CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Announcements

- Readings from Silberchatz [3nd chapter]
- Project 1 is out
- Homework 2 would be out next week

Process Context Switch

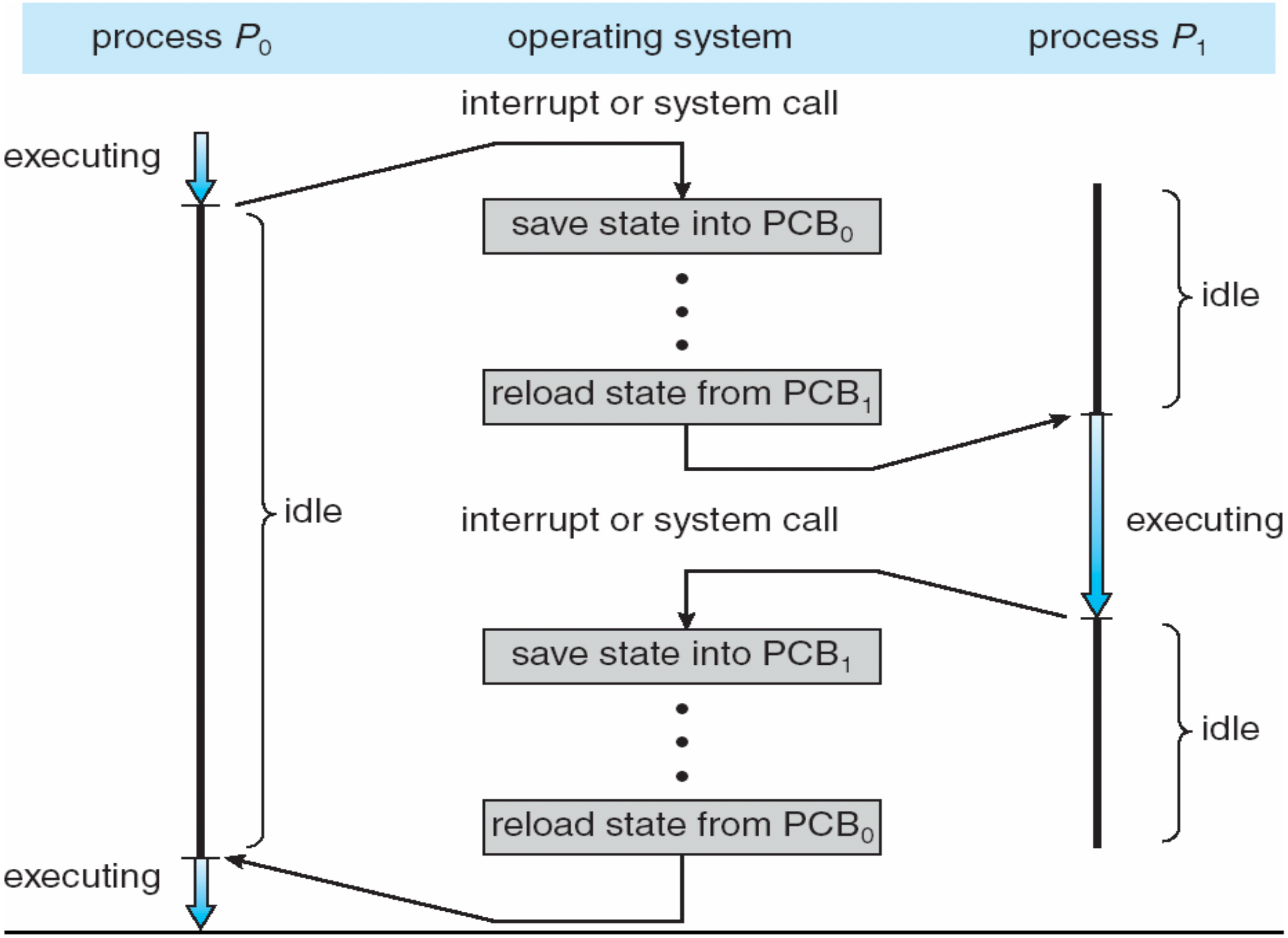
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

