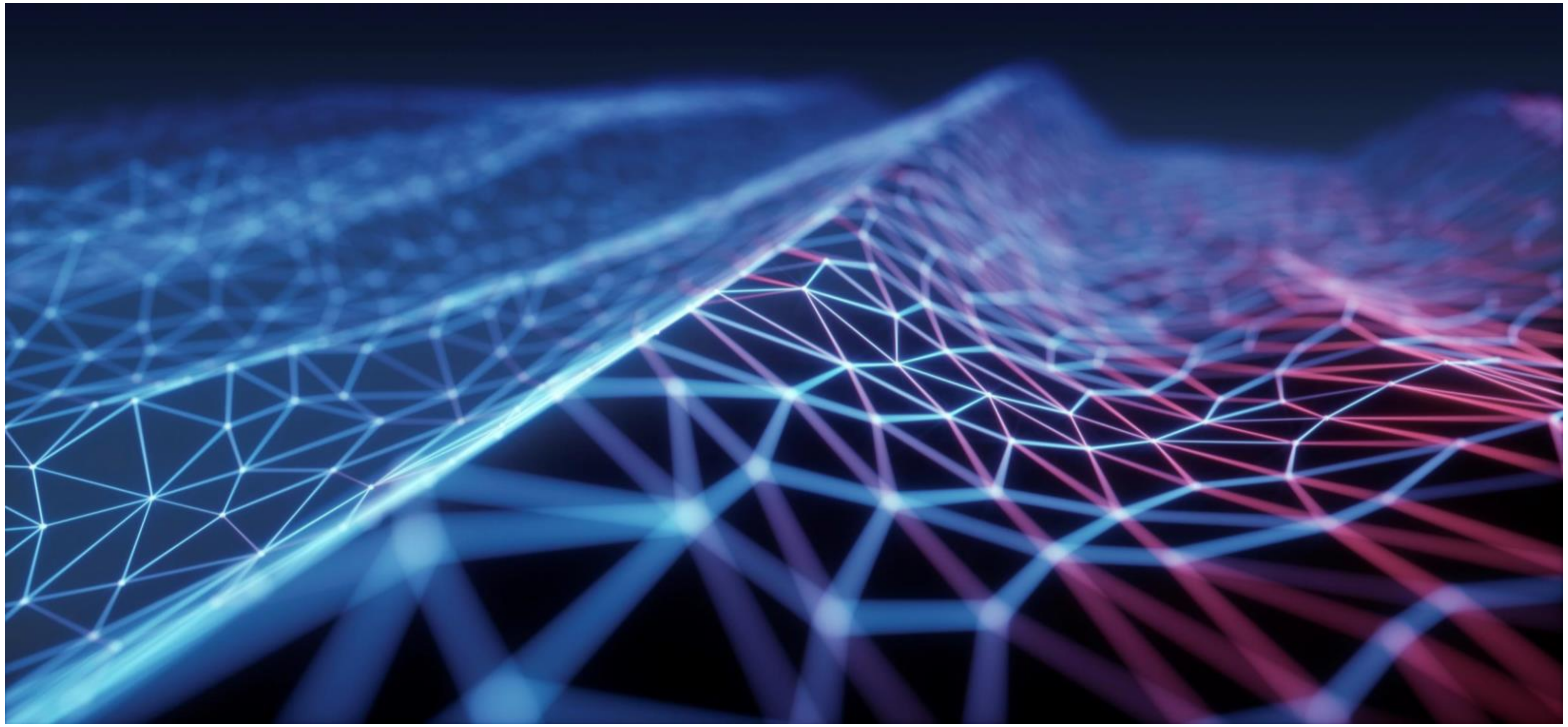


Vidyayāmruthamashnuthe

UNIX PROGRAMMING

AKASH HEGDE

ASSISTANT PROFESSOR

DEPARTMENT OF ISE

MODULE 4

Process Control	User and Group IDs
	system Function
	I/O Redirection
IPC	popen and pclose
	Coprocesses
	FIFOs
	Message Queues
Shared Memory	Client-Server Properties
	Stream pipes
	File Descriptors
	Client-Server Connection

CHANGING USER IDs AND GROUP IDs

- User and Group ID – unique integer used to identify users and groups in UNIX.
- Can be dynamically changed as per privilege of the task.
- Three types of IDs 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 USER IDs AND GROUP IDs

- Changing system date, access control, read or write permissions of a particular file – all based on user IDs and group IDs.
- Programs need additional privileges or to gain access to resources – change user/group ID to some ID that has appropriate privilege/access.
- Programs need lower privileges or to prevent access to resources – change user/group ID to some ID that does not have appropriate privilege/access.

CHANGING USER IDs AND GROUP IDs

- *Least privilege model* – programs should use the least privilege necessary to accomplish any given task.
- Reduces security compromises by malicious users who trick programs to use their privileges in unintended/harmful ways.
- *setuid* – set real user ID and effective user ID
  setgid – set real group ID and effective group ID

```
#include<unistd.h>

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

- Both return 0 if successful, -1 on error.

CHANGING USER IDs AND GROUP IDs

- 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: _POSIX_SAVED_IDS is true in current system implementation.
  Saved IDs are a mandatory feature in POSIX.1.

CHANGING USER IDs AND GROUP IDs

- Real user ID –
  - Only a superuser process can change real user ID.
  - 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*.

CHANGING USER IDs AND GROUP IDs

- Effective user ID –
  - 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.

CHANGING USER IDs AND GROUP IDs

- Saved set-user-ID –
  - 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.

CHANGING USER IDs AND GROUP IDs

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

CHANGING USER IDs AND GROUP IDs

- *setreuid* and *setregid* functions – swapping of 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.
- Rule – unprivileged user can always swap between real user ID and effective user ID.
- Allows set-user-ID program to swap to user's normal permissions and swalp back again later for set-user-ID operations.

CHANGING USER IDs AND GROUP IDs

- *seteuid* and *setegid* functions – changing effective user ID or effective group ID.

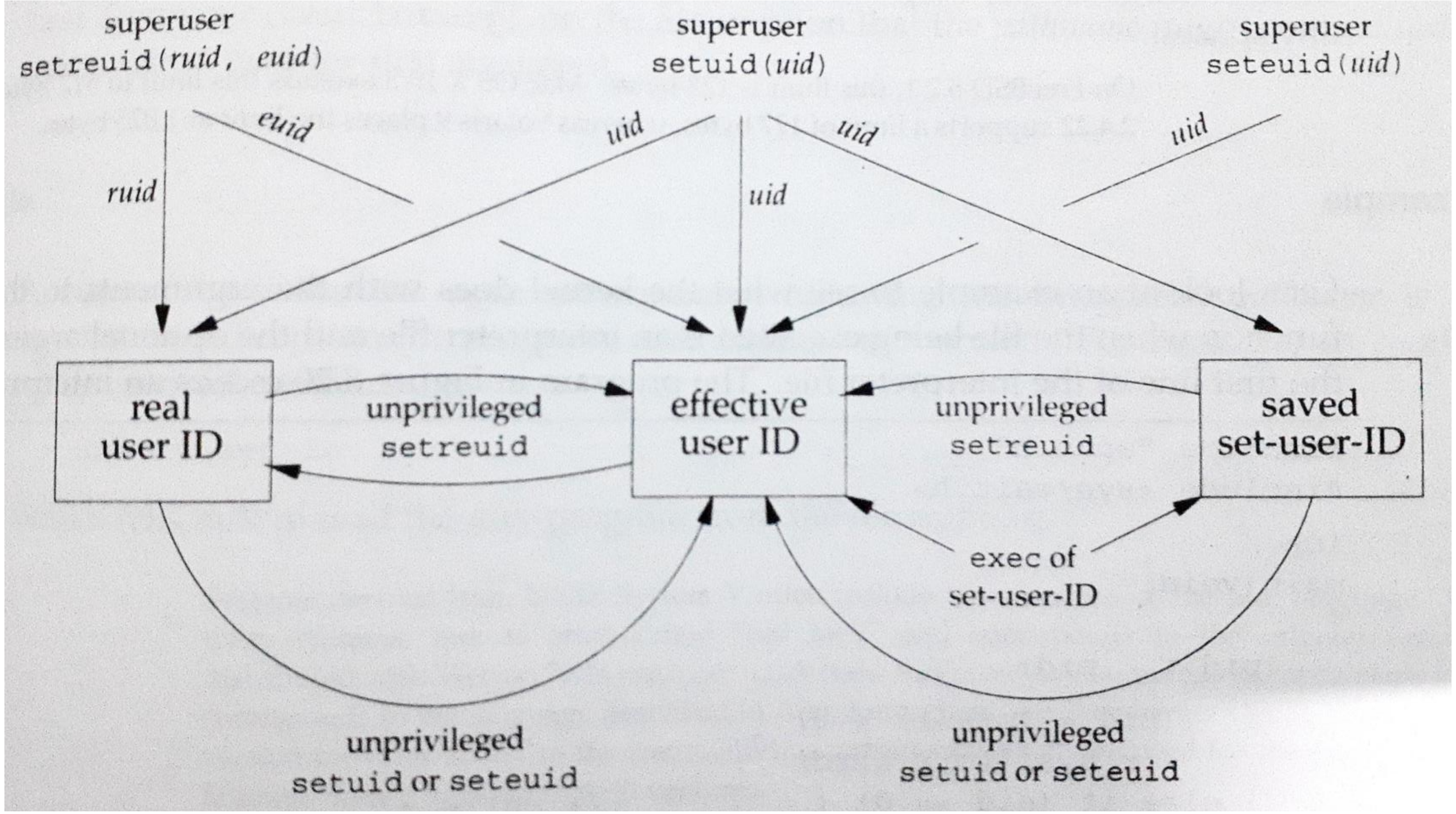
```
#include<unistd.h>

