

Inter-Process Communication (IPC)



By
Ravindra Raju Kolahalam

IPC Fundamentals

- What is IPC?
 - Mechanisms to transfer data between processes
- Why is it needed?
 - Not all procedures can be easily built in a single process

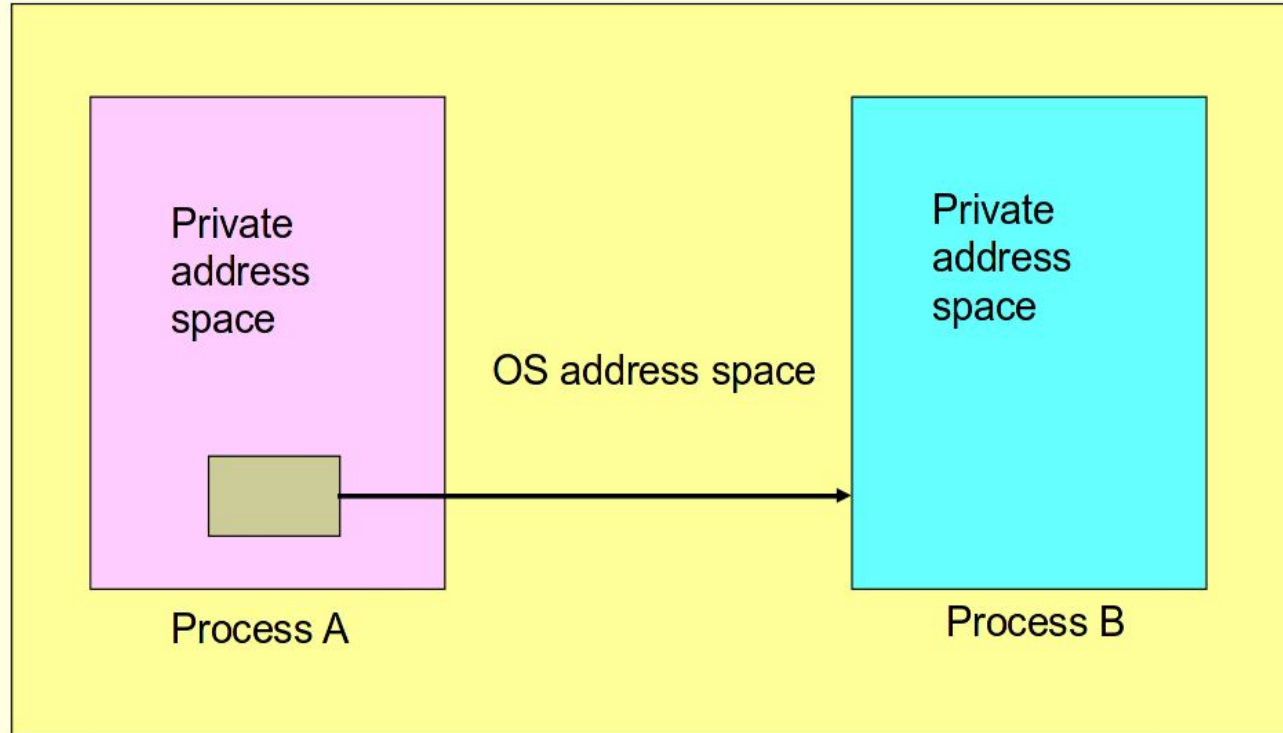
Why do processes communicate?

- To share resources
- Client/server paradigms
- Inherently distributed applications
- Reusable software components
- Other good software engineering reasons

The Basic Concept of IPC

- A process needs to send data to a receiving process
 - Sender wants to avoid details of receiver's condition
 - Receiver wants to get the data in an organized way

IPC from the OS Point of View



Fundamental IPC Problem for the OS

- Each process has a private address space
- Normally, no process can write to another process's space
- How to get important data from process A to process B?

OS Solutions to IPC Problem

- Fundamentally, two options
 1. Support some form of shared address space
 - Shared memory
 2. Use OS mechanisms to transport data from one address space to another
 - Files, messages, pipes, RPC, signal, ...