- Context of a process represented in the PCB

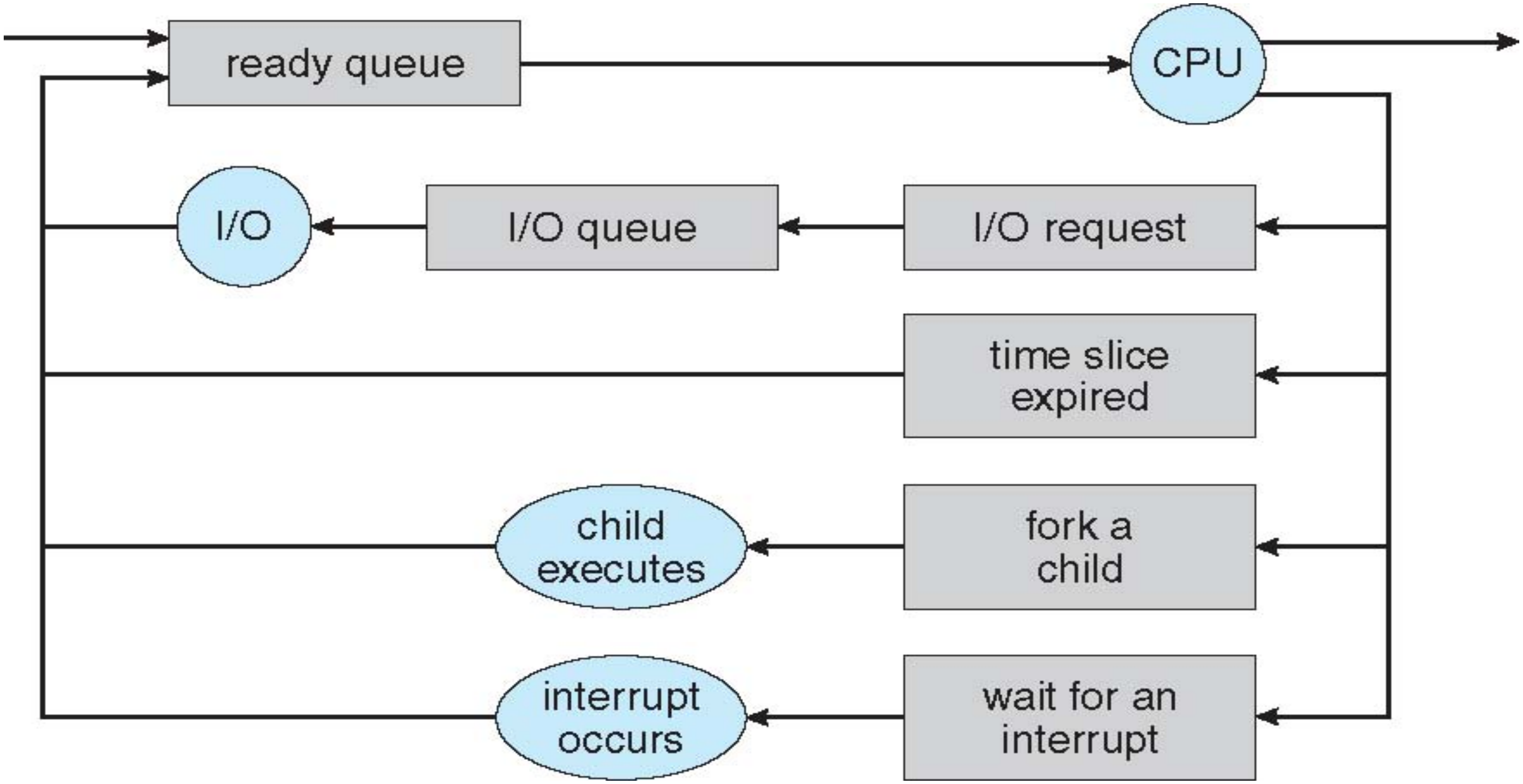
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

