Advanced UNIX System Programming

2

Agenda

1 Daemon Process

2 System V IPC

3 Socket Programming

Daemon Process

Introduction

- Daemon process starts during system startup
- They frequently spawn other process to handle services requests
    - Mostly started by initialization script /etc/rc
- Waits for an event to occur
- perform some specified task on periodic basis (cron job)
- perform the requested service and wait
    - Example print server

Characteristics

- executed at the background process
- Orphan process
- No controlling terminal
- run with super user privileges
- process group leaders
- session leaders

Daemon -program

```
int  init_daemon ( void ) {
     if ( ! fork ( ) ) {
             setsid ( ) ;
             chdir ( " / " ) ;
             umask ( 0 ) ;
       /* Specify Your Job  */
             return ( 0 );
     }
     else
             exit ( 0 );
}
```

System V IPC

Introduction

- Pipe and FIFO do not satisfy many requirements of many applications.
- Sys V IPC is implemented as a single unit
- System V IPC Provides three mechanisms namely
  - Message Queues,
  - Shared Memory
  - Semaphores.
- Persist till explicitly delete or reboot the system

Common Attributes

- Each IPC objects has the following attributes.
  - key
  - id
  - Owner
  - Permission
  - Size
    - Message queue – used-bytes, number of messages
    - Shared memory – size, number of attach, status
    - Semaphore – number of semaphores in a set
- The ipc_perm structure holds the common attributes of the resources.

System Limitations

```
$ ipcs -l
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767

------ Messages: Limits --------
max queues system wide = 16
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

Get a key

- If we wish to communicate between different processes using an IPC resource, the first step is to create a shared unique identifier.
- The system generates a number dynamically for a given mechanism by using the *ftok* library function.
- But apart from the creator, other processes that want to communicate with the creator process should agree to the key value.
- Syntax: key_t ftok (const char *filename, int id);

Get an id

- The syntax for a *get* function is:

int xxxget (key_t key, int xxxflg);

(xxx may be msg or shm or sem)

- If successful, returns to an identifier; otherwise -1 for error.

- The key can be generated in three different ways
  - from the *ftok* library function
  - by choosing some static positive integer value
  - by using the IPC_PRIVATE macro
- flags commonly used with this function are IPC_CREAT and IPC_EXCL.

Control a object

- The syntax for the *control* function is:

int xxxctl (int xxxid, int cmd, struct xxxid_ds *buffer);

(xxx may be msg or shm or sem);

- If successful, the *xxxctl* function returns zero, otherwise it returns -1.
- The command argument may be
  - IPC_STAT
  - IPC_SET
  - IPC_RMID

Message Queues - Introduction

- Message queue overcomes FIFO limitation like storing data and setting message boundaries.
- Create a message queue
- Send message (s) to the queue
- Any process who has permission to access the queue can retrieve message (s).
- remove the message queue.

Message queues

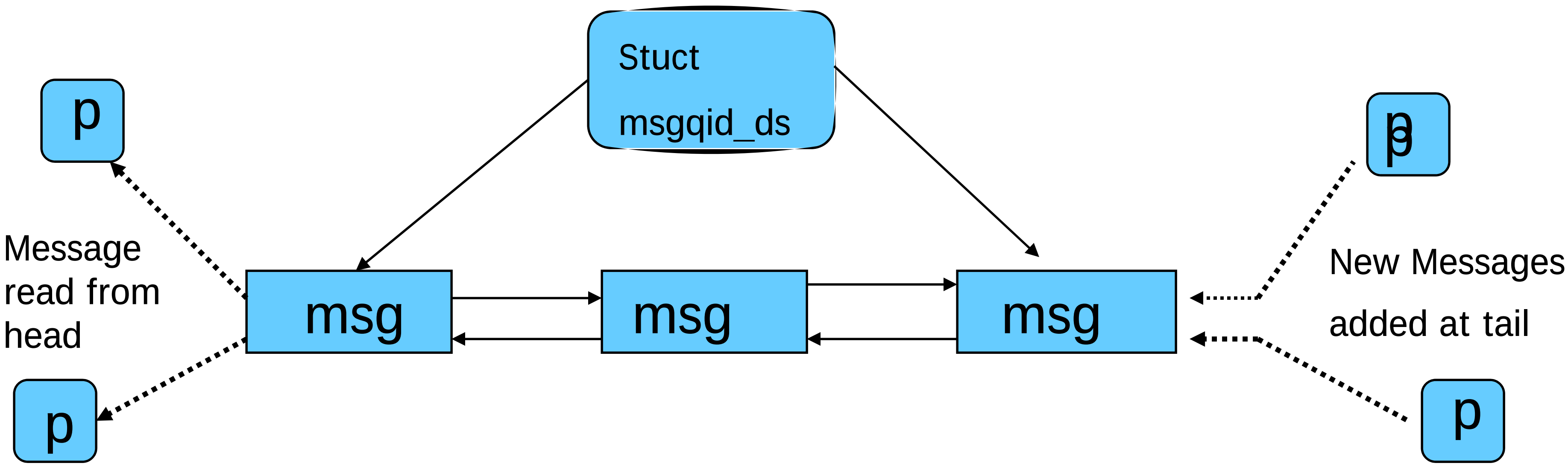
```
struct msgbuf {
        long mtype;
        char mtext [1];
}; Standard structure
```

