Design and Engineering of Computer Systems

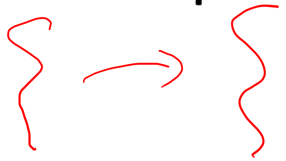
Lecture 27: Inter-process communication

Mythili Vutukuru

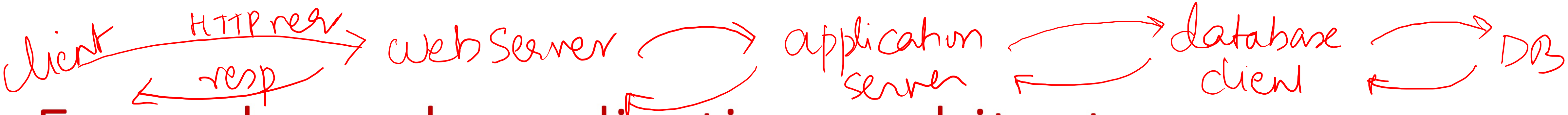
IIT Bombay

Multi-process application design

- Previous lecture: how multiple threads in a process work together
- This lecture: how multiple processes in one machine work together
- 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



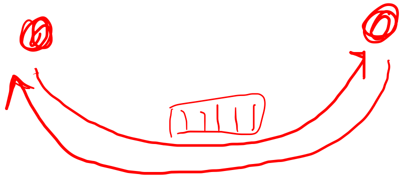
- How do processes in a system communicate with each other?
- Inter-process communication (IPC) mechanisms, available via operating system syscalls, allow processes to exchange information



Example: web application architecture

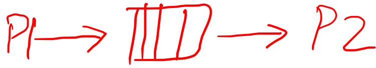
- Example: web application architecture composed of multiple processes
- Web server process handles HTTP requests/responses via TCP/TLS sockets
  - Written in a language like C/C++ for high performance
  - Returns responses for static content directly by reading files from disk
- Requests needing dynamic 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
  - Developer freed from responsibility of setting up processes and IPC

IPC mechanisms

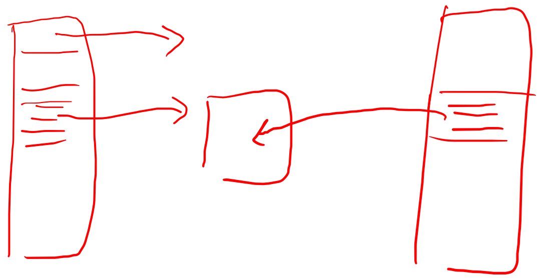


- Unix domain sockets: processes open sockets, send and receive messages to each other via socket system calls
  - Transmitted messages are stored in socket buffers, read by receiving process, no transmission over the network involved for local sockets
- Message queues: sender posts a message to a mailbox, receiver retrieves message later on from mailbox

- Pipes: unidirectional communication channel between two processes



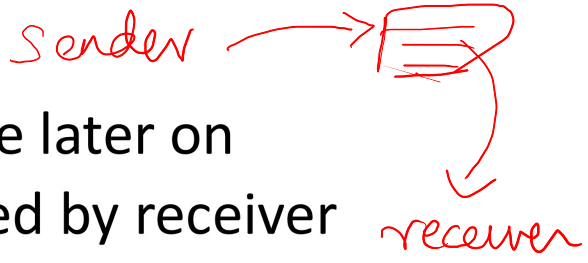
- Signals: standardized set of messages sent to processes, e.g., pressing Ctrl+C on keyboard sends SIGINT signal to running process
- Shared memory: same physical memory frame mapped into virtual address space of multiple processes in order to share memory
- Different IPC mechanisms are useful in different scenarios

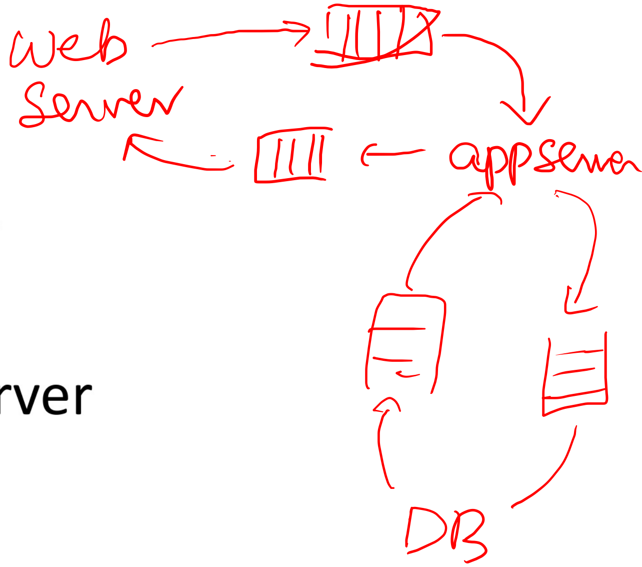


Message queues

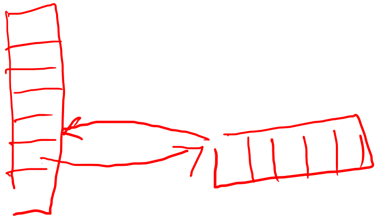
```
msgid = msgget(key, …)
msgsnd(msgid, message, …)
msgrcv(msgid, message, …)
```

- Message queues used for exchanging messages between processes
  - 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
  - App server posts database requests into message queue
  - Database client queries database and posts response to app server