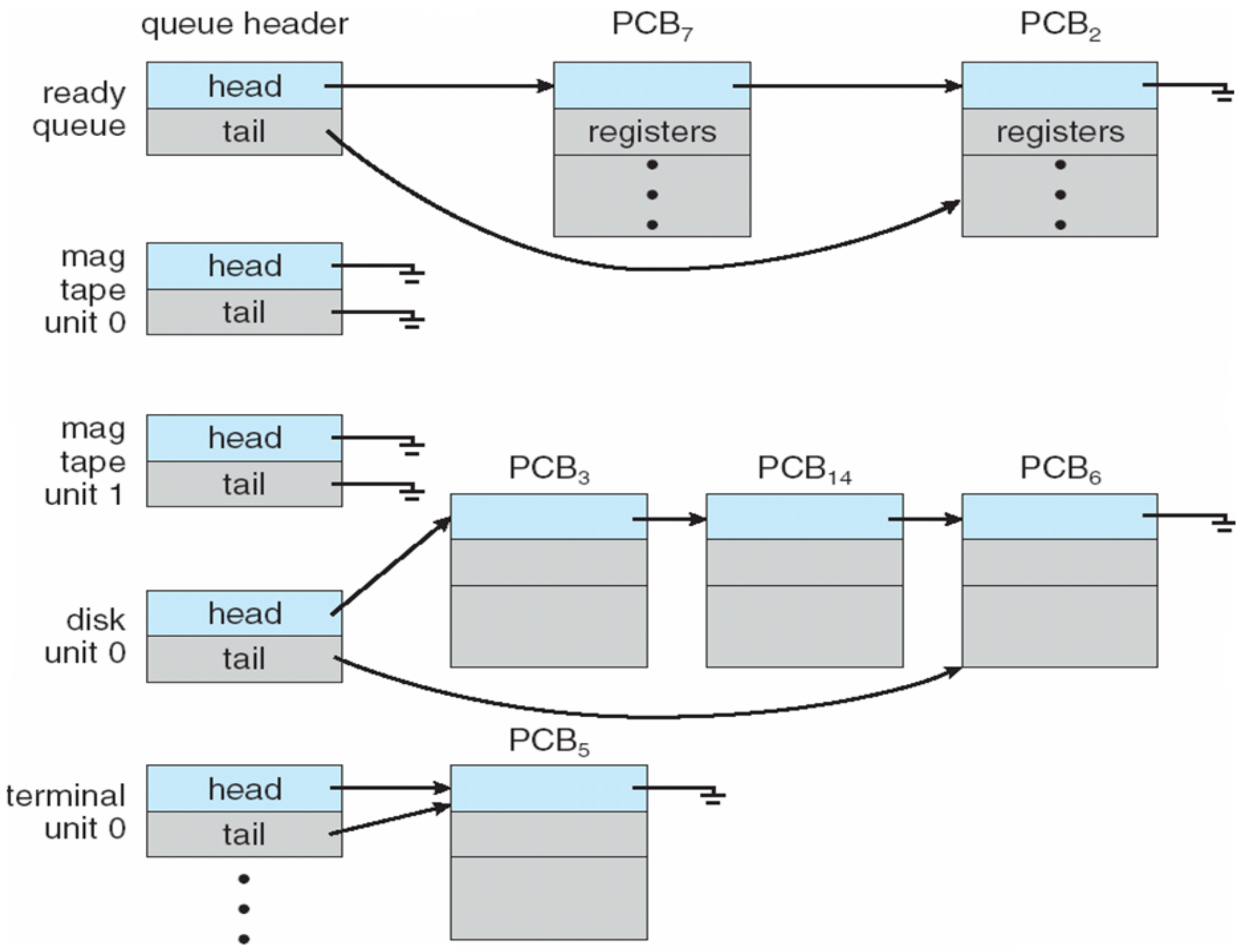
Process Context Switch



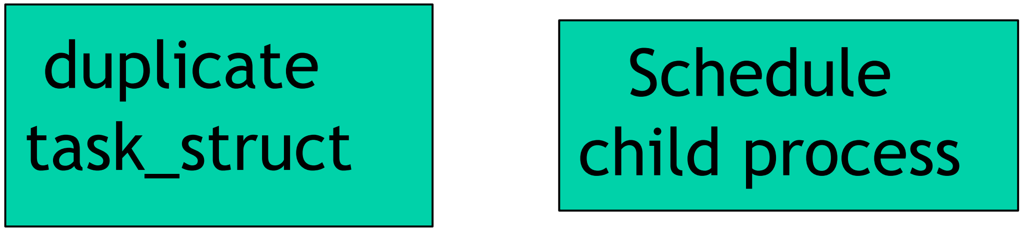
Process Scheduling



Process Queues



Lets take a kernel dive to study the process data structure and fork() system call



