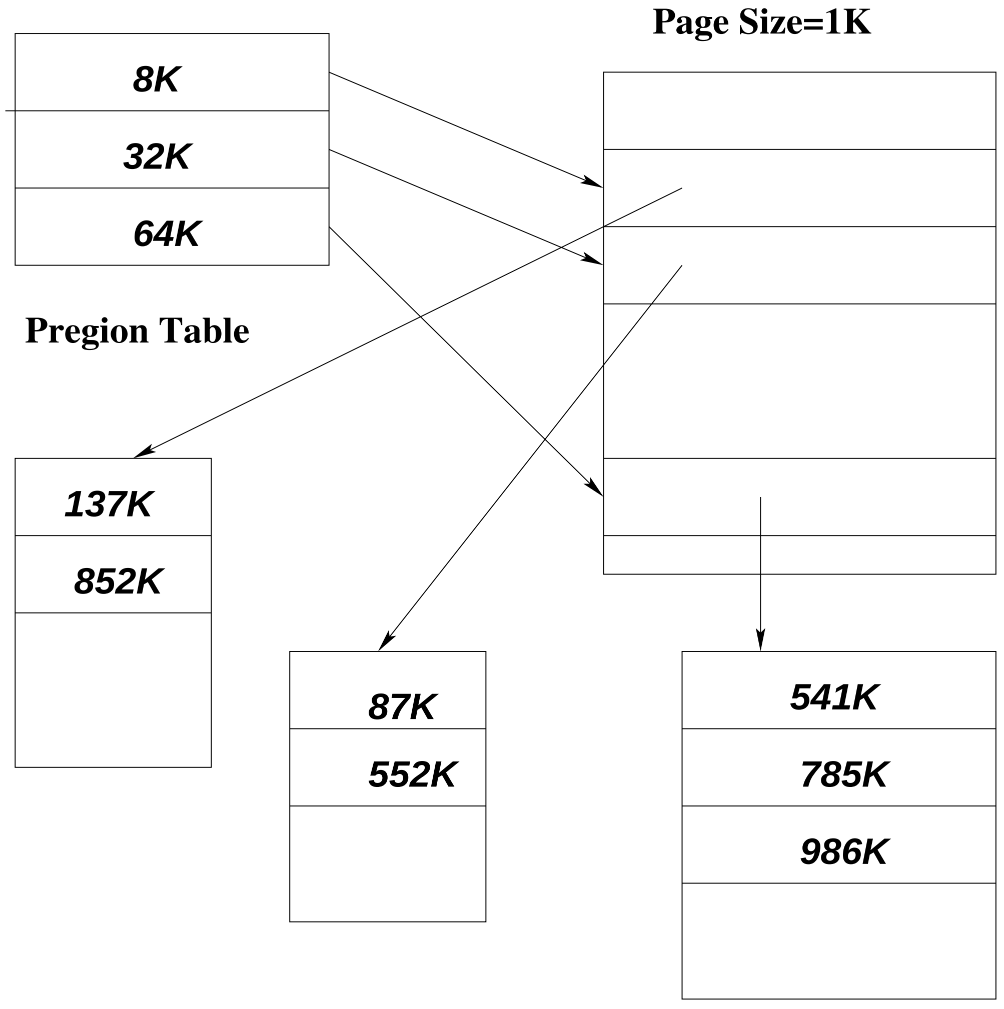
Each process has a per process region table.

Each entry in the pregion table points to the system-wide Region table (like user file table pointing the Inode table).

Entries in pregion table also show the starting virtual address of the region.

Region table entries show- among other things- the permission, sharing information, no.of pages and contain pointer to the page table.

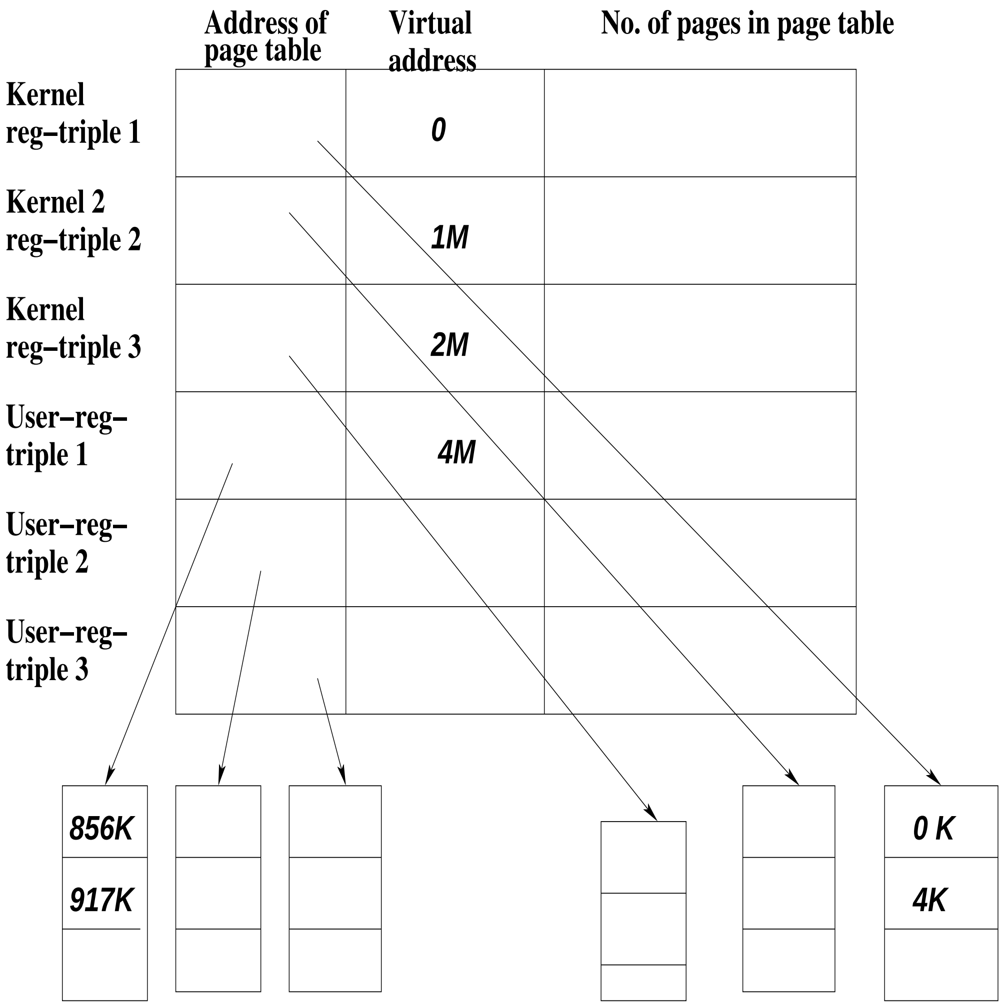
Address Calculation with region



Virtual Memory Address
68,432= Offset 848 in page starting at Physical address 986K.

Implementation

Implementation is through register triples.



Details of Region table entries

- Pointer to the inode of the file which were originally loaded into the region
- Region type (text, shared memory, private data or stack)
- Region size
- Status which may be a combination of
  - locked
  - in demand
  - being loaded into memory
  - valid, loaded into memory
- Reference count, giving the number of processes referencing the region

Operations on Regions

- lock
- unlock
- allocate
- attach
- change size
- load from file
- free
- detach
- duplicate contents

Allocation of Region

Allocation of region (algorithm *allocreg* in UNIX code) happens on *fork, exec* and *shmget* system calls.

Region table entries appear in a free linked list or on an active linked list.

Region is removed from linked list of free regions.

Region type is assigned.

Region is placed in the list of active regions.

Inode of the file whose contents are loaded is assigned.

Attaching a region

During *fork, exec* and *shmat* calls.

A region gets attached to the address space of a process.

The system imposed limit for the highest virtual address should be exceeded.

Virtual address of the new region should not overlap with the address of existing region.

Changing region size

A process may expand or contract its virtual address space with *sbrk* system call.

Extra page table entries are created (freed) and pages allocated (deallocated).

Stack grows automatically.

Shared region size is never changed.

Loading a region

kernel copies the executable file in the main memory.

Loading the process regions at virtual addresses specified in the executable file.

After loading the region all the processes waiting for the region to be loaded are woken up.

Freeing a region

When a region is no longer attached to any process, the Kernel can free it and return it to the list of free regions.

Page tables and memory pages are released.

Detaching a region from a process

The kernel detaches regions in *exec, exit, shmdt* system calls.

Region entry is updated.

Register triple is invalidated.

Region reference count and process size are decremented.

Duplicating a region

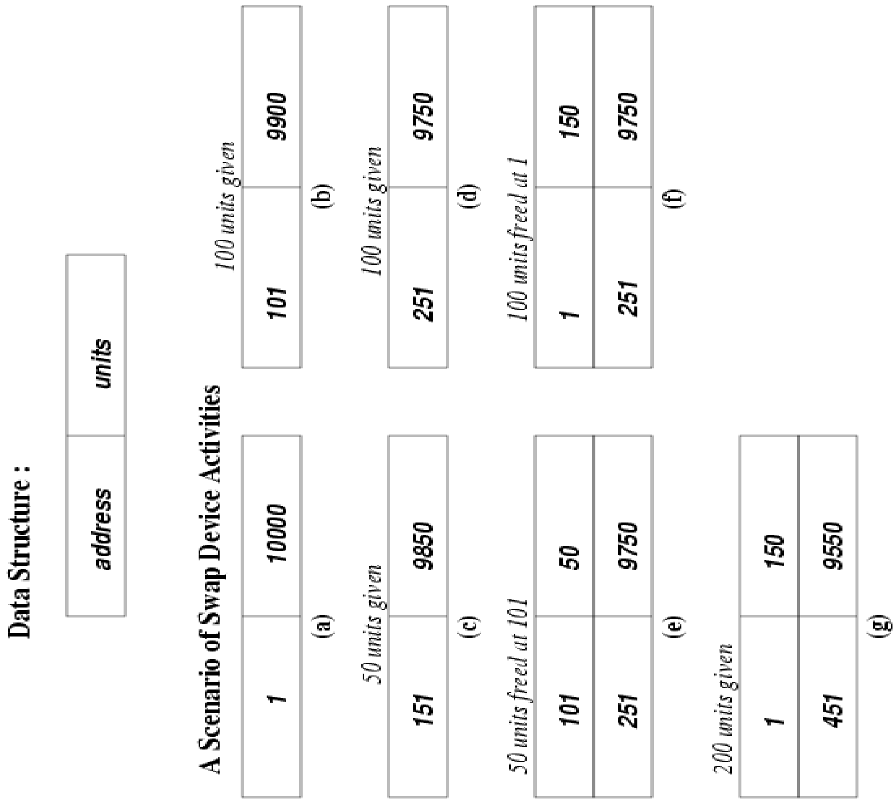
Fork system call requires duplication of process region.

