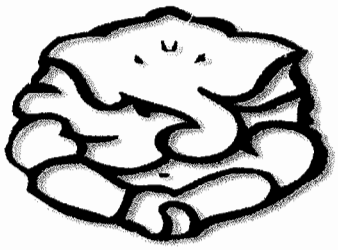


6TH CSE/ISE

# UNIX SYSTEM PROGRAMMING

ASHOK KUMAR K
VIVEKANANDA INSTITUTE OF TECHNOLOGY
MOB: 9742024066
e-MAIL: celestialcluster@gmail.com

S. K. L. N ENTERPRISES
Contact: 9886381393

UNIT 1

# INTRODUCTION

Syllabus

* UNIX and ANSI standards.
* The ANSI C standard.
* The ANSI / ISO C++ standards.
* Differences b/w ANSI C and C++.
* The POSIX standards.
* The POSIX.1FIPS standard.
* The X/OPEN standards.
* The POSIX APIs.
* The UNIX and POSIX Development environment.
* API common characteristics.

– 6 Hours.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

## THE ANSI C STANDARD

* In 1989, American National Standard Institute (ANSI) proposed C programming language standard X3.159-1989 to standardize the language constructs and libraries. This is termed as ANSI C standard. This attempts to unify the implementation of the C language supported on all computer system.

* The major differences between ANSI C and K&R C (Kernighan & Ritchie) are as follows

i.) Function prototyping
ii.) support of const & volatile data type qualifiers.
iii.) support wide characters & internationalization.
iv.) permits function pointers to be used without dereferencing.

### Function prototyping.

* ANSI C adopts C++ function prototype technique where function definition and declaration include function names, arguments' data type, and return value data types. This enables ANSI C compilers to check for function calls in user programs that pass invalid no. of arguments or incompatible argument data types.
These fix a major weakness of K&R C compilers: Invalid function calls in user programs often pass compilation but cause programs to crash when they are executed.

eg: unsigned long foo (char *fmt, double data)
{ /* body of foo */ }
external declaration of this function foo is
unsigned long foo (char *fmt, double data);

eg2: int printf (const char *fmt, ...); specifies variable no. of arguments.

## Support of const and volatile data type qualifiers.

* The const key word declares that some data cannot be changed.

eg: int printf (const char *fmt, ... ); declares a fmt argument that is of a const char * datatype, meaning that the function printf cannot modify data in any character array that is ~~present~~ passed as an actual argument value to fmt.

* Volatile keyword specifies that the values of some variables may change asynchronously, giving an hint to the compiler's optimization algorithm not to remove any "redundant" statements that involve "volatile" objects.

eg: char get_io()
```
{ volatile char *io_port = 0X7777;
  char ch = *io_port;  /* read first byte of data */
  ch = *io_port;       /* read second byte of data */
}
```

If ~~io~~ io_port variable is not declared to be volatile when the program is compiled, the compiler may ~~ele~~ eliminate second ch = *io_port statement, as it is considered redundant w.r.t previous statement.

* const & volatile datatype qualifiers are also supported in C++.

## Support wide characters and internationalization

* ANSI C supports internationalization by allowing c-program to use wide characters. Wide characters use more than one byte of storage per character.
* ANSI C defines the setlocale function, which allows users to specify the format of date, monetary, & real number representations.
For eg, most countries display the date in dd/mm/yyyy format, whereas US display it in mm/dd/yyyy format.
* function prototype of setlocale is

```
#include <locale.h>
char setlocale (int category, const char *locale);
```

* The category values specify what format (class (=) is to be changed. (All possible values for it are defined in (locale.h>)
* some possible values of category argument are;

| Category value | Effect on std C functions/macros. |
|---|---|
| LC_CTYPE | Affects behaviour of <ctype.h> macro. |
| LC_TIME | affects date & time format |
| LC_NUMERIC | affects number representation formats |
| LC_MONETARY | Affects monetary value format |
| LC_ALL | combines the effects of all above. |

Permit function pointers to be used without dereferencing.

* ANSI C specifies that a function pointer may be used like a function name. No dereference is needed when calling a function whose address is contained in the pointer.

eg: ~~void foo (double char, char *matt)~~

```
void foo (double xyz, const int * iptr);
void (*funcptr) (double, const int *) = foo;
```

the func. foo may be invoked by either directly calling foo or via the funcptr. Thus following two statements are functionally equivalent:

```
foo (12.78, "Hello world");
funcptr (12.178, "Hello world");
```

where as K&R C requires funcptr be dereferenced to call ~~equivalent above statement~~ foo.

ie `(*funcptr) (12.178, "Hello world");`

* Both ANSI C & K&R C function pointer uses are supported in C++.
* In addition, ANSI C also defines a set of cpp (c pre processor) symbols which may be used in user programs. These symbols are assigned actual values at compile time.

| cpp symbol | use. |
|---|---|
| -STDC- | Feature Test macro<br>value 1 ⇒ compiler is ANSI C conforming<br>0 ⇒ otherwise. |
| -LINE- | Evaluated to physical line number of a source file for which this symbol is reference |
| -FILE- | value is the filenumber of a module that contains this symbol. |
| -DATE- | value is the date that a module containing this symbol is compiled. |
| -TIME- | value is the time that a module containing this symbol is compiled. |

eg: Test_Ansi_C.c

```
#include <stdio.h>
int main()
{ #if __STDC__ == 0
      printf("cc is not ANSI C compliant");
  #else
      printf("%s compiled at %s %s. This statement
              is at line %d", __FILE__, __DATE__, __TIME__
                              __LINE__);
  #endif
      return 0;
}
```

* Finally, ANSI C defines a set of std library function & associated headers. These headers are the subset of the C libraries available on most system that implement K&R C.

## ANSI / ISO C++ STANDARD.

These compilers support C++ classes, derived classes, virtual functions, operator overloading. Furthermore, template classes, template functions, exception handling, & the iostream library classes.

# DIFFERENCES BETWEEN ANSI C AND C++

| ANSI C | C++ |
|---|---|
| * uses K&R C default function declaration for any functions that are referred before their declaration in the program. | * Requires that all functions must be declared / defined before they can be referenced. |
| * int foo(); ANSI C treats this as old C function declaration & interprets it as declared in following manner. int foo(...); meaning that foo may called with any number of arguments | * int foo(); C++ treats this as int foo(void); meaning that foo may not accepts any arguments. |
| * Does not employ type-safe linkage technique, & does not catch user errors | * encrypts external function names for type-safe linkage thus reports any user errors. |

# THE POSIX STANDARDS

↳ (portable operating system interface)

## need for posix standard.

* Because many versions of unix exist today and each of them provides its own set of API functions, it is difficult for system developers to create applications that can be easily ported to different versions of UNIX.
* To overcome this problem, IEEE society formed POSIX in 1980s to create, a set of standards for OS interfacing.
  ↳ (or develop)
* some of the subgroups of POSIX are POSIX.1, POSIX.1b, POSIX.1c → concerned with development of set of standards for system developers.

POSIX.1

* This committee proposes a standard for a base OS API; this standard specifies APIs for the manipulating of files & processes. It is formally known as IEEE standard 1003.1-1990L and it was also adopted by ISO as international standard ISO/IEC 9945:1:1990.

POSIX.1b

* This committee proposes a set of standard APIs for a real time OS interface: these include IPC. This std is formally known as IEEE standard 1003.4-1993[7].

POSIX.1c.

* This standard specifies multithreaded programming interface this is the newest posix standard.

* these standards are proposed for a generic OS that is not necessarily be UNIX system.

ex: VMS from DEC, OS/2 from IBM, & windows-NT from microsoft corporation are posix-compliant, yet they are not UNIX systems.

* To ensure a user program conforms to posix.1 standard, the user should either define the manifested constant _POSIX_SOURCE at the beginning of each source module of the program (before inclusion of any header) as;

```
#define _POSIX_SOURCE
```

or specify the -D_POSIX_SOURCE option to a C++ compiler (cc) in a compilation;

```
% CC -D_POSIX_SOURCE *.C
```

* POSIX.1b defines different manifested constant to check conformance of user prog. to that standard. The new macro is -POSIX_C_SOURCE, & its value indicates posix version to which a user program conforms. Its value can be.

198808L ⇒ First version of posix.1 compliance
199009L ⇒ second version of posix.1 compliance
199209L ⇒ posix.1 and posix.1b compliance.

~~#pos~~

* _POSIX_C_SOURCE may be used in place of _POSIX_SOURCE. However, some systems that supports posix.1 only may not accept the _POSIX_C_SOURCE definition.
* There is also a _POSIX_VERSION constant defined in <unistd.h> header. It contains the posix version to which the system conforms.

prg to check and display _POSIX_VERSION constant of the system on which it is run.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <iostream.h>
#include <unistd.h>
int main()
{ #ifdef _POSIX_VERSION
    cout<< "system conforms to POSIX: "<<
            _POSIX_VERSION<<endL;
  #else
    cout<< "_POSIX_VERSION is undefined";
  #endif
    return 0;
}
```

## The POSIX feature Test macros

* Some unix features are optional to be implemented on a posix-conforming system. Thus POSIX.1 defines a set of feature-test macros, which, if defined on a system, means that the system has implemented the corresponding ~~function~~ features.
* These macros, if defined, can be found in the <unistd.h> header. Their names and uses are;

| Feature Test Macro | Effect if defined on a system |
|---|---|
| _POSIX_JOB_CONTROL | The system supports the BSD-style job control. |
| _POSIX_SAVED_IDS | Each process running on the system keeps the saved set-UID & set-GID, so that it can change its effective UID & GID to those values via seteuid & setegid APIs respectively. |
| _POSIX_CHOWN_RESTRICTED | If the defined value is -1, users may change ownership of files owned by them else only users with special privilege can do so. |
| _POSIX_NO_TRUNC | If -1, then any long path name is automatically truncated to NAME_MAX else an ~~error~~ error is generated. |
| _POSIX_VDISABLE | If -1, then there is no disabling character for special characters for all terminal devices. Otherwise the value is the disabling character value. |

program to print the posix-defined configuration options supported on any given system.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE   199309L
#include<iostream.h>
#include<unistd.h>
int main()
{
  #ifdef _POSIX_JOB_CONTROL
     cout<<"system supports job control";
  #else
     cout<<"system does not support job control";
  #endif

  #ifdef _POSIX_SAVED_IDS
     cout<<"system supports saved set-UID and
           saved set-GID";
  #else
     cout<<"system does not support saved set-UID
           and saved set-GID";
  #endif

  #ifdef _POSIX_CHOWN_RESTRICTED
     cout<<"chown-restricted option is:"<<
        _POSIX_CHOWN_RESTRICTED<<endl;
  #else
     cout<<"system does not support chown-restricte
        option";
  #endif

  #ifdef _POSIX_NO_TRUNC
     cout<<"pathname trunc option is:"<<_POSIX_NO
                                  TRUNC<<endl;
  #else
     cout<<"system does not support system-wide
```

```
#ifdef _POSIX_VDISABLE
    cout<< "Disable char. for terminal files is: "
        << _POSIX_VDISABLE <<endl;
#else
    cout<< "system does not support _POSIX_VDISABLE"
#endif
    return 0;
}
```

## certain constants defined in <limits.h>

- _POSIX_CHILD_MAX 6 → max no. of child that can be created
- _POSIX_OPEN_MAX 16
- _POSIX_STREAM_MAX 8 → max no of I/O streams that can be opened simultaneously by a process.
- _POSIX_ARG_MAX 4096 → size in bytes, that may be passed to exec system call.
- _POSIX_NGROUP_MAX 0
- _POSIX_PATH_MAX 255 → max no. of char. in path name
- _POSIX_NAME_MAX 14 → " " file name
- _POSIX_LINK_MAX 8
- _POSIX_PIPE_BUF 512
- _POSIX_MAX_INPUT 255 → max capacity in bytes of terminal I/P queue
- _POSIX_MAX_CANON 255 → " " Canonical I/P queue
- _POSIX_SSIZE_MAX 32767 → max value that can be stored in a ssize_t typed object
- _POSIX_TZNAME_MAX 3 → max no of char in a time zone name.

prototypes

```
long sysconf (const int limit_name);
long pathconf (const char *pathname, int flimit_name);
long fpathconf (const int fd, int flimit_name);
```

some limit_names input to sysconf.

- SC_ARG_MAX
- SC_CHILD_MAX
- SC_OPEN_MAX
- SC_CLK_TCK
- SC_JOB_CONTROL
- SC_SAVED_IDS.
- SC_VERSION
- SC_TIMERS

some flimit_names I/P to pathconf

- PC_CHOWN_RESTRICTED
- PC_NO_TRUNC
- PC_VDISABLE
- PC_PATH_MAX
- PC_LINK_MAX
- PC_NAME_MAX
- PC_PIPE_MAX.

c prg illustrating use of sysconf, pathconf, & fpathconf.

```
#define _POSIX_SOURCE
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <iostream.h>
#include <unistd.h>

int main()
{ int res;
    if ((res = sysconf (_SC_OPEN_MAX)) == -1)
        perror("sysconf");
    else
        cout << "OPEN_MAX : " << res;
```

```
if ( (res = pathconf ("/", _PC_PATH_MAX )) == -1 )
    perror ("pathconf" );
else
    cout<< " Max path name: "<< (pos + 1);

if ((res = fpathconf (0, _PC_CHOWN_RESTRICTED)) == -1 )
    perror ("fpathconf");
else
    cout << " CHOWN_RESTRICTED by stdin:" << res;

return 0;
}
```

## UNIX AND POSIX APIs

**APIs:** A set of application programming interface functions that can be called by user programs to perform system specific functions.

## API common characteristics.

* Many APIs returns an integer value which indicates the termination status of their execution.
* API return -1 to indicate the execution has failed, and the global variable errno (declared in <errno.h>) is set with an error code.
  A user process may call perror func. to print a diagnostic message of the failure to the std o/p, or it may call the strerror func. & gives it errno as the actual argument value, the strerror function returns a diagnostic message string and the user process may print that message in its preferred way (eg o/p to a error log file)
* If an API execution is successful, it returns either a zero value or a pointer to some data record where user-requested information is stored.
* possible error status codes that may be assigned to errno by any API are defined in <errno.h>
* Following is a list of commonly occur error status codes

| Error status code | Meaning |
|---|---|
| EACCESS | A process does not have access perm. to perform an operation via a API |
| EPERM | A API was aborted because calling process does not have s.u. privilege. |
| ENOENT | An invalid filename was specified to an API |
| BADF | An API was called with invalid file descriptor. |
| EINTR | API execution was aborted due to a ~~sig~~ signal interruption. |
| EAGAIN | An API was aborted because some system resource it requested was temporarily unavailable. API should be called again later. |
| ENOMEM | An API was aborted because it could not allocate dynamic memory. |
| EIO | I/O error occured in a API execution. |
| EPIPE | An API ~~was~~ attempted to write data to a pipe which has no reader. |
| EFAULT | An API was passed an invalid addr. in one of its arguments. |
| ENOEXEC | An API could not execute a program via one of the exec API. |
| ECHILD | A process does not have any child process which it can wait on. |

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

GIRISH RAO SALANKE N.S.
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore-74.