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

- Both 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*.

CHANGING USER IDs AND GROUP IDs

- *Summary of modification of all 3 user IDs –*



INTERPRETER FILES

- All contemporary UNIX systems support interpreter files.
- Text files that begin with a line of the form –
  #! pathname [optional-argument]
- Space between ! and *pathname* is optional.
- Examples - #! /bin/sh, #!/bin/bash
- *pathname* – absolute pathname, as no special operations are performed on it (PATH is not used).

INTERPRETER FILES

- Recognition of interpreter files done within kernel as part of processing the *exec* system call.
- 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.
  Size limit on first line on various systems (FreeBSD - 128 bytes, Mac OS X – 512 bytes, Linux – 127 bytes, Solaris 9 – 1023 bytes).

INTERPRETER FILES

- *optional-argument* usually the ***–f*** option – tells *pathname* where to read a particular program file.
- Uses of interpreter files –
  - 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*.

SYSTEM FUNCTION

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

SYSTEM FUNCTION

- Prototype of *system* function –

```
#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 FUNCTION

- *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*.

SYSTEM FUNCTION

- Advantage of using *system* instead of using *fork* and *exec* directly – *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.
- Change back to normal permissions after the *fork*, before calling *exec*.
- The *system* function should <u>never</u> be used from set-user-ID or set-group-ID program.

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.
- *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.

PROCESS ACCOUNTING

- 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
};
```

PROCESS ACCOUNTING

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

ac_flag	**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

PROCESS ACCOUNTING

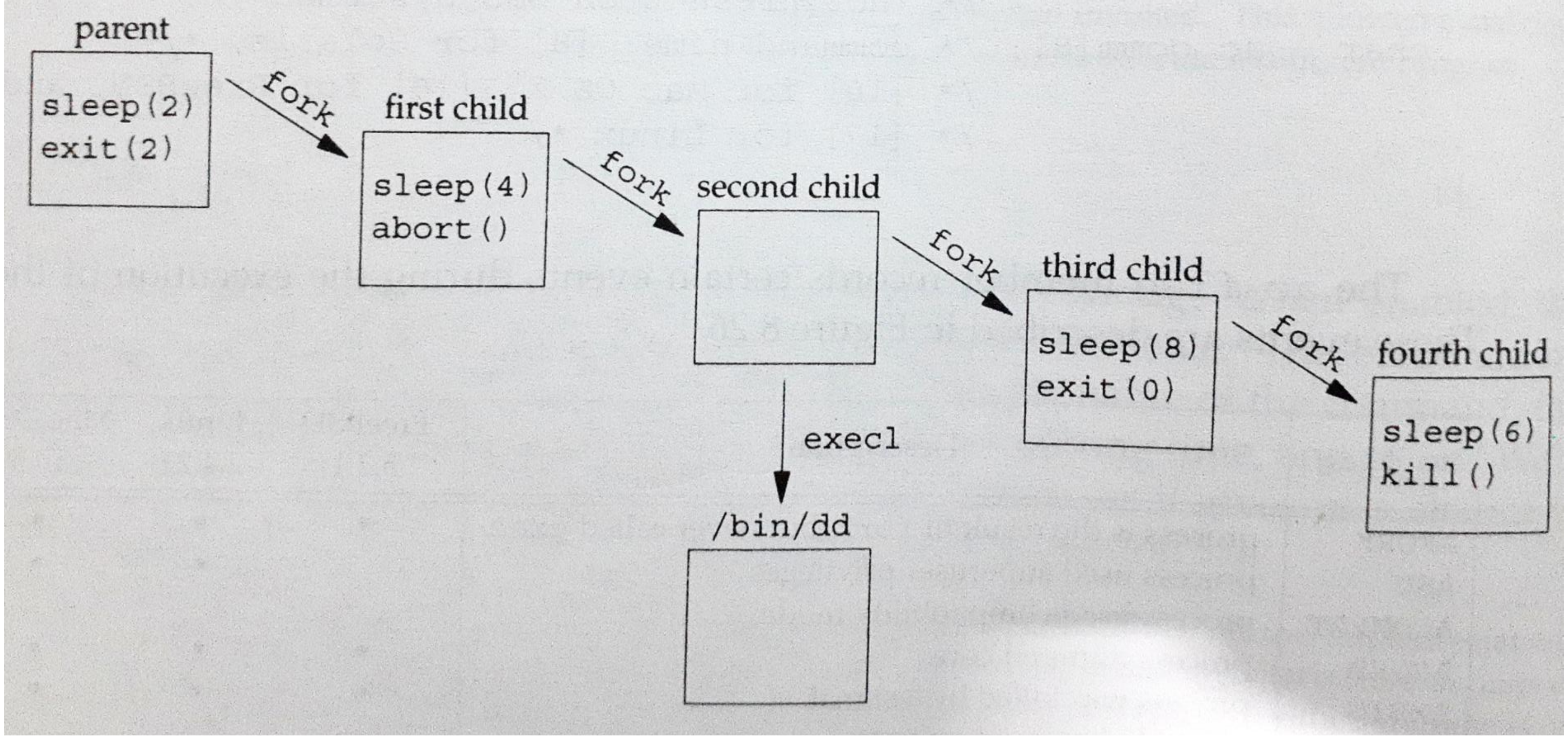
- Data required for accounting record is kept by kernel in the process table and initialized whenever a new process is created.
- Each accounting record is written when the process terminates.
- Order of records in accounting file corresponds to termination order of the processes, not the order in which they are started.
- To know starting order, go through 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.

PROCESS ACCOUNTING

- 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.

PROCESS ACCOUNTING

- Process structure to obtain accounting data - Program calls *fork* four times. Each child does something different and then terminates.



PROCESS ACCOUNTING

- Test procedure:
  - 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.

