

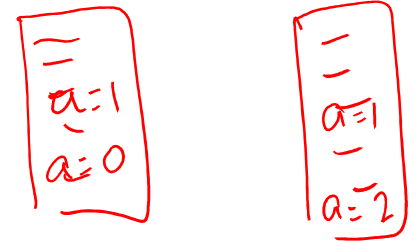
CS 347M (Operating Systems Minor)

Spring 2022

Lecture 6: Inter-process communication

Mythili Vutukuru
CSE, IIT Bombay

Why inter-process communication



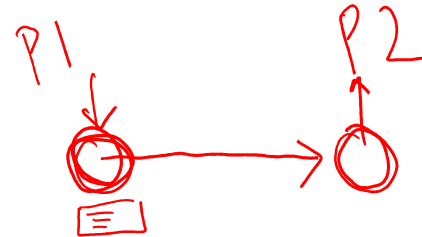
- Application logic in a single system is often distributed across multiple processes: why?
 - Different processes developed independently by different teams
 - Different programming languages and frameworks used for different tasks
- Processes in a system do not share any memory with each other by default, so how do they communicate information with each other?
 - Cannot share variables or data structures in programs across processes
 - Parent and child have identical but separate memory images after fork, changes made in one process and not seen by other
- Inter-process communication (IPC) mechanisms, available via operating system syscalls, allow processes to exchange information



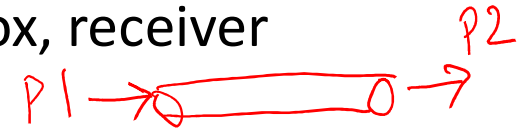


- Example: web applications typically composed of multiple processes
- **Web server** process handles HTTP (web) requests/responses
 - Written in a language like C/C++ for high performance
 - Returns responses for static content directly by reading files from disk
- Requests needing dynamic response are handled by **application server**
 - App server parses HTTP requests, generates HTTP response according to the business logic specified by user, sends response back to client via web server
 - Scripting languages may be used for easy text parsing and manipulation
- Application server stores/retrieves app data in a **database**
 - Local database, or remote database accessed by a local database client stub
- Several web application frameworks available to build web applications having such architectures, e.g., Python Django, React etc.

IPC mechanisms



- Unix domain sockets: processes open sockets, send and receive messages to each other via socket system calls
- Message queues: sender posts a message to a mailbox, receiver retrieves message later on from mailbox
- Pipes: unidirectional communication channel between two processes
- Signals: standardized set of messages sent to processes, e.g., pressing Ctrl+C on keyboard sends SIGINT signal to running process
- Shared memory: same physical memory frame mapped into virtual address space of multiple processes in order to share memory
- Different IPC mechanisms are useful in different scenarios

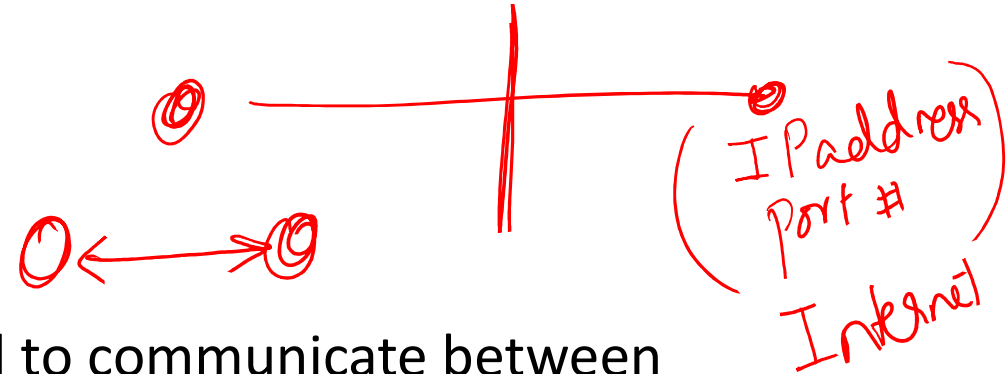


Sockets

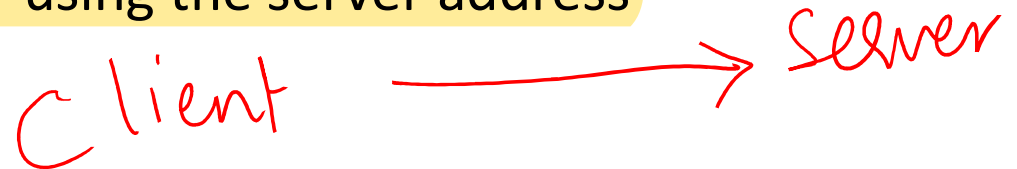


- Sockets = abstraction to communicate between two processes
 - Each process opens socket, and pair of sockets can be connected
 - Client-server paradigm: one process opens socket first (server) and another process connects its socket to the first one (client)
 - One process writes a message into one socket, another process can read it, and vice versa (bidirectional communication)
 - Processes can be in same machine or on different machines
 - If processes on same machine, messages stored temporarily in OS memory before delivering to destination process
 - If processes on different machines, messages sent over network

