UNIX Programming (18CS56)

Module 4

(I) Changing User IDs and Group IDs

- User ID and Group ID are unique integers used to identify users and groups in UNIX.
- These IDs can be dynamically changed as per privilege of the task.
- Three types of IDs are defined for a process:
  - Real ID – owner of the process. Defines which files the owner has access to.
  - Effective ID – normally same as real ID, but sometimes changed to enable non-privileged users to access files that are usually accessed only by root.
  - Saved ID – when a process is running with elevated privileges but switches to under-privileged account to do some tasks.
- All descriptions for user IDs apply to group IDs as well.
- Changing system date, access control, read or write permissions of a particular file are all based on user IDs and group IDs.
- If a program needs additional privileges or to gain access to resources, then it has to change its user/group ID to some ID that has appropriate privilege/access.
- If a program need lower privileges or to prevent access to resources, then it has to change its user/group ID to some ID that does not have appropriate privilege/access.
- *Least privilege model* states that programs should use the least privilege necessary to accomplish any given task.
- This mechanism reduces security compromises by malicious users who trick programs to use their privileges in unintended/harmful ways.
- *setuid* is used to set real user ID and effective user ID.
  setgid is used to set real group ID and effective group ID.

```
#include<unistd.h>

int setuid(uid_t uid);
int setgid(uid_t gid);
```

- Both functions return 0 if successful, -1 on error.
- There are three rules to change IDs –
  - If process has superuser (root) privileges, *setuid* sets the real user ID, effective user ID and saved set-user-ID to *uid*.
  - If process does not have superuser (root) privileges, but *uid* equals either real user ID or saved set-user-ID, *setuid* sets only effective user ID to *uid*. Real user ID and saved set-user-ID are not changed.
  - If neither is true, *errno* is set to EPERM, and –1 is returned.

- Assumption is that _POSIX_SAVED_IDS constant is true in current system implementation. Saved IDs are a mandatory feature in POSIX.1 specification.
- *Real user ID* –
  - Only a superuser process can change real user ID.
  - It is set by *login* program when user logs in and never changes.
  - *login* is a superuser process and sets all three user IDs when it calls *setuid*.
- *Effective user ID* –
  - It is set by *exec* functions only if set-user-ID bit is set for the program file.
  - If set-user-ID bit is not set, *exec* functions do not change the current value of effective user ID.
  - *setuid* can be called at any time to set effective user ID to either real user ID or saved set-user-ID.
  - Effective user ID cannot be set to any random value.
- *Saved set-user-ID* –
  - It is copied from effective user ID by *exec*.
  - If the file's set-user-ID bit is set, copy is saved after *exec* stores the effective user ID from file's user ID.
- The following table summarizes how the different user IDs are changed when used with *exec* and *setuid* functions.

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 are used to swap real IDs and effective IDs.

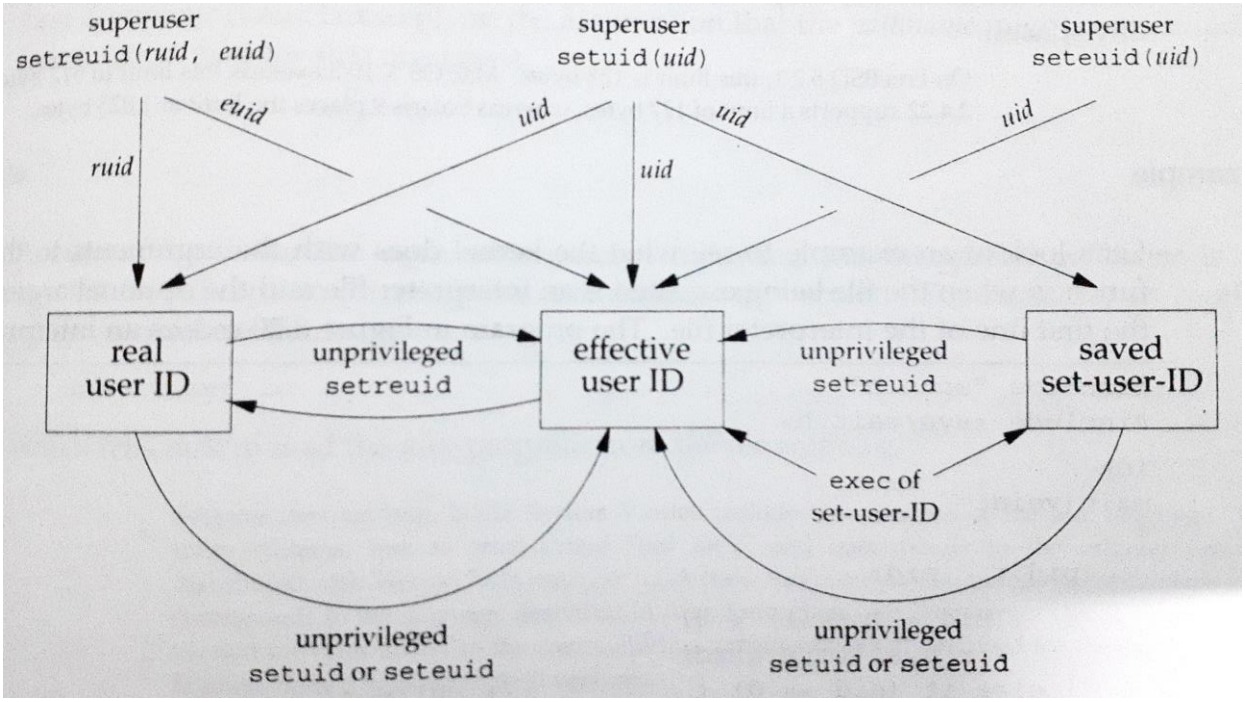
```
#include<unistd.h>

int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

- Both return 0 if successful, -1 on error.
- Value of –1 can be specified to indicate corresponding ID should remain unchanged.
- The main rule is that unprivileged user can always swap between real user ID and effective user ID.
- This allows set-user-ID program to swap to user's normal permissions and swap back again later for set-user-ID operations.

- *seteuid* and *setegid* functions are used to change effective user ID or effective group ID.

```
#include<unistd.h>

int seteuid(uid_t uid);
int setegid(gid_t gid);
```

- Both functions return 0 if successful, -1 on error.
- Unprivileged user can set its effective user ID to either its real user ID or its saved set-user-ID.
- For privileged user, only effective user ID is set to *uid*.
- The following diagram summarizes how all 3 user IDs are modified.



(II) Interpreter Files

- All contemporary UNIX systems support interpreter files.
- These are text files that begin with a line of the form –
  #! *pathname* [*optional-argument*]
- The space between ! and *pathname* is optional.
- Examples of interpreter files - #! /bin/sh, #!/bin/bash
- *pathname* – absolute pathname, as no special operations are performed on it (PATH is not used).
- Recognition of interpreter files is done within the kernel as part of processing the *exec* system call.

- The actual file that gets executed by kernel is not the interpreter file, but the file specified by *pathname* on the first line of interpreter file.
- Interpreter file – text file that begins with #!
  Interpreter – specified by *pathname* on the first line of interpreter file
- First line of interpreter file includes - #!, *pathname*, *optional-argument*, terminating newline and spaces.
- The size limit on first line on various systems are given as:
  FreeBSD - 128 bytes
  Mac OS X – 512 bytes
  Linux – 127 bytes
  Solaris 9 – 1023 bytes
- *optional-argument* is usually the ***–f*** option, which tells *pathname* where to read a particular program file.
- Uses of interpreter files –
  - Used to hide that certain programs are scripts in some other language.
  - Interpreter scripts provide an efficiency gain for the user at some expense in the kernel, as it recognizes these files.
  - Interpreter scripts allow us to write shell scripts using shells other than */bin/sh*.
- The program given below shows how an interpreter file is *exec*-ed.

```
#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){                  //child
                if(execl("/home/sar/bin/test", "test",
                        "arg1", "arg2", (char *)0) < 0)
                        err_sys("execl error");
        }
        if(waitpid(pid, NULL, 0) < 0)   //parent
                err_sys("waitpid error");
        exit(0);
}
```

- The kernel takes the *pathname* from the *execl* call instead of the first argument (*testinterp*), on the assumption that the *pathname* might contain more information than the first argument.

- The *awk* program is also represented as an interpreter file, as follows:

```
#!/bin/awk -f
BEGIN{
        for(i=0; i<ARGC; i++)
                printf "ARGV[%d] = %s\n", i, ARGV[i]
        exit
}
```

- The pathname of the interpreter file (*/usr/local/bin/awkexample*) is passed to the interpreter.
- The filename portion of this pathname is not adequate, because the interpreter (*/bin/awk* in this example) cannot be expected to use the PATH variable to locate files.
- When it reads the interpreter file, *awk* ignores the first line, since the # (pound) sign is *awk*'s comment character.

(III) system Function

- If we are executing a command string from within a program, then the *system* function can be used.
  Example – put a timestamp into a certain file during execution.
- ISO C defines *system* function, but its operation is strongly dependent on system implementation.
- POSIX.1 includes *system* interface, expanding on ISO C definition to describe its behaviour in POSIX environment.
- Prototype of *system* function is given below –

```
#include<stdlib.h>

int system(const char *cmdstring);
```

- If *cmdstring* is null pointer, *system* returns non-zero only if command processor is available.
- This feature determines if *system* function is supported on a given OS or not.
- Under UNIX specification, *system* is always available.
- *system* is implemented by calling *fork*, *exec* and *waitpid*.
- Thus, there are 3 types of return values –
  - If either *fork* fails or *waitpid* returns an error other than EINTR, it returns –1 with *errno* set to indicate the error.
  - If *exec* fails (shell cannot be executed), return value is as if shell had executed *exit(127)*.

- If all 3 functions succeed, return value from *system* is termination status of the shell in the format specified for *waitpid*.

- The implementation of the *system* function without signal handling is shown below:

```
#include<sys/wait.h>
#include<errno.h>
#include<unistd.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){
                execl("/bin/sh","sh","-c",cmdstring,(char *)0);
                _exit(127);
        }
        else{
                while(waitpid(pid, &status, 0) < 0){
                        if(errno!=EINTR){
                                status = -1;
                                break;
                        }
                }
        }
        return(status);
}
```

- The shell's ***-c*** option tells it to take the next command-line argument *cmdstring*, in this case, as its command input instead of reading from standard input or from a given file.
- The shell parses this null-terminated C string and breaks it up into separate command-line arguments for the command.
- The actual command string that is passed to the shell can contain any valid shell commands.
  For example, input and output redirection using < and > can be used.

- The above code is upgraded using the *system* function is shown below:

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

int main(void){
        int status;
        if((status = system("date")) < 0)
                err_sys("system() error");
        pr_exit(status);
        if((status = system("nosuchcommand")) < 0)
                err_sys("system error");
        pr_exit(status);
        if((status = system("who; exit 44")) < 0)
                err_sys("system error");
        pr_exit status;
        exit(0);
}
```

- The advantage of using *system* instead of using *fork* and *exec* directly is that *system* does all the required error handling and signal handling.
- If a process is running with special permissions, either set-user-ID or set-group-ID, and wants to *spawn* another process, it should use *fork* and *exec* directly.
- It has to be changed back to normal permissions after the *fork*, before calling *exec*.
- The *system* function should never be used from set-user-ID or set-group-ID program.

(IV) Process Accounting

- Most UNIX systems provide an option to do process accounting.
- When enabled, kernel writes an accounting record each time a process terminates.
- Typically small amount of binary data - name of command, amount of CPU time used, user ID and group ID, starting time is written as accounting records.
- *acct* function enables and disables process accounting.
- Use of *acct* is from ***accton*** command - superuser executes ***accton*** with a pathname argument to enable accounting.
- Accounting records are written to specified file - */var/account/pacct* on Linux.
- Accounting turned off by executing ***accton*** without any arguments.

- The structure of accounting records defined in *<sys/acct.h>*

```
struct acct{
    char ac_flag; //flag
    char ac_stat; //termination status
    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 r & w
    comp_t ac_rw; //blocks read or written
    char ac_comm[8]; //command name
};
```

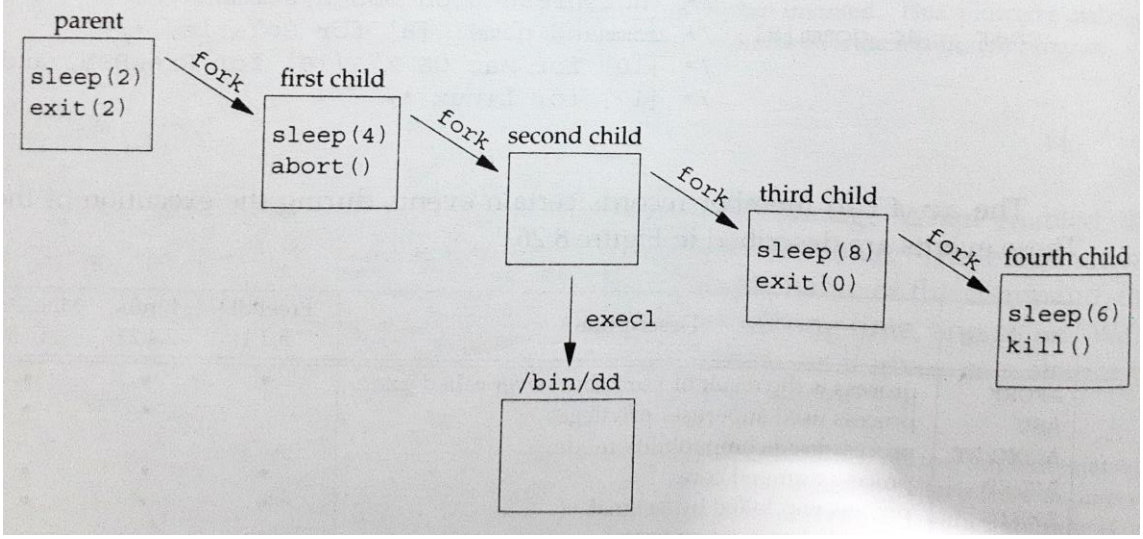
- *ac_flag* member records certain events during the execution of the process. These events are described in the table below.