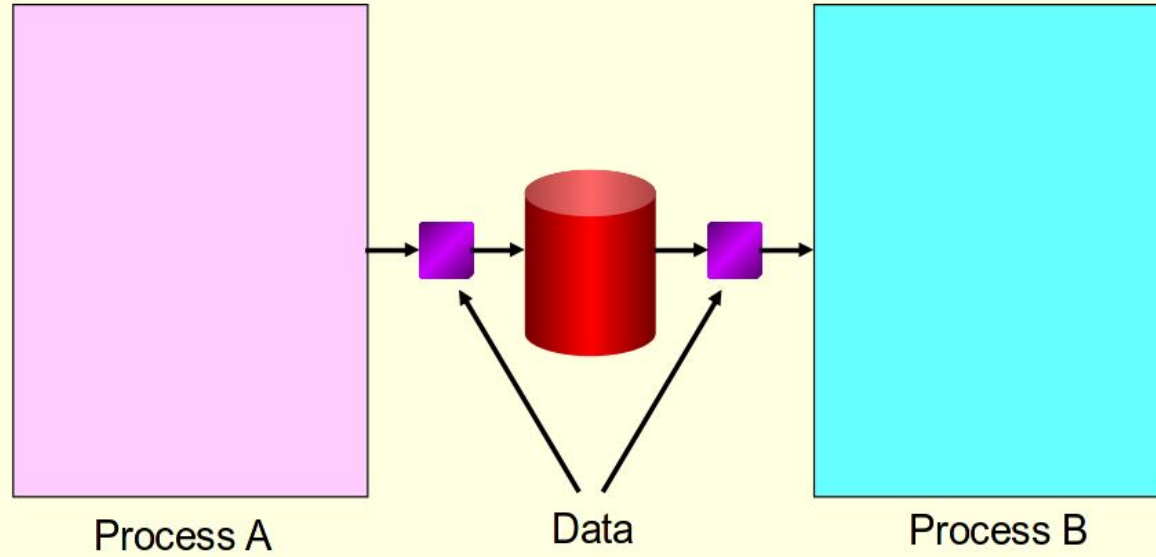
Fundamental Differences in OS Treatment of IPC Solutions

- Shared memory
 - OS has job of setting it up
 - And perhaps synchronizing
 - But not transporting data
- Messages, etc
 - OS involved in every IPC
 - OS transports data

IPC Through the File System

- Sender writes to a file
- Receiver reads from it
- But when does the receiver do the read?
 - Often synchronized with file locking or lock files
- Special types of files can make file-based IPC easier