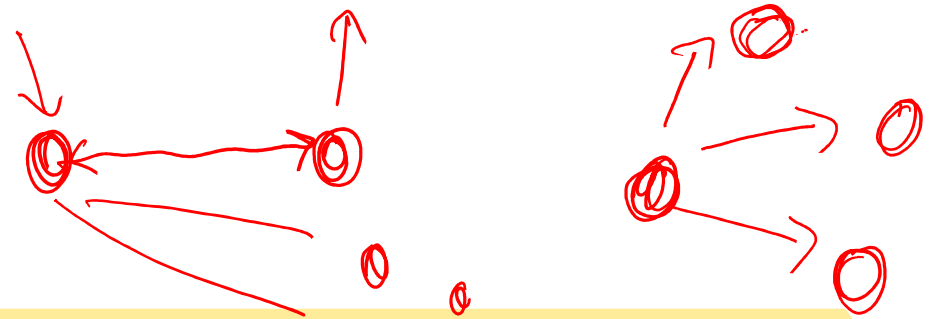
Types of sockets (1)



- **Unix domain (local) sockets** are used to communicate between processes on the same machine
- **Internet sockets** are used to communicate between processes in different machines
- Local sockets identified by a pathname, Internet sockets identified by IP address and port number *string "1.1.1.1"*
- Client and server sockets differentiated by who starts first and who connects later: server sockets started first on a well-known "address", client process connects to server using the server address *globally unique*

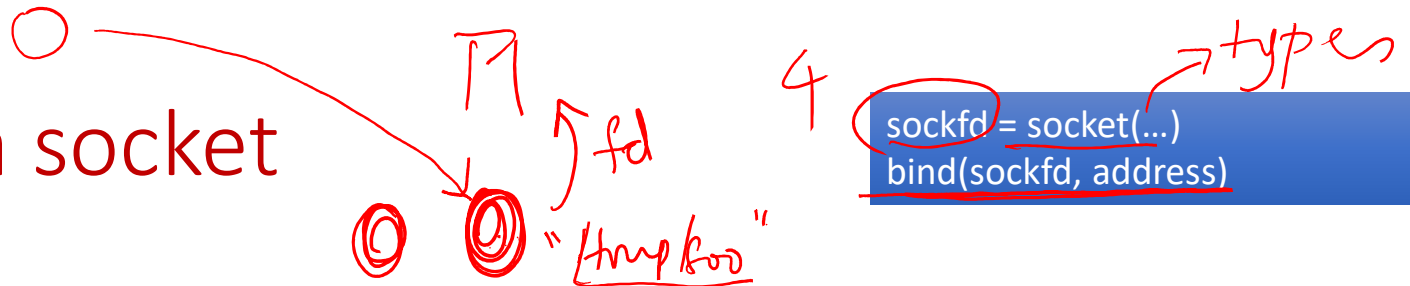


Types of sockets (2)

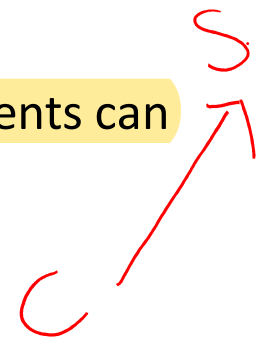


- **Connection-based sockets:** one client socket and one server socket are explicitly connected to each other TCP
 - After connection, the two sockets can only send and receive messages to each other
- **Connection-less sockets:** one socket can send/receive messages to/from multiple other sockets UDP
 - Address of other endpoint can be mentioned on each message
- Type of socket (local or internet, connection-oriented or connection-less) is specified as arguments to system call that creates sockets

Creating a socket



- System call “socket” used to create a socket
 - Takes type of socket as arguments
 - Returns socket file descriptor (similar to file descriptor returned when file is opened)
 - Used as handle for all future operations on the socket
- A socket can optionally bind to an address (pathname or IP address/port number) using “bind” system call
 - Server sockets must always bind to well known address, so that clients can connect
 - Client sockets need not bind, OS can assign temporary address
- Close system call closes a socket when done