***ac_flag* value**	**Description**
AFORK	Process is the result of *fork*, but never called *exec*
ASU	Process used superuser privileges
ACOMPAT	Process used compatibility mode
ACORE	Process dumped core
AXSIG	Process was killed by a signal
AEXPND	Expanded accounting entry

- The data required for accounting record is kept by kernel in the process table and initialized whenever a new process is created.
- Each accounting record is written when the process terminates.
- The order of records in the accounting file corresponds to termination order of the processes, not the order in which they are started.
- To know the starting order, one has to go through the accounting file and sort by starting calendar time.
- Elapsed time is more accurate than starting time, but ending time is not known.
- Thus, accurate starting order cannot be reconstructed from data in accounting file.
- Accounting records correspond to processes, not programs.
- New record is initialized by the kernel for the child after a *fork*, not when a new program is executed.

- Although *exec* does not create new accounting record, command name changes and AFORK flag is cleared.
- Example – if there is a chain of 3 programs – A *exec* B, then B *exec* C, and C *exit* - only single accounting record is written.
- Command name in the accounting record corresponds to program C, but CPU times are sum for programs A, B and C.

- The following diagram gives a process structure to obtain accounting data.
  The program calls *fork* four times.
  Each child does something different and then terminates.



- The test procedure is as follows:
  - Become superuser and enable accounting with ***accton*** command.
  - Exit superuser shell and run a program to append 6 records to accounting file: one for superuser shell, one for test parent, one for each of the four test children.
  - Become superuser and turn accounting off.
  - Run a program to print the selected fields from the accounting file.
- The program to generate accounting data is given below.
  It appends 6 records to accounting file: one for superuser shell, one for test parent, one for each of the four test children.

```c
#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
        }
        if((pid = fork()) < 0) //first child
                err_sys("fork error");
        else if(pid != 0){
                sleep(4);
                abort();  //terminate with core dump
        }
        if((pid = fork()) < 0) //second child
                err_sys("fork error");
        else if(pid != 0){
                execl("/bin/dd","dd","if=/etc/termcap",
                        "of=/dev/null",NULL);
                exit(7);
        }
        if((pid = fork()) < 0) //third child
                err_sys("fork error");
        else if(pid != 0){
                sleep(8);
                exit(0);  //normal exit
        }
        sleep(6);  //fourth child
        kill(getpid(), SIGKILL); //terminate with signal
        exit(6);
}
```

- The program to print the selected fields from the accounting file is given below:

```c
#include "apue.h"
#include<sys/acct.h>

#ifdef HAS_SA_STAT
#define FMT "%-*.*s e=%6ld,chars=%7ld,stat=%3u:%c%c%c%c\n"
#else
#define FMT "%-*.*s e=%6ld,chars=%7ld,%c%c%c%c\n"
#endif
#ifndef HAS_ACORE
#define ACORE 0
#endif
#ifndef HAS_AXSIG
#define AXSIG 0
#endif
static unsigned long compt2ulong(comp_t comptime){
        unsigned long val;
        int exp;
        val = comptime & 0x1fff; //13 bit fraction
        exp = (comptime >> 13) & 7; //3 bit exponent
        while(exp-- > 0)
                val *= 8;
        return(val);
}
int main(int argc, char *argv[]){
        struct acct acdata;
        FILE *fp;
        if(argc!=2)
                err_quit("usage: pracct filename");
        if((fp = fopen(argv[1], "r")) == NULL)
                err_sys("can't open %s, argv[1]");
        while(fread(&acdata,sizeof(acdata),1,fp) == 1){
                printf( FMT, (int)sizeof(acdata.ac_comm),
                        (int)sizeof(acdata.ac_comm), acdata.ac_comm,
                        compt2ulong(acdata.ac_etime),
                        compt2ulong(acdata.ac_io),
                        #ifdef HAS_SA_STAT
                                (unsigned char) acdata.ac_stat,
                        #endif
                        acdata.ac_flag & ACORE ? 'D':'',
                        acdata.ac_flag & AXSIG ? 'X':'',
                        acdata.ac_flag & AFORK ? 'F':'',
                        acdata.ac_flag & ASU ? 'S':''
                );
        }
        if(ferror(fp))
                err_sys("read error");
        exit(0);
}
```

(V) User Identification

- Any process can find out its real and effective UID and GID.
- Finding out login name of the user who is running the program can be done using the function *getpwuid(getuid())*.
- If single user has multiple login names, each with same user ID (different login shells for each entry), the UNIX system keeps track and *getlogin* can fetch the login name.

```
#include<unistd.h>

char *getlogin(void);
```

- The above function returns a pointer to string giving login name if OK, otherwise NULL on error.
- *getlogin* can fail if the process is not attached to a terminal that a user logged in to.
- These types of processes are called *daemons*.
- Given login name, we can use the information to look up the user in the password file using *getpwnam*.
  Example – to determine the login shell.
- **LOGNAME** environment variable is initialized with user's login name by *login* and inherited by the login shell.
  But it can be modified by user, so *getlogin* should be used instead.

(VI) Process Times

- Three times can be measured for a process – wall clock time, user CPU time, system CPU time.
- Any process can call the *times* function to obtain these values for itself and its terminated child processes.

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

clock_t times(struct tms *buf);
```

- The above function returns elapsed wall clock time (in clock ticks) if OK, -1 on error.
- The *tms* structure pointed to by *buf* is:

```
struct tms{
    clock_t tms_utime; //user CPU time
    clock_t tms_stime; //system CPU time
    clock_t tms_cutime; //total user CPU time
    clock_t tms_cstime; //total system CPU time
};
```

- The above structure does not contain any measurement for wall clock time.

- The *times* function returns wall clock time as the value of the function, each time it is called.
- Value is measured from some arbitrary point in the past because absolute value cannot be used; relative value is used instead.
- *Example*: call *times* and save the return value.
  At a later time, call *times* again and subtract the earlier return value from the new value.
  Difference between the two times will give the wall clock time.
- Two structure fields for child processes (*tms_cutime*, *tms_cstime*) contain values only for child processes that were waited for using *wait, waitid,* or *waitpid*.
- All *clock_t* values returned by *times* function are converted to seconds using the number of clock ticks per second – the _SC_CLK_TCK value returned by *sysconf*.
- The following program executes each command-line argument (CLA) as a shell command string, timing the command and printing the values from the *tms* structure.

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

static void pr_times(clock_t,struct tms *,struct tms *);
static void do_cmd(char *);

int main(int argc, char *argv[]){
        int i;
        setbuf(stdout, NULL);
        for(i=1; i<argc; i++)
                do_cmd(argv[i]); //once for each CLA
        exit(0);
}
static void do_cmd(char *cmd){ //execute and time the cmd
        struct tms tmsstart, tmssend;
        clock_t start, end;
        int status;
        printf("Command: %s\n", cmd);
        if((start = times(&tmsstart)) == -1) //starting values
                err_sys("times error");
        if((status = system(cmd)) < 0)  //execute command
                err_sys("system() error");
        if((end = times(&tmssend)) == -1)  //ending values
                err_sys("times error");
        pr_times(end-start, &tmsstart, &tmssend);
        pr_exit(status);
}
```

```
static void pr_times(clock_t real, struct tms *tmsstart,
                        struct tms *tmssend){
        static long clktck = 0;
        if(clktck == 0) //fetch clock ticks/second first time
                if((clktck = sysconf(_SC_CLK_TCK)) < 0)
                        err_sys("sysconf error");
        printf("Real: %7.2f\n",real\(double)clktck);
        printf("User: %7.2f\n",
                (tmssend->tms_utime - tmsstart->tms_utime)/(double)clktck);
        printf("Sys: %7.2f\n",
                (tmssend->tms_stime - tmsstart->tms_stime)/(double)clktck);
        printf("Child User: %7.2f\n",
                (tmssend->tms_cutime - tmsstart->tms_cutime)/(double)clktck);
        printf("Child Sys: %7.2f\n",
                (tmssend->tms_cstime - tmsstart->tms_cstime)/(double)clktck);
}
```

(VII) I/O Redirection

- A process can use the C library function *reopen* to change its standard input and standard output ports to refer to text files instead of the console.
- Example to change process standard output to file *foo*:

```
FILE *fptr = freopen("foo", "w", stdout);
printf("Greeting message to foo\n");
```

- Example to change process standard input to file *foo*:

```
char buf[256];
FILE *fptr = freopen("foo", "r", stdin);
while(gets(buf))
        puts(buf);
```

- *freopen* function relies on *open* and *dup2* system calls to do redirection of standard input or standard output.
- To redirect standard input of a process from file *src_stream:*

```
#include<unistd.h>
int fd = open("src_stream", O_RDONLY);
if(fd != -1)
        dup2(fd, STDIN_FILENO), close(fd);
```

- *src_stream* file is now referenced by the STDIN_FILENO descriptor of the process.
- To redirect standard output of a process to file *dest_stream:*

```
#include<unistd.h>
int fd = open("dest_stream", O_WRONLY|O_CREAT|O_TRUNC, 0644);
if(fd != -1)
        dup2(fd, STDOUT_FILENO), close(fd);
```

- *dest_stream* file is now referenced by the STDOUT_FILENO descriptor of the process.
- Implementation of *freopen* function:

```
FILE *freopen(const char* filename, const char *mode, FILE *old_fstream){
        if(strcmp(mode,"r") && strcmp(mode,"w"))
                return NULL; //invalid mode
        int fd = open(file_name, *mode=="r" ? O_RDONLY:
                        O_WRONLY|O_CREAT|O_TRUNC, 0644);
        if(fd == -1)
                return NULL;
        if(!old_stream)
                return fdopen(fd, mode);
        fflush(old_fstream);
        int fd2 = dup2(fd, fileno(old_fstream));
        close(fd);
        return(fd2 == -1)? NULL : old_fstream;
}
```

- In the above function, if the *mode* argument value is not "r" or "w", the function returns a NULL stream pointer, as the function does not support other access modes.
- If the file named by the *file_name* argument cannot be opened with the specified mode, the function will also return a NULL stream pointer.
- If the *open* call succeeds and the *old_fstream* argument is NULL, there is no old stream to redirect.
  The function will just convert *fd* file descriptor to a stream pointer via the *fdopen* function and return it to the caller.
- If the *old_fstream* argument is not NULL, the function will first flush all data stored in that stream's I/O buffer using *fflush* and then it will use *dup2* to force the file descriptor associated with the *old_fstream* to refer to the opened file.
- After *dup2*, the function closes *fd*, as it is no longer needed and returns either the *old_fstream* which now references the new file, or NULL, if the *dup2* call fails.

(VIII) Inter Process Communication (IPC)

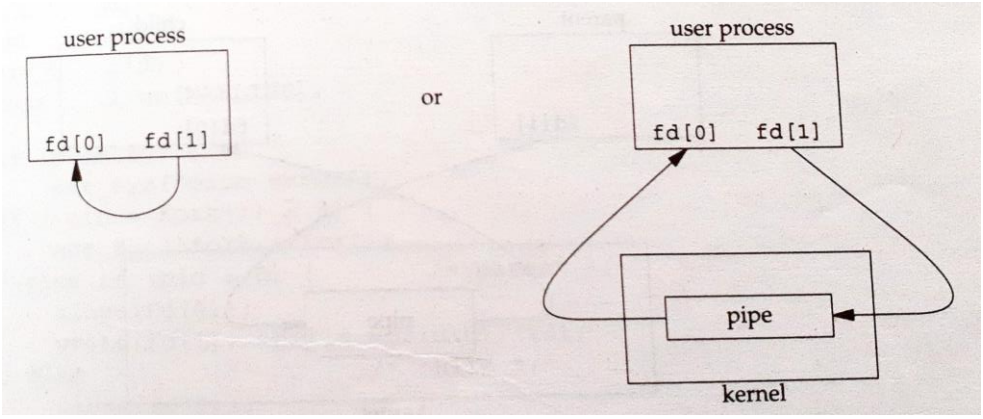
- Inter Process Communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions.
- The communication is a method of co-operation between the processes.
- They can communicate by sharing memory or passing messages.
- There are many types of IPC in UNIX system, namely half-duplex pipes, full-duplex pipes, named full-duplex pipes, FIFOs, message queues, semaphores, shared memory, sockets and STREAMS.