```
struct My_msgQ {
        long mtype;
        char mtext [1024];
        void *   xyz;
}; Our own structure
```

msgid xxx

mtype x_1	msg text
mtype x_2	msg text
mtype x_3	msg text
mtype x_4	msg text
mtype x_5	msg text
---------	
mtype x_n	msg text

Messages in a queue



msqid_ds

```
struct msqid_ds
{
  struct ipc_perm msg_perm;
  __time_t msg_stime;
     __time_t msg_rtime;
  __time_t msg_ctime;
  unsigned long int __msg_cbytes;
  msgqnum_t msg_qnum;
  msglen_t msg_qbytes;
  __pid_t msg_lspid;
  __pid_t msg_lrpid;
};
```

msgget

- **int msgget (key_t key, int msgflg);**
- The first argument key can be passed from the return value of the ftok function or made IPC_PRIVATE.
- To create a message queue, IPC_CREAT ORed with access permission is set for the msgflg argument.
- Ex: msgid = msgget (key, IPC_CREAT | 0744);
  msgid = msgget (key, 0);

msgsnd

- The syntax of the function is:

int msgsnd (int msqid, structu msgbuf *msgp, size_t msgsz, int msgflg);

- Arguments:
  - message queue ID
  - address of the structure.
  - size of the message text
  - message flag
    - 0 or IPC_NOWAIT

msgrcv

- syntax of the function is:

ssize_t msgrcv (int msqid, struct msgbuf *msgp, size_t msgsz, long msgtype, int msgflg);

- msgtype argument is used to retrieve a particular message.
  - 0 -retrieve in FIFO order
  - +ve - retrieve the the exact value of the message type
  - –ve - first message or <= to the absolute value.

- on success, msgrcv returns with the number of bytes actually copied into the message text

Destroying a message queue

- There are many ways:
- From command line, using one of the ways
  - $ ipcrm msg msqid
  - $ ipcrm –q msqid
  - $ ipcrm –Q msgkey
- Using system call
  - msgctl (msgid, IPC_RMID, 0);

Message queue : pseudo code

- key = ftok (“.“, ‘a‘);
- msqid = msgget (key, IPC_CREAT|0666);
- msgsnd (msqid, &struct, sizeof (struct), 0);

- msgrcv (msqid, &struct, sizeof (struct), mtype, 0);
- msgctl (msqid, IPC_RMID, NULL);
- $ipcrm msg msqid

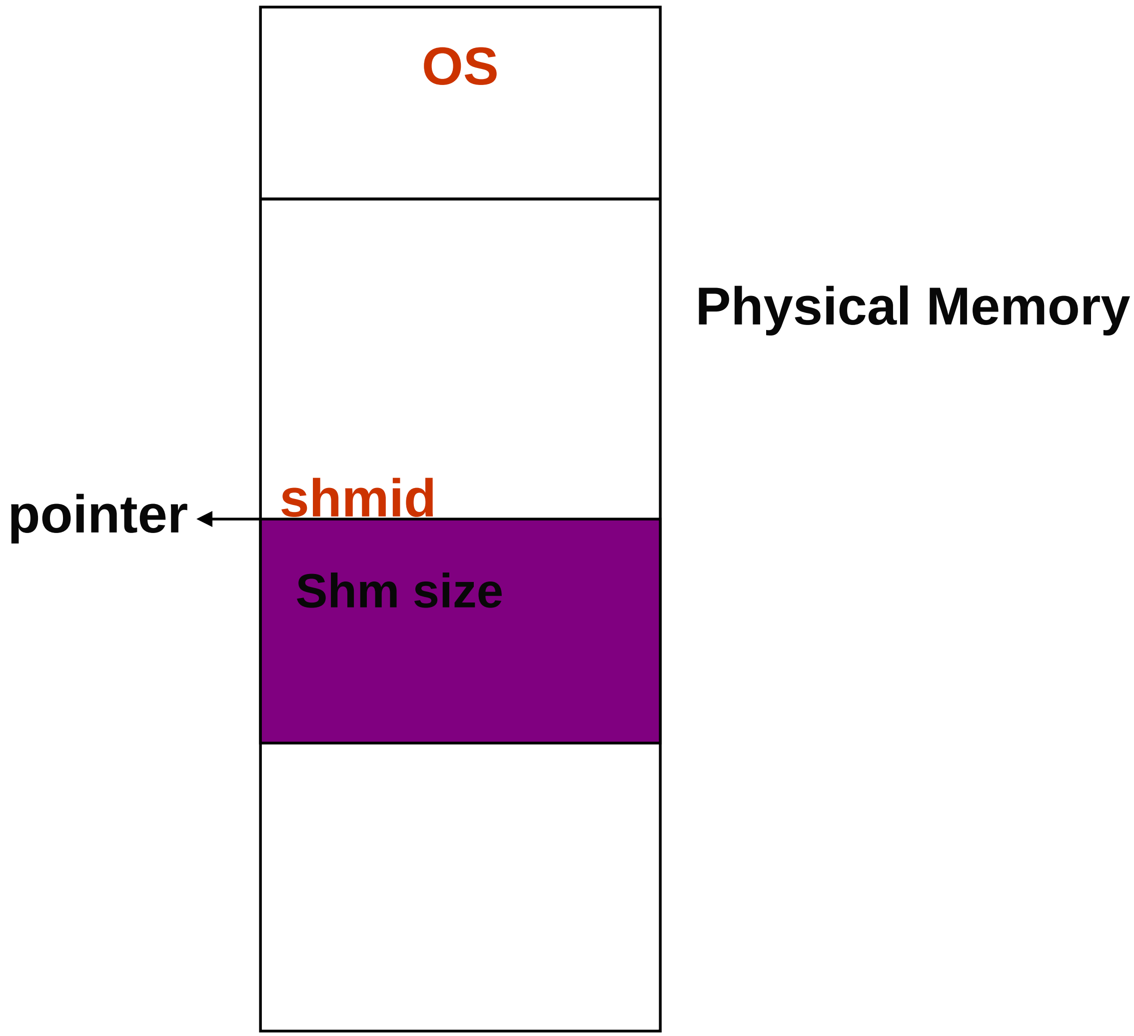
Limitations

- Message queues are effective if a small amount of data is transferred.
- Very expensive for large transfers.
- During message sending and receiving, the message is copied from user buffer into kernel buffer and vice versa
- So each message transfer involves two data copy operations, which results in poor performance of a system.
- A message in a queue can not be reused

Shared Memory - Introduction

- Very flexible and ease of use.
- Fastest IPC mechanisms
- shared memory is used to provide access to
  - Global variable
  - Shared libraries
  - Word processors
  - Multi-player gaming environment
  - Http daemons
  - Other programs written in languages like Perl, C etc.,

Shared Memory – shmat ( )



Why go for Shared Memory

- Shared memory is a much faster method of communication than either semaphores or message queues.
- does not require an intermediate kernel buffer
- Using shared memory is quite easy. After a shared memory segment is set up, it is manipulated exactly like any other memory area.

Shared memory - Data structures

- The data structures used in shared memory are:
  - shmid_ds
  - ipc_perm
  - shminfo
  - shm_info
  - shmid_kernel

ipc_perm structure

