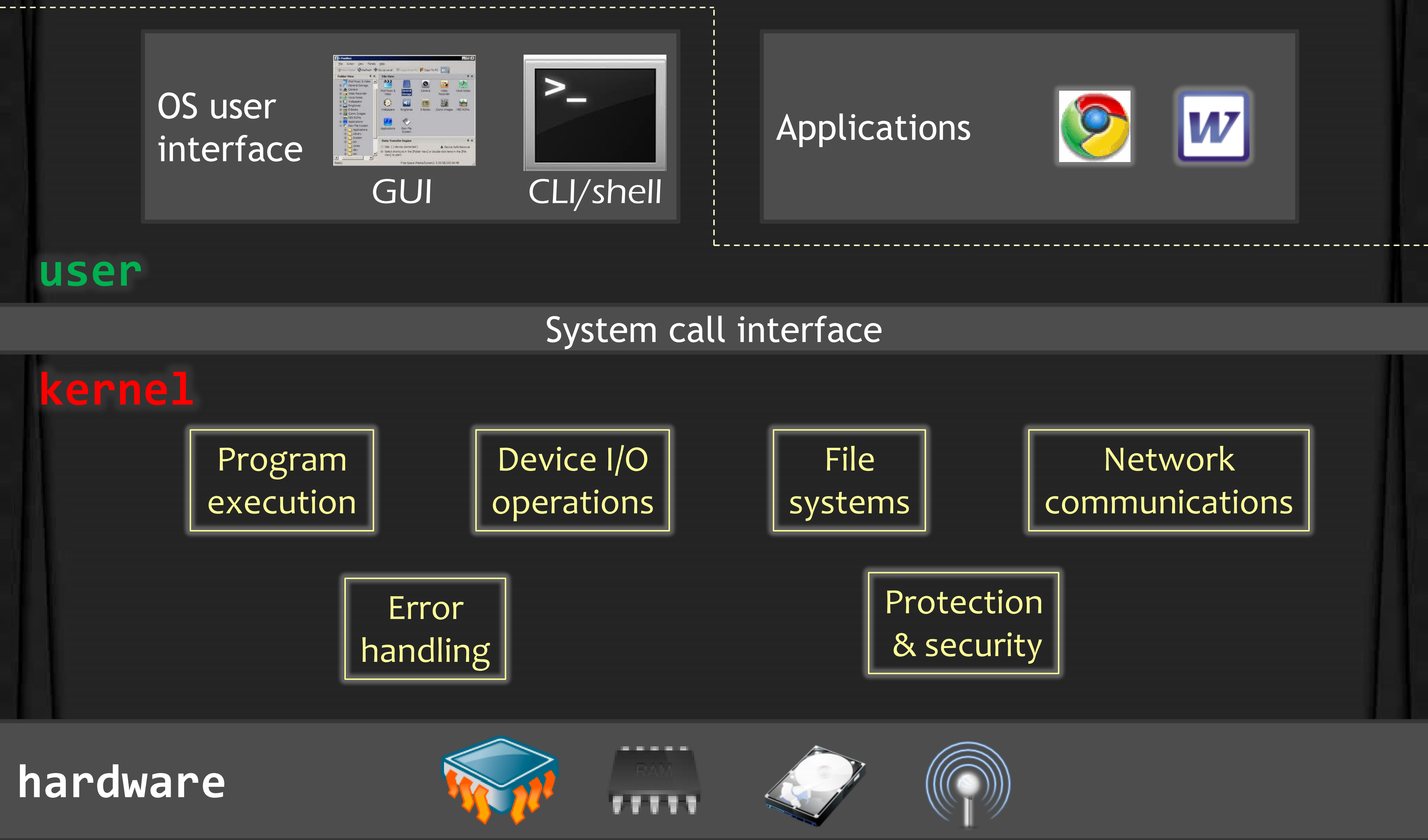
THE UNIVERSITY of NEW ORLEANS

CSCI 4621
INTRO CYBERSECURITY

OS Review: Processes & Address Spaces

Vassil Roussev
vassil@cs.uno.edu

Operating System Services



Core OS Services

- Program execution
  - Load program code from HDD into RAM
  - Run/execute the code
  - End execution
- Device I/O
  - Provide access to various devices
    - USB, ATA, PCIe, etc.
- File systems
  - Provide (abstract) access to storage devices
    - HDD, CD, DVD, B-Ray, etc.

Core OS Services (2)

- Network & inter-process communications
  - Allow exchange of data b/w processes and via the network
- Protection & security
  - Provide means to manage and enforce security policies
- Error handling
  - Detect and recover from hardware and software errors
  - System debugging

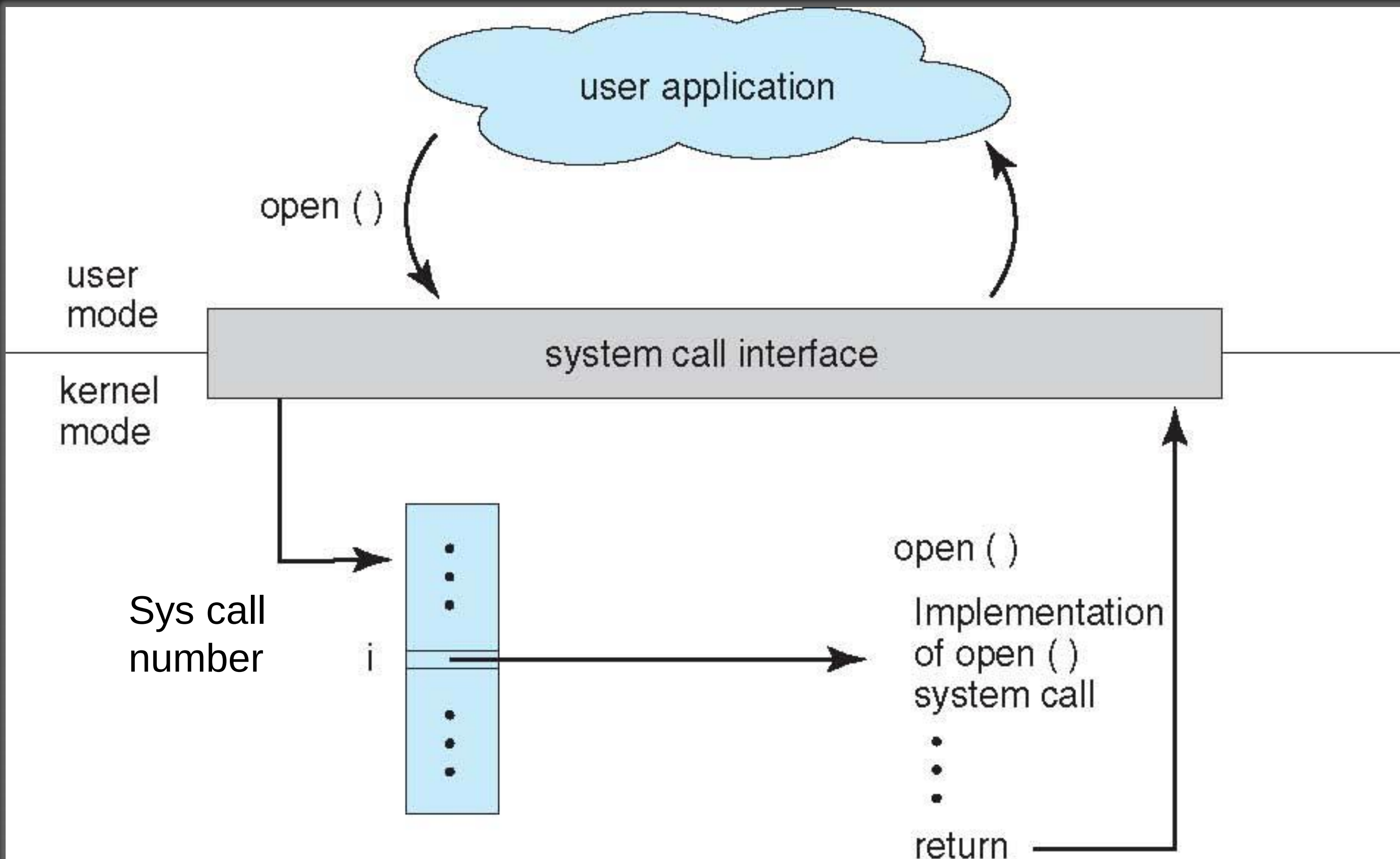
Core OS Services (3)

