

Inter-process communication (IPC)

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 application architecture

- 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**
- 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
- **Shared memory**: same physical memory frame mapped into virtual address space of multiple processes in order to share memory
- **Signals**: specific messages via kill system call
- 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 (Internet Protocol) address and port number
- 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

Types of sockets (2)

- **Connection-based sockets**: one client socket and one server socket are explicitly connected to each other
 - 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
 - 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
- In this course: Unix domain sockets, connection-less

Creating a socket

```
sockfd = socket(...)  
bind(sockfd, address)
```

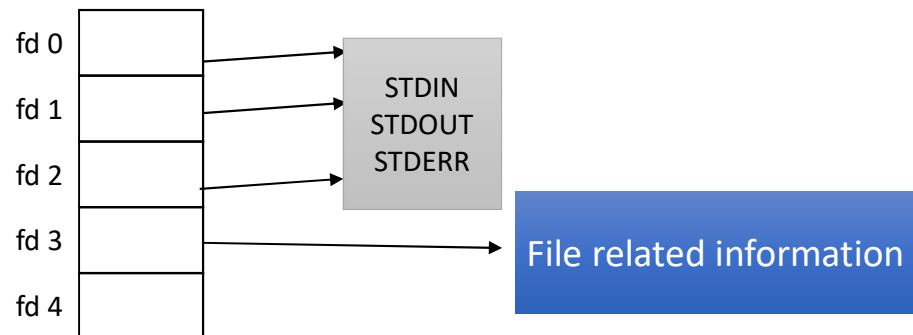
- System call “socket” used to create a socket
 - Takes type of socket as arguments
 - Returns **socket file descriptor** (similar to file descriptor when file is opened)
 - Used as handle for all future operations on the socket
- A socket can optionally **bind to an address** (pathname for Unix domain sockets or IP address/port number for Internet sockets) using “bind” system call
 - Server sockets 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

The concept of file descriptors

```
fd = open("/home/foo/a.txt")
char buf[64]
read(fd, buf, 64)
buf[0] = ...
write(fd, buf, 64)
```

- Many IPC mechanisms like sockets return a file descriptor, which is simply an integer “handle” to access a file or socket or pipe
- PCB of process contains list of all open files in an array
- On file open, new entry is created in array, new index returned
- All future system calls (read, write) will be given the file descriptor as one of the arguments

File descriptor table
(list of open files of a process)
(part of PCB of process)



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, and use this address to reply back

Client

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

Server

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

Message queues

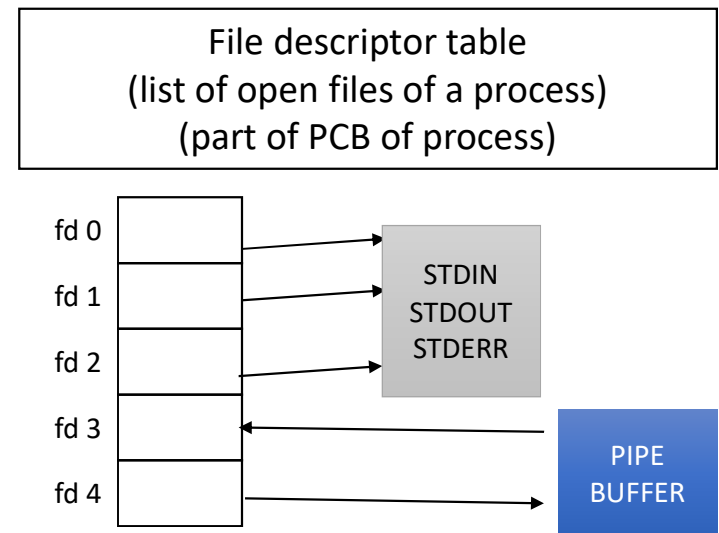
```
msgid = msgget(key, ...)  
msgsnd(msgid, message, ...)  
msgrcv(msgid, message, ...)
```