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 – *getpwuid(getuid())*
- If single user has multiple login names, each with same user ID (different login shells for each entry) - system keeps track and *getlogin* can fetch the login name.

```
#include<unistd.h>

char *getlogin(void);
```

- Returns – pointer to string giving login name if OK, NULL on error

USER IDENTIFICATION

- *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.

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);
```

- Returns – elapsed wall clock time (in clock ticks) if OK, -1 on error.

PROCESS TIMES

- 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
};
```

PROCESS TIMES

- Structure does not contain any measurement for wall clock time.
- 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 – 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 = wall clock time.

PROCESS TIMES

- 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*.

I/O REDIRECTION

- 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);
```

I/O REDIRECTION

- *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.

I/O REDIRECTION

- 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.

I/O REDIRECTION

- 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;
}
```

INTERPROCESS COMMUNICATION (IPC)

- 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.
- Types of IPC in UNIX system - half-duplex pipes, full-duplex pipes, named full-duplex pipes, FIFOs, message queues, semaphores, shared memory, sockets and STREAMS.

PIPES

- Oldest form of UNIX System IPC and provided by all UNIX systems.
- 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.
- 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.

PIPES

- Pipe is created by calling the *pipe* function:

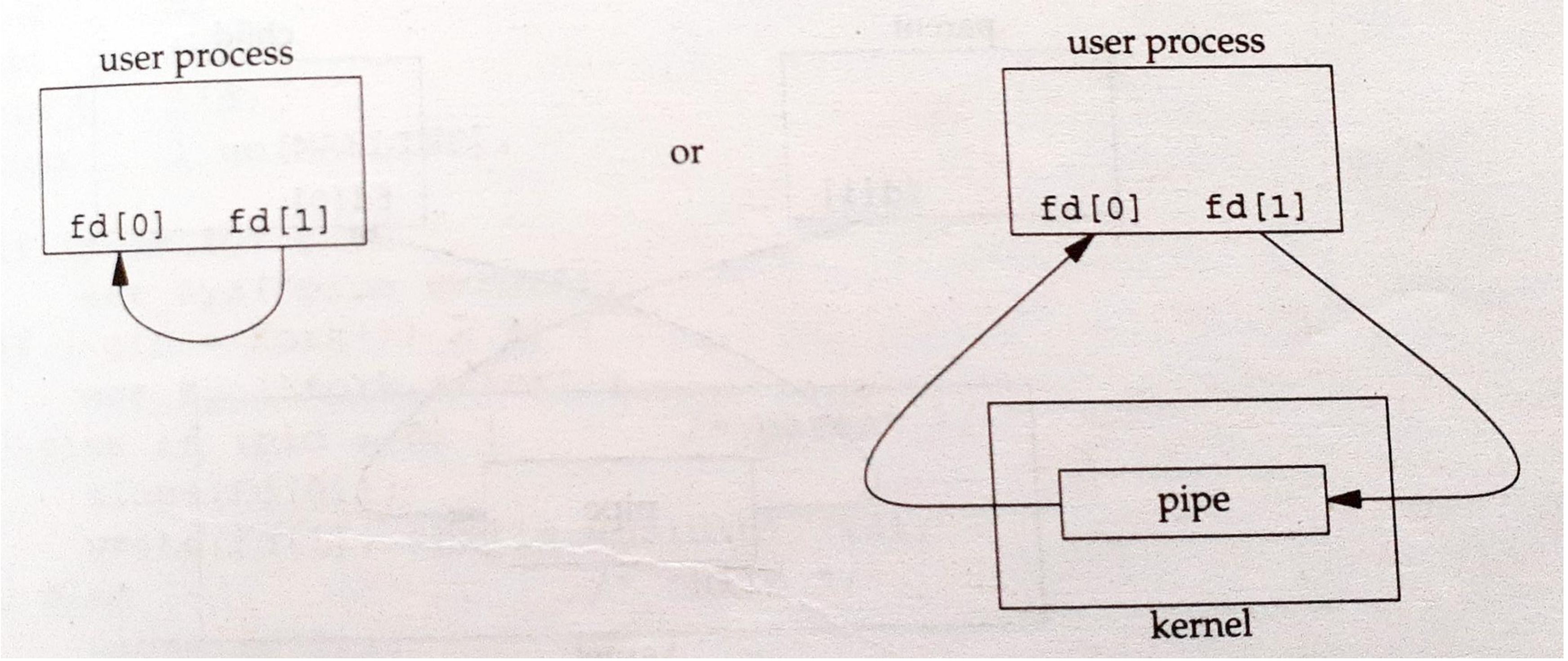
```c
#include<unistd.h>

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

- Returns: 0 if OK, -1 on error
- Two file descriptors returned through *fd:*
  - *fd[0]* - open for reading
  - *fd[1]* - open for writing
- Output of *fd[1]* is the input of *fd[0]*.

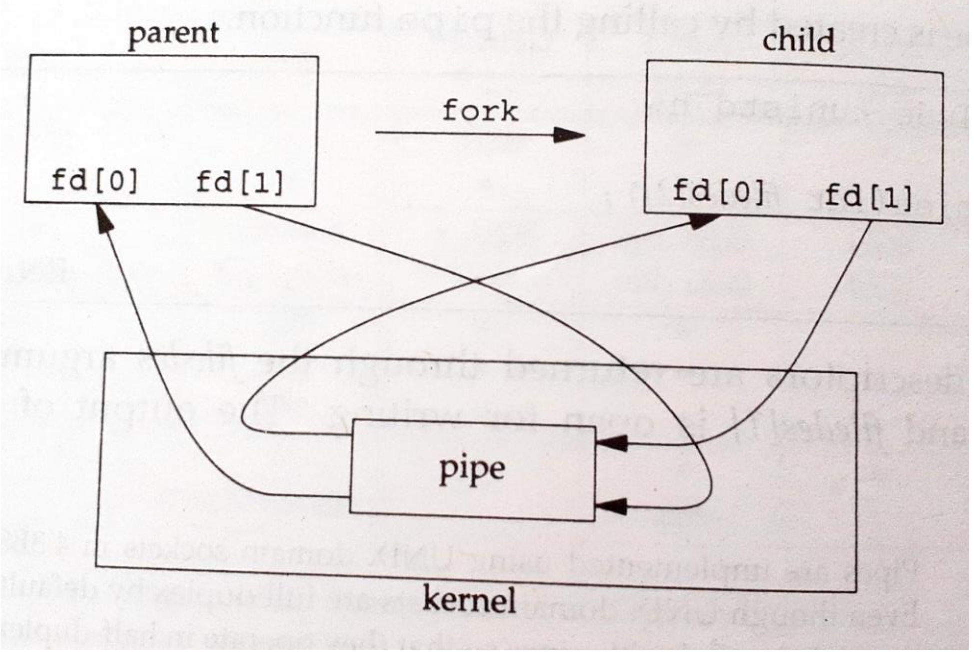
PIPES

- 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.



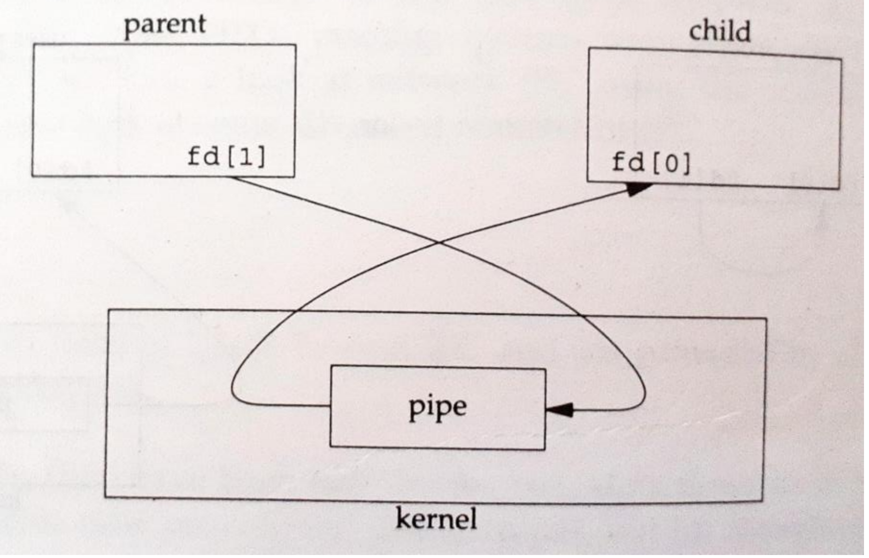
PIPES

- *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. Process that calls *pipe* then calls *fork*, creating an IPC channel from parent to child process, or vice versa.



PIPES

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



PIPES

- 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.

PIPES

- 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*.

POPEN AND PCLOSE FUNCTIONS

- Work done by *popen* and *pclose* functions – 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.

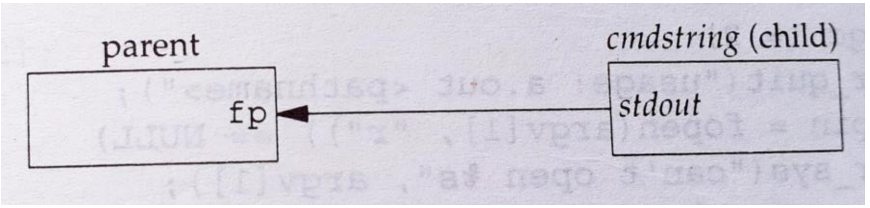
```
#include<stdio.h>