(IX) Pipes

- Pipes are the oldest form of UNIX System IPC and provided by all UNIX systems.
- However, there are a few limitations of pipes:
  - Half duplex – data flows in only one direction.
  - Can be used only between processes that have a common ancestor.
- Pipe is created by a process, that process calls *fork* and then the pipe is used between the parent and the child.
- When there is a sequence of commands in the pipeline for shell to execute, shell creates a separate process for each command and links the standard output of one command to the standard input of the next command using a pipe.
- Pipe is created by calling the *pipe* function:

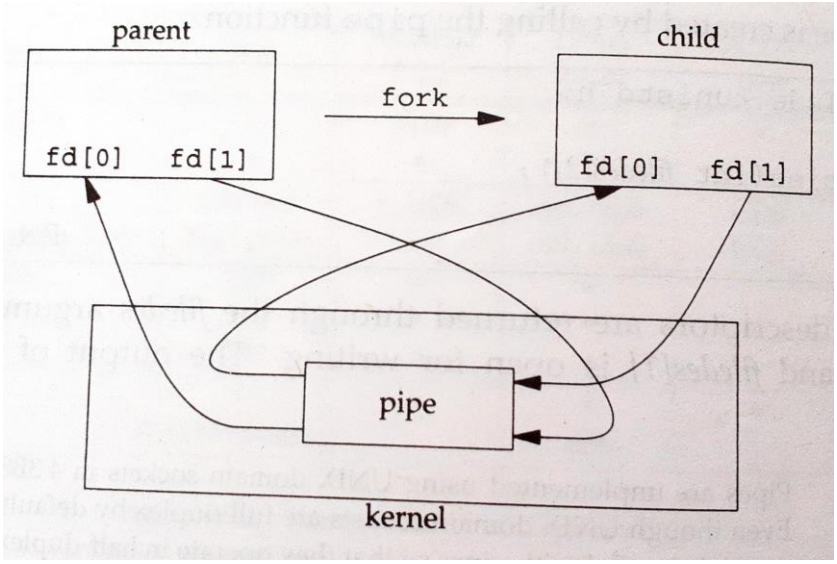
```
#include<unistd.h>

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

- The function returns 0 if OK, -1 on error
- Two file descriptors are returned through *fd:*
  - *fd[0]* - open for reading
  - *fd[1]* - open for writing
- Output of *fd[1]* is the input of *fd[0].*
- There are two ways to visualize a half-duplex pipe: two ends of the pipe connected in a single process, or data in the pipe flows through the kernel.

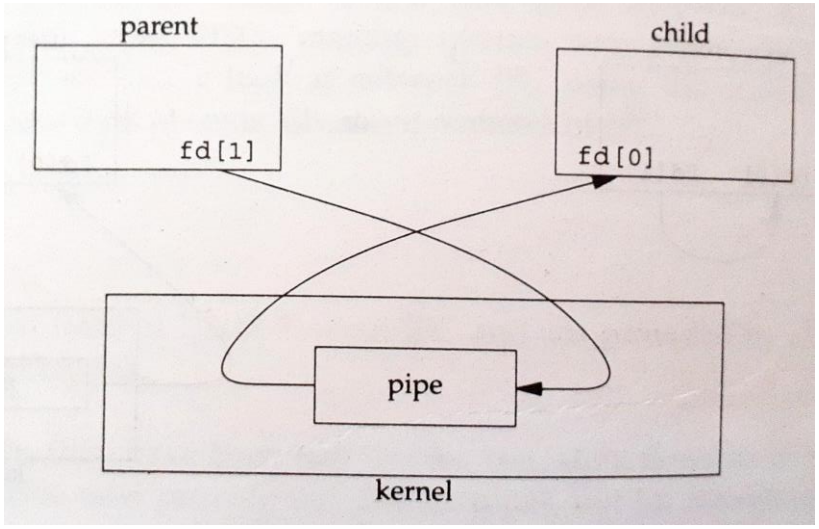


- *fstat* function returns a file type of FIFO for the file descriptor of either end of a pipe.
- Pipe can be tested using the S_ISFIFO macro.
- Pipe in a single process is useless. That is why a process that calls *pipe* then calls *fork*, creating an IPC channel from parent to child process, or vice versa.

This is depicted in the figure below:



- What happens after *fork* depends on the direction of data flow that is required.
- If there is a pipe from parent to child - parent closes read end of the pipe (*fd[0]*) and child closes write end of the pipe (*fd[1]*).
- If there is a pipe from child to parent - parent closes write end of the pipe (*fd[1]*) and child closes read end of the pipe (*fd[0]*).



- When one end of a pipe is closed, two rules apply:
  - If we *read* from a pipe whose write end has been closed, *read* returns 0 to indicate an end of file (EOF) after all the data has been read.
  - If we *write* to a pipe whose read end has been closed, the SIGPIPE signal is generated. If we either ignore the signal or catch it and return from the signal handler, *write* returns -1 with *errno* set to EPIPE.
- When writing to pipe or FIFO, constant PIPE_BUF specifies kernel's pipe buffer size.

- A *write* of PIPE_BUF bytes or less will not be interleaved with *writes* from other processes to the same pipe or FIFO.
- If multiple processes are writing to a pipe or FIFO, and if we *write* more than PIPE_BUF bytes, data might be interleaved with data from other writers.
- Value of PIPE_BUF can be determined using *pathconf* or *fpathconf*.
- The following program shows how a pipe can be created between a parent and its child, and how data can be sent through it.

```
#include "apue.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", 12);
        }
        else{  //child
                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);
        }
        exit(0);
}
```

- In the above program, *read* and *write* are called directly on the pipe descriptors.
- The pipe descriptors are duplicated onto standard input or standard output.
- Often, the child then runs some other program, and that program can either read from its standard input (the pipe that was created) or write to its standard output (the pipe).

- Consider a program that displays some output that it has created, one page at a time.
- To avoid writing all the data to a temporary file and calling *system* to display that file, the output has to be piped directly to the pager.
- To do this, a pipe is created, a child process is *fork*ed, the child's standard input is set up to be the read end of the pipe, and the user's pager program is *exec*-ed.

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

#define DEF_PAGER "/bin/more"

int main(int argc, char *argv[]){
        int n, fd[2];
        pid_t pid;
        char *pager, *argv0, line[MAXLINE];
        FILE *fp;

        if(argc != 2)
                err_quit("Usage: a.out <path>");
        if((fp = fopen(argv[1], "r")) == NULL)
                err_sys("Can't open %s", argv[1]);
        if(pipe(fd) < 0)
                err_sys("pipe error");
        if((pid = fork()) < 0)
                err_sys("fork error");
        else if(pid > 0){       //parent
                close(fd[0]); //close read end
                //parent copies argv[1] to pipe
                while(fgets(line,MAXLINE,fp)!=NULL){
                        n = strlen(line);
                        if(write(fd[1],line,n)!=n)
                                err_sys("write error");
                }
                if(ferror(fp))
                        err_sys("fgets error");
                close(fd[1]); //close write end for reader
                if(waitpid(pid, NULL, 0) < 0)
                        err_sys("waitpid error");
                exit(0);
        }
```

```
        else{                   //child
                close(fd[1]); //close write end
                if(fd[0]!=STDIN_FILENO){
                        if(dup2(fd[0],STDIN_FILENO)!=STDIN_FILENO)
                                err_sys("dup2 error");
                        close(fd[0]);
                }
                //get argument for execl
                if((pager=getenv("PAGER"))==NULL)
                        pager = DEF_PAGER;
                if((argv0=strrchr(pager,'/'))!=NULL)
                        argv0++;
                else
                        argv0 = pager;
                if(execl(pager,argv0,(char *)0) < 0)
                        err_sys("execl error for %s", pager);
        }
        exit(0);
}
```

- The implementation of TELL_WAIT, TELL_PARENT, TELL_CHILD, WAIT_PARENT and WAIT_CHILD using pipes is shown below.

```
#include "apue.h"

static int pfd1[1],pfd2[2];

void TELL_WAIT(void){
        if(pipe(pfd1)<0 || pipe(pfd2)<0)
                err_sys("pipe error");
}

void TELL_PARENT(pid_t pid){
        if(write(pfd2[1],"c",1)!=1)
                err_sys("write error");
}
```

```
void WAIT_PARENT(void){
        char c;
        if(read(pfd1[0],&c,1)!=1)
                err_sys("read error");
        if(c!='p')
                err_quit("WAIT_PARENT: incorrect data");
}

void TELL_CHILD(pid_t pid){
        if(write(pfd1[1],"p",1)!=1)
                err_sys("write error");
}

void WAIT_CHILD(void){
        char c;
        if(read(pfd2[0],&c,1)!=1)
                err_sys("read error");
        if(c!='c')
                err_quit("WAIT_CHILD: incorrect data");
}
```

- Two pipes are created before the *fork*.
- The parent writes the character "p" across the top pipe when TELL_CHILD is called, and the child writes the character "c" across the bottom pipe when TELL_PARENT is called.
- The corresponding WAIT_PARENT and WAIT_CHILD functions do a blocking read for the single character.

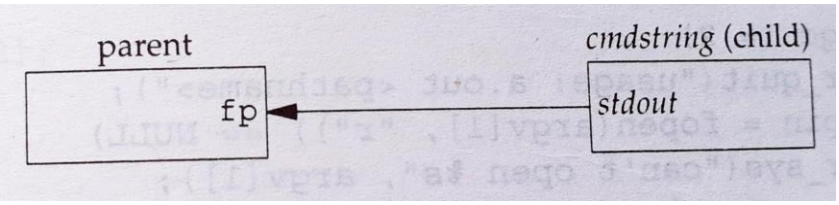
(X) popen and pclose Functions

- Work done by *popen* and *pclose* functions are – creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

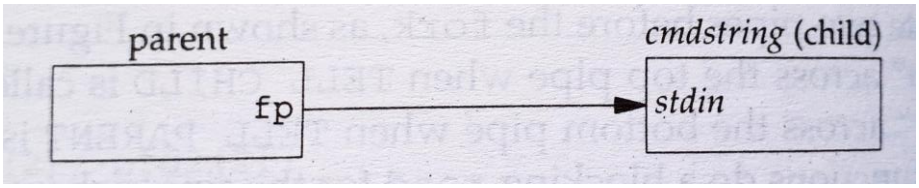
```
#include<stdio.h>

FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *fp);
```

- Return values of *popen* are: file pointer if OK, NULL on error.
- Return values of *pclose* are: termination status of *cmdstring*, -1 on error.
- *popen* does a *fork* and *exec* to execute the *cmdstring*, and returns a standard I/O file pointer.

- If *type* is "r", file pointer is connected to standard output of *cmdstring*.



- If *type* is "w", file pointer is connected to standard input of *cmdstring*.



- *pclose* closes the standard I/O stream, waits for the command to terminate and returns the termination status of the shell.
- If the shell cannot be executed, termination status returned by *pclose* is as if the shell had executed *exit(127)*.

- The implementation of the *popen* and *pclose* functions are shown below.

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

static pid_t *childpid = NULL;
static int maxfd;

FILE *popen(const char cmdstring,const char *type){
        int i, pfd[2];
        pid_t pid;
        FILE *fp;
        if((type[0]!='r' && type[0]!='w')||type[1]!=0){
                errno = EINVAL;
                return(NULL);
        }
```

```
        if(childpid == NULL){
                //allocate zeroed out array for child PIDs
                maxfd = open_max();
                if((childpid=calloc(maxfd,sizeof(pid_t)))==NULL)
                        return(NULL);
        }
        if(pipe(pfd) < 0)
                return(NULL); //errno set by pipe()
        if((pid=fork())<0)
                return(NULL); //errno set by fork()
        else if(pid == 0){      //child
                if(*type=='r'){
                        close(pfd[0]);
                        if(pfd[1]!=STDOUT_FILENO){
                                dup2(pfd[1],STDOUT_FILENO);
                                close(pfd[1]);
                        }
                }
                else{
                        close(pfd[1]);
                        if(pfd[0]!=STDIN_FILENO){
                                dup2(pfd[0],STDIN_FILENO);
                                close(pfd[0]);
                        }
                }
                //close all descriptors in childpid[]
                for(i=0; i<maxfd; i++)
                        if(childpid[i] > 0)
                                close(i);
                execl("/bin/sh","sh","-c",cmdstring,(char *)0);
                _exit(127);
        }
        //parent continues
        if(*type == r){
                close(pfd[1]);
                if((fp=fdopen(pfd[0],type))==NULL)
                        return(NULL);
        }
```

```
        else{
                close(pfd[0]);
                if((fp=fdopen(pfd[1],type))==NULL)
                        return(NULL);
        }
        childpid[fileno(fp)] = pid;
        return(fp);
}
```

```
int pclose(FILE *fp){
        int fd, stat;
        pid_t pid;
        if(childpid == NULL){
                errno = EINVAL;
                return -1;
        }
        fd=fileno(fp);
        if((pid=childpid[fd])==0){
                errno = EINVAL;
                return -1;
        }
        childpid[fd] = 0;
        if(fclose(fp) == EOF)
                return -1;
        while(waitpid(pid,&stat,0) < 0)
                if(errno != EINTR)
                        return -1;
        return stat;
}
```

- Each time *popen* is called, we have to remember the process ID of the child that we create and either its file descriptor or FILE pointer.
- We choose to save the child's process ID in the array *childpid*, which we index by the file descriptor.
- This way, when *pclose* is called with the FILE pointer as its argument, we call the standard I/O function *fileno* to get the file descriptor, and then have the child process ID for the call to *waitpid*.
- Since it is possible for a given process to call *popen* more than once, we dynamically allocate the *childpid* array (the first time *popen* is called), with room for as many children as there are file descriptors.
- Calling *pipe* and *fork* and then duplicating the appropriate descriptors for each process is then done.
- POSIX.1 requires that *popen* close any streams that are still open in the child from previous calls to *popen*.

- To do this, we go through the *childpid* array in the child, closing any descriptors that are still open.
- If the caller of *pclose* has established a signal handler for SIGCHLD, the call to *waitpid* from *pclose* would return an error of EINTR.
- Since the caller is allowed to catch this signal (or any other signal that might interrupt the call to *waitpid*), we simply call *waitpid* again if it is interrupted by a caught signal.
- If the application calls *waitpid* and obtains the exit status of the child created by *popen*, we will call *waitpid* when the application calls *pclose*, find that the child no longer exists, and return 1 with errno set to ECHILD. This is the behavior required by POSIX.1 in this situation.
- *popen* should never be called by a set-user-ID or set-group-ID program.
  If this is done, *popen* will become the equivalent of *execl* executing the shell and command with environment inherited by the calling process.
  Malicious user can manipulate this environment and force the shell to execute commands other than intended.

- *popen* is suited to execute simple filters to transform input/output of the running command. Example – command wants to build its own pipeline.
- Consider an application that writes a prompt to standard output and reads a line from standard input.
- With *popen*, we can interpose a program between the application and its input to transform the input.
- The transformation could be pathname expansion, for example, or providing a history mechanism (remembering previously entered commands).
- A simple filter to demonstrate this operation is shown below.

