

Linux/Unix inter-process communication

Mohammad S. Hasan

Staffordshire University, UK

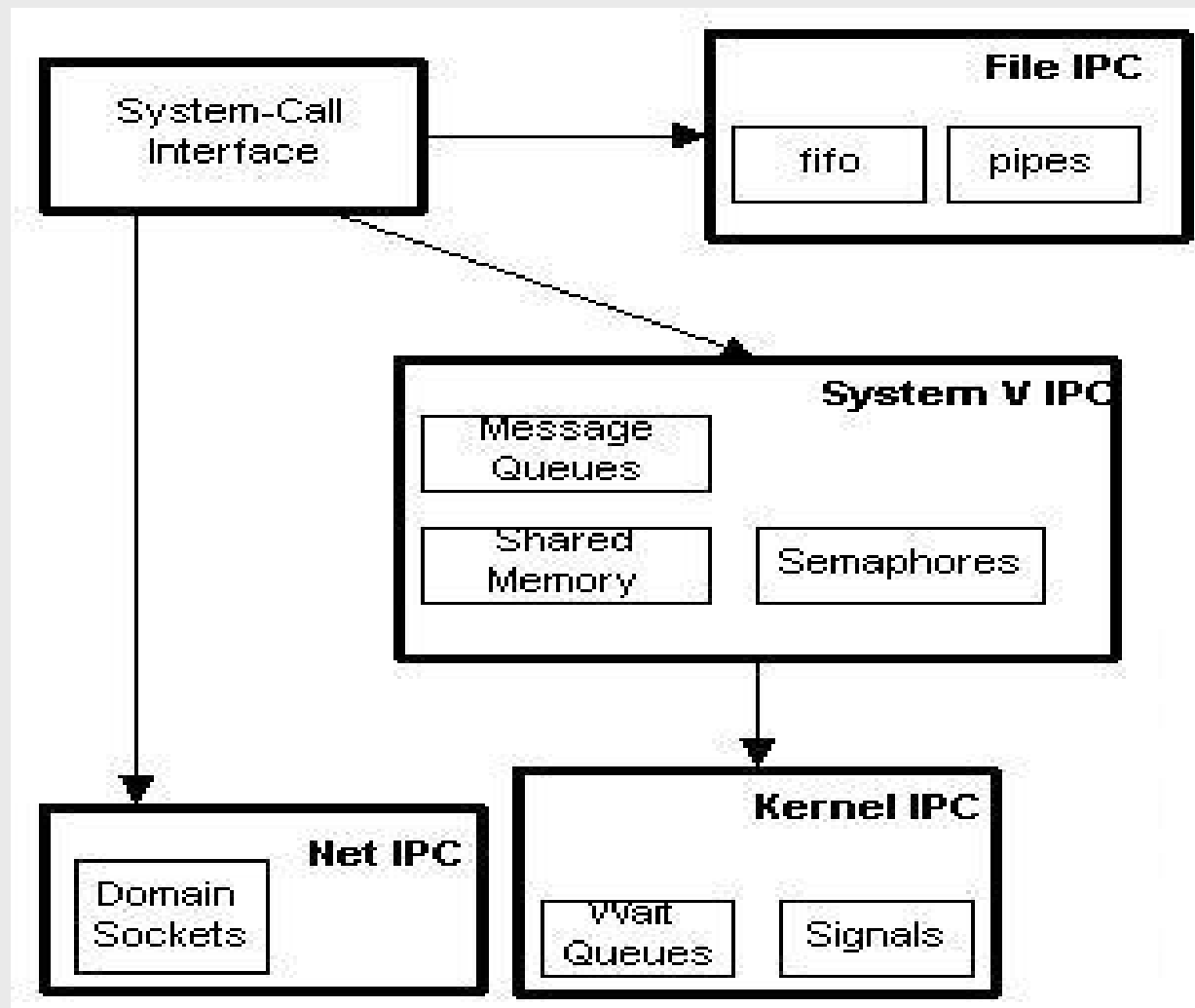
Lecture Outline

- ◆ **Inter-process communication methods in Linux/Unix**
 - **Files**
 - **Pipes**
 - **FIFOs**
 - **Signals**
 - **Sockets**
 - **System V IPC - semaphores, messages, shared memory**

Interprocess communication (IPC)

