27/11/22

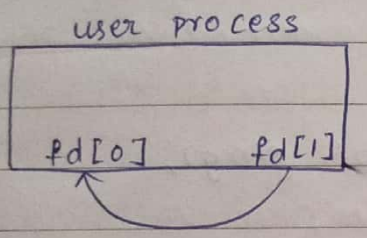
Assignment No. 3

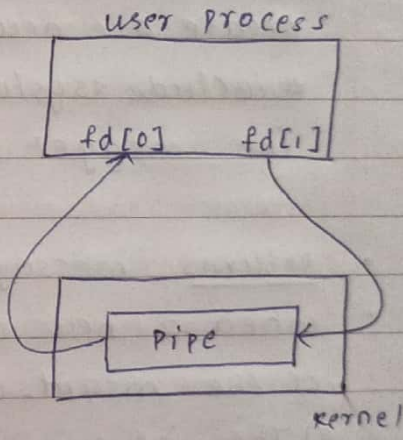
1. what are stream pipe? what are the different ways to view stream pipe?

Ans

stream pipe

- A stream pipe in a UNIX interprocess communication (IPC) facility that allows processes on the same computer to communicate with each other.
- There are two ways to view the stream pipe
  a) The two ends of the pipe connected in a single process
  (b) The data in the pipe flows through the kernel.





2. Explain with example ii Message Queue iii Semaphore?

Ans

Message Queue

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- We will call the message queue just a queue and its identifier a queue ID
- Each queue has the following msqid_ds structure

associated with it:

```
struct msqid_ds
{
  struct ipc_perm   msg_perm;
  msgqnum_t         msg_qnum;   /* no. of messages on queue */
  msglen_t          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 */
    ;
};
```

1. msgget function - 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.
- when a new queue is created, the following members of the msqid_ds structure are initialized

a) The ipc_perm structure is initialized
b) msg_qnum, msg_lspid, msg_lrpid, msg_stime and msg_rtime are all set to 0
c) msg_ctime is set to the current time
d) msg_qbytes is set to the system limit

2. msgctl function - It 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
- The cmd argument specifies the command to be performed on the queue specified by msqid
  IPC_RMID - remove the message queue msqid and destroy the corresponding msqid_ds
  IPC_SET - set members of the msqid_ds data structure from buf
  IPC_STAT - copy members of the msqid_ds data structure into buf.

3. msgsnd - 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
- where the ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data.

```
struct mymesg
{
    long mtype; /* positive message type */
    char mtext[512]; /* message data, of
                        length nbytes */
};
```

4. msgrcv - 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);
```

- Returns: size of data portion of message if OK, -1 on error.
- The type argument let us specify which message we want
  type==0 - The first message on the queue is returned
  type>0 - The first message on the queue whose message type equals type is returned
  type<0 - The first message on the quee whose message type is lowest value less than or equal to the abosolute value of type is returned

Semaphores

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- The kernel maintains a semid_ds structure for each semaphore set:

```
~~set s~~ struct semid_ds {
struct ipc_perm sem_perm;
unsigned short sem_nsems; /* No. of semaphores in set */
time_t sem_otime; /* last-semop() time */
time_t sem_ctime; /* last-change time */
:
};
```

- Each semaphore is represented by an anonymous structure containing at least the following members

```
struct {
    unsigned short semval; /* Semaphore value, >=0 */
    pid_t sempid; /* pid for last operation */
    unsigned short semncnt; /* No. of processes
                               awaiting semval>curval */
    unsigned short semzcnt; /* No. of processes
                               awaiting semval==0 */
```

;
};

1. semget - This function is used 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
- when a new set is created, the following members of the semid_ds structure are initialized

a) The ipc_perm structure is initialized
b) sem_otime is set to 0
c) sem_ctime is set to the current time
d) sem_nsems is set to nsems

2. Semctl - The semctl function is the catchall for various semaphore operations

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

- Returns: semaphore ID if OK, -1 on error.
- cmd parameter takes values

GETALL - return values of the semaphore set in arg.array
GETVAL - return value of a specific semaphore element
GETPID - return process ID of last process to manipulate element
GETNCNT - return number of processes waiting for element to increment
GETZCNT - return number of processes waiting for element to become 0
SETALL - set values of semaphore set from arg.array
SETVAL - set value of a specific semaphore element to arg.val

3. Semop - 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);
```

- Returns: 0 if OK, 1 on error
- ```
  struct sembuf {
  unsigned short sem_num; /* member number in set */
  short sem_op; /* operation (negative, 0, or positive) */
  short sem_flag; /* IPC_NOWAIT, SEM_UNDO */
  ```
- The nops argument specifies the number of operations.