- Accounting & resource management
  - Typically split across other core services
  - Keep track of resource usage & statistics
  - Manage & enforce resource restrictions

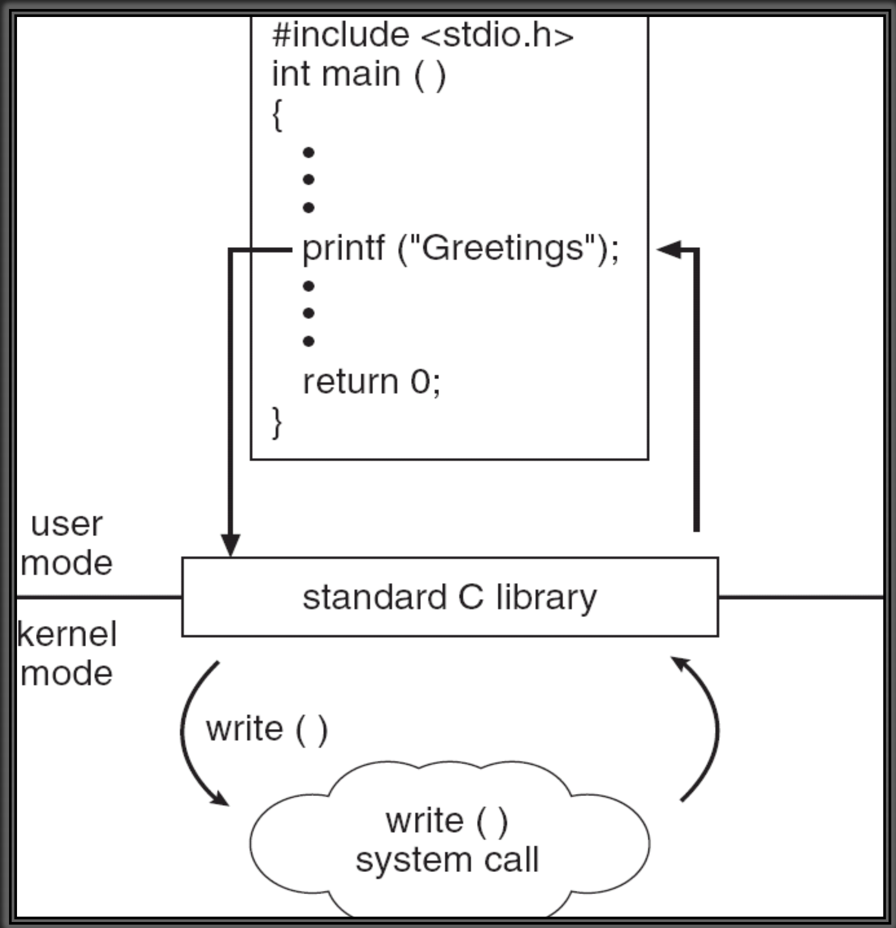
UI Services

- These are (system) applications that exercise the system call interface
  - *They can be replaced with no ill effects*
- UI
  - *GUI: graphical interface to*
    - invoke/terminate programs,
    - manage user data,
    - manage access control, etc.
  - *CLI: command line interface*
    - textual interface to access OS services
    - allows scripting/automation

API – System Call – OS Relationship



Sys call example: standard C library

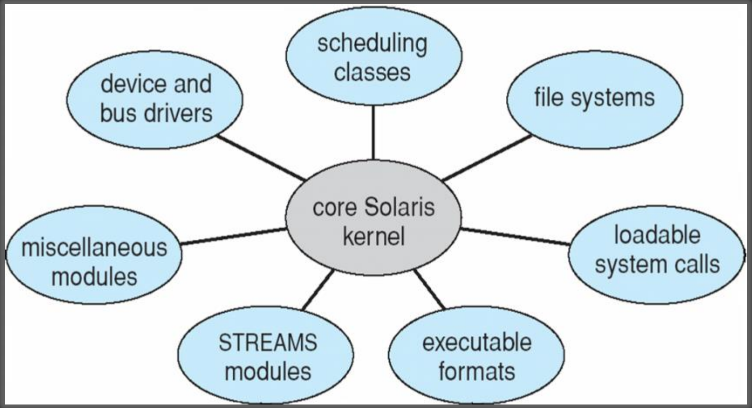


Sys call interfaces: Windows & Unix (POSIX)

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Kernel Modules

- KM allow (to various extents) modular extension of the kernel
  - Object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel

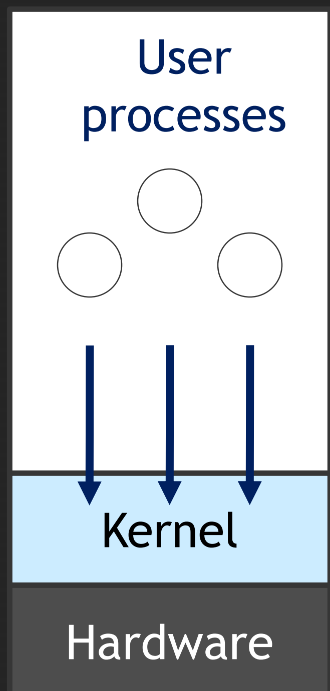


e.g.: **Solaris**

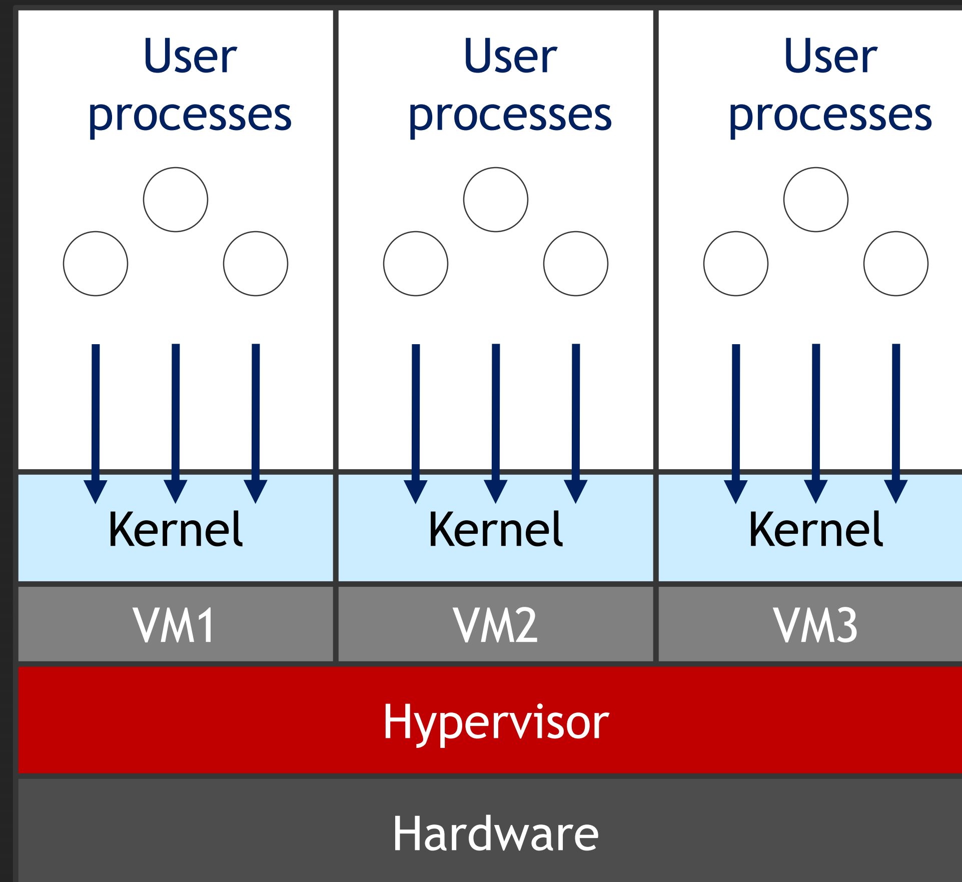
Virtual Machines

- A ***virtual machine*** provides an interface identical to the underlying bare hardware.
  - The hypervisor creates the illusion that a process has its own CPU, RAM, I/O devices.
  - Each guest provided with a (virtual) copy of underlying computer.
  - Hardware support is needed for VMs to run efficiently