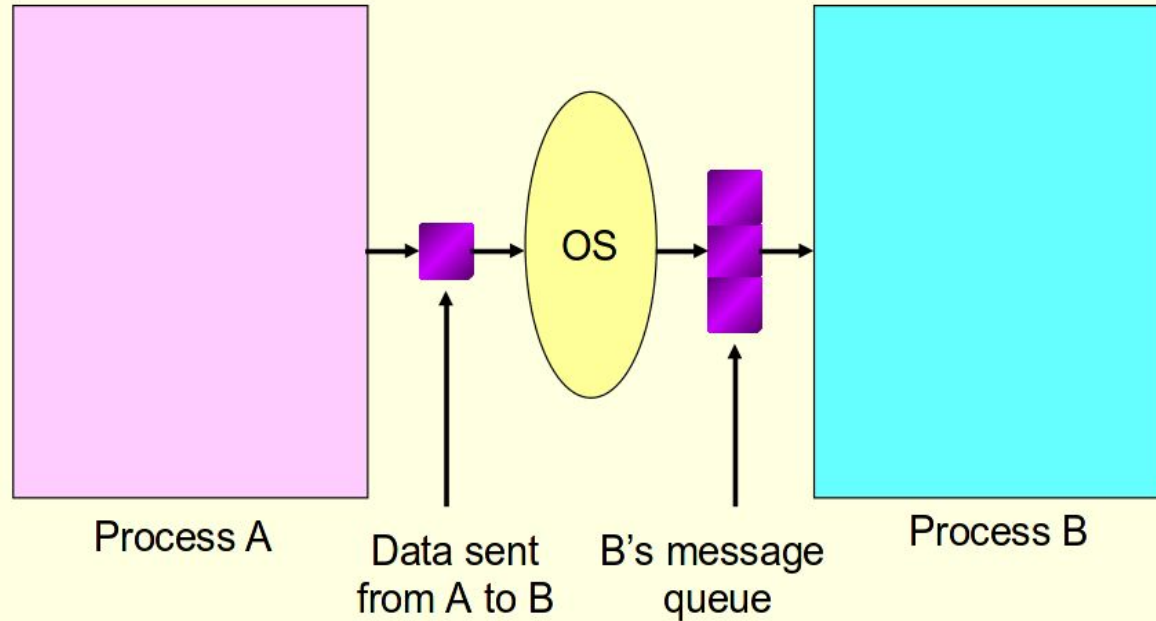
File IPC Diagram



Message-based IPC

- Sender formats data into a formal message
 - With some form of address for receiver
- OS delivers message to receiver's message input queue (might signal too)
- Receiver (when ready) reads a message from the queue
- Sender might or might not block

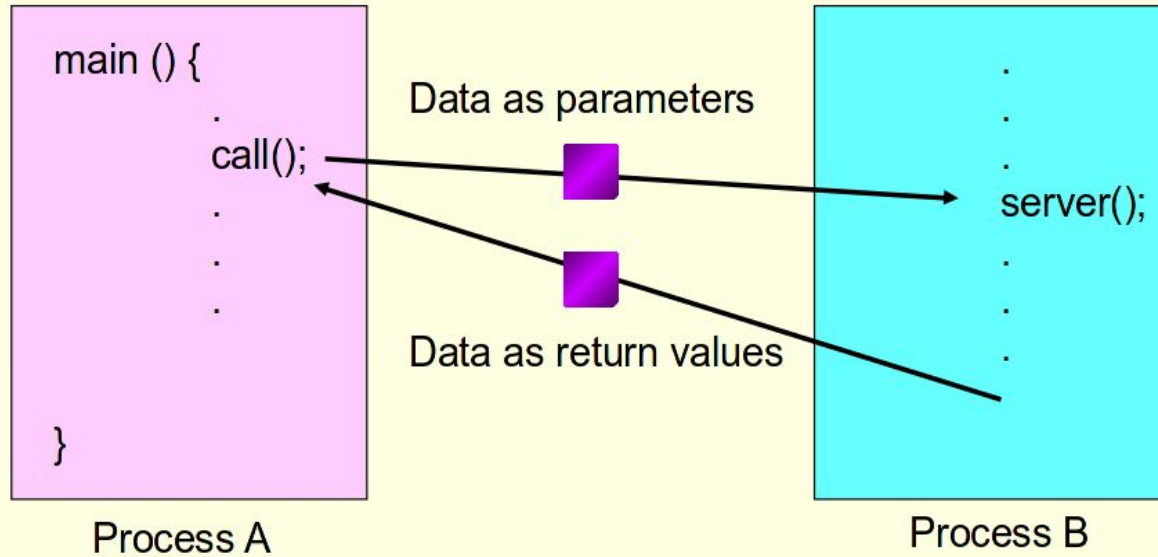
Message-based IPC Diagram



Procedure Call IPC

- Uses same procedure call interface as intraprocess
 - Data passed as parameters
 - Info returned via return values
- Complicated since destination procedure is in a different address space
- Generally, calling procedure blocks till call returns

