Assignment # 3

Module 4 & Module 5

① what are stream pipe? what are the different ways to view stream pipe?

A stream pipe is a UNIX interprocess communication (IPC) facility that allows processes on the same computer to communicate with each other.

Advantages:-

- Unlike shared-memory connections, stream pipes do not pose the security risk of being overwritten or read by other programs that explicitly access the same portion of shared memory.
- unlike shared memory connections, stream-pipe connections allow distributed transactions between database servers that are on the same computer.

Disadvantages:-

- stream pipe connections might be slower than shared memory connections on some computers.
- stream pipes are not available on all platforms.
- When you use shared memory or stream pipes for client/server communications, the hostname entry is ignored.

You can view socketpairs as an extension of pipes. different ways to view stream pipes are :-

1) pair of connected sockets for one-way stream communication
2) pair of connected sockets for two-way stream communication

→ stream communication takes place across a connection b/w two sockets. It's reliable, error-free, & as with pipes, there are no message boundaries.

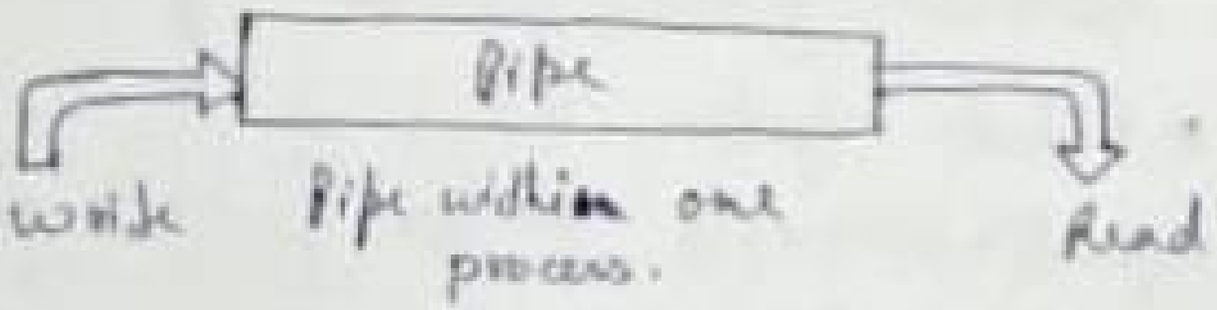
Reading from stream may get data sent by one or more calls to the write() function. In some cases, the data received from a read() call is only part of the data that was written.

The protocol implementing stream communication retransmits messages that are received with errors. It also reports error when a process tries to send a message after the connection has been broken.

→ Forming a stream connection is asymmetrical: one process requests a connection with particular sockets, the other process accepts connection requests. Before a connection can be accepted, a socket must be created & an address bound to it.

→ This system call would create a pipe for one-way communication i.e. it creates two descriptors, first one is connected to read from the pipe & the other one is connected to write into the pipe.

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



② Explain with example:

i) Message Queues:- A message queue is a linked list of messages stored within the Kernel & identified by a message queue identifier. We'll call the message queue just a queue & its identifier a queue ID.

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

```
struct msqid_ds
{
  struct ipc_perm   msg_perm;   /* see usedion IPC-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 */
};
```

This structure defines the current status of the queue.

*) 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);
```

*) The msgctl function performs various operation on queue.

```
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds * buf);
```

cmd argument specifies the command to be performed on queue

→ IPC_RMID: removes the message queue id & destroys msqid_ds
→ IPC_SET: set members of msqid_ds data from buf.
→ IPC_STAT: set copy members of msqid_ds DS from buf.

*) 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);
```

• Messages are retrieved from a queue by msgrcv.

#include <sys/msg.h>

ssize_t msgrcv(int msqid, void * ptr, size_t nbytes, long type
, int flag);

ii) Semaphores :— A semaphore is a counter used to provide access to a shared object for multiple processes.

When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.