# UNIX FILES

Actually files are the building blocks of an O.S. When you execute the command in Unix, the Unix kernel fetches the corresponding executable file from the file system, loads its instruction text to memory and creates a process to execute the command.

**File types**:-

A file in a Unix or posix system may be one of the following types:

1) Regular file
2) Directory file
3) FIFO file
4) Character device file
5) Block device file

→ device file

## (1) Regular files:-

- A regular file may be either a text file or a binary file.
- These files may be read or written to by users with the appropriate access permission.
- You can create regular file by any of text editor, eg:- vi, pico, etc.

to remove or delete use 'rm'

## (2) Directory files:-

- Is nothing but a folder that contains other files, including subdirectory files i.e it provides a means for users to organize their files into some hierarchical structure based on file relationship.
- Any(process) that has read permission for a directory file can read the contents of the directory, but only the kernel can write to a directory file.
- Directory may be created by mkdir command to remove use rmdir to view the contents of directory use 'ls' command.

The following command will create a directory called vit, if it does not exist.

$ mkdir /usr/vit.

A Unix directory is considered to be empty if it contains no other files except the "." and " . . " files. The rmdir command is used to delete a directory.

$ rmdir /usr/vit

(3) **Device file**:- Block device file

Character device file

Block device file represents a physical device that transmits data as a block at a time
eg :- hard disk drivers, floppy disk drivers.

➔ Character device file represents a physical device that transmits data in a character-based manner.
Ex:- line printers, modems etc. , console(keyboard)

-> A device file is created by "mknod" command
ex:- mknod /dev/xyz c 115 5

mknod should be used with superuser privilage

character device file — major # — minor #

➔ Major # is an index to a kernel table that contains the addresses of all device driver function known to the system i.e whenever a process reads or write data to a device file, kernel use the devices major # for i/o function.

➔ Minor # tells the device driver function what actual physical device it is talking to. and the i/o buffering scheme to be used for data transfer

➔ For block device file use argument 'b' instead of c

(4) **FIFO File**:-

A type of file used for inter process communication between processes. It also called as named pipe.

➔ i.e it provides a temporary buffer for 2 or more processes to communicate by writing data of reading data from the buffer.
➔ ~~It will be having 2 file descriptors~~.
➔ A FIFO file is created by mkfifo command.

The following command creates a FIFO file called myfifo if it does not exits
$ mkfifo /usr/vit/myfifo

We can also use mknod command to create FIFO file but is should be used with option p
$ mknod /usr/vit/myfifo p

➔ It can remove by rm command similarly to that of regular files.

PIPE-BUF <limits.h>

USV -> both mkfifo/mknod.

BSD -> only mkfifo.

GIRISH RAO SALANKE N.S.
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore 74.

**(5)Symbolic link file:-**

BSD unix and SV4 defines a symbolic link file. A symbolic link file contains a pathname which references another file either the local or a remote file system.

- ➔ Posix does not support symbolic link.
- ➔ 'ln' command is used to create symbolic link file with option '-s'.

$ ln –s /usr/vit/xxx /usr/vtu/yyy

the above command creates a symbolic link for a file by name xxx present in vit directory ,by new name yyy present in vtu directory.

**The unix and posix File system:-**

- ➔ Files in unix/posix are stored in tree-like hierarchical file system. The root of a file system is the root directory, denoted by "/" character. The leaf nodes of a file system tree are either empty directory files or other types of file.
- ➔ Absolute pathname of a file consists of the names of all the directories, starting from the root
  Ex:- /usr/abc/xxx
- ➔ Relative pathname may consists of '.' and '..' characters
  . -> Current directory
  ..-> Parent directory
- ➔ In BSD (SV5) Unix file name and pathname are limited to 14 and 1024 bytes which are defined in <limits.h> as
  NAME_MAX - 14 bytes
  PATH_MAX - 1024 bytes

➔ In posix standard _POSIX_NAME_MAX 14 bytes
_POSIX_PATH_MAX 255 bytes

- ➔ Filename can be any of these from
  A to Z
  a to z
  0 to 9
  _ (underscore)
- ➔ Pathname of a file is called the "hardlink", after ln command, a single file can be referenced by 2 different links

NOTE: If option for command ln is –s then it is soft(symbolic) link.

➔ The following files are commonly defined in most Unix system

| File | Use |
|---|---|
| /etc | stores system administrative files |
| /etc/passwd | stores all user information |
| /etc/shadow | stores user passwords |
| /etc/group | stores all group information |
| /bin | stores all the system programs like cat, rm, cp, mv etc. |
| /dev | stores devices files |
| /usr/lib | stores standard lib |
| /temp | stores temporary files created by programs |

**Unix and posix file attributes:-**

The general file attributes for each file in a file system are:-

(1) File type
(2) Access permission
(3) Hard link count – no. of hard links of a file
(4) Uid - owner user identification number.
(5) Gid - file group id
(6) File size – in bytes
(7) Time stamp
  a) mtime(modification time stamp) *(mtime)
  b) atime (access time stamp)
  c) ctime (creation time stamp)
(8) Inode number – the system inode # of the file
(9) File system id – the file system id where the file is stored

➔ In addition for device file it stores major # and minor #

The kernel to a file assigns all the above attributes when it is created. Some of these remains unchanged for the entire life
They are –

File type
File inode #
File system id
Major and minor #

GIRISH RAO SALANKE N.S.
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore-74.

➔ Some of the above attributes can be changed by using some commands.

| i.e | Command | system call |
|---|---|---|
| to change permission | chmod | chmod |
| to change uid | chown | chown |
| to change gid | chgrp | chown |
| to increase hardlink count | ln | link |
| to decrease hardlink count | rm | unlink |
| to change last access time<br>modification time. | touch | utime. |

## Inodes in unix system V :

➔ In Unix system V, a file system has an inode table, which keeps tracks of all files. Each entry of the inode table is an inode record which contains all the attributes of the file, including an unique inode # & the physical disk address where the data of the file is stored.

➔ For any operation if a kernel needs to access information of a file with an inode # 15, it will scan the inode table to find an entry, which contains an inode # 15, in order to access the necessary data.

➔ An inode # is unique within a file system a file inode record is identified by a file system ID and an inode #.

➔ Generally an operating system does not keep the name of a file in its record, because the mapping of the filenames to inode # is done via directory files i.e A directory file contains a list of names of their respective inode # for all file stored in that directory.
Ex: A sample directory file content (ABC)

| Inode # | filename |
|---|---|
| 89 | . |
| 56 | .. |
| 201 | xyz |
| 346 | a.out |
| 201 | xyz_a |

Where xyz_a is the hard link of xyz

➔ To access a file, say for ex:- /usr/cse , the Unix kernel always knows the "/" directory (root directory) inode# of any process, it will scan the "/" directory file to find the inode# of the usr file. Once it gets the usr file inode # after checking access permission it then looks for the inode # of the cse file.

➔ Generally, whenever a new file is created, the Unix kernel allocates a new entry in the inode table to store the information of the new file and it will assign the inode # of the filename to the corresponding directory file.

**Application program interface to files**:-

General interface to files are:-
➔ Files are identified by pathnames

➔ Files should be created before they can be used
The various commands and system calls are:-

| | | |
|---|---|---|
| Regular file | vi, ex etc.. | open,creat |
| Directory file | mkdir | mkdir,mknod |
| Fifo file | mkfifo | mkfifo, mknod |
| Device file | mknod | mknod |
| Symbolic links | ln –s | symlink |

➔ For any application to access files, first it should be opened, generally we use "open" system call to open a file, the returned value is an integer which is generally termed as file descriptor.

➔ There are certain limits of a process to open files. A max. number of OPEN_MAX files can be opened. The value is defined in <limit.h> header file.

➔ The data transfer function on any opened file is carried out by read and write system call.

➔ File hard link can be increased by link system call, and decreased by unlink system call.

➔ File attributes can be changed by chmod, chown, link system calls.

➔ File attributes can be queried (found out or retrieved) by stat or fstat system call.

GIRISH [illegible]
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore-74.

➔ Unix and posix.1 define a structure of data type stat i.e defined in <sys/stat.h> header file. This contains the user accessable attributes of a file

```
Struct stat
{
        dev_t     st_dev;      ------>  file system id
        ino_t     st_ino;      ------>  file inode number
        mode_t    st_mode;     ------>  contains file type and access flags
        nlink_t   st_nlink;    ------>  hard link count
        uid_t     st_uid;      ------>  file user id
        gid_t     st_gid ;     ------>  file group id
        dev_t     st_rdev;     ------>  contains major and minor #
        off_t     st_size;     ------>  file size /(bytes)
        time_t    st_atime;
        time_t    st_mtime;    ------>  time stamp
        time_t    st_ctime;
};
```

**Unix kernel suppor for files:-**

➔ In Unix system v, the kernel maintains a file table that has an entry of all opened files and also there is an inode table that contains a copy of file inodes most recently accessed.

➔ The process, which will be having their own data space (data structure) wherein it will be having file descriptor table. The file descriptor table will be having an maximum of OPEN_MAX files entries. Whenever the process calls the open function to open a file to read/write, the kernel will resolve the pathname to the file inode.

➔ The steps involved are :-

(1) The kernel will search the process descriptor table and look for the first unused entry. If an entry is found, that entry will be designated to reference the file. The index of the entry will be returned to the process as the file descriptor of the opened files.

(2) The kernel will scan the file table in its kernel space to find an unused entry that can be assigned to reference the file.

If an unused entry is found the following events will occur.

➔ The process file descriptor table entry will be set to point to this file table entry.

➔ The file table entry will be set to point to the inode table entry where the i node record of the file is stored.

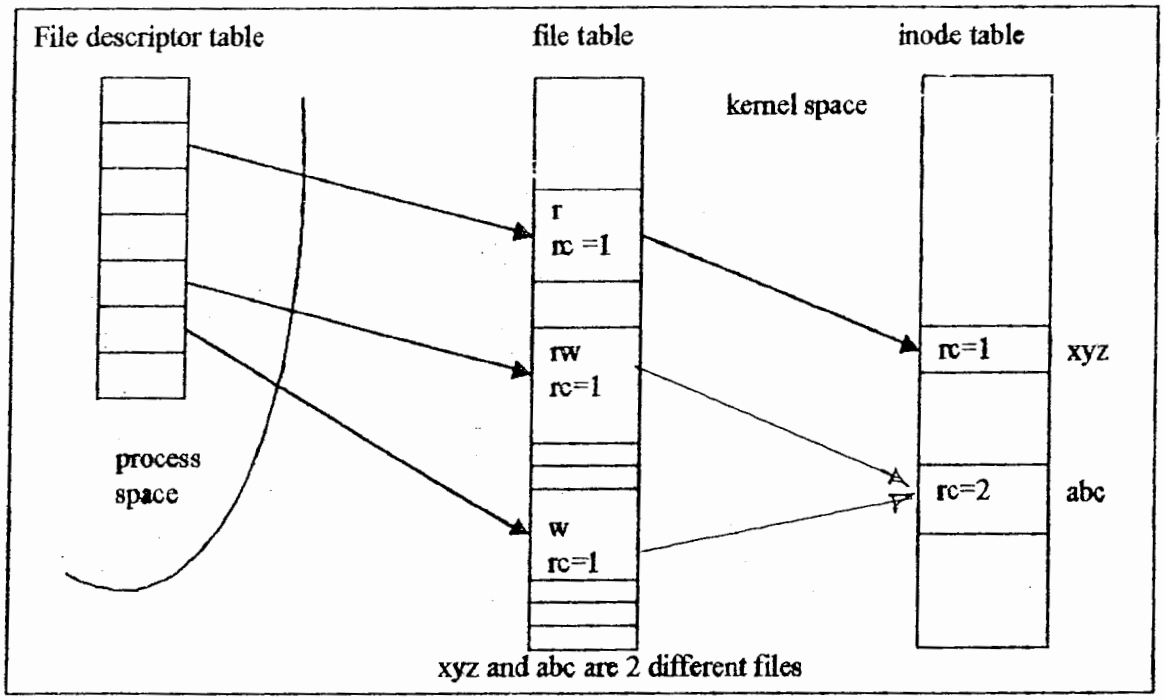
➔ The file table entry will contain the current file pointer of the open file i.e this is an offset from the pointer of the open file i.e this is an offset from the beginning of the file where the next read/write will occur.

➔ The file table entry will contain an open mode that specifies that the file opened is for read only, write only or read and write etc/.. This should be specified in open S.C

➔ The reference count (rc) in the file table entry is set to 1. rc is used to keep track of how many file descriptors from any process are referring the entry.

➔ The reference count (rc) in inode table is increased by one if 2 file table entry are referencing the same

If either (1) and (2) fails then the 'open' system call returns –1 (failure/error)




➔ Normally the reference count in the file table entry is 1 if we wish to increase the rc in file table entry this can be done using fork, dup, or dup2 system call.

GIRISH RAO SALANKE N.S.
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore-74.

➔ When a open system call is succeeded , its return value will be an integer (file descriptor). Whenever the process wants to read/write data from the file, it should use the file descriptor as one of its argument.

➔ The following events will occur whenever a process calls the 'close' function to close the files that are opened

(1) The kernel sets the corresponding file descriptor table entry to be unused.
(2) It decrements the rc in the corresponding file table entry by 1, if rc≠0 go to (6).
(3) The file table entry is marked as unused.
(4) The rc in the corresponding file inode table entry is decremented by one, if rc≠0 go to (6).
(5) If the hard-link count of the inode is not zero, it returns to the caller with a success status otherwise it marks the inode table entry as unused. & deallocates all the physical disk storage of the file.
(6) It returns to the process with a 0 (success) status.

## Hard and Symbolic link :-

➔ A hard link is a unix pathname for a file. Generally most of the unix files will be having only one hard link.

➔ In order to create a hard link we use the command 'ln'

Ex:- Consider a file /usr/cse/old, to this we can create a hard link by

ln /usr/cse/old /usr/cse/new

After this we can refer the file by either /usr/cse/old or /usr/cse/new.

->Symbolic link can be created by the same command 'ln' but with option –s

ln –s /usr/cse/old /usr/cse/new

➔ ln differs from the cp command in that cp creates a duplicated copy of a file to another file with a different path name whereas ln creates a new directory entry to reference a file.

➔ For ex:

ln /usr/cse/abc /usr/ise/xyz

Then directory files /usr/cse /usr/ise will contain

| inode # | filename |
|---|---|
| 90 | . |
| 110 | .. |
| (201) | abc |
| 150 | xxx |

| inode # | filename |
|---|---|
| 78 | . |
| 98 | .. |
| 100 | yyy |
| (201) | xyz |

both /usr/cse/abc and /usr/ise/xyz refer to the same inode# 201, thus there is no new file created.

➔ For the same operation if ln –s or cp command is used then a new inode will be created

i.e

| inode # | filename |
|---|---|
| 90 | . |
| 110 | .. |
| (201) | abc |
| 150 | xxx |

| inode # | filename |
|---|---|
| 78 | . |
| 98 | .. |
| 100 | yyy |
| (210) | xyz |

If cp is used then the data contents will be identical and the 2 files will be separate objects in the file system if we use ln –s the data will contain only the path name /usr/cse/abc.