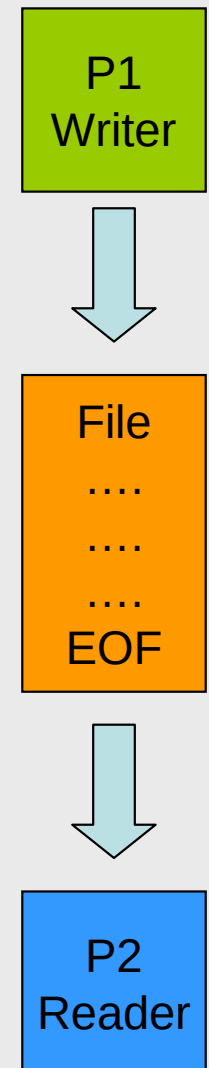
- ◆ There are many interprocess communication mechanisms in UNIX and Linux
- ◆ Each mechanism has its own
 - system calls,
 - advantages and disadvantages
- ◆ Files, pipes, FIFOs, signals, semaphores, messages, shared memory, sockets, streams
- ◆ We will look a number of them

Interprocess communication (IPC) (Cont.)



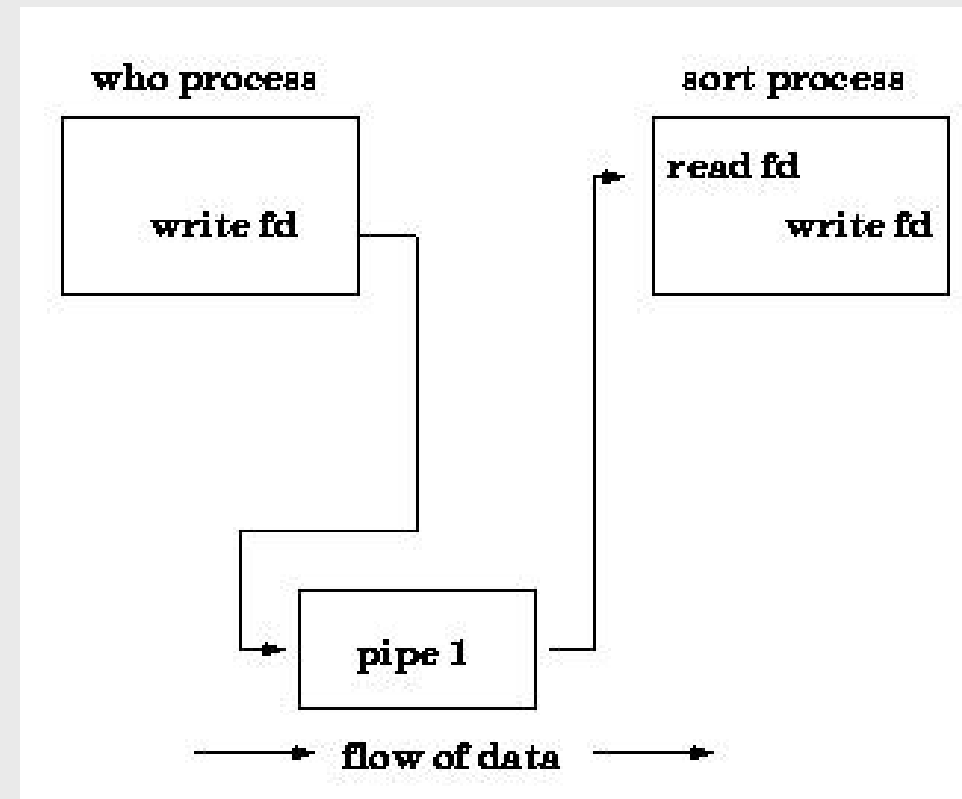
Files

- ◆ **These can handle simple IPC but have 2 main problems for serious IPC work:**
 - if reader faster than writer, then reader reads EOF, but cannot tell whether it has simply caught up with writer or writer has completed (synchronisation problem)
 - writer only writes to end of file, thus files can grow very large for long lived processes (disk/memory space problem)