Data exchange using connection-less sockets

- Function sendto is used to send a message from one socket to another connection-less socket in another process
 - Arguments: socket fd, message to send, address of remote socket
- Function recvfrom is used to receive a message from a socket
 - Arguments: socket fd, message buffer into which received message is copied, socket address structure into which address of remote endpoint is filled
- When a process receives a message on connection-less socket, it can find out the address of other endpoint which sent message, and use this address to reply back

process

```
Client
sockfd = socket(..)
char message[1024]
sendto(sockfd, message, server_sockaddr, ..)
```

Server

```
Server
sockfd = socket(..)
bind(sockfd, server_address)
recvfrom(sockfd, message, client_sockaddr, ..)
```

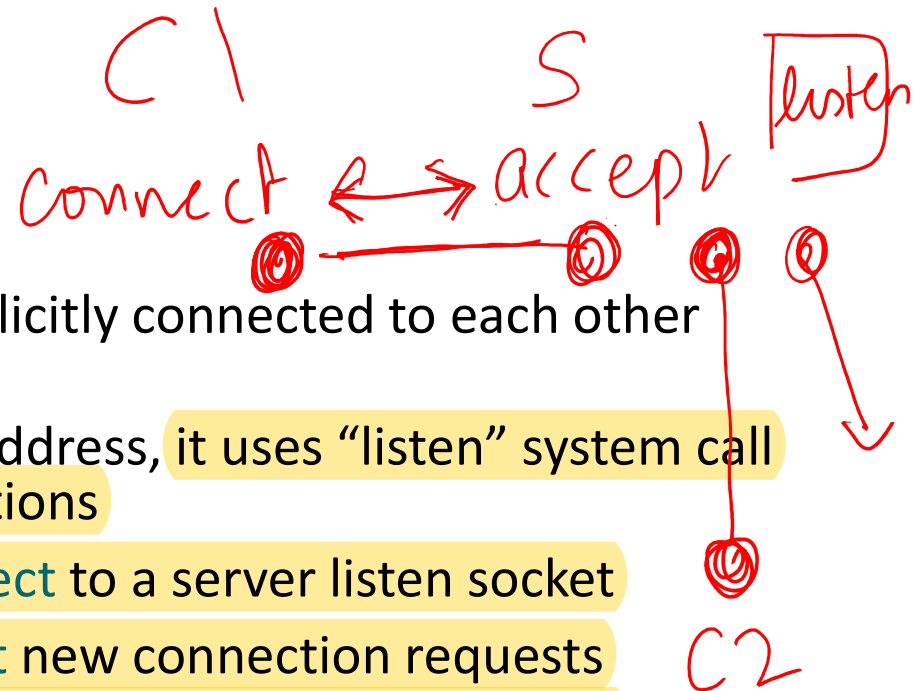
process

name

return

block

Connecting sockets



- Connection-oriented sockets must be explicitly connected to each other before exchanging messages
- After server binds socket to well-known address, it uses “listen” system call to make the socket **listen** for new connections
- Client uses “connect” system call to **connect** to a server listen socket
- Server uses “accept” system call to **accept** new connection requests
 - Returns a new socket file descriptor to communicate exclusively with a client
- At server: one **listen socket** to accept new connections, one **connected socket** for every connected client to send/rcv messages

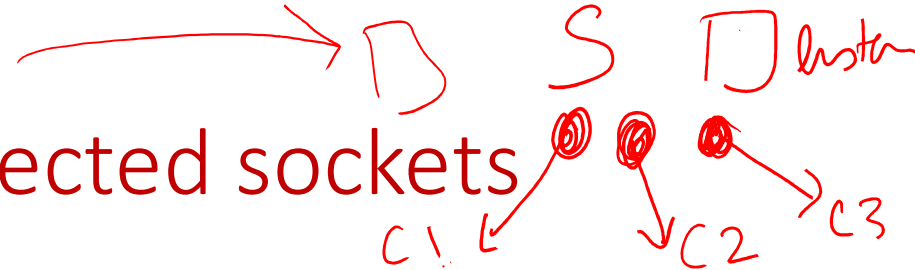
Client

```
sockfd = socket(..)  
connect(sockfd, server_sockaddr, ..)
```

Server

```
sockfd = socket(..)  
bind(sockfd, server_address)  
listen(sockfd, ..) ✓  
newsockfd = accept(sockfd, ..)
```

Data exchange using connected sockets



- After client connects to server, pair of sockets used to exchange data
 - Note that per-client connected socket is used at server, not listen socket
 - System calls send/write used to send message on a connected socket
 - System calls recv/read used to receive message on a connected socket
- Arguments to send/recv: socket fd, message buffer, buffer length, flags
 - Return value is number of bytes read/written or error
 - No need to specify socket address on every message, as connected already

