

# Operating Systems and Computer Architecture



# How does a computer work?



- Signals control hardware
- Signals can have a high (1) or low (0) values
- Integers are represented as binary strings
- Floats are represented in IEEE754 format

# Exponential Notation



- The following are equivalent representations of **1,234**

$$123,400.0 \times 10^{-2}$$

$$12,340.0 \times 10^{-1}$$

$$1,234.0 \times 10^0$$

$$123.4 \times 10^1$$

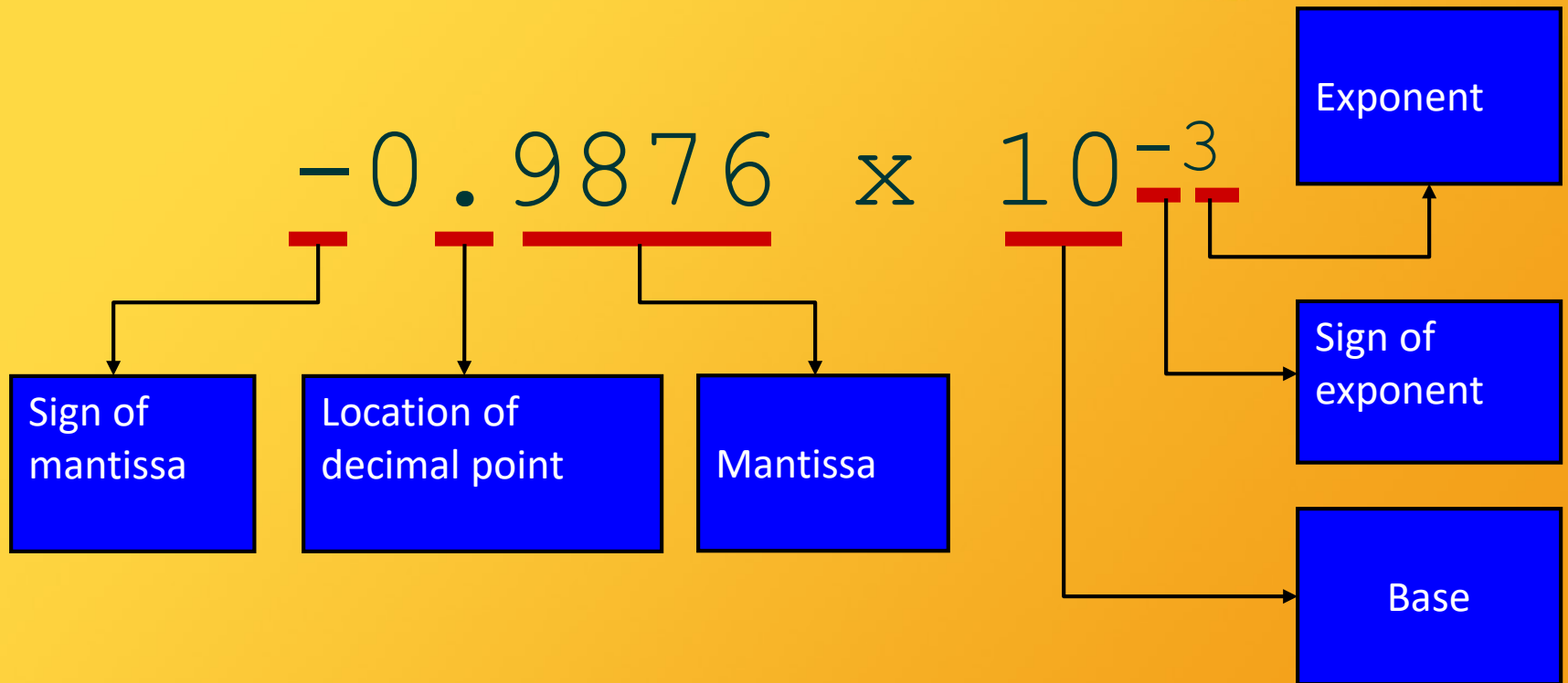
$$12.34 \times 10^2$$

$$1.234 \times 10^3$$

$$0.1234 \times 10^4$$

The representations differ in that the decimal place – the “point” -- “floats” to the left or right (with the appropriate adjustment in the exponent).

# Parts of a Floating Point Number



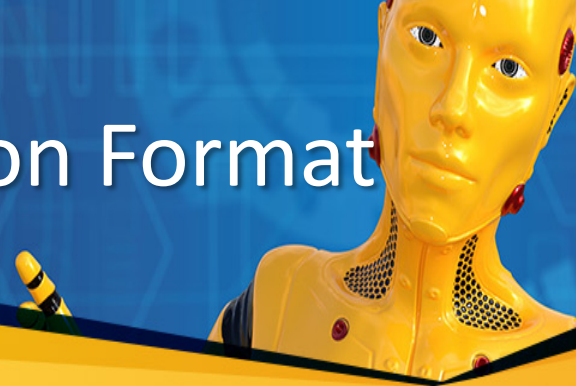
# IEEE 754 Standard



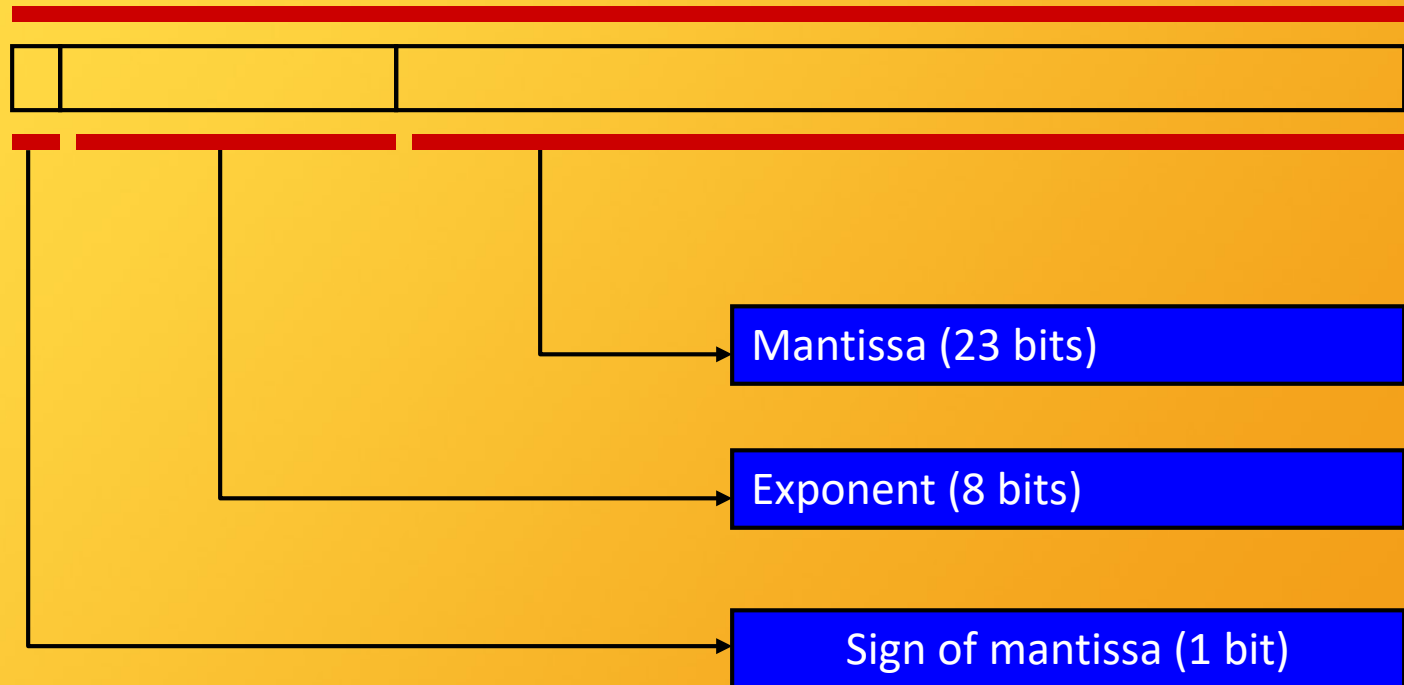
- Most common standard for representing floating point numbers
  - Single precision: 32 bits, consisting of...
    - Sign bit (1 bit)
    - Exponent (8 bits)
    - Mantissa (23 bits)
  - Double precision: 64 bits, consisting of...
    - Sign bit (1 bit)
    - Exponent (11 bits)
    - Mantissa (52 bits)



