CMR
INSTITUTE OF
TECHNOLOGY

USN | | | | | | | | | | |

Internal Assesment Test - II

Sub:	**Unix System Programming**							Code:	**10CS62**
Date:	08 /05/2017	Duration:	90 mins	Max Marks:	50	Sem:	VI	Branch:	**ISE**

Answer Any FIVE FULL Questions

		Marks	OBE	
			CO	RBT
1 (a)	Discuss the file and record locking in Unix system. Explain the fcntl API for file locking.	[10]	CO3	L2
2 (a)	List and explain the different forms of exec function with prototype declaration along with meaning and diagram that shows relationship among them. Write a program to echo all its command line arguments and environment variable	[10]	CO4	L3
3 (a)	What is process accounting? Explain with neat diagram? Write a program to illustrate the generation of accounting data	[10]	CO4	L3
4 (a)	What are pipes? What are its limitations? Write a program to create a pipe between parent and its child to send the data down the pipe	[10]	CO4	L2
5 (a)	Write short notes on the following: a) Message Queues b) Semaphores	[10]	CO4	L2
6 (a)	With neat diagram explain inter process communication using FIFO.	[6]	CO4	L3
(b)	Write a ‘C’ program to illustrate the concept of co-processes	[4]	CO4	L2
7 (a)	What is job control? Summarize the job control features with help of neat diagram	[5]	CO4	L2
7 (b)	Explain terminal login and network login, with suitable diagram	[5]	CO4	L3

**********All the best**********

#132, AECS Layout, IT Park Road, Kundalahalli, Bangalore – 560 037, T:+9180 28524466 / 77

CMR
INSTITUTE OF
TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE & ENGINEERING

Scheme and Solution for USP(10CS62) –IAT2 2017

Q. 1 Discuss the file and record locking in Unix system. Explain the fcntl API for file locking

Explanation- 2M

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.
- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- The differences between the read lock and the write lock is that when write lock is set, it prevents the other process from setting any over-lapping read or write lock on the locked file.
- Similarly when a read lock is set, it prevents other processes from setting any overlapping write locks on the locked region.
- The intension of the write lock is to prevent other processes from both reading and writing the locked region while the process that sets the lock is modifying the region, so write lock is termed as "**Exclusive lock**".
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a read lock is also called as "**shared lock** ".
- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file, no process can use the read or write system calls to access the data on the locked region.

Prototype 1M

#include<fcntl.h>

int ***fcntl***(**int** fdesc, **int** cmd_flag, ...);

Flags-1M

F_SETLK sets a file lock, do not block if this cannot succeed immediately.

F_SETLKW sets a file lock and blocks the process until the lock is acquired.

F_GETLK queries as to which process locked a specified region of file.

Structure flock 1M

struct flock {

short l_type; /* what lock to be set or to unlock file */

short l_whence; /* Reference address for the next field */

off_t l_start ; /*offset from the l_whence reference addr*/

off_t l_len ; /*how many bytes in the locked region */

pid_t l_pid ; /*pid of a process which has locked the file */

l_type value Use 1M

F_RDLCK Set a read lock on a specified region

F_WRLCK Set a write lock on a specified region

F_UNLCK Unlock a specified region

l_whence value Use 1M

SEEK_CUR The l_start value is added to current file pointer address

SEEK_SET The l_start value is added to byte 0 of the file SEEK_END

The l_start value is added to the end of the file

Program – 3M

```
#include <unistd.h>
 #include<fcntl.h> int main ( )
 {
 int fd; struct flock lock;
 fd=open("divya",O_RDONLY);
lock.l_type=F_RDLCK;
lock.l_whence=0;
lock.l_start=10;
 lock.l_len=15;
fcntl(fd,F_SETLK,&lock);
}
```

Q. 2 List and explain the different forms of exec function with prototype declaration along with meaning and diagram that shows relationship among them. Write a program to echo all its command line arguments and environment variable

