Politecnico di Milano

Inter-process Communication

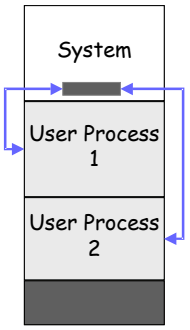
Lecturer:
William Fornaciari
Politecnico di Milano
fornacia@elet.polimi.it
www.elet.polimi.it/~fornacia

Inter-Process Communication

- Shared memory
  - Language run-time like a small OS
  - Multi-threading available
- Inter-process communication
  - Processes in separate address spaces
  - Between different computers (network)
  - Flexibility in location of processes

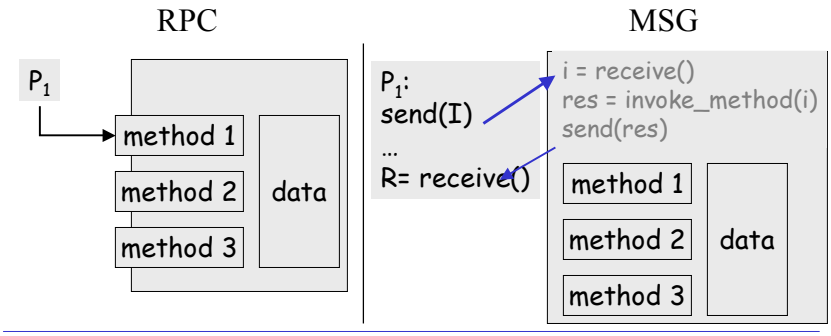
Inter-Process Communication

- Inter-Process communication
  - Message passing
  - Signals
  - Pipes and named pipes
  - Tracing and debugging
  - *Sockets*
  - *Remote procedure calls*



Duality

- Duality of two models: whatever we can do in one, we can do it in the other.

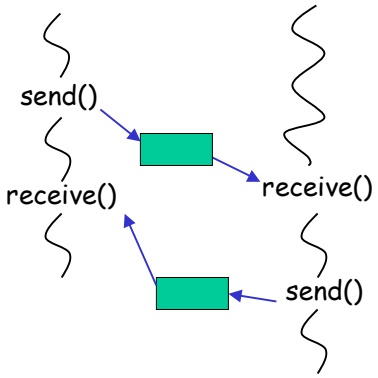


Message Passing

- Is a general method used for inter-process communication (IPC)
  - For processes inside the same computer
  - For processes in a distributed system
- Yet another mean to provide process synchronization and mutual exclusion
- We have at least two primitives:
  - Send(destination, message)
  - Received(source, message)
- In both cases, the process may or may not be blocked

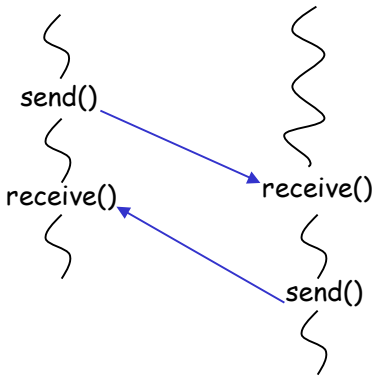
Asynchronous Communication

- System buffers
- Memory allocation
- Minimal application synchronisation



Synchronous Communication

- Processes synchronize
- No resource allocation
- Strict constraints on application



Inter-Process Communication - 7 - © 2002 William Fornaciari

Synchronization in Message Passing

- For the sender: it is more natural not to be blocked after issuing send(.,.)
  - can send several messages to multiple dest.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails)
- For the receiver: it is more natural to be blocked after issuing receive(.,.)
  - the receiver usually needs the info before proceeding
  - but could be blocked indefinitely if sender process fails before send(.,.)