A common form of semaphore is called binary semaphore. It controls a single resource, & its value is initialized to 1. A semaphore can be initialized to any positive value, with a value indicating how many units of shared resource are available for sharing.

• The first function to call is semget to obtain semaphore ID

#include <sys/sem.h>

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

• The semctl function is the catchall for various semaphore operations.

#include <sys/sem.h>

int semctl (int semid, int semnum, int cmd, ...);

• 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);

→ nops argument specifies the no. of operations in array.
→ semoparray[] is a pointer to an array of semaphore operations, represented by sembuf structures.

```
struct semBuf {
{ unsigned short sem_num;
  short         sem_op;
  short         sem_flg;
};
```

(3) what are signals? Mention different source of signal. write a program to setup signals handlers for SIGINT & SIGALRM?

Signals are software interrupts. Signals provide a way of handling asynchronous events: a user at a terminal typing the interrupt key to stop a program or the next program in a pipeline terminating prematurely.

```
# include <signal.h>

void ( * signal (int sig_no, void ( * handler)(int))) (int);
```

→ sig_no is a signal identifier like SIGINT or SIGALRM.
→ The handler argument is the function pointer of a user-defined signal handler function.

Different source of signal are :—

Name	Description	Default-action
SIGABRT	abnormal termination (abort)	terminate + core
SIGALRM	timer expired (alarm)	terminate
SIGBUS	hardware fault	terminate + core
SIGCANCEL	Threads library internal use	ignore
SIGCHLD	change in status of child	ignore
SIGILL	illegal instruction	terminate + core
SIGINT	terminal interrupt character	terminate
SIGINFO	status request from keyboard	ignore
SIGKILL	termination	terminate
SIGPIPE	write to pipe with no readers	terminate
SIGQUIT	terminal quit character	terminal + core.

Program:

```
# include <iostream.h>
# include <signal.h>
/* signal handler function */
void catch_sig ( int sig_num)
{
    signal ( sig_num, catch_sig);
    cout << " catch_sig : " << sig_num << endl;
}
/* main function */
int main ()
{
    signal ( SIGTERM, catch_sig);
    signal ( SIGINT, SIG_IGN);
    signal ( SIGALRM, SIG_DFL);
    pause ();
}
```

④ Explain the Kill() & alarm() API?

→ Kill():- A process can send a signal to a related process via the Kill API. This is a simple means of inter-process communication or control.

```
# include <signal.h>
int Kill ( pid_t pid, int signal_num);
```

→ signal_num argument is the integer value of a signal to be sent to one or more processes designated by pid.

pid > 0 ⇒ signal is sent to the process whose process ID is pid.
pid == 0 ⇒ signal is sent to all processes whose process group ID equals the process group ID of sender.
pid < 0 ⇒ signal is sent to all processes whose process group ID equals the absolute value of pid.
pid == 1 ⇒ signal is sent to all processes on system for which sender has permission to send signal.

The Unix Kill command invocation syntax is:

KILL [-<signal_num>] <pids> ....

where signal_num can be integer number or the symbolic name of a signal. <pids> is process ID.

→ ALARM():- The alarm API can be called by a process to request the Kernel to send the SIGALRM signal after a certain number of real clock seconds. The function prototype is:

```
# include <signal.h>
unsigned int alarm (unsigned int time_interval);
```

(5) Explain the sigsetjmp & siglongjmp functions with an example?

The function prototype of the APIs are:-

```
# include <setjmp.h>
int sigsetjmp (sigjmp_buf env, int savemask);
int siglongjmp (sigjmp_buf env, int val);
```

The sigsetjmp & siglongjmp are created to support signal mask processing.