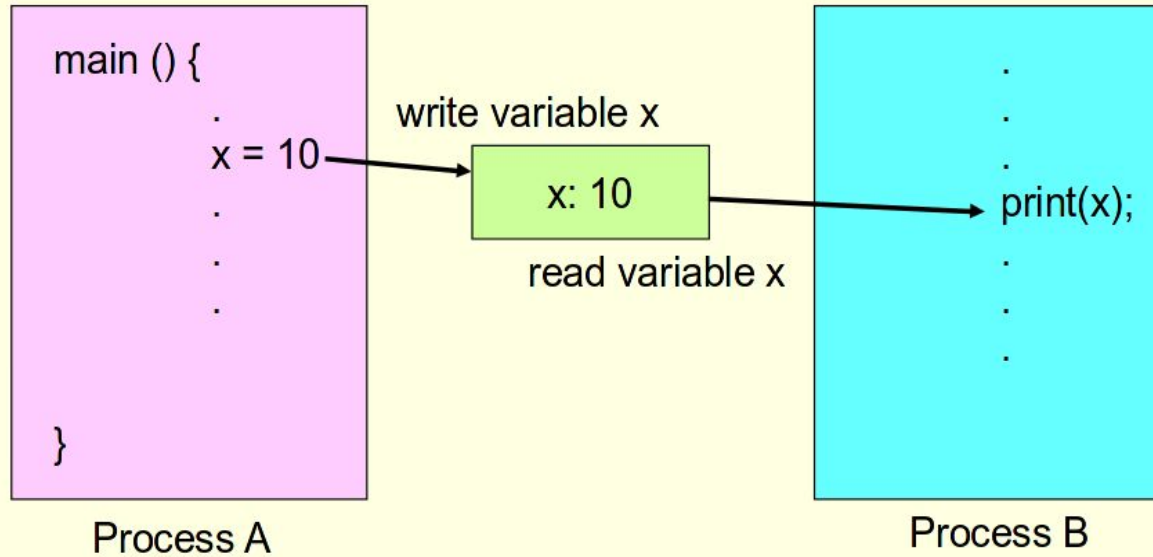
File IPC Diagram

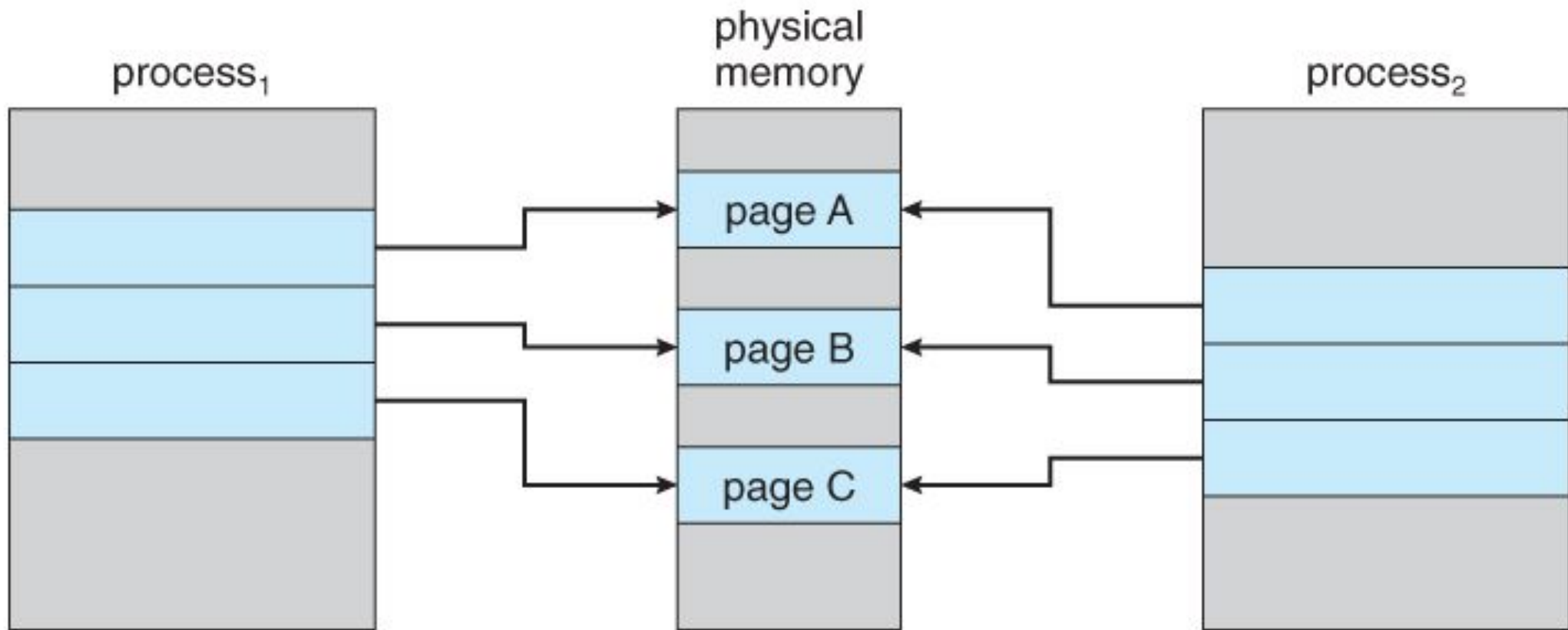


Shared Memory IPC

- Different processes share a piece of memory
 - Either physically or virtually
- Communications via normal reads/writes
- May need semaphores or locks
 - In or associated with the shared memory