---

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

Ans Signals:

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

SIGABRT - abnormal termination (abort) - terminate+core
SIGALRM - timer expired - terminate
SIGBUS - hardware fault - terminate+core
SIGCANCEL - threads library internal use - ignore
SIGCHLD - change in status of child - ignore
SIGINT - terminal interrupt character - terminate
SIGKILL - termination - terminate
SIGSTOP - stop - stop process
SIGTERM - termination - terminate
  ```

- when a signal is sent to a process it is pending on the process to handle it. The process can react to pending signals in one of three ways.
  i. Accept the default action of the signal
  ii. Ignore the signal
  iii. Invoke a user-defined function.

- prototype:

```
#include<signal.h>
void (*signal(int sig_no, void (*handler)(int)))(int);
```

where,
sig_no is a signal identifier like SIGINT or SIGTERM
The handler argument is the function pointer of a user defined signal handler function.

- program:

```
#include<iostream.h>
#include<signal.h>
void catch_sig(int sig_num)
{
    signal(sig_num, catch_sig);
    cout<<" catch_sig : "<<sig_num<<endl;
}

int main()
{
    signal(SIGTERM, catch_sig);
    signal(SIGINT, SIG_IGN);
    signal(SIGALRM, SIG_DFL);
    pause();
}
```

4. Explain the kill() and alarom API

Ans

Kill

- A process can send a signal to a related proces via the kill API. This is a simple means of inte process communication or control.
- prototype:

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

- Returns: 0 on success, -1 on failure.
- where, the signal_num is the integer value of a signal to be sent to one or more processes designated by pid
- the possible values for pid are

pid > 0 - The signal is sent to the process where proces ID is pid

pid == 0 - The signal is sent to all processes whose process group id equals the process groupid of the sender

pid < 0 - The signal is sent to all processes whose proces group id equals the absolute value of pid

pid == 1 - The signal is sent to all processes on the system ~~for which~~

~~alaram~~

Example:

```
if (kill(pid, sig) == -1)
        perror("kill");
else
        cerr << "invalid pid:" << argv[0] << endl
return 0;
```

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.
- prototype:

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

- Returns: 0 or number of seconds until previously set alarm.

Example:

```
if (sigaction(SIGALARM, &action, 0) == -1)
{
    perror("sigaction");
    return -1;
}
(void) alarm (timer);
(void pause();
return 0;
```

---

5. Explain the sigsetjmp and siglongjmp functions with an example?

Ans prototype:

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

- The setjmp and longjmp are created to support signal mask processing. specifically, it is implementation dependent on whether a process signal mask is saved and restored when it ~~inokes~~ invokes the setjmp and longjmp APIs respectively.

Example:

```
#include <stdio.h>
#include <setjmp.h>

sigjmp_buf mark;
void p(void);
void recover(void);

int main(void)
{
  if (sigsetjmp(mark,1)!=0){
     printf("siglongjmp() has been called\n");
     recover();
     exit(1);
  }
  printf("sigsetjmp() has been called\n");
  :
  p();
  :
}

void p(void)
 {
    int error=0;
  :
  error=9;
  :
  if (error!=0)
     siglongjmp(mark,-1);
  :
}