The only difference b/w these functions and the setjmp & longjmp functions is that sigsetjmp has an additional argument. If savemask is non zero, then sigsetjmp also saves the current signal mask of the process in env. When siglongjmp is called, if the env argument was saved by a call to sigsetjmp with a non zero savemask, then siglongjmp restores the saved signal mask.

```
# include <iostream.h>
# include <stdio.h>
# include <unistd.h>
# include <signal.h>
# include <setjmp.h>
```

```
sigjmp_buf  env;
void callme (int sig_num)
{
    cout << " catch signal: " << sig_num << endl;
    siglongjmp(env, 2);
}
int main()
{
  struct sigaction action, old_action;
  sigemptyset (& sigmask);
  if (sigaddset (& sigmask, SIGTERM) == -1)
        perror ("set signal mask");
    sigemptyset ( & action.sa_mask);
    sigaddset ( & action.sa_mask, SIGSEGV);
    action.sa_handler = (void (*) ()) callme;
    action.sa_flags = 0;
  if (sigaction (SIGINT, & action, & old_action) == -1)
        perror ("sigaction")
  if (sigsetjmp(env,1) != 0)
    {
      cerr << " return from signal interruption ";
      return 0;
    }
  else
      cerr << " return from first time sigsetjmp is called";
      pause();
}.
```

(6) What is FIFO ? With a neat diagram, explain client server communication using FIFO ?

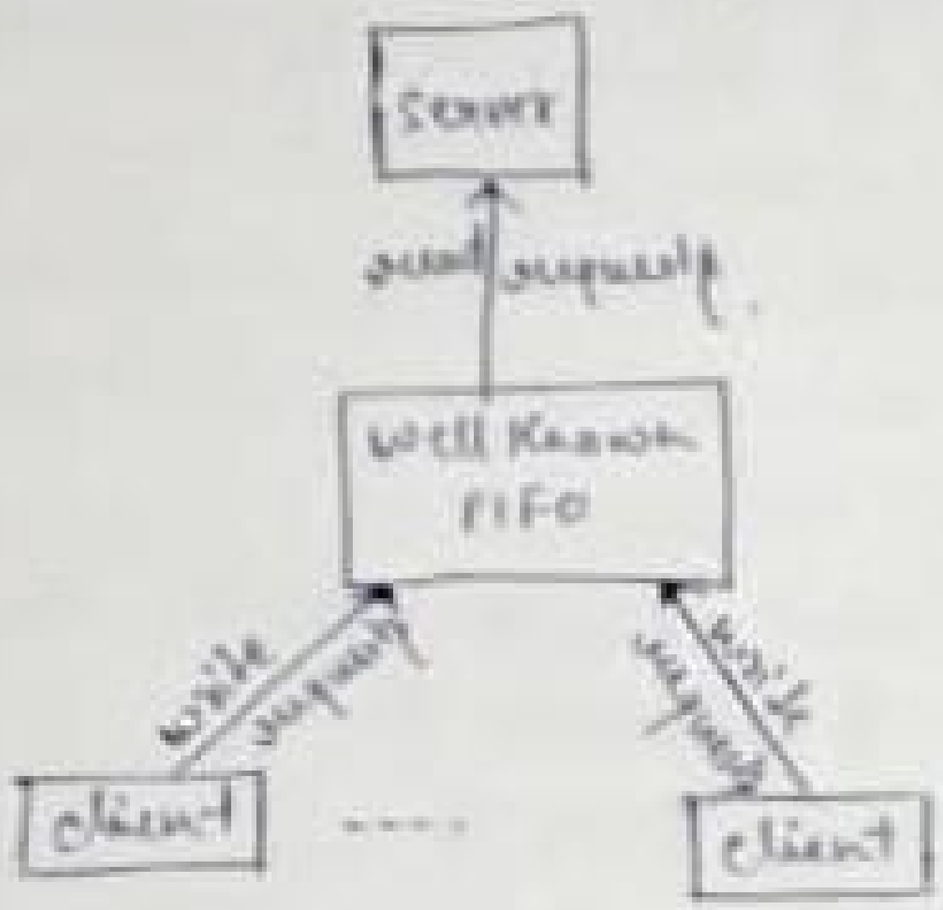
FIFOs are sometimes called named pipes. Pipes can be used only b/w related processes when a common ancestor has created the pipe.