Inter-Process Communication - 8 - © 2002 William Fornaciari

Synchronization in Message Passing

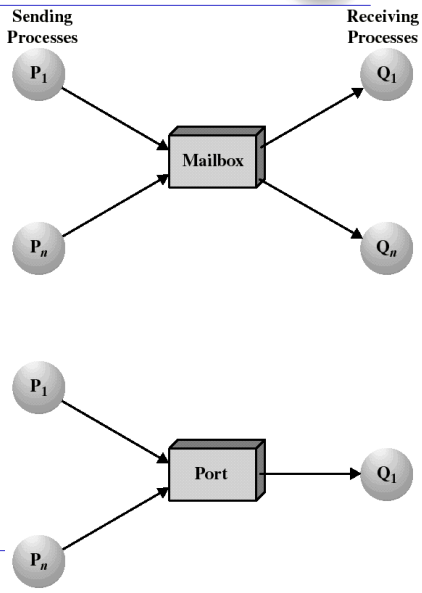
- Hence other possibilities are sometimes offered
- Ex: blocking send, blocking receive:
  - both are blocked until the message is received
  - occurs when the communication link is unbuffered (no message queue)
  - provides tight synchronization (*rendez-vous*)

Addressing in Message Passing

- Direct addressing:
  - When a specific process identifier is used for source/destination
  - But it might be impossible to specify the source ahead of time (ex: a print server)
- Indirect addressing (more convenient):
  - Messages are sent to a shared mailbox which consists of a queue of messages
  - Senders place messages in the mailbox, receivers pick them up

Mailboxes and Ports

- A mailbox can be private to one sender/receiver pair
- The same mailbox can be shared among several senders and receivers
  - the OS may then allow the use of message types (for selection)
- Port: is a mailbox associated with one receiver and multiple senders
  - used for client/server applications: the receiver is the server



Inter-Process Communication - 11 -

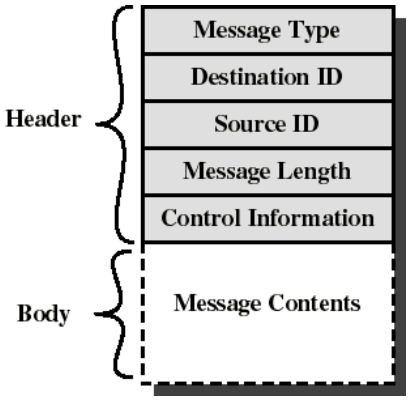
Ownership of Ports and Mailboxes

- A port is usually own and created by the receiving process
- The port is destroyed when the receiver terminates
- The OS creates a mailbox on behalf of a process (which becomes the owner)
- The mailbox is destroyed at the owner's request or when the owner terminates

Inter-Process Communication - 12 - © 2002 William Fornaciari

Message format

- Consists of header and body of message
- In Unix: no ID, only message type
- control info:
  - what to do if run out of buffer space
  - sequence numbers
  - priority...
- **Queuing discipline: usually FIFO but can also include priorities**



Enforcing Mutual Exclusion With Message Passing

- create a mailbox ***mutex*** shared by *n* processes
- `send()` is non blocking
- `receive()` blocks when ***mutex*** is empty
- Initialization: `send`(***mutex***, "go");
- The first Pi who executes `receive()` will enter CS. Others will be blocked until Pi resends msg.

```
msg: message;
for(;;)
{
   receive(mutex,msg);
   CS
   send(mutex,msg);
   RS
}
```


The bounded-buffer P/C problem with message passing

- The producer place items (inside messages) in the mailbox ***mayconsume***
- ***mayconsume*** acts as our buffer: consumer can consume item when at least one message is present
- Mailbox ***mayproduce*** is filled initially with k null messages (k= buffer size)
- The size of ***mayproduce*** shrinks with each production and grows with each consumption
- can support multiple producers/consumers

The bounded-buffer P/C problem with message passing

```
/* Producer: */
pmsg message;
for(;;)
{
  receive(mayproduce,
  pmsg);
  pmsg= produce();
  send(mayconsume,
  pmsg);
}
```

```
/* Consumer: */
cmsg: message;
for(;;)
{
  receive(mayconsume,
  cmsg);
  consume(cmsg);
  send(mayproduce,
  null);
}
```