```
struct ipc_perm
 {
  __key_t __key;                  - Key
  __uid_t uid;                   - Owner's user ID
  __gid_t gid;                  - Owner's group ID
  __uid_t cuid;                - Creator's user ID
  __gid_t cgid;               - Creator's group ID
  unsigned short int mode;    - r/w permission unsigned
   short int __seq;  - Sequence number
};
```

shmid_ds

```
struct shmid_ds
{
  struct ipc_perm shm_perm;
  size_t shm_segsz;
  __time_t shm_atime;
  __time_t shm_dtime;
  __time_t shm_ctime;
  __pid_t shm_cpid;
  __pid_t shm_lpid;
  shmatt_t shm_nattch;
};
```

Steps to access Shared Memory

- The steps involved are
  - Creating shared memory
  - Connecting to the memory & obtaining a pointer to the memory
  - Reading/Writing & changing access mode to the memory
  - Detaching from memory
  - Deleting the shared segment

shmat

- used to attach the created shared memory segment onto a process address space.
- **`void *shmat(int shmid,void *shmaddr,int shmflg)`**
- Example**: `data=shmat(shmid,(void *)0,0);`**
- A pointer is returned on the successful execution of the system call and the process can read or write to the segment using the pointer.

Reading/ writing to SM

- Reading or writing to a shared memory is the easiest part.
- The data is written on to the shared memory as we do it with normal memory using the pointers
- Eg. Read:
- printf(“SHM contents : %s \n”, data);
- Write:
- prinf(“”Enter a String : ”);
- scanf(“ %[^\n]”,data);

Shmdt & shmctl

- The detachment of an attached shared memory segment is done by shmdt to pass the address of the pointer as an argument.
- Syntax: int shmdt(void *shmaddr);
- To remove shared memory call:

int shmctl(shmid,IPC_RMID,NULL);

- These functions return –1 on error and 0 on successful execution.

Shared Memory – pseudo code

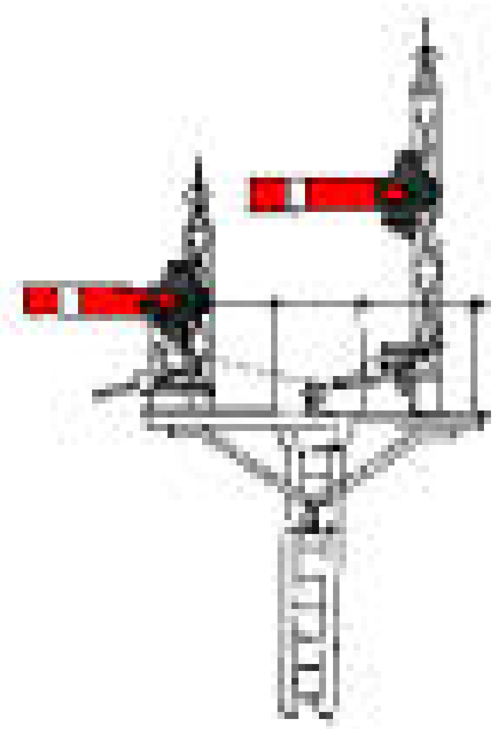
- shmid = shmget (key, 1024, IPC_CREAT|0744);
- void *shmat (int shmid, void *shmaddr, int shmflg);
  if the shm is read only pass SHM_RDONLY else 0
- (void *)data = shmat (shmid, (void *)0, 0);
- int shmdt (void *shmaddr);
- int shmctl (shmid, IPC_RMID, NULL);

Limitations

- Data can either be read or written only. Append is not allowed
- Race condition
  - Since many processes can access the shared memory, any modification done by one process in the address space is visible to all other processes. Since the address space is a shared resource, the developer should implement a proper locking mechanism to prevent the race condition in the shared memory.

Semaphores

Synchronization Tool
An Integer Number
P ( ) And V ( ) Operators
Avoid Busy Waiting
Types of Semaphore



Used in :
shared memory segment
message queue
file

Semaphores (Contd.).

- If a process wants to use the shared object, it will “lock” it by asking the semaphore to decrement the counter
- Depending upon the current value of the counter, the semaphore will either be able to carry out this operation, or will have to wait until the operation becomes possible
- The current value of counter is >0, the decrement operation will be possible. Otherwise, the process will have to wait

System V IPC – Semaphores

- System V semaphore provides a semaphore set - that can include a number of semaphores. It is up to user to decide the number of semaphores in the set
- Each semaphore in the set can be a binary or a counting semaphore. Each semaphore can be used to control access to one resource - by changing the value of semaphore count

Semaphore - Initialization

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

union semun arg;

semid = semget (key, 1, IPC_CREAT | 0644);
arg.val = 1; /* 1 for binary else > 1 for Counting Semaphore */
semctl (semid, 0, SETVAL, arg);
```