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

There are two uses for FIFOs:-

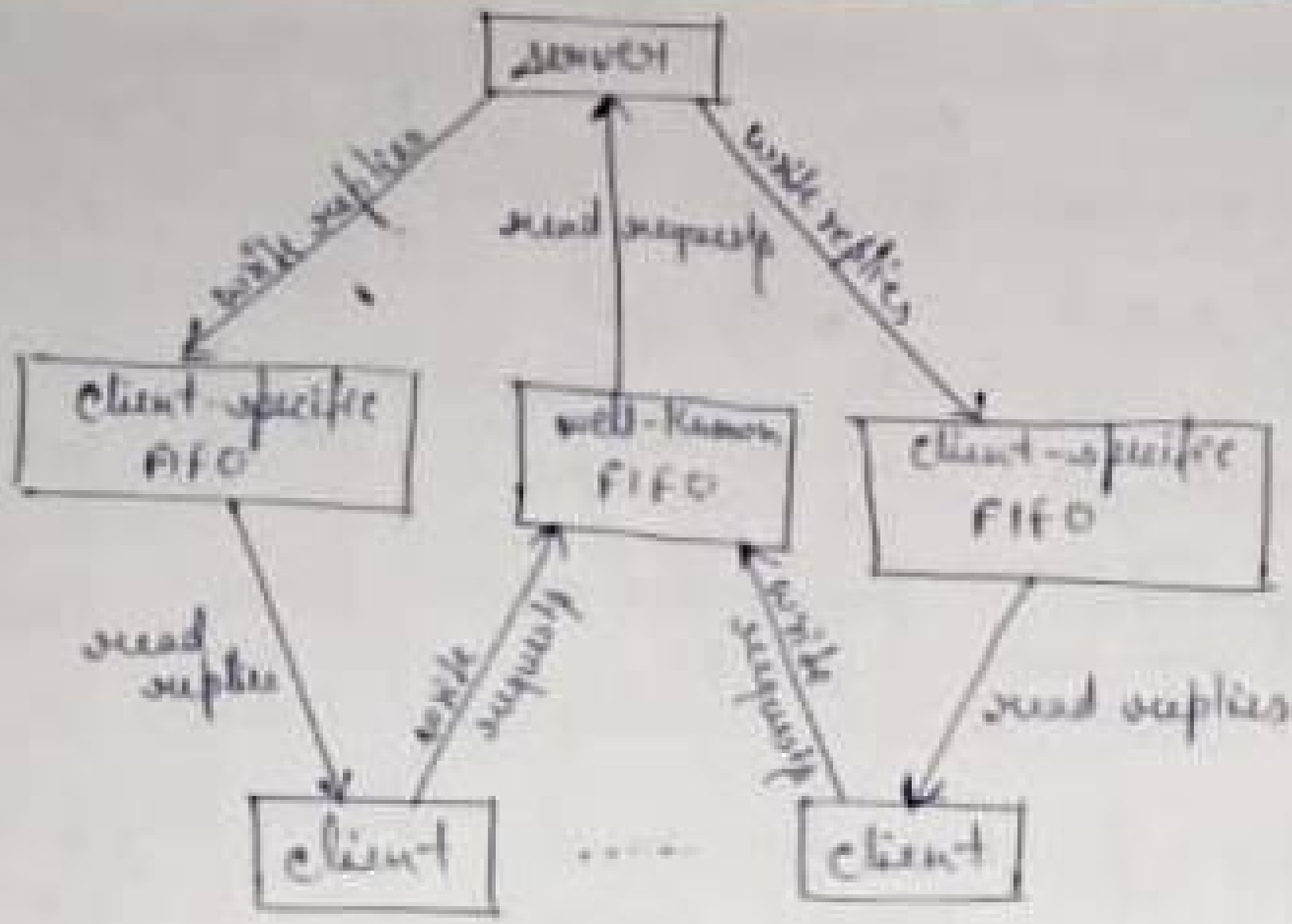
→ FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

→ FIFOs are used as rendezvous points in client-server applications to pass data b/w the clients & servers.