# Single Precision Format



32 bits



# Normalization



- The mantissa is *normalized*
- Has an implied decimal place on left
- Has an implied “1” on left of the decimal place
  - E.g.,
    - Mantissa → 1010000000000000000000000000
    - Represents...  $1.101_2 = 1.625_{10}$

# Excess Notation



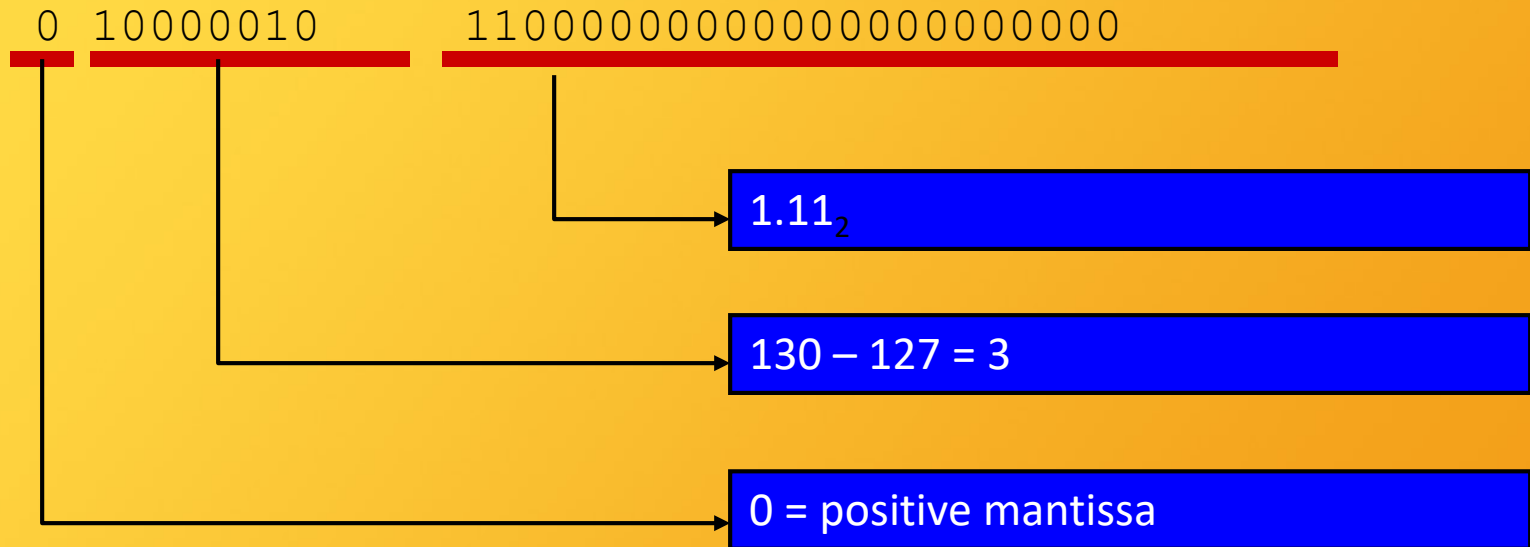
- To include +ve and -ve exponents, “excess” notation is used
  - Single precision: excess 127
  - Double precision: excess 1023
- The value of the exponent stored is larger than the actual exponent
  - E.g., excess 127,
    - Exponent → 10000111
    - Represents...  $135 - 127 = 8$



# Example



- Single precision



$$+1.11_2 \times 2^3 = 1110.0_2 = 14.0_{10}$$

# Hexadecimal



- It is convenient and common to represent the original floating point number in hexadecimal
  - The preceding example...

0	100000	010	110	000000	0000000	0000000	000000
4	1	6	0	0	0	0	0

# Converting from Floating Point



- E.g., What decimal value is represented by the following 32-bit floating point number?

$C17B0000_{16}$



- Step 1

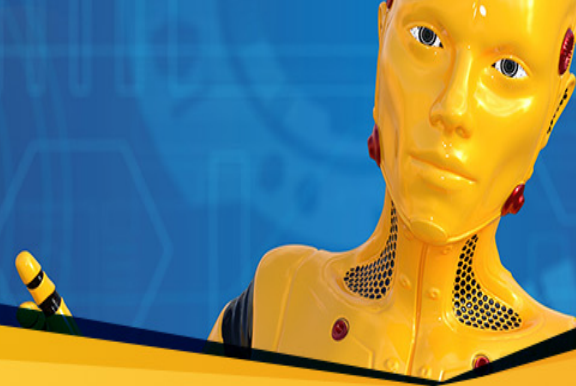
– Express in binary and find S, E, and M

$C17B0000_{16} =$

1 10000010 111101100000000000000000<sub>2</sub>



↑  
1 = negative  
0 = positive



- Step 2

- Find “real” exponent,  $n$

- $n = E - 127$

- $= 10000010_2 - 127$

- $= 130 - 127$

- $= 3$





- Step 3

- Put  $S$ ,  $M$ , and  $n$  together to form binary result
- (Don't forget the implied "1." on the left of the mantissa.)

$$-1.1111011_2 \times 2^n =$$

$$-1.1111011_2 \times 2^3 =$$

$$-1111.1011_2$$



- Step 4

– Express result in decimal

$-1111.1011_2$

$-15$

$$2^{-1} = 0.5$$

$$2^{-3} = 0.125$$

$$2^{-4} = \underline{0.0625}$$

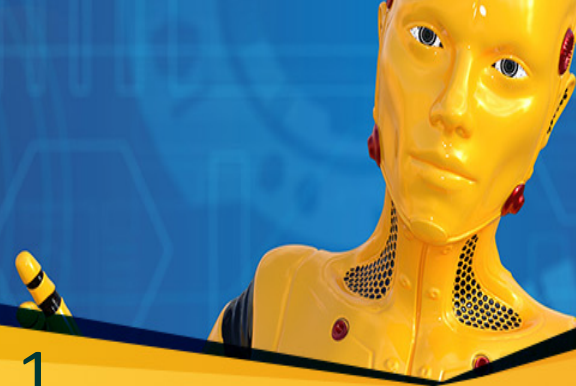
$0.6875$

Answer:  $-15.6875$

# Converting to Floating Point



- E.g., Express  $36.5625_{10}$  as a 32-bit floating point number (in hexadecimal)