**Limitations of hard link**:-

(1) User cannot create hard links for directories, unless he has superuser(root) privileges.

(2) User cannot create hard link on a file system that references files on a different system, because inode # is unique to a file system.

GIRISH RAO SALANKE N.S.
M.Tech
Asst. Professor
Department of CSE/ISE
Vivekananda Institute of Technology
Kumbalgodu, Bangalore-74.

The differences between hard link and symbolic link

| Hard link | Symbolic link |
|---|---|
| (1) Does not create a new inode | (1) Creates a new inode |
| (2) Cannot link directories, unless this is done by root | (2) Can link directories. |
| (3) Cannot link files across file systems | (3) Can link file across file systems. |
| (4) Increases the hardlink count of the linked inode | (4) Does not change the hardlink count of the linked inode. |

# Chapter 7: UNIX FILE API'S (Application Program Interface)

## General File API's :

### Open :-

This is used to establish a connection between a process and a file i.e it is used to open an existing file for some data transfer function OR else it may also be used to create a new file. The returned value of the open system call is a file descriptor ( row # of the file table ) which contains the inode information.

**Syntax:-**

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

int open(const char *pathname, int accessmode, mode_t permission)
```

The first argument is the name of the file to be created or opened. This may be an absolute pathname or relative pathname.

absolute -> we have to specify from the root / very long and non portable
(should start from "/")
relative -> portable and smaller ( advantages are more)
If the given pathname is symbolic reference, the function will resolve the symbolic link reference to a non symbolic link file to which the link refers

The second argument is access modes or flags which is an integer which specifies how actually the file should be accessed by the calling process.Generally the access modes are specified in <fcntl.h>

The available access modes are

O_RDONLY -> opens the files for read only
O_WRONLY -> opens the file for write only
O_RDWR -> opens the file for read and write

And there are other access modes which are termed as access modifier flags and one or more of the following can be specified by bitwise OR ing them with one of the above access mode flags to alter the access mechanism of the file

UNIT 7 :

# INTERPROCESS COMMUNICATION

Syllabus

+ Introduction
+ pipes
+ popen, pclose functions
+ Coprocesses
+ FIFOs
+ XSI IPC
+ message queues
+ Semaphores

- 6 Hours.

Ashok kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

## Introduction.

* **Definition:**

IPC is a set of techniques for processes to communicate with each other.

(or) IPC is a set of techniques for the exchange of data among multiple threads in one or more processes.
processes may be running on one or more computers connected by a network.

* Diff unix systems implement diff methods for IPC.

* IPC methods for communication b/w two processes running on one unix system are -
  - pipes (half duplex)
  - FIFOs (named pipes)
  - Message queues
  - semaphores
  - shared mem.
  - sockets

* IPC methods for communication b/w two processes running on different systems are -
  - BSD unix sockets (INET Domains)
  - AT and T Unix's TLI (Transport layer interface)
  - OS/2 named pipes
  - IBM's NETBIOS.

## PIPES.

* oldest and most common form of unix IPC.
* Basically pipe is a one way comm$^n$ channel where o/p of one is fed as i/p to other.

eg. $ who | wc -l

will display total no. of users currently working on unix.

* A pipe is created by calling the pipe function

prototype:

```
#include <unistd.h>
int pipe (int filedes[2]);
```

returns: 0 → if OK
-1 → on error.

* Two file descriptors are returned through the filedes arg.
  - filedes[0] is open for reading
  - filedes[1] is open for writing.
* The output of fieldes[1] is the input for fieldes[0].
* There are two ways to picture a pipe as shown below -

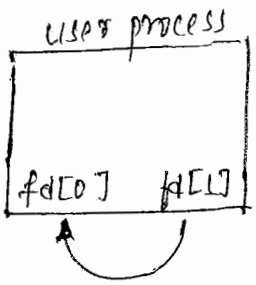



fig: Two ends of the pipe connected in a single process.

or

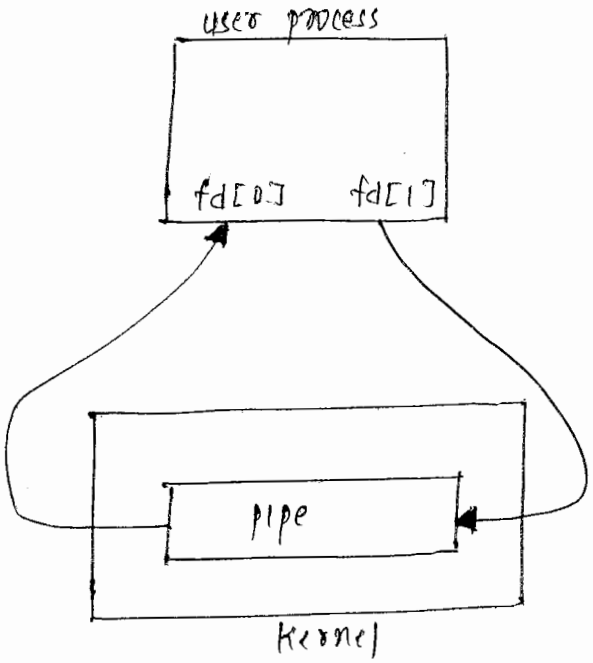



fig: Data in the pipe flows through Kernel.

* Normally the process that calls pipe then calls fork, creating an IPC channel from parent to child or vice versa. Fig below shows this scenario -

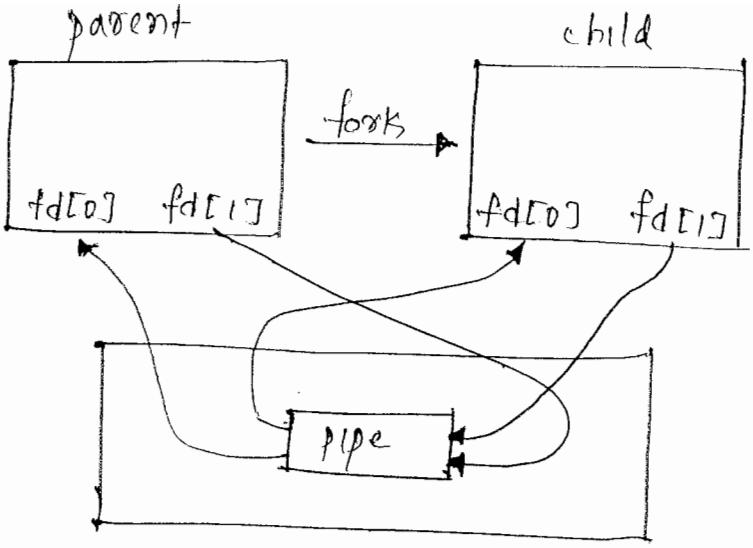



fig: Half duplex pipe after a fork.

* For a pipe from parent to child, the parent closes the read end of the pipe (fd[0]) and the child closes write end (fd[1]). Fig below shows the resulting arrangement of descriptors -

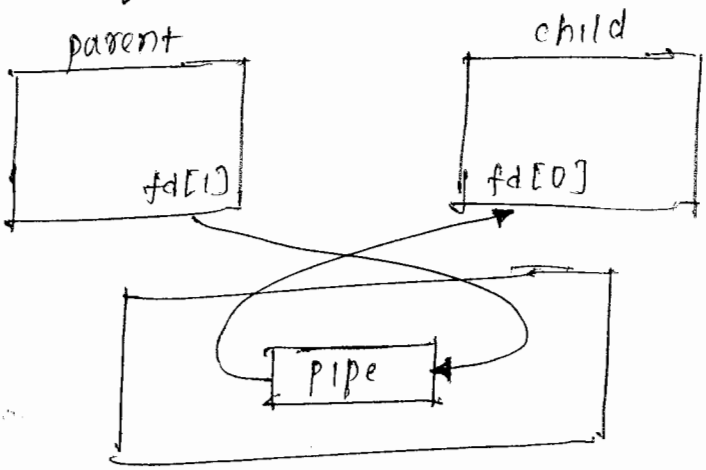



fig: pipe from parent to child.

* Limitations of pipe -

1. They are half duplex. Data flows only in one direction.
2. They can be used only b/w processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, & the pipe is used b/w parent and the child.

Example -

prog to create a pipe from parent to child, and send data down the pipe -

```
#include "ourhdr.h"
int main (void)
{
    int n, fd[2];
    pid_t pid;
    char line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys ("pipe error");

    if ((pid = fork()) < 0)
        err_sys ("fork error");
    else if (pid > 0)    // parent
    {
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    }
    else    // child
    {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

# popen and pclose FUNCTIONS

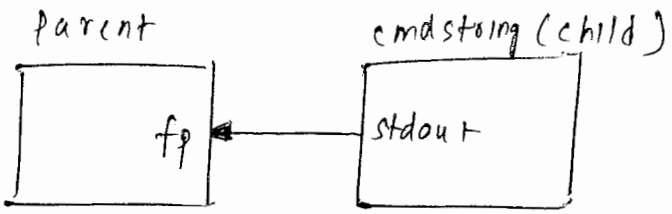
* These are functions provided by standard I/O library.
* These two functions handles -
  - creation of ~~a pipe~~ a pipe
  - fork of a child
  - closing unused ends of the pipe
  - execing a shell to execute the command, and
  - waiting for the command to terminate

prototype:

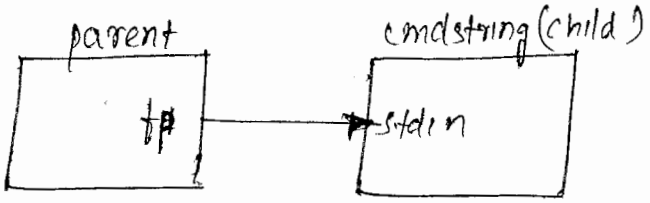
```
#include <stdio.h>
FILE *popen (const char *cmdstring, const char *type);
    returns: file pointer → if OK
             NULL → on error.
int pclose (FILE *fp);
    returns: termination status of cmdstring → if OK
             -1 → on error
```

* The func. popen does a fork and exec to execute the cmdstring, and returns a std I/O file pointer.

- If type is "r", the file pointer is connected to the std o/p of cmdstring as shown -




- If type is "w", the file pointer is connected to the std i/p of cmdstring as shown -



* The pclose func. closes the std I/O stream, waits for the command to terminate, and returns the termination status of the shell.

example - copy file to pager prog using popen.

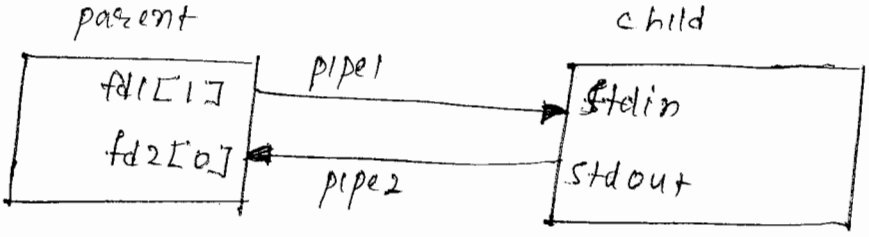
```
#include <sys/wait.h>
#include "ourhdr.h"
#define PAGER "${PAGER:-more}"
int main(int argc, char *argv[])
{
    char line[MAXLINE];
    FILE *fpin, *fpout;
    if (argc != 2)
        err_quit("usage: a.out <pathname>");
    if ((fpin = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if ((fpout = popen(PAGER, "w")) == NULL)
        err_sys("popen error");

    while (fgets(line, MAXLINE, fpin) != NULL)
    {
        if (fputs(line, fpout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (ferror(fpin))
        err_sys("fgets error");
    if (pclose(fpout) == -1)
        err_sys("pclose error");
    exit(0);
}
```

# COPROCESSES

* A unix filter is a program that reads from std input and produces o/p on std output.
Filters are normally connected linearly in shell pipelines.
A filter becomes a coprocess when the same program generates its input and reads its output.

* A coprocess normally runs in the background from a shell and its standard input and standard output are connected to another program using a pipe. In this, we have two one way pipes to the other process -

example:
The process creates two pipes : one is the std i/p of the coprocess and the other is the standard output of the coprocess. Fig below shows this arrangements -




Code below is a simple coprocess that reads two nos from std input, computes their sum & writes the sum to its std o/p -

```
#include "ourhdr.h"
int main(void)
{ int n, int1, int2;
  char line[MAXLINE];
```

```
while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0 )
{  line[n] = 0;  // null terminate
   if (sscanf (line, "%d %d", &int1, &int2) == 2 )
   {  sprintf (line, "%d", int1 + int2 );
      n = strlen (line);
      if ( write(STDOUT_FILENO, line, n) != n )
         err_sys ( "write error" );
   }
   else
   {
      if((write (STDOUT_FILENO, "invalid arg", 13)) != 13 );
         err_sys ("write_error");
   }
}
exit(0);
}
```

## FIFOs

* FIFOs are sometimes called named pipes.
(ie its like a pipe except that it has a name)
* pipes can be used only b/w related processes when a commo ancestor has created the pipe.
with FIFOs, however, unrelated processes can exchange data.
* creating a fifo is similar to creating a file .ie fifo is just a type of file.

prototype:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

returns: 0 if ok
-1 on error.

specification of mode arg is same as that for open function.

* we use reg I/O operations on FIFO files (ie open(2), read(2), write(2), unlink(2) etc).

* There are two uses of FIFOs -

1. FIFOs are used by shell commands to pass data from one shell pipeline to another, without creating intermediate temporary files.
2. FIFOs are used in a client-server application to pass data between the clients and server.

Example 1: Using FIFOs to duplicate Output Streams in a series of shell commands.

consider the procedure that needs to process a filtered i/p stream twice. fig below shows this arrangement -

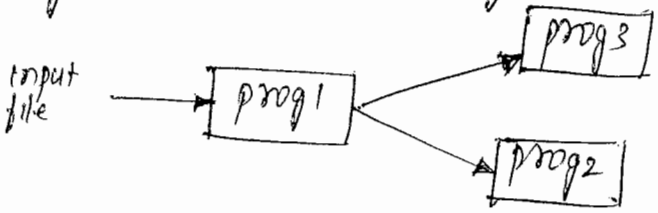



fig: procedure that processes a filtered input stream twice.

we can accomplish this procedure using a FIFO & unix prog tee(1) without using temporary file.