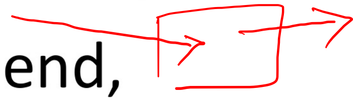
Pipes

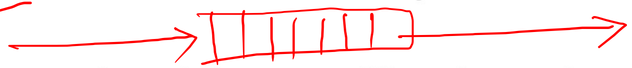


```
int fd[2]
pipe(fd) //anonymous
read(fd[0], message, ..)
write(fd[1], message, ..)
```

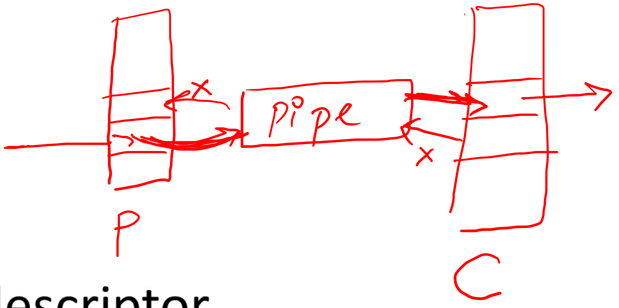
```
mkfifo(name, ..)
fd0 = open(name, O_RDONLY)
read(fd0, message, ..)
fd1 = open(name, O_WRONLY)
write(fd1, message, ...)
```

- 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



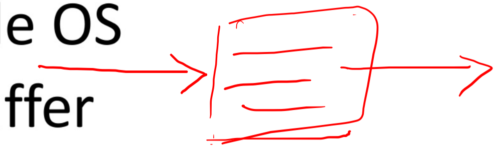


- Anonymous pipes created by “pipe” only available for use within process
  - Example: Parent P opens pipe, P forks child C, P closes read end, C closes write end
  - P writes into write end of pipe, C reads from read end of pipe
- How to use pipes between unrelated processes? Named pipes
  - Named pipes opened with a pathname, accessible across processes
  - One process accesses read end of pipe, another opens write end
  - Messages sent via write end file descriptor are read via read end file descriptor



Blocking vs. non-blocking IPC

- Same high level concept across sockets, pipes, message queues
  - Sender sends message, temporarily stored in a buffer inside OS
  - Receiver retrieves message later on from temporary OS buffer



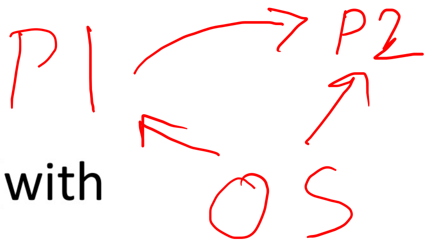
- Send/receive system calls can block
  - Sender can block if temporary buffer is full
  - Receiver can block if temporary buffer is empty

- Possible to configure IPC to be non-blocking (e.g., set socket options)
  - Send/receive will return with error instead of blocking

Signals

- Signals: a way to send notifications to processes, used between processes, or between OS and process
- Standard signals available in operating systems, for example:
  - Signal to interrupt (SIGINT) via Ctrl+C
  - Signal to kill (SIGKILL), signal to stop (SIGSTOP), user defined signals, …
- How are signals sent?

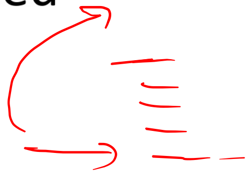
kill (process ID, signal)



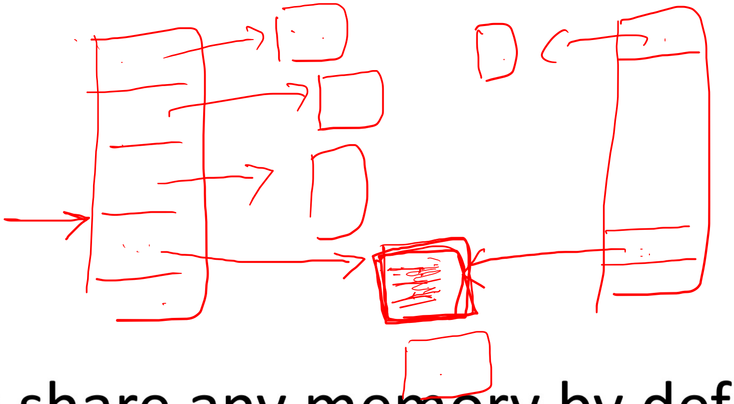
  - System call "kill" can be used to send any signal from one process to other (with some restrictions for isolation and security)
  - Signals can also be generated by OS, e.g., when it handles Ctrl+C keyboard event
- Signal handler: function executed by process to handle a signal

signal — handler

  - When signal arrives, process execution flow interrupted, signal handler executed
  - Default signal handlers exist (e.g., terminate when Ctrl+C), can be overridden



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 physical memory
  - Same physical memory frame mapped into virtual address space of multiple processes (possibly at different virtual addresses)
  - Page table entries in page tables of different processes point to same frame
  - Shared memory "segment" identified by a unique key
  - Process can request to map or "attach" a specific shared memory segment into its virtual address space by using key
  - Once mapped, shared memory segment can be used like any other virtual memory
- Processes may need extra mechanisms for coordination besides shared memory
  - E.g., when producer writes into shared memory, how does consumer know?
  - E.g., how to avoid concurrent/inconsistent updates to shared data in shared memory?

Summary

- In this lecture:
  - Mechanisms for inter-process communication
- Programming exercise: write code using any of the IPC mechanisms discussed in this lecture. Send a message from one process and receive it in another process.
- Programming exercise: write a simple program with a new signal handler for Ctrl+C signal (SIGINT), which will print a message before terminating.