(client-sending sequence to a server using FIFO)

Eg
- a) FIFOs can be used to send data between a client & 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.
- b) This prevents any interleaving of the client writes.
- c) A single FIFO can be used, as clients would server know when to send their response versus responses for other clients.
- d) For example, the server can create a FIFO with name /tmp/server.pid, where pid is replaced with the clients process ID.



(client-Server Communication using FIFO)

(3) Explain setuid & setgid functions with example & explain various ways to change user id?

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

setuid - sets the user id
setgid - set the group id.

There are rules for who can change the ids.

→ If the process has superuser privileges, the setuid function sets the real user id, effective user id, & saved set-user-ID to uid.

→ if the user process does not have superuser privileges, but uid equals either the real user ID or the saved set-user-ID setuid sets only the effective user ID to uid. The real user ID & the saved set-user-ID are not changed.

→ If neither of these two conditions is true, errno is set to EPERM & -1 is returned.

ID	exec		setuid(pid)	
	set-user-ID off	set-user-ID on	superuser	unprivileged
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 user ID	copied from effective user ID	set to uid	unchanged

The above figure summarizes the various ways these user ID's can be changed.

⑤ What are Interpreter files? Give the difference between Interpreter files & Interpreters?

Interpreter files are text files that begin with a line of the form #! pathname [optional-argument]

The space between the exclamation point & the pathname is optional. The most common of these Interpreter files begin with the file.

#!/bin/sh

The pathname is normally an absolute pathname, since no special operations are performed on it. The recognition of these files is done within the Kernel as part of processing the exec system call. The actual file that gets executed by the kernel is not the Interpreter file, but the file specified by the pathname on the first line of the interpreter file.

Be sure to differentiate between the interpreter file text file that begins with #! and the interpreter, which is specified by the pathname on the first line of the interpreter file.

Be aware that systems place a size limit on the first line of an interpreter file. This limit includes the #!, the pathname,

the optional argument, the terminating newline, & any spaces.

(9) What are Daemon Process? Explain their characteristics?

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

Daemon Characteristics:-

- Daemons run in background
- Daemons have super-user privilege.
- Daemons don't have controlling terminal.
- Daemons are session & group leaders.

(10) What is error logging explain? Also explain single Instance Daemons?

The problem is decision when is how to handle error messages. It can't simply write to standard error, since it shouldn't have a controlling terminal. A central daemon error-logging facility is required.

There ways to generate log messages:-

→ Kernel routines can call the log function. These messages can be read by any user process that opens & reads the /dev/klog device.

→ Most user processes (daemons) call the syslog(3) function to generate log messages. This causes the message to be sent to the UNIX-domain datagram socket /dev/log.

→ A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514.

```
#include <syslog.h>
void openlog(const char *ident, int option, int facility);
void syslog(int priority, const char *format, ...);
void closelog(void);
int setlogmask(int maskpri);
```

Single-Instance Daemons:-

Some daemons are implemented so that only a single copy of the daemon should be running at a time for proper operation. The file & record locking mechanism provide the basis for one way to ensure that only one copy of a daemon is running. If each daemon creates a file & places a write lock on the entire file, only one such write lock will be allowed to be created. Successive attempts to create write locks will fail, serving as an indication to successive copies of the daemon that another instance is already running.

File & Record locking provides a convenient mutual-exclusion mechanism. If the daemon obtains a write-lock on an entire file, the lock will be removed automatically if the daemon exits. This simplifies recovery, removing the need for us to clean up from the previous instance of the daemon.

— x — x —