void recover(void)
 { ...
 }
```

6. What is a FIFO? With a neat diagram explain client server communication using FIFO?

Ans FIFOs

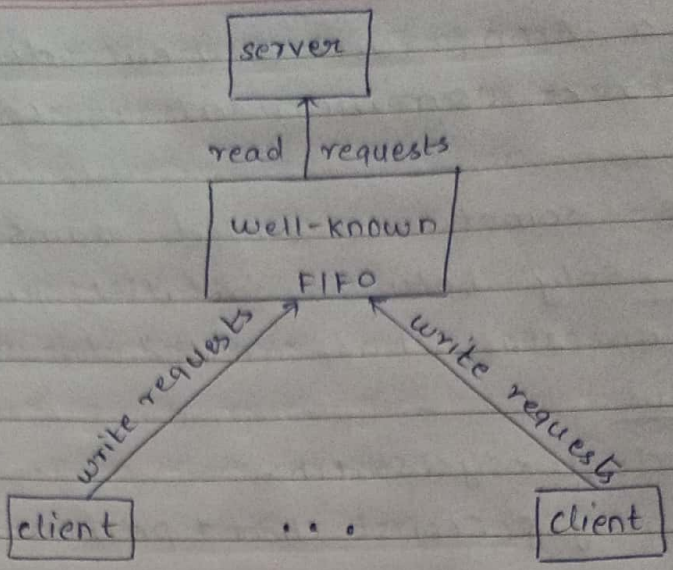
- FIFOs are sometimes called named pipes. Pipes can be used only between related processes when a common ancestor has created the pipe.
- prototype:

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

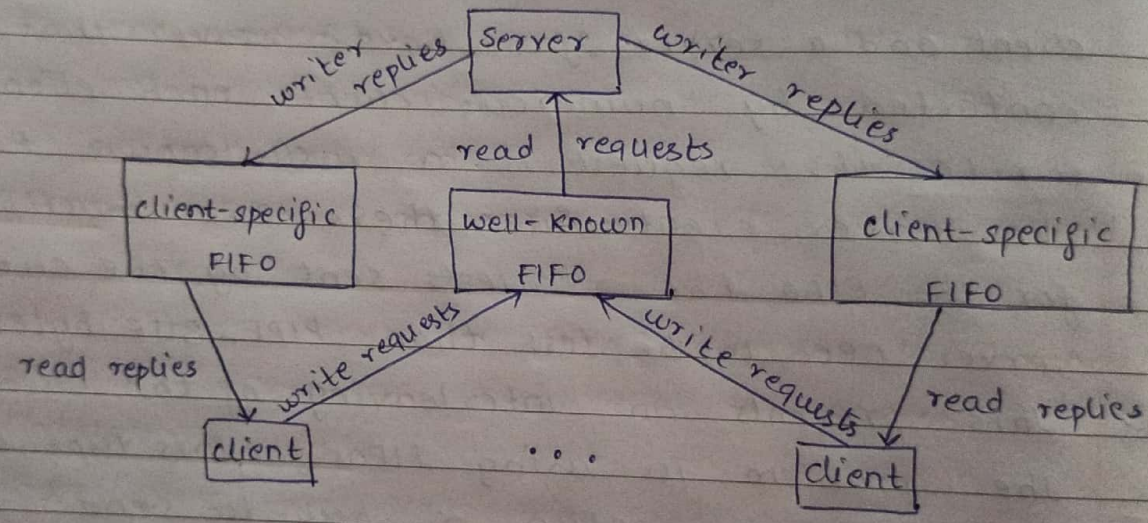
- Returns: 0 if OK, 1 on error.

Client-server Communication using a FIFO

- FIFOs 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 an 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's process ID.
- 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 and no reader.



clients sending requests to a server using a FIFO



client-server communication using FIFO's

7. Explain setuid and setgid functions with example and explain various ways to change user ids.

Ans when our programs need additional privileges or need to gain access to resources that they currently aren't allowed to access, they need to change their user or group ID to an ID that has the appropriate privilege or access.

- prototype:

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

- Both return: 0 if OK, 1 on error
- These are rules for who can change the IDs
  a) If the process has superuser privileges, the setuid function sets the real user ID, effective user ID, and saved set-user-ID to uid
  b) If the 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 and the saved set-user-ID are not changed.
  c) If neither of these two conditions is true, errno is set to EPERM and 1 is returned.
- The three user IDs maintained by kernel
  a) only a superuser process can change the real user ID
  b) The effective user ID is set by the exec functions
  c) The saved set-user-ID is copied from the effective user ID by exec
- various ways these three user IDs can be changed

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID 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 user ID	copied from effective user ID	set to uid	unchanged

Setreuid and setregid functions

- swapping of the real user ID and the effective user ID with the setreuid function
- prototype:

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

- Return: 0 if OK, -1 on error

Seteuid and setegid functions

- These functions only change effective user ID or effective group ID
- prototype:

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

- Return: 0 if OK, 1 on error.

---

8. what are interpreter files? Give the difference between interpreter files and Interpreter!

Ans Interpreter files

- These files are text files that begin with a line of the form

```
#! pathname [optional-argument]
```

- The space between the exclamation point and the pathname is optional. The most common of these interpreter files begin with the line

```
#! /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 files a text file that begins with #! and the interpreter which is specified by the pathname on the first line of the interpreter file.