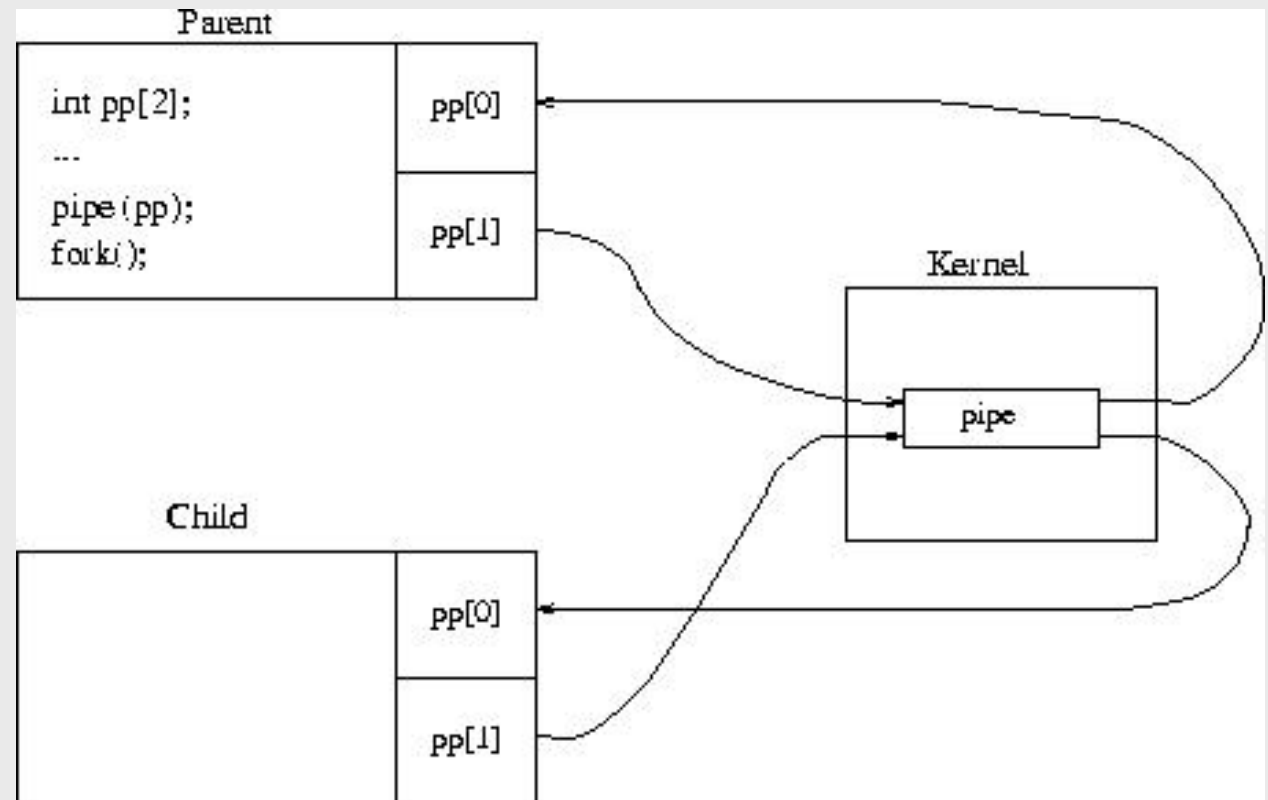
Pipes

- ◆ You should already be familiar with *pipe* from practical exercises e.g.
 - **who | sort**
- ◆ This sets up a one directional pipeline that takes the output from 'who' and gives it as input to 'sort'



Pipes (Cont.)

- ◆ The data is transmitted via a small fixed size buffer (usually 4KB)
- ◆ Essentially an example of a producer and consumer communication link



Pipes (Cont.)

- ◆ Pipes are a classic Unix/Linux IPC mechanism

- ◆ There are 2 types of pipes in UNIX/Linux

- **Anonymous pipes**

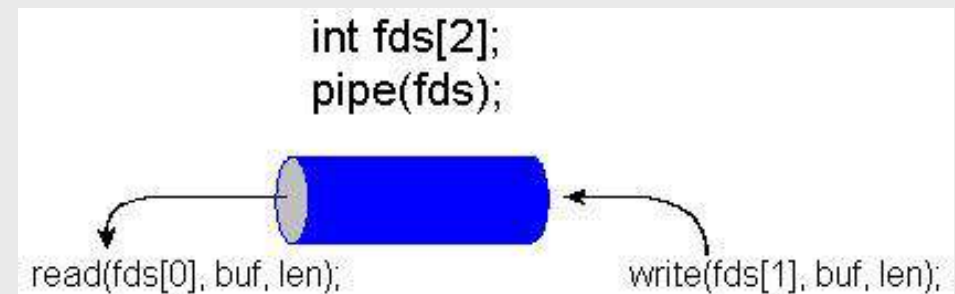
- Has i-nodes (defines location) but NO directory links (i.e. NOT part of file system).

- **Named pipes (FIFOs)**

- Has both i-nodes (defines location) and directory links (part of file system).