Signals

- A **signal** is a single bit message.
- Signals are a way to asynchronously notify the occurrence of an event:
  - Timer
  - I/O completion
  - Program exceptions,
  - other user-defined actions
- Sender asynchronous,
- Receive is asynchronous

Unix SVR4 Concurrency Mechanisms

- To communicate data across processes:
  - Pipes
  - Messages
  - Shared memory
- To trigger actions by other processes:
  - Signals
  - Semaphores

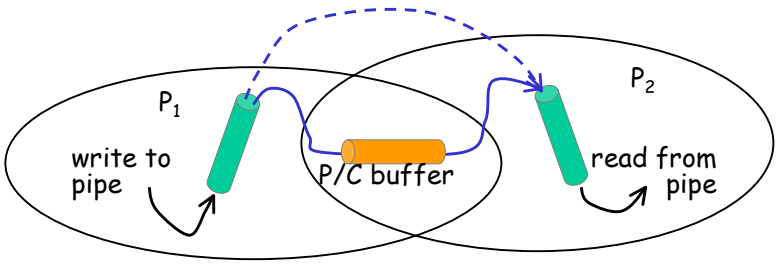
Unix Pipes

- A shared bounded FIFO queue written by one process and read by another
  - based on the producer/consumer model
  - OS enforces Mutual Exclusion: only one process at a time can access the pipe
  - if there is not enough room to write, the producer is blocked, else he writes
  - consumer is blocked if attempting to read more bytes that are currently in the pipe
  - accessed by a file descriptor, like an ordinary file
  - processes sharing the pipe are unaware of each other's existence

Unix Pipes

- Byte offset stored in the i-nnode table, not in the entries of the global file table.
- Pipes use only the direct blocks of the inode for greater efficiency, although this places a limit on the size of the file.
- Direct blocks are managed as a circular queue.
- Cannot use lseek with pipes.

Unix Pipes



The *pipe()* system call

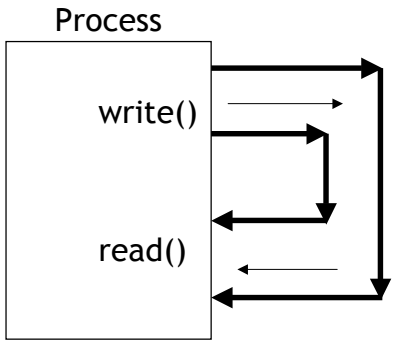
- At the process level we use the *pipe()* system call.
  - prototype: *int pipe(int filedes[2])*
  - filedes[0] will be a file descriptor open for reading
  - filedes[1] will be a file descriptor open for writing
  - the return value of pipe() is -1 if it could not successfully open the file descriptors.
- But how does this help to communicate between processes??

First pipe example

```
#include ....
main()
{
        int p[ 2], pid;
        char buf[ 64];
if( pipe( p) == -1)
{
        perror(" pipe call");
        exit( 1);
}
/* at this point we have a pipe p with p[ 0]
  opened for reading and p[ 1] opened for
  writing - just like a file */
```

```
write( p[ 1], "hi there", 9);

read( p[ 0], buf, 9);

printf("% s\ n", buf);


}
```