Virtual Machines

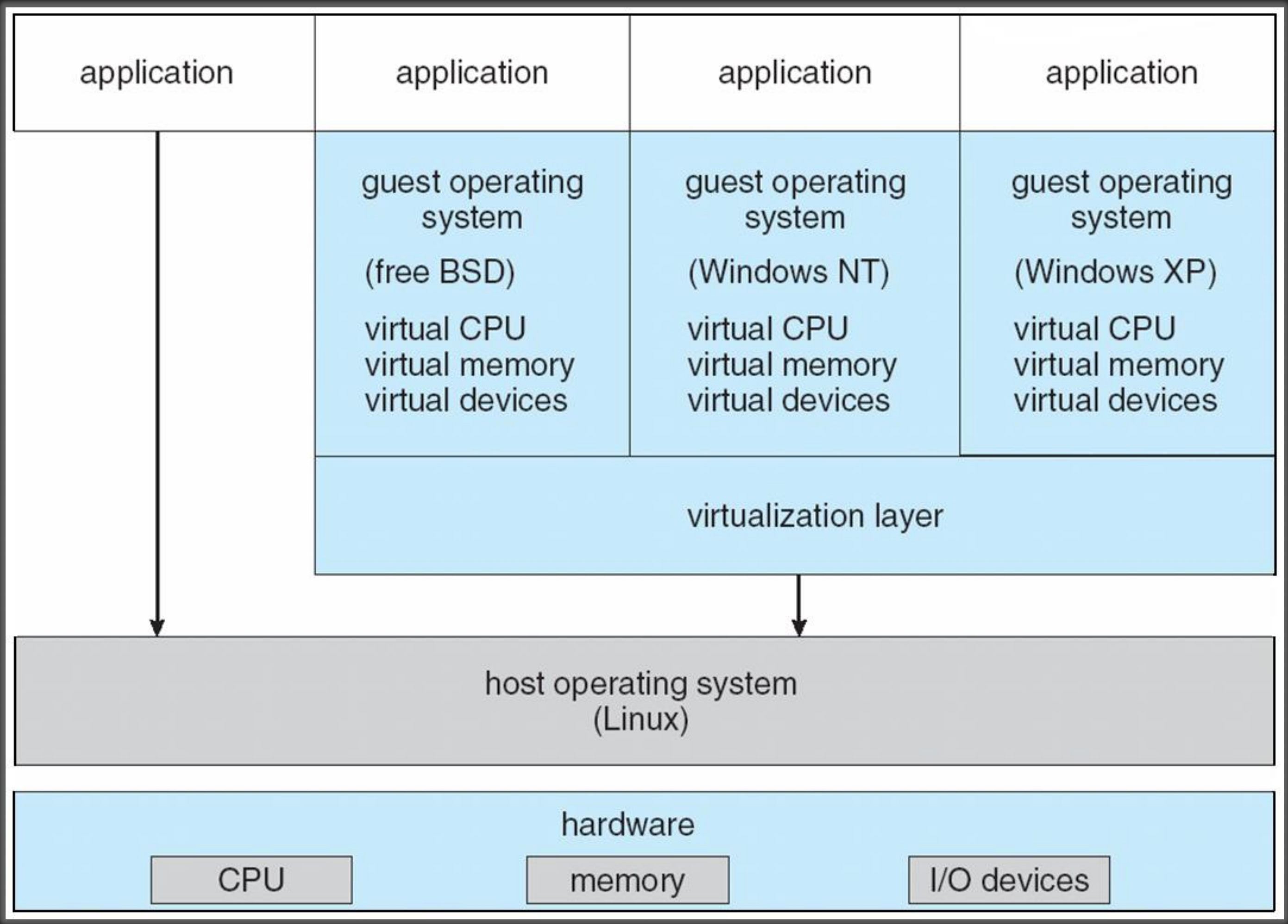


Non-virtualized machine:
Single OS instance is present



Virtualized machine:
Multiple OS instances may be present

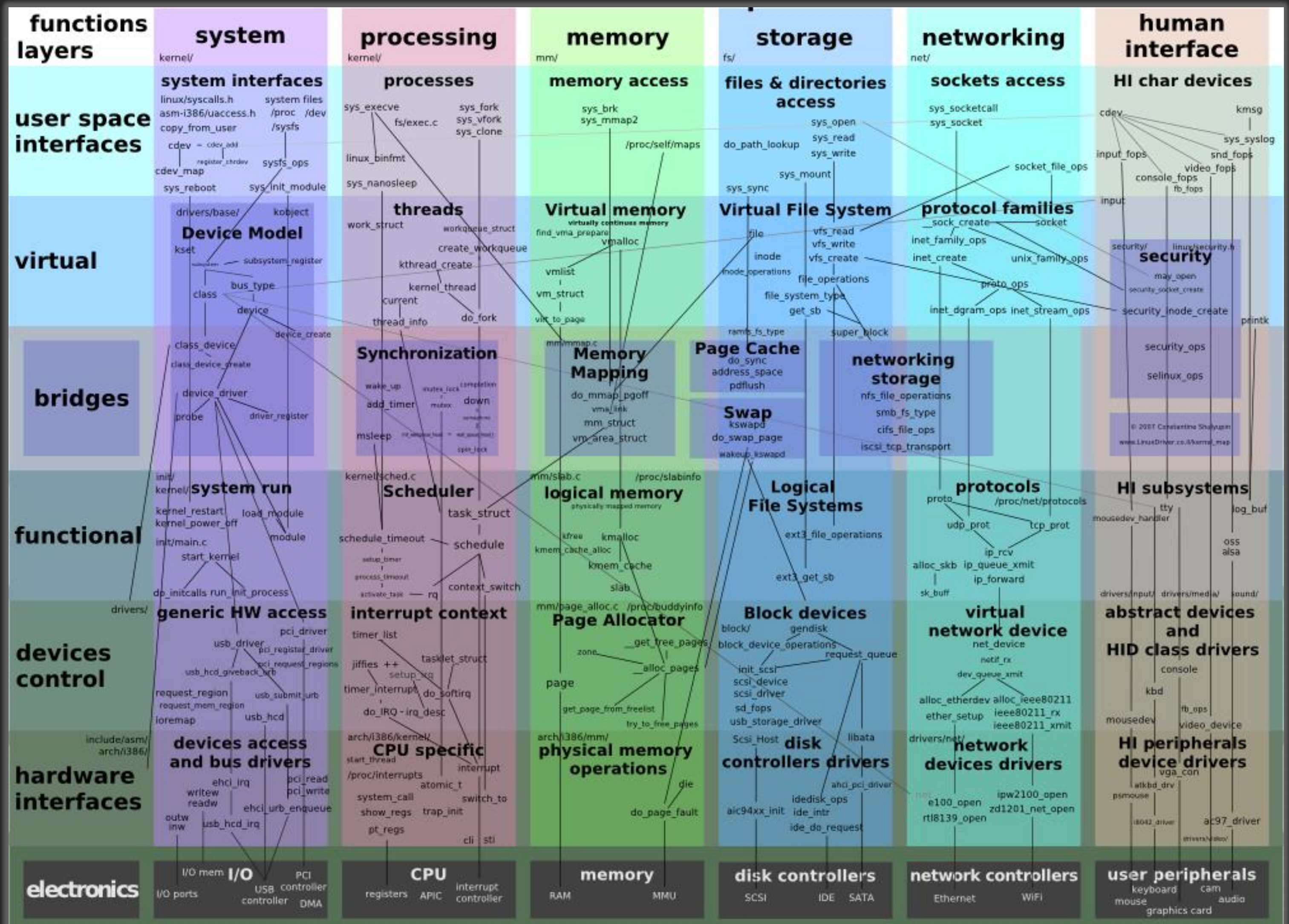
Example VMWare Deployment



Linux: modular monolithic kernel

- All of kernel is one process
  - all kernel components have access to everything
- Loadable kernel modules (LKM)
  - autonomous blocks of code
  - can be loaded/unloaded at run time
    - either explicitly, or as necessary
- Module stacking
  - modules are arranged in a hierarchy
  - modules may serve as libraries for other (client) modules