Text is shared.

New regions are required for data and stack.

Swapping Data Structure

A scenerio of swap device activities.

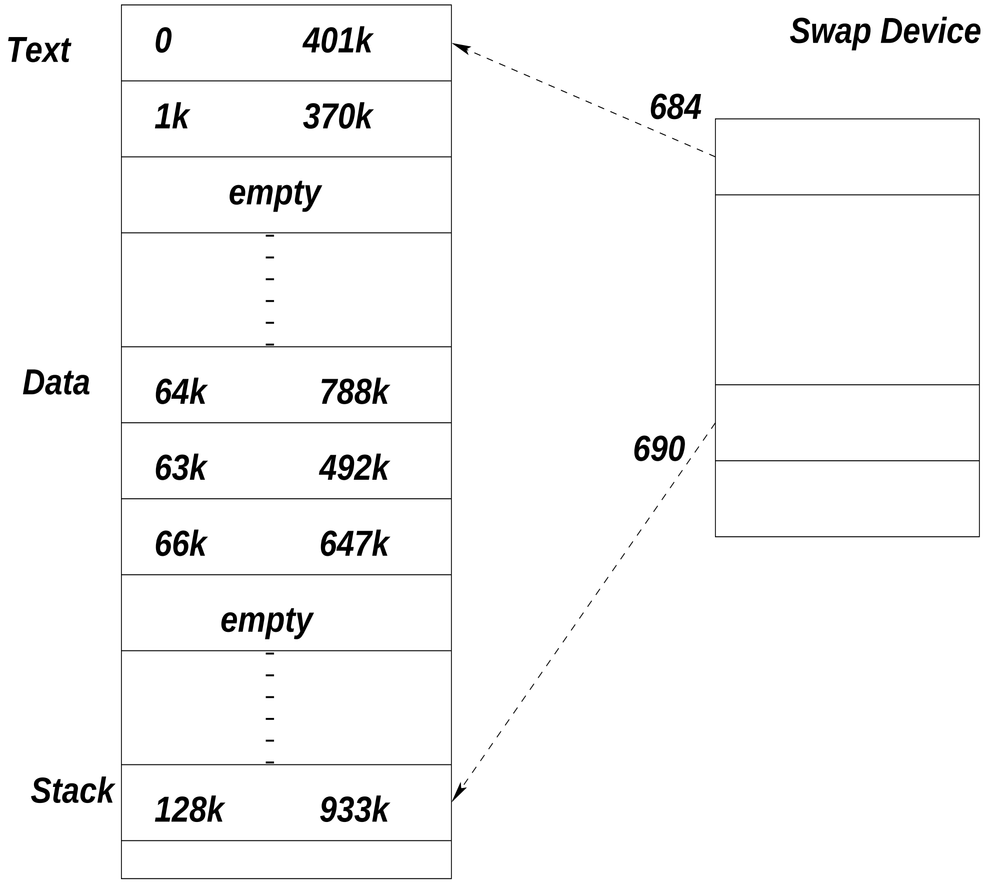


Swapping out a process

Swapping out of a process happens when

- *fork* allocates space for child.
- *brk* increases process size.
- stack grows.
- new process is swapped in.

Swapping a Process into Memory



Swapping a process into memory

Demand Paging

Data structures for Demand paging:

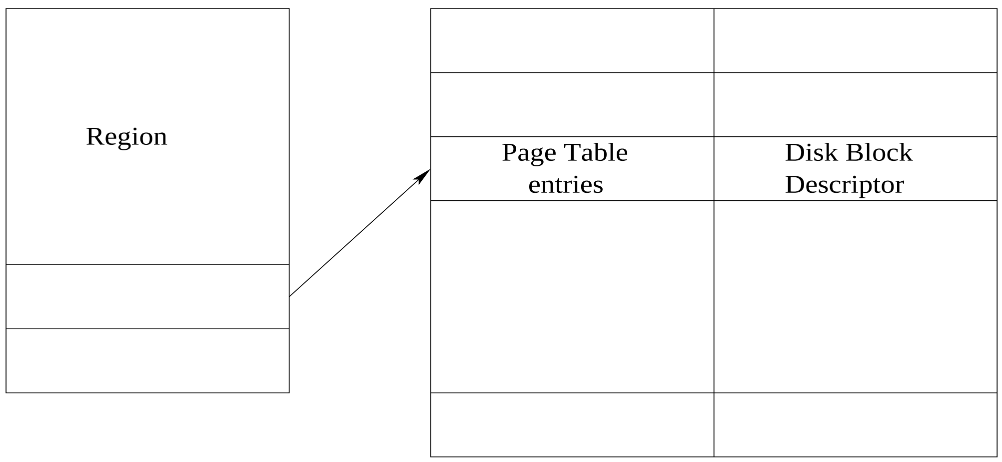
- Page Table Entries
- Disk Block Descriptors
- Page Frame Data table
- Swap use Table

Bits Used

- Protection Bits (*read, write, execute*)
- For demand paging:
  - valid
  - Reference
  - Modify
  - Copy on write
  - Age

Demand Paging Data Structures

Demand Paging



Page Table Entry

Page(phy.) Address	Age	Cp/Wt.	Mod	Ref	Val	Prot

Disk Block Description

Swap Destination	Block Number	Type

pfdata table describes each page of physical mem. Its entries have following fields:

- page state (page on swap device or executable file)
- no. of processes referencing the page
- logical device (swap or file system) and the block no. that contains a copy of the page.
- pointers to other pfdata table entries.

Devices and UNIX

Device Management

Concerned with

- Allocating and Deallocating device storage
- Keeping track of device status

Classification of Devices

- Based on Service
  - *Dedicated device*, which is allocated for the job's entire duration, for example, card reader, tape or printer
  - *Shared devices*, which are shared concurrently by several processes, for example, disks and drums
  - *Virtual devices*, which are normally dedicated devices, but converted to shared devices through *spooling*, for example, printer
- Based on I/O/Storage
  - *Input-output* devices like terninals and printers
  - *Storage* devices like hard disks, floppies etc.

- Based on the variation of access time t_{ij}, which is defined as the time to access item j given that the last access was item i
  - *Serial access* where t_{ij} has a large variance, e.g, tape.
  - *Completely Direct Access* where t_{ij} is constant, e.g., core.
  - *Direct Access* where t_{ij} has only a small variance, e.g., disk.
- Electromechanical and Electromagnetic Devices
  - The electromechanical devices are Card Readers, Printers, Teletypes etc. Only capable of Sequential I/O.
  - The electromagnetic devices Tapes and Disks. The magnetic properties of the I/O medium change as a result of recording.

Magnetic Tapes

Strip of magnetic material on which all bits of a byte are recorded laterally by employing an array of magnetic heads.

Information recorded by a head is said to form a *track* along the I/O medium. Thus, the recording is multitrack.

Error detection is by means of parity check and cyclic checksum of bytes.

Inter-record gap between consecutive records provides for the start/stop motion of the medium.

Access time in Tapes

A *record* is defined as the information read/written during *one* read/write operation on the magnetic tape.

The recording density, i.e., the number of bits per unit length of a track is of the order of 6400 bits per inch (bpi).

Time taken for an I/O operation = $t_a + s/dv$ seconds,

where, t_a (5 milliseconds) is the time taken for setting up the mechanical motion at a constant velocity v (2-8 m/s), d (6400 bpi) is the recording density and s is the size of a record in bytes. (no data transfer takes place during t_a),

This leads to a transfer rate of 64-256 K bytes/second.

Tape Commands

The commands accepted by a magnetic tape device are:

```
read a record
write a record
skip a record
rewind tape
read backwards
skip a record backwards
```

Disk Storage Devices

This medium is a circular surface made of a magnetic material, in continuous motion around its own axis whenever the device is in *ready* state.

Continuous motion eliminates mechanical start/stop motion thereby reducing the access times.

Records along a track generally have the form of a *record descriptor* followed by the record itself.

The record descriptor stores the record address, record length etc.

Successive records on a track are separated by *inter-record gaps.*

Disk (continued)

Continuous motion of the surface implies that no read/write operation can be performed until the required record passes under the head.

This rotational delay is known as the *latency time* of the device.

A record is read/written/addressed using the two component address
$<$*track number, record number on track*$>$.

Disks are called Directly Addressible Storage Devices (DASDs) because of the *two-dimensional* addressibility of records and the possibility of accessing any record in approximately the same amount of time, irrespective of its position along a track (like RAM).

Disk Performance

To provide faster access:

```
More head Per Track (HPT).
multiple heads along each track
   (latency=1/2n), where n=no. of
                         heads on track.
```

To provide higher capacities:

```
Increased recording density along track.
Increased track density along radial
     direction.
Disk Pack.
```

Disk Operations

Three Dimensional addressing in disk packs. Record address specified by

<cylinder no., track no. in cylinder, record no. in track>

Commands supported by a disk unit:

```
read a record
write a record
Seek (for moving head disks)
```

Transfer rates for HPT disks are of the order of a few mega-bytes per second, while for moving head disks, the rates are around 150-200 K bytes per second.