Explanation 2M

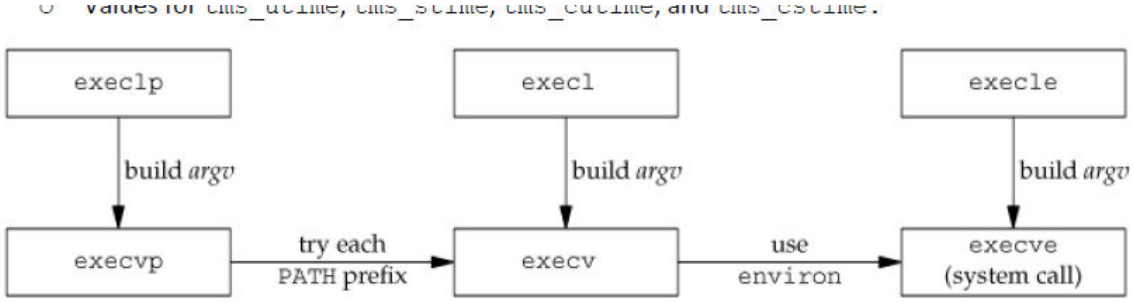
When a process calls one of the `exec` functions, that process is completely replaced by the new program, and the new program starts executing at its `main` function. The process ID does not change across an `exec`, because a new process is not created; `exec` merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk.
There are 6 exec functions:

All forms of exec function-2M

```
#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 []);
```

All six return: -1 on error, no return on success.

Diagram-2M



Relationship of the six exec functions

Program-2M

Example of exec functions

```
#include "apue.h"
#include <sys/wait.h>

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
            err_sys("execlp error");
    }

    exit(0);
}
```

Program-2M

Echo all command-line arguments and all environment strings

```
#include "apue.h"

Int main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

Q. 3 What is process accounting? Explain with neat diagram? Write a program to illustrate the generation of accounting data

Explanation 2M

Most UNIX systems provide an option to do process accounting. When enabled, the kernel writes an accounting record each time a process terminates.

These accounting records are typically a small amount of binary data with the name of the command, the amount of CPU time used, the user ID and group ID, the starting time, and so on.

A superuser executes accton with a pathname argument to enable accounting.

The accounting records are written to the specified file, which is usually /var/account/acct. Accounting is turned off by executing accton without any arguments.

The data required for the accounting record, such as CPU times and number of characters transferred, is kept by the kernel in the process table and initialized whenever a new process is created, as in the child after a `fork`.

Each accounting record is written when the process terminates.

This means that the order of the records in the accounting file corresponds to the termination order of the processes, not the order in which they were started.

Structure 2M

The structure of the accounting records is defined in the header `<sys/acct.h>` and looks something like

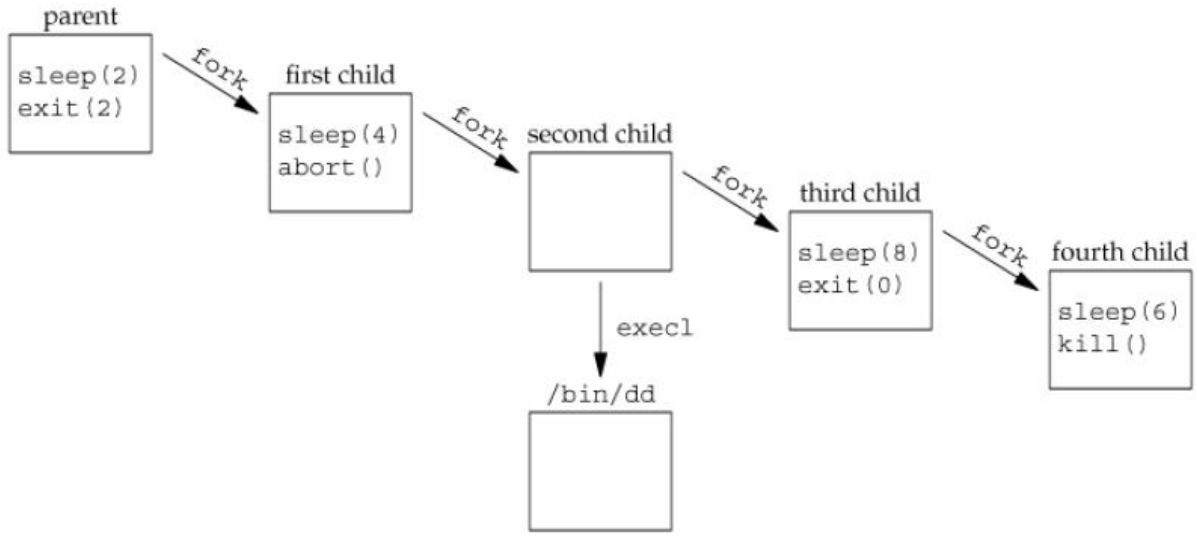
```
typedef  u_short comp_t;   /* 3-bit base 8 exponent; 13-bit fraction */