Client


```
sockfd = socket(..)
connect(sockfd, server_sockaddr, ..)
n = send(sockfd, req_buf, req_len, ..)
n = recv(sockfd, resp_buf, resp_len, ..)
```

Server

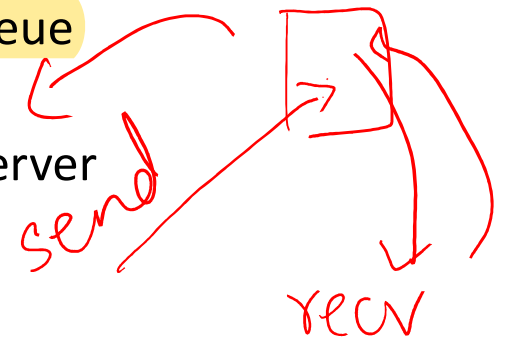
```
sockfd = socket(..)
bind(sockfd, server_address)
listen(sockfd, ..)
newsockfd = accept(sockfd, ..)
n = recv(newsockfd, req_buf, req_len, ..)
n = send(newsockfd, resp_buf, resp_len, ..)
```

Message queues

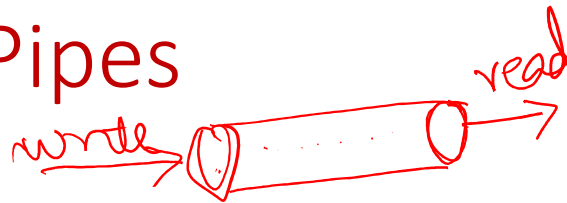
```
msgid = msgget(key, ...)  
msgsnd(msgid, message, ...)  
msgrcv(msgid, message, ...)
```

handle 

- Message queues used for exchanging messages between processes
 - Open connection to message queue identified by a “key”, get a handle
 - Sender opens connection to message queue, sends message
 - Receiver opens connection to message queue, retrieves message later on
 - Message buffered within message queue / mailbox until retrieved by receiver
- Example: IPC in web application using message queues
 - Web server posts dynamic HTTP requests into message queue
 - App server retrieves requests and processes them
 - App server posts responses into message queue for web server



Pipes

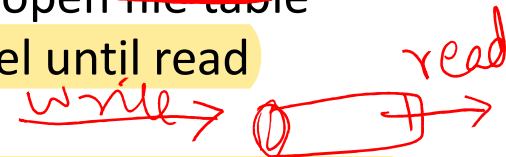


5, 6

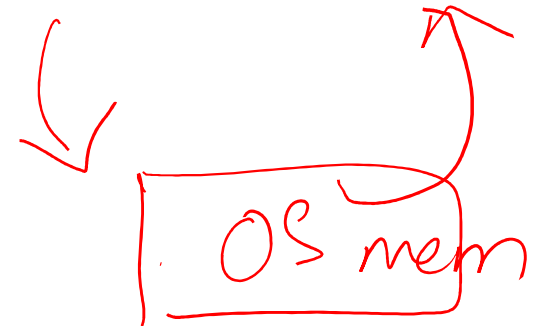
```
int fd[2];  
pipe(fd) //anonymous  
read(fd[0], message, ..)  
write(fd[1], message, ..)
```

```
mkfifo(name, ..)  
fd0 = open(name, O_RDONLY)  
read(fd0, message, ..)  
fd1 = open(name, O_WRONLY)  
write(fd1, message, ...)
```

- Pipe is a **unidirectional FIFO channel** into which bytes are written at one end, read from other end
 - System call "pipe" creates a pipe channel, with two file descriptors for two endpoints
 - ~~Pipe file descriptors point to pipe channel data structure via open file table~~
 - Data written into pipe is stored in a buffer of the pipe channel until read
 - Bi-directional communication needs two pipes
- **Anonymous pipes only available for use within process and its children**
 - Pipe file descriptors point to same pipe structure in parent and child
- How to use pipes between unrelated processes? **Named pipes**
 - Named pipes opened with a pathname, accessible across processes
 - One process accesses read end of pipe, another opens write end



Blocking vs. non-blocking IPC



- Same high level concept across sockets, pipes, message queues
 - Sender sends message, temporarily stored in some memory inside OS
 - Receiver retrieves message later on from temporary OS memory
- Send/receive system calls can block
 - Sender can block if temporary buffer is full
 - Receiver can block if temporary buffer is empty
- Possible to configure IPC to be non-blocking using syscalls
 - Send/receive will return with error instead of blocking

default

read = 0