```
#include "apue.h"
#include<ctype.h>

int main(void){
        int c;
        while((c=getchar())!=EOF){
                if(isupper(c))
                        c=tolower(c);
                if(putchar(c)==EOF)
                        err_sys("output error");
                if(c=='\n')
                        fflush(stdout);
        }
        exit(0);
}
```

- The filter copies standard input to standard output, converting any uppercase character to lowercase.
- We compile this filter into the executable file *uclc*, which we then invoke from the following program using *popen*.

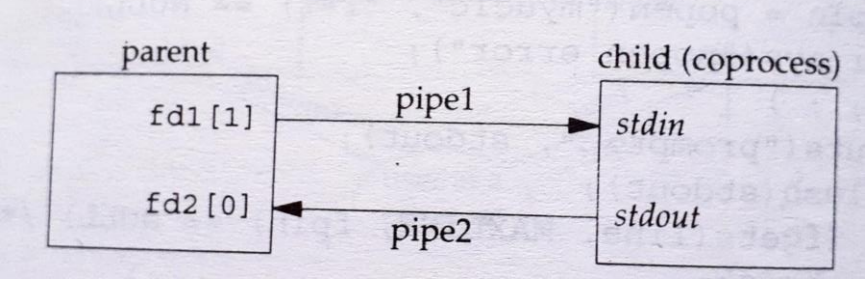
```
#include "apue.h"
#include<sys/wait.h>

int main(void){
        char line[MAXLINE];
        FILE *fpin;
        if((fpin=popen("uclc","r"))==NULL)
                err_sys("popen error");
        for(;;){
                fputs("prompt> ", stdout);
                fflush(stdout);
                //read from pipe
                if(fgets(line,MAXLINE,fpin)==NULL)
                        break;
                if(fputs(line,stdout)==EOF)
                        err_sys("fputs error to pipe");
        }
        if(pclose(fpin) == -1)
                err_sys("pclose error");
        putchar('\n');
        exit(0);
}
```

- We need to call *fflush* after writing the prompt, because the standard output is normally line buffered, and the prompt does not contain a newline

(XI) Coprocesses

- Filter is a program that reads from standard input and writes to standard output.
- It is normally connected linearly in shell pipelines.
- A filter becomes a ***coprocess*** when same program generates filter's input and reads the filter's output.
- Only Korn shell provides coprocesses.
- These normally run in the background from a shell.
- Standard input and standard output are connected to another program using a pipe.
- The main distinction between the *popen* function and coprocess is:
  popen – one-way pipe to standard input or from standard output of another process.
  Coprocess – two one-way pipes to other process: one to its standard input and one from its standard output.

- Coprocess allows a process to write to its standard input, let it operate on data, read from its standard output.



- The following program is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

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

- We compile this program and leave the executable in the file *add2*.

- The next program invokes the *add2* coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

```c
#include "apue.h"

static void sig_pipe(int); //signal handler

int main(void){
        int n, fd1[2], fd2[2];
        pid_t pid;
        char line[MAXLINE];
        if(signal(SIGPIPE,sig_pipe)==SIG_ERR)
                err_sys("signal error");
        if(pipe(fd1)<0 || pipe(fd2)<0)
                err_sys("pipe error");
        if((pid=fork()) < 0)
                err_sys("fork error");
        else if(pid > 0){       //parent
                close(fd1[0]);
                close(fd2[1]);
                while(fgets(line,MAXLINE,stdin)!=NULL){
                        n = strlen(line);
                        if(write(fd1[1],line,n)!=n)
                                err_sys("write error to pipe");
                        if((n=read(fd2[0],line,MAXLINE))<0)
                                err_sys("read error from pipe");
                        if(n == 0){
                                err_msg("child closed pipe");
                                break;
                        }
                        line[n] = 0; //null terminate
                        if(fputs(line,stdout)==EOF)
                                err_sys("fputs error");
                }
                if(ferror(stdin))
                        err_sys("fgets error on stdin");
                exit(0);
        }
        else{           //child
                close(fd1[1]);
                close(fd2[0]);
                if(fd1[0]!=STDIN_FILENO){
                        if(dup2(fd1[0],STDIN_FILENO)!=STDIN_FILENO)
                                err_sys("dup2 error to stdin");
```

```
                        close(fd1[0]);
                }
                if(fd2[1]!=STDOUT_FILENO){
                        if(dup2(fd2[1],STDOUT_FILENO)!=STDOUT_FILENO)
                                err_sys("dup2 error to stdout");
                        close(fd2[1]);
                }
                if(execl("./add2","add2",(char *)0)<0)
                        err_sys("execl error");
        }
        exit(0);
}

static void sig_pipe(int signo){
        printf("SIGPIPE caught\n");
        exit(1);
}
```

- Here, we create two pipes, with the parent and the child closing the ends they do not need.
- We have to use two pipes: one for the standard input of the coprocess and one for its standard output.
- The child then calls *dup2* to move the pipe descriptors onto its standard input and standard output, before calling *execl*.

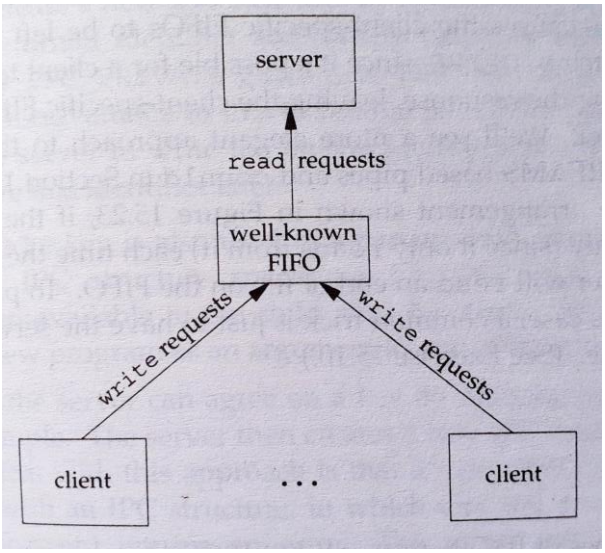
(XII) FIFOs

- FIFOs are sometimes called *named pipes*.
- The main distinction between the two are:
  Pipes – can be used only between related processes when a common ancestor has created the pipe.
  FIFOs – unrelated processes can exchange data.
- Creating a FIFO can be done as follows:

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

int mkfifo(const char *pathname, mode_t mode);
```

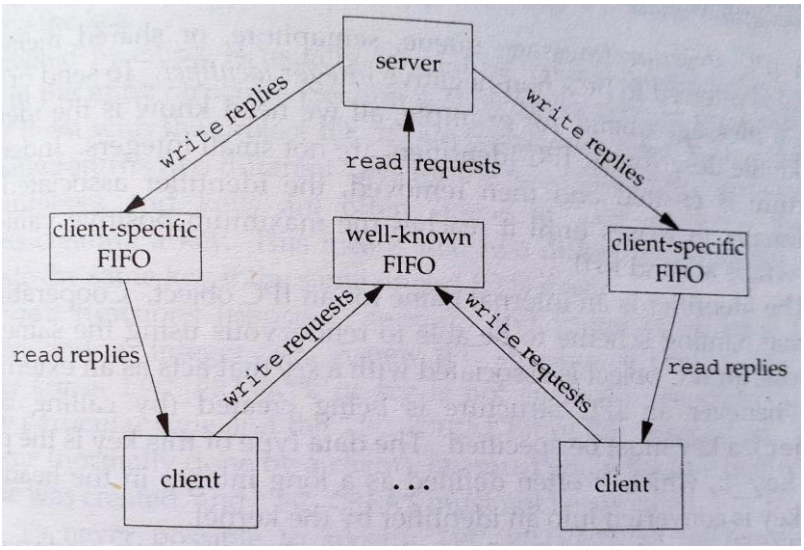
- The function returns 0 if OK, otherwise -1 on error.
- After *mkfifo*, FIFO can be opened using *open*.
  Normal file I/O functions such as *close*, *read*, *write*, *unlink* all work with FIFOs.
- When FIFO is *open*ed, the O_NONBLOCK flag affects the outcome:

- If O_NONBLOCK is not specified – *open* for read-only blocks until some other process opens the FIFO for writing. *open* for write-only blocks until some other process opens the FIFO for reading.
- If O_NONBLOCK is specified – *open* for read-only returns immediately. *open* for write-only returns –1 with *errno* set to ENXIO if no process has the FIFO open for reading.
- If we *write* to a FIFO that no process has open for reading, SIGPIPE is generated.
- When last writer for a FIFO closes it, end of file is generated for the reader.
- *Client-server communication using a FIFO:*
- Server is contacted by numerous clients.
  Each client can write its request to a well-known FIFO that the server creates.
- Pathname of the server is known to all the clients that need to contact the server.



- If there are multiple writers for the FIFO, requests sent by the clients to the server must be less than PIPE_BUF bytes in size.
- This prevents any interleaving between the *write* calls of the clients.
- But the main problem is how to send back the replies from server to each client.
- Single FIFO cannot be used as clients would not know when to read their response versus responses for other clients.

- The solution for this is: each client sends its PID with the request. Server then creates unique FIFO for each client using a pathname based on client's PID.

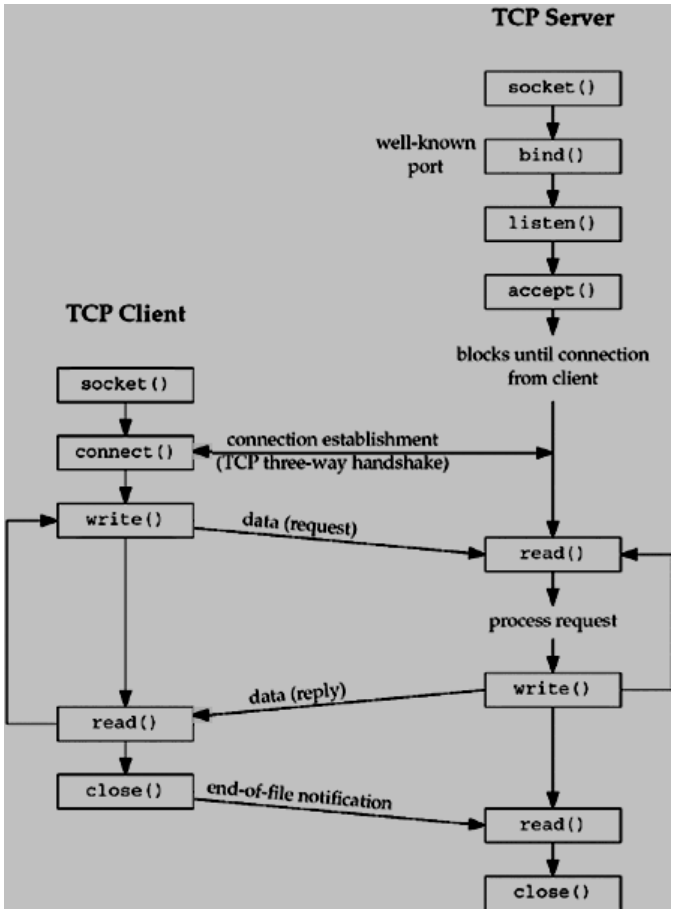


- The drawback of this approach is that it is impossible for the server to tell whether a client crashes.
- This causes client-specific FIFOs to be left in the file system.
- Server must catch SIGPIPE, since it is possible for a client to send a request and terminate before reading the response – this leaves the client-specific FIFO with one writer (server) and no reader.
- If server opens its well-known FIFO read-only each time the number of clients reduces from 1 to 0, server will read an end-of-file on FIFO.
- To prevent this, server must open the well-known FIFO for read-write.

(XIII) Client-Server Connection

- Functions used in client-server communication are:
  - socket()
  - connect()
  - bind()
  - listen()
  - accept()
  - send() / write()
  - recv() / read()
  - close()

- The communication between the client and server is as shown:



- *socket*() is necessary to perform network communication. Mainly specifies protocol type and family used for communication.

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

int socket(int family, int type, int protocol);
```

- *socket()* returns a socket descriptor if successful, -1 on error.
- The parameters passed are: *family* (protocol family), *type* (kind of socket), *protocol* (specific protocol type or 0 for system default)
- The various protocol families are shown below:

Family	**Description**
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	UNIX domain protocols
AF_ROUTE	Routing sockets
AF_KEY	Key socket

- The various types of sockets are shown below:

Type	Description
SOCK_STREAM	Stream socket
SOCK_DGRAM	Datagram socket
SOCK_SEQPACKET	Sequenced packet socket
SOCK_RAW	Raw socket

- The specific protocol types are given below. The system default is specified as 0.

Protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

- *connect*() is used by TCP client to establish connection with a TCP server.

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

int connect(int sockfd, struct sockaddr *serv_addr,
                int addrlen);
```

- *connect()* returns 0 if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *serv_addr* (pointer to socket structure that contains destination IP and port), *addrlen* (size of socket structure)
- *bind*() assigns local protocol address to a socket.