Semaphore - Implementation

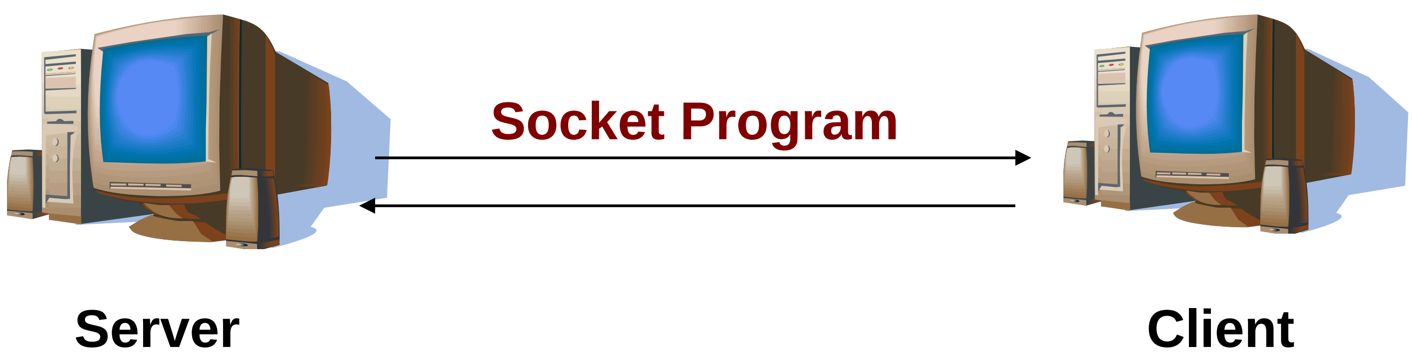
```
struct sembuf {
    short sem_num;  /* semaphore number: 0 means first */
    short sem_op;   /* semaphore operation: lock or unlock */
    short sem_flg;  /* operation flags : 0, SEM_UNDO, IPC_NOWAIT */
  };
struct sembuf buf = {0, -1, 0};  /* (-1 + previous value) */
semid = semget (key, 1, 0);

semop (semid, &buf, 1); /* locked */
-----Critical section--------
buf.sem_op = 1;
semop (semid, &buf, 1); /* unlocked */
```