duplicate task_struct

Schedule child process

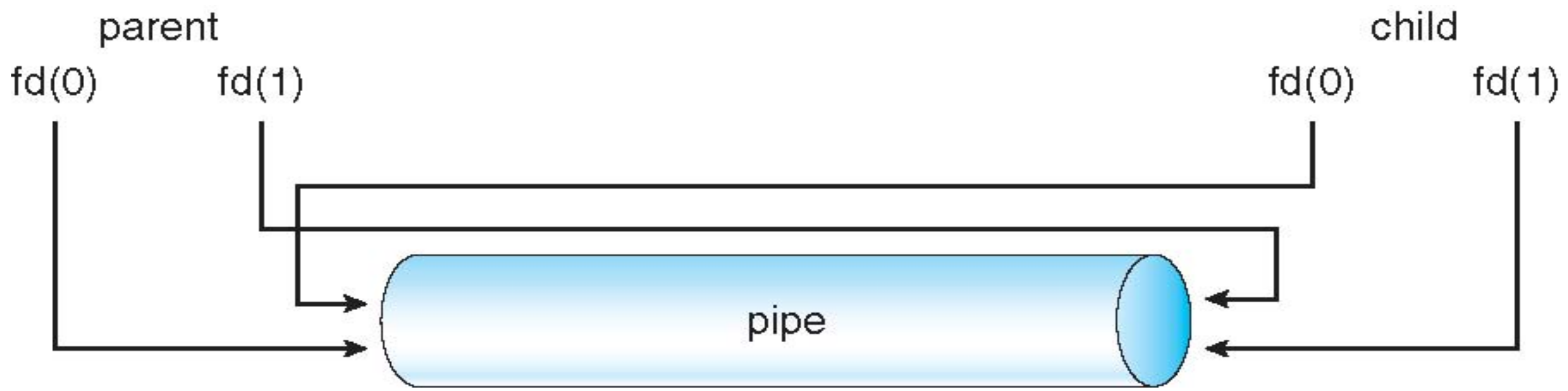
Inter-process communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Convenience
- Cooperating processes need **interprocess communication (IPC)**

IPC mechanisms

- Pipes (unidirectional)
  - Anonymous pipes (we have seen this)
  - Named pipes (FIFOs) (communication between processes that are not child and parent) (makes use of semaphores)
- Shared memory
  - Share a chunk of memory for read/write between processes
- Mmaped files
  - Communication through a file mapped to memory
- Message passing
  - Network sockets
- Signals
  - Slightly weird way of IPC

Pipes (unidirectional data transfer)



- Anonymous pipes
  - Defined within a process
  - Communication between parent and child processes
- Named pipes (FIFO)
  - Allows communication between processes which are not child/ parent
  - Linux utilty: mkfifo