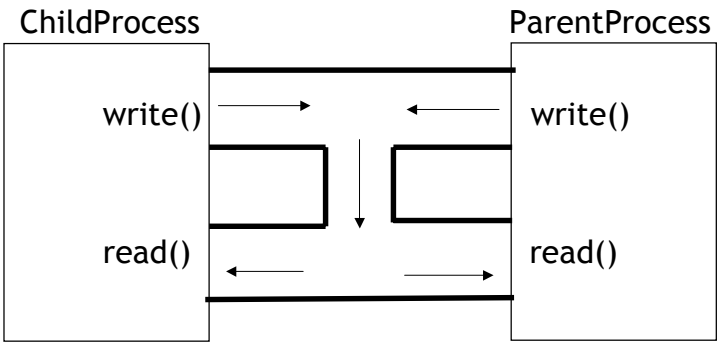

A pipe : loop-back mechanism



A channel between parent/child processes

- the child is created by a *fork()* call that is executed by the parent.
- the child process is an image of the parent process ---> all the file descriptors that are opened by the parent are now available in the child.
- The file descriptors refer to the same I/ O entity, in this case a **pipe.**
- The **pipe** is inherited by the child and may be passed on to the grand-children by the child process or other children by the parent.
- This can easily lead to a chaotic conglomeration of **pipes** throughout our system of processes

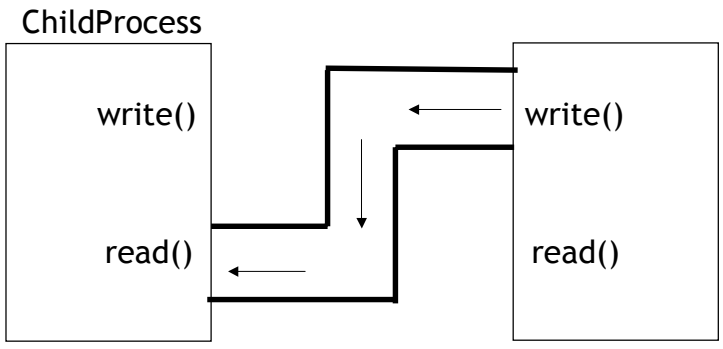
The open pipe problem



How to solve the open pipe problem

```
int pid, p[2];
...
pipe( p );
if ( ( pid = fork() ) == 0 ) {
close( p[1] );
... read using p[0] as file descriptor ...
}
close( p[0] );
... write using p[1] as file descriptor ...
close( p[1] );  /* send EOF to the read port */
wait( &status );
```


How to solve the open pipe problem



Standard I/O Pipe

- Connect a pipe to parent's stdout and child's stdin

```
int pid, p[2];
...
pipe( p );
if ( ( pid = fork() ) == 0 ) {
close( p[1] );
dup2( p[0], 0 );  /* duplicate p[0] using stdin */
close( p[0] );
... read from stdin ...
exit( 0 );
}
close( p[0] );
dup2( p[1], 1 );   /* duplicate p[1] using stdout */
close( p[1] );
... write to stdout ...
wait( &status );
```


Closing the pipe

- The file descriptors associated with a pipe can be closed with the *close(fd)* system call
- Some Rules:
  - A read() on a pipe will generally block until either data appears or all processes have closed the write file descriptor of the pipe!
  - Closing the write fd while other processes are writing to the pipe does not have any effect!
  - Closing the read fd while others are still reading will not have any effect!
  - Closing the read while others are still writing will cause an error to be returned by the write and a signal is sent by the kernel (Broken Pipe!!)

The size of a pipe

- In most cases, we only transfer small amounts of data through a pipe - but we for some applications we may want to send and receive large data blocks.
- A valid question is: *How much data will fit into a pipe ??*
- Why do we care? Remember - a *write()* will block until the requested number of bytes have been written.
- The POSIX standard specifies a minimum size of 512 bytes!

Named Pipes

- `int mkfifo(const char *pathname, mode_t mode);`
- Creates a file of type fifo, or named pipe.
- After the file is created, it can be accessed using the usual open(), read(), write(), close(), unlink() and so forth.
- Opening a fifo for reading will block until someone else has opened the fifo for writing, unless the fifo is opened with the O_NONBLOCK option (in the second argument of the open() system call).

Named Pipes

- Fifos are similar to pipes but can be used between processes without a parent-child relationship.
- Here is an example of using fifos from the shell.