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

int bind(int sockfd, struct sockaddr *my_addr,
                int addrlen);
```

- *bind()* returns 0 if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *my_addr* (pointer to socket structure that contains local IP and port), *addrlen* (size of socket structure)
- *listen*() converts unconnected socket into a passive socket to accept incoming connection requests.

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

int listen(int sockfd, int backlog);
```

- *listen()* returns 0 if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *backlog* (number of allowed connections)

- *accept*() is called by a TCP server to return the next completed connection from the front of the completed connection queue.

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

int accept(int sockfd, struct sockaddr *cli_addr,
                socklen_t *addrlen);
```

- *accept()* returns non-negative value if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *cli_addr* (pointer to socket structure that contains client IP and port), *addrlen* (size of socket structure)
- *send*() / *write()* is used to send data over stream sockets or connected datagram sockets.

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

int send(int sockfd, const void *msg,
                int len, int flags);
```

- *send() / write()* returns the number of bytes sent if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *msg* (pointer to data that must be sent), *len* (length of data to be sent), *flags* (set to 0)
- *recv*() / *read()* is used to receive data over stream sockets or connected datagram sockets.

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

int recv(int sockfd, void *buf,
                int len, unsigned int flags);
```

- *recv() / read()* returns number of bytes read into the buffer if successful, -1 on error.
- The parameters passed are: *sockfd* (socket descriptor), *buf* (pointer to buffer that reads incoming information), *len* (maximum length of buffer), *flags* (set to 0)
- *close()* is used to close the communication between the client and the server.

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

int close(int sockfd);
```

- *close()* returns 0 if successful, -1 on error.
- The parameter passed is: *sockfd* (socket descriptor)

- The program for server is given below.

```c
#include<stdio.h>
#include<stdlib.h>
#include<netdb.h>
#include<netinet/in.h>
#include<string.h>

int main(int argc, char *argv[]){
        int sockfd,newsockfd,portno,clilen;
        char buffer[256];
        struct sockaddr_in servaddr,cliaddr;
        //create socket
        sockfd = socket(AF_INET,SOCK_STREAM,0);
        if(sockfd < 0){
                perror("socket error");
                exit(1);
        }
        //initialize socket structure
        bzero((char *)&servaddr, sizeof(servaddr));
        portno = 5001;
        servaddr.sin_family = AF_INET;
        servaddr.sin_addr.s_addr = INADDR_ANY;
        servaddr.sin_port = htons(portno);
        //bind host address
        if(bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr))<0){
                perror("bind error");
                exit(1);
        }
        //listening for clients
        listen(sockfd, 5);
        clilen = sizeof(cliaddr)
        //accept client connections
        newsockfd=accept(sockfd,(struct sockaddr *)&cliaddr,&clilen);
        if(newsockfd < 0){
                perror("accept error");
                exit(1);
        }
        //communication after connection establishment
        bzero(buffer,256);
        n = read(newsockfd,buffer,255);
        if(n < 0){
                perror("read error");
                exit(1);
        }
        printf("Message: %s", buffer);
        //response to client
        n = write(newsockfd,"Received message",16);
        if(n < 0){
                perror("write error");
                exit(1);
        }
        return 0;
}
```

- The program for client is given below.

```c
#include<stdio.h>
#include<stdlib.h>
#include<netdb.h>
#include<netinet/in.h>
#include<string.h>

int main(int argc, char *argv[]){
        int sockfd, portno, n;
        struct sockaddr_in servaddr;
        struct hostent *server;
        char buffer[256];

        if(argc < 3){
                fprintf(stderr,"Usage: %s <host> <port>",argv[0]);
                exit(0);
        }
        portno = atoi(argv[2]);
        //create socket
        sockfd = socket(AF_INET,SOCK_STREAM,0);
        if(sockfd < 0){
                perror("socket error");
                exit(1);
        server=gethostbyname(argv[1]);
        if(server == NULL){
                fprintf("no such host");
                exit(0);
        }
        //initializations
        bzero((char *)&servaddr, sizeof(servaddr));
        servaddr.sin_family = AF_INET;
        bcopy((char *)server->h_addr,(char *)&servaddr.sin_addr.s_addr,
                        server->h_length);
        servaddr.sin_port = htons(portno);
        //connect to server
        if(connect(sockfd,(struct sockaddr *)&servaddr),sizeof(servaddr))<0){
                perror("connection error");
                exit(1);
        }
        //ask for message from server
        printf("Please send message");
        bzero(buffer, 256);
        fgets(buffer, 255, stdin);
        //send message to server
        n = write(sockfd, buffer, strlen(buffer));
```

```
        if(n < 0){
                perror("write error");
                exit(1);
        }
        //read response from server
        bzero(buffer, 256);
        n = read(sockfd, buffer, 255);
        if(n < 0){
                perror("read error");
                exit(1);
        }
        printf("%s", buffer);
        return 0;
}
```

(XIV) Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
- Message queue is referred to as *queue*, message queue identifier is referred to as *queue ID*
- *msgget* is a function with which new queue is created or existing queue is opened.
- *msgsnd* is a function with which new messages are added to the end of the queue.
- *msgrcv* is a function with which messages are fetched from a queue.
- Contents of message are – type field, length, actual data bytes.
- Permission structure *ipc_perm* is given as –

```
struct ipc_perm{
        uid_t uid; //owner's effective UID
        gid_t gid; //owner's effective GID
        uid_t cuid; //creator's effective UID
        gid_t cgid; //creator's effective GID
        mode_t mode; //access modes
        ...
};
```

- Each queue has *msqid_ds* structure which defines the current status of the queue.

```
struct msqid_ds{
        struct ipc_perm msg_perm; //permission structure
        msgqnum_t msg_qnum; //no. of messages in 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
        ...
};
```

- *msgget* is used to either open an existing message queue or create new queue.

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

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

- It returns message queue ID if OK, -1 on error
- When a queue is created, initialization of few members of *msqid_ds* is done.
  - *ipc_perm* is initialized, *mode* member is set to corresponding permission bits of *flag*
  - *msg_qnum, msg_lspid, msg_lrpid, msg_stime, msg_rtime* all are set to 0.
  - *msg_ctime* is set to current time.
  - *msg_qbytes* is set to system limit.
- *msgctl* performs various control operations on a message queue.

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

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

- It returns 0 if OK, -1 on error
- *cmd* argument specifies command to be performed on the queue specified by *msqid*.
  - IPC_STAT: fetch *msqid_ds* structure for the queue, storing it in structure pointed to by *buf*
  - IPC_SET: copy *msg_perm.uid, msg_perm.gid, msg_perm.mode* and *msg_qbytes* from structure pointed to by *buf* to *msqid_ds* structure associated with the queue.
  - IPC_RMID: remove message queue from the system and any data still on the queue.
- *msgsnd* is used to place data onto a message queue.

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

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

- It returns 0 if OK, -1 on error
- Each message is composed of positive long integer type field, non-negative length and actual data bytes corresponding to the length.
- Messages are always placed at the end of the queue.
- *ptr* points to long integer that contains the positive integer message type and is immediately followed by message data.
- No message data is present if *nbytes* is 0.
- If largest message to be sent is 512 bytes, structure *mymsg* is defined and *ptr* then points to *mymsg* structure.

```
struct mymsg{
        long mtype; //positive message type
        char mtext[512]; //message data
};
```

- Message type can be used to fetch messages in an order other than first in, first out.

- If *msgsnd* returns successfully, *msqid_ds* updated to indicate PID that made the call (*msg_lspid*), time that the call was made (*msg_stime*) and that one more message is on the queue (*msg_qnum*).
- *flag* value of IPC_NOWAIT can be specified for non-blocking of *msgsnd*.
- If message queue is full (total no. of messages on the queue = system limit, or total no. of bytes on the queue = system limit) – it causes *msgsnd* to return immediately with an error of EAGAIN.
- If IPC_NOWAIT is not specified, the operation is blocked until there is space for the message, or the queue is removed from the system (EIDRM), or a signal is caught and signal handler returns (EINTR).
- Ungraceful removal of message queue is said to be done because there is no reference count with each queue (this feature is however, present in open files).
- *msgrcv* – messages are retrieved from a queue.

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

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

- It returns the size of data portion of message if OK, -1 on error.
- *ptr* points to long integer where the message type of returned message is stored followed by a data buffer for actual message data.
- *nbytes* specifies size of the data buffer.
  - If returned message > *nbytes* and MSG_NOERROR bit in *flag* is set, message is truncated.
  - If message is too big and flag is not specified, error E2BIG is returned and message stays on the queue.
- *type* argument lets us specify which message we want.
  - *type* == 0 – first message on queue is returned.
  - *type* > 0 – first message on queue whose message type = *type* is returned.
  - *type* < 0 – first message on queue whose message type is the lowest value lesser than or equal to the absolute value of *type* is returned.
- If *msgrcv* returns successfully, *msqid_ds* updated to indicate PID that made the call (*msg_lrpid*), time that the call was made (*msg_rtime*) and that one less message is on the queue (*msg_qnum*).
- *flag* value of IPC_NOWAIT can be specified for non-blocking of *msgrcv*.
- If message of specified type is not available, it causes *msgrcv* to return immediately with a value of –1 and an error of ENOMSG.
- If IPC_NOWAIT is not specified, the operation is blocked until message of specified type is available, or the queue is removed from the system (EIDRM), or a signal is caught and signal handler returns (EINTR).

(XV) Semaphores

- 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:
  - test the semaphore that controls the resource.
  - if the value of semaphore is positive, use the resource. Process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
  - if value of semaphore is 0, go to sleep until the value is greater than 0. When it wakes up, it returns to first step.
- When a process is done with a shared resource that is controlled by a semaphore, value of semaphore is incremented by 1.
- If any other processes are asleep and waiting for the semaphore, they are awakened.
- For the correct implementation of semaphore, test of semaphore value and decrement of the value must be atomic operations.
  Thus, semaphores are normally implemented inside the kernel.
- *Binary semaphore* is the one which controls a single resource and its value is initialized to 1.
- Semaphore can be initialized to any positive value.
  (value – no. of units of the shared resource that are available for sharing)
- Kernel maintains *semid_ds* structure for each semaphore set:

```
struct semid_ds{
        struct ipc_perm sem_perm; //permission structure
        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:

```
struct{
        unsigned short semval; //semaphore value
        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
        ...
};
```

- *semget* is used obtain a semaphore ID.

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

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

- It returns semaphore ID if OK, -1 on error

- When a semaphore is created, initialization of few members of *semid_ds* is done.
  - *ipc_perm* is initialized, *mode* member is set to corresponding permission bits of *flag*
  - *sem_otime* is set to 0.
  - *sem_ctime* is set to current time.
  - *sem_nsems* is set to *nsems* value.
- Number of semaphores in the set is *nsems*.
- If a new set is being created (typically in the server), *nsems* must be specified.
- If an existing set is being referenced (typically in the client), *nsems* can be 0.
- *semctl* comprises of various semaphore operations.

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

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

- It returns a value based on *cmd* for all GET commands (except GETALL), 0 for remaining commands.
- Value of *semnum* is between 0 and (*nsems*-1)
- Fourth argument is optional depending on the command requested.
  If present, it is of type *semun* – union of various command-specific arguments.

```
union semun{
        int val; //for SETVAL
        struct semid_ds *buf; //for IPC_STAT and IPC_SET
        unsigned short *array; //for GETALL and SETALL
};
```

- *cmd* specifies one of ten commands to be performed on the set specified by *semid*.
  - IPC_STAT: fetch *semid_ds* structure for the set, storing it in structure pointed to by *arg.buf*
  - IPC_SET: set *sem_perm.uid*, *sem_perm.gid*, *sem_perm.mode* fields from structure pointed to by *arg.buf* in the *semid_ds* structure associated with the set.
  - IPC_RMID: immediately remove the semaphore set from the system.
  - GETVAL: return value of *semval* for the member *semnum*.
  - SETVAL: set value of *semval* for the member *semnum*. Value specified by *arg.val*
  - GETALL: fetch all semaphore values in the set stored in the array pointed to by *arg.array*
  - SETALL: set all semaphore values in the set to values pointed to by *arg.array*
  - GETPID: return value of *sempid* for the member *semnum*.
  - GETNCNT: return value of *semncnt* for the member *semnum*.
  - GETZCNT: return value of *semzcnt* for the member *semnum*.

- *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);
```

- It returns 0 if OK, -1 on error.
- *semoparray* argument consists of a pointer to an array of semaphore operations represented by *sembuf* structures.

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

- *nops* argument – specifies number of operations in the array.
- Operation on each member of the set is specified by corresponding *sem_op* value (negative, 0, or positive).

(XVI) Shared Memory

- Shared memory concept allows two or more processes to share a given region of memory.
- It is the fastest form of IPC as data does not need to be copied between client and server.
- It also provides synchronization of access to a given region among multiple processes.
- Semaphores are used to synchronize shared memory access.
- The kernel maintains a structure for each shared memory segment:

```
struct shmid_ds{
        struct ipc_perm shm_perm; //permission structure
        size_t shm_segsz; //size of segment
        pid_t shm_lpid; //PID of last shmop()
        pid_t shm_cpid; //PID of creator
        shmatt_t shm_nattch; //no. of current attaches
        time_t shm_atime; //last attach time
        time_t shm_dtime; //last detach time
        time_t shm_ctime; //last change time
        ...
};
```

- *shmget* – this function is used to obtain a shared memory identifier.