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

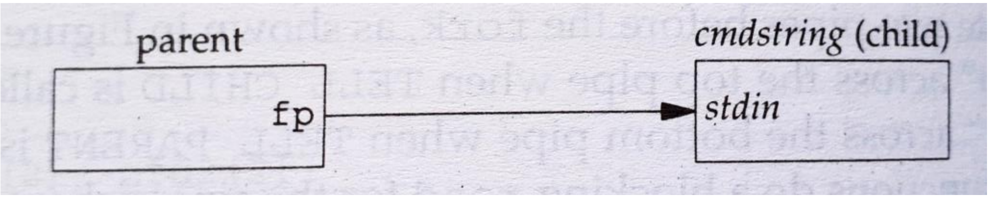
- Return of *popen*: file pointer if OK, NULL on error
- Return of *pclose*: termination status of *cmdstring*, -1 on error

POPEN AND PCLOSE FUNCTIONS

- *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*.



POPEN AND PCLOSE FUNCTIONS

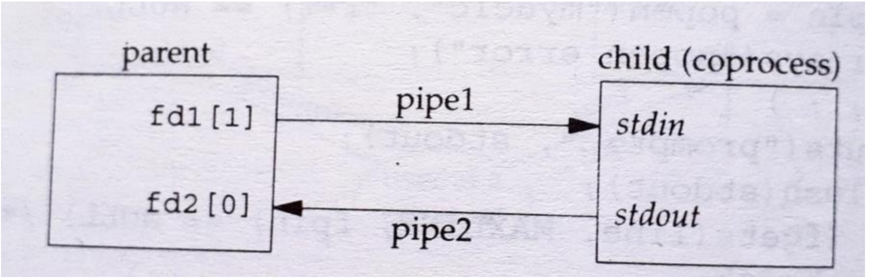
- *pclose* closes the standard I/O stream, waits for the command to terminate and returns the termination status of the shell.
- If the shell cannot be executed, termination status returned by *pclose* is as if the shell had executed *exit(127)*.
- *popen* should never be called by a set-user-ID or set-group-ID 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.

COPROCESSES

- Filter – program that reads from standard input and writes to standard output.
- Normally connected linearly in shell pipelines.
- Filter becomes a ***coprocess*** when same program generates filter's input and reads the filter's output.
- Only Korn shell provides coprocesses.
- Normally runs in the background from a shell.
- Standard input and standard output are connected to another program using a pipe.

COPROCESSES

- *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.
- Write to its standard input, let it operate on data, read from its standard output.



- Process creates two pipes: one is standard input of the coprocess, other is standard output of the coprocess.