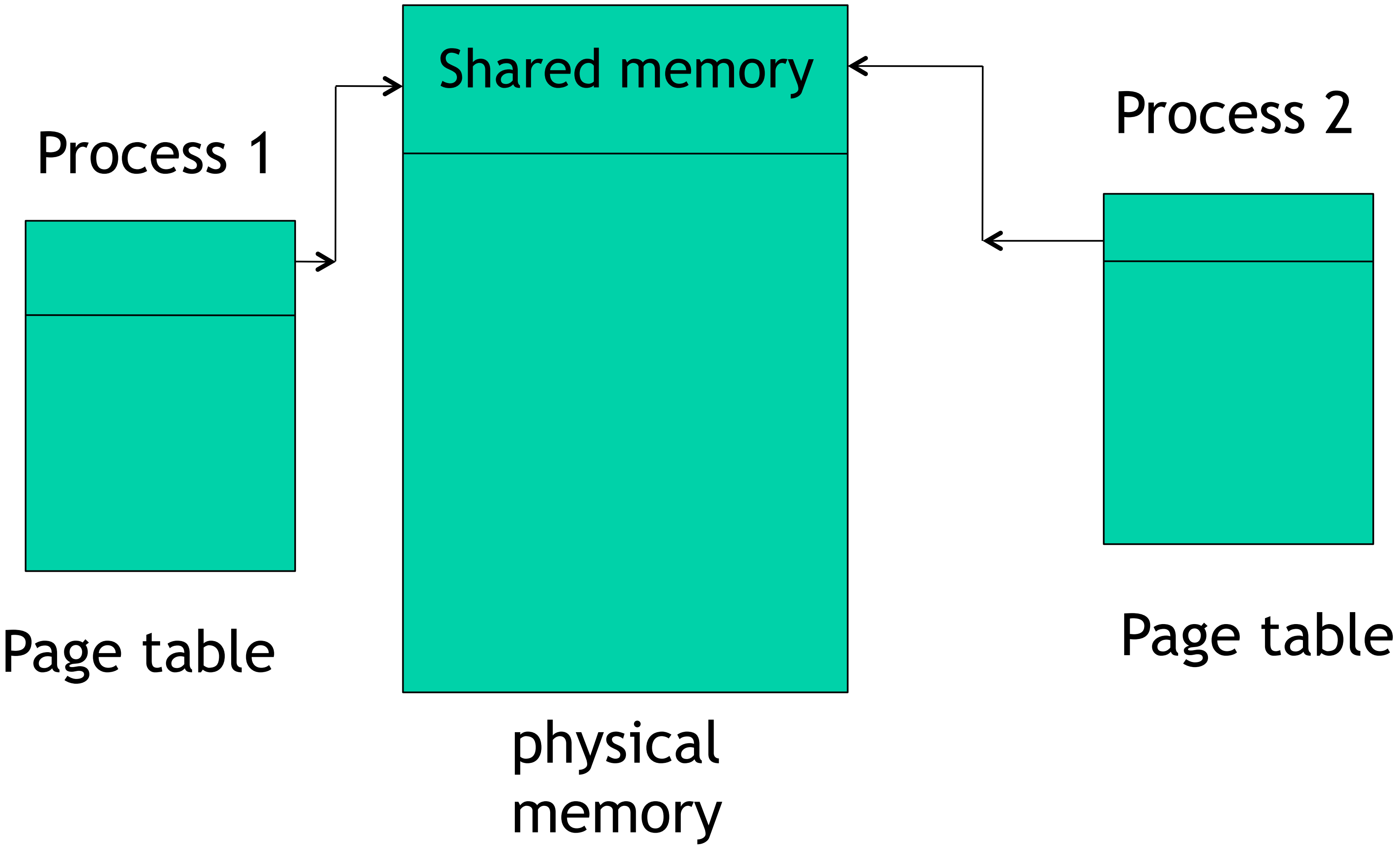
mkfifo

- `int retval = mkfifo(“path to the pipe”, permissions)`
  - Creates a pipe
  - Use this pipe for reading writing, just like a file

- `int fid = open(“path to file”, O_RDWR);`

- use `read(fid, char *, length)` and
- `write(fid, char *, length)` to read and write from the pipe