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

int shmget(key_t key, size_t size, int flag);
```

- It returns: shared memory ID if OK, -1 on error
- When a memory segment is created, initialization of members of *shmid_ds* is done.
  - *ipc_perm* is initialized, *mode* member is set to corresponding permission bits of *flag*
  - *shm_lpid*, *shm_nattach*, *shm_atime*, *shm_dtime* are all set to 0.
  - *shm_ctime* is set to the current time.
  - *shm_segsz* is set to the *size* requested.
- *shmctl* – comprises of various shared memory operations.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- It returns: 0 if OK, -1 on error
- *cmd* argument specifies commands to be performed on the memory segment specified by *shmid*.
  - IPC_STAT: fetch *shmid_ds* structure for the segment, storing it in structure pointed to by *buf*
  - IPC_SET: set *shm_perm.uid*, *shm_perm.gid*, *shm_perm.mode* from structure pointed to by *buf* to *shmid_ds* structure associated with the shared memory segment.
  - IPC_RMID: remove shared memory segment set from the system.
- *shmat* – process attaches to the address space of a memory segment after its creation.

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

void *shmat(int shmid, const void *addr, int flag);
```

- It returns: pointer to shared memory segment if OK, -1 on error
- Address in calling process at which segment is attached depends on *addr* argument and SHM_RND bit specified in the *flag* –
  - if *addr* is 0, segment is attached at the first available address selected by kernel.
  - if *addr* is nonzero and SHM_RND is not specified, segment is attached at address given by *addr*.
  - if *addr* is nonzero and SHM_RND is specified, segment is attached at address given by (*addr* - (*addr* mod SHMLBA))
- *shmdt* – detach a shared memory segment.

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

int shmdt(void *addr);
```

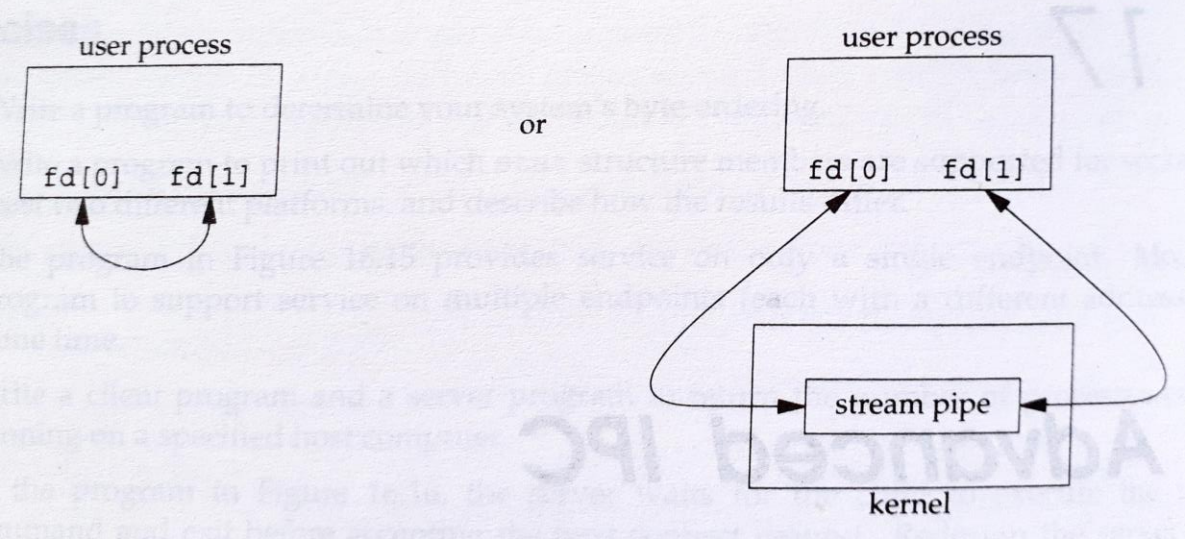
- It returns: 0 if OK, -1 on error

- It does not remove the identifier and its associated data structure from the system.
- *addr* argument – value that was returned by previous call to *shmat*.
- If successful, *shmdt* will decrement the *shm_nattch* counter in the associated *shmid_ds* structure.
- Where a kernel places shared memory segments that are attached with an address of 0 is highly system dependent.
- The following program prints some information on where one particular system places various types of data.

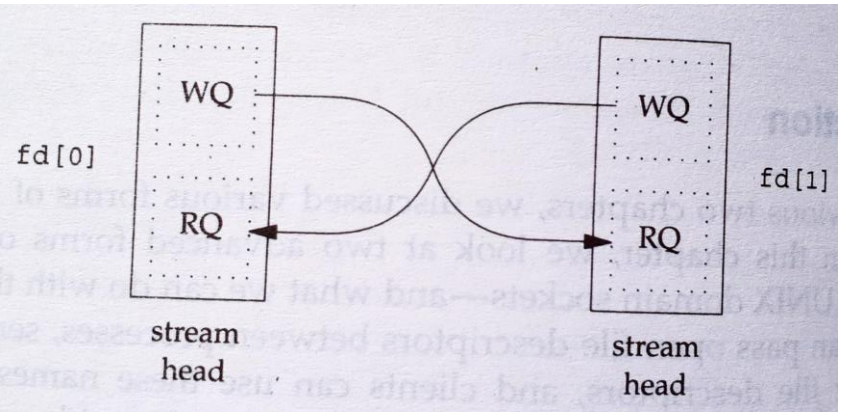
```
#include "apue.h"
#include<sys/shm.h>

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 //user read/write

char array[ARRAY_SIZE]; //bss

int main(void){
        int shmid;
        char *ptr, *shmptr;

        printf("array[] from %lx to %lx",(unsigned long)&array[0],
                (unsigned long)&array[ARRAY_SIZE]);
        printf("stack around %lx",(unsigned long)&shmid);

        if((ptr=malloc(MALLOC_SIZE))==NULL)
                err_sys("malloc error");
        printf("malloced from %lx to %lx",(unsigned long)ptr,
                (unsigned long)ptr+MALLOC_SIZE);

        if((shmid=shmget(IPC_PRIVATE,SHM_SIZE,SHM_MODE))<0)
                err_sys("shmget error");
        if((shmptr=shmat(shmid,0,0))==(void *)-1)
                err_sys("shmat error");
        printf("shared memory attached from %lx to %lx",(unsigned long)shmptr,
                (unsigned long)shmptr+SHM_SIZE);

        if(shmctl(shmid,IPC_RMID,0)<0)
                err_sys("shmctl error");
        exit(0);
}
```

(XVII) STREAMS Pipes

- A STREAMS pipe is a bidirectional (full-duplex) pipe that can be used for IPC between parent and child.
- There are two ways to visualize a STREAMS pipe –

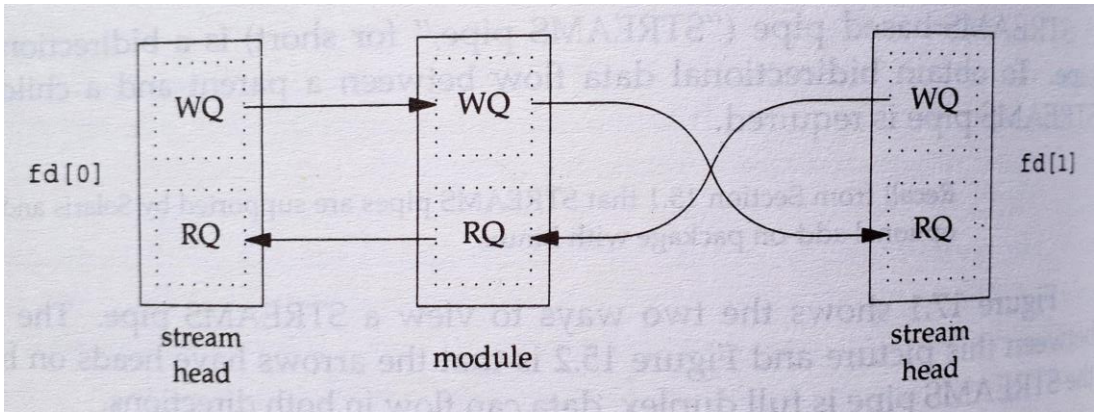


- There are two stream heads in a STREAMS pipe – each write queue (WQ) pointing at other's read queue (RQ).
- The data written to one end of pipe is placed in messages on other's read queue.
- The inside part of a STREAMS pipe is depicted below –



- A STREAMS module can be pushed onto either end of the pipe to process data written to the pipe.
- However, the module must be removed from same end on which it has pushed.

- The inside part of a STREAMS pipe with a module is depicted below –



- STREAMS mechanism provides a way for processes to give a pipe a name in the file system – bypasses problem of dealing with unidirectional FIFOs.
- The coprocess *add2* is redefined again, with the help of a single STREAMS pipe as shown below.

```
#include "apue.h"

static void sig_pipe(int);

int main (void){
        int n, fd[2];
        pid_t pid;
        char line[MAXLINE];

        if(signal(SIGPIPE,sig_pipe)==SIG_ERR)
                err_sys("signal error");
        if(s_pipe(fd)<0)
                err_sys("pipe error");
        if((pid=fork())<0)
                err_sys("fork error");
        else if(pid > 0){       //parent
                close(fd[1]);
                while(fgets(line,MAXLINE,stdin)!=NULL){
                        n = strlen(line);
                        if(write(fd[0],line,n)!=n)
                                err_sys("write error to pipe");
```

```
                        if((n=read(fd[0],line,MAXLINE)) < 0)
                                err_sys("read error from pipe");
                        if(n == 0){
                                err_msg("child closed pipe");
                                break;
                        }
                        line[n] = 0; //null terminate
                        if(fputs(line,stdout)==EOF)
                                err_sys("fputs error");
                }
                if(ferror(stdin))
                        err_sys("fgets error on stdin");
                exit(0);
        }
        else{           //child
                close(fd[0]);
                if(fd[1]!=STDIN_FILENO && dup2(fd[1],STDIN_FILENO)
                                        !=STDIN_FILENO)
                        err_sys("dup2 error to stdin");
                if(fd[1]!=STDOUT_FILENO && dup2(fd[1],STDOUT_FILENO)
                                        !=STDOUT_FILENO)
                        err_sys("dup2 error to stdout");
                if(execl("./add2","add2",(char *)0) < 0)
                        err_sys("execl error");
        }
        exit(0);
}

static void sig_pipe(int signo){
        printf("SIGPIPE caught");
        exit(1);
}
```

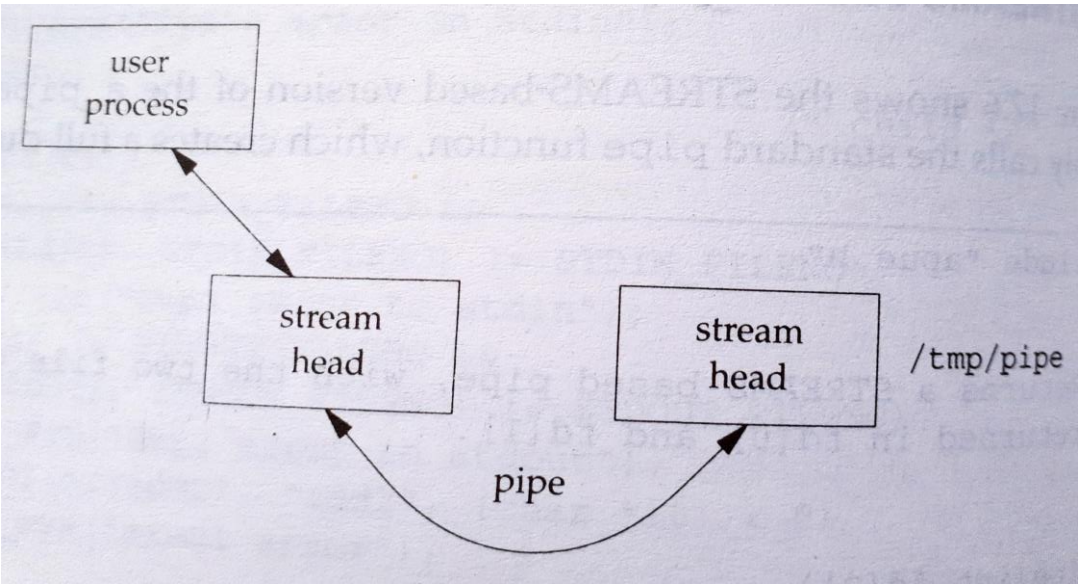
- The parent uses only *fd[0]*, and the child uses only *fd[1]*.
- Since each end of the STREAMS pipe is full duplex, the parent reads and writes *fd[0]*, and the child duplicates *fd[1]* to both standard input and standard output.

- *fattach* - gives a name to a STREAMS pipe in the file system.