struct  acct
{
  char    ac_flag;      /* flag  */
  char    ac_stat;      /* termination status (signal & core flag only) */
                        /* (Solaris only) */
  uid_t   ac_uid;       /* real user ID */
  gid_t   ac_gid;       /* real group ID */
  dev_t   ac_tty;       /* controlling terminal */
  time_t  ac_btime;     /* starting calendar time */
  comp_t  ac_utime;     /* user CPU time (clock ticks) */
  comp_t  ac_stime;     /* system CPU time (clock ticks) */
  comp_t  ac_etime;     /* elapsed time (clock ticks) */
  comp_t  ac_mem;       /* average memory usage */
  comp_t  ac_io;        /* bytes transferred (by read and write) */
                        /* "blocks" on BSD systems */
  comp_t  ac_rw;        /* blocks read or written */
                        /* (not present on BSD systems) */
  char    ac_comm[8];   /* command name: [8] for Solaris, */
                        /* [10] for Mac OS X, [16] for FreeBSD, and */
                        /* [17] for Linux */
};
```

Program 3M

Program to generate accounting data

```
#include "apue.h"

Int main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {        /* parent */
        sleep(2);
        exit(2);                /* terminate with exit status 2 */
    }

                                /* first child */
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid != 0) {
        sleep(4);
        abort();                /* terminate with core dump */
    }

                                /* second child */
   if ((pid = fork()) < 0)
       err_sys("fork error");
   else if (pid != 0) {
       execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
       exit(7);                 /* shouldn't get here */
   }

                                /* third child */
   if ((pid = fork()) < 0)
       err_sys("fork error");
   else if (pid != 0) {
       sleep(8);
       exit(0);                 /* normal exit */
   }

                                /* fourth child */
   sleep(6);
   kill(getpid(), SIGKILL);     /* terminate w/signal, no core dump */
   exit(6);                     /* shouldn't get here */
}
```

Diagram 3M



Process structure for accounting example

Q. 4 What are pipes? What are its limitations? Write a program to create a pipe between parent and its child to send the data down the pipe

Pipe definition + explanation -2M

Limitation -2M

Pipe prototype- 1M

Pipes are the oldest form of UNIX System IPC. Pipes have two limitations.

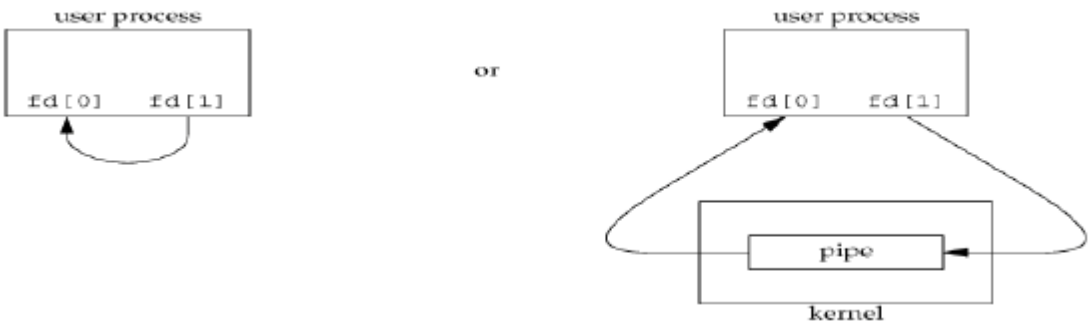
- Historically, they have been half duplex (i.e., data flows in only one direction).
- Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls `fork`, and the pipe is used between the parent and the child.

A pipe is created by calling the `pipe` function.

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns: 0 if OK, 1 on error.

Two file descriptors are returned through the filedes argument: filedes[0] is open for reading, and filedes[1] is open for writing. The output of filedes[1] is the input for filedes[0].



Program -5M