Linux kernel map

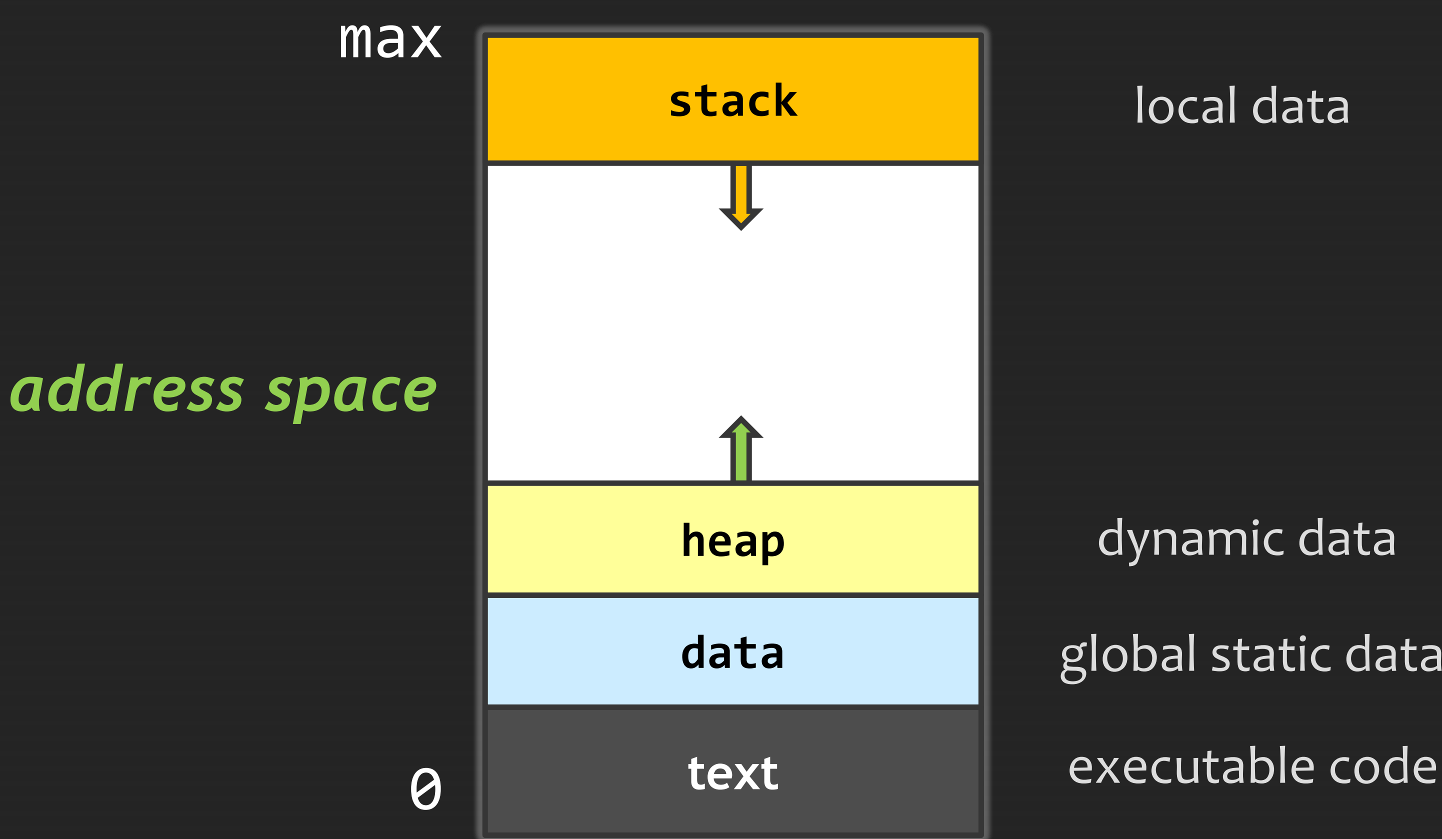


PROCESSES

Process definition

- A ***program*** is
  - a sequence of instructions (a computation) that accomplishes a specific purpose.
  - stored on persistent storage, loaded and executed when necessary.
- A ***process/task*** is a program in execution.
  - each process gets a (dynamic) allocation of the system's CPU, RAM, and other resources.
  - multiple instances of the same program can co-exist at the same time.
  - by default, each process has its own address space that is protected from those of other processes.
  - the number of processes executing in parallel is limited by available CPUs/cores.

Process in memory

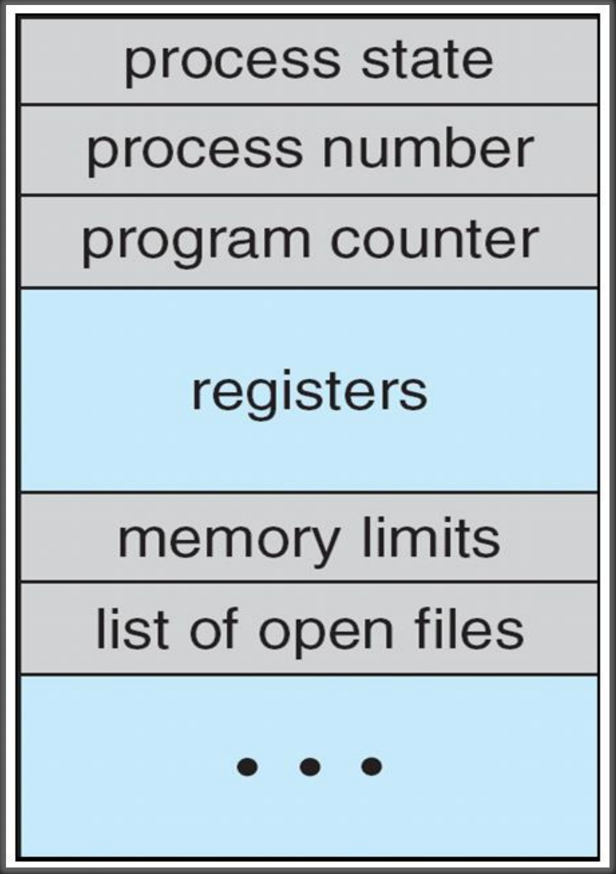


Process context

- Process context consists of all the information that describes the exact state of the process' computation.
    - *the context consists of the contents of CPU registers and some scheduling info.*
    - *if we take a snapshot of the process' state, we can freeze it and restart it later.*

Process/task Control Block

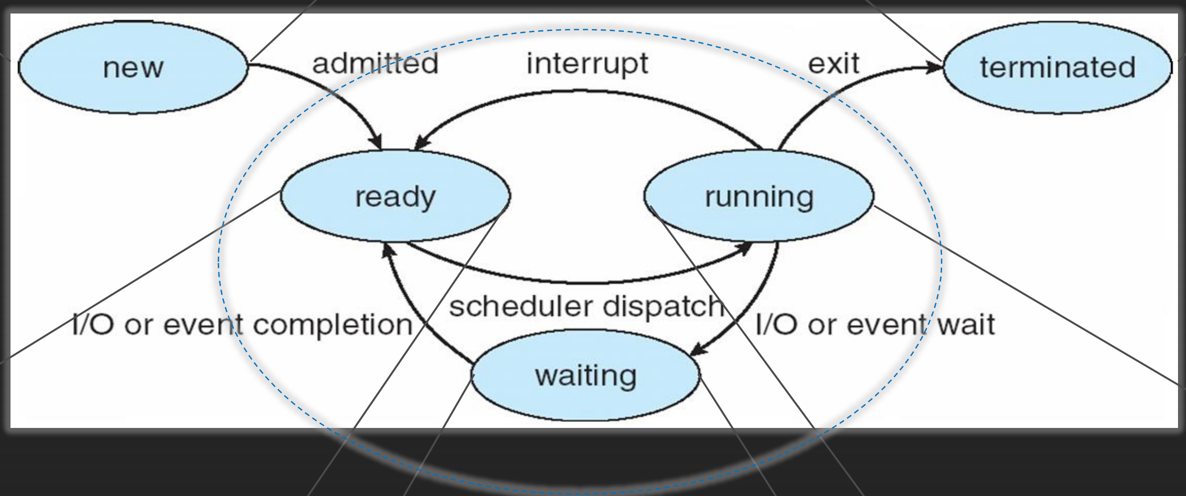
- Stores the process' context information:



Process states

process is being created

process has completed execution

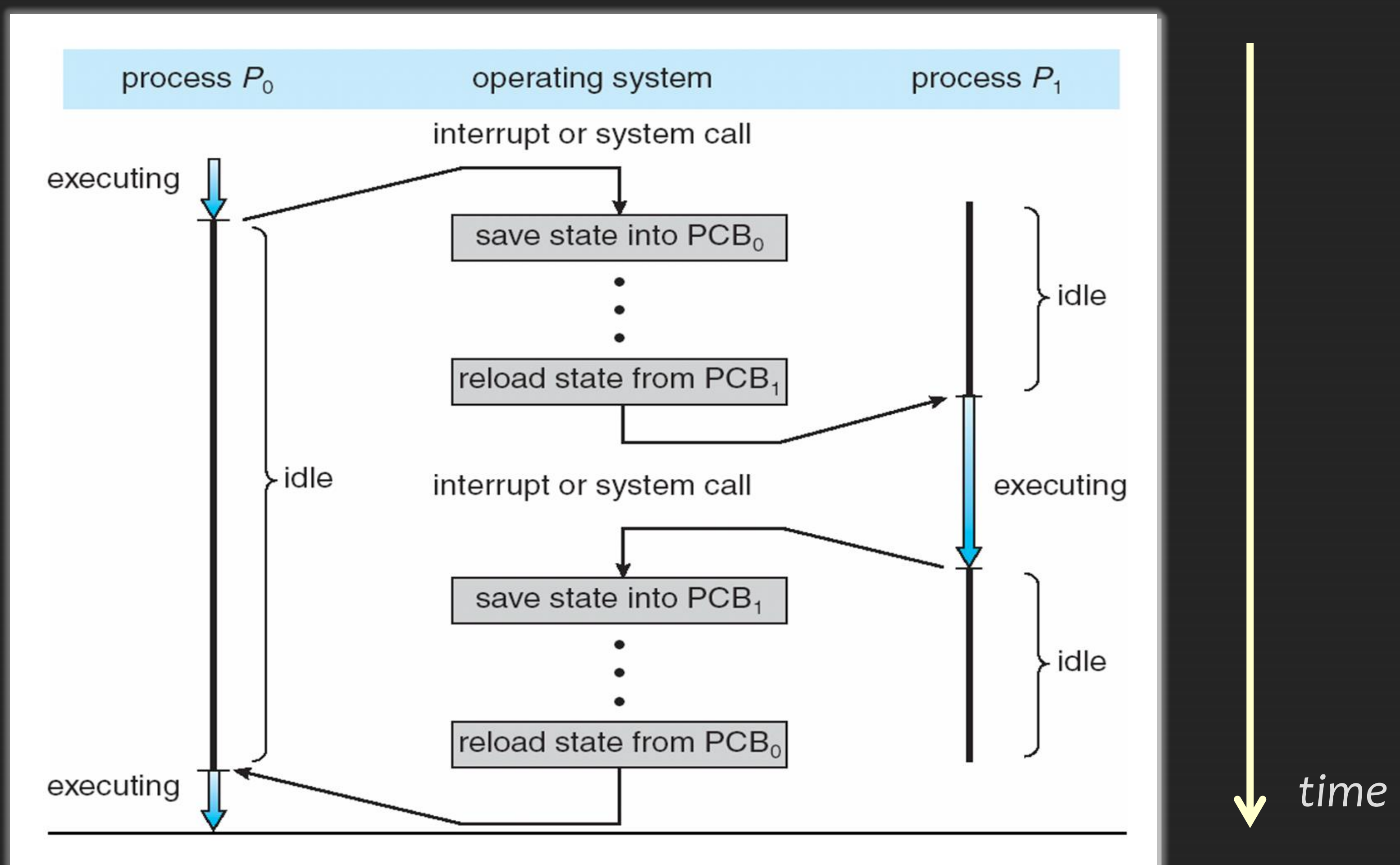


process is ready to run on the CPU

process is currently executing.

process is waiting for an external event (e.g., I/O)

Context switch



Context switch w/ cooperation

```
P0 () {
   // some processing
   …
   yield();
   // more processing
   …
}
```

```
yield () {
   // find next process
   // to be scheduled
   ctxsw(old, new);
}
```

```
P1 () {
   // some processing
   …
   yield();
   // more processing
   …
}
```

Minimal context switch: MIPS R4000

```
.globl ctxsw
        .ent ctxsw
ctxsw:
        sw $2, 8($4)
        sw $3, 12($4)
        …
        sw $14, 56($4)
        sw $16, 64($4)
        …
        sw $25, 100($4)
        sw $29, 116($4)
        sw $30, 120($4)
        sw $31, 124($4)
        sw $31, 128($4)
        mflo $15
        sw $15, 132($4)
        mfhi $15
        sw $15, 136($4)

        s.d $f0, 140($4)
        …
        s.d $f30, 260($4)
```

```
        lw $2, 8($5)
        lw $3, 12($5)
        lw $4, 16($5)
        lw $6, 24($5)
        …
        lw $14, 56($5)
        lw $16, 64($5)
        …
        lw $25, 100($5)
        lw $29, 116($5)
        lw $30, 120($5)
        lw $31, 124($5)
        lw $15, 132($5)
        mtlo $15
        lw $15, 136($5)
        mthi $15
        l.d $f0, 140($5)
        …
        l.d $f30, 260($5)
        lw $15, 128($5)
        lw $5, 20($5)
        j $15
.end ctxsw
```

PCB in Linux

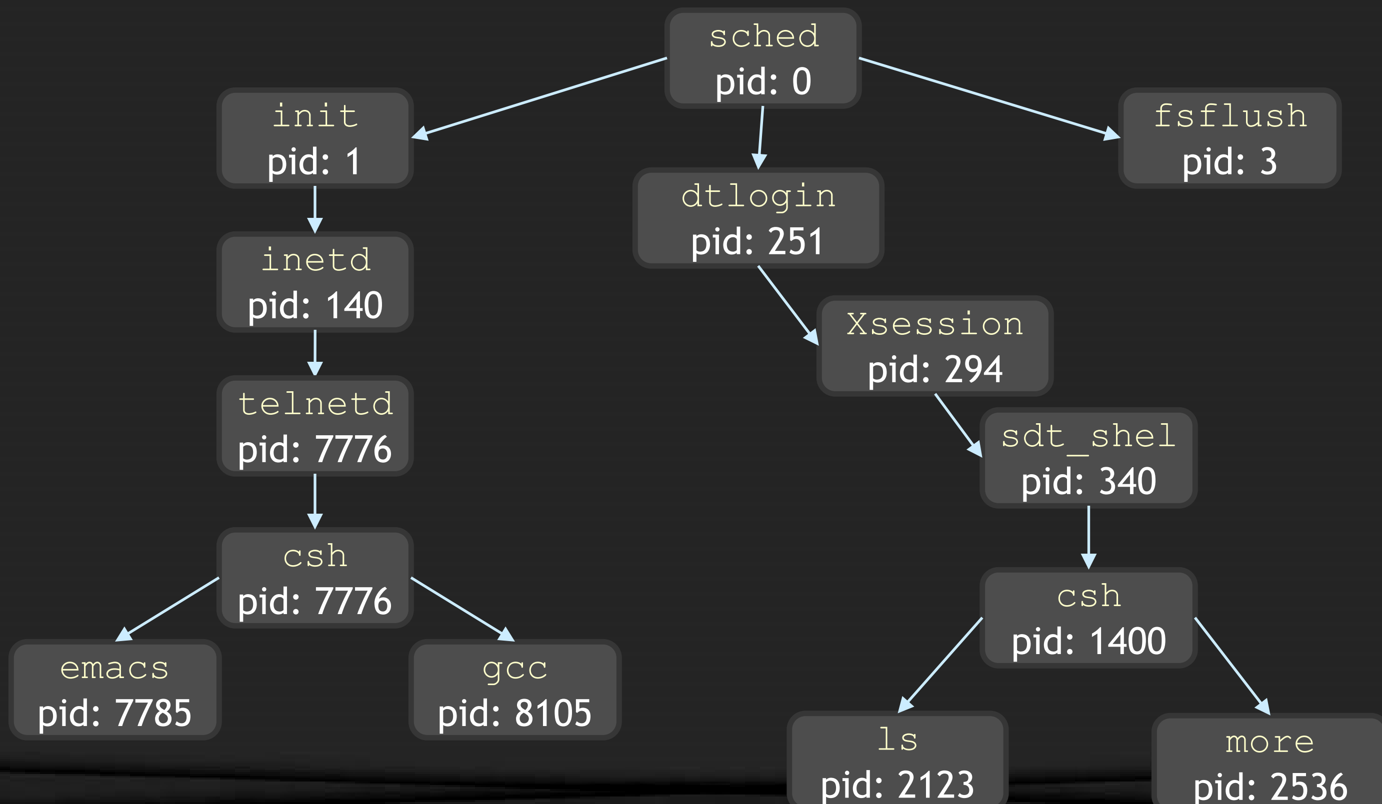
```
pid t pid;                      /* process identifier       */
long state;                     /* state of the process     */
unsigned int time slice         /* scheduling information   */
struct task struct *parent;     /* this process's parent    */
struct list head children;      /* this process's children  */
struct files struct *files;     /* list of open files       */
struct mm struct *mm;           /* address space of this pro */
                                  …
```