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < input_file | tee fifo1 | prog2
```


Unix Messages

- A process can create or access a message queue (like a mailbox) with the ***msgget*** system call.
- ***msgsnd*** and ***msgrcv*** system calls are used to send and receive messages to a queue
- There is a "type" field in message headers
  - FIFO access within each message type
  - each type defines a communication channel
- Process is blocked (put asleep) when:
  - trying to receive from an empty queue
  - trying to send to a full queue

Shared memory in Unix

- A block of virtual memory shared by multiple processes
- The ***shmget*** system call creates a new region of shared memory or return an existing one
- A process attaches a shared memory region to its virtual address space with the ***shmat*** system call
- Mutual exclusion must be provided by processes using the shared memory
- Fastest form of IPC provided by Unix

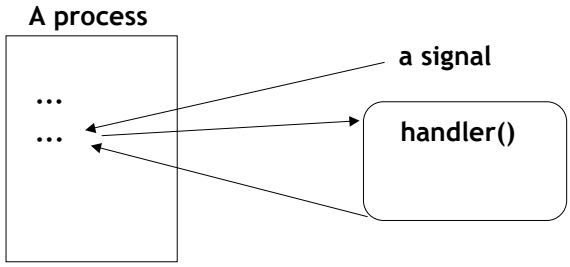
Signals

- Outside world signals are interrupts

  CTRL-C is typed

  phone is hung up

  . . .
- Processes may need to respond to the signals



Unix Signals

- Each signal is represented by a numeric value. Ex:
  - 02, `SIGINT`: to interrupt a process
  - 09, `SIGKILL`: to terminate a process
  - 3, `SIGQUIT`: abort
  - 14, `SIGALRM`: timer interrupt
  - 18, `SIGCLD`: child exited
- Each signal is maintained as a single bit in the process table entry of the receiving process: the bit is set when the corresponding signal arrives (no waiting queues)
- A signal is processed as soon as the process runs in user mode
- A default action (eg: termination) is performed unless a signal handler function is provided for that signal (by using the ***signal*** system call)

UNIX Signal Facility

- UNIX provides a mechanism for catching a signal (like an exception)

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

The function "handler()" will be invoked on a signal "sig"

```
handler( sig );
```

Return the old signal handler on success and -1 on an error

- Examples

```
#include <signal.h>
...
signal( SIGINT, SIG_IGN );   /* ignore interrupts */
...
signal( SIGINT, SIG_DFL );   /* restore the default handler */
```

- There are more than 30 predefined signals in <signal.h>

Signal Handling Example

- Remove "tempfile" on an interrupt:

```
#include <sys/signal.h>
char *tempfile = "temp.xxx";
void cleanup( int sig )
{
    unlink( tempfile );
    exit( 1 );
};
void main( void )
{
    int fd;
    signal( SIGINT, cleanup );        /* set up cleanup() for
    interrupt */
    fd = open( tempfile, O_CREAT, 0666 );
    ...                               /* processing using
    tempfile */
    close( fd );
}
```

Inter-Process Communication - 39 - © 2002 William Fornaciari

Signals with Fork

- Signals are sent to all your processes
- You often want to disable signals and enable them later
- fork() may cause two processes to read your terminal (/dev/tty) at the same time
- An example:

```
#include <signal.h>
...
if ( fork() == 0 )
        execlp( ... );
h = signal( SIGINT, SIG_IGN );   /*parent ignores interrupts*/
wait( &status );                 /* until child is done */
signal( SIGINT, h );             /* restore interrupts */
```

Inter-Process Communication - 40 - © 2002 William Fornaciari

Alarm

- Create a child process and kill it in 20 seconds

```
#include <signal.h>
int pid;
void OnAlarm( int sig )
{
    kill( pid, SIGKILL );
}
...
if ( ( pid = fork() ) == 0 ) {
     execlp( . . . );
}
signal( SIGALARM, OnAlarm );            /* setup the handler */
alarm( 20 );                            /* fire off the alarm */
...
```


Alarm

- What if you want to have an alarm in less than a second?