- 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

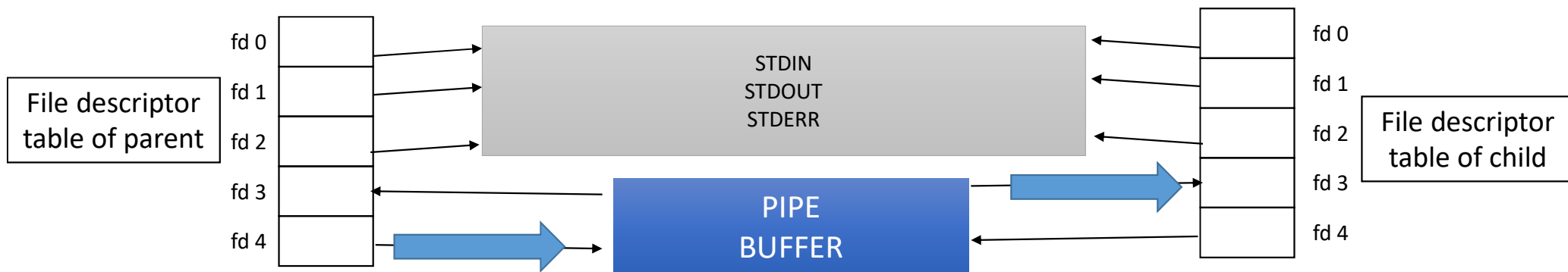
- Pipe is a **unidirectional FIFO channel** into which bytes are written at one end, read from other end
- System call “pipe” creates a pipe channel, with two file descriptors for endpoints, returns 2 integers
- One file descriptor used to write into pipe, one to read from pipe
- Data written into pipe is stored in a buffer of the pipe channel until read
- Bi-directional communication needs two pipes

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



Anonymous pipes

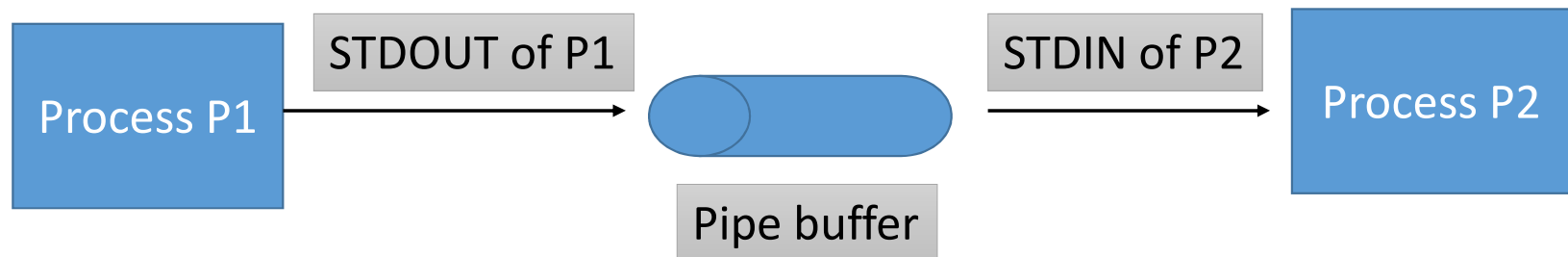
- **Anonymous pipes** (using pipe system call) only available for use within process and its children
- Open pipe before fork, so pipe file descriptors shared between parent and child, point to same pipe structure
- One of parent/child closes read end, other closes write end
- Pipe file descriptors used to read/write messages between parent/child



Pipes in shell commands

```
$cat foo.txt | grep something
```

- How does shell run commands with pipes (output of one command given as input to another command)?
- Shell opens a pipe, shared with child processes that run commands
- Shell duplicates stdout of first child to write end of pipe, read end of pipe to stdin of second process
- Processes must close file descriptors they are not using



Named pipes

- 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
- Named pipe also provides uni-directional communication
- Writing to pipe with no reader open will throw an error

Reader

```
mkfifo(name, ..)  
fd = open(name, O_RDONLY)  
read(fd, message, ..)
```

Writer

```
fd = open(name, O_WRONLY)  
write(fd, message, ...)
```

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

Shared memory

```
shmid = shmget(key, ..)  
char *data = shmat(shmid, ..)
```

- Processes in a system do not share any memory by default
 - Child process gets copy of parent memory image, modifies independently
- Shared memory: a way for two processes to share memory
 - Same memory appears in memory image of multiple processes
 - Shared memory **segment** identified by a **unique key**
 - Process can request to map or “attach” a specific shared memory segment into its memory image by using key
- Processes may need extra mechanisms for coordination besides shared memory
 - E.g., how does one process know when another process has modified shared memory?

Signals

- Signals are also a type of IPC
- But can only send limited amount of information
- Signals can be used with other IPC mechanisms
- Example: P1 places data in shared memory, signals P2 that data is ready
- User-defined signals can be used to avoid terminating process