Shared Memory IPC Diagram





```
int main() {  
// ftok to generate unique key  
key_t key = ftok("shmfile",65);  
  
// shmget returns an identifier in shmid  
int shmid = shmget(key,1024,0666|  
IPC_CREAT);  
  
// shmat to attach to shared memory  
char *str = (char*)  
shmat(shmid,(void*)0,0);
```

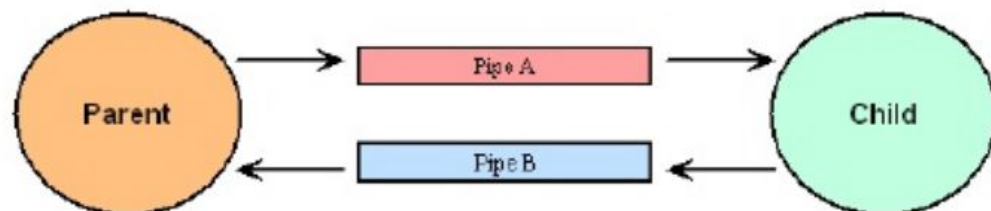
```
int main()  
{  
// ftok to generate unique key  
key_t key = ftok("shmfile",65);  
  
// shmget returns an identifier in shmid  
int shmid =  
shmget(key,1024,0666|IPC_CREAT);  
  
// shmat to attach to shared memory  
char *str = (char*)  
shmat(shmid,(void*)0,0);
```

Pipes

- Only IPC mechanism in early UNIX systems (other than files)
 - Uni-directional
 - Unformatted
 - Uninterpreted
 - Interprocess byte streams
- Accessed in file-like way

What is PIPE

- A **pipeline** is a set of processes chained by their standard streams, so that the output of each process (*stdout*) feeds directly as input (*stdin*) to the next one.
- Each connection is implemented by an anonymous pipe.



Pipe Details

- One process feeds bytes into pipe
 - A second process reads the bytes from it
- Potentially blocking communication mechanism
- Requires close cooperation between processes to set up
 - Named pipes allow more flexibility

Sockets

- Introduced in 4.3 BSD
- A socket is an IPC channel with generated endpoints
- Great flexibility in its characteristics
 - Intended as building block for communication
- Endpoints established by the source and destination processes

Socket

- Sockets provide point-to-point, two-way communication between two processes.
- Sockets are very versatile and are a basic component of inter process and intersystem communication.
- A socket is an endpoint of communication to which a name can be bound.
- It has a type and one or more associated processes.
- Sockets exist in communication domains.
- A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain.