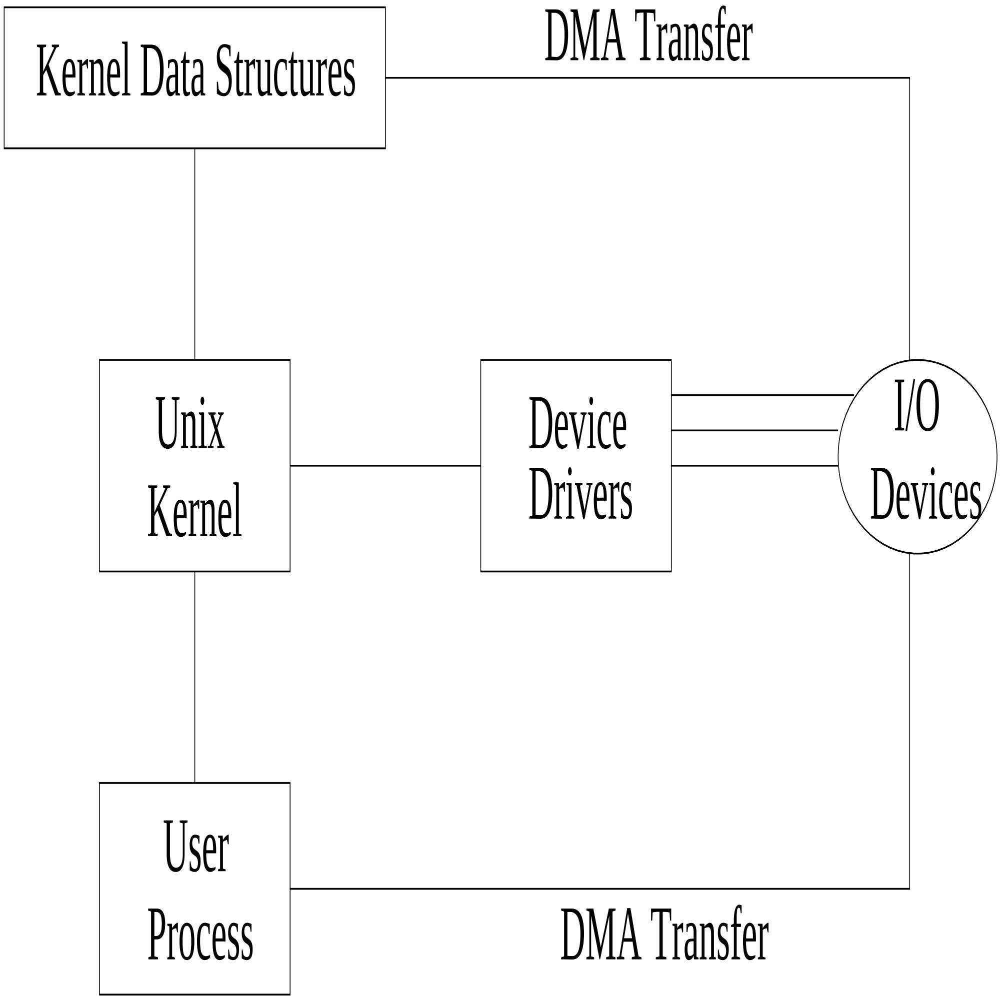
Floppy disks are low cost moving head disks with low capacities and transfer rates.

Device Driver Definition

Traslates generic requests received from the operating system to device specific commands.

Interface between the kernel and the physical device.

A device driver can manage several devices of the same kind.



Need for Device Driver

Operating system can be written without any knowledge of the specific devices that will be connected to it.

This allows the O/S to view all devices uniformly.

New devices can be supported without any change to the operating system.

Adding new devices to a system is easier.

Devices from different vendors can be supported on the system without any change to the O/S.

Seperates generic I/O operations from device specific operations.

Translation of User Requests

Write (fd, buf, 20)	User
Write data in buf to logical block 5000	Kernel
Transfer 512 bytes held at addr 235400, to Disk 2, Cyl 12, Head 10, Sector 5	Driver
Hardware	

Types of Device Drivers

Block Device Driver

Transfers data in blocks.

I/O requests are presented to the driver in terms of a desired block number.

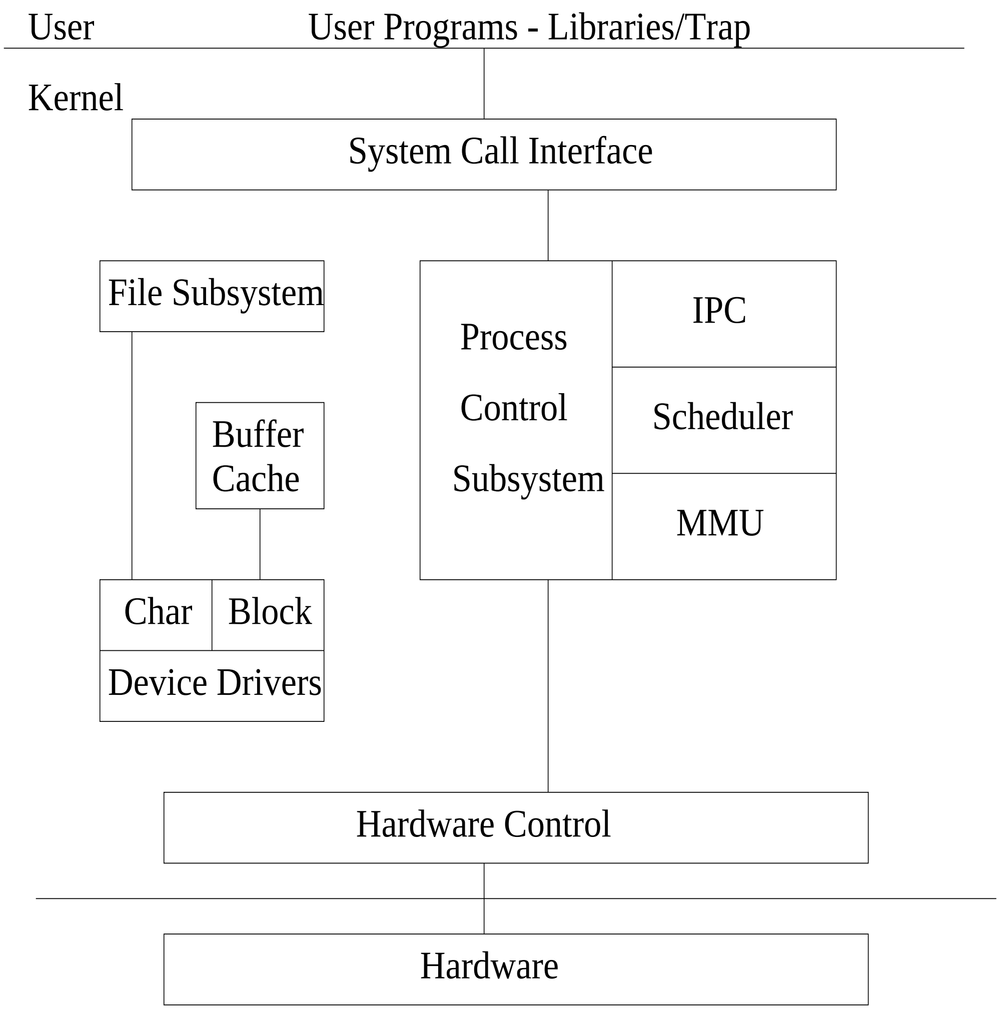
Buffering done by the kernel.

Character Device Driver

Transfers streams of data of varying size.

Buffering done by the driver.

Position of Device Drivers in the System



Driver Entry Points

Functional interface between drivers and Kernel.

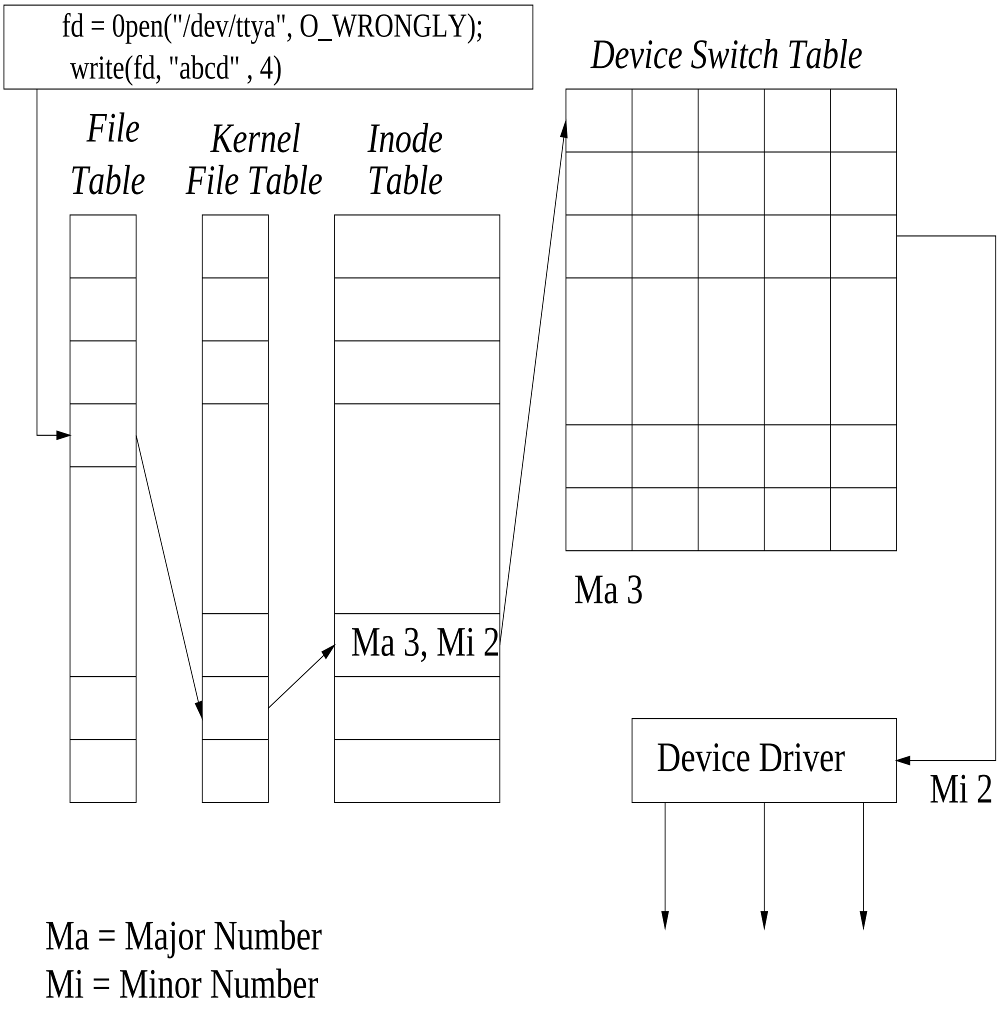
Routines that are called by the kernel when a device specific function is requested.