```
unsigned ualarm( unsigned value, unsigned interval )
int setitimer( int which, struct itimerval *value,
            struct itimerval *ovalue )
```


More on Signals

Signals can be sent also through a system call:
`int kill (pit_t pid, int sig).`

Success: returns 0.
If sig = 0, check existance of the process pid
(-1 comes back if it does not exist);

More Alarm Examples

- **Implement time out in communication protocols**
  - Retransmission if no acknowledgment
  - Decide "NFS is not responding."
- **Implement Timestamps for performance measurements**
  - The typical way to get a timestamp is:
  - #include <sys/time.h>
  - ...
  - **gettimeofday(struct timeval *tp; struct timezone *tzp**);
  - But, this call takes x0 to x00 microseconds
  - So, use ualarm to implement something less expensive.

Unix Semaphores

- Are a generalization of the counting semaphores (more operations are permitted).
- A semaphore includes:
  - the current value S of the semaphore
  - number of processes waiting for S to increase
  - number of processes waiting for S to be 0
- We have queues of processes that are blocked on a semaphore
- The system call ***semget*** creates an array of semaphores
- The system call ***semop*** performs a list of operations: one on each semaphore (atomically)

Unix Semaphores

- Each operation to be done is specified by a value **sem_op**.
- Let S be the semaphore value
  - if sem_op > 0:
    - S is incremented and process awaiting for a particular value of S are awaken
  - if sem_op = 0:
    - If S=0: do nothing
    - if S!=0, block the current process on the event that S=0

Unix Semaphores

- if sem_op < 0 and |sem_op| <= S:
  - set S:= S - |sem_op|
  - then if S=0: awake processes waiting for S=0
- if sem_op < 0 and |sem_op| > S:
  - current process is blocked on the event:
    S >= |sem_op|

- Hence: flexibility in usage (many operations are permitted)

BSD Sockets

- Introduced in BSD Unix
- End-points of two-way communication between processes
  - *IP networking*
  - *Unix Pipes*
- File abstraction
- Connection oriented
  - *open socket*
  - *do communication*
  - *close socket*

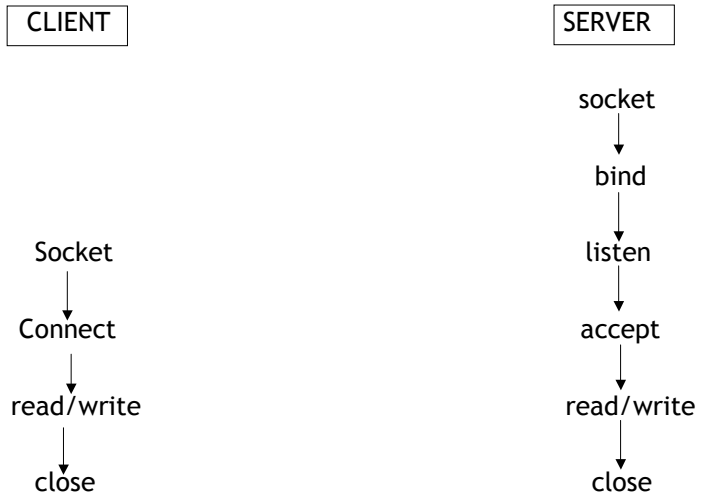
Socket Address Families

- UNIX
  - *unix domain sockets*
- INET
  - *Internet domain sockets over TCP/IP*
- Others
  - *Amateur Radio*
  - *Novell IPX*
  - *AppleTalk*
  - *X25*

Socket Types

- Stream Sockets
  - *bi-directional, reliable, sequenced, no duplicates*
  - *supported over TCP*
- Datagram Sockets
  - *bi-directional, but unreliable*
  - *supported over UDP*
- Raw Sockets
  - *allows direct access to underlying communication protocols*

Stream Sockets



Socket Creation

- Create Socket

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
AF_UNIX or AF_INET

SOCK_STREAM or SOCK_DGRAM

IPPROTO_TCP for SOCK_STREAM
IPPROTO_UDP for SOCK_DGRAM
```