More on Sockets

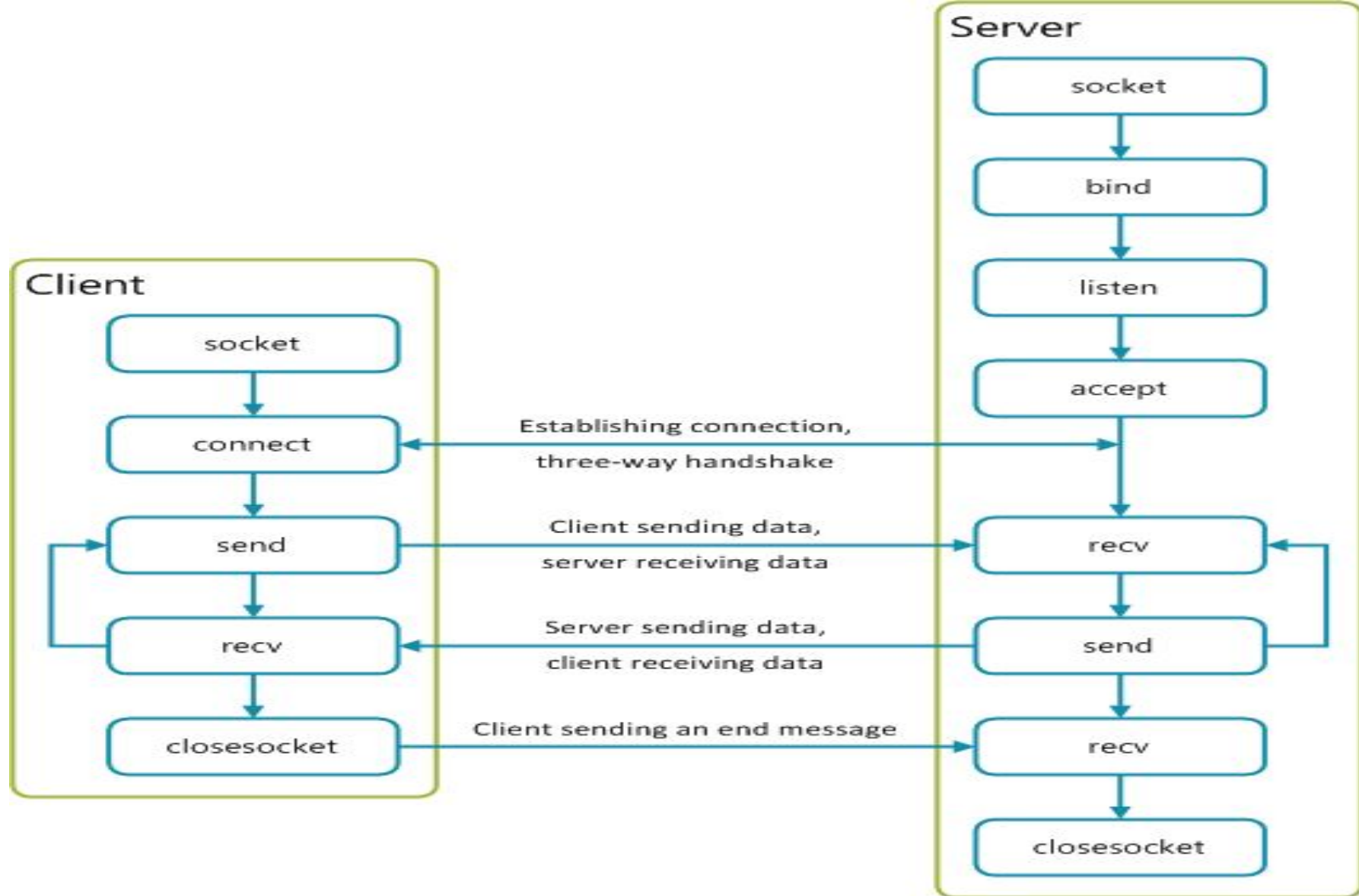
- Created using the `socket()` system call
- Specifying domain, type, and protocol
- Sockets can be connected or connectionless
 - Each side responsible for proper setup/access

Socket Domains

- the socket domain describes a protocol family used by the socket
 - Generally related to the address family
- Domains can be:
 - Internal protocols
 - Internet protocols
 - IMP (interface message processors) link layer protocols