Type of device decides what entry points pertain to it.

Entry points for block devices include *xxx_open, xxx_close, xxx_strategy* etc.

Entry points for character devices include *xxx_open, xxx_close, xxx_read, xxx_write* etc.

Control path from System call to Driver



Switch Tables

Interface between deiver and file subsystem.

Direct the system call to appropriate driver calls.

Seperate switch tables for character and block devices.

Switch Tables (contd...)

Block Device Switch Table

Entry	Open	· ·	Close	Strategy
0	*hdopen*	· ·	*hdclose*	*hdstrategy*
· ·	· ·	· ·	· ·	· ·
n	*gdopen*	· ·	*gdclose*	*gdstrategy*

Character Device Switch Table

Entry	Open	Close	· ·	read	write	iocti
0	*lpopen*	*lpclose*	· ·	*lpread*	*lpwrite*	*lpiocti*
· ·	· ·	· ·	· ·	· ·	· ·	· ·
n	*ttyopen*	*ttyclose*	· ·	*ttyread*	*ttywrite*	*ttyiocti*

```
/*************************************
* A sample Pseudo driver for Linux.
*
* The implemented functions are mymem_open,
  mymem_read, mymem_write, mymem_release.

* The related files that are to be changed
  are mem.c, Makefile.

* This is infact is a pseudo driver for
  memory with Major device number 21.
*/
```

```
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/tty.h>
#include <linux/signal.h>
#include <linux/errno.h>
```

```
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#include <asm/irq.h>
```

```
/* The memory area for the pseudo mem*/
static int mymem_data_size=128;
static char mymem_data[128]="Yes
this is the pseudo device driver\n";
```

```
struct file_operations mymem_fops=
{
 NULL,    /* mymem_seek */
 mymem_read,  /* mymem_read */
 mymem_write,  /* mymem_write */
 NULL,     /* mymem_readdir */
 NULL,     /* mymem_select */
 NULL,     /* mymem_ioctl */
 NULL,    /* mymem_mmap */
 mymem_open,  /* mymem_open */
 mymem_release  /* mymem_release */
};
```

```
unsigned long mymem_init(unsigned
        long kmem_start)
{
 printk("mymem installed.\n");
 if (register_chrdev(21,"mymem",
              \&mymem_fops))
   printk("Memdrv error: Cannot register to
            major device 21!\n");
 return kmem_start;
}
```

```
static int mymem_open(struct inode *inode,
         struct file *file)
{
 printk("mymem opened.\n");
 return 0;
}

static mymem_release(struct inode *inode,
          struct file *file)
{
 printk("mymem released.\n");
}
```

```
static int mymem_read(struct inode * inode,
struct file *file, char *buffer, int count)
{
 int offset;

 printk("reading %d bytes\n",count);
 if (count<=0) return -EINVAL;
 for (offset=0; offset<count; offset++)
   put_fs_byte(mymem_data[offset %
    (sizeof(mymem_data)-1)], buffer+offset);
 /*put from kernel address space to user
   address space */
 return offset;
}
```

```
static int mymem_write(struct inode * inode,
       struct file * file,
       char * buffer, int count)
{
 printk("Write %d bytes\n",count);
 printk("Memdata size %d bytes\n",count);
 if (count>127) return -ENOMEM;
 if ((!mymem_data_size) || (count<=0))
            return -EINVAL;
 memcpy_fromfs((void *)mymem_data,
      (void *)buffer, (unsigned long)count);
 mymem_data[127]=0;
 mymem_data_size = count;
 return count;
}
```

Device Driver Installation

1. Put *mymemdrv.c* in */root/linux/drivers/char* directory.
2. Include *mymemdrv.c* and *mymemdrv.o* in the *.OBJ* and *.SRC* lists in the *Makefile.*
3. Insert a line *mem_start=mymem_init(mem_start)* in the *mem.c* file in */root/linux/drivers/char* directory.
4. Compile the kernel from the above directory using *make* command.
5. Copy the newly created *zImage* file to the root directory with the name *vmlinux.new* (say).

6. A similar entry needs to be made in the *LILO.CONF* file in the */etc* directory.
7. Execute the command *lilo* whcih puts the newly compiled kernel in a way so that lilo loader can recognize it in the next bootup.
8. Reboot the system with the new kernel.
9. Give the commands *mknod /dev/mymem c 21 0* and *chmod 666 /dev/mymem*. Now the user can read from and write to */dev/mymem* like an ordinary file.

References

- *Silverschatz and Galvin*, Operating System Concepts.
- *M. J. Bach*, The Design of the UNIX Operating System.
- *J.I.Egat and T.J.Teixeira*, Writing a UNIX Device Driver.

UNIX SYSTEM ADMINISTRATION

Basic Tasks:

- Creation of new login and deletion of old ones
- Maintainance of file store security
- Monitoring of user resource usage
- Maintainance of documentation
- Machine performance considerations

In most of this discussion "ROOT" previlege or superuser permission is assumed.

Boot Process

Init daemon runs as the last step in the boot process. (/etc/init)

Daemons are background processes created at system boot and normally terminated at system shutdown. Usually not attached to any terminal.

Typical sequence of events started by init.
(all programs in /etc)

1. Runs *rc. boot.*
2. Runs *fsck* to check for any file system inconsistency.
3. If fsck does not report problem, revokes *rc.*
4. *rc* mounts file systems on local disks, passes control to *rc.local.*

5. *rc.local* starts daemons on the local m/c that handles NFS, NIS and mail requests. Finally, it returns control to *rc*.

6. *rc* starts up standard daemons, preserves editor files and initiates other system activities like system accounting.

7. When *rc* finishes running, the system comes multiuser.

Exercise: Examine the contents of rc* files in /etc.

System Shut down

Commands to shutdown a system in emergency

/sbin/shutdown

/sbin/halt

/sbin/reboot

/sbin/fastboot

Superuser can run them.

Differences

"shutdown"

most versatile.

- used for shutting down a system with multiple users.
- message broadcast facility
- inhibits logins and aumatically executes *sync.*
- brings system to single user mode at specified time.
- has flags to request halt and reboot

"halt"

for immediate shutdown; no warning, no delay.

- no facility for leaving messages and specifying time
- for diskless and dataless clients
- writes out info and halts processor

“reboot”

- on m/c with single user
- on server in single user mode
- after syncing initiates multiuser reboot; fsck runs

“fastboot/fasthalt”

- Shell scripts which can call “reboot/halt”

File System Maintenance

- Backing up and restoring file systems
- Checking disk usage when file systems are full
- Setting up a disk quota system

Typical default filesystems are: / (bootable image), */var/spool/mail*, */usr* (unix commands) and */home* (user directories and files).

File systems reside on partitions.
Check using *df* or *mount* command.
Also look at */etc/fstab* file.

Different kind of files

- ascii text: editable file
- commands text: shell commands scripts
- *.<lang>: program files (*lang* is is the programming language used.
- *.o: object files (linkable).
- *.lib: binary library files (linkable).

Some important directories

/bin and /usr/bin

Public commands stored in these directories.

Command interpreter looks in these directories.

Commands could be binary files or shell scripts. Need execute permission. Later needs both read and execute permissions.

Sequence of search for command defined by the PATH string.

$ echo $PATH

A typical 'PATH'
:$HOME/bin:/bin:/usr/bin

/tmp and /usr/tmp

(temporary files)

Editors, Compilers etc. need temporary files for working space.
They last only for the duration of the process.

Reboot sequence normally clears these files.

However, some recovery files may need to be kept.

If a shell script needs temporary files, use process number.