```
int
main(void)
{
    int     n;
    int     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);
}
```

Q. 6 Write short notes on the following:

a) Message Queues b) Semaphores

Explanation 1M

Structure msqid_ds-1M

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. We'll call the message queue just a queue and its identifier a queue ID.

A new queue is created or an existing queue opened by `msgget`. New messages are added to the end of a queue by `msgsnd`. Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to `msgsnd` when the message is added to a queue. Messages are fetched from a queue by `msgrcv`. We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Each queue has the following `msqid_ds` structure associated with it:

```
struct msqid_ds
{
        struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
        msgqnum_t        msg_qnum;     /* # of messages on queue */
        msglen_t         msg_qbytes;   /* max # 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 */
        .
        .
        .
};
```

Function msgget prototype-1M

The first function normally called is `msgget` to either open an existing queue or create a new queue.

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

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

Returns: message queue ID if OK, 1 on error

Function msgctl prototype-1M

The `msgctl` function performs various operations on a queue.

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf );
```

Returns: 0 if OK, 1 on error.

Function msgsnd prototype- 1M

Data is placed onto a message queue by calling `msgsnd`.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

Returns: 0 if OK, 1 on error.

Q. 5 b)

Definition-1M

A semaphore is a counter used to provide access to a 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 positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
3. Otherwise, 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.

Semaphore structure-1M

The kernel maintains a `semid_ds` structure for each semaphore set:

```
struct semid_ds {
    struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
    unsigned short   sem_nsems; /* # of semaphores in set */
    time_t           sem_otime; /* last-semop() time */
    time_t           sem_ctime; /* last-change time */
    .
    .
    .
};
```

Function prototype 1M* 3=3M

The first function to call is semget to obtain a semaphore ID.

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

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

Returns: semaphore ID if OK, 1 on error

The semctl function is the catchall for various semaphore operations.