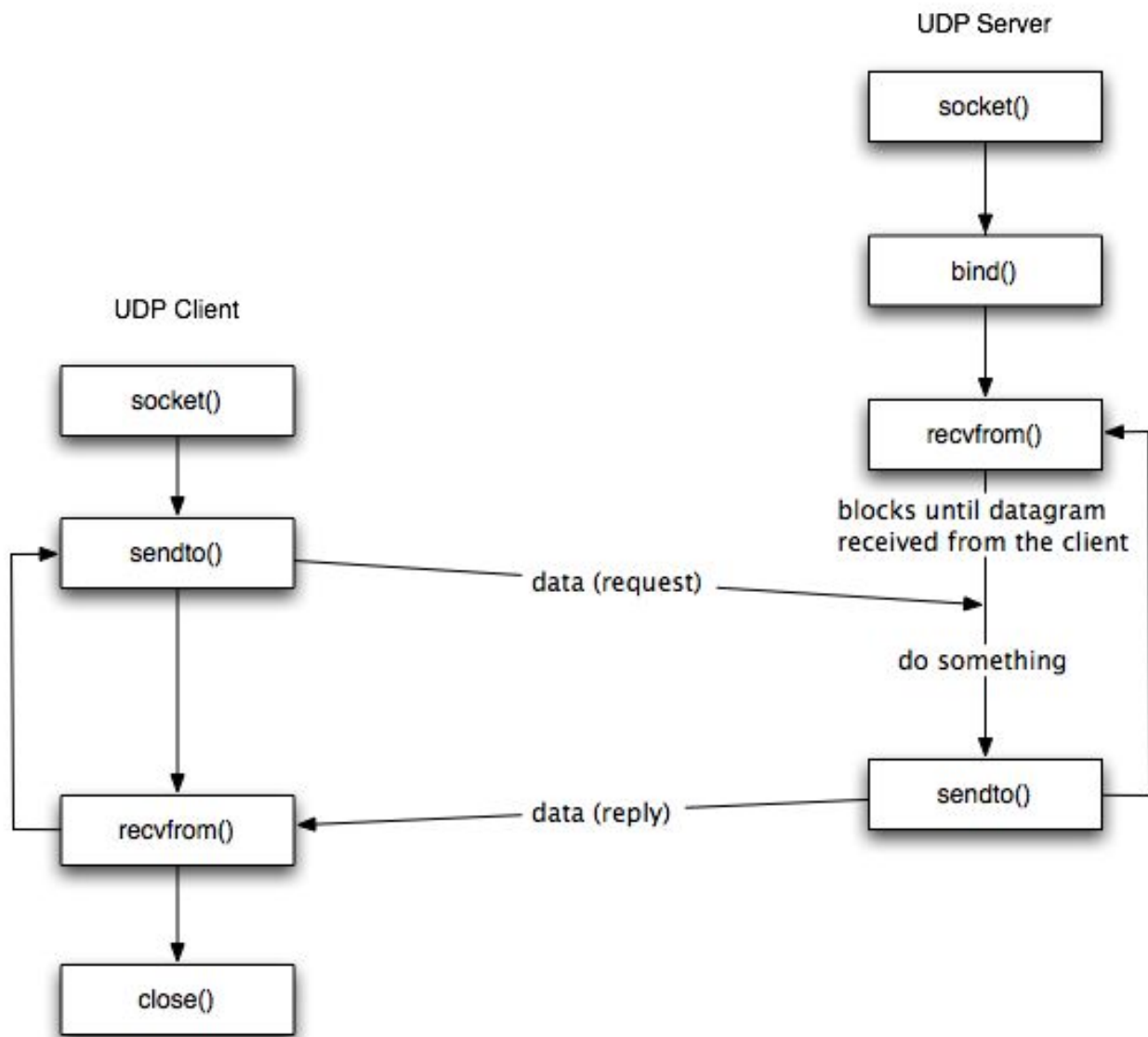
Socket Types

- The socket type describes what the socket does
- Several types are defined
 - SOCK_STREAM
 - SOCK_DGRAM
 - SOCK_SEQPACKET
 - SOCK_RAW
 - SOCK_RDM



- The BSD server creates a socket, uses bind to attach that socket to a port, and configures it as a listening socket. This allows the server to receive incoming connection requests.
- Afterwards, accept is called, which will block the socket, until an incoming connection request is received.
- When accept returns, the SOCKADDR structure will have been filled out with the originating IP Address and port of the incoming connection.
- Then, accept creates a new socket, which is then used to receive data until the connection

BSD Server



Signal

- **Signal** is a limited form of IPC used in Unix and Unix-like operating systems.
- Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred.
- When a signal is sent to a process, the operating system interrupts the process's normal flow of execution.
- Execution can be interrupted during any instruction.

Signal	Portable number	Default action	Description
SIGABRT	6	Terminate (core dump)	Process abort signal
SIGALRM	14	Terminate	Alarm clock
SIGBUS	N/A	Terminate (core dump)	Access to an undefined portion of a memory object
SIGCHLD	N/A	Ignore	Child process terminated, stopped, or continued
SIGCONT	N/A	Continue	Continue executing, if stopped
SIGFPE	8	Terminate (core dump)	Erroneous arithmetic operation
SIGHUP	1	Terminate	Hangup
SIGILL	4	Terminate (core dump)	Illegal instruction
SIGINT	2	Terminate	Terminal interrupt signal
SIGKILL	9	Terminate	Kill (cannot be caught or ignored)
SIGPIPE	13	Terminate	Write on a pipe with no one to read it