Shared memory



– Linux utilty: shmget, shmat, shmdt

Shared memory

- POSIX Shared Memory
  - Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR
  | S IWUSR);
```

  - Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

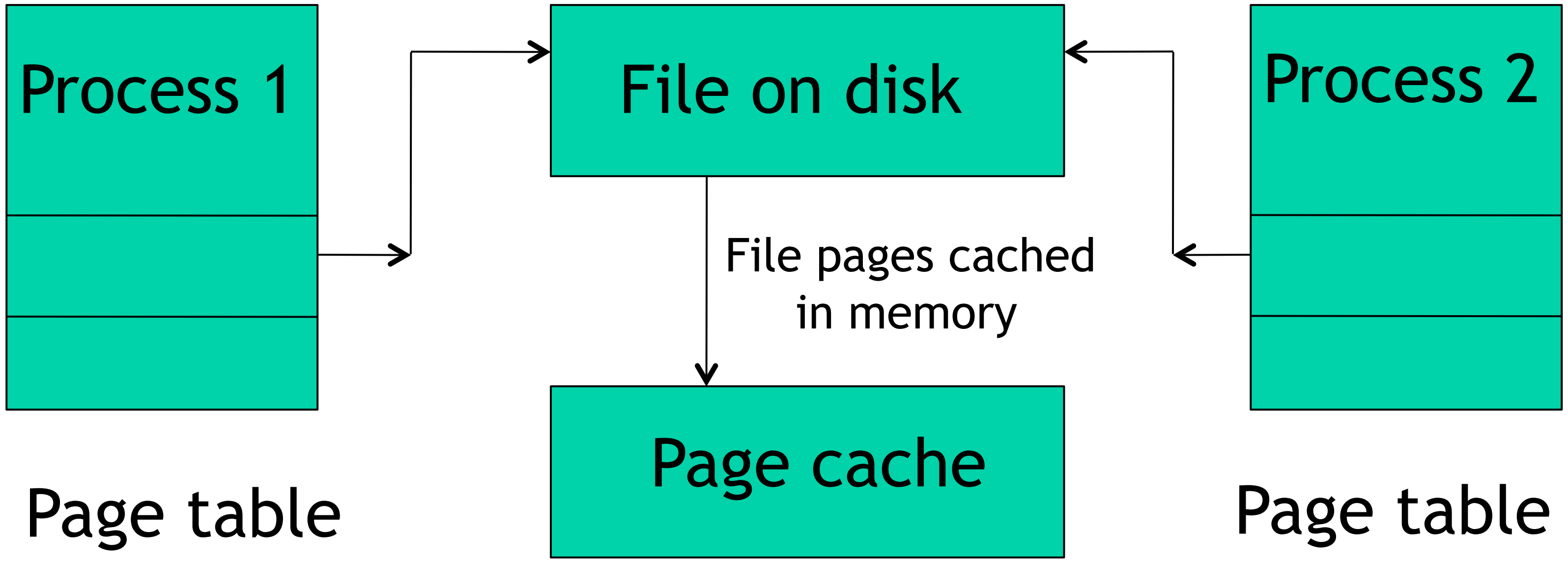
  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared
  memory");
```

  - When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

Memory mapped files (awesome hack!)



- Linux utility
  - mmap()

Mmap()

- First create a file of all zeros

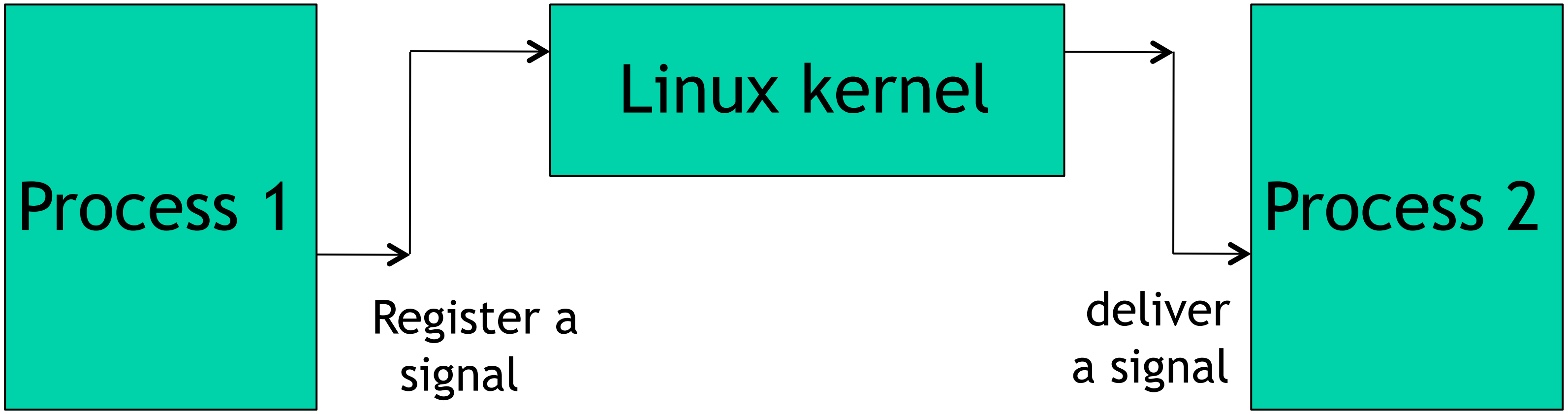
  dd if=/dev/zero of="your file" bs=1024 count = 1024

  - Creates a file of size 1M

- Open that file

- Memory map that file
  - mmap(start_addr, length, protocol (PROT_READ|PROT_WRITE), flags (MAP_SHARED), <fd of the open file>, offset)
  - Returns a pointer to read and write from the mmaped region