CPU scheduling & dispatch

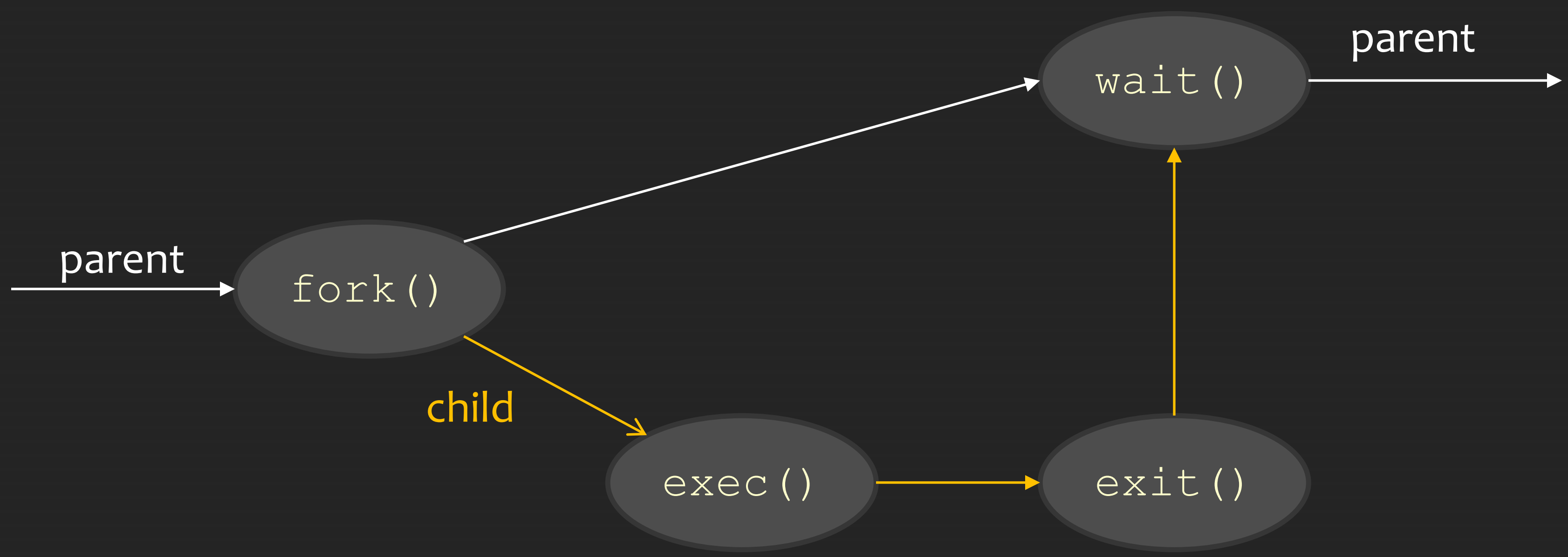
- Scheduling
  - the algorithm by which CPU time is allocated to processes.
- Dispatch
  - the procedure concerned with the immediate switch to a process selected for execution
    - context switch
    - switch to user mode
    - restart of the process

Process (pro)creation

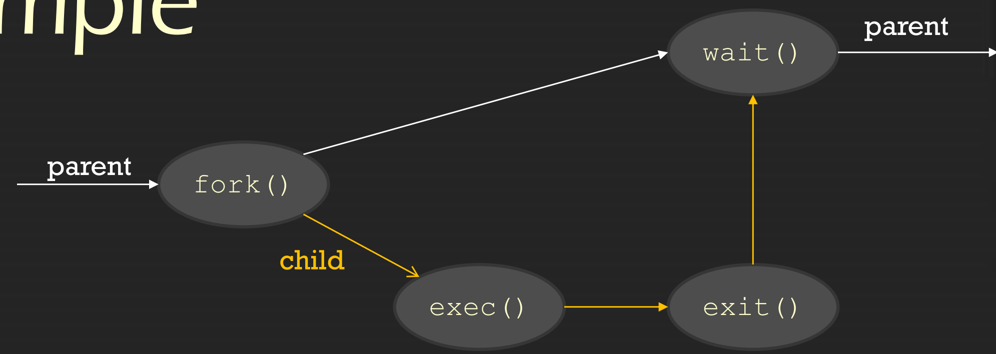
- Hardware creates first process (pid 0)
- Remaining processes are *forked/spawned*; e.g.:



Forking/spawning a process



UNIX fork() example



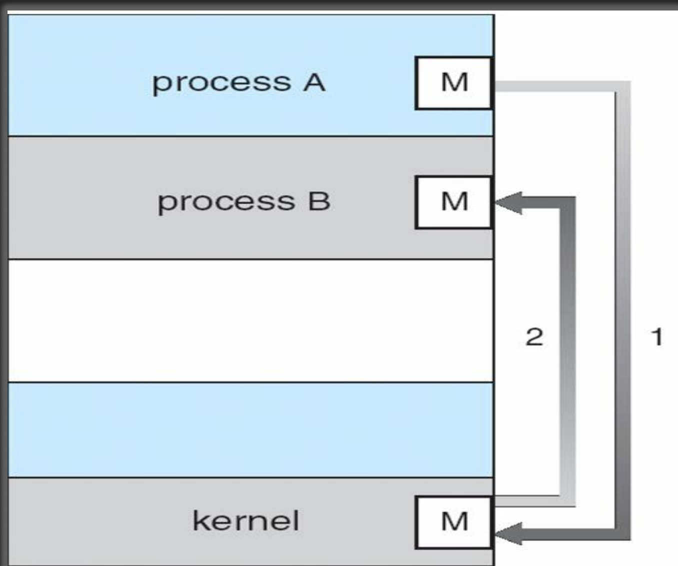

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid_t  pid;
      pid = fork();              /* fork another process */
      if (pid < 0) {             /* error occurred */
            fprintf(stderr, "Fork Failed");
            return 1;

      } else if (pid == 0) {  /* child process */
            execlp("/bin/ls", "ls", NULL);