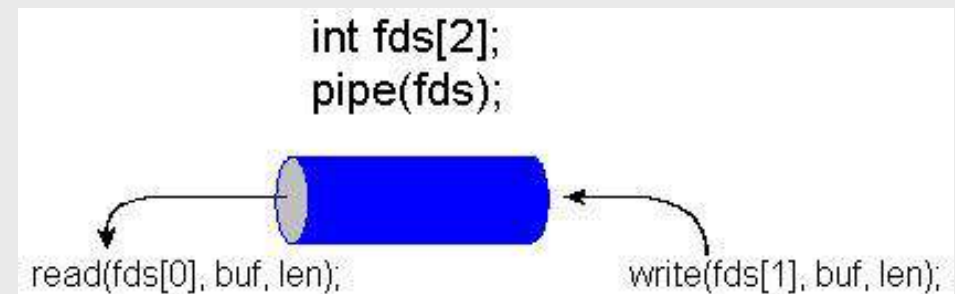
Anonymous pipes

- Any process can create an anonymous pipe with the *pipe()* system call
- the pipe system call replaces the *open()* or *creat()* system calls used for ordinary files
- It creates a buffer with 2 file descriptors (pointers to file structures) that point at it,
 - one for the read end of the pipe and
 - the other for the write



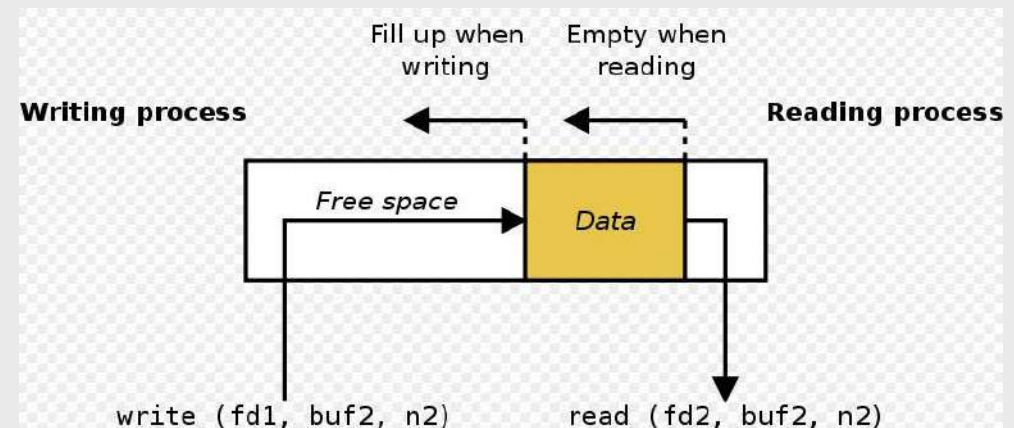
Anonymous pipes (Cont.)

- ◆ The pointers are held in an int array e.g. `int fds[2]`.
 - `fds[0]` = read end of pipe
 - `fds[1]` = write end of pipe



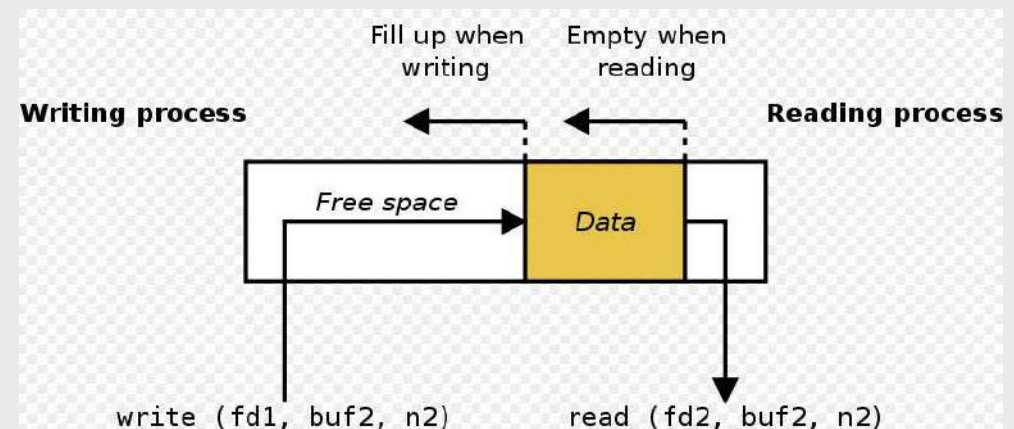
Writing to a pipe

- By default, data is written to a pipe in order of arrival.
- If the pipe becomes full then the `write()` will sleep until enough data has been read to make space for the new data
- Data is read in the order written and can only be read once



Reading from a pipe

- ◆ By default, a `read()` on an empty pipe will cause the `read()` to sleep until some data is written
- ◆ The `close()` system call can be used with pipes just as with ordinary files.
 - The file descriptors are released for future use.