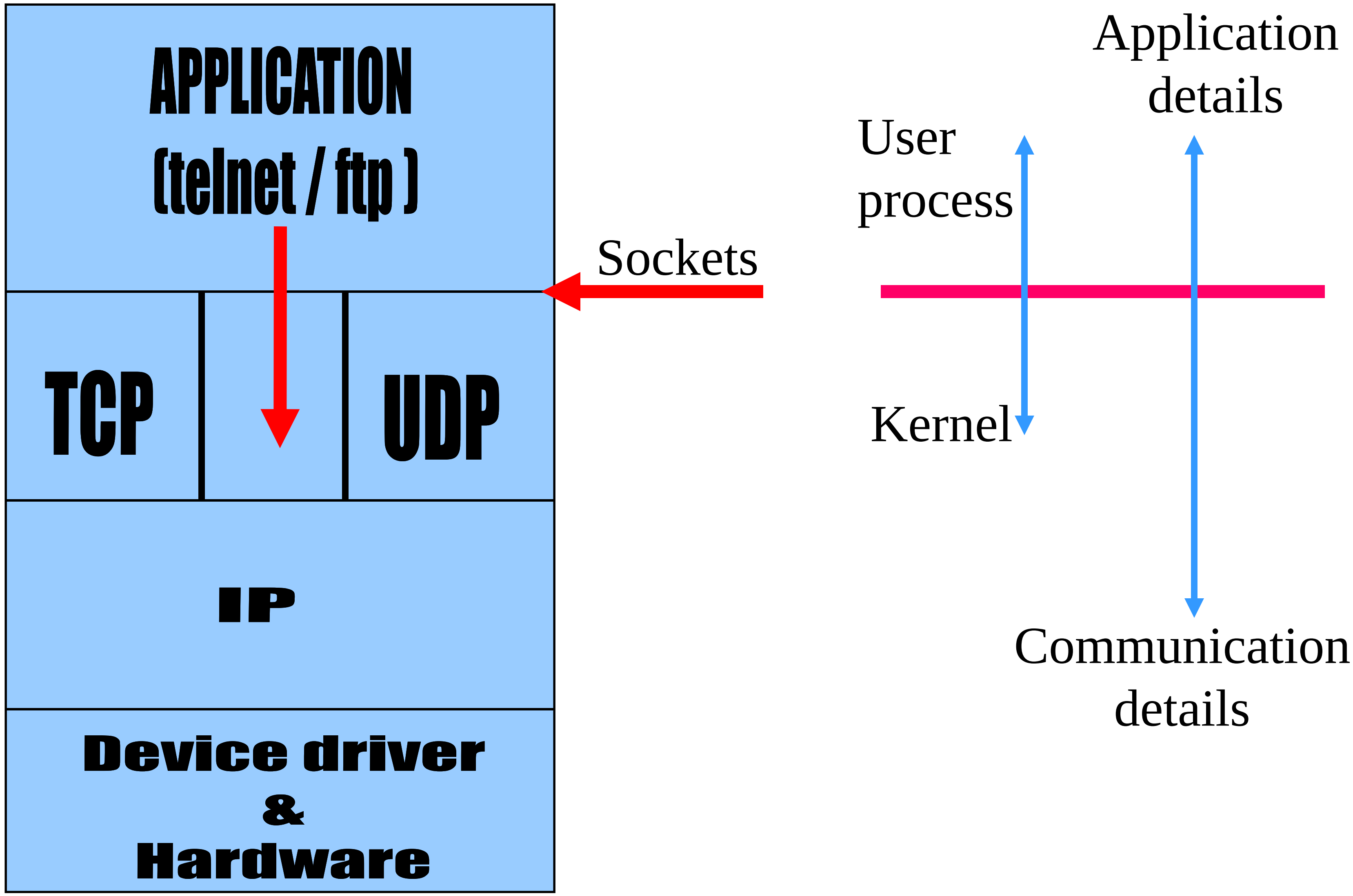
Socket Programming

Socket

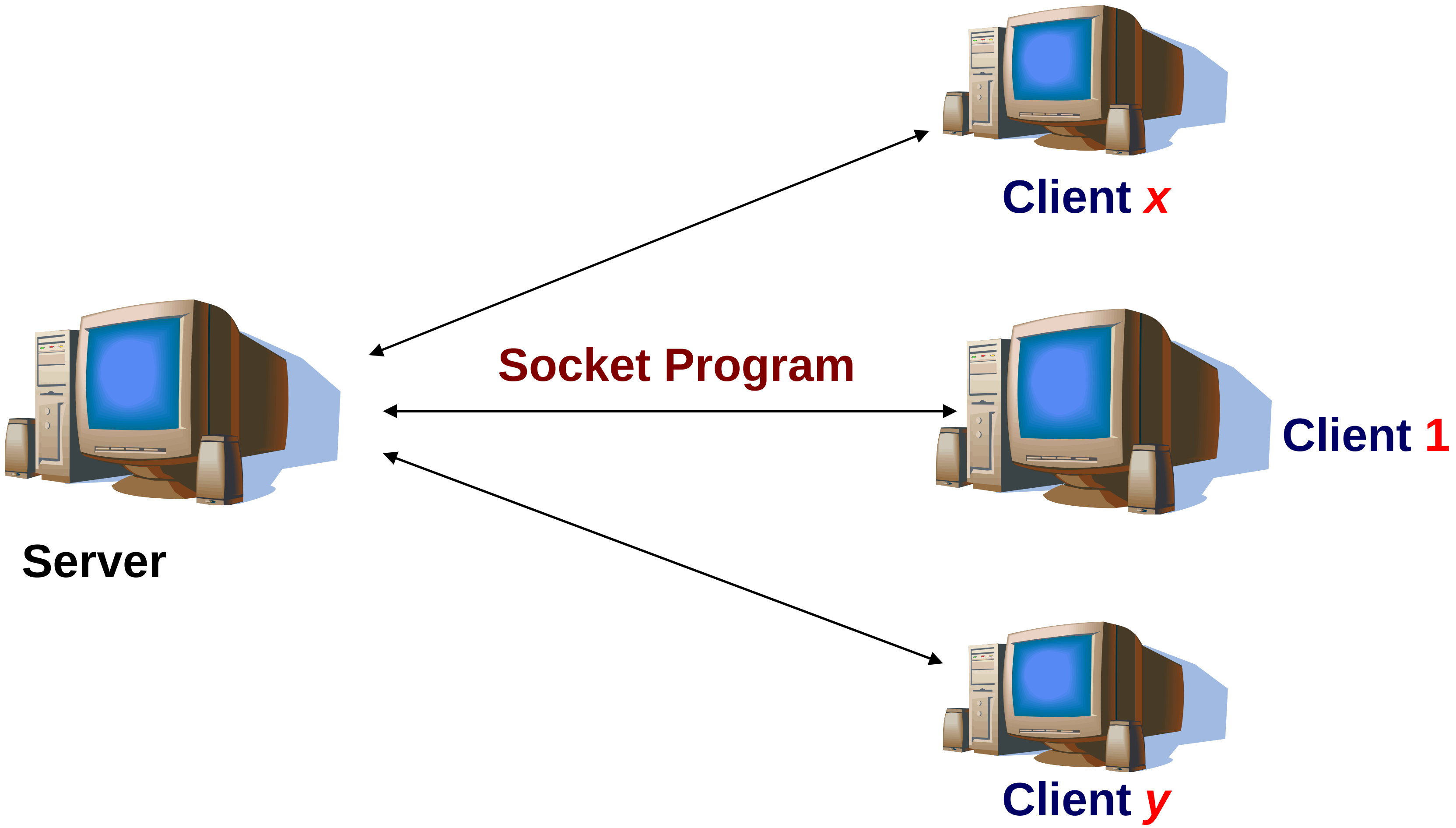
A socket is used to communicate between different machines (different IP addresses). Socket of type SOCK_STREAM is full-duplex byte streams.