e.g. TMPFILE = /tmp/ef$$
trap "rm $TMPFILE; exit" 0 1 2 3 15

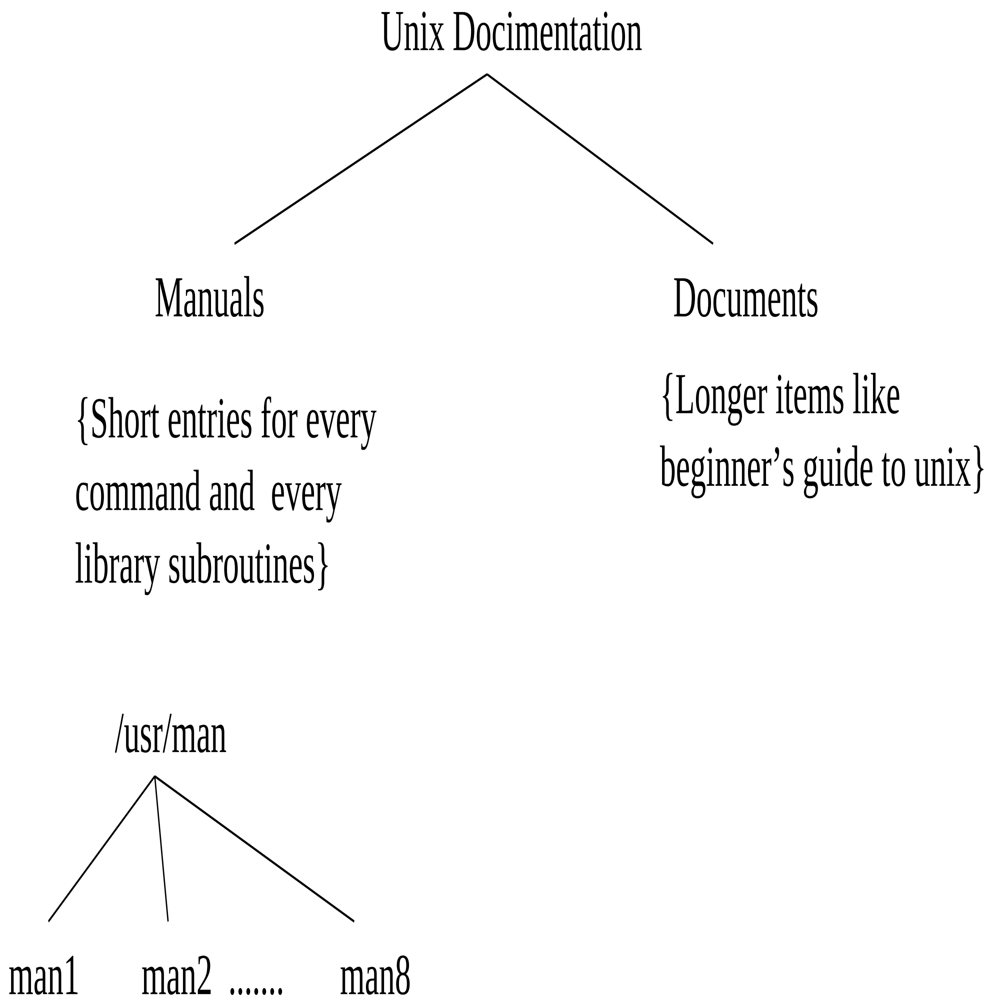
Exercise: Do editing with 'vi' and examine the temporary files.

/lib and /usr/lib

(System libraries)

1. Subroutine libraries to be linked.
2. Programs invoked by commands but not seen by users, e.g., various passes of compiler.
3. Datafiles referenced by commands, e.g., fonts used by typesetting commands.

Manual


Unix Docimentation
Manuals
Documents
{Short entries for every command and every library subroutines}
{Longer items like beginner's guide to unix}
/usr/man
man1
man2 ....
man8

man1: Ordinary Shell commands (in /bin & /usr/bin)
man2: Basic System Calls
man3: Library subroutine calls
man4: Special files like devices and their details
man5: File formats
man6: Games
man7: Word processions, troff
man8: Management commands

/usr/doc

(documentation)

Documents associated with UNIX are stored under this.

Smaller Documentations in files (e.g. */usr/doc/bc*).

bigger documents in files under directories (e.g. */usr/doc/beginners* contains files for *beginners guide*).

The file *run* in */usr/doc* includes details for the full production of each document (e.g. *run adb* produces adb document).

Exercise: Use *run* to read the document of your choice.

/usr/adm

(administration)

Contains various files to do with administrative operations.

e.g., */usr/adm/wtmp* maintains a record of all logins and logouts.

/usr/adm/sulog keeps details of every call to *su*.

Ex: examine the contents of above files.

/usr/dict

(word processing)

Contains various files concerned with text handling commands such as *spell*.

e.g., */usr/dict/words*: used by spell.

/usr/dict/roget: roget thesaurus commands for suggesting alternative words.

/usr/games

(games)

Houses game programmes.

/usr/include

(include files)

As part of C compiler, C preprocessor allows lines of the form
include “myfile”
and
include <stdio.h>

Former searches present directory, later the /usr/include/directory.

There is also */usr/include/sys* containing files with description of structures for system calls.

/usr/spool

(areas for spooling)

Contains secure temporary files to be retained across times when the system is taken down and rebooted.

Standard subdirectories:

/usr/spool/lpd: when user calls *lpr*, it stores the output in the temporary *spool* file. *lpd daemon* spools the stored output to the printer and removes the spool files after printing.

/usr/spool/lpd/tmp: stores output being piped to the *lpr*.

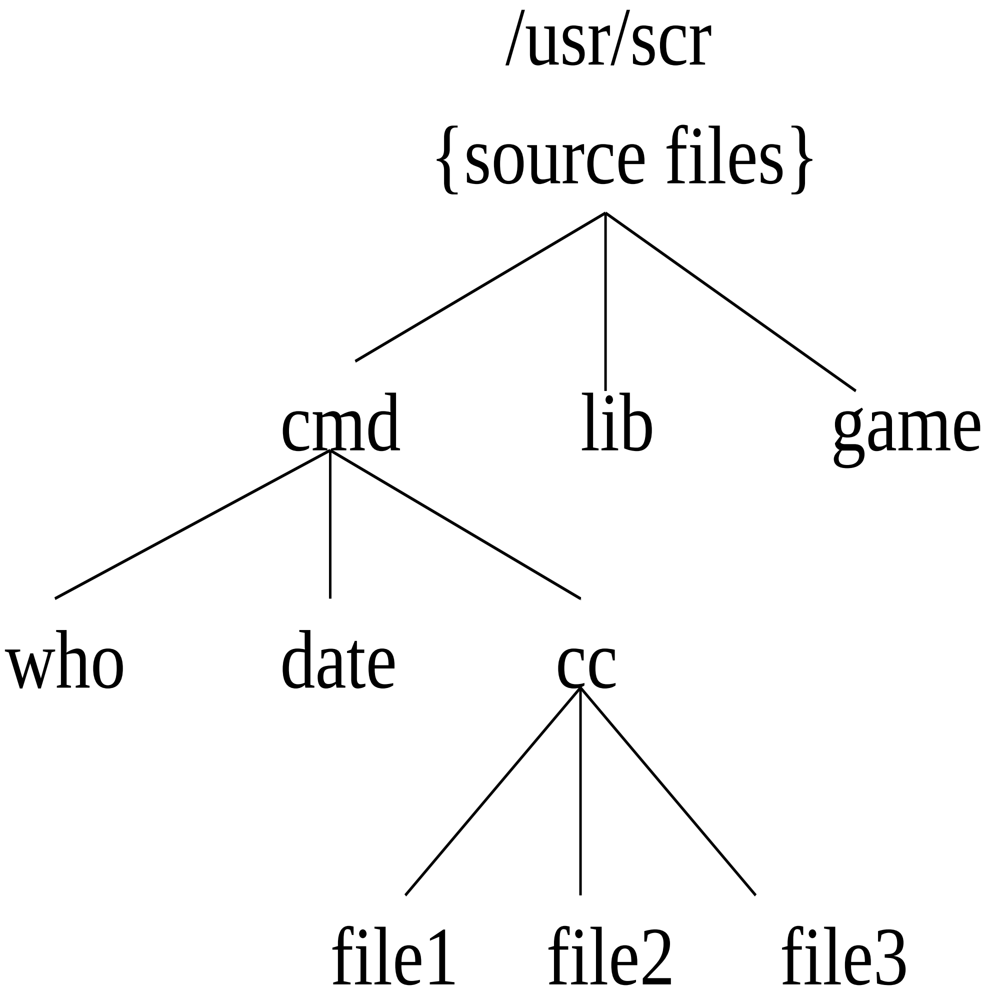
/usr/spool/uucp: temporary files used by *uucico* for intermachine spooled file transfer.

/usr/spool/mail: stores mails.

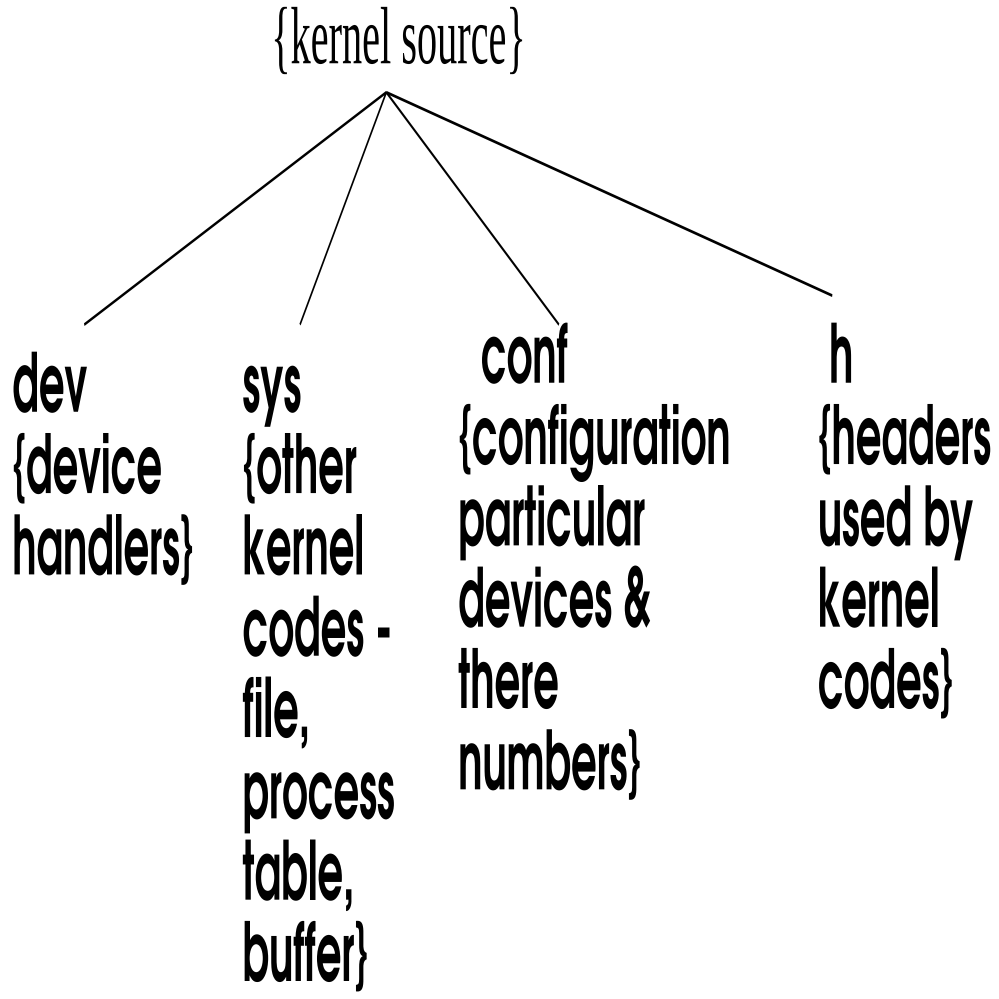
/usr/spool/at

at command enables the user to issue a command which is to be executed at a given future date and time and stores a file in this directory for each command to be executed. The name of each file indicates the time when it is to be executed.