POSIX Signals



Process 1

Register a
signal

Linux kernel

deliver
a signal

Process 2

❖

Name	*Description*	*Default Action*
`SIGINT`	Interrupt character typed	terminate process
`SIGQUIT`	Quit character typed (`^\`)	create core image
`SIGKILL`	`kill -9`	terminate process
`SIGSEGV`	Invalid memory reference	create core image
`SIGPIPE`	Write on pipe but no reader	terminate process
`SIGALRM`	`alarm()` clock ‘rings’	terminate process
`SIGUSR1`	user-defined signal type	terminate process
`SIGUSR2`	user-defined signal type	terminate process

signals

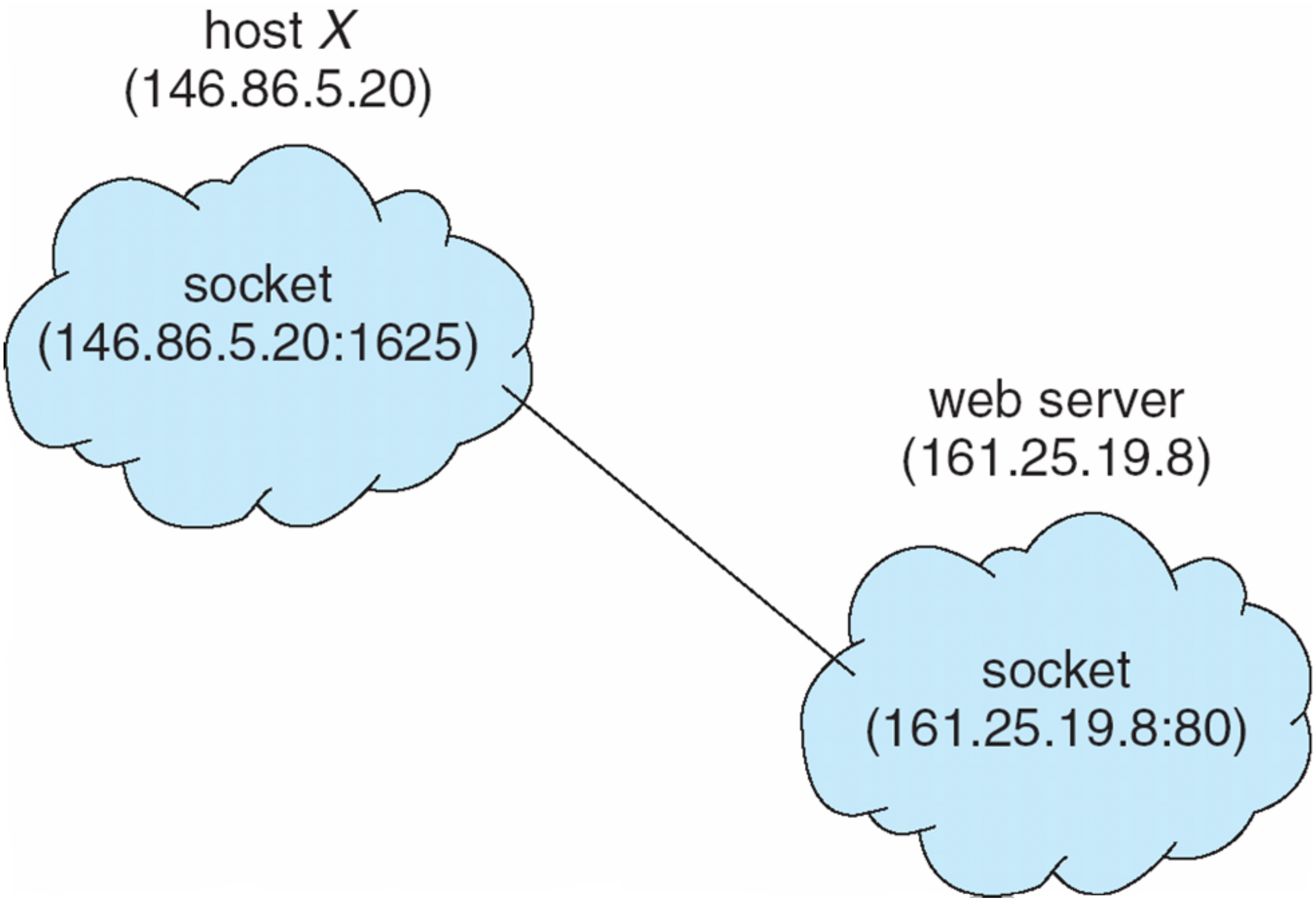
- int kill(pid_t pid, int signo);
  - Send a signal to a process with a process id