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2.
```

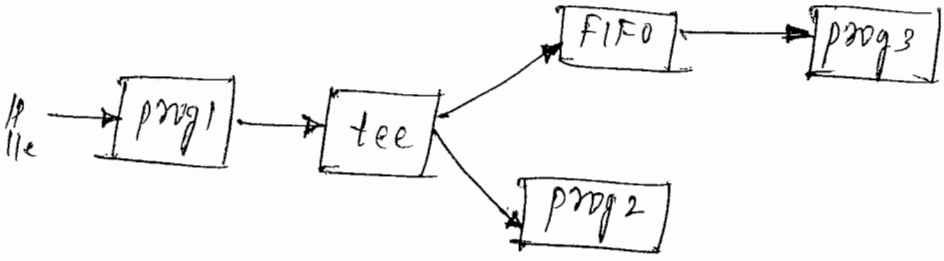
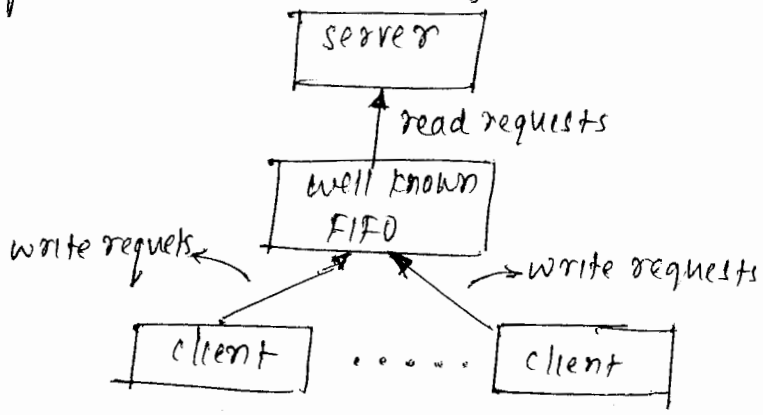


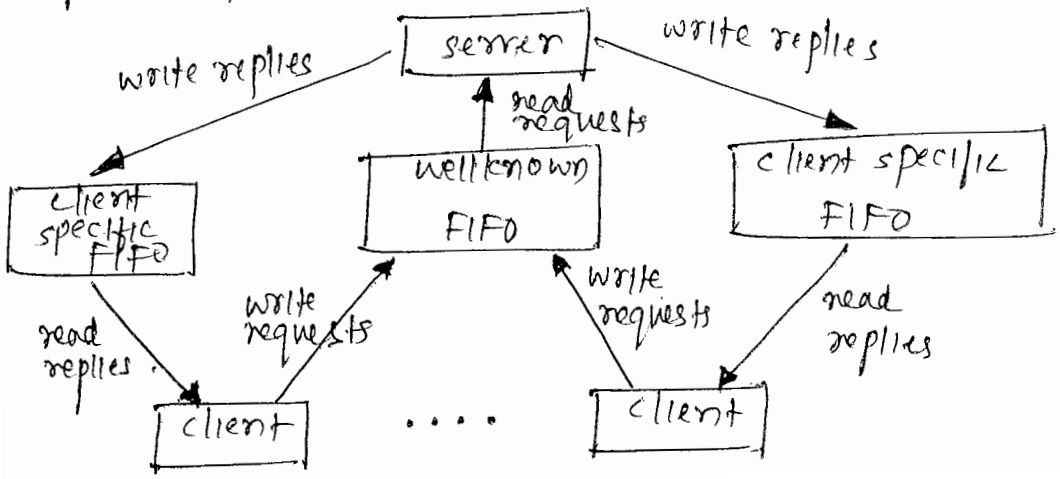

fig: using a FIFO & tee to send a stream to two diff. processes.

## Example 2: client-server communication using a FIFO.

* Each client write its request to a well-known FIFO that the server creates.
  ↳ ⇒ pathname of FIFO is known to all clients.
* Fig below shows this arrangement -




* problem in using FIFOs for this type of client server comm' is how to send replies back from the server to each client. A single FIFO can't be used as the clients would never know when to read their response,
* one soln is for each client to send its process ID with the request. the server then creates a unique FIFO for each client using a pathname based on the client's process ID.
* For ex: the server can create a FIFO with the name /tmp/serv1.XXXX where XXXX is replaced with the client's process ID.
* Fig below depicts this arrangement -



# SYSTEM V IPC

* Three more types of IPC originating from system V are -
  1. Message Queues
  2. Semaphores
  3. Shared m/m.

* There are many similarities b/w these types of IPC. We discuss these similarities here -

## identifier and keys.

* Each IPC structure (msg queue, semaphore, or shared m/m) in the kernel is refered to by a non-ve integer identifier. These are not small integers as in file descriptors.
* when a given IPC structure is created & then removed, the identifier associated with that structure continually increases until it reaches the max. +ve value for an integer & then wraps around to 0.
* whenever an IPC structure is being created (by calling msgget, semget, or shmget), a key must be specified. Its data type is key_t (long integer) defined in <sys/types.h> This key is converted into an identifier by the kernel.
* There are various way for a client and server to rendezvous (A meeting at a pre arranged time & place) at the same IPC structure.

1. Server can create a new IPC structure by specifying a key of IPC and store the returned identifier some where (such as file) for the client to obtain. The key IPC_PRIVATE guarantees that the server creates a brand new IPC structure.
2. client and server can agree on a key by defining the key in a common header. For ex, the server then creates a new IPC structure specifying the key.

3. The client & server can agree on a pathname & a project ID and call the func. to convert these two values into a key. This key is then used in step2.
   ↳ (use ftok func.)

## Permission structure

* system V IPC associates an ipc_perm structure with each IPC structure. This structure defines the permissions & owner.

```
struct ipc_perm
{
    uid_t uid;  // owner's effective uid
    gid_t gid;  //  ---))---------  gid
    uid_t cuid; // creator's effective uid
    gid_t cgid; //  ---))---------  gid
    mode_t mode; // access modes
    ulong seq;  // slot usage seq. no
    key_t key;  // key
};
```

* All ~~vatu~~ fields other than seq are initialized when IPC structure is created.

## Configuration limits

* All these forms of system V IPC have build in limits that we may encounter. Most of these can be changed by reconfiguring the kernel.

## MESSAGE QUEUES

* Message queue works like a kind of FIFO, but supports some additional functionality.
Generally messages are taken off the queue in the order they are put on. However, there are ways to pull certain messages out of the queue before they reach the front.
Its like cutting in line.

* A process can create a new message queue or it can connect to an existing one. In this way, two processes can exchange information through the same message queue.

* To connect to a queue or to create it if it doesn't exists, use msgget() system call.

```
int msgget (key_t key, int msgflag);

returns message queue ID on success
        -1 on failure (& sets errno)

key → system wide unique identifier describing the queue
      you want to connect to (or create)

msgflag → tells what to do with the queue.
          To create a queue, this field must be set to IPC_CREAT
          bitwise ORed with the permissions for this queue.
```

*

* Each queue has the following msqid_ds structure associated with it.
This structure defines the current status of the queue.

```
struct msqid_ds
{
  struct ipc_perm msg_perm;
  struct msg *msg_first; // first msg on queue
  struct msg *msg_last; // last  -"- ———
  ulong msg_cbytes; // current no. of bytes on queue
  ulong msg_qnum; // no. of msges on que.
  ulong msg_qbytes; // max no. of bytes on queue.
  pid_t msg_lspid; // pid of last msgsnd()
  pid_t msg_lrpid; // pid of last msgrcv()
  time_t msg_stime; // last msgsnd() time
  time_t msg_rtime; // last msgrcv() time
  time_t msg_ctime; // last-change time
};
```

notes:

1. message queues are the linked list of messages stored within the kernel and identified by a message queue identifier.
2. The message queue is called "queue"
3. Its identifier is called "queue-ID"
4. New messages are added to the end of a queue by msgsnd()

* First function usually called is msgget.

```
#include<sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
int msgget(key_t key, int flag);
```

<u>when a queue is created</u>,

→ ipc_perm structure is initialized.

→ msg_qnum, msg_lspid, msg_lrpid, msg_stime, & msg_rtime are set to 0.

→ msg_ctime is set to current time

→ msg_qbytes is set to system limit.

* Various other operations performed are -

1.
```
#include <sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds *buf);
    └▷ performs various operations on queue.

0 → OK
-1 → error.
```

'cmd' arg specifies the command to be performed on the queue specified by msqid.

IPC_STAT → fetch the msqid_ds structure for this queue, storing it in structure pointed by buf.

IPC_SET → set msg_perm.uid, msg_perm.gid, msg_perm.mode msg_perm.qbytes from structure pointed to by buf in the structure associated with the queue.

IPC_RMID → remove the message queue from the system & any data still on the queue.

2. Data is placed on the queue by calling msgsnd().

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

0 → ok
-1 → error.

nbytes → +ve long integer type field

ptr → points to a long integer that contains the +ve integer message type, & it is immediately followed by msg data.

3. msgrcv() - through this we retrieve msg from queue.

```
int msgrcv(int msqid, void *ptr, size_t nbytes,
           long type, int flag);
returns size of data portion of msg → ok
                                  -1 → error.
```

eg:

```
#include<sys/types.h>
        <sys/ipc.h>
        <sys/msg.h>
#define PERMS 0666
main()
{ key_t i;
  int msqid;
  for(i=0; i<50; i++)
    { msqid = msgget(i, PERMS | IPC_CREAT);
      if(msqid<0)
        { perror("msgget failed");
          exit(1);
        }
      printf("\n msqid=%d", msqid);
    }
}
```

o/p

So message queues will be created.

## SEMAPHORES

* A semaphore isn't really a form of IPC.
* A semaphore is a counter used to provide access to shared data object for multiple processes.
* To obtain a shared resource, a process needs to do the following -
  1. Test the semaphore that controls the resource.
  2. If the value of the semaphore is +ve, the process can use the resource. The process decrements the semaphore value by 1, indicating that it was used one unit of the resource.
  3. If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.
* The kernel maintains a semid_ds structure for each semaphore -

```
struct semid_ds
{
  struct ipc_perm sem_perm;
  struct sem *sem_base; // ptr to first semphore in set
  ushort sem_nsems; // no. of semaphores in set
  time_t sem_otime; // last_semop() time
  time_t sem_ctime; // last-change time.
};
```

* sem_base points to an array of "sem" structures containing sem_nsems elements, one element in the array for each semaphore value in the set.

```
struct sem
{
    ushort semval; // semaphore value, always >= 0
    pid_t sempid; // pid for last operation.
    ushort semncnt; // no. of processes awaiting
                    semval > curval
    ushort semzcnt; // no. of processes awaiting
                    semval = 0.
};
```

* First func^n to call is semget to obtain a semaphore ID.

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

int semget (key_t key, int nsems, int flag);

returns semaphore ID → O/S
        -1 → error.
```

+ when a new set is created -

1. ipc_perm structure is initialized.
2. sem_otime is set to 0
3. sem_ctime is set to current time
4. sem_nsems is set to nsems.

# the semctl func. is the catchall for various semaphore operations

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

int semctl (int semid, int semnum, int cmd,
                union semun arg);
```

```
union semun
{
    int  val; // for SETVAL
    struct semid_ds *buf; // for IPC_STAT and IPC_SET
    ushort *array;
};
```

* cmd arg specifies one of the following 10 commands to be performed on the set specified by semid.

IPC_STAT - fetch the semid_ds structure for this set, storing in the structure pointed to by arg.buf.

IPC_SET - set sem_perm.uid, sem_perm.gid & set_perm.mode from the structure pointed by arg.buf in the structure associated with the set.

IPC_RMID - remove the semaphore set from the system.

GETVAL - return the value of semval for the member semnum.

SETVAL - set the value semval for the member semnum.

GETPID - return the value sempid for the member semnum.

GETNCNT - return the value of semncnt for the member semnum

GETZCNT - return the value of semzcnt for the member semnum.

GETALL - fetch all the semaphore values in the set & store in array pointed to by arg.array.

SETALL - set all the semaphore values in the set to the values pointed to by arg.array.

for all GET commands other than GETALL, the func. returns the coorresponding value.

for remaining commands, the return value is 0.

the funcn semop automatically performs an array of operations on a semaphore set.

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

int semop (int semid, struct sembuf semopparray[],
                size_t nops);
```

0 → ok
-1 → error.

```
struct sembuf
{ ushort sem_num; // member no. in set (0,1... nsems-1)
  short sem_op; // operation (-ve, 0, or +ve)
  short sem_flg; // IPC_NOWAIT, SEM_UNDO
};
```

nops specifies no. of operations (elements) in the array.

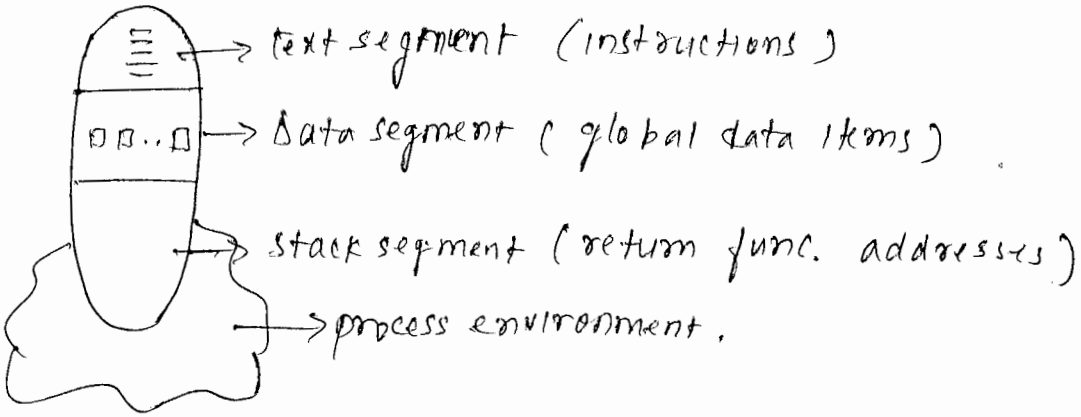
Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

# UNIT 4 : UNIX PROCESSES.

# ENVIRONMENT OF UNIX PROCESS.

* Defn: A process is a program in execution. It may also be defined as strong contender to the system resources.

* There are 4 important parts for a process -




1.) Text segment - m/c executable instructions.
- user's text.
- user's libraries
- shared libraries

2.) Data segment - Global and static data items.
- initialized data - they have stored values, initialized at compilation time
- Block storage - values are not set at compilation time.