Pipes – advantages

- ◆ Pipes solve the synchronisation problems of files with blocking read and write
- ◆ The small size of pipes means that pipe data are seldom written to disk; they usually are kept in memory (faster than file)
- ◆ They also solve the problem of file sizes growing indefinitely (with shared files) by being of fixed size (usually 4K)

Pipes - disadvantages

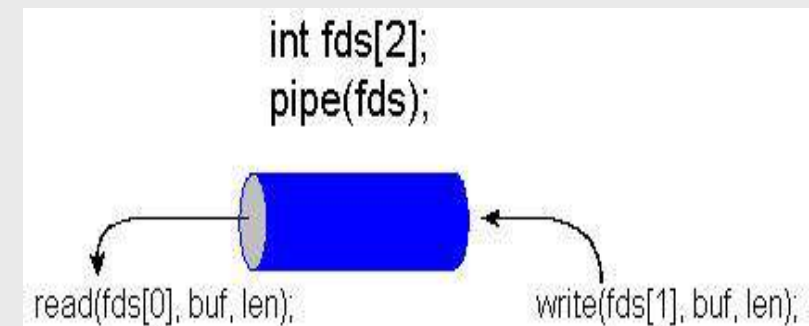
◆ Use of pipes has 3 disadvantages:

- File descriptor for pipe is private to process that created it and its descendants.
 - reader and writer must be related e.g. parent, child
- Reads and writes need to be atomic
- Pipes can be too slow – if there are lots of data to copy via buffer

Example skeleton pipe code

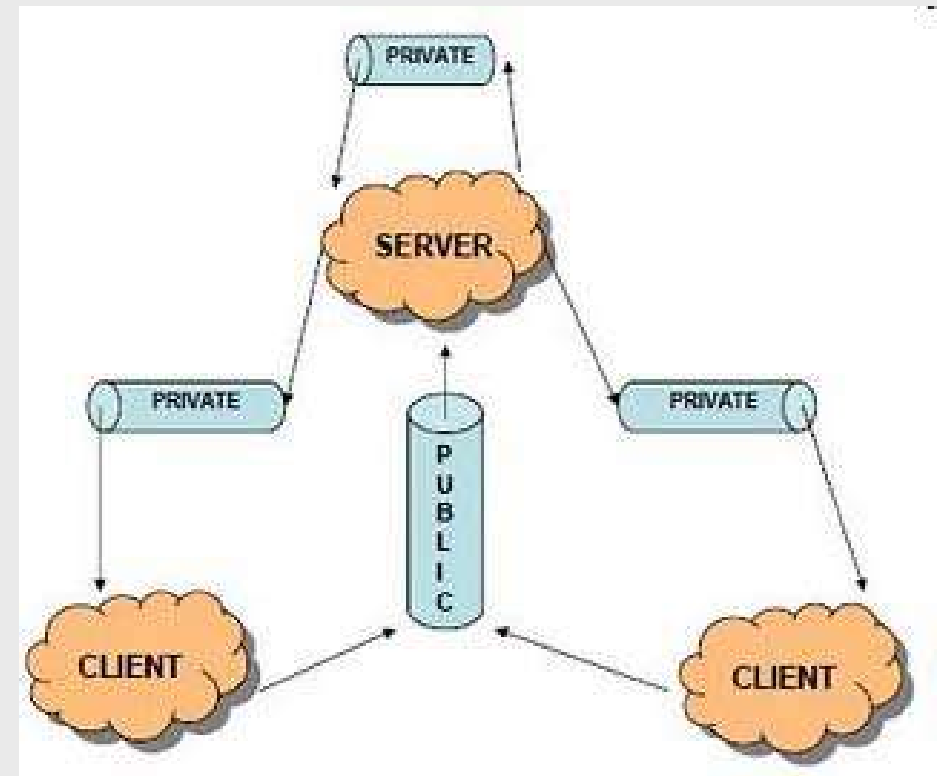
◀ Setting up a pipeline is as follows:

```
int fds[2] // declaration of file descriptor array for pipe
pipe (fds); // create the pipe
if (fork()) { // parent process – producer process
    close(fds[0]); // close pipe read
    write(fds[1], "a", 1); // write to pipe
}
else { // child process – consumer process
    close(fds[1]); // close pipe write
    read(fds[0], buf, 1); // read from pipe
}
```



FIFOs (or Named pipes)

- ◆ Unlike pipes FIFOs have directory links (i.e. file names)
 - Unrelated processes can open() and use them provided process has access permissions to file
- ◆ FIFOs are created with a system call called *mknod()*
- ◆ The *mknod()* call for a FIFO replaces the pipe and the *creat()* call for a file



FIFOs (Cont.)