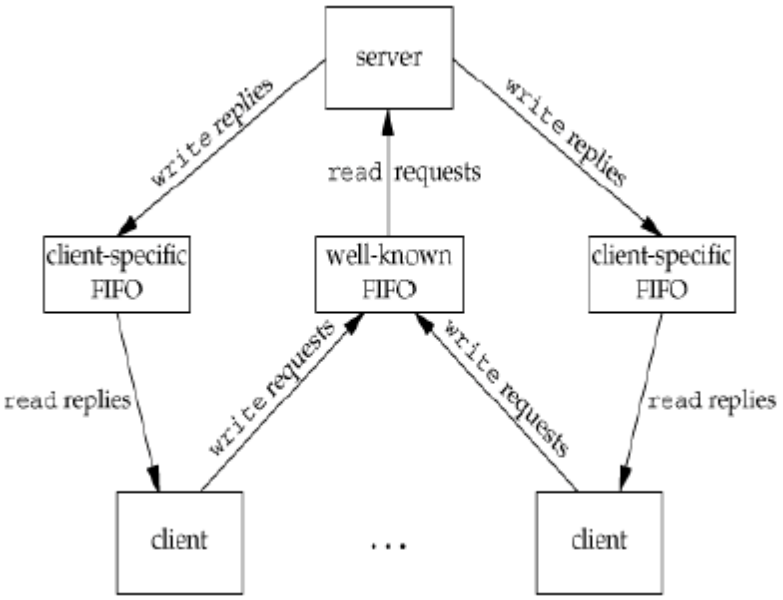
```
#include <sys/sem.h>

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

The function `semop` atomically performs an array of operations on a semaphore set.

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

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

Q. 6. A) With neat diagram explain inter process communication using FIF

Digram-2M



Explanation-3M

FIFO's can be used to send data between a client and a server. If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates. Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.

This prevents any interleaving of the client writes. The problem in using FIFOs for this type of client server communication 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 versus responses for other clients. One solution 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'sprocess ID.

For example, the server can create a FIFO with the name /vtu/ ser.XXXXX, where XXXXX is replaced with the client's process ID. This arrangement works, although it is impossible for the server to tell whether a client crashes. This causes the client-specific FIFOs to be left in the file system.

The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.

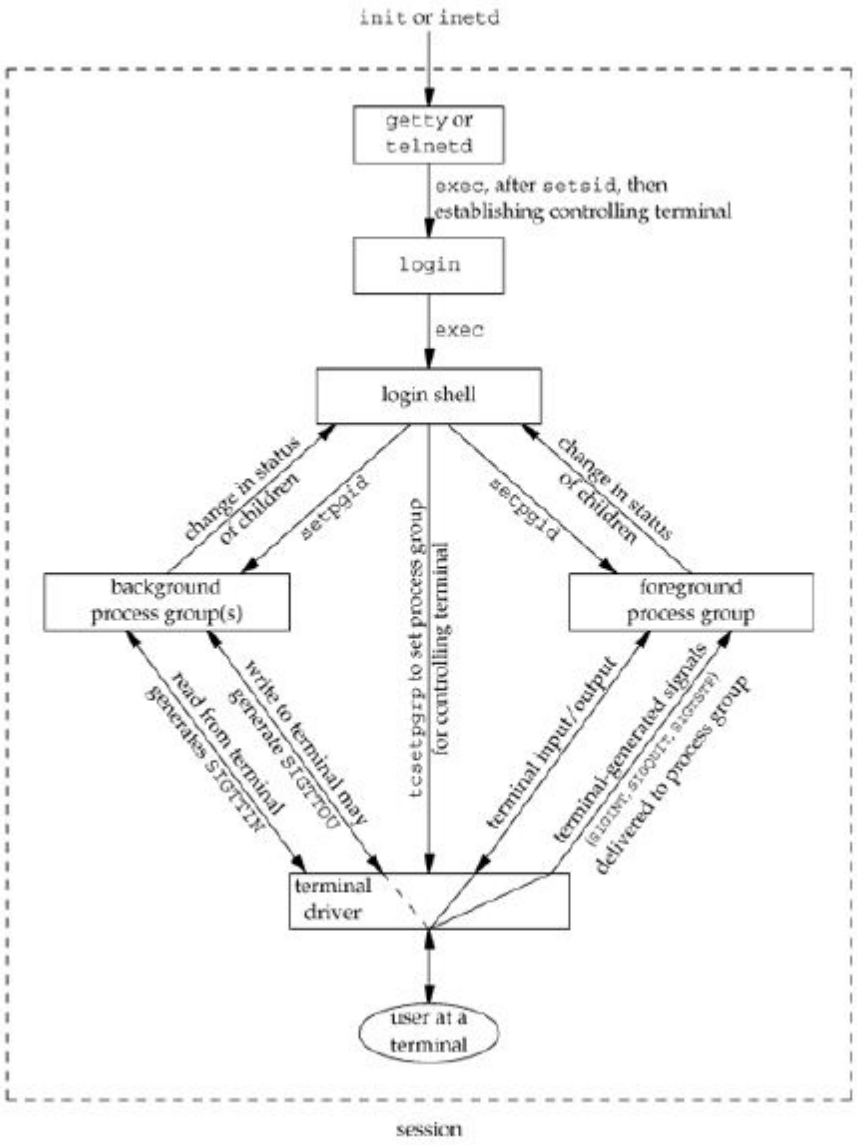
Q. 6 b) Write a 'C' program to illustrate the concept of co-processes