TCP/ IP Protocol Stack



Concurrent Server



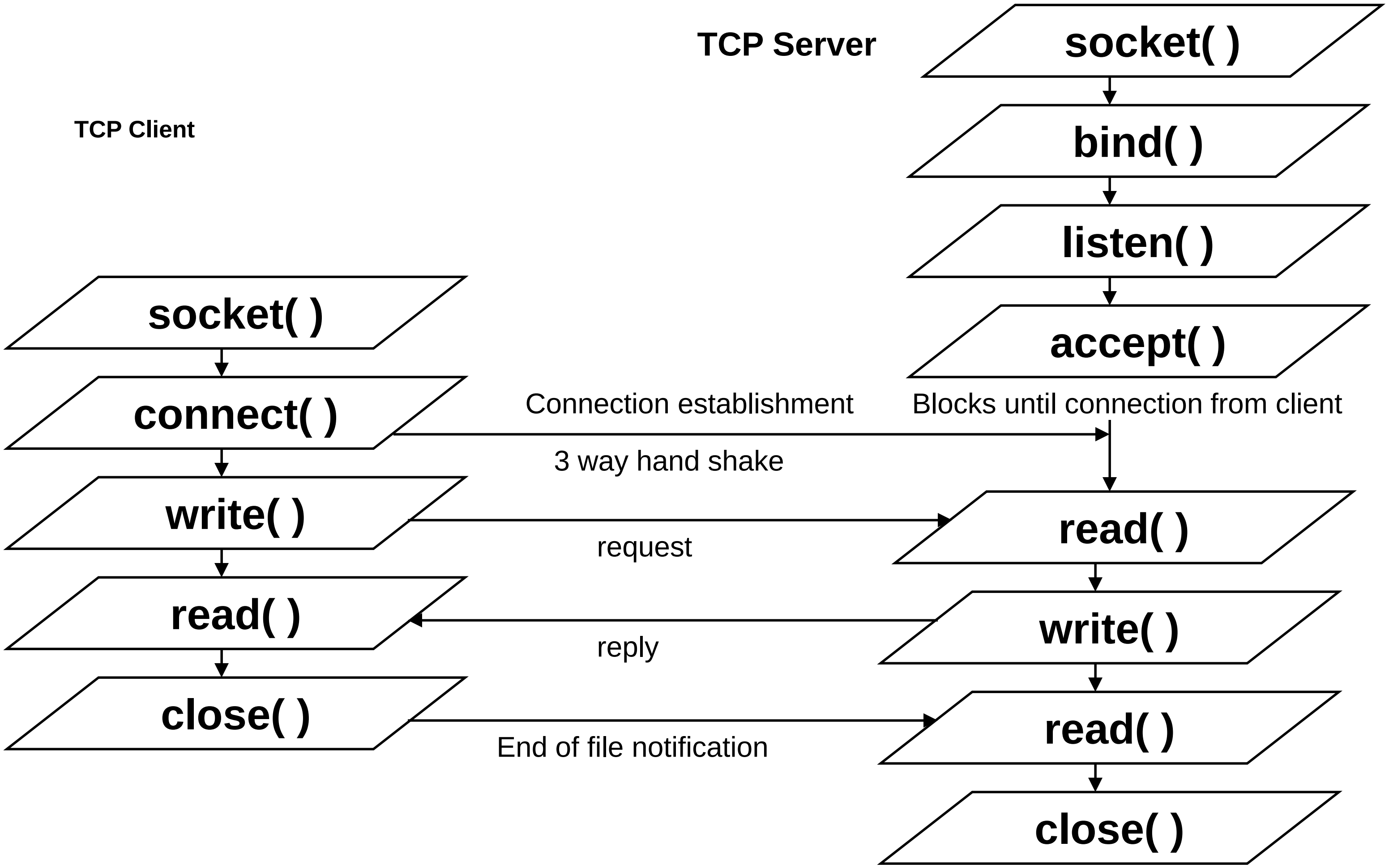
Sockets (Contd.).

- A socket is a communication endpoint and represents abstract object that a process may use to send or receive messages.
- The socket frame work is procedural not message based
- The two most prevalent communication APIs for Unix Systems are Berkeley Sockets and System V Transport Layer Interface(TLI)

Sockets (Contd.).