```
#include<stropts.h>

int fattach(int fd, const char *path);
```

- It returns: 0 if OK, -1 on error.
- *path* - must refer to an existing file and calling process must either own the file and have write permissions to it or be running with superuser privileges.
- Once a STREAMS pipe is attached to the file system namespace, the underlying file is inaccessible.

- Any process that opens the name will gain access to the pipe, not the underlying file.
- Any processes that had the underlying file open before *fattach* was called can continue to access the underlying file.
- A pipe mounted on a name in the file system is shown below –



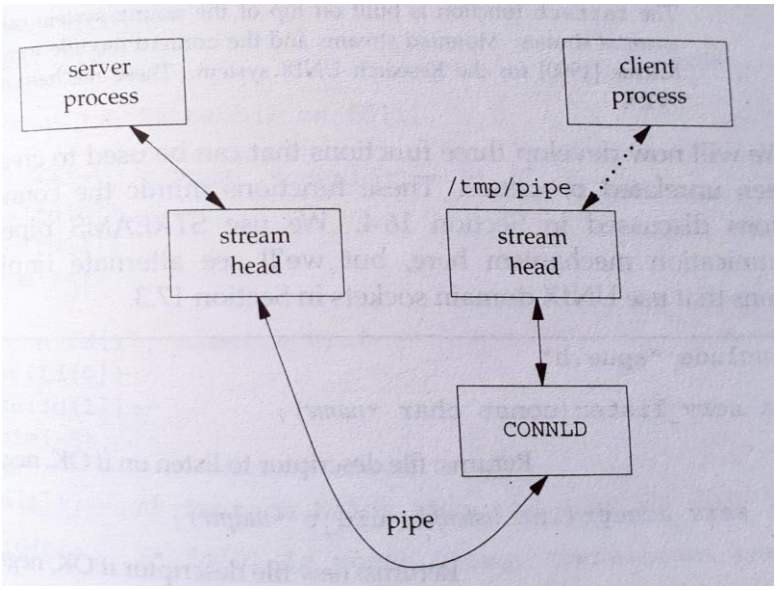
- Only one end of STREAMS pipe is attached to name in file system, other end is used to communicate with processes that open the attached filename.
- *fdetach* – undo the association between STREAMS pipe and name in the file system.

```
#include<stropts.h>

int fdetach(const char *path);
```

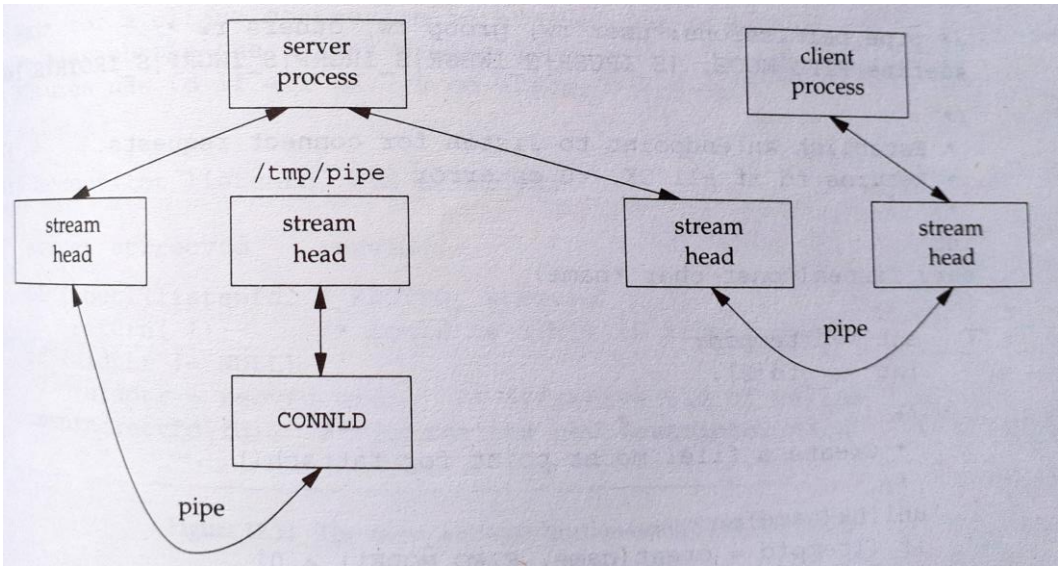
- It returns: 0 if OK, -1 on error
- After *fdetach* is called, any processes that had accessed the STREAMS pipe by opening *path* will continue to access the stream, but subsequent opens of the *path* will access the original file residing in the file system.

- Although we can attach one end of a STREAMS pipe to the file system namespace, we still have problems if multiple processes want to communicate with a server using the named STREAMS pipe.
- Data from one client will be interleaved with data from other clients writing to pipe.
- Even if we guarantee that the clients write less than PIPE_BUF bytes so that the writes are atomic, we have no way to write back to an individual client and guarantee that the intended client will read the message.
- With multiple clients reading from the same pipe, we cannot control which one will be scheduled and actually read what we send.
- The *connld* STREAMS module solves this problem.

- Before attaching a STREAMS pipe to a name in the file system, a server process can push the *connld* module on the end of the pipe that is to be attached.
- Setting up of *connld* for unique connections is done as follows –



- The server process has attached one end of its pipe to the path */tmp/pipe*.
- We show a dotted line to indicate a client process in the middle of opening the attached STREAMS pipe.
- Using *connld* to make unique connections is done as follows –



- The client process never receives an open file descriptor for the end of the pipe that it opened.
- Instead, the operating system creates a new pipe and returns one end to the client process as the result of opening */tmp/pipe*.

- The system sends the other end of the new pipe to the server process by passing its file descriptor over the existing (attached) pipe, resulting in a unique connection between the client process and the server process.
- Functions used to create unique connections between unrelated processes are -

```
#include "apue.h"

int serv_listen(const char *name);
int serv_accept(int listenfd, uid_t *uidptr);
int cli_conn(const char *name);
```

- *serv_listen* : used by server to announce its willingness to listen for client connect requests on a well-known name.
- The implementation of the function is shown below.

```
#include "apue.h"
#include<fcntl.h>
#include<stropts.h>
//pipe permissions
#define FIFO_MODE (S_IRUSR|S_IWUSR|S_IRGRP|
                        S_IWGRP|S_IROTH|S_IWOTH)
//listen for connect requests
int serv_listen(const char *name){
        int tempfd, fd[2];
        //create a mount point for fattach()
        unlink(name);
        if((tempfd=creat(name,FIFO_MODE)) < 0)
                return(-1);
        if(close(tempfd) < 0)
                return(-2);
        if(pipe(fd) < 0)
                return(-3);
        //push connld & fattach() on fd[1]
        if(ioctl(fd[1],I_PUSH,"connld") < 0){
                close(fd[0]);
                close(fd[1]);
                return(-4);
        }
        if(fattach(fd[1],name) < 0){
                close(fd[0]);
                close(fd[1]);
                return(-5);
        }
        close(fd[1]); //fattach() holds this end open
        return(fd[0]); //fd[0] - client arrives here
}
```

- Clients will use this name when they want to connect to the server.
- The return value is the server's end of the STREAMS pipe.
- *serv_accept* : used by server to wait for client's connect request to arrive.
- When one arrives, the system automatically creates a new STREAMS pipe, and the function returns one end to the server.
- Additionally, the effective user ID of the client is stored in the memory to which *uidptr* points.
- The implementation of the function is shown below.

```
#include "apue.h"
#include<stropts.h>
//wait for client connection to arrive and accept it
int serv_accept(int listenfd, uid_t *uidptr){
        struct strrecvfd recvfd;
        if(ioctl(listenfd,I_RECVFD,&recvfd) < 0)
                return(-1);
        if(uidptr != NULL)
                *uidptr = recvfd.uid; //effective UID of caller
        return(recvfd.fd); //return new descriptor
}
```

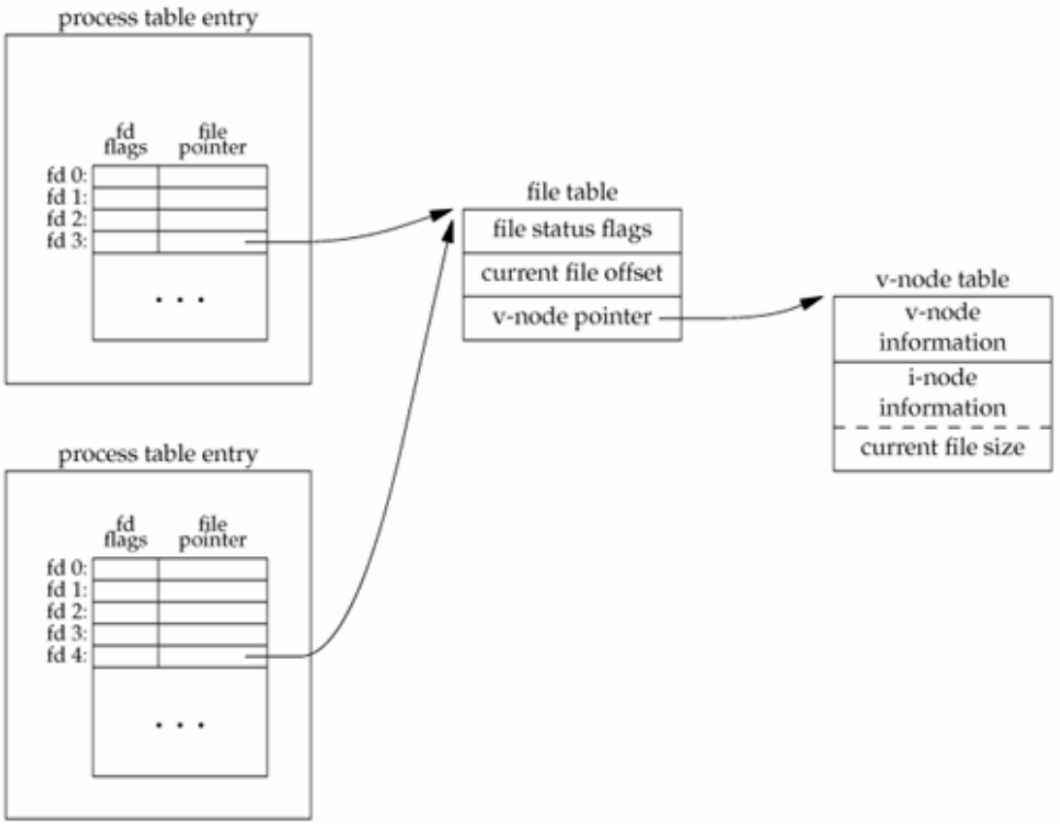
- *cli_conn* : called by client to connect to the server.
- The implementation of the function is shown below.

```
#include "apue.h"
#include<fcntl.h>
#include<stropts.h>
//create client endpoint & connect to server
int cli_conn(const char *name){
        int fd;
        //open the mounted stream
        if((fd=open(name,O_RDWR)) < 0)
                return(-1);
        if(isastream(fd) == 0){
                close(fd);
                return(-2);
        }
        return(fd);
}
```

- The *name* argument specified by the client must be the same name that was advertised by the server's call to *serv_listen*.
- On return, the client gets a file descriptor connected to the server.

(XVIII) Passing File Descriptors

- It allows one process to do everything required to open a file and simply pass back a descriptor to the calling process that can be used with all I/O functions.
- When open file descriptor is passed from one process to another, passing process and receiving process share the same file table entry.
- This is shown as follows:



- We are passing a pointer to an open file table entry from one process to another.
- This pointer is assigned the first available descriptor in the receiving process.
- What normally happens when a descriptor is passed from one process to another is that the sending process, after passing the descriptor, then closes the descriptor.
- Closing the descriptor by the sender doesn't really close the file or device, since the descriptor is still considered open by the receiving process (even if the receiver hasn't specifically received the descriptor yet).

- Functions used to send and receive file descriptors are -

```
#include "apue.h"

int send_fd(int fd, int fd_to_send);
int send_err(int fd, int status, const char *errmsg);
int recv_fd(int fd, ssize_t (*userfunc)(int, const void *, size_t));
```

- *send_fd* : used by process to pass a descriptor to another process.
- The *send_fd* function sends the descriptor *fd_to_send* across using the STREAMS pipe or UNIX domain socket represented by *fd*.
- The implementation of the function is given below.

```
#include "apue.h"
#include<stropts.h>
//pass file descriptor to another process
int send_fd(int fd, int fd_to_send){
        char buf[2];
        buf[0]=0;
        if(fd_to_send < 0){     //error
                buf[1] = -fd_to_send;
                if(buf[1] == 0)
                        buf[1] = 1;
        }
        else
                buf[1] = 0; //OK
        if(write(fd,buf,2) != 2)
                return(-1);
        if(fd_to_send >= 0)
                if(ioctl(fd,I_SENDFD,fd_to_send)<0)
                        return(-1);
        return(0);
}
```

- *send_err* : used by process to send error message to another process.
- The *send_err* function sends the *errmsg* using *fd*, followed by the status byte.
- The value of status must be in the range 1 through 255.
- The implementation of the function is given below.

```
#include "apue.h"

int send_err(int fd,int errcode,const char *msg){
        int n;
        //send the error message
        if((n=strlen(msg)) > 0)
                if(writen(fd,msg,n) != n)
                        return(-1);
        if(errcode >= 0)
                errcode = -1; //must be negative
        if(send_fd(fd, errcode) < 0)
                return(-1);
        return(0);
}
```

- *recv_fd* : called by client to receive a descriptor.
- The implementation of the function is given below.