```
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\n", int1 + int2);
            n = strlen(line);
            if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13)
                err_sys("write error");
        }
    }
    exit(0);
}
```

Q. 7 a) What is job control? Summarize the job control features with help of neat diagram

Diagram 3M



Explanation -2M

This feature allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background. Job control requires three forms of support:

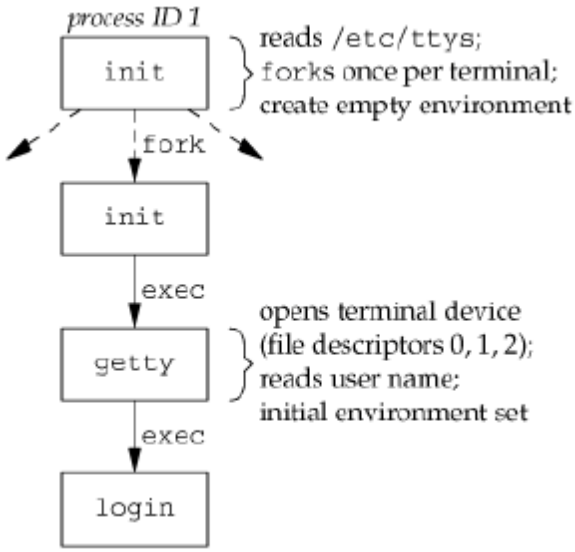
- A shell that supports job control
- The terminal driver in the kernel must support job control
- The kernel must support certain job-control signals

This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal. The following demonstrates this:

```
$ cat > temp.foo &          start in background, but it'll read from standard input
   [1]     1681
   $                           we press RETURN
   [1] + Stopped (SIGTTIN)     cat > temp.foo &
   $ fg %1                     bring job number 1 into the foreground
   cat > temp.foo              the shell tells us which job is now in the foreground

   hello, world                enter one line

   ^D                          type the end-of-file character
   $ cat temp.foo              check that the one line was put into the file
   hello, world
```

Q. 7 b) Explain terminal login and network login, with suitable diagram

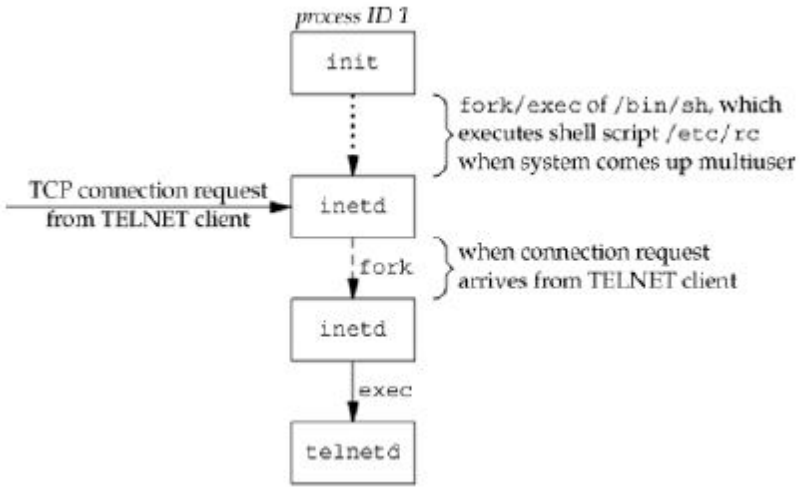
Terminal login-2.5M



The terminals were either local (directly connected) or remote (connected through a modem). In either case, these logins came through a terminal device driver in the kernel.

The system administrator creates a file, usually `/etc/ttys`, that has one line per terminal device. Each line specifies the name of the device and other parameters that are passed to the `getty` program. One parameter is the baud rate of the terminal, for example. When the system is bootstrapped, the kernel creates process ID 1, the `init` process, and it is `init` that brings the system up multiuser. The `init` process reads the file `/etc/ttys` and, for every terminal device that allows a login, does a `fork` followed by an `exec` of the program `getty`. This gives us the processes shown in Figure 9.1.

Network login



The `telnetd` process then opens a pseudo-terminal device and splits into two processes using `fork`. The parent handles the communication across the network connection, and the child does an `exec` of the `login` program. The parent and the child are connected through the pseudo terminal. Before doing the `exec`, the child sets up file descriptors 0, 1, and 2 to the pseudo terminal. If we log in correctly, `login` performs the same steps we described in Section 9.2: it changes to our home directory and sets our group IDs, user ID, and our initial environment. Then `login` replaces itself with our login shell by calling `exec`. Figure 9.5 shows the arrangement of the processes at this point.