The command relies on the program *atrun* being started regularly by *cron*, which looks to see if there is a command ready for execution.


/usr/scr
{source files}
cmd
lib
game
who
date
cc
file1
file2
file3

/usr/sys or /usr/src/sys



Ex: Examines the differences between contents of */usr/sys/dev* and */usr/sys/conf*.

/etc

(administration and management)

Some important files in this are:

accton: file to setup shell accounting.

cron: program to do process schedule by time table.

fsck: program for file system consistency check.

getty: program forming part of login process.

init or *inittab*: Program for controlling multiprogramming (mandatory).

mkfs: program for making file system.

mknod: program to make a new device.

motd: file for message of the day, listed during login.

mount: program to mount and enquire about mounted file system.

mtab: file with list of mounted file systems.

passwd: main file of user details.

rc: shell scripts to be executed when going multiuser at the end of boot-up process.

shutdown: shell script for close down.

termcap: text file of terminal capabilities, edited when new terminal is added.

ttytype: text file specifying achieve terminal lines and their baud rates.

umount: program to unmount files.

update: program to flush memory fo disc.

utmp: file containing list of logged in users.

wall: program to write to all users.

Ex: Thorough study of /etc.

/dev

(devices)

all special files representing devices in the system.

/mnt: temporary mounting of file systems at this point

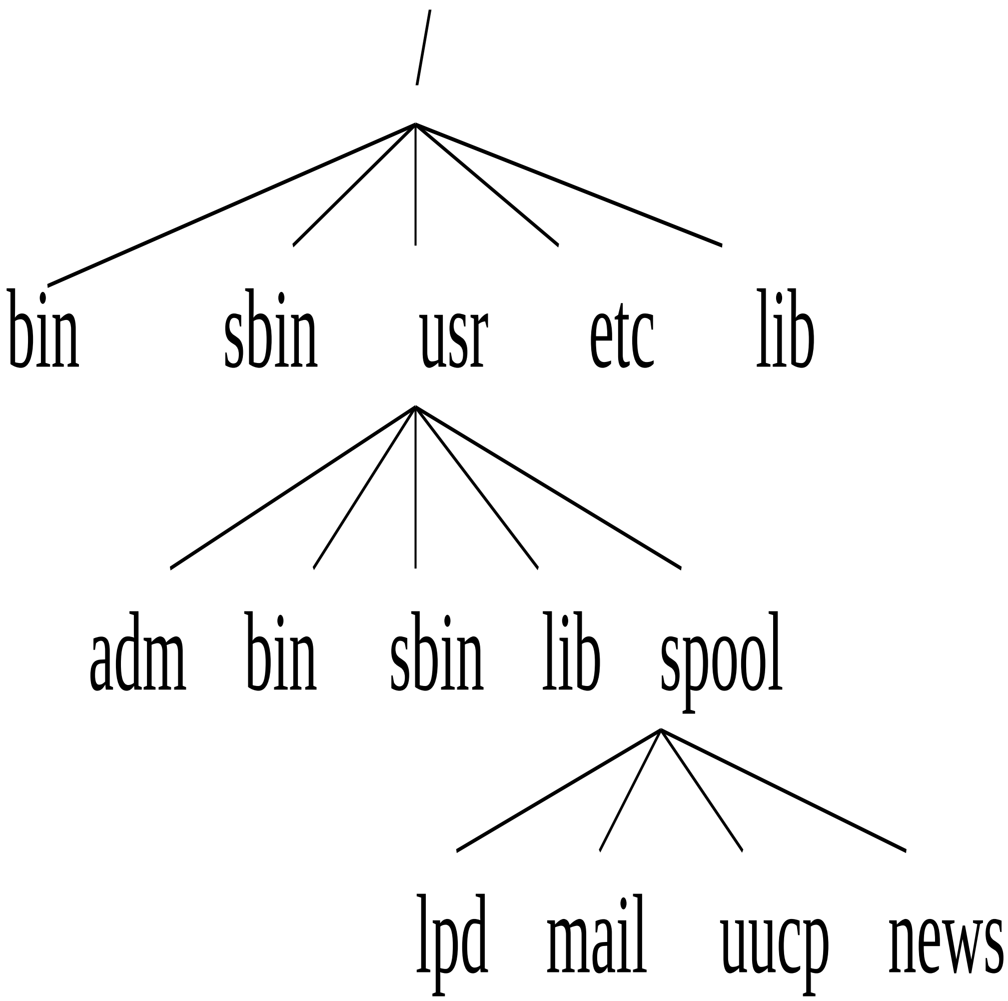
/etc/mount */dev/f2* */mnt*

lost+found: in it *fsck* stores files suspected of being corrupted.

Managing File Systems

Tree structured UNIX file system

A segment of directory tree



Mounting and unmounting of file system

mount <device> <mountpoint>

Device contains a file system (made through 'mkfs') and mountpoint is a directory in the tree. Mount commands can be given only by the superuser.

Some important command line arguments for mount are:

- -f fakes mounting; check for any error message
- -v verbose
- -w mount with read and write
- -r read only
- -n no write in */etc/mtab*
- -a mount all filesystems in */etc/fstab*
- -t type

Examples of *Mount*

- *mount /dev/hdb3 /mnt*
  mount /dev/hdb3 under /mnt.
- *mount -r -t iso9660 /dev/sr0 /mnt*
  SCSI CD-ROM drive /dev/sr0
  which is read only and of ISO9660 fileformat.
- *mount -vat nfs*
  mount all file systems which are NFS listed in /etc/fstab.

Note: *mount -vf <dev> <mountpt>* is a very useful diagnostic tool.

/etc/fstab file

Controls boot time mounting and shut down time unmounting.

Lists the file systems to be mounted one file per line. Fields in each line separated by tabs.

A sample:

# device	directory	type	options
/dev/hda1	/	ext2	defaults
/dev/hda2	/usr	ext2	defaults
/dev/hda3	none	swap	sw
/dev/sda	/dosc	msdos	defaults
/proc	/proc	proc	none

Type *ext2* means local file system with normal file operations.

Default means *read/write enabled, blk sp edvice, asynchronous file I/O, ordinary users cannot mount* and so on.

/dev/hda3 is the swap partition with no mount point. Mentioned in *fstab* to let the system know where it is physically.

/dev/sda is the MS-DOS partition at */dosc* mount point.

Unmounting is possible only if device is NOT BUSY.

Command syntax:
umount <device> | <mountpoint>
umount -a
umount -t fstype

in the first specify only the device or the mount pt.
-a unmounts all the devices appearing in /etc/fstab.
-t unmounts all the filesystems of specified type.

Maintaining File System Consistency

Check filesystems periodically for damaged or corrupted files.

Normal file systems (*ext2*) have *clean bit*. Synched and umounted file systems have *clean bit* set. In this case the file system is not checked for consistency.

fsck filesys

options:

- -A check all as in fstab
- -v verbose
- -t fstype fstype
- -a automatic repair
- -l list all names in the fs
- -s list superblock

To fsck the /, need to boot from floppy.

After any repair in fs, system should be rebooted with *shutdown -r*.

Creating and formatting partitions before the filesystem is made is done with *fdisk* command.

It is an interactive utility and creates partitions whose names, sizes and types are given by the user.

mkfs builds the file system.

mkfs [-V][-t fstype][-c][-l filenm] device [blocks]

> -c to check for bad blocks before making fs
> -l to read bad blocks from filenm

Swap partitions and files

mkswap [-c] device blocks

creates swap partitions of “blocks” size on “device”. -c checks for bad blocks

swapon filesys

activates swap area

swapon -a

activities all swap areas in /etc/fstab.

One could create swap file to expand swap space using following commands:

*dd if=/dev/zero of=/swap *
bs = 1024 count = 10240
mkswap /swap 10240
etc/sync
swapon /swap
swapon /swap
swapoff /swap
rm /swap

Printing Service

- Basic tasks:
  - Configure Printer
  - Send file to Printer
  - check printer status
  - cancel printer job
  - cancel printer job
  - Trouble shooting
- Simplest way to Print
  ls > /dev/lp0

Basic mechanism for printing

Spooling the data.

For each printer, spool area defined. One file in the spool area per job.

Printer *daemon* (backgound process) constantly scans spool area for new files to print.

After the data is sent to the printer, it is despooled.

First in first out strategy for printing.

Spool area is the print queue.

/etc/printcap file

contains info on

physical device to use
spool area to look in
remote m/c
printer for remote m/c
etc.,.

Important programs for printing

- /usr/bin/lpr
  - – submit job
- /usr/bin/lpq
- /usr/bin/lpc
  - – inspect queue and printer status
- /usr/bin/lprm
  - – cancel
- /usr/sbin/lpd
  - – daemon

- *lpd:* daemon handles print jobs. *lpd -l* creates a log file and logs all print requests.
- *lpr:* queues a print job. If no file specified STDIN assumed.
- *lpq:* shows the contents of the spool directory for a given printer. Displays job IDs. Shows 'rank' for each job. 'active' means actually getting printed.
- *lprm:* removes job from queue.
- *lpc:* checks printer status and controls some aspects of their use.
  lpc down myprinter
  lpc enable yourprinter
  lpc topq 37

Some commands to lpc

stop <printer>

start <printer>

exit

status <printer>

/usr/spool/lpd or */var/spool/lpd* : main spool area. under this create directories for each printer preferably with same name.