      } else {                   /* parent process */
            wait (NULL);       /* wait for the child */
            printf ("Child Complete");
      }
      return 0;
}
```

Inter-Process Communication (IPC)

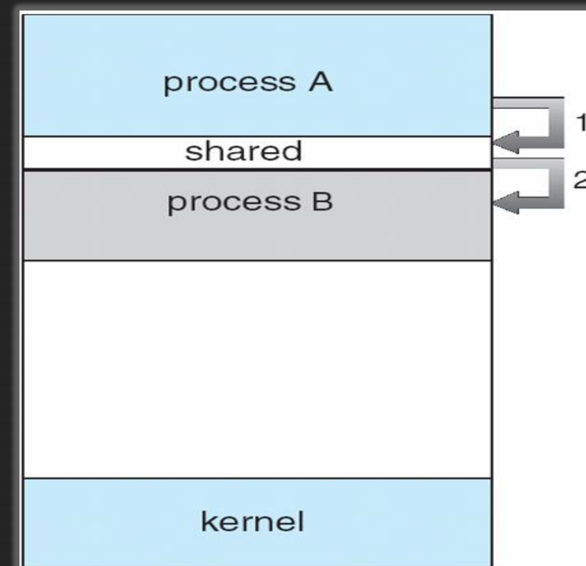
- By default, processes are isolated from each other; yet data exchange if often necessary for:
  - *resource sharing:*
    - allow files, cached objects, etc. to be shared
  - *computational speedup*
    - allow parallel computation
  - *modularity*
    - separation of concerns
    - fault/security isolation

IPC models

Message Passing



Shared Memory



Copy of data is passed along

+ more generic & secure
- less efficient (memory copy)

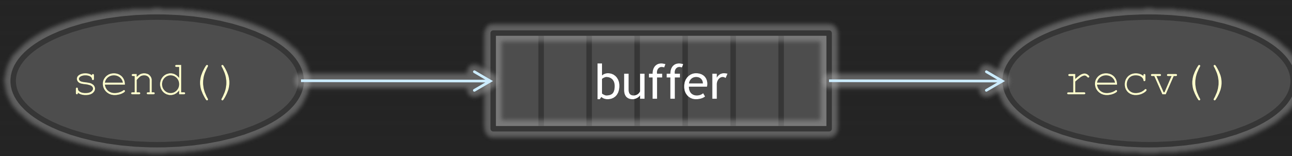
Reference to the data is passed

+ more efficient
- requires trust & synchronization

Process synchronization & IPC

- **Synchronization** is any explicit order imposed on the execution of processes.
- *Blocking IPC* is a.k.a. *synchronous*:
  - blocking `send()` stops sender execution until receipt by receiver
  - blocking `receive()` stops receiver until message is available
- *Non-blocking IPC* is a.k.a. *asynchronous*:
  - non-blocking `send()` returns immediately before result is known
  - non-blocking `receive()` returns immediately either with waiting message, or `null`
- *Note:*
  - synchronous/asynchronous is independently applicable to `send()`/`receive()`

Buffering and synchronization (local)



- Producer-consumer system
- Single shared buffer
  - $|buffer| = 0$ ➔ blocking send/receive, a.k.a. rendezvous
  - $|buffer| = k < \infty$ ➔ up to **k** asynchronous send/recv
  - $|buffer| = \infty$ ➔ completely asynchronous system—unlimited asynchronous send/recv

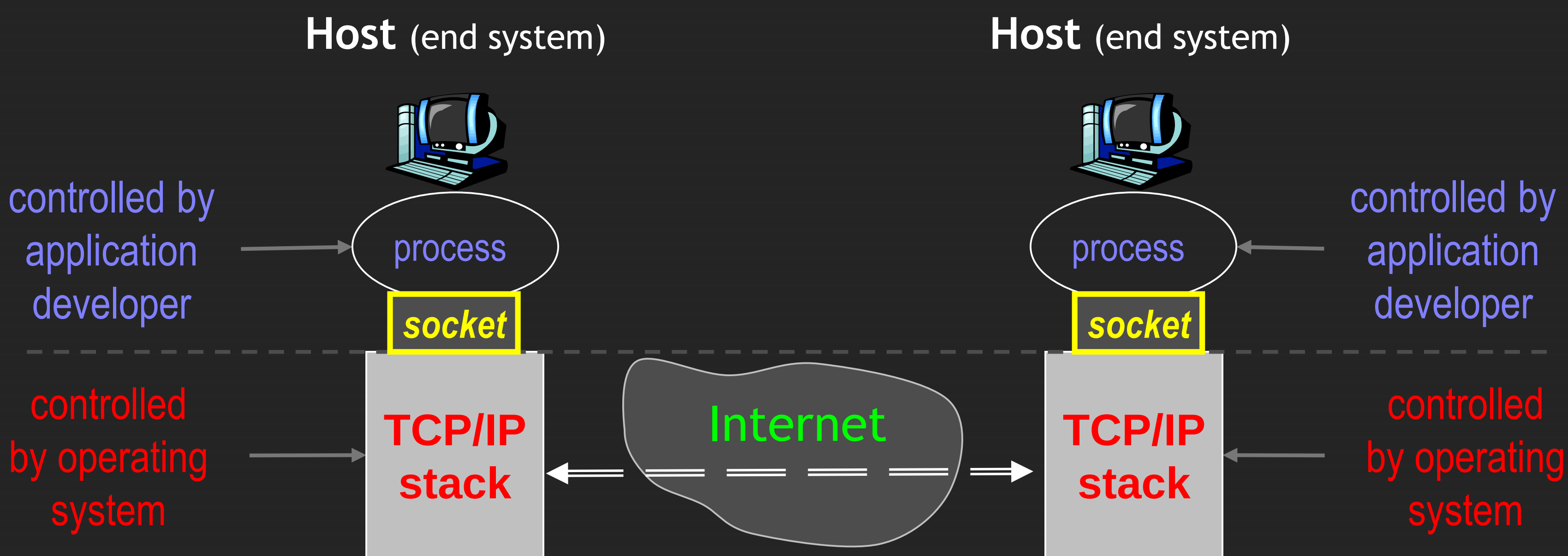
Buffering and synchronization (remote)

- Separate send/receive buffers
  - $|buffer_1|+|buffer_2| = 0$ ➔ blocking send/receive, a.k.a. rendezvous