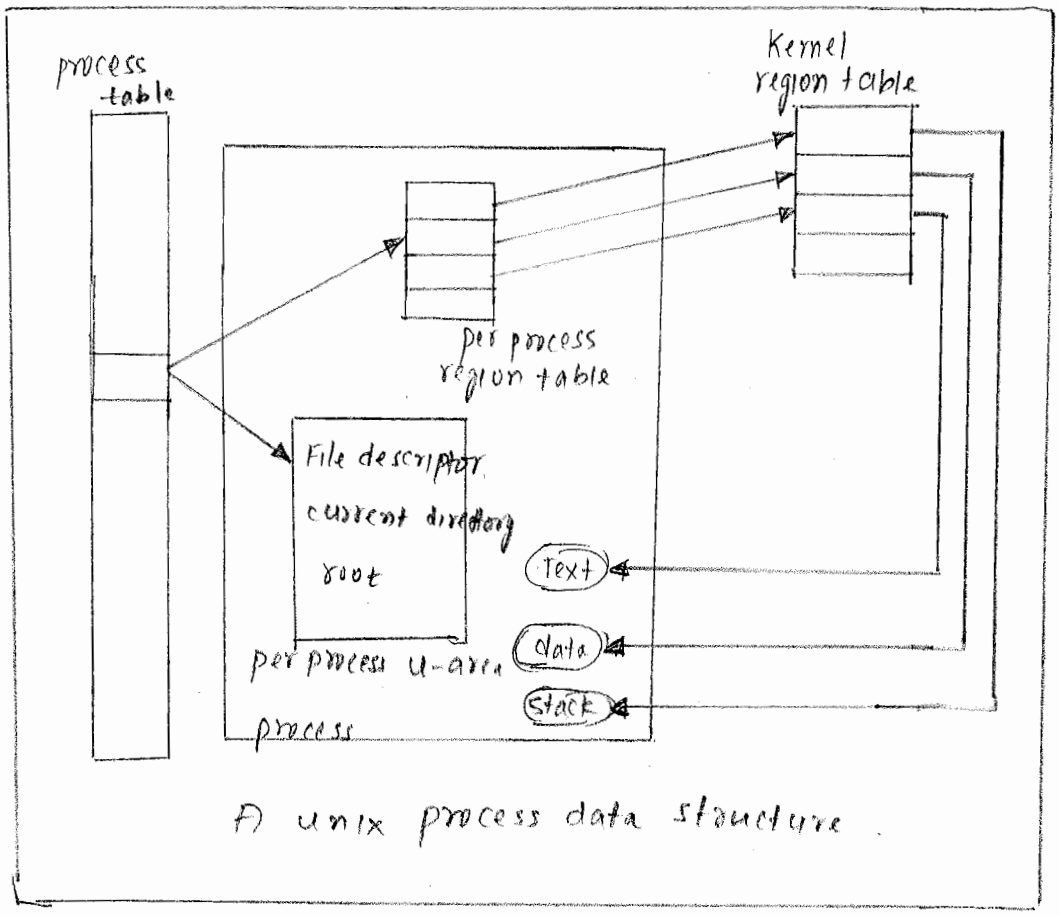
3.) Stack segment - does not exist at the compilation time, normally will be created during execution time
- local variables -> created and destroyed
- parameters -> valid for proceduse definition execution.
- return address -> called instruction.

4.) Environment → (mostly inherited by all the process)
→ argv → command line arg (process parameters)
→ argc → redundant (some process is capable of doing
→ argv → env. variables different things at diff. time)

note:
Unix shell is a process that is created when a user logs on to a system.
When a user enters a command like "vi temp" to a shell prompt, the shell creates a new process (ie its child process)
The child process inherits many attributes from its parent process, and is scheduled by the kernel to run independently from its parent.

By creating a child process, we have more adv -
- An user can perform multitasking operations.
- As child process is having its own virtual addr. space its success or failure in execution will not affect its parent. But a parent process can query the exit status and run-time statistics of its child process after it has terminated.

# UNIX KERNEL SUPPORT FOR PROCESSES
(As per unix system V)

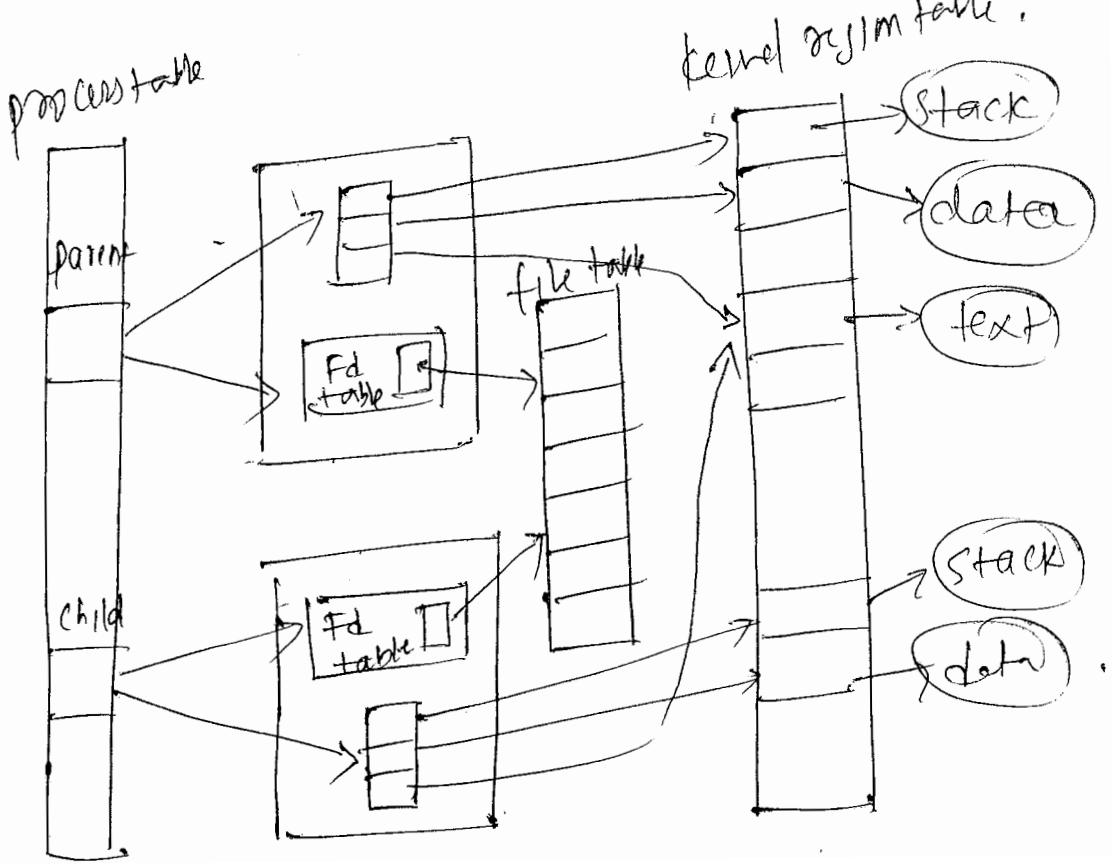



A unix process data structure

* The data structure and execution of processes are dependent on O.S ~~sy~~ implementation.
As shown in the above fig, a unix process minimally ~~maintains~~ consists of a text segment, data, & stack segment.

* A segment is an area of memory, which is managed by the system as a unit.
A text segment consists of program text in m/c executable instruction code format.
A data segment contains static and global variables and their corresponding data.
A stack segment contains a runtime stack. A stack provides storage for func. arguments, automatic variables, and return addresses of all active functions for a process.

* Unix kernel has a process table that keeps track of all active processes. Some of the processes are belonging to the kernel and are called "system processes".
Every entry in the process table contains pointers to the text, data, and stack segments and also to U-Area of a process.
The U-Area is an extension of process table entry and contains other process specific data such as the file descriptor table, current root and working directory, inode numbers, and a set of system imposed process units.



# THE main FUNCTION.

* Every C prog starts its execution from a function called main.

prototype :

```
int main (int argc, char *argv[] );
```

argc - number of command line arguments.
argv - array of pointers to the arguments.

* A C program is started by kernel.
* A special startup routine is called ~~before~~ by the kernel before main function is called.
* The executable prog file specifies this start-up routine as the starting address for the program. This is set up by the link editor when it is invoked by the C compiler.
* This start up routine takes values from the kernel, and handles any initialization that is required and then calls the main function.

PROCESS TERMINATION

* Normal termination
  - return from main
  - calling exit function.
  - Calling _exit function.

* Abnormal Termination
  - Calling abort
  - Terminated by a signal.

note: A process, in addition may also terminate in following ways

- return of last thread from its start routine
- Calling pthread_exit from last thread.

} Normal Termination

- response of the last thread to a cancellation request

} Abnormal Termination.

exit and _exit functions.

* Both are used to terminate a process.
* _exit returns to kernel immediately.
* exit performs certain clean up processing and then returns to the kernel.

prototype

```
#include <stdlib.h>
    void exit ( int status );

#include <unistd.h>
    void _exit ( int status );
```

Both accepts single integer arg - exit status.

* exit status of a process is undefined in following situations -
- either exit or _exit are called without any arguments.
- Main does a return without a return value.
- main "falls off the end". ie. if the exit status of the process is undefined.

atexit function.

* In ANSI C, a process can register upto 32 functions that are automatically called by exit functions. these are called exit handlers and are registered by calling the atexit function.
* whenever a process executes exit function, then the functions (which are registered through atexit) are called in the reverse order of their registration.
* Each function is called as many times as it was registered.

prototype

```
#include <stdlib.h>
int atexit ( void (*func) (void) );
```

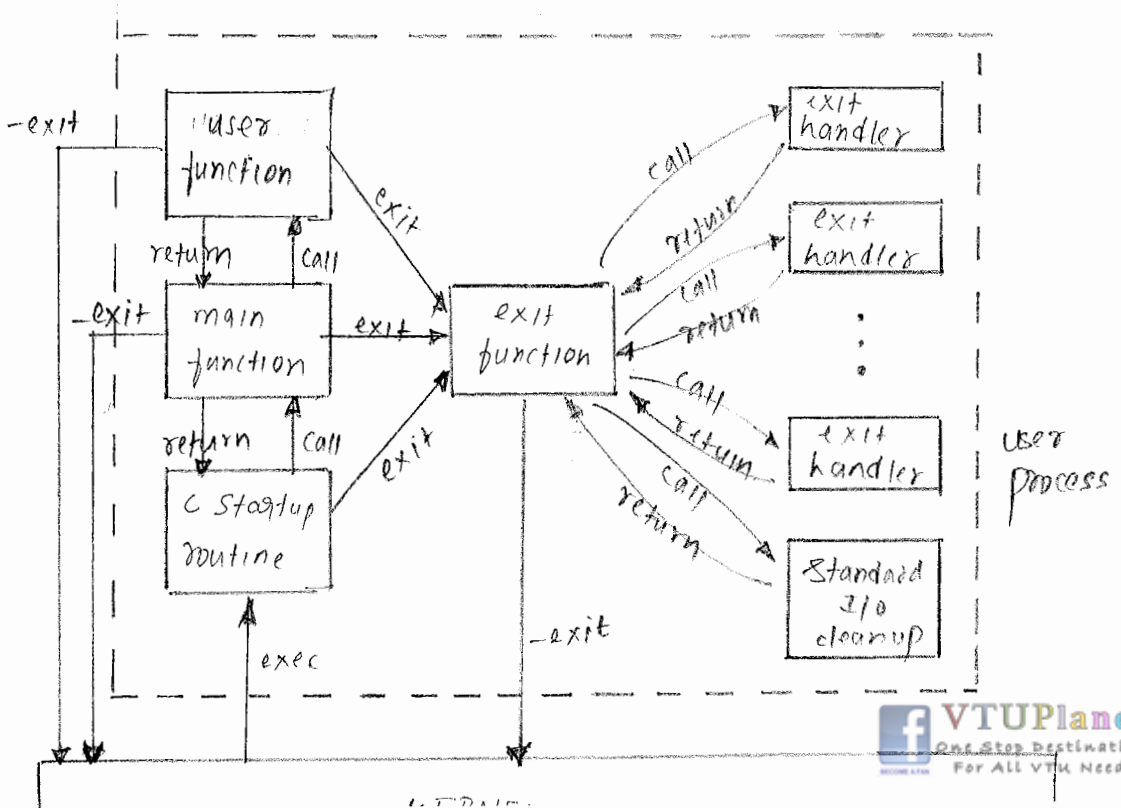
* useful for cleaning up open files, freeing certain resources etc

ex:

```
#include<stdio.h>
#include<stdlib.h>
void exit_fn1 (void)
{ printf ("exit func. no 1 called");
}
void exit_fn2 (void)
{ printf ("exit func no 2 called");
}
int main (void)
{ atexit (exit_fn1);
  atexit (exit_fn2);
  return 0;
}
```

output
exit func. no 2 called.
exit func. no 1 called.

* Below fig shows how actually a C prg is started & terminated.



command line arguments

program to echo command line arguments.

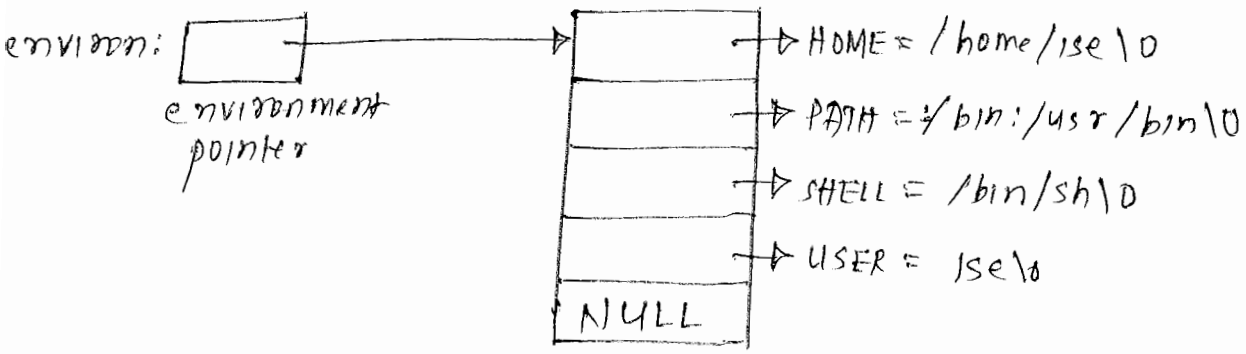
```
int main (int argc, char *argv[])
{
    for (int i=0; i<argc; i++)
        printf ("argv[%d]: %s", i, argv[i]);
}
```

ENVIRONMENT LIST

* Basically we use env. variables to customize the environment of a user.
The env. list is an array of character pointers (strings) with each each pointer containing the address of a null terminated string.
* The address of the array of pointers is contained in the global variable environ.
The declaration of this variable is -

```
extern char **environ;
```

* for eg: if the env. consists of 4 strings, it could look like -




* Generally any env. var. is of the form

| name = value |
|---|

most predefined names are in uppercase.

```
int main(int argc, char *argv[], char *envp[])
{ int i;                                  ↳ addr of the
  for(i=0; envp[i]; i++)                    env. list
      printf("%s\n", envp[i]);
}
```

## MEMORY LAYOUT OF A C PROGRAM

* We can imagine a C program as a composition of following segments -

1. Text segment

* consists of set of m/c instructions that are executed by CPU.
* Usually text seg. is sharable so that only a single copy of the text seg resides in m/m for frequently executed programs like text editors, C compilers, shells etc.
* Text seg is read only in order to prevent any modifications.