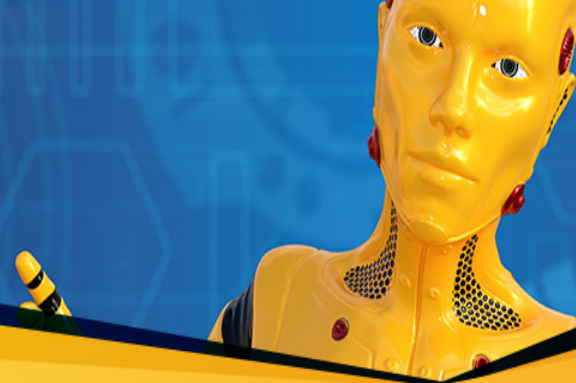


- Step 1

– Express original value in binary

$$36.5625_{10} =$$

$$100100.1001_2$$



- Step 2
- Normalize

$$100100.1001_2 =$$

$$1.001001001_2 \times 2^5$$





- Step 3

- Determine S, E, and M

$$+1.001001001_2 \times 2^5$$

S

M

$n$

$$\begin{aligned} E &= n + 127 \\ &= 5 + 127 \\ &= 132 \\ &= 10000100_2 \end{aligned}$$

S = 0 (because the value is positive)



- Step 4

– Put S, E, and M together to form 32-bit binary result

0 10000100 001001001000000000000000<sub>2</sub>

S

E

M



- Step 5

– Express in hexadecimal

0 10000100 001001001000000000000000<sub>2</sub> =

0100 0010 0001 0010 0100 0000 0000 0000<sub>2</sub> =

4 2 1 2 4 0 0 0<sub>16</sub>

**Answer: 42124000<sub>16</sub>**



User Space

Services / Hypervisor

System Calls

Device Drivers / Hardware Abstraction Layer (HAL)

Kernel / BIOS

Instruction Set Architecture

Hardware

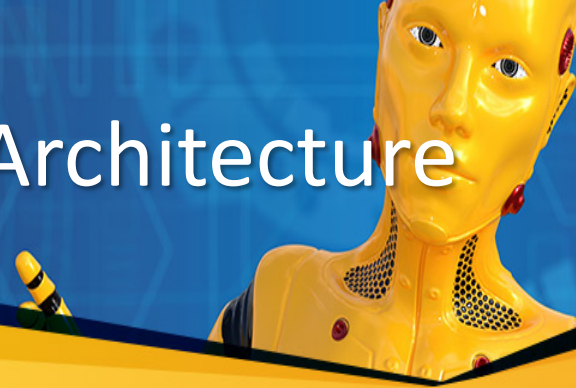
# Hardware



- CPU (central processing unit)
  - Registers
  - Flags
- ALU (Arithmetic Logic Unit)
- FPU (Floating-Point Unit)
  - Cache
  - RAM/ROM/Flash
  - Memory
- DMA (Direct Memory Access)
  - Bus



# Instruction Set Architecture



- Machine code: chain of instructions in 32/64 bit binary format
- Assembly: mnemonics which get translated to machine code
  - Low level programming
    - Set of mnemonics are the ISA
      - Contract between hardware and software
        - OS follow that contract
        - Programs get compiled into that contract
      - Processor supports all the functions specified by the contract

# ISA Types



- CISC: complex instruction set computing
  - Include many instructions in the ISA
  - CPU tends to be larger and more power consuming
    - E.g. INTEL x86
- RISC: reduced instruction set computing
  - Support only limited functions
  - Hardware can be reduced and more specialized
    - Emulate remaining functions (e.g. mul = adds)
      - E.g. MIPS, Sparc
- There are no longer traditional RISCs
  - Multiple cores tend to be RISCs

# x86 General Purpose Registers



- 8 bits
  - al and ah, bl and bh, cl and ch, dl and dh
- 16 bits
  - ax, bx, cx, dx
- 32 bits
  - eax, ebx, ecx, edx
- 64 bits
  - rax, rbx, rcx, rdx
- Specialty: a (arithmetic), b (base), c (counter), d (data)

# x86 Other Registers



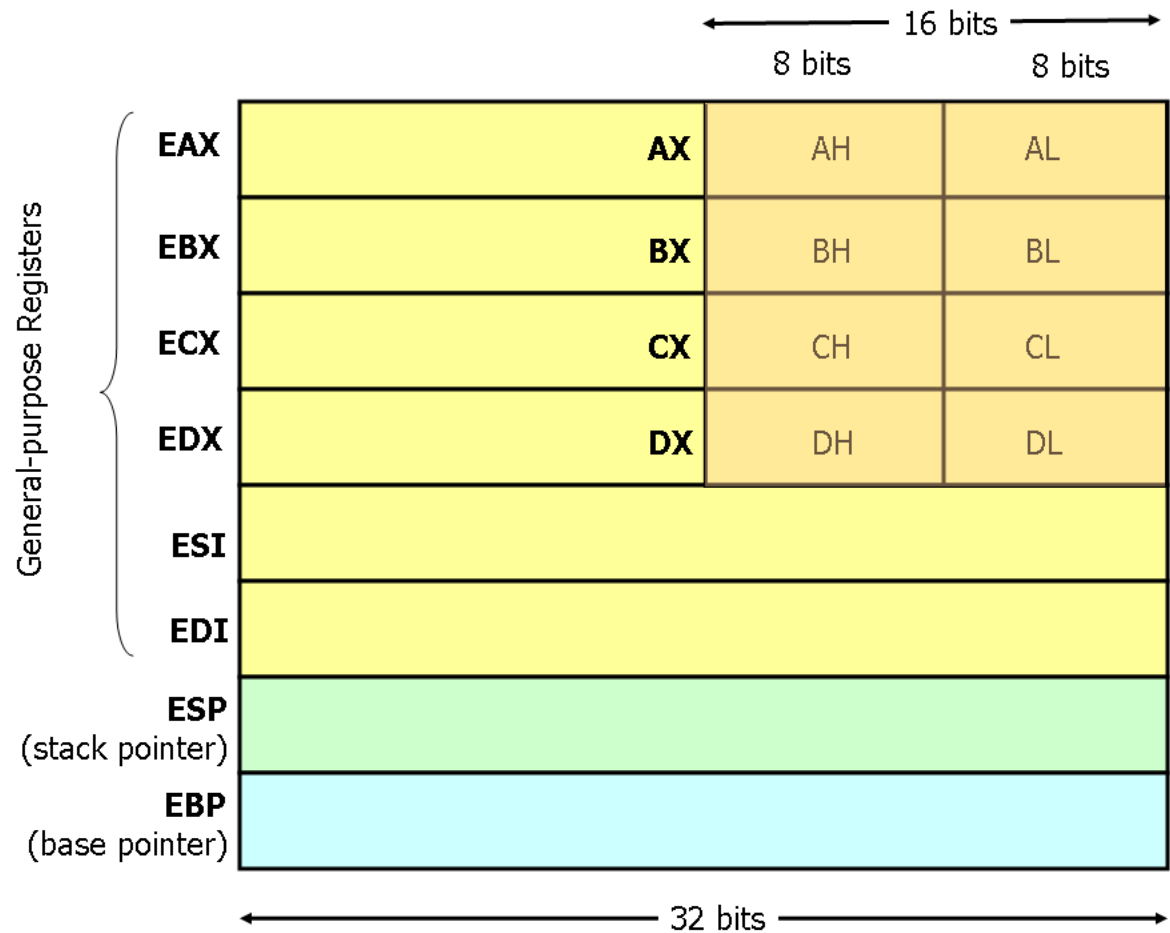
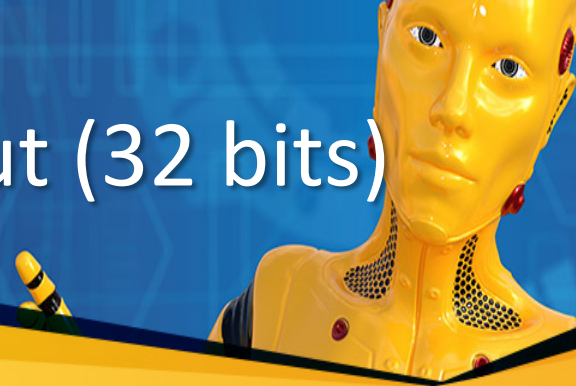
- ESP: stack pointer
- ESI, EDI: index registers (source, data)
- EBP: frame/base pointer (on stack)
  - EIP: instruction pointer
  - PC: program counter
  - Some segments:
    - CS: code segment
    - DS: data segment
    - SS: stack segment
    - ES, FS, GS: extra segments