/etc/printcap file

sample entry with two aliases
myprinter/laserwriter:

lp is the device to print to
:lp = /dev/lp0:
sd is the spool directory
:sd = /usr/spool/lpd/myprinter:

Some important fields in printcap

:lf	error log file
:if	input filter
:mx	max. allowable print job size
:sh	suppress/obtain header (banner)
:sf	suppress/obtain end-of form feed

Steps of setting up a printer

1. Check permissions and locations of *lpm, lpc, lpq, lpd.*

2. Create spool directory for the printer (Call it, say, *myprinter*). Use the following commands:

   mkdir /usr/spool/lpd
   mkdir /usr/spool/lpd/myprinter
   chown root.daemon /usr/spool/lpd
   *   /usr/spool/lpd/myprinter*
   chmod uq=rwx, 0=rx /usr/spool/lpd
   *   /usr/spool/lpd/myprinter*

3. In */usr/spool/lpd/myprinter* create necessary files
   cd /usr/spool/lpd/myprinter
   touch .seq errs status lock
   chown root.daemon .seq errs status loc

chmod uq=rwx, o=rx /usr/spool/lpd /usr/spool/lpd/myprinter

4. Create the shell script input_filter in */usr/spool/lpd/myprinter*:
   cd /usr/spool/lpd/myprinter
   chmod ug=rwx, o=rx input_filter

5. Create the file */etc/printcap* if it does not already exist. Remove all entries in it and add the appropriate printcap entry. File should be owned by root and read only to everyone else.

6. Edit *rc.local*. Add */etc/lpd* to the end. Presently just run
   lpd.

7. Do a test print by typing
   ls -l | lpr -Pmyprinter

8. Look in */tmp* for a file called *testlp.out*.

9. Edit */etc/printcap*:

   - In the first printer entry, change both occurences of *myprinter* to *testlp* in the first line only.
   - In the second entry, change */dev/null* to */dev/lp0*
   - In the second entry, remove the *if* line completely.

   Now, copy the *myprinter* entry so that you have two identical entries in the file.

10. Reboot the system, else kill *lpd* and restart it.

11. Do a test print again.

Managing user Accounts

Every user has unique login and password

- Adding user:
  login-name: encrypted passwd: user-id:
  group-id: userinfo: login-dir: login-shell

- Adduser and useradd
  Former is interactive. Latter provides options

  *useradd -d /users/sundar *
  *-s /bin -c "sundar R. *
  accounts deptt" sundar

  copies .profile etc. from /etc/skel
  (default)

- other options:
  -u UID
  -g group
  -k dir (copy dir for initialisation)

Setting user passwd

$*passwd sundar*
Enter new passwd: <new passwd>
Re-type new password: <-do->

Encrypted passwd goes into passwd file.
Password should be 6-8 char long and must contain small/capital letters, numerals and punctuation marks.

Removing User:

- Remove login ability: put * in password field.
- Remove user entry from password file, but keep the files. Use *chown* and *mv*.
- Remove password entry and files:
  $ *find* $<$ *users_home_directory* $>$ *-exec rm* {}*;*
  $ *rmdir* $<$ *users_home_dir* $>$

Groups

Each user of a group has info about groups kept in /etc/group.

e.g.

sales::21: sundar, milind, sekhar

group-id members

a group can be added by

groupadd -g 21 sales

Deletion by

groupdel sales

Backup and Restoration

Timely backup needed to prevent loss from accidental removal of data.

- Full or incremental backup
- File systems which should be backed up
- Type of backup media
- Effect of backup on users
- Commands to use for backup (*tar*, ...)
- Documentation of backed up files

Types of Backup

- Incremental backup: Files that have changed since last backup
- Complete backup: All files in the file system

One may have full backup once a month and incremental backup daily or weekly.

tar (tape archive) is

- Simple to use
- Archives can be read in any UNIX system
- On its own can backup only the complete file system (How to do incremental backup ?)

Common tar options

c	creates archive
x	extract file
f	<name> creates archive on or read archive from <name>
Z	compress/decompress tar archive
z	compress/decompress with gzip
M	creates multi volume tape backup
t	creates index of all files and lists on stdout
v	verbose mode

e.g. :

$ find /home -mtime -1 -type f -print > bkuplist

$ tar cvfzM /dev/fd0 ‘cat bkuplist‘

backup all files modified on the last day put

30 01 * * * tar cvfz /dev/fd0/home > homeindex

In the cron file of root to backup up home everyday at 1.30 AM.

$ tar cvf /home.backup.tar/home/dave

Creates the file /home/backup.tar which contains a backup of the directory /home/dave/ and all files and directories below it.

When creating a tar file of all the files in the directory, it is a good practice to specify a different location for the tar file than the current directory. This avoids confusion and recursive *tar*ing.

cpio

General purpose command for copying file archives.

Backup with -o option
Restore with -i option

- can backup any set of files
- can backup special files
- more efficient in storage than *tar*
- skips bad sector when restoring
- portable archive

Options are:

-o copy out on stdout
-B blocks i/o at 5120 bytes/record
-i copy in from stdin
-t creates table of content of input

Examples of cpio

$ ls /home | cpio -o > /dev/fd0

copies files in /home to floppy

$ cpio -it < /dev/fd0 > bkindex

extracts files on floppy and creates index on 'bkindex'

$ echo "/home/dave/notes.txt" | cpio -i < /dev/fd0

restore from floppy

Security issues

Some don'ts

- Don't leave a system, tape drives, disk drives, terminals, work stations unattended for prolonged time.
- Don't leave the console or other terminal device logged on as root and unattended.
- Don't keep sensitive information on systems having modem or network connections.
- Keep backups in secure area.

Password security

Users want simple, easy to remember passwords and do not want to change them.

Change root password often, choose it wisely, only two people should know it.

Never use a password that is a dictionary word. Name, spouse's name, etc. should not be used.

Good strategy

- Use numeral, alphabet, punctuation symbol in the password.
- Choose two short words and join them with punctuation symbol
  e.g. milk@chair
- First letters of a common phrase
  e.g. "honesty is the best policy" - hitbp

PASSWORD SHOULD NEVER BE WRITTEN DOWN. ALSO NEVER SEND PASSWORD BY EMAIL

Login security

- Many hackers succeed by simply finding an account that does not have a password. Disable such accounts. Change password field into *.
- Remove unused user accounts
userdel -r <name>

- Installation of OS creates many default accounts like *root*, *bin*, etc.. They initially do not have a password. Immediately create password for them or lock the password field.
- Guest accounts can pose security threat.
- Disallow several people- supposedly forming a group- to have same login id. Use group instead.

File security

- permissions
- Set user id on execution. The password program. it must execute with root previlleges to edit the */etc/password* file. */bin/password* has SU id bit on.

  An interesting point:
  If you can get su previllege temporarily then you can become the root anytime you want by doing the following:

cp /bin/sh /home/$LOGNAME
(copy the shell)

chmod u+s /home/$LOGNAME/sh
(set user id on execution)

Next time you login *exec* the copied shell and you become the root.

- Record use of *su* command in /var/adm/syslog. This file should be examined periodically.

Using scheduling commands

Useful for running tasks at times when the system is not too busy, e.g., query database to answer external services.

at

Runs commands at specified times

$ at 1.23
lp /usr/sales/reports/*
echo "files printed" | mailx boss
$\hat{\text{D}}$

response :

job 756603300 at Sat Dec 21 01:23:00 1997

can cancel using the job number.

options

at hh:mm
at hh:mm month day year
at -l list scheduled jobs
at -r job_id cancel job_id

Batch

Run long tasks at times *convenient* to the system (unlike *at*)

$ batch
sort /usr/sales/reports/* | lp
echo "files printed" | mailx boss
D̂

response:

job 7789001234.b at Sat Dec 21 11:43:09 1996

cron and crontab

at and *batch* schedule commands on a one time basis.

To schedule commands on a regular basis use *cron*.

Specify times and dates in *crontab* file.

cron

Started only once when the system is booted.

Individual users should not have permission to run *cron.*

cron should be executed by */etc/rc.*

cron's activities

- Once started, *cron* checks queues for *at* jobs to run.
- Sees whether users or root have scheduled jobs using *crontab* file.
- If there is nothing to do *cron* goes to sleep.
- Wakes up every minute to check whether there are commands to run.

Each user has one *crontab* file started in */usr/spool/cron/crontabs* directory. File is given user's name. Install *crontab* file using the *crontab* command.

For a user to be able to use *crontab* command, the user must be listed in */etc/cron.d/cron.allow* file.

root has to add new user to the *cron.allow* file with a text editor.

Example

Suppose user name is *bhat*. Use text editor to create a file called *mycron*. Install this by typing

$ crontab mycron

This creates or overwrites */usr/spool/cron/crontabs/bhat*.

Each line in *crontab* file contains a time pattern and a command.

mm hh dd month day_of_week command

30 07 * * 01 sort /sales/weekly | mailx abc

specifies minute as 30, hour as 07, all days of the month with *, all months of the year with *, day_of_week as 01 (Monday).

This achieves sorting of /sales/weekly and mailing the output to abc. Any problem or error is mailed to the user who created the *crontab* file.

* 09,11,14,16 10 03,09 01,02,04 stkquote

at 9 am, 11 am, 2 pm, 4 pm, on Mondays, Tuesdays and Thursdays of every week and definitively on 10th of March and September.

Time field options

minute	0-59
hour	0-23
day of month	1-31
month of year	1-12
day of week	0-6 (0 - Sunday)

crontab options

-e edits the contents of *crontab* file using the editor defined in the EDITOR variable.

-r removes the current *crontab* file from *crontab* directory.

-l lists contents of current *crontab* file.

Always the *crontab* file for the user is /usr/spool/cron/crontabs/<logname>

Some useful commands

find

Recursive search through directory structure to find files matching criterion like name, owner, size, time of last access, time of last modification.