- signal(<signal name>, <pointer to handler>)
  - Handle a maskable signal in your code

Message Passing Using Sockets

- A **socket** is defined as an *endpoint for communication*
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets

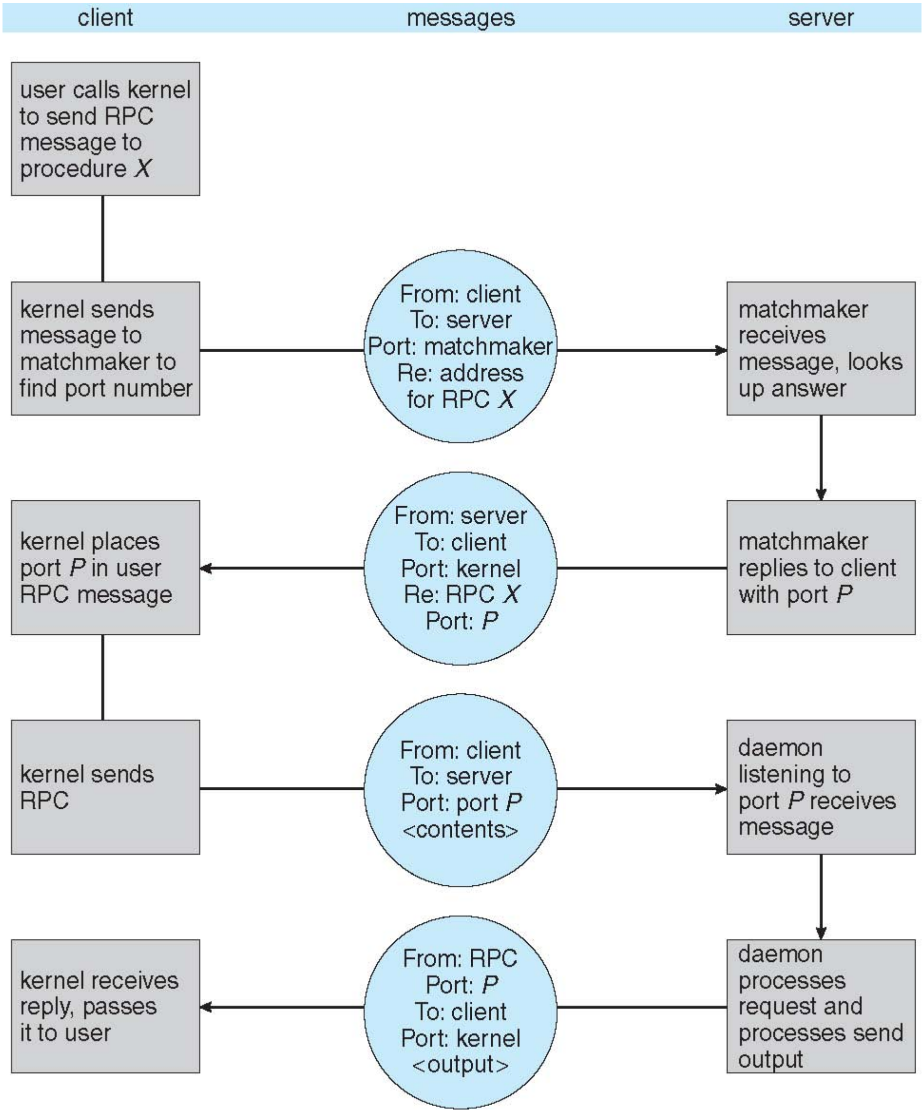
Message Passing Using Sockets



Concept of Remote Procedure calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and *marshalls* the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

Execution of RPC


client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client
To: server
Port: matchmaker
Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server
To: client
Port: kernel
Re: RPC *X*
Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client
To: server
Port: port *P*
<contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC
Port: *P*
To: client
Port: kernel
<output>
kernel receives reply, passes it to user

An in-class discussion