2. Initialized data segment

* contains variable that are specifically initialized in a program
  eg: int sum=100;

3. Uninitialized data segment

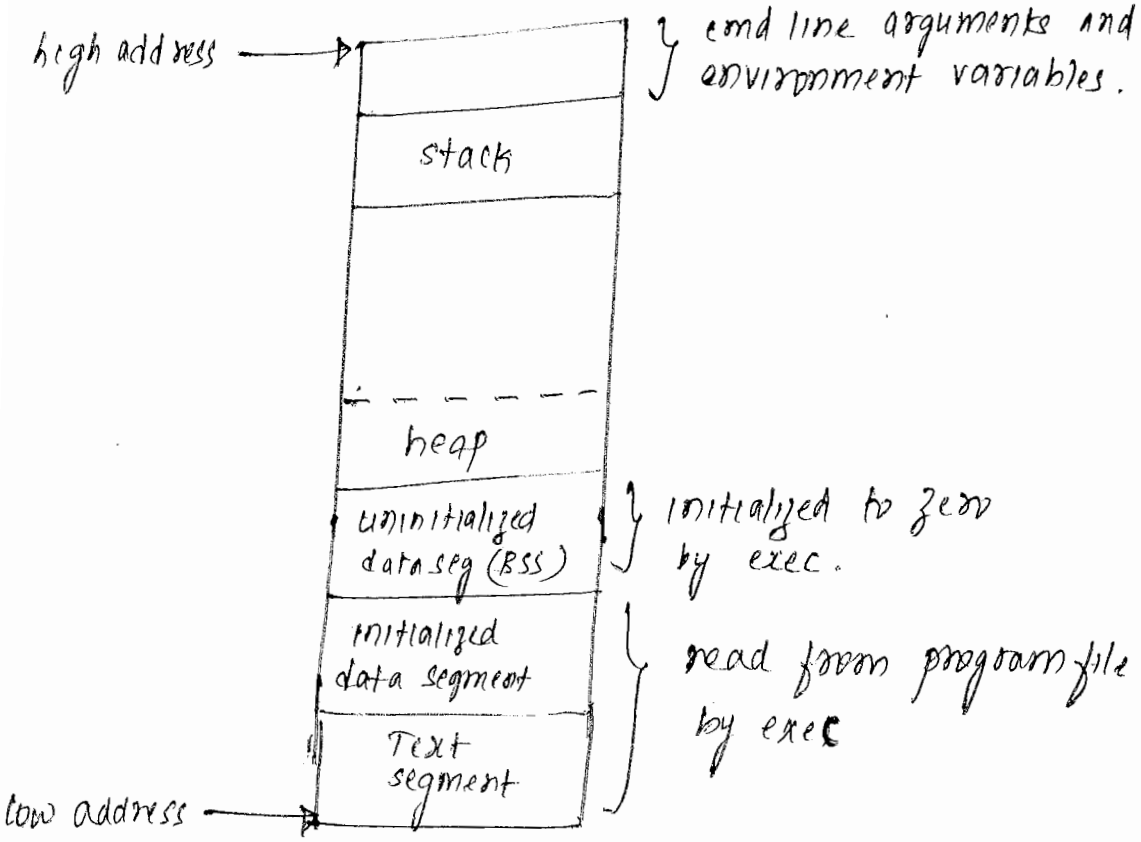
* often termed as BSS (block storage segment)
* used to store variables which are not initialized to any value.
* Actually the kernel initialized this data to arithmetic 0 or NULL before program starts executing.
* eg: long sum[1000];

4. Stack segment

* Used to store the return addresses of functions, values of local variables (automatic variables) etc
* Usually it is used to implement recursive function.

5. Heap Segment

* Dynamic memory allocation usually takes place on heap seg.
* Normally this seg is allocated b/w uninitialized data seg and the bottom of stack segment.



# SHARED LIBRARIES

# ENVIRONMENT VARIABLES

* used to customize the environment of a user (process)
* env. strings are usually of the form

  name = value.

* unix kernel never looks at these strings - there interpretation is up to the various applications.
* the functions that are used to set and fetch the values for these variables are
  - setenv()
  - putenv()
  - getenv()

1. getenv()

   prototype `#include <stdlib.h>`

   `char *getenv(const char *name);`

   returns a pointer ~~the~~ to the value of a name=value string and returns NULL if not found.

   we should always us getenv() to fetch a specific value from the environment, instead of using environ directly.

2. putenv()

   ```
   #include <stdlib.h>
   int putenv (const char *str);
   ```

   returns 0: ok
   nonzero: error.

   str will be of the form name=value & putenv() places it in the env. list. if the name already exists, its old definition is first removed.

3. setenv()

```
#include<stdlib.h>
int setenv (const char *name, const char *value,
                                    int rewrite);
```

returns 0 : ok
nonzero : error.

setenv() sets name to value.
If name already exists in the env. then

→ If rewrite is nonzero, the existing definition for name is first removed

→ If rewrite is 0, then existing definition of the name is returned and no error is returned.

note: Changing the value of the env. var. only affects the env. of a child process, the env of parent process remains unchanged.

4. unsetenv()

```
void unsetenv (const char *name);
```

It removes any definition of name.

MEMORY ALLOCATION

NOTE:- All the above allocation functions are implemented by the sbrk or the brk system call. Normally, this system call expands or contracts the heap region of a process.

- Although we can expand or contract by using sbrk(2) system call, most version of malloc( ) and free( ) never decrease their memory size.The space that we free is available for later allocation i.e., free space is not returned to the kernel, it is kept in the malloc pool or process address space.

- Normally what happens is that most implementations allocate a little more space than is requested and uses the additional space for record keeping such as size of the allocated block, a pointer to the next block etc ,

- Other possible errors may be trying to free a block, which is already fixed, trying to call free with a pointer that was not obtained from one of the three alloc functions. If a process calls continuously malloc( ) function without calling free function, then there is ' Memory Leakage'. Some versions of these routines provide additional error checking facility since it is very difficult to detect errors.

- The operation of the memory allocation is often crucial to the run-time performance of certain applications. Some systems provide additional capabilities. The mallopt function in SVR4 allows a process to set certain variables that control the operation of the storage allocator. The function malloc info is used to provide statistics on the memory allocator.

**Alloca function :**

- The alloca function has the same calling sequence as malloc, but instead of allocation on heap as malloc does, alloc function allocates the memory from the stack frame of the current function.

- The advantage of allocating memory on stack is that we do not have to free the space. It is automatically taken off when the function returns. Alloca function increases the size of stack frame.

- The disadvantage is some systems do not support alloca function, if it is impossible to increase the size of the stack frame after the function has

## Memory Allocation :

The main three functions that are used for memory allocation are

1. malloc( ) – allocates a specified number of bytes. The initial value of the memory is indeterminate.
2. calloc( ) – allocates space for a specified number of objects of a specific size. The space is initialized to all zero bits.
3. realloc( ) – changes the size of previously allocated area ( increases or decreases) when the size increases, it may involve moving previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indetermined.

The prototype of these functions are –

```
#include<stdlib.h>
void *malloc ( size_t size);
void *calloc( size_t nobj, size_t size);
void *realloc( void *ptr, size_t newsize);
```

→ All these 3 functions return non-null pointer if ok, Null on error.

- The pointer returned by the three allocation functions is guaranteed to be suitably aligned so that it can be used for any objects. Since all three return generic void pointer, they remove the overhead of typecasting. Say for eg.,

```
int *ptr;
ptr = malloc(100 *sizeof(int))
```

↑ no need of typecasting

- The function free( ) causes the space pointed by the ptr to be de-allocated. This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.

```
void free(void *ptr);
```

- In realloc function, we should note that always the return address of the function may not be the same. It the first argument of the realloc( ) function i.e., ptr is a NULL pointer. It behaves just similar to malloc() and allocates the space of the specified new size.

➔ The functions tcgetpgrp( ) returns the process group id of the foreground process group associated with the terminal open on filedes.
If the process has a controlling terminal, the process can call tcsetpgrp( ) to set the foreground process group id to pgrpid. The value of pgrpid must be the process group id of a process group in the same session, filedes must refer to the controlling terminal of the session.

UNIT 5 :

# PROCESS CONTROL

Syllabus

Chapter 5a : process control.

* introduction.
* process identifiers.
* fork
* vfork
* exit
* wait and waitpid
* wait3 and wait4
* Race conditions.
* exec
* changing user IDs and Group IDs
* Interpreter files
* System function.
* process Accounting
* User identification
* process times

Chapter 5b : process relationship.

* Introduction.
* Terminal logins
* Network logins.
* process groups.
* sessions.
* Controlling terminal.
* tcgetpgrp and tcsetpgrp func.
* Job control.
* Shell execution of programs.
* orphaned process groups

- 7 hours.

chapter 5A:

# PROCESS CONTROL

process control is concerned with the creation of new process, executing programs, and termination of process.

## PROCESS IDSNTIFIERS

* Every process has a unique process ID, a non-ve integer.
* special processes -

processID = 0 → schedular process, also known as swapper.

processID = 1 → init process
it never dies. It is a normal user process run with super user privilege.

processID = 2 → pagedaemon. It is responsible for supporting the paging of virtual m/m system.

* In addition to process ID, there are other identifiers for every process. The following functions returns these identifiers.

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

Returns

pid_t getpid(void); → pid of calling process
pid_t getppid(void); → parent pid of calling process.
uid_t getuid(void); → real user ID of calling process.
uid_t geteuid(void); → effective ——"——
gid_t getgid(void); → real groupID of calling process
gid_t getegid(void); → effective ——"——

# fork FUNCTION

* The only way the unix process is created by UNIX kernel is when an existing process calls the fork function.

prototype

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

newly created process is termed as "child process".

Return value

- on success: a child process is created and fork returns the child pid to the parent. The child process returns a zero value from the fork.
- on failure: fork does not creates a child process and it will return a value -1 and sets the corresponding error code in errno variable.

ie fork is called once, and returns twice.

* The reason for why the child process ~~get~~ id is returned to parent is — A process can have more than one child and there is no function that allows a process to obtain a pid of its children.

* The reason for why fork returns 0 to the child is because - A process can have only one single parent, so that the child always call getppid() to obtain ppid of its parent.

* A child is a copy of parent. Child gets a copy of parents text, data, heap, and stack.

* Instead of completely copying, we can use copy on write (COW) technique.

# After the creation of child process, both parent and child continues their execution with the instructions followed by fork();

for eg:

```
#include <sys/types.h>
#include "ourhdr.h"

int glob = 6;
char buf[] = "a write to stdout\n";
int main(void)
{
    int var;
    pid_t pid;
    var = 88;
    if(write(STDOUT_FILENO, buf, sizeof(buf)-1) !=
            sizeof(buf)-1)
        err_sys("write error");
    printf("Before fork");
    if((pid = fork()) < 0)
        error_sys("fork error");
    else if(pid == 0)    // child.
    {
        glob++;
        var++;
    }
    else                 // parent
        sleep(2);

    printf("pid = %d glob = %d var = %d", getpid(), glob, var)
    exit(0);
}
```

$ a.out
a write to stdout
before fork
pid = 430 glob = 7 var = 89 → childs var. were changed
pid = 429 glob = 6 var = 88 → parents copy were not changed.

eg2:

```
int main()
{   printf("usp");
    fork();
    printf("Richard stevens");
}
```

o/p

usp
Richard stevens
Richard stevens.

File sharing

* After successful creation of child process by fork(), what are all the file descriptors that are being opened by parent process are also been shared by child process.
* There are 2 ways of handling descriptors after fork -
  1. parent waits for the child to complete.
  2. After fork, the parent closes all the descriptors that it doesn't need and the child does the same thing.

* Besides open files, the other properties inherited by child are
  - real uid, real gid, effective uid, effective gid
  - supplementary group ids.
  - process gid
  - session ID
  - controlling terminal
  - set-uID and ~~get~~ set-gid flag.
  - current working dir.
  - root dir.
  - env. variables.
  - attached shared m/m segments.
  - resource limits etc

* Differences b/w parent and child are -
- return value from fork.
- process IDs are different.
- parent ppids are different.
- values of tms_utime, tms_stime, tms_cutime, tms_ustime is 0 for child.
- File lock set by parent are not inherited by child.
- pending alarms are cleared for child.
- set of pending signals for the child is set to empty set.

* there are two uses for fork -
1. A process can duplicate itself so that parent & child can each execute diff. sections of code at the same time.
2. A process can execute a different program. (child does exec right after calling fork)

vfork FUNCTION.

* same as fork.
* It is intended to create a new process when the purpose of new process is to exec a new program.
* It doesn't fully copy the addr. space of parent into child, since child won't reference that address space, the child just calls exec (or exit) right after the vfork.
* child runs in the same addr space as parent until it calls either exec or exit.
* vfork guarantees that child runs first, until the child calls exec or exit. When child calls either of these func, parent resumes -

```
#include<sys/types.h>
#include"unistd.h>

pid_t vfork (void);
```

Note:
Vfork is unsafe to use. If the child modifies any data of the parent before it calls exec or exit, those changes will remain when parent resumes execution.
Soln: use COW technique.

wait and waitpid functions.

* whenever a process is terminated, its parent is notified by the kernel by sending a ~~SIGCHLD~~ SIGCHLD signal
Termination of an child is an asynchronous event.

* wait and waitpid system calls are used by parent process to wait for its child process to terminate and to retrieve the child exit status.

* The process that calls wait or waitpid can
- Block (if all of its children are still running)
- return immediately with the termination status of a child (if a child is terminated and is waiting for its termination status to be fetched)
- return immediately with an error (if it doesn't have any child process).

prototype -

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

pid_t wait (int *statloc);
pid_t waitpid (pid_t pid, int *statloc, int options);
```

returns : pid of child on success
-1 on error.

The difference b/w these two funcs are -

- wait can block the caller until a ~~Calle~~ child process terminates, while waitpid has an option that prevents it from blocking.
- waitpid doesn't wait for the first child to terminate - it has a no. of options that control which process it waits for

* Diff. values that can be supplied for pid_t arg are -

1. a child pid - waits for the child with pid.
2. -1 - waits for any child.
3. 0 - Waits for any child in same process group as the parent.
4. -ve value but not -1 - waits for any child whose process gid is same as absolute value of child pid.

* statloc is a pointer to integer.
If statloc is not a null pointer, termination status of the terminated process is stored in the loc$^n$ pointed to by this argument.

* macros which provide info. about how a process terminated.

WIFEXITED (status) : True, if child terminated normally.
WEXITSTATUS (status) → fetches lower order 8 bits of arg that the child passed to exit or _exit.

WIFSIGNALED (status) : True, if child terminated abnormally
WTERMSIG (status) - fetch the signo that caused termination
WCOREDUMP (status) - True if core file was generated.

WIFSTOPPED (status) : True, for a child that is currently stopped.
WSTOPSIG (status) - fetch the sig no that caused child to stop.

* The caller can direct the waitpid to be either blocking / non blocking and to wait for any child that is / not stopped due to job control. These are specified via options arg -
  - If WNOHANG flag is set in the option field, the caller will be nonblocking
  - If WNOTRACED flag is set, the func. will also wait for a child ~~for~~ that is stopped due to job control.

wait3 and wait4

* 4.3 + BSD provides 2 additional func. wait3 and wait4
* These func. ~~are~~ are same as waitpid but provide additional information about the resources used by the terminated process.

returns : pid → ok
-1 → error.

prototype

```
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/times.h>
#include <sys/resources.h>

pid_t wait3 (int *statloc, int options, struct
                    rusage *usage);

pid_t wait4 (pid_t pid, int *statloc, int options,
                    struct rusage *usage);
```

note: Arg supported by various wait func.

| func. | pid | option | rusage |
|---|---|---|---|
| wait | X | X | X |
| waitpid | ✓ | ✓ | X |
| wait3 | X | ✓ | ✓ |
| wait4 | ✓ | ✓ | ✓ |

RACE CONDITION:

* Race condition occurs when multiple processes are trying to do something with the shared data and final outcome depends on the order in which the processes run.

* the fork func. is a lively breeding ground for race condn. - If any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork.

* consider the ex: when a second child prints its ppid. If the second child runs before the first child, then its parent process will be 1st child But if the first child runs first and exits, the parent process of second child is init. problems of this kind are hard to debug & are referred as "Race condition".

* Ex: Below prog contains a race condition because the o/p depends on the order in which the processes are run by kernel & how long a process runs.