# EFLAGS



- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

# Register Layout (32 bits)





# Register Layout (32 bits)



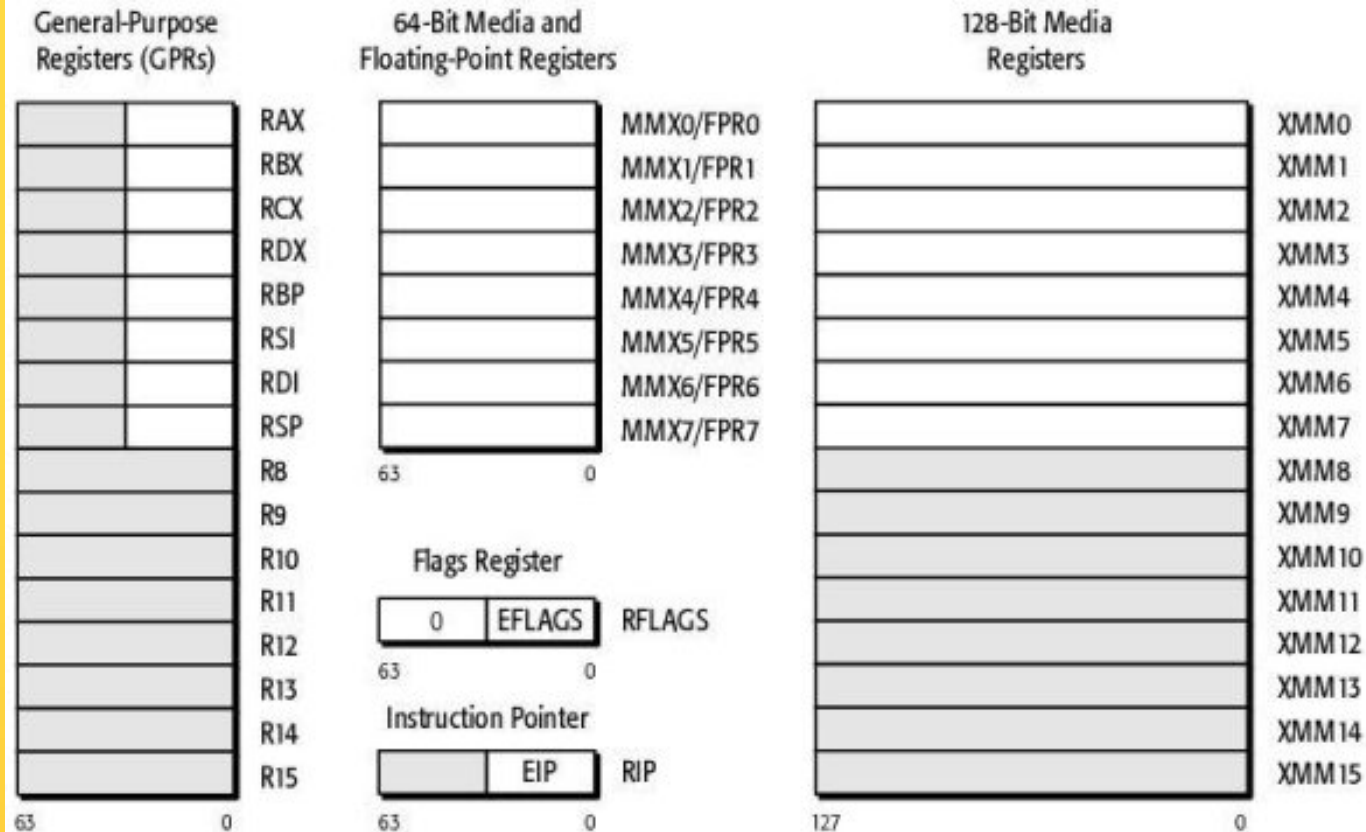
- AX is the "accumulator"; some of the operations, such as **MUL** and **DIV**, require that one of the operands be in the accumulator. Some other operations, such as **ADD** and **SUB**, may be applied to any of the registers (that is, any of the eight general- and special-purpose registers) but are more efficient when working with the accumulator.
- BX is the "base" register; it is the only general-purpose register which may be used for indirect addressing. For example, the instruction **MOV [BX], AX** causes the contents of AX to be stored in the memory location whose address is given in BX.
- CX is the "count" register. The looping instructions (**LOOP**, **LOOPE**, and **LOOPNE**), the shift and rotate instructions (**RCL**, **RCR**, **ROL**, **ROR**, **SHL**, **SHR**, and **SAR**), and the string instructions (with the prefixes **REP**, **REPE**, and **REPNE**) all use the count register to determine how many times they will repeat.
- DX is the "data" register; it is used together with AX for the word-size **MUL** and **DIV** operations, and it can also hold the port number for the **IN** and **OUT** instructions, but it is mostly available as a convenient place to store data, as are all of the other general-purpose registers.





- SP is the stack pointer, indicating the current position of the top of the stack. You should generally never modify this directly, since the subroutine and interrupt call-and-return mechanisms depend on the contents of the stack.
- BP is the base pointer, which can be used for indirect addressing similar to BX.
- SI is the source index, used as a pointer to the current character being read in a string instruction (LODS, MOVS, or CMPS). It is also available as an offset to add to BX or BP when doing indirect addressing; for example, the instruction `MOV [BX+SI], AX` copies the contents of AX into the memory location whose address is the sum of the contents of BX and SI.
- DI is the destination index, used as a pointer to the current character being written or compared in a string instruction (MOVS, STOS, CMPS, or SCAS). It is also available as an offset, just like SI.

# Register Layout (64 bits)



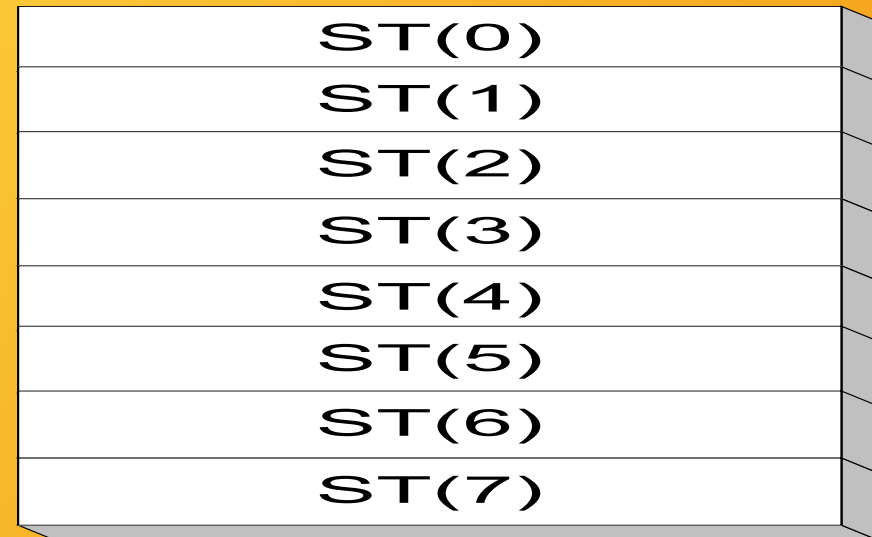
Legacy x86 registers, supported in all modes  
 Register extensions, supported in 64-bit mode

Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers

# Floating Point Registers



- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - arranged in a stack
  - used for all floating-point arithmetic
- Eight 64-bit MMX registers
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations



# x86 Instructions



- Typical syntax
  - Mnemonic dst, src
- Some instructions
  - NOP ; correct way of putting the CPU to sleep
    - Mov ax, dx
    - Add cx, 8
    - Sub bx, 1
  - Mul al, 2 ;result is stored in ax
  - Div 4 ;ax is an implicit operator
    - Jmp
  - Loop ;cx is implicit count
    - Cmp bx, 0
  - Int 24h or 10h ; most common interrupt services

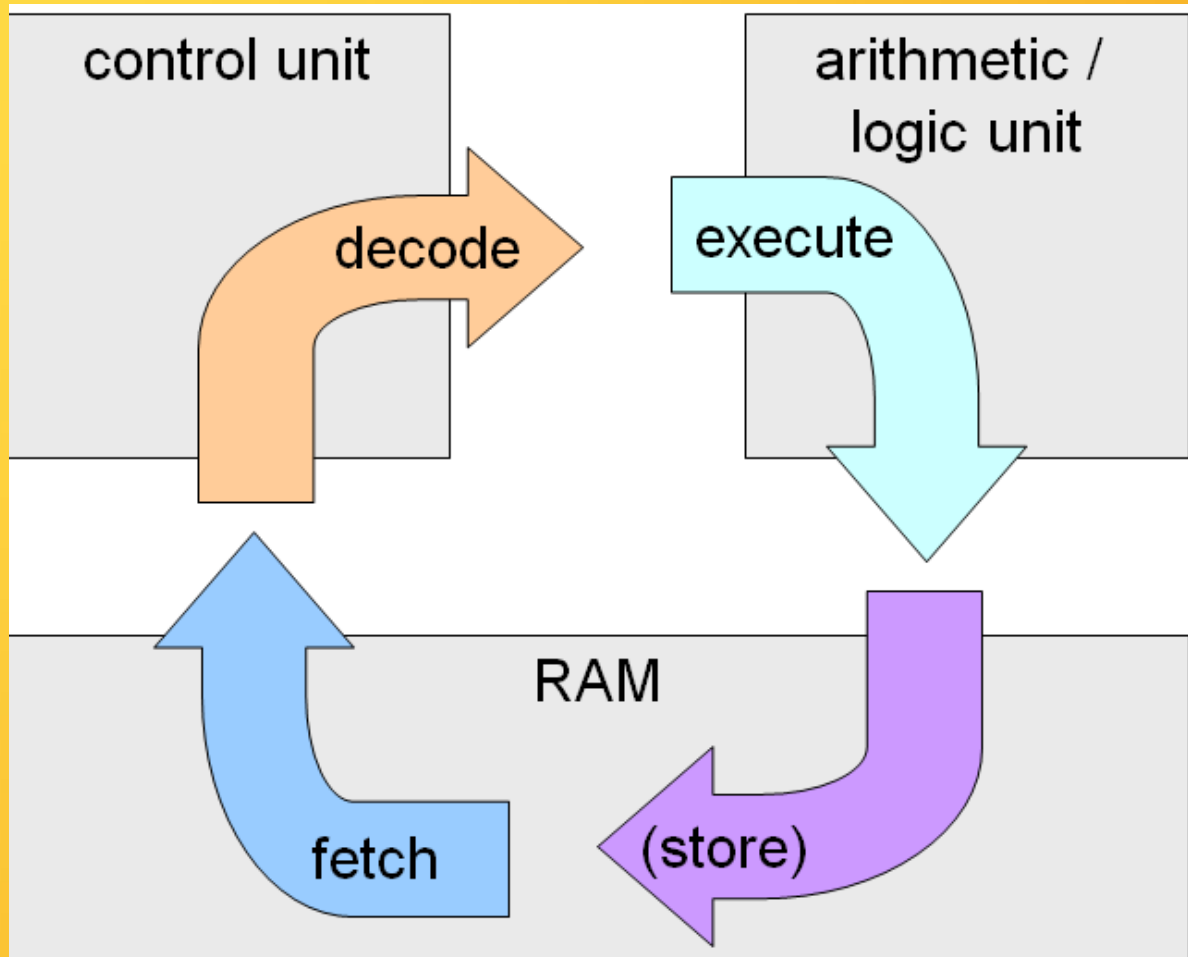
# Addressing Modes



- There is always a register in the operation (if it receives an operand)
  - Register (default)
    - Only registers are involved
    - Memory
      - Reading from or writing to memory
      - Immediate
        - A constant in the instruction
        - Direct or Indirect
          - Direct: address is on the instruction
          - Indirect: address is on another location to be search



# Instruction Cycle



# What happens during an interrupt?



- Could be explicitly called via `int` mnemonic.
  - An execution is never interrupted
    - Can interrupt after or before executing the instruction.
- From the OS perspective the activation record of the function is recorded, along with the states of all the variables



# MBR and BIOS



- Master Boot Record (MBR)
  - Decides which OS will boot
  - Bootloader (e.g. Grub) should be installed on a dual booting (2 OS) computer
- Basic Input/Output System (BIOS)
  - Provides basic IO (as the name implies) to help booting the machine
  - POST test helps check every hardware

# OS Kernel



- Main core of the operating system
- Mostly written in C (with many gotos ... YES gotos)
- In windows the core libraries are in C:\windows
  - Common of viruses to attack this area
- In linux the core is scattered
  - /bin contains binaries
  - /usr local installs (by user)
  - /media mounted drives
    - /dev list of devices
  - Usb, serials, parallel ports, hard disk, etc.
    - /home user directories
    - /etc everything else

# Device Drivers



- Allow easy access from software to hardware by forcing a pre-defined interface
  - POSIX (Unix)
  - Win32 (Windows)

# POSIX interface



POSIX is a family of standards, specified by the IEEE, to clarify and make uniform the application programming interfaces (and ancillary issues, such as commandline shell utilities) provided by Unix-y operating systems. When you write your programs to rely on POSIX standards, you can be pretty sure to be able to port them easily among a large family of Unix derivatives (including Linux, but not limited to it!); if and when you use some Linux API that's not standardized as part of Posix, you will have a harder time if and when you want to port that program or library to other Unix-y systems (e.g., MacOSX) in the future.

- Create
- Destroy
- Open
- Close
- Read
- Write
- IOctl
- Could we enforce security?