FIFOs

- Sometimes called *named pipes*.
  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);
```

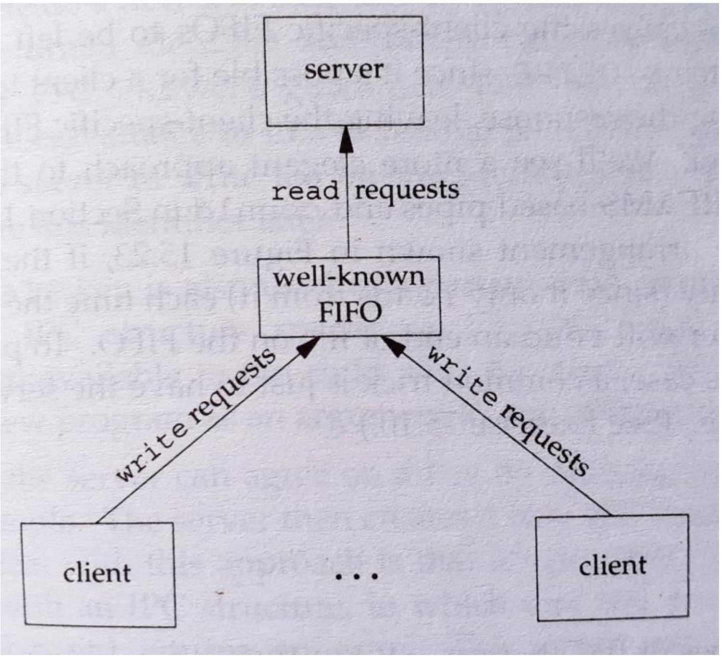
- Returns: 0 if OK, -1 on error.

FIFOs

- 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:
  - O_NONBLOCK 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.
  - O_NONBLOCK 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.

FIFOs

- *Client-server communication using a FIFO:*
- Server 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.

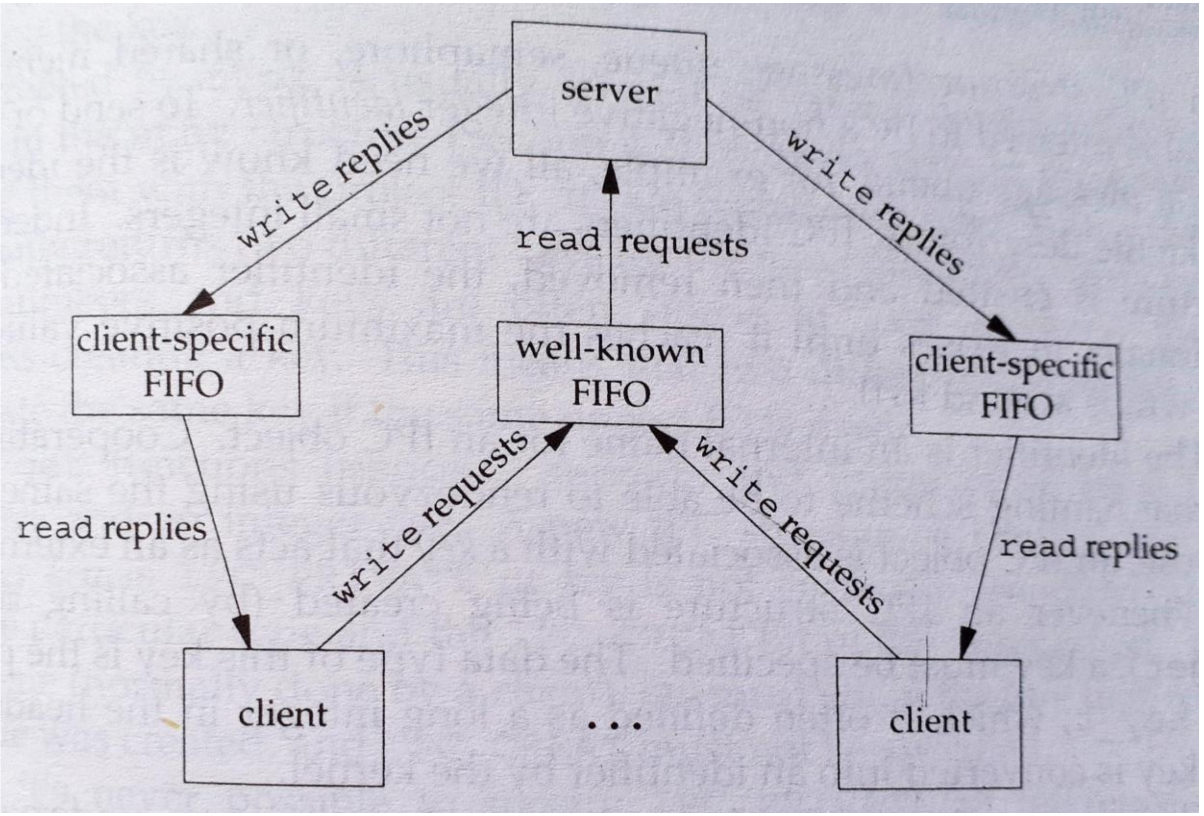


FIFOs

- Multiple writers for the FIFO – requests sent by the clients to the server must be < PIPE_BUF bytes in size.
- Prevents any interleaving between the *write* calls of the clients.
- Problem – how to send back the replies from server to each client.
- Single FIFO cannot be used as clients wouldn't know when to read their response versus responses for other clients.

FIFOs

- Solution - each client sends its PID with the request. Server then creates unique FIFO for each client using a pathname based on client's PID.

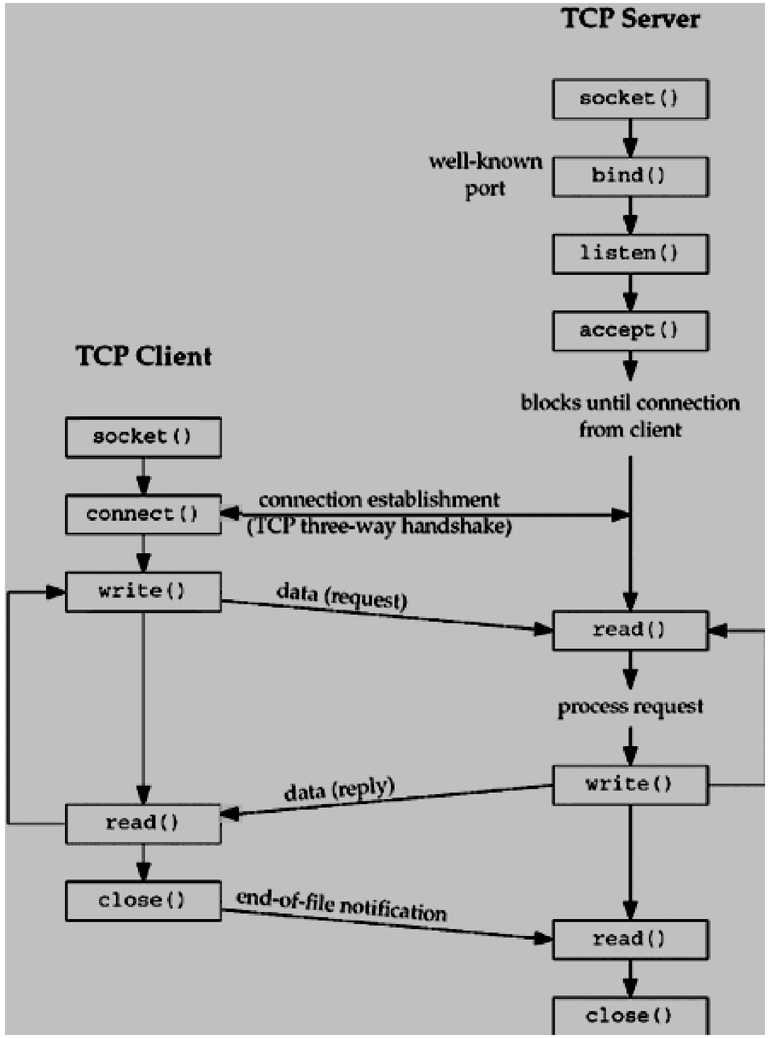


FIFOs

- Drawback – impossible for the server to tell whether a client crashes.
- 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 – 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.

CLIENT-SERVER CONNECTION

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



CLIENT-SERVER CONNECTION

- *socket*() - 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);
```

- Returns: socket descriptor if successful, -1 on error.
- Parameters: *family* (protocol family), *type* (kind of socket), *protocol* (specific protocol type or 0 for system default)

CLIENT-SERVER CONNECTION

- *connect*() - 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);
```

- Returns: 0 if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *serv_addr* (pointer to socket structure that contains destination IP and port), *addrlen* (size of socket structure)

CLIENT-SERVER CONNECTION

- *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);
```

- Returns: 0 if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *my_addr* (pointer to socket structure that contains local IP and port), *addrlen* (size of socket structure)

CLIENT-SERVER CONNECTION

- *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);
```

- Returns: 0 if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *backlog* (number of allowed connections)

CLIENT-SERVER CONNECTION

- *accept*() - 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);
```

- Returns: non-negative if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *cli_addr* (pointer to socket structure that contains client IP and port), *addrlen* (size of socket structure)

CLIENT-SERVER CONNECTION

- *send*() / *write()* - 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);
```

- Returns: number of bytes sent if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *msg* (pointer to data that must be sent), *len* (length of data to be sent), *flags* (set to 0)

CLIENT-SERVER CONNECTION

- *recv*() / *read()* - used to send 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);
```

- Returns: number of bytes read into the buffer if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor), *buf* (pointer to buffer that reads incoming information), *len* (maximum length of buffer), *flags* (set to 0)

CLIENT-SERVER CONNECTION

- *close()* - used to close the communication between the client and the server.

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

int close(int sockfd);
```

- Returns: 0 if successful, -1 on error.
- Parameters: *sockfd* (socket descriptor)

MESSAGE QUEUES

- Linked list of messages stored within the kernel and identified by a message queue identifier.
- Message queue – *queue*, message queue identifier – *queue ID*
- *msgget* – new queue is created or existing queue is opened
- *msgsnd* – new messages are added to the end of the queue
- *msgrcv* – messages are fetched from a queue
- Contents of message – type field, length, actual data bytes.

MESSAGE QUEUES

- 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
        ...
};
```

MESSAGE QUEUES

- Permission structure *ipc_perm* –

```
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
        ...
};
```

MESSAGE QUEUES

- *msgget* – either open an existing message queue or create new queue.

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

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

- 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.

MESSAGE QUEUES

- *msgctl* – performs various control operations on a queue.

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

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

- Returns: 0 if OK, -1 on error
- *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.

MESSAGE QUEUES

- *msgsnd* – data is placed onto a message queue.

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

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

- 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.

MESSAGE QUEUES

- No message data if *nbytes* is 0.
- If largest message to be sent is 512 bytes, structure *mymesg* is defined and *ptr* then points to *mymesg* 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.
- *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*).

MESSAGE QUEUES

- *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) – causes *msgsnd* to return immediately with an error of EAGAIN.
- If IPC_NOWAIT not specified – operation 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 – no reference count with each queue (present in open files).