◆ Advantages

- FIFOs are a cross between *File* and *Pipe* having the advantages of both
- FIFOs reads and writes are atomic so multiple readers and writers are easy to deal with

◆ Disadvantages

- The only problem with FIFOs is that like pipes they can still be too slow in critical applications

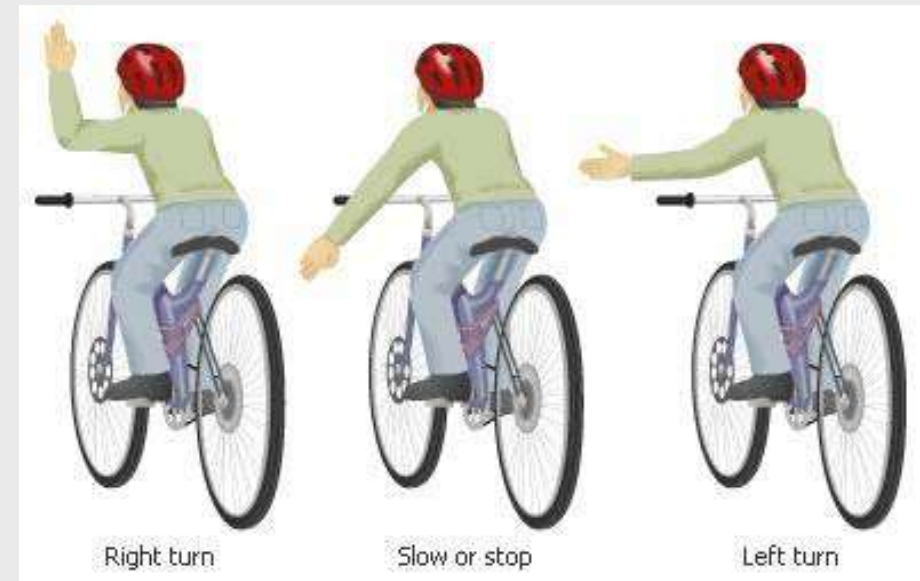
Signals

- ◆ Signals provide a method for handling exceptional conditions
- ◆ Signals don't pass info and can only be used to alert a process of the occurrence of some event
- ◆ To send a signal to a process requires the `kill()` system call which passes on the specific type of signal to a process



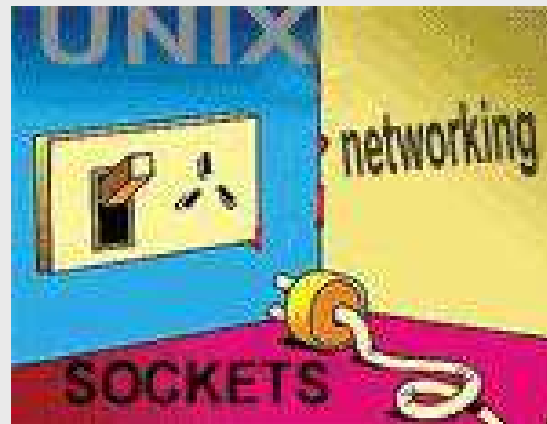
Signals (Cont.)

- ◆ There are many types of signal.
- ◆ The default effect of most of them is to terminate the receiving process
- ◆ All these signals (except SIGKILL) can be either ignored or be processed by special code on receipt of a given signal
- ◆ The cyclist signals to stop.



Sockets

- ◆ Berkeley UNIX introduced *sockets* to enable communication between processes on separate machines as well as within one machine.
- ◆ As a result sockets support communication using many different network protocols.
- ◆ A socket is an endpoint of communication.



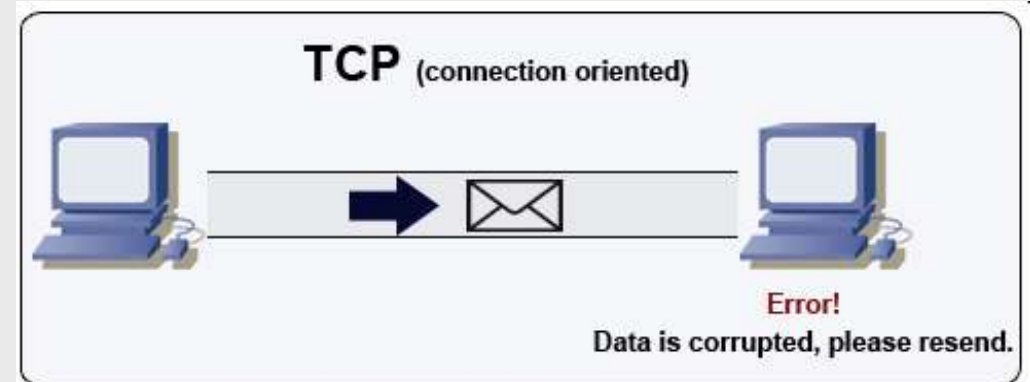
Sockets (Cont.)