- The typical client -server relationship is not symmetrical
- Network Connection can be connection-oriented or connectionless
- More parameters must be specified for network connection, than for file I/O
- The unix I/O system is stream oriented
- The network interface should support multiple communication protocol

Socket Functions



socket ()

- int socket (int domain, int type,
  int protocol);
- Domain
  – AF_UNIX for UNIX domain
  – AF_INET for Internet domain
- Socket type
  – SOCK_STREAM for TCP (Connection Oriented)
  – SOCK_DGRAM for UDP (Connectionless)
- Protocol
  – Protocol number is used to identify an application. List of the protocol number and the corresponding applications can be seen at /etc/protocols.
- The socket system call returns a socket descriptor on success and -1 for failure.

sockaddr_in structure

- struct sockaddr_in {
  short int sin_family;
  unsigned short int sin_port;
          struct in_addr sin_addr;
  }
- sin_family - address family
- sin_port   - port number
- sin_addr   - internet address (IP addr)

- The in_addr structure used to define sin_addr is as under

struct in_addr {
    unsigned long s_addr; /* refers to the four byte IP address */
}

Byte ordering ex: 91,329 hex: 00 01 64 C1

Little endian byte order : **Ex: Intel series**

High order byte	low order byte

Address A+1 00 01 64 C1 Address A

Increasing memory address

Big endian byte order : **ex: IBM 370, Motorola**

High order byte	low order byte

Address A+1 C1 64 01 00 Address A

Increasing memory address

Byte ordering functions

- Internet protocols use big endian byte ordering called network byte order
- Functions below allow conversions between the formats.

```
#include <netinet/in.h>
htons() – "Host to Network Short"
htonl() – "Host to Network Long"
ntohs() – "Network to Host Short"
ntohl() – "Network to Host Long"
```

- h stands for host n stands for network
- s stands for short l stands for long

bind ( )

- int bind(int sockfd, struct sockaddr *my_addr,int addrlen);
- sockfd - the socket file descriptor returned by socket().
- my_addr - a pointer to a struct sockaddr that contains information about IP address and port number.
- addrlen - set to sizeof (struct sockaddr)

connect( )

- **`int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);`**
- sockfd - the socket file descriptor returned by socket().
- serv_addr - is a struct sockaddr containing the destination port and IP address.
- addrlen - set to sizeof (struct sockaddr).

listen( )

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

- sockfd - the socket file descriptor returned by socket().
- backlog - the number of connections allowed on the incoming queue.
- Backlog should never be zero as servers always expect connection from client.
- The listen function converts an unconnected socket into a passive socket,
- On successful execution of listen is indicating that the kernel should accept incoming connection requests directed to this socket.

accept( )

- **int accept (int sockfd, void *addr, int *addrlen);**
- sockfd
  - the socket file descriptor returned by socket().
- addr
  - a pointer to a struct sockaddr_in. The information about the incoming connection like IP address and port number are stored.
- addrlen
  - a local integer variable that should be set to sizeof (struct sockaddr_in) before its address is passed to accept().

close ( )

- Socket descriptor can be closed like file descriptor.
- **`close (sockfd);`**
- Close system call prevents any more reads and writes to the socket. For attempting to read or write the socket on the remote end will receive an error.

shutdown ( )

- **`int shutdown (int sockfd, int how);`**

- sockfd - socket file descriptor of the socket to be shutdown.
- how – if it is
  - 0 - Further receives are disallowed
  - 1 - Further sends are disallowed
  - 2 - Further sends and receives are disallowed.

- The shutdown system call gives more control (than close (sockfd) over how the socket descriptor can be closed.

pseudo code

SERVER

```
struct sockaddr_in serv, cli;

sd = socket (AF_INET, SOCK_STREAM, 0);

serv.sin_family = AF_INET;
serv.sin_addr.s_addr = INADDR_ANY;
serv.sin_port = htons (portno);

bind (sd, &serv, sizeof (serv));
listen (sd, 5);

nsd = accept (sd, &cli, &sizeof (cli));
read / write (nsd, ….);
```

CLIENT

```
struct sockaddr_in serv;

sd = socket (AF_INET, SOCK_STREM, 0);

serv.sin_family = AF_INET;
serv.sin_addr.s_addr = inet_addr (“ser ip”);
serv.sin_port = htons (portno);

connect (sd, &server, sizeof  (server));

read / write (sd, ….);
```

Iterative Server

- One client request at a time.

```
while (1) {
   nsd = accept (sd, &cli,…);
   read/write(nsd, …);
   }
```

Concurrent Server

- Many clients requests can be serviced concurrently

```
while (1) {
  nsd =(accept (sd, &cli, ….);
   if (!fork( )) {
          close(sd);
          read/write(nsd, …..);
          exit();
   } else
        close(nsd);
}
```

Bibilography

1. Stevens. W R. Unix Network Programming Volume I & II
2. Chan, Terrance. Unix System Programming Using C++
3. Bach, Maurice J. Design of Unix

References :

http://linuxdocs.org/HOWTOs/DB2-HOWTO/kernel24.html

Thank You