if 0 system will chose appropriate protocol automatically

Socket Binding

- Bind a socket

```
(not necessary for client if AF_INET,
if AF_UNIX - MUST DO)

# include <sys/types.h>
# include <sys/socket.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen);


#include <sys/un.h>
struct sockaddr_un {
   short sun_family; /* == AF_UNIX */
   char sun_path[108];
                                  };

#include <netinet/in.h>
struct sockaddr_in {
   short sin_family; /* == AF_INET */
   u_short sin_port;  /* port number */
   struct in_addr sin_addr;  /* host IP address */
   char sin_zero[8];  /* not used */     };
```


Socket Connect & Listen

- Connect

  (for datagram type is not necessary, for stream - MUST DO)

```
# include <sys/types.h>
# include <sys/socket.h>
int connect (int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- Listen

```
# include <sys/types.h>
# include <sys/socket.h>
int listen (int sockfd, int backlog);
```

  - backlog: specifies how many outstanding connection requests can be queued

Socket Accept

- Accept

```
# include <sys/types.h>
# include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr, int *addrlen);
```

  - addr: address of connected peer process (client)
  - *creates a new socket for each connection request, and returns the descriptor of the new socket*
  - *new socket is used for transferring data*
- Read/Write
  - *read (s, buf, nbytes)*
  - *write (s, buf, nbytes)*

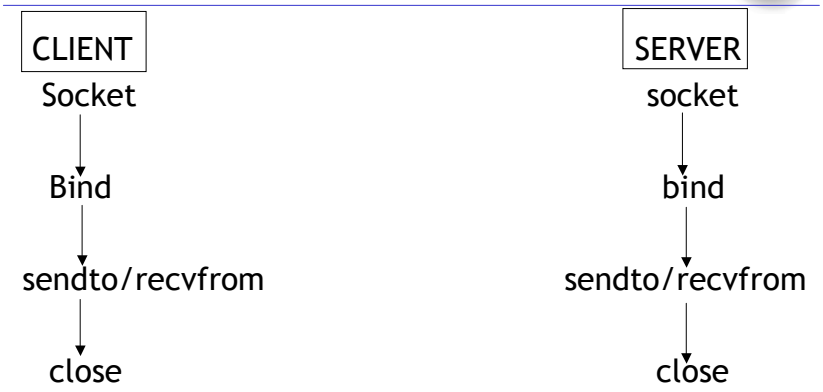
Socket Stream Send and Receive

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

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

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

both return the length of sent (received) chunk of data

flags:
MSG_OOB - urgent send/receive
MSG_PEEK - pick up message leaving it in socket

Datagram Sockets



Socket Interface

- Send-To and Recv-From

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len,  unsigned
   int flags, const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int
   flags, struct sockaddr *from, int *fromlen);
```

  - from/to: Address from (to) which packet is to be sent
  - addrlen: Size of the from/to argument
- Note:
  - *Address is specified in each call*
    - since no notion of connection
    - In recv from, protocol layers fill in the address of who sent the data

Closing a socket

```
# include <sys/types.h>
# include <sys/socket.h>
int shutdown (int sockfd, int mode);
```

0 - for read
1 - for write
2 - for both

Could use close also. Difference - if connection oriented, close will be blocked until system delivers all data.