Action too can be specified.

name:
$ find <dir> -name core -print

wild card in name:
$ find <dir> -name *.old -print

last access:
$ find <dir> -name *.old -atime +7 -print

logical operation:
$ find <dir> -name *.old -a -atime +7 -print
(+ means *older than*).

action:

$ find /tmp -atime +2 -exec rm {} \;

(Note: Blank after *rm* and all the paraphranelia after this are must).

$ find /usr/news -atime +30 -ok rm {} \;

(*ok* is for interactive operation).

exercise

Read up *find* from manual and try all its options.

ps

Gives the process status information.

$ ps -el

F	S	UID	PID	PPID
3	S	0	0	0
C	PRI	NI	ADDR	SZ
0	0	20	2a40	2
WCHAW	TTY	TIME	CMD	
47472	?	0:01	swapper	
F	S	UID	PID	PPID
1	R	201	51	36
C	PRI	NI	ADDR	SZ
17	58	20	3c00	6
WCHAN	TTY	TIME	CMD	
	04	4.09	yes	

F	=	process flags (3 means bits 0 and 1 are on; system process in memory)
S	=	process state (R: running, S: sleeping, W: waiting, T: stopped, Z: terminated)
UID	=	user identification (0: root)
PID	=	process id (ppid = parent process id)
C	=	CPU utilization in %
PRI	=	priority (lower value better treatment)
NI	=	nice value (can be increased by user, decreased by super user only)

ADDR	gives address in memory or hard disk
SZ	gives size in blocks
WCHAN	sleep/wakeup address
TTY	terminal (? for daemon)
TIME	execution time
CMD	the command being executed

make

Reads the specification of how the components of a program depend upon each other and how to process them to create an up-to-date version of the program.

$ cat makefile

```
hoc1 : hoc.o
        cc hoc.o -o hoc1
```

$ make

hoc1 will be generated. An immediate *make* command after this will report *hoc1 up-to-date.*

$cat makefile

```
YFLAGS = -d
OBJS = hoc.o init.o math.o symbol.o

hoc3 : $(OBJS)
	cc $(OBJS) -lm -o hoc3

hoc.o : hoc.h
init.o symbol.o : hoc.h y.tab.h

pr :
	@pr hoc.y hoc.h init.c math.c symbol.c

# the @ symbol suppresses displaing the
command when it is executed

clean :
	rm -f $(OBJS) y.tab.[ch]
```

To see what *make* does without running it :
$ make -n

To force the file times into a consistent state:
$ make -t

An empty dependency rule means the item is *Never up-to-date* and so the do action will always be done.

If the commands are in a different file other than the *makefile* then:
$ make -f $< filename >$

Now we show a typical makefile:
$ cat makefile

```
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc4 : $(OBJS)
        cc $(OBJS) -lm -o hoc4

hoc.o code.o init.o symbol.o : hoc.h
code.o init.o symbol.o : x.tab.h
```

```
x.tab.h : y.tab.h
        cmp -s x.tab.h y.tab.h ||
                cp y.tab.h x.tab.h

pr : hoc.y hoc.h code.c init.c math.c symbol.c
    @ pr $?
    @ touch pr

clean :
        rm -f $(OBJS) [xy].tab.[ch]
```

touch command is used to make a file *not-upto-date.*

$ touch hoc.y

$ make

yacc -d hoc.y
conflicts : 1 shift / reduce
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cc hoc.o code.o init.o math.o symbol.o -lm -o hoc4

$ make -n pr

pr hoc.y
touch pr

-n option displays the commands without executing them.

SHELL programming

- Shell as command interpreter
- Shell as programming language
  - for
  - if
  - while and until etc.
- Emphasis on examples

SHELL

- Command Interpreter
  - Filename Shorthand
  - I/O redirection
  - Pipe
  - Background processing
- Ex. lpr ch[123456789]*
  - what does it print
- ex. Differences among the following commands:
  - $ ls junk $ echo junk
  - $ ls / $ echo /
  - $ ls $ echo
  - $ ls * $ echo *
  - $ ls '*' $ echo '*'

SHELL as a programming language

There are variables, loops, decision making and so on.

```
# Program to compare old version
# with new version of chapters
for i in ch?.*
do
    echo $i
    diff old/$i $i
    echo # give blank line
done
```

Example of Features

Task: To mail to pb@cse all the files in your bin directory

```
$cd /usr/you/bin
$ for i in *
> do
> echo === This is file $i ===
> cat $i
> done | mail pb@cse
$
```

However pb gets a big bundled file which he has to unbundle using the editor!

could the mailed file contain mechanism of unbundling itself?

bundle **file**

```
$ cat bundle
# bundle: group files into distribution package
echo '# to unbundle, sh this file'
for i
do
    echo "echo $i 1> & 2"
    echo "cat >$i<< 'Endof $i' "
    cat $i
    echo "End of $i"
done
$
```

```
$bundle cx lc > junk

$cat junk

# To unbundle sh this file
echo cx 1 > & 2
cat > cx << 'End of cx'
chmod +x $*
End of cx
echo lc 1> & 2
cat >lc << 'End of lc'
#lc: count the number of lines in files
wc -l $*
End of lc
```

```
$ mkdir test

$ cd test

$ sh ../junk
cx
ls

$ ls
cx
lc
```

```
$ cat cx
chmod +x $*
$ cat lc
# lc: count number of lines in files
wc -l $*
$cd ..
$ rm junk test/* ; rmdir test
$pwd
/usr/you/bin
$ bundle * | mail pb@cse
```

Case

$cal

usage: cal [month] year

$cal october 1983

Bad Argument

$cal 10 1983

```
      October 1983

S  M  Tu  W  Th  F  S
                    1
2  3   4  5   6  7  8
.....................
....................
...................
```

cal **program**

- Recognize month by name
- Behave as such with two arguments but convert month name to number
- print the calendar for month or year for one argument as appropriate.
- print the calendar for current month and year in case of no argument.

$ cat mycal

```
\# cal: nicer interface to /usr/bin/cal
case $# in
0) set ‘date‘; m=$2; y= $6;;
1) m=$1; set ‘date‘; y=$6;;
*) m=$1; y=$2;;
esac

case $m in
jan* | Jan* ) m=1;;
feb* | Feb* ) m=2;;
.
dec* | Dec* ) m=12;;
[1-9]| 10 | 11 | 12 ) ;; #numeric month
* ) y=$m; m= "";; #plain year
esac
/usr/bin/cal $m $y
```

$ date

sat oct 1 06:05:18 EDT 1983

Condition Checking

```
if <condition command>
then
  <commands if condition true>
else
  <commands if condition false>
fi
```

Example Task:

We would like to run
which <command>

to tell us where in the PATH the command exists

Eg..
$ which troff
/usr/bin/troff

Which **Shell script**

```
$ cat which

# which cmd
case $# in
0) echo ’usage: which command’ 1 >& 2
   exit 2
esac
for i in ‘echo $PATH | sed ’s/^:/.:/
        s/::/:.:/g  s/:$/:./  s/:/ /g ’‘
do
   if test -f $i/$1
   then
       echo $i/S1
       exit 0 # found
   fi
done
exit 1 # not found

$ $echo $PATH
:/usr/you/bin:/bin:/usr/bin
```

While and *Until*

Example task: To watch for a specific user to login.

$cat watchfor

```
# watchfor : watch for someone to login
PATH=/bin:/usr/bin
case $# in
0) echo 'usage: watch for persons' 1>&2.
   exit 1;;
esac
until who | egrep "$1"
do
    sleep 60
done
```

```
$ chmod +x watchfor
$ watchfor you
you tty0 oct 1 08:01
```

Continued

Task: *watch all people logging in and out*

$cat watchwho

```
# watchwho: watch who logs in and out.
PATH=/bin:/usr/bin
new=/tmp/wwho1.$$ #incorporate Process id
old=/tmp/wwho.2$$
> $old   # create empty file
while :    # loop forever
do
    who> $new
    diff $old $new
    mv $new $old
    sleep 60
done | awk '/>/ $1 = "in: "; Print
/</ $1 = "out: "; Print'
```

$

Interactiveness

Read Command

Example task: Pick from a set of arguments

$ cat pick

```
# pick select arguments
PATH=/bin:/usr/bin
for i   # for each argument
do
    echo -n $i? > /dev/tty # no newline
    read response
    case $response in
       y*) echo $i ;;
       q*) break;;
    esac
done < /dev/tty
```

Arithmetic Tests

Task: *send message to all users after ensuring that there is atleast one user more than the sender logged in.*

```
$cat warn
# warn users
if test `who | wc -l` -gt 1
then
	/etc/wall < warning # from warning file
fi
$
```

elif

Task: *Put some lines of 1st file to 2nd file which match a pattern.*

$cat putpattern

```
# put lines matching a pattern
if [ $# -ne 3 ]
then
    echo " you have not given 3 args"
    exit 3
elif grep "$1" $2 > $3 2 > /dev/null
then
    echo "pattern found- job over"
else
    echo "pattern not found- job over"
    rm $3
fi
$
```

Task: *Checking status of a file.*

$cat flstat

```
echo " Enter file name: \c "
read flname
if [ ! -f $ flname]
then
    echo "File non-existing"
elif [! -r $flname]
then
    echo "File not readable"
elif [! -w $flname]
then
    echo "file not writable"
else
    echo "file readable and writable"
fi
```

Task: *searching for a pattern in a file.*

```
echo "Enter the string to be searched :\c"
read pname
if [ -z "$pname"]
then
    echo "string not entered"; exit 1
fi
echo " enter filename: \c"
read filename
if [! -n "$flname"]
then
    echo " Filename not entered"; exit 2
fi
echo "search $pname in $flname?: \c"
read answer
if [ "$answer = "y"]
then
    grep "$pname" "flname" || echo "Not found"
else
   exit 3
fi
```

Laboratory Exercise

- Copy file1 to file2. If file2 exists then if file2 can be written into, then warn user and get his permission to proceed; else exit; else copy file.
- Say ‘good morning’ to user if time <12 else ‘good afternoon’ if time < 17, else say ‘good evening’