MESSAGE QUEUES

- *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);
```

- Returns: 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.

MESSAGE QUEUES

- *type* argument lets us specify which message we want.
  - *type* == 0 – first message on queue is returned.
  - *type* > 0 – first message on queue whos message type = *type* is returned.
  - *type* < 0 – first message on queue who message type is the lowest value <= absolute value of *type* is returned.
- *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*).

MESSAGE QUEUES

- *flag* value of IPC_NOWAIT can be specified for non-blocking of *msgrcv*.
- If message of specified type is not available – causes *msgrcv* to return immediately with a value of –1 and an error of ENOMSG.
- If IPC_NOWAIT not specified – operation 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).

SEMAPHORES

- Counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource, 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.

SEMAPHORES

- 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.
- 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* – 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)

SEMAPHORES

- 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
        ...
};
```

SEMAPHORES

- *semget* – obtain a semaphore ID.

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

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

- Returns: semaphore ID if OK, -1 on error
- When a 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.

SEMAPHORES

- 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.

SEMAPHORES

- *semctl* – comprises of various semaphore operations.

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

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

- Returns: value based on *cmd* for all GET commands (except GETALL), 0 for remaining commands
- Value of *semnum* is between 0 and (*nsems*-1)

SEMAPHORES

- 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
};
```

SEMAPHORES

- *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*

SEMAPHORES

- *cmd* specifies one of ten commands to be performed on the set specified by *semid.*
  - 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);
```

- Returns: 0 if OK, -1 on error.

SEMAPHORES

- *semoparray* argument – 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).

SHARED MEMORY

- Allows two or more processes to share a given region of memory.
- Fastest form of IPC – data does not need to be copied between client and server.
- Synchronization of access to a given region among multiple processes.
- Semaphores are used to synchronize shared memory access.

SHARED MEMORY

- 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
        ...
};
```

SHARED MEMORY

- *shmget* – obtain a shared memory identifier.

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

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

- 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.

SHARED MEMORY

- *shmctl* – comprises of various shared memory operations.

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

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

- 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.

SHARED MEMORY

- *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);
```

- 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))

SHARED MEMORY

- *shmdt* – detach a shared memory segment.

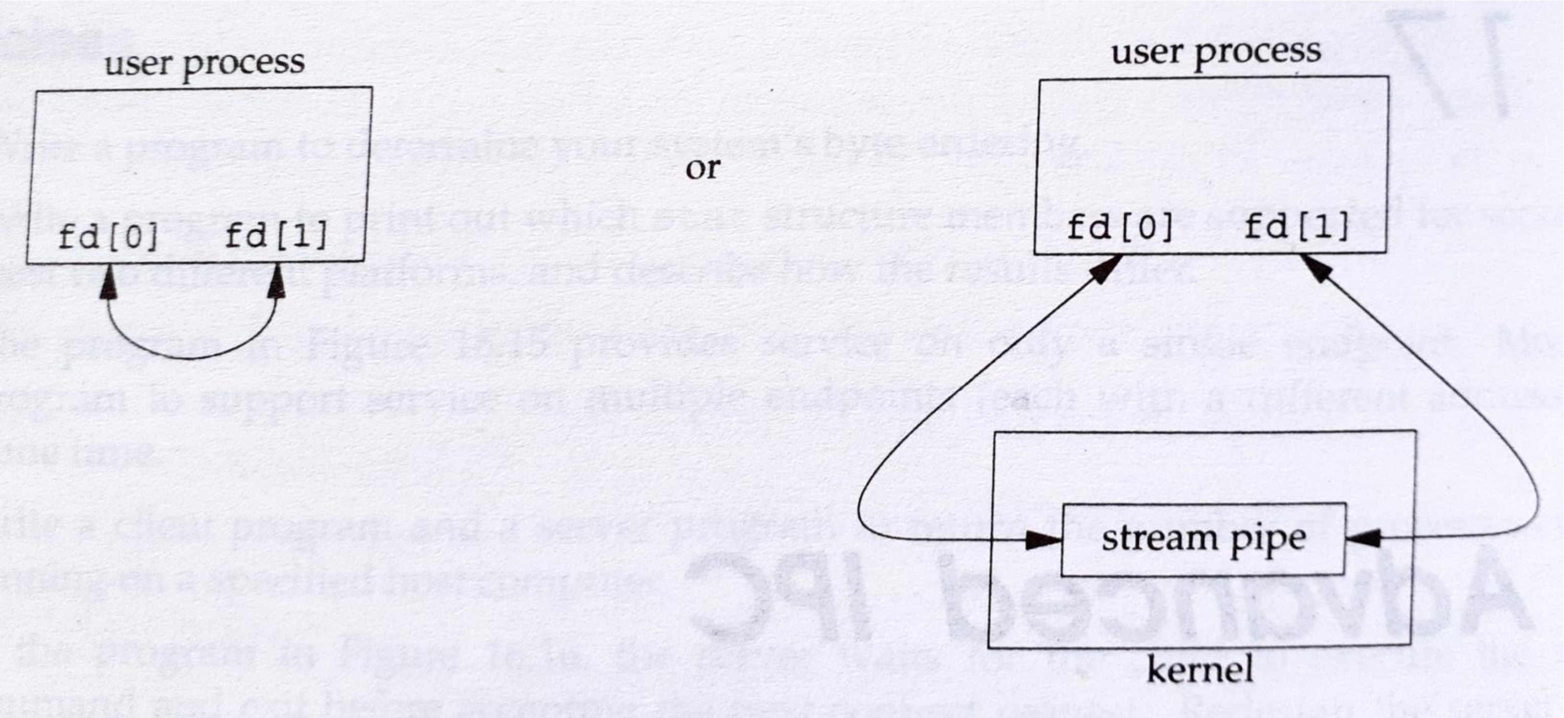
```
#include<sys/shm.h>

int shmdt(void *addr);
```

- Returns: 0 if OK, -1 on error
- 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.

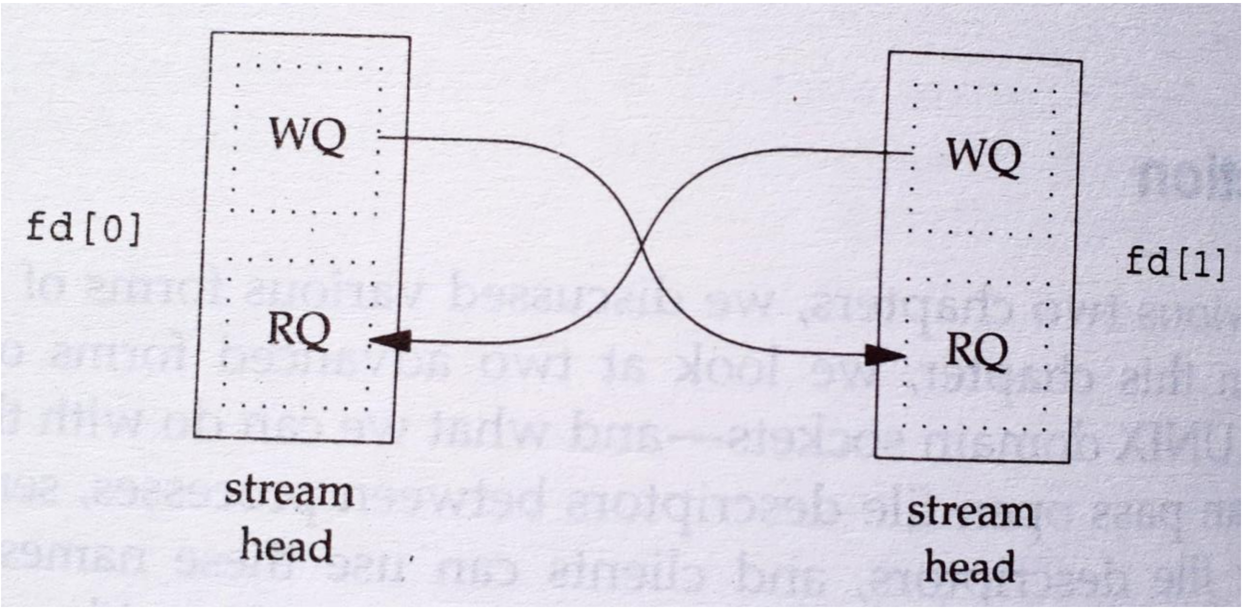
STREAMS PIPES

- Bidirectional (full-duplex) pipe that can be used for IPC between parent and child.
- Ways to view a STREAMS pipe –