# System Calls

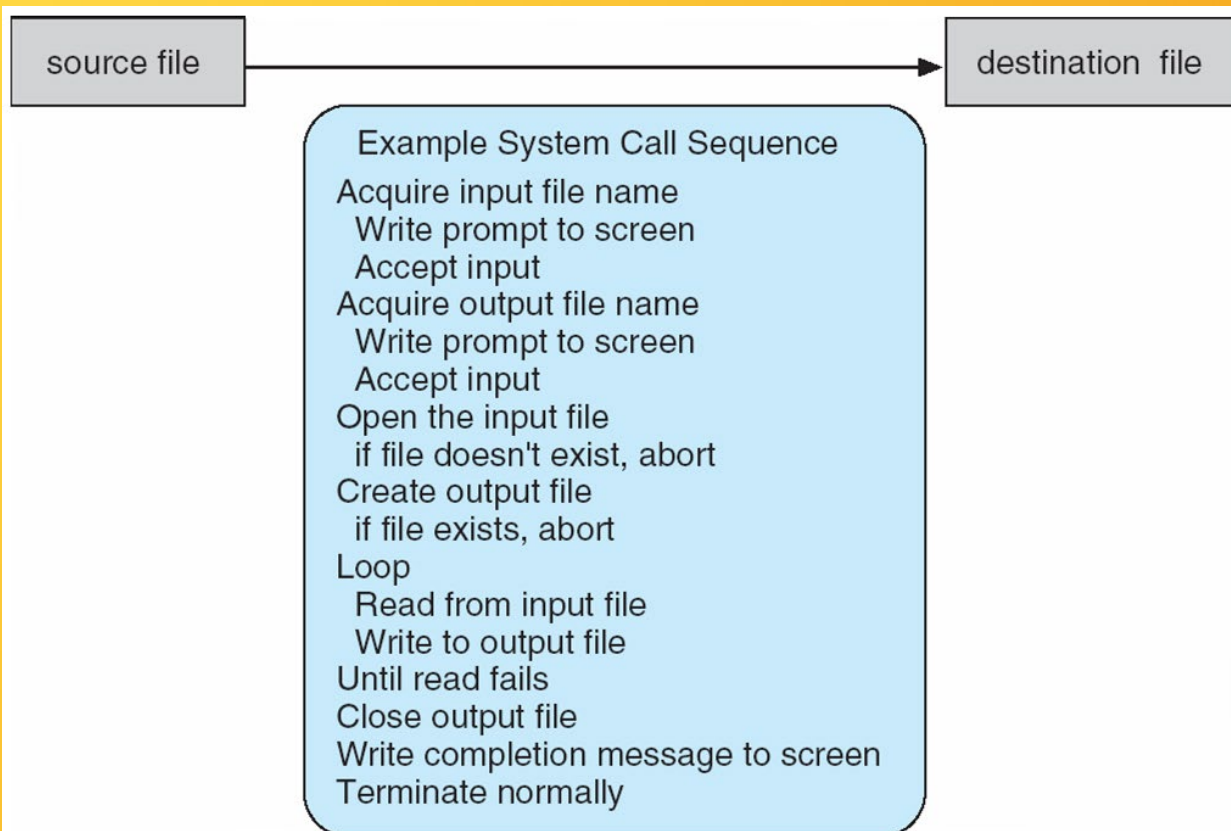


- Give the programs access to the HAL
  - In C may be called with
    - System
    - Exec functions
  - In Java via the Runtime