  - $|buffer_1|+|buffer_2| = k_1 + k_2 < \propto$
    - ➔ up to $k_1 + k_2$ asynchronous send
    - ➔ up to k_2 asynchronous recv

Local communication mechanisms

- Shared address space
  - parent/child processes
- Shared segments
- Ordinary pipes
  - unidirectional buffered communication
  - requires parent/child relationship
- Named pipes
  - bi-directional communications
  - no parent/child relationship necessary

A socket is the basic **remote communication** abstraction provided by the OS to processes.

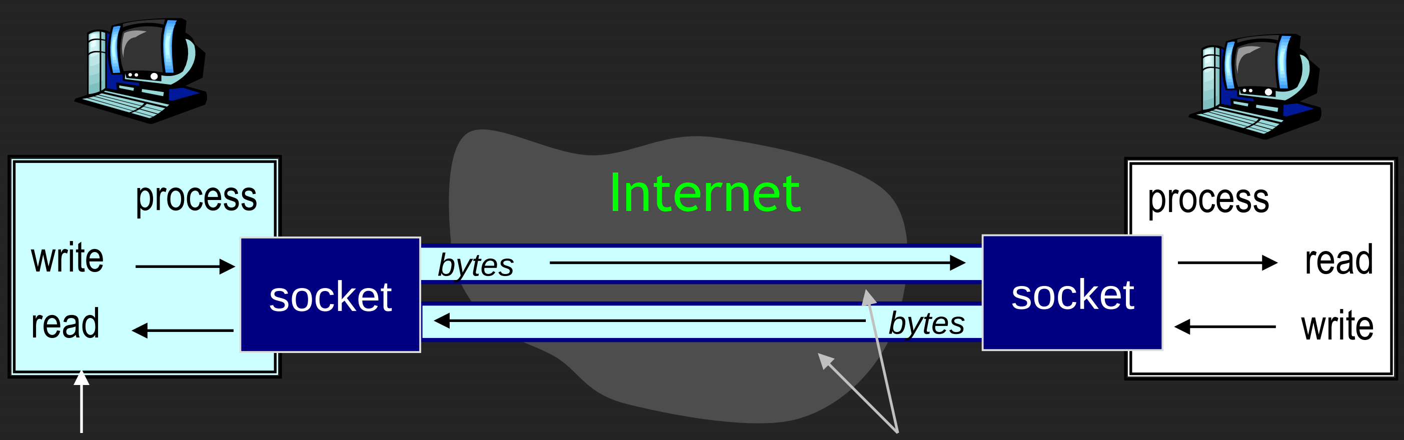


Network access is managed by the OS
→ OS provides a socket API to allow network communication

Socket programming using **TCP** service

Application Viewpoint

TCP provides reliable, in-order transfer of a stream of bytes between two end points (processes).

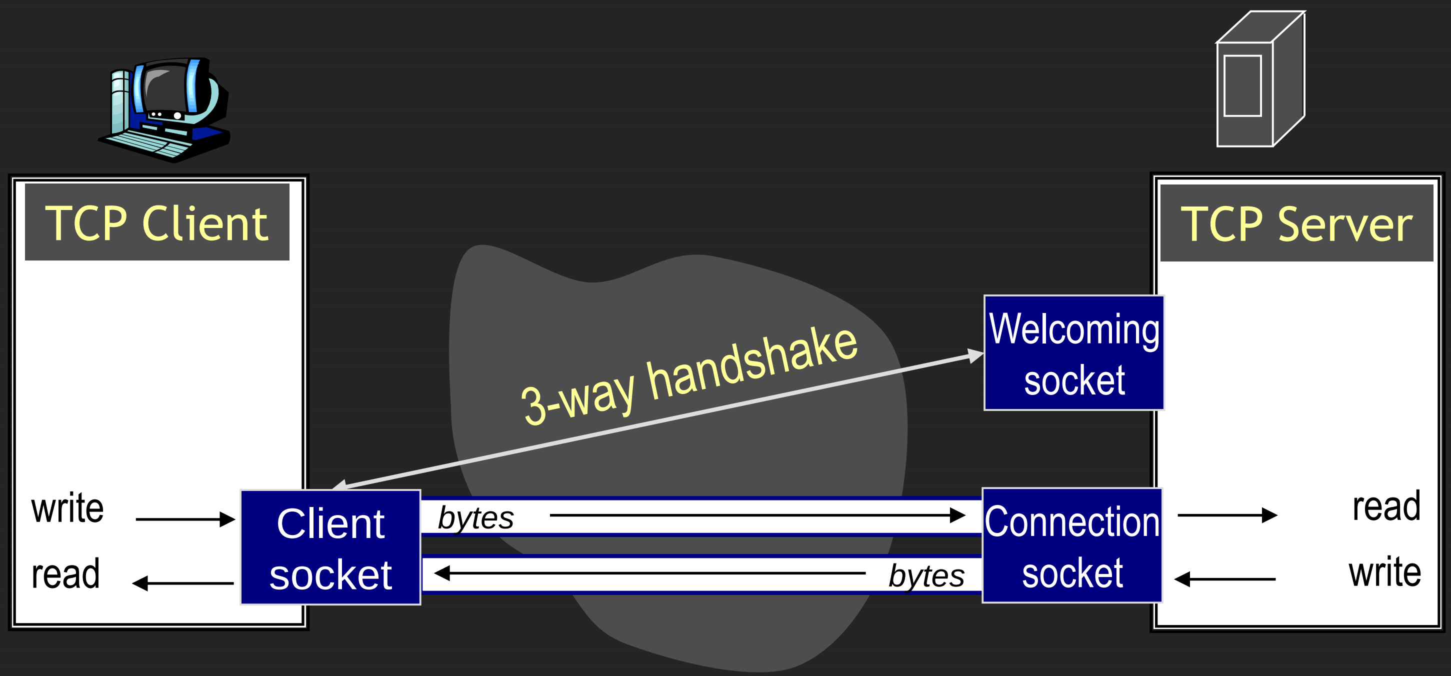


Same interface as other stream I/O (files, pipes, etc.).

Pair of pipes abstraction

TCP communication is based on the client/server concept.

Server is started first &
waits for connection requests.



Client initiates connection via 3-way handshake.

Once established, connection is completely symmetrical.

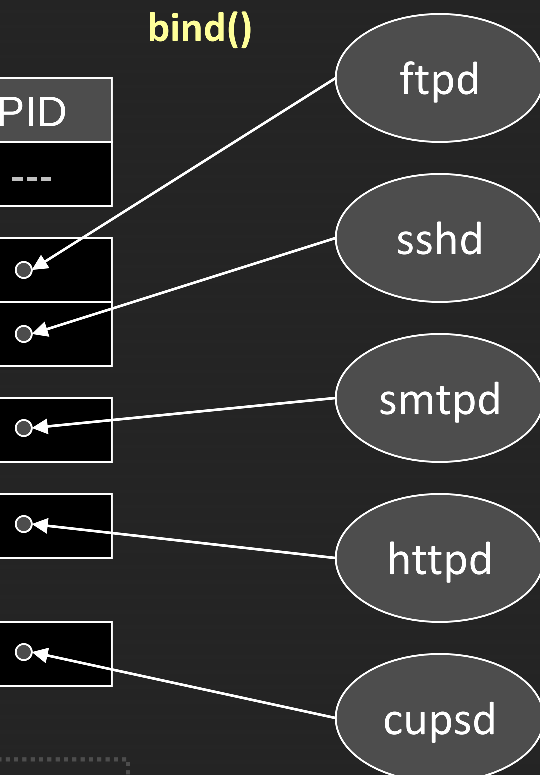
A port number is a 16-bit number that uniquely identifies a network process on a host.

Client must know port number to reach server.

Server ***binds*** to a desired port number (welcoming socket).

Most standard services have a default port:

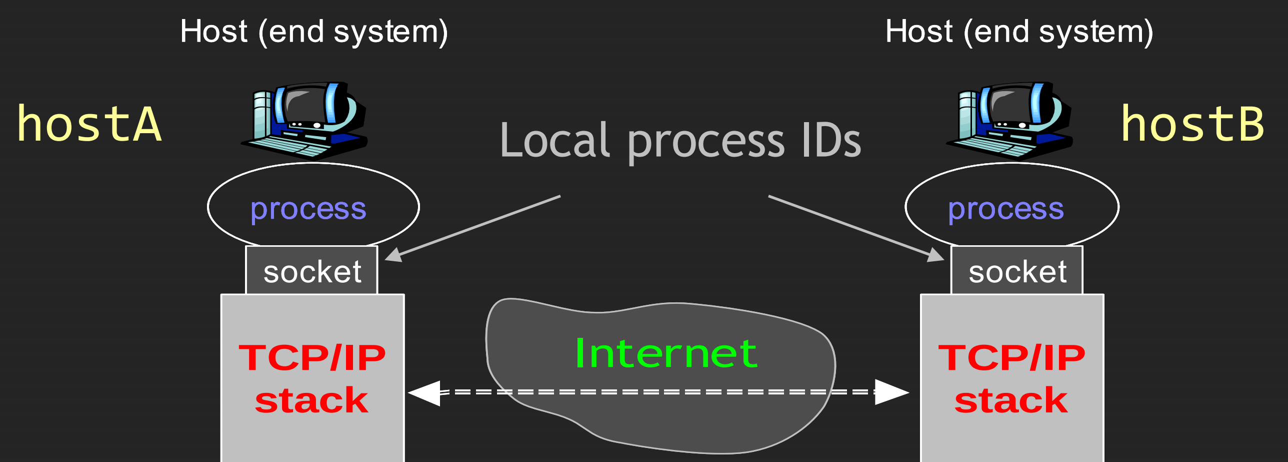
	Port	PID
	0000	---
	...	
File Transfer →	0021	
Secure Shell →	0022	
Simple Mail Transfer →	0025	
	...	
World Wide Web →	0080	
	...	
Print Service →	0631	
	...	



Client sockets use any available port >1023.

Network addressing for sockets

A ⇔ B socket connection: <IP-hostA, portA, IP-hostB, portB>



Mnemonic, logical ID

Unique, routable ID

Host name: `cook.cs.uno.edu` → **DNS** → **IP address:** `137.30.120.32`