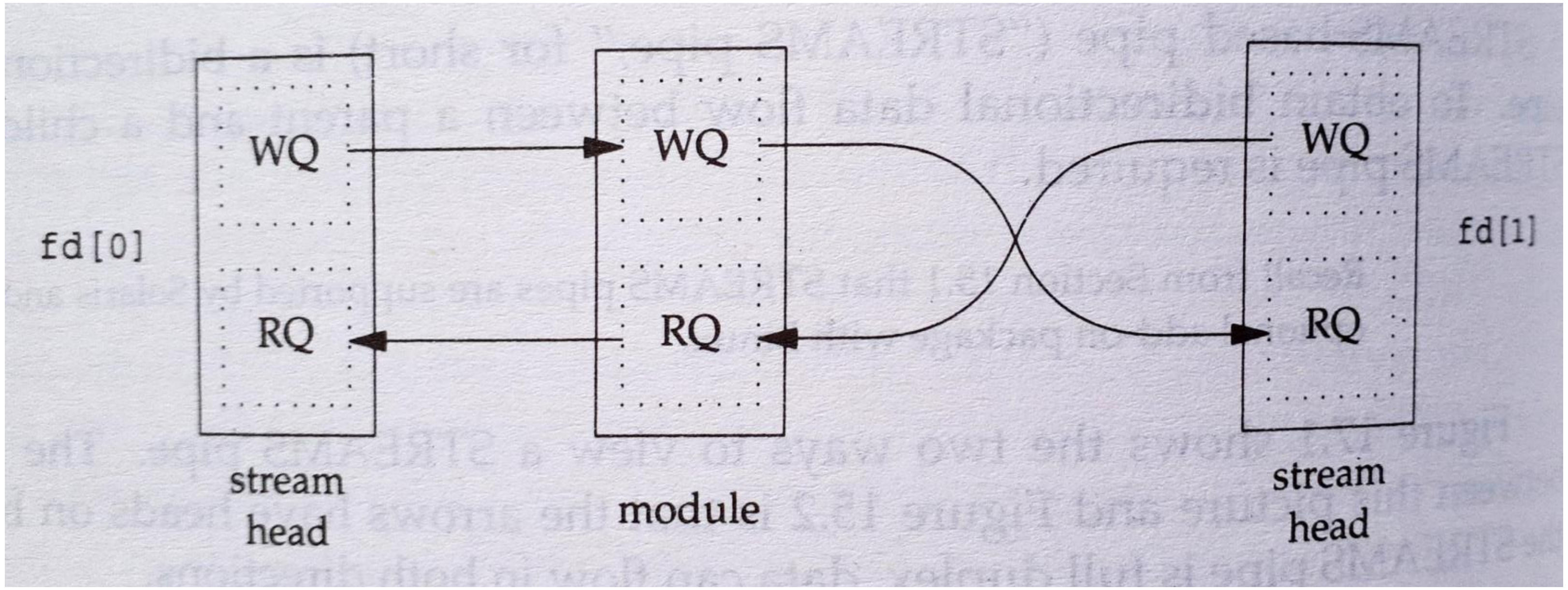
STREAMS PIPES

- Two stream heads – each write queue (WQ) pointing at other's read queue (RQ).
- Data written to one end of pipe is placed in messages on other's read queue.
- Inside a STREAMS pipe –



STREAMS PIPES

- A STREAMS module can be pushed onto either end of the pipe to process data written to the pipe.
- Module must be removed from same end on which it has pushed.
- Inside a STREAMS pipe with a module –



STREAMS PIPES

- STREAMS mechanism provides a way for processes to give a pipe a name in the file system – bypasses problem of dealing with unidirectional FIFOs.
- *fattach* - gives a name to a STREAMS pipe in the file system.

```
#include<stropts.h>

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

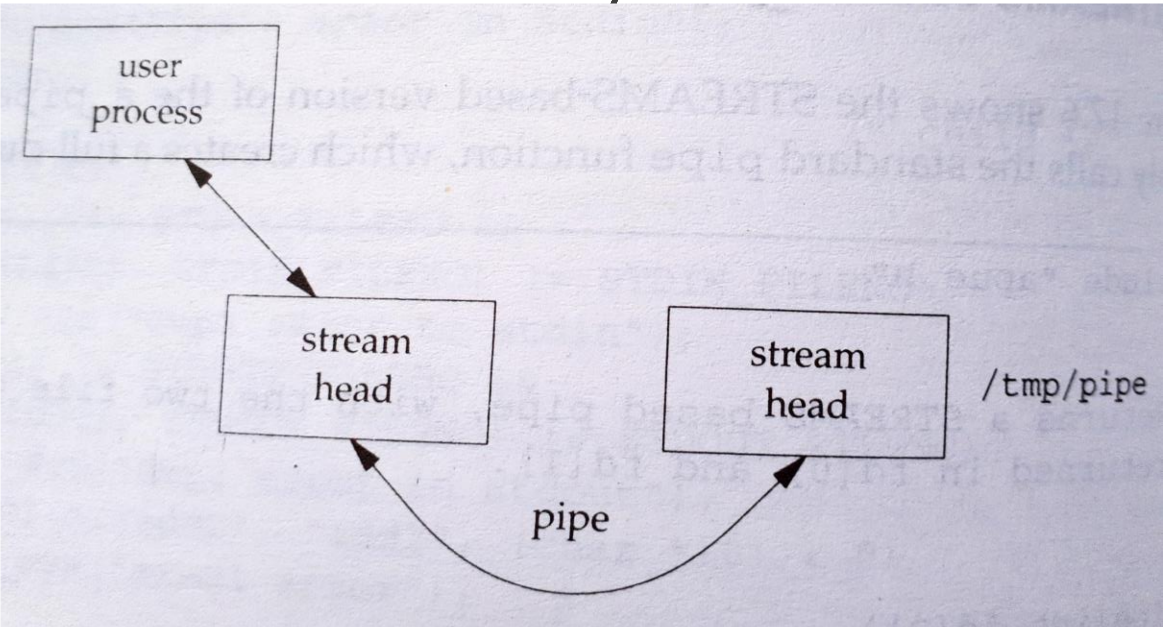
- Returns: 0 if OK, -1 on error.
- *path* - must refer to an existing file and calling process must either own the file and have write permissions to it or be running with superuser privileges.

STREAMS PIPES

- 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.

STREAMS PIPES

- Pipe mounted on a name in the file system –



- 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.

STREAMS PIPES

- *fdetach* – undo the association between STREAMS pipe and name in the file system.

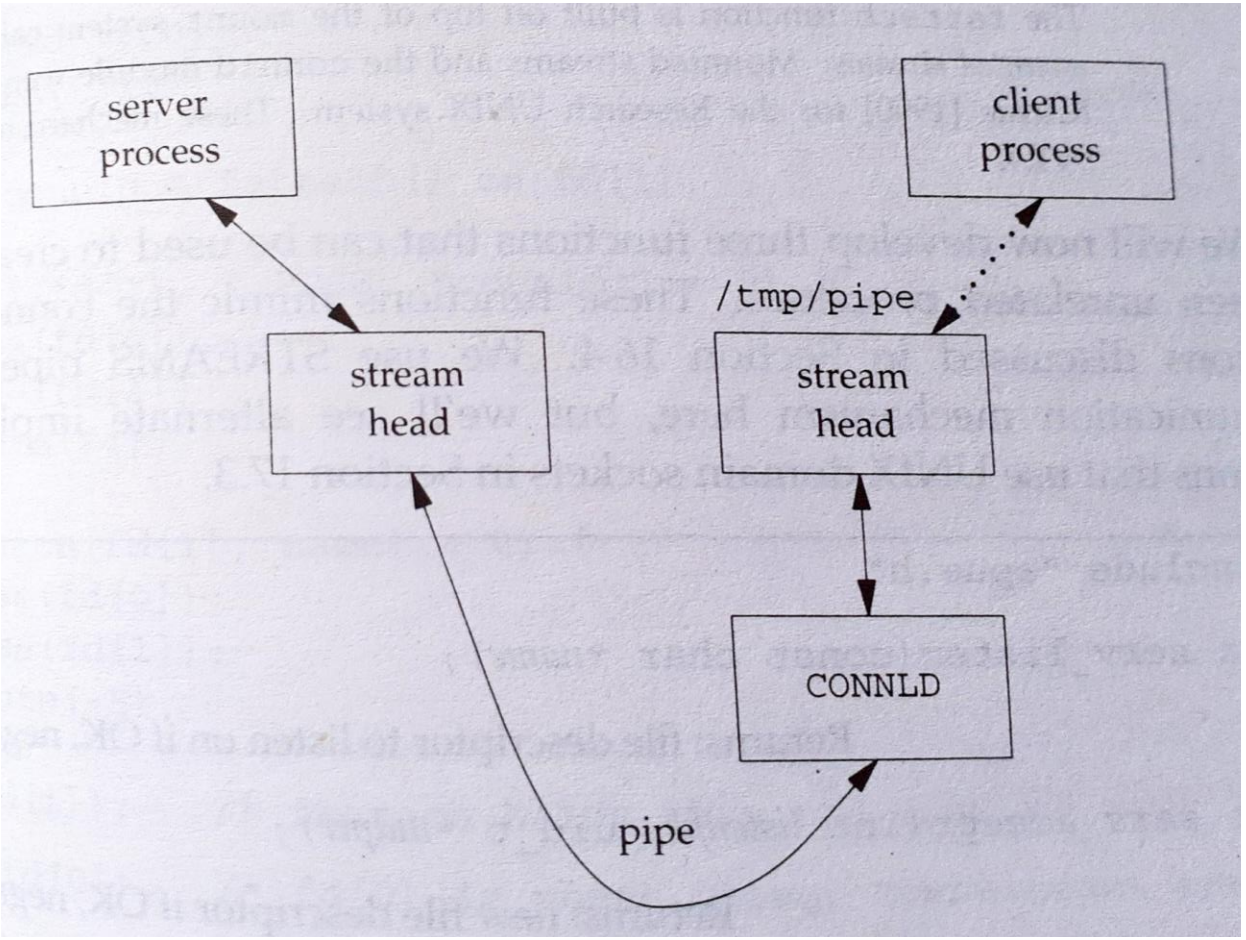
```
#include<stropts.h>