- ◆ An in-use socket is usually bound with an address
 - the nature of the address depends on the *communication domain* of the socket.
- ◆ Processes communicating in the same domain use the same *address format*.
- ◆ The read() and write() system calls work on sockets in the same way as Files and Pipes
- ◆ There are a large variety of possible connection types and protocols available

Socket Types

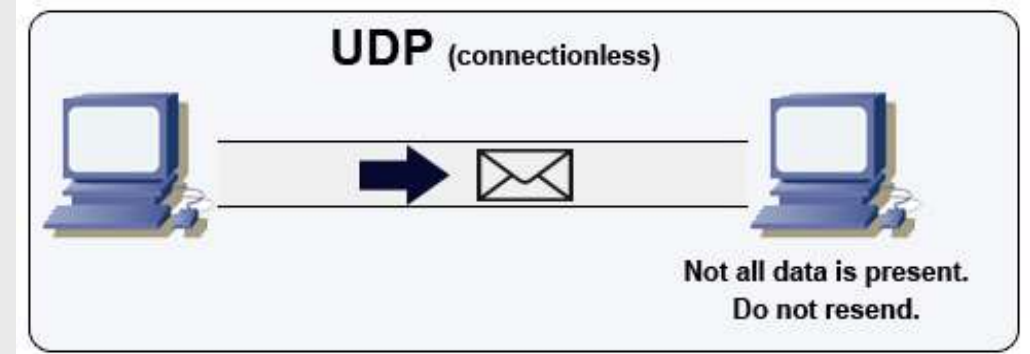
Stream sockets

- provide reliable, duplex, sequenced data streams.
- Supported in Internet domain by the TCP protocol.



Datagram sockets

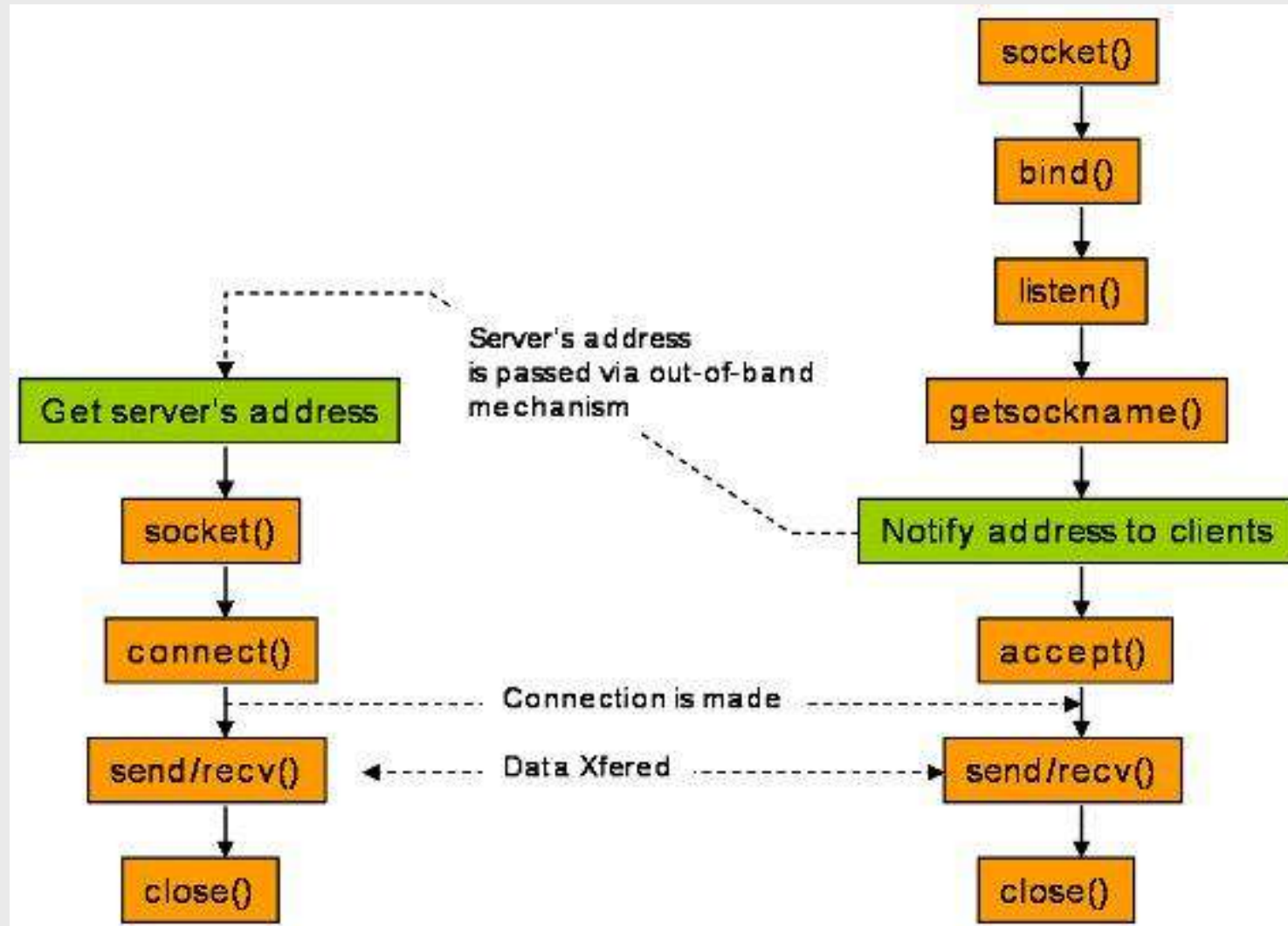
- transfer messages of variable size in either direction.
- Supported in Internet domain by UDP protocol