GetHostByName..

```
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Retrieve information about a remote computer using either its name ("www.yahoo.com") or its address (as we have seen several address types exist).

The hostent structure is defined in <netdb.h> as follows:

```
struct hostent {
     char   *h_name;        /* official name of host */
     char  **h_aliases;      /* alias list */
     int     h_addrtype;    /* host address type, AF_INET at present */
     int     h_length       /* The length of the address in bytes.
     char  **h_addr_list;   /* zero-terminated array of hosts' network
                               addresses in network byte order. */
     char   *h_addr;        /* First address in h_addr_list */
}
```

When using gethostbyname() or gethostbyaddr(), the variable errno and the function perror() won't work. The two functions will return NULL on error, but in order to know which error occurred and to print an appropriate error message, replace errno and perror by the following (which do the same):

```
extern int h_errno;
void herror(const char *s);
```

Linux BSD Socket Data Structures

- files_struct
  - *one entry for each file descriptor*
  - *points to struct* **file**
- struct **file**
  - *f_op: pointers to file operations*
  - *f_inode: pointer to V-Node (struct* **inode***)*
- Vnode (struct inode)
  - *Generic Vnode information*
  - *Pointer to struct* **socket**

Linux BSD Socket Data Structures

- Socket (struct **socket**)
  - *state:*
  - *type:*
  - *ops: Address Familty Socket Operations*
  - *Pointer to struct sock*
- Inet socket layer data structure (struct **sock**)
  - *local and dest ports & addresses*
  - *address family*
  - *Data Queues*
  - ….

Implementation

- Creating a Socket
  - *Set up the data structures*
  - *Vnode is marked as "socket"*
  - *Within socket, protocol specific ops are set up*
- Binding (an address to a socket)
  - *The socket must not be "in-use"*
  - *Stored in*
    - recv_addr and saddr fiels in struct sock
    - If port number is not specified, it is chosen automatically

Implementation

- Connection (outbound)
  - *socket must be in state "unconnected"*
- UDP
  - *No actual network connection is made, but still useful*
    - send does not require specification of remote address
    - socket will only receive from a particular addr
  - *Involves nothing the remote addr in the data structures*

Implementation

- TCP Connect
  - *Establish connection with the remote site*
  - *Build a TCP message and send it to the remote IP host*
    - includes initialization information
    - wait for ack/retransmit if necessary
- Listen (for incoming connections)
  - *When an incoming connect request arrives*
    - set up data structures (struct **sock**)

Implementation

- Accept Connection Requests
  - *If there is a connection request, then set it up*
  - *else,*
    - either block or return with failure
  - *The sock data structure is linked to a new file descriptor and returned to the application*

Memory Mapped Files

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int
   flag, int filedes, off_t off);
```

- Maps a file into memory.
- 'filedes' is the descriptor of the file to map; the file must be open, however closing it later will not un-map it.
- 'addr' is the address in memory to map to – it is recommended to pass 0 as this argument to let the system decide.
- 'off' is the offset in the file from which to map;
- 'len' is the number of bytes to map, starting from that offset.
- 'prot' is the protection mode for the mapped file – it must match the permissions the file was opened with and can be a combination of PROT_READ, PROT_WRITE and PROT_EXEC.
- returns the address in which the mapped file is. If 'flag' is MAP_SHARED then store operations will modify the mapped file: In other words, writing to this memory will be equivalent to the write() system call.

Memory Mapped Files

```
int munmap(caddr_t addr, size_t len);
```

- Unmaps a file. This also happens automatically when the process terminates.
- Copying a file using mmap() is much faster than using read() and write(), because there is no need to copy buffers from kernel to user space and back during the copy.
- FASTCOPY:

1. Open the source file with open(), then map it with mmap().
2. Create the destination file with open().
3. Read the size of the input file using fstat(), then set the size of the destination file to that same size using lseek(fdout, src_size-1, SEEK_SET) and write(fdout, "", 1).
4. Map the destination file with mmap(), with the flag argument MAP_SHARED.
5. Call memcpy(dst, src, src_size) to copy the files by memory-to-memory copy.

REFERENCES

- Stallings W. **Sistemi Operativi** Jackson Libri
- Silberschatz A. **Sistemi Operativi** Addison-Wesley
- Tanenbaum A. **I Moderni Sistemi Operativi** Jackson Libri
- Cay S. Horstmann. **Concetti di Informatica e fondamenti di JAVA 2** Apogeo
- Web Material (see links)