# Example of System Calls



- System call sequence to copy the contents of one file to another file





# Example of Standard API

## EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t  read(int fd, void *buf, size_t count)
```

return value	function name	parameters
--------------	---------------	------------

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

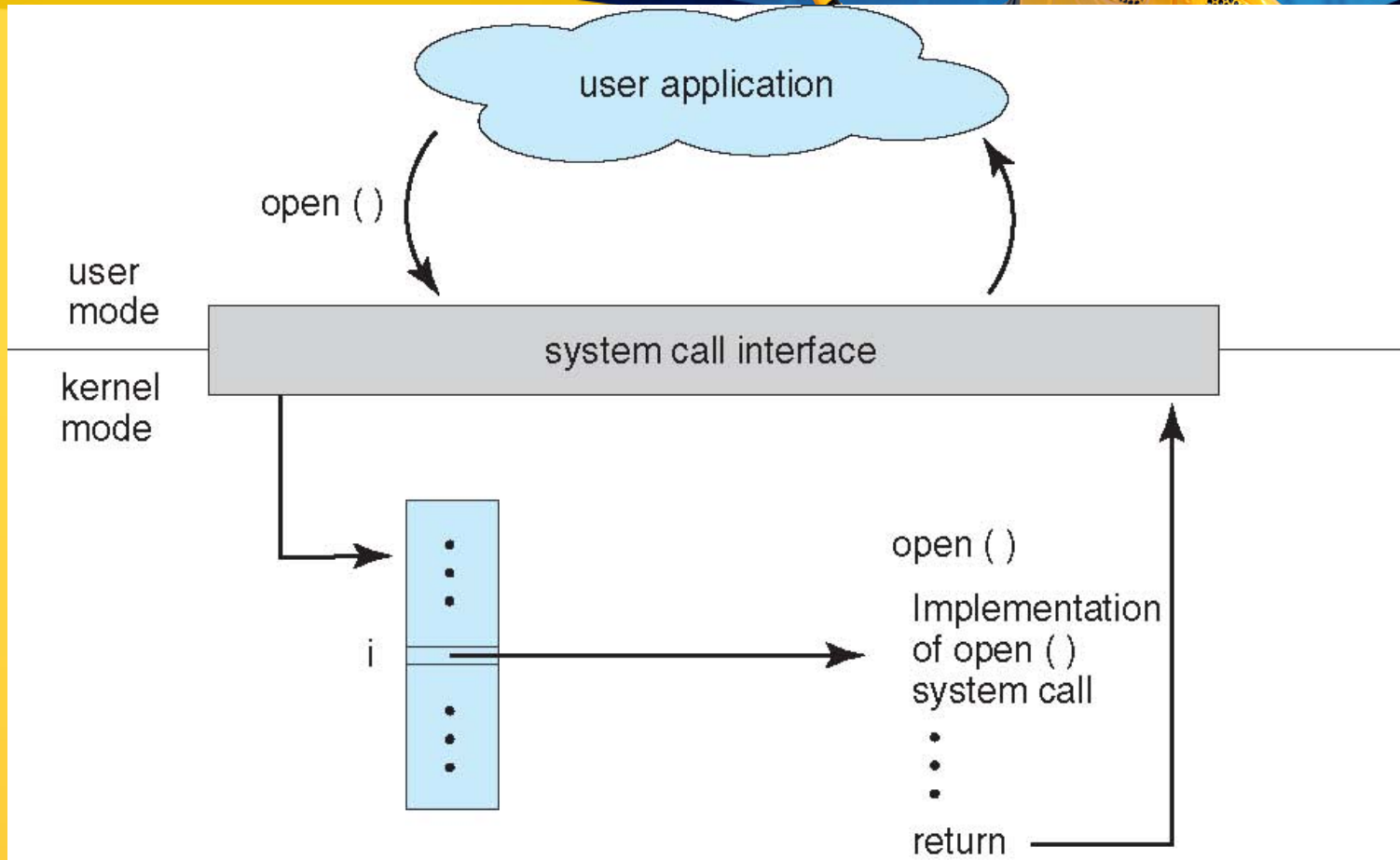


# System Call Implementation



- Typically, a **number associated with each system call**
  - **System-call interface** maintains a **table indexed according to these numbers**
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface **hidden from programmer by API**
    - Managed by **run-time support library** (set of functions built into libraries included with compiler)

# API – System Call – OS Relationship

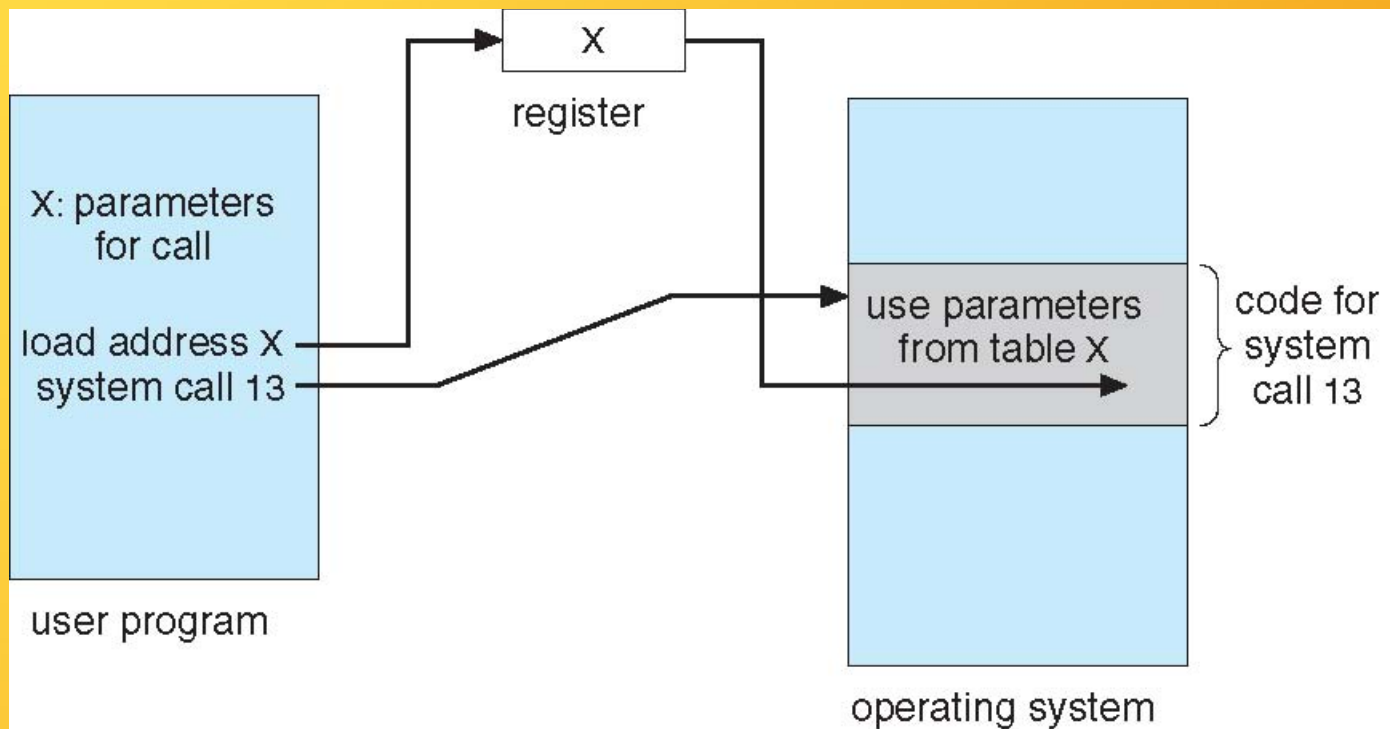


# System Call Parameter Passing



- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in **registers**
    - In some cases, may be more parameters than registers
  - Parameters stored in a **block**, or table, in memory, and **address** of block passed as a parameter in a **register**
    - This approach taken by Linux and Solaris
  - Parameters placed, or **pushed**, onto the **stack** by the program and **popped** off the stack by the operating system
  - Block and stack methods **do not limit** the number or length of parameters being passed

# Parameter Passing via Table



# Types of System Calls



- **Process control**
  - create process, terminate process
    - end, abort
    - load, execute
  - get process attributes, set process attributes
    - wait for time
      - wait event, signal event
    - allocate and free memory
      - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
    - **Locks** for managing access to shared data between processes

# Types of System Calls



- **File management**
  - create file, delete file
    - open, close file
  - read, write, reposition
  - get and set file attributes
- **Device management**
  - request device, release device
    - read, write, reposition
  - get device attributes, set device attributes
    - logically attach or detach devices



# Types of System Calls (Cont.)



- **Information maintenance**