Socket System Calls

- ◆ The *socket()* call creates a socket;
- ◆ Input arguments
 - specifications of the communication domain,
 - socket type, and
 - protocol to be used
- ◆ Return value
 - a small integer called a socket descriptor.
- ◆ A name is bound to a socket by the *bind* system call.

Socket System Calls (Cont.)



Socket System Calls (Cont.)

- ◆ The *connect* system call is used to initiate a connection.
- ◆ A server process uses *socket* to create a socket and bind to *bind* an address for its service to that socket.
 - Uses *listen* to tell the kernel that it is ready to accept connections from clients.
 - Uses *accept* to accept individual connections.
 - Uses *fork* to produce a new process after the *accept* to service the client while the original server process continues to listen for more connections.

Socket System Calls (Cont.)

◆ The `Close()` system call on its socket descriptor

- to terminate a connection
- to destroy the associated socket

◆ The *select* system call

- can be used to multiplex data transfers on several file descriptors and /or socket descriptors

Streams

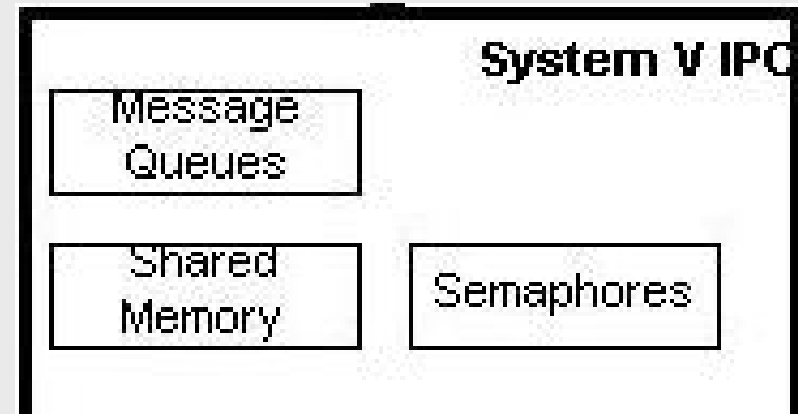
- **Streams are similar to sockets (produced by AT&T).**
- **Provide a dynamically configurable and bi-directional communication channel between processes on the same or different machine**

System V IPC mechanisms

- ◆ This includes 3 IPC types - semaphores, messages and shared memory
- ◆ As they were defined at the same time they all share some common features
 - a table that is the equivalent of the global file table
 - a numeric key that acts like a file name
 - a `get()` type system call that is similar to `open()` that returns a value similar to a file descriptor

System V IPC mechanisms (Cont.)

- a permission structure that specifies access permissions
 - a status structure
 - a control mechanism
- ◆ These IPC mechanism are very fast but are difficult to master



Example - shared memory

- ◆ The same block of memory is made visible (via system calls) within the address space of two or more processes
- ◆ Shared memory is particularly fast – a process will write directly into shared memory and the data is then immediately available to the other process
- ◆ NO copying to/from a file or buffer area is needed

References

- ◆ **Operating System Concepts. Chapter 21 & Appendix A.**