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

- 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.

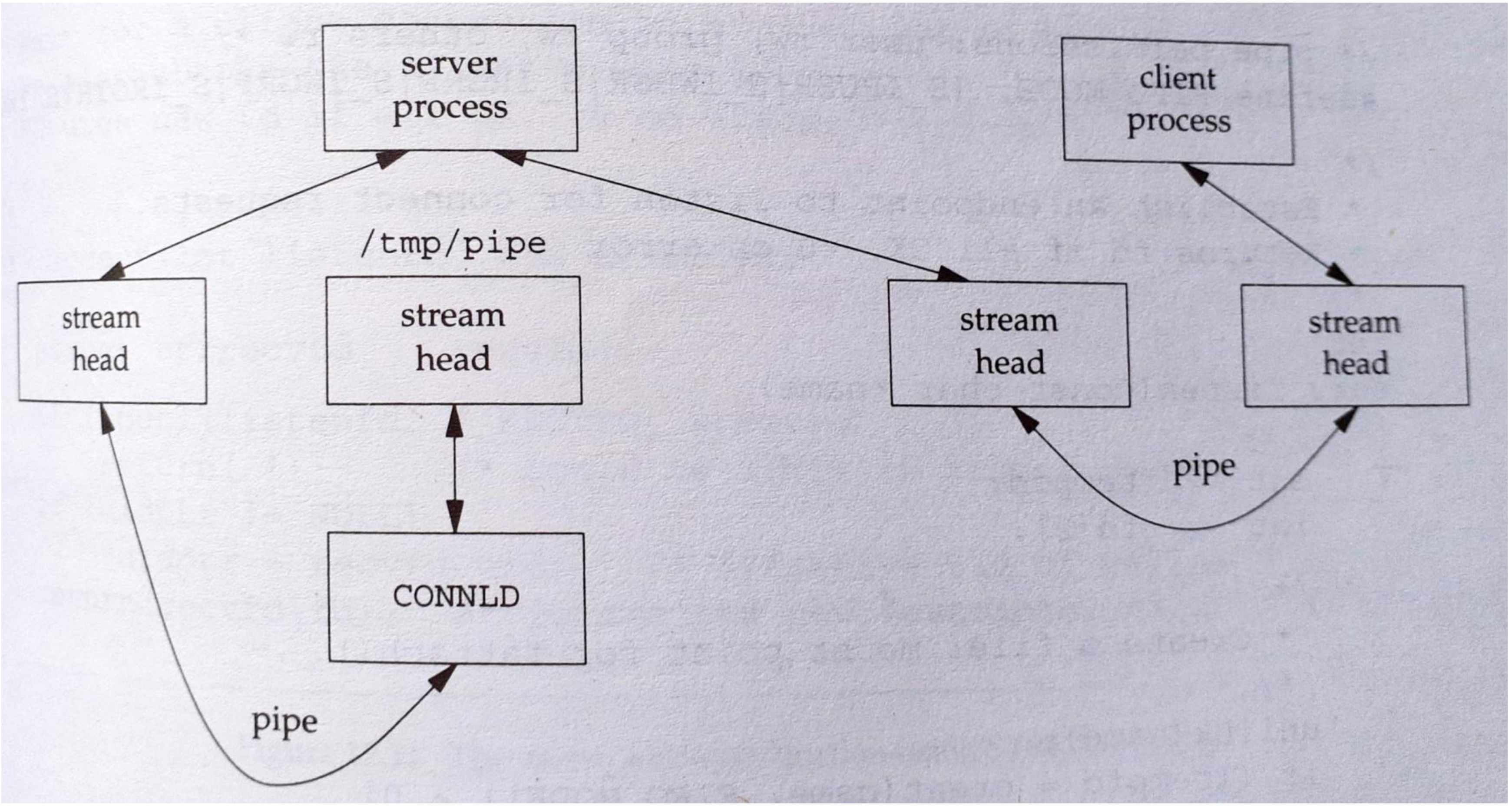
STREAMS PIPES

- Setting up *connld* for unique connections –



STREAMS PIPES

- Using *connld* to make unique connections –



STREAMS PIPES

- Functions used to create unique connections between unrelated processes -

```c
#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.
- *serv_accept* : used by server to wait for client's connect request to arrive.
- *cli_conn* : called by client to connect to the server.

PASSING FILE DESCRIPTORS

- 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.
- Passing a pointer to an open file table entry from one process to another.
- Pointer is assigned the first available descriptor in the receiving process.

PASSING FILE DESCRIPTORS

- Functions used to send and receive file descriptors -

```
#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.
- *send_err* : used by process to send error message to another process.
- *recv_fd* : called by client to receive a descriptor.

THANK YOU