```
#include<stdlib.h>
#include<sys/types.h>
static void charatatime (char *);
int main(void)
  { pid_t pid;
    if((pid=fork()) < 0 )
        printf ("fork error");
    else if(pid==0)
    { charatatime ("output from child");
    }
    else
    { charatatime ("output from parent");
    exit(0);
```

```
static void charatatime (char *str)
{   char *ptr;
    int c;
    setbuf (stdout, NULL);
    for (ptr = str; c = *ptr++; )
        putc (c, stdout);
}
```

* setbuf sets the std o/p unbufferred so that every char. generates a write.
* o/p of this prog may not be same all the time

```
$ a.out
output from child
output from parent.
$ a.out
OOuuttppuutt ffrroomm pcahrieldnt
$ a.out
ooutput from parent
utput from child.
```

* To avoid race condition we can use TELL and WAIT func. as follows -

```
#include <sys/types.h>
static void charatatime (char *);
int main ()
{   pid_t pid;
    TELL-WAIT ();
    if ((pid = fork()) < 0)
        printf ("fork error");
    else if (pid == 0)
    {   WAIT_PARENT (); /* parent goes first */
        charatatime ("output from child");
    }
    else
    {   charatatime ("output from child");
```

```
static void charatatime(char *str)
{
    char *ptr;
    int c;
    setbuf(stdout, NULL);
                /* set unbuffered */
    for(ptr=str; c= *ptr++; )
        putc(c, stdout);
}
```

exec FUNCTIONS.

* exec system call causes a calling process to change its context and execute a different program.
* with fork, we can create a new process, and with exec, we can initiate new programs
* there are 6 versions of exec system call, they all have same func. but differ in its arg list.

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, .... /*(char *)0*/);
int execv(const char *pathname, char * const argv[]);
int execle(const char *pathname, const char * arg0, .../*(char *)0,*/
                                char * const envp[] */);
int execve(const char *pathname, char * const argv[],
                                char * const envp[]);
int execlp(const char *filename, const char *arg0, .../*(char *)0 */);
int execvp(const char *filename, char * const argv[]);
```

returns : nothing → on success
-1 → on error.

Differences.

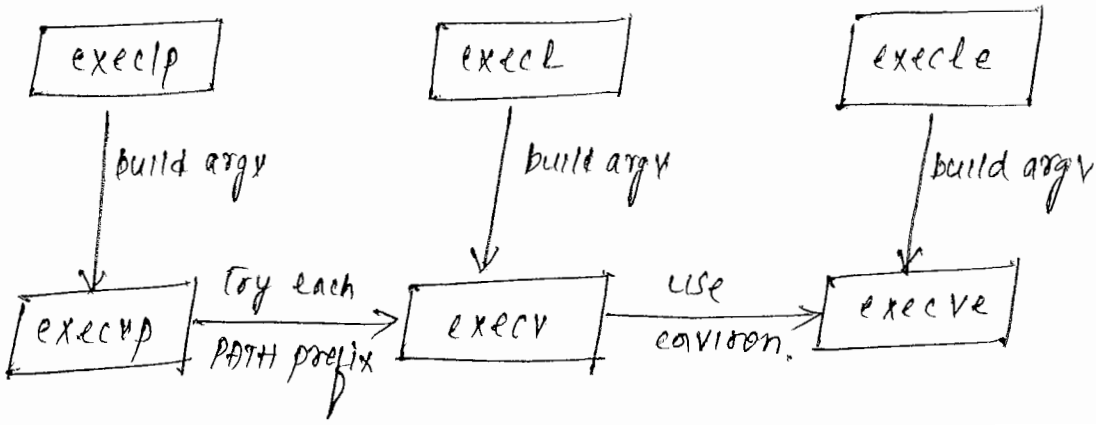
1. four takes pathname arg. other two take filename arg.
2. passing of the arg list (l -> list), the func execl, execle, execlp require each of the cmd line arg to the new prog.
3. passing the arg vector (v -> vector), have to build an array of pointer to the arg.
4. passing the arg list to the new prg to the func. execle and execve allow us to pass pointer to an array of pointers to the env. strings.

note:

→ the last arg to execl, execlp, execle must be NULL
ie (char *) 0
typecasting is compulsary. else 0 is interpreted as constant.

→ there are some limitation on total size of arglist & env. vector, this limit is given by ARG_MAX & its value can be upto 4096 ~~bytes~~.

* fig shows the rel^n ship b/w exec func.



ex:

1. To illustrate execl.

prg1.c

```
int main()
{ printf("first prg");
  printf("ID before exe %d ", getpid());
  printf("exec starts");
  execl("/usr/ise/prg2", "prg2", (char *)0);
}
```

prg2.c

```
int main()
{ printf("after exec");
  printf("pid is %d \n", getpid());
  printf("Exec ends");
}
```

2. To illustrate execv.

```
int main()
{ char *temp[3];
  temp[0] = "ls";
  temp[1] = "-l";
  execv("\bin\ls", temp);
}
```

## CHANGING USER IDS AND GROUP IDs

* We can set the real user ID and effective ~~usrid~~ user ID using setuid func.
We can set the real group ID and effective group ID using setgid func.

prototype -

```
#include <sys/types.h>
#include <unistd.h>
int setuid (uid_t uid);
int setgid (gid_t gid);
```

Returns: 0 → success
-1 → failure.

* Rules in order to change these IDs -

1. If a process has superuser privilege, the setuid func. sets the real uid, effective uid & saved-set-user id to uid.
2. If a process does not have superuser privilege, but uid equals either real uid or saved-set-uid, setuid sets only the effective uid to uid. the real uid & saved-set uid are not ~~use~~ changed.
3. If neither of these two conditions is true, errno is set to EPERM and an error is returned.

* Fig below shows how actually you can change the three user ids.

| ID | exec | | setuid(uid) | |
|---|---|---|---|---|
| | setuserid bit off | setuserid bit on | superuser | unprivileged user. |
| Real user ID | unchanged | unchanged | set to uid | unchanged |
| effective user ID | unchanged | set from user id of program file | set to uid | set to uid |
| saved-set user ID | copied from effective uid | copied from effective uid | set to uid | unchanged |

## setreuid and setregid functions

prototype

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

int setreuid (uid_t uid, uid_t euid);

int setregid (gid_t gid, gid_t egid);
```

returns : 0 → success
-1 → failure.

* using the above two func, we can swap the real user id and effective user id. Similarly real group id and effective group id.

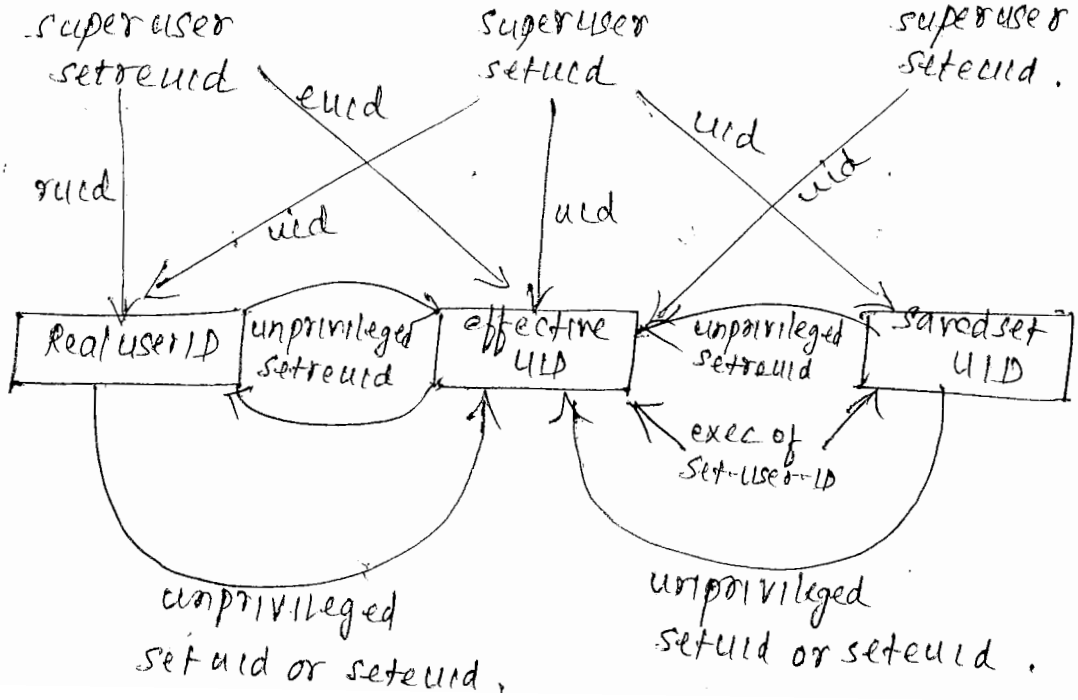
## seteuid and setegid functions.

* using these, we can change only effective uid & ~~eg~~ effective gid (only privileged user)

```
#include <sys/types.h>
#include <unistd.h>
int seteuid (uid_t uid);
int setegid (gid_t gid);
```

returns : 0 → success
-1 → failure.

summary.



# INTERPRETER FILES.

* Both SVR4 and BSD support interpreter files.
Interpreter files are text files that begins with a line of the form

#! pathname [ optional argument ]

↳ space is optional.

where pathname → normally an absolute pathname.

* the actual file that gets execed by the kernel is not the interpreter file, but the file specified by the pathname on the first line of the interpreter that is execed is termed as interpreter.

most common ex:
#! /bin/bash

ex:

```
#include <sys/types.h>
#include <sys/wait.h>
int main()
{ pid_t pid;
  switch ( pid=fork() )
  { case -1: printf ("fork error");
             exit(0);
    case 0 : if( execl("/home/ise/testinterp",
                       "testinterp", "arg1", "arg2",
                       ((char*)0) < 0 )
               printf ("error in exec");
  }
  if (waitpid(pid, NULL, 0) < 0 )
      printf ("error in waitpid");
  exit(0);
}
```

testinterp

```
#! /home/ise/echoarg hello
```

↳ interpreter

o/p

```
$a.out
argv[0] = /home/ise/echoarg
argv[1] : hello.
argv[2] : /home/ise/testinterp.
argv[3] : arg1
argv[4] : arg2 .
```

echoarg is an executabe file obtained by following prg.

```
#include <stdio.h>
int main (int argc , char *argv[] )
{ int i;
    for(i=0; i<argc; i++)
      printf(" argv[%d] = %s\n", i , argv[i]);
}
```

```
$cc -o echoarg filename.c .
```

Uses of interpreter files.

1. They hide the fact that certain programs are scripts in some other language.
2. They provide an efficiency gain.
3. They help us to write shell scripts using shells other than /bin/sh.

## system FUNCTION.

* used to execute a command string within a program.
* It is implemented by calling fork, exec, and waitpid.

prototype

```
#include <stdlib.h>
int system (const char *cmdstring);
```

returns:

-1 → if either fork fails or waitpid returns an error other than EINTR.

127 → if exec fails [as if shell has executed exit(127)]

termination status of shell → if all three func. succeed.

eg: implementation of system function.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
int system (const char *cmdstring)
{
    pid_t pid;
    int status;
    if(cmdstring == NULL)
        return(1);
    if((pid = fork()) < 0)
        status = -1;
    else if(pid == 0)    //child.
    {
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        exit(127); //execl error
    }
```

```
else
{
  while (waitpid(pid, &status, 0) < 0) // parent
    if (errno != EINTR)
    {
      status = -1;
      break;
    }
  }
  return(status);
}
```

USER IDENTIFICATION.

* Any process can find out its real & effective UID & GID.
Sometimes we want to find out login name of the users, who's running the prog.
We normally get it thru' getpwuid(getuid()), but suppose a single user has multiple login names, each with same UID, - we use getlogin() func.

prototype
```
#include <unistd.h>
char *getlogin(void);
```

returns: ptr to string giving login names → if OK
NULL pointer → on error.

note: this func. can fail if the process is not attached to a terminal that a user logged into. we call these processes as daemons.

PROCESS TIMES.

* execution time of a process has 3 values -
  - clock time
  - user cpu time
  - system cpu time.

prototype:

```
#include <sys/times.h>
clock_t times (struct tms *buf);
```

returns: elapsed wall clock time in clock ticks → if OK
-1 → on error.

```
struct tms
{ clock_t tms_utime; // user cpu time
  clock_t tms_stime; // system cpu time
  clock_t tms_cutime; // user cpu time, terminated children
  clock_t tms_cstime; // system ——— // ———————
};
```

PROCESS ACCOUNTING

* when process accounting is enabled, kernel writes an accounting record each time a process terminates.
* This accounting record is 32 bytes of binary data with name of the command, amount of cpu time used, the uid & gid the starting time & so on.
* super user executes acct on command with a pathname arg to ~~enable ac~~ enable accounting.
The pathname is usually /var/adm/pacct.
* The structure of the accounting records is defined in the header <sys/acct.h> and looks like -

```
struct acct
{ char ac_flag; // flag
  char ac_stat; // termination status
  uid_t ac_uid; // real UID
  gid_t ac_gid; // real GID
  dev_t ~~ac_tty~~ ac_tty; // controlling terminal.
```

```
time_t    ac_btime;   // starting calender time
comp_t    ac_utime;   // user cpu time
comp_t    ac_stime;   // system cpu time
comp_t    ac_etime;   // wall clock time
comp_t    ac_mem;     // avg m/m usage
comp_t    ac_io;      // bytes transfered by read & write
comp_t    ac_rw;      // block read or written
char      ac_comm[8]; // command name.
};
```

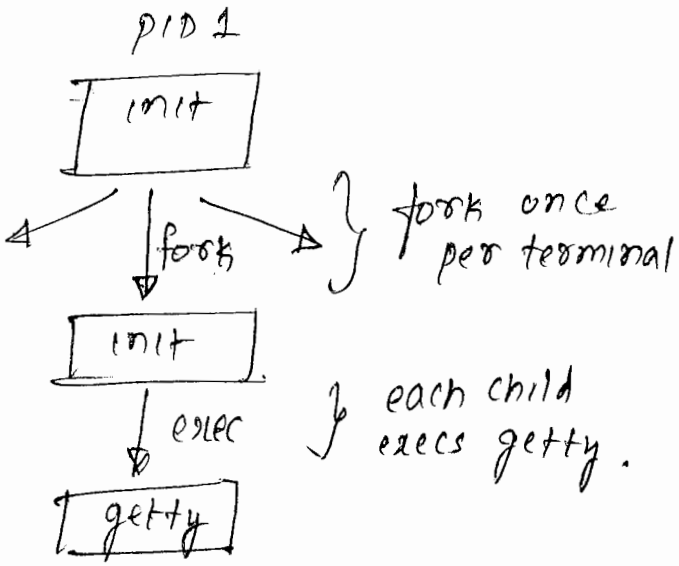
* The ac_flag member records certain events during the execution of process. They are -

AFORK → process is a result of fork, but never called exec.
ASU → process used super user privileges.
ACOMPAT → process used compatibility mode (VAX only)
ACORE → process dumped core.
AXSIG → process was killed by a signal.