Example:

```
#include "apue.h"
#include <sys/wait.h>
Int main (void)
{
    pid_t pid;
    if ((pid = fork()) < 0)
    {
        err_sys (" fork error");
    }
    else if (pid == 0)
    {
      if execl ("/home/sar/bin/testinterp",
        "testinterp", 'myarg1," "MY ARG2", (char *)0) < 0)
        err_sys ("execl error");
    }
   if (waitpid (pid, NULL, 0) < 0)
        err_sys ("waitpid error");
    exit(0);
}
```

Output:

```
$ cat /home/sar/bin/testinterp
#! /home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
```

argv[2]: /home/saral/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2

9. what are Daemon process? Enlist their characteristi

Ans Daemon process

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

Characteristics

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

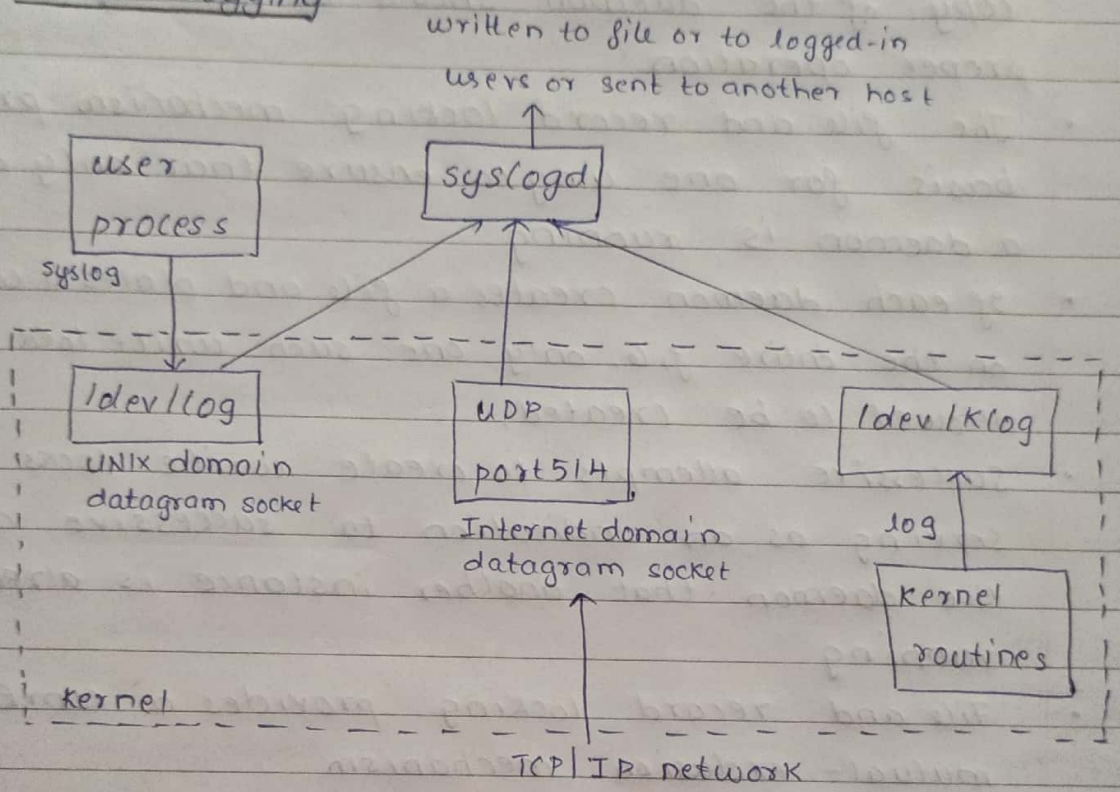
Coding Rules

- Call umask to set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions.
- Call fork and have the parent exit. This does several things. First, if the daemon was started as a simple ~~child~~ shell command, having the parent terminate makes the shell think that the command is done.
- Call setsid to create a new session. The process (a) becomes a session leader of a new session (b) becomes the process group leader of a new process group and (c) has no controlling terminal
- Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system
- Unneeded file descriptors should be closed.
- Some daemons open file descriptors, 0, 1 and 2 to /dev/null so that any library routines that try to read

from standard input or write to standard output or standard error will have no effect.

10. What is error logging explain? Also Explain single instance Daemon?

Ans Error logging



There are three ways to generate log messages

- kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device
- Most user processes call the syslog(3) function to generate log messages. This cause 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
- prototype:

```
#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 and record locking mechanism provides the basis for one way to ensure that only one copy of a daemon is running
- If each daemon creates a file and 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 and 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.