```
#include "apue.h"
#include<stropts.h>
//receive file descriptor
int recv_fd(int fd,ssize_t (*userfunc)(int,const void *,size_t)){
        int newfd, nread, flag, status;
        char *ptr, buf[MAXLINE];
        struct strbuf dat;
        struct strrecvfd recvfd;
        status = -1;
        for(;;){
                dat.buf = buf;
                dat.maxlen = MAXLINE;
                flag = 0;
                if(getmsg(fd,NULL,&dat,&flag) < 0)
                        err_sys("getmsg error");
                nread = dat.len;
                if(nread == 0){
                        err_ret("connection closed by server");
                        return(-1);
                }
                for(ptr=buf; ptr < &buf[nread]; ){
                        if(*ptr++ == 0){
                                if(ptr != &buf[nread-1])
                                        err_dump("msg format error");
                                status = *ptr & 0xFF;
                                if(status == 0){
                                        if(ioctl(fd,I_RECVFD,&recvfd)<0)
                                                return(-1);
                                        newfd = recvfd.fd;
                                }
                                else
                                        newfd = -status;
                                nread -= 2;
                        }
                }
                if(nread > 0)
                        if((*userfunc)(STDERR_FILENO,buf,nread)!=nread)
                                return(-1);
                if(status >= 0)         //final data has arrived
                        return(newfd);  //descriptor or status
        }
}
```

- Clients call *recv_fd* to receive a descriptor.
- If all is OK (the sender called *send_fd*), the non-negative descriptor is returned as the value of the function.
- Otherwise, the value returned is the status that was sent by *send_err* (a negative value in the range 1 through -255).
- Additionally, if an error message was sent by the server, the client's *userfunc* is called to process the message.
- The first argument to *userfunc* is the constant STDERR_FILENO, followed by a pointer to the error message and its length.
- The return value from *userfunc* is the number of bytes written or a negative number on error.
- Often, the client specifies the normal write function as the *userfunc*.
- The protocol used in the above implementations is unique.
- To send a descriptor, *send_fd* sends two bytes of 0, followed by the actual descriptor.
- To send an error, *send_err* sends the *errmsg*, followed by a byte of 0, followed by the absolute value of the status byte (1 through 255).
- The *recv_fd* function reads everything on the s-pipe until it encounters a null byte.
- Any characters read up to this point are passed to the caller's *userfunc*.
- The next byte read by *recv_fd* is the status byte.
- If the status byte is 0, a descriptor was passed; otherwise, there is no descriptor to receive.

(XIX) Client-Server Properties

- Some of the properties of clients and servers that are affected by the various types of IPC used between them are listed below.
- The simplest type of relationship is to have the client *fork* and *exec* the desired server.
- Two half-duplex pipes can be created before the *fork* to allow data to be transferred in both directions.
- The server that is executed can be a set-user-ID program, giving it special privileges.
- Also, the server can determine the real identity of the client by looking at its real user ID.

- With this arrangement, we can build an open server.
- It opens files for the client instead of the client calling the *open* function.
- This way, additional permission checking can be added, above and beyond the normal UNIX system user/group/other permissions.
- We assume that the server is a set-user-ID program, giving it additional permissions (root access).

- The server uses the real user ID of the client to determine whether to give it access to the requested file.
- This way, we can build a server that allows certain users permissions that they don't normally have.

- In this example, since the server is a child of the parent, all the server can do is pass back the contents of the file to the parent.
- Although this works fine for regular files, it cannot be used for special device files, for example.
- We would like to be able to have the server open the requested file and pass back the file descriptor.
- Whereas a parent can pass a child an open descriptor, a child cannot pass a descriptor back to the parent.

- Another case is when the server is a daemon process that is contacted using some form of IPC by all clients.
- We can't use pipes for this type of client-server.
- A form of named IPC is required, such as FIFOs or message queues.
- With FIFOs, we saw that an individual per client FIFO is also required if the server is to send data back to the client.
- If the client-server application sends data only from the client to the server, a single well-known FIFO suffices.
- Multiple possibilities exist with message queues –
  – A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient.
  For example, the clients can send their requests with a type field of 1.
  Included in the request must be the client's process ID (PID).
  The server then sends the response with the type field set to the client's PID.
  The server receives only the messages with a type field of 1, and the clients receive only the messages with a type field equal to their PIDs.
  – Alternatively, an individual message queue can be used for each client.
  Before sending the first request to a server, each client creates its own message queue with a key of IPC_PRIVATE.
  The server also has its own queue, with a key or identifier known to all clients.
  The client sends its first request to the server's well-known queue, and this request must contain the message queue ID of the client's queue.
  The server sends its first response to the client's queue, and all future requests and responses are exchanged on this queue.

- Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).
- The problem with this type of client-server relationship (the client and the server being unrelated processes) is for the server to identify the client accurately.
- Unless the server is performing a non-privileged operation, it is essential that the server know who the client is.
  This is required, for example, if the server is a set-user-ID program.
- Although all these forms of IPC go through the kernel, there is no facility provided by them to have the kernel identify the sender.
- With message queues, if a single queue is used between the client and the server (so that only a single message is on the queue at a time, for example), the *msg_lspid* of the queue contains the process ID of the other process.
- But when writing the server, we want the effective user ID of the client, not its process ID.
  There is no portable way to obtain the effective user ID, given the process ID.
- Naturally, the kernel maintains both values in the process table entry, but other than rummaging around through the kernel's memory, we can't obtain one, given the other.

- Another technique can be used to allow the server to identify the client.
- The client must create its own FIFO and set the file access permissions of the FIFO so that only user-read and user-write are on.
- We assume that the server has superuser privileges (or else it probably wouldn't care about the client's true identity), so the server can still read and write to this FIFO.
- When the server receives the client's first request on the server's well-known FIFO which must contain the identity of the client-specific FIFO, the server calls either *stat* or *fstat* on the client-specific FIFO.
- The server assumes that the effective user ID of the client is the owner of the FIFO (the *st_uid* field of the *stat* structure).
- The server verifies that only the user-read and user-write permissions are enabled.
- As another check, the server should also look at the three times associated with the FIFO (the *st_atime*, *st_mtime*, and *st_ctime* fields of the *stat* structure) to verify that they are recent (no older than 15 or 30 seconds, for example).
- If a malicious client can create a FIFO with someone else as the owner and set the file's permission bits to user-read and user-write only, then the system has other fundamental security problems.

(XX) Open Server Version 1

- Using file descriptor passing, an *open server* is developed: a program that is executed by a process to open one or more files.
- But instead of sending the contents of the file back to the calling process, the server sends back an open file descriptor.
- This lets the server work with any type of file (such as a device or a socket) and not simply regular files.
- It also means that a minimum of information is exchanged using IPC: the filename and open mode from the client to the server, and the returned descriptor from the server to the client.
- The contents of the file are not exchanged using IPC.

- There are several advantages in designing the server to be a separate executable program.
  – The server can easily be contacted by any client, similar to the client calling a library function. We are not hard coding a particular service into the application, but designing a general facility that others can reuse.
  – If we need to change the server, only a single program is affected. Conversely, updating a library function can require that all programs that call the function be updated (i.e., relinked with the link editor). Shared libraries can simplify this updating.
  – The server can be a set-user-ID program, providing it with additional permissions that the client does not have. A library function (or shared library function) cannot provide this capability.

- The client process creates an s-pipe (either a STREAMS-based pipe or a UNIX domain socket pair) and then calls *fork* and *exec* to invoke the server.
- The client sends requests across the s-pipe, and the server sends back responses across the s-pipe.
- The following application protocol is defined between the client and the server.
  – The client sends a request of the form "*open <path_name> <open_mode> \0*" across the s-pipe to the server. The *<open_mode>* is the numeric value, in ASCII decimal, of the second argument to the open function. This request string is terminated by a null byte.
  – The server sends back an open descriptor or an error by calling either *send_fd* or *send_err*.

- We first have the header, *open.h*, which includes the standard headers and defines the function prototypes.

```
#include "apue.h"
#include<errno.h>

#define CL_OPEN "open" //client's request for server

int csopen(char *, int);
```

- The *main* function is a loop that reads a pathname from standard input and copies the file to standard output.
- The function calls *csopen* to contact the open server and return an open descriptor.

```
#include "open.h"
#include<fcntl.h>

#define BUFSIZE 8192

int main(int argc, char *argv[]){
        int n, fd;
        char buf[BUFSIZE], line[MAXLINE];
        //read filename to cat from stdin
        while(fgets(line,MAXLINE,stdin)!=NULL){
                if(line[strlen(line)-1]=='\n')
                        line[strlen(line)-1] = 0;
                //open the file
                if((fd=csopen(line,O_RDONLY)) < 0)
                        continue;
                //cat to stdout
                while((n=read(fd,buf,BUFSIZE)) > 0)
                        if(write(STDOUT_FILENO,buf,n)!=n)
                                err_sys("write error");
                if(n < 0)
                        err_sys("read error");
                close(fd);
        }
        exit(0);
}
```

- The function *csopen* does the *fork* and *exec* of the server, after creating the s-pipe.

```
#include "open.h"
#include<sys/uio.h>
//open the file
int csopen(char *name, int oflag){
        pid_t pid;
        int len;
        char buf[10];
        struct iovec iov[3];
        static int fd[2] = {-1,-1};
        if(fd[0] < 0){ //fork/exec open server
                if(s_pipe(fd) < 0)
                        err_sys("s_pipe error");
                if((pid=fork()) < 0)
                        err_sys("fork error");
                else if(pid == 0){      //child
                        close(fd[0]);
                        if(fd[1]!=STDIN_FILENO &&
                            dup2(fd[1],STDIN_FILENO)!=STDIN_FILENO)
                                err_sys("dup2 error to stdin");
                        if(fd[1]!=STDOUT_FILENO &&
                            dup2(fd[1],STDOUT_FILENO)!=STDOUT_FILENO)
                                err_sys("dup2 error to stdout");

                        if(execl("./opend","opend",(char *)0) < 0)
                                err_sys("execl error");
                }
                close(fd[1]);           //parent
        }
        sprintf(buf,"%d",oflag); //oflag to ASCII
        iov[0].iov_base = CL_OPEN " "; //string concatenation
        iov[0].iov_len = strlen(CL_OPEN)+1;
        iov[1].iov_base = name;
        iov[1].iov_len = strlen(name);
        iov[2].iov_base = buf;
        iov[2].iov_len = strlen(buf)+1;
        len = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;
        if(writev(fd[0],&iov[0],3) != len)
                err_sys("writev error");
        //read descriptor, returned errors handled by write()
        return(recv_fd(fd[0], write));
}
```

- The child closes one end of the pipe, and the parent closes the other.
- For the server that it executes, the child also duplicates its end of the pipe onto its standard input and standard output.
- Another option would have been to pass the ASCII representation of the descriptor *fd[1]* as an argument to the server).
- The parent sends to the server the request containing the pathname and open mode.

- Finally, the parent calls *recv_fd* to return either the descriptor or an error.
- If an error is returned by the server, *write* is called to output the message to standard error.

- The program *opend* that is executed by the client in the previous function *csopen* is the open server.
- First, we have the *opend.h* headee, which includes the standard headers and declares the global variables and function prototypes.

```
#include "apue.h"
#include<errno.h>

#define CL_OPEN "open" //client's request for server

extern char errmsg[]; //error message to return to client
extern int oflag; //open() flag
extern char *pathname; //file to open for client

int cli_args(int, char **);
void request(char *, int, int);
```

- The *main* function reads the requests from the client on the s-pipe (its standard input) and calls the function request.

```
#include "opend.h"

char errmsg[MAXLINE];
int oflag;
char *pathname;

int main(void){
        int nread;
        char buf[MAXLINE];
        //read arg buffer from client, process request
        for(;;){
                if((nread=read(STDIN_FILENO,buf,MAXLINE))<0)
                        err_sys("read error on stream pipe");
                else if(nread == 0)
                        break; //client closed the pipe
                request(buf, nread, STDOUT_FILENO);
        }
        exit(0);
}
```

- The function *request* does all the work.
- It calls the function *buf_args* to break up the client's request into a standard *argv*-style argument list and calls the function *cli_args* to process the client's arguments.
- If all is OK, *open* is called to open the file, and then *send_fd* sends the descriptor back to the client across the s-pipe (its standard output).
- If an error is encountered, *send_err* is called to send back an error message, using the client-server protocol.

```
#include "opend.h"
#include<fcntl.h>

void request(char *buf, int nread, int fd){
        int newfd;
        if(buf[nread-1] != 0){
                sprintf(errmsg,"request not null terminated: %*.*s",nread,nread,buf);
                send_err(fd, -1, errmsg);
                return;
        }
        //parse args & set options
        if(buf_args(buf,cli_args) < 0){
                send_err(fd, -1, errmsg);
                return;
        }
        if((newfd = open(pathname,oflag)) < 0){
                sprintf(errmsg,"can't open %s: %s",pathname,strerror(errno));
                send_err(fd, -1, errmsg);
                return;
        }
        //send the descriptor
        if(send_fd(fd, newfd) < 0)
                err_sys("send_fd error");
        //done with descriptor
        close(newfd);
}
```

- The client's request is a null-terminated string of white-space-separated arguments.
- The function *buf_args* in breaks this string into a standard *argv*-style argument list and calls a user function to process the arguments.

- The ISO C function *strtok* is used to tokenize the string into separate arguments.

```
#include "apue.h"

#define MAXARGC 50 //max no. of arguments in buf
#define WHITE " \t\n" //for tokenizing arguments

int buf_args(char *buf,int (*optfunc)(int,char **)){
        char *ptr, *argv[MAXARGC];
        int argc;
        if(strtok(buf,WHITE) == NULL) //argv[0] required
                return(-1);
        argv[argc = 0] = buf;
        while((ptr = strtok(NULL,WHITE))!=NULL){
                if(++argc >= MAXARGC-1)
                        return(-1);
                argv[argc] = ptr;
        }
        argv[++argc] = NULL;
        return((*optfunc)(argc, argv));
}
```

- The server's function that is called by *buf_args* is *cli_args*.
- It verifies that the client sent the right number of arguments and stores the pathname and open mode in global variables.

```
#include "opend.h"

int cli_args(int argc, char **argv){
        if(argc!=3 || strcmp(argv[0],CL_OPEN)!=0){
                strcpy(errmsg,"Usage: <path> <oflag>");
                return(-1);
        }
        //save ptr to pathname to open
        pathname = argv[1];
        oflag = atoi(argv[2]);
        return(0);
}
```

- This completes the open server that is invoked by a *fork* and *exec* from the client.
- A single s-pipe is created before the *fork* and is used to communicate between the client and the server.
- With this arrangement, we have one server per client.