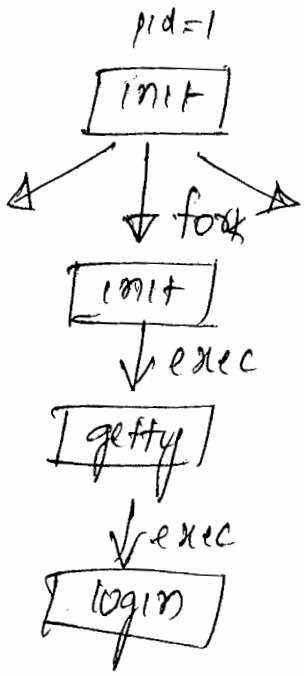
# process control.

## Terminal logins

### ~~4.3B~~ 4.3+ BSD Terminal logins




* when the system is bootstrapped, the kernel creates process with id=1; init that brings the system up multi user.
* init reads the file /etc/ttys (created by sys admin, it has one line per terminal device) & for every terminal device that allows a login, init does a fork followed by exec of the program getty.




} reads /etc/ttys
forks once per terminal
creates empty environment

} opens terminal device (fd's 0,1,2)
reads user name, initial
env. ~~set~~ is set.

* getty ~~invokes login, & it~~ o/ps something like -

login:

When user enters username, getty process invokes login program i.e to

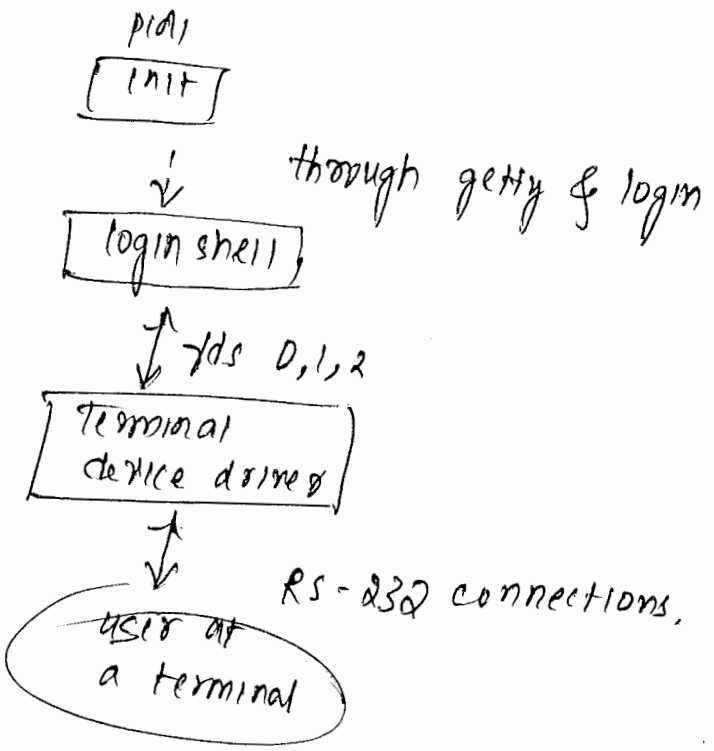
execle ("usr/bin/login", "login", "-p", username, (char *)0, envp );

After login is invoked, the state of these processes looks like

* then login calls getpwnam() to fetch our pw file entry. It then calls getpass(3) to display the prompt -

password:

reads our pw <echo disabled>.

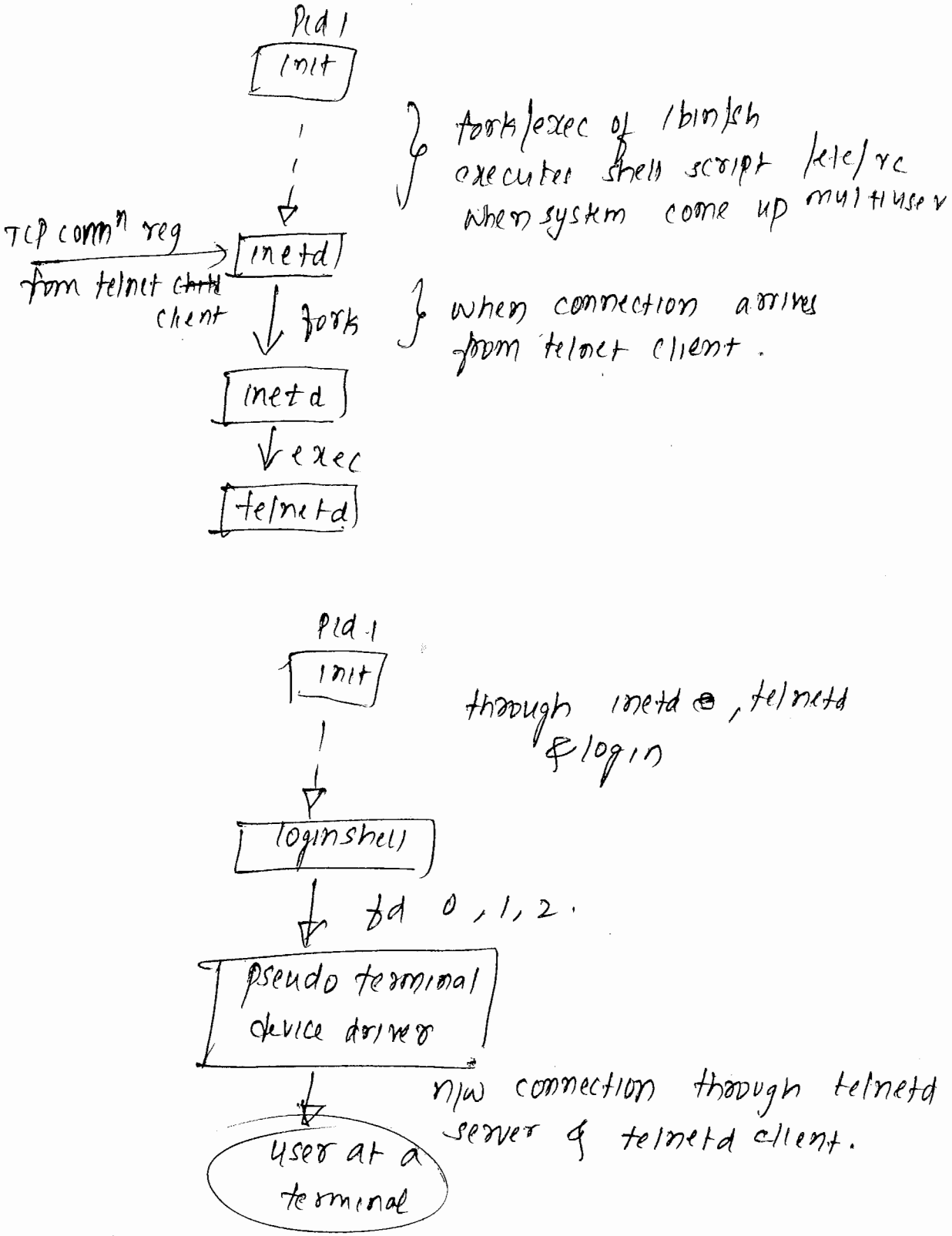



login shell now reads its start up files (usually changes some of the env. var. & may add some additional env. var) to the present list.

When start up files are done, we finally get the shell prompt ($) and can enter commands.

# network logins

## 4.3 + BSD network logins



## process groups

↳ collection of one or more processes.
↳ have unique process group ID (pid_t type)
↳ use getpgrp to retrieve it

```
pid_t getpgrp (void);   {sys/types.h>
                        <unistd.h>
```

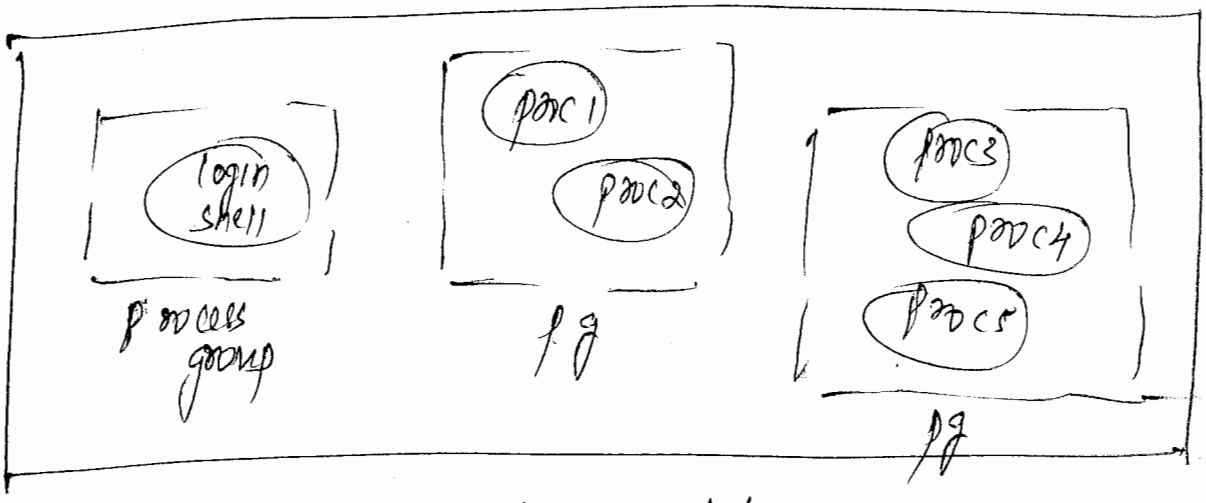
↳ have process leader
  ↳ pid = pgid.

```
int setpgid (pid_t pid, pid_t pgid);
```

↳ to join an existing / create an new pg.

## Sessions

↳ collection of one or more process groups.




these process groups can be generated as -

proc1 | proc2 &

proc3 | proc4 | proc5.

A process establishes a new session using

pid_t setsid(void);

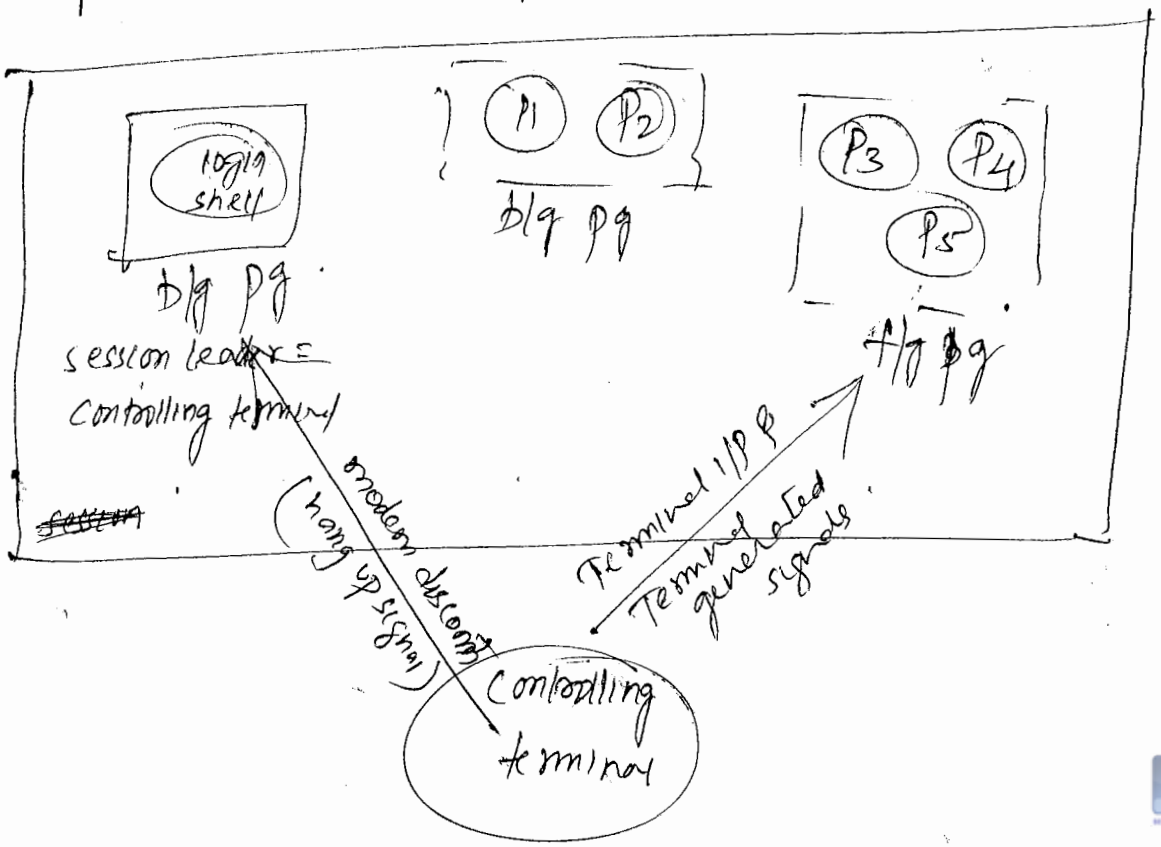
If calling process is not a pg leader, this func. creates new session. 3 things can happen -

1. process becomes session leader of this new session
2. process becomes pg leader of a new pg. new pg id is equal to pid.
3. process has no controlling terminal. If it is having, then the association is broken.

## Controlling terminal

↳ terminal through which the process has been created char. of sessions & process groups.

1. session can have a single controlling terminal. This is usually terminal device (in case of terminal login) or pseudo terminal device (in case of n/w login).
2. session leader establishes the connection to the controlling terminal is called the controlling process.
3. process groups within a session can be divided into a single foreground process group & all other as b/g pg.
4. whenever we type our terminal interrupt keys (ctrl c & ctrl d & DELETE). this causes either interrupt signal or Quit signal to send to all processes in f/g pg.
5. If a modem disconnect is detected by terminal interface, the hang up signal is sent to controlling process (session leader)



pid_t tcgetpgrp (int filedes);
returns pid of fg process on o/c -1, error. —8—

int tcsetpgrp (int filedes, pid_t ~~pg~~ ~~pg~~ pgrpid);
↳ to set fg process.

---

↳ collection of process, / pipeline of processes.

## JOB CONTROL.

* Allows the user to start multiple jobs from a single terminal & control which jobs can access the terminal & which jobs are to run in bg.
* It needs 3 forms of support -
  1. Shell that supports job control
  2. Terminal driver in the kernel must ~~cont~~ support JC
  3. support for certain JC signals must be provided.
* When we start a bg job, the shell assigns it a job identifier & prints one or more PIDs

```
$ make all &
  [1] 1475
$ pr *.c | lpr &
  [2] 1490
$ ↵
[2] + Done      pr *.c | lpr &
[1] + Done      make all &.
```