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes
- **Communications**
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
    - From **client** to **server**
  - **Shared-memory model** create and gain access to memory regions
    - transfer status information
    - attach and detach remote devices



# Types of System Calls (Cont.)



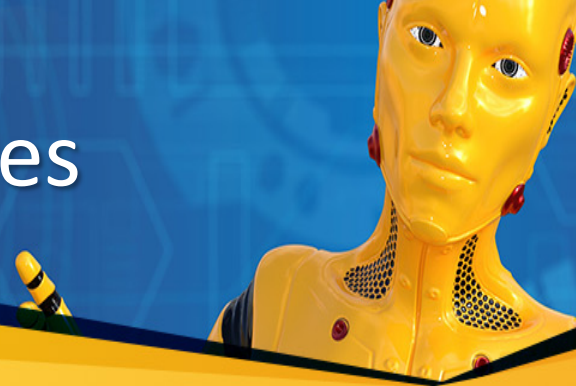
- **Protection**
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

# Examples of Windows and Unix System Calls



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

# Services



- File system
  - Network
    - Printer
      - Email
- Timed services
  - Cron
- Automatic updates

# Hypervisor



- A program that acts as the master boot record of the virtual operating systems
- Will be covering more on VMs (virtual machines) in the course

# User space

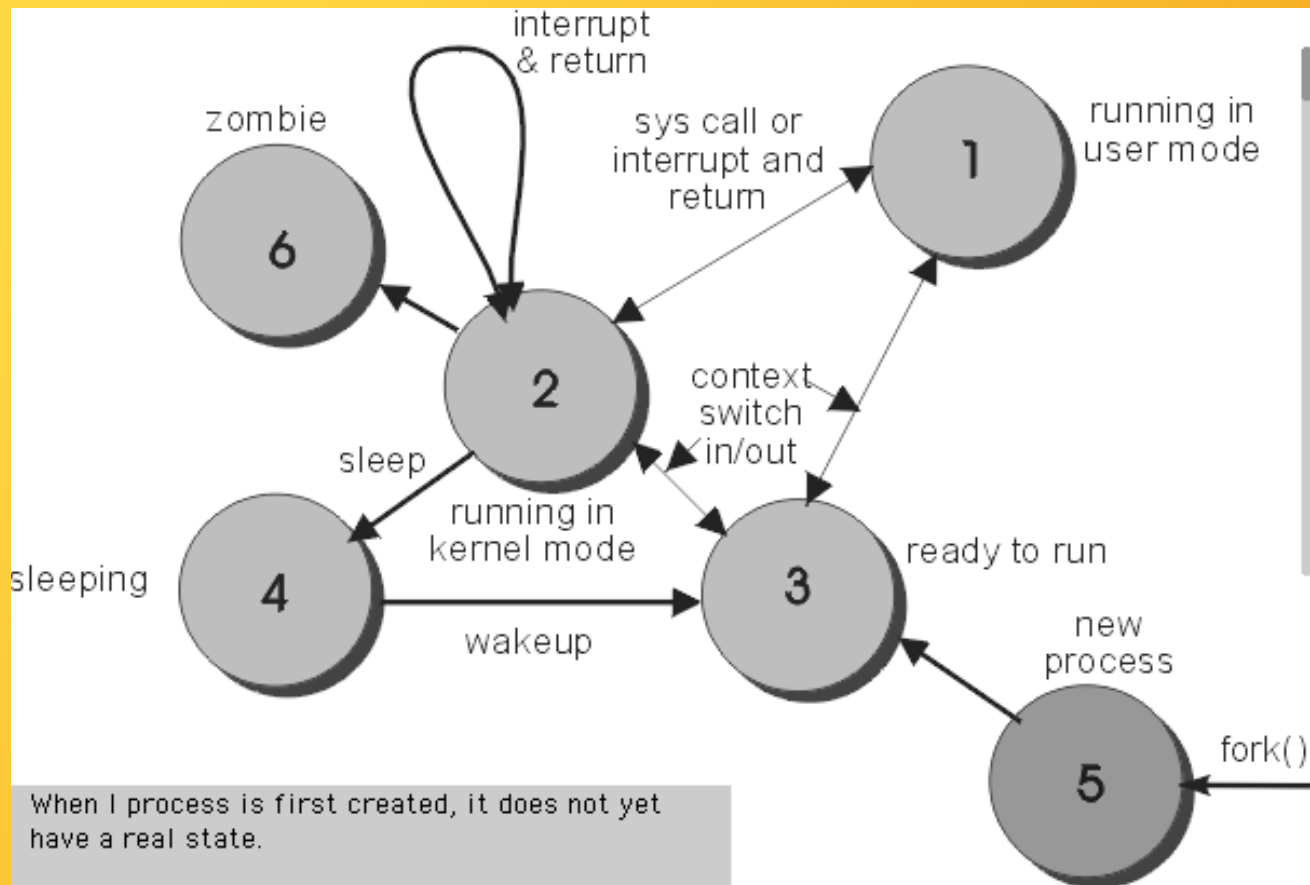


- Applications
  - Shell
  - Utilities
- Databases

# Process and Threads



- A process is an application which passing through its life cycle (i.e. not dead)





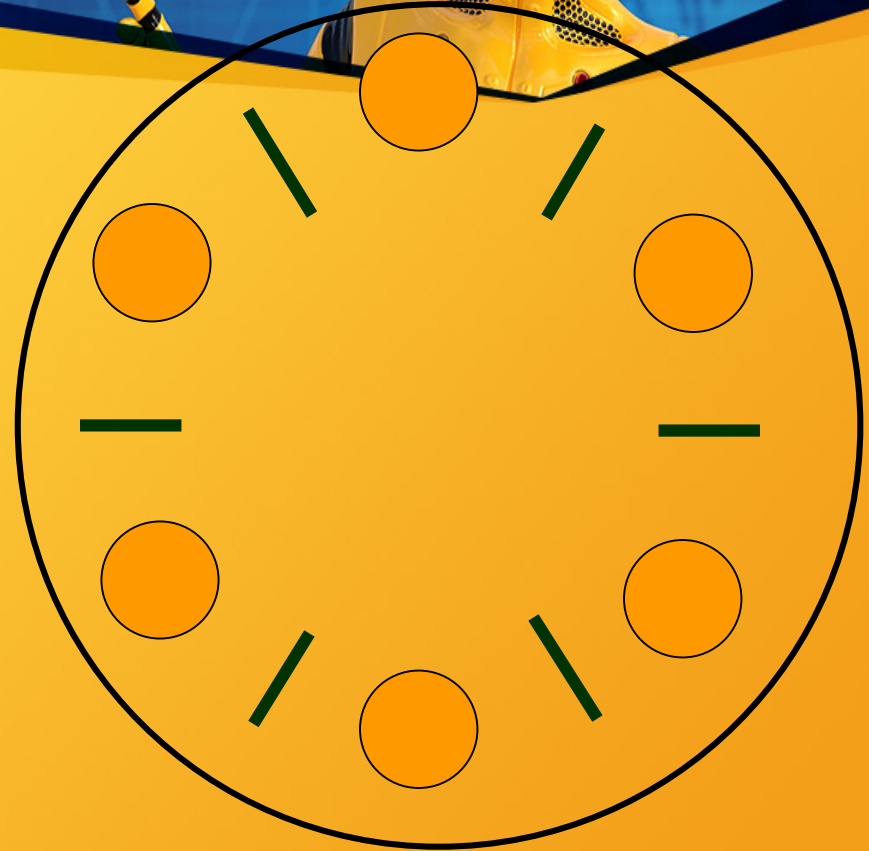
# Some concepts



- Context switching
  - Occurs when a process is put to sleep so that another can continue running
    - Gives the illusion of multitasking
- Deadlock
  - If A is waiting for B, B is waiting for C, and C is waiting for A
  - In other words a deadlock is a state where no process can continue
    - Avoid it with good process philosophies
      - Diners, Readers/Writers, Producer/Consumer
      - A priority queue is a good data structure here

# The Dining Philosophers Problem

- Philosophers
  - think
  - take forks (one at a time)
  - eat
  - put forks (one at a time)
- Eating requires 2 forks
- Pick one fork at a time
- How to prevent deadlock?
- What about starvation?
- What about concurrency?

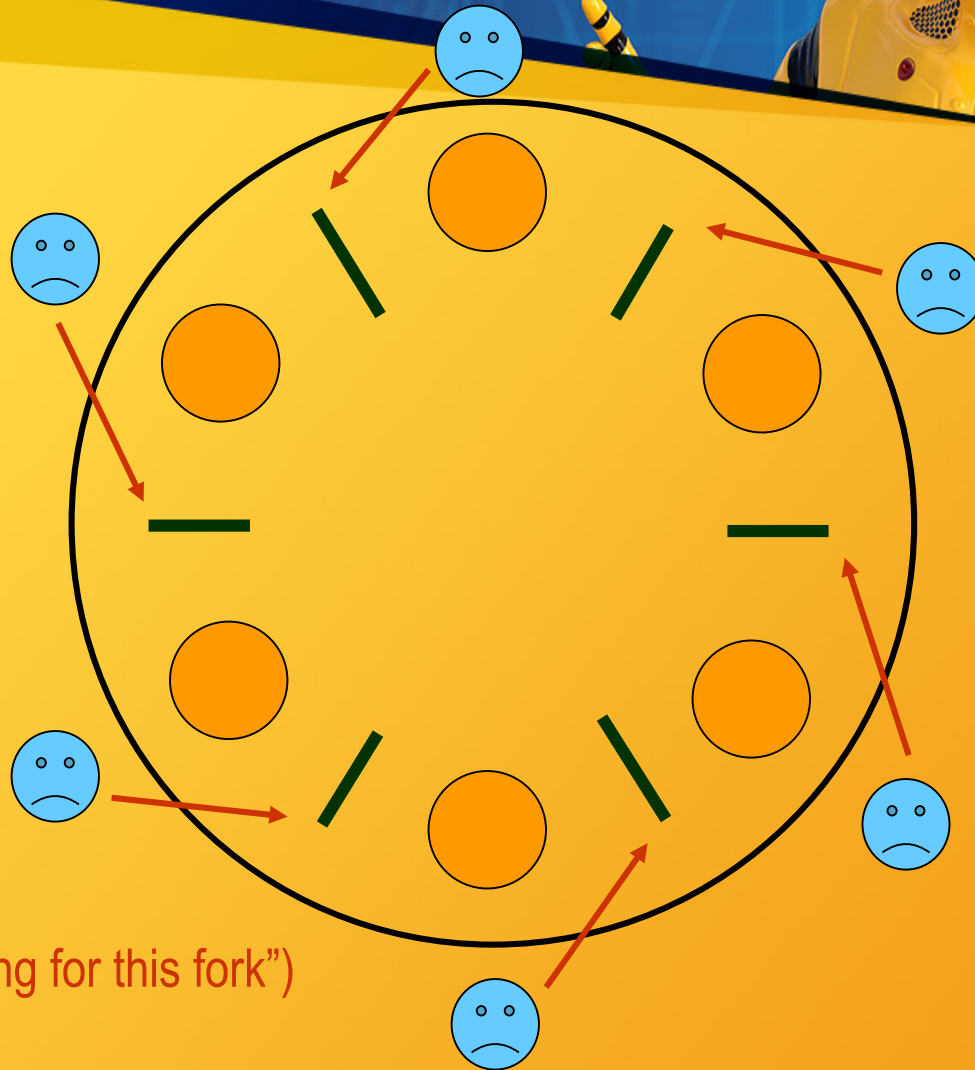


# *Dining philosophers: definition*




- Each process needs two resources
- Every pair of processes compete for a specific resource
  - *A process may proceed only if it is assigned both resources*
- Every process that is waiting for a resource should sleep (be blocked)
- Every process that releases its two resources must wake-up the two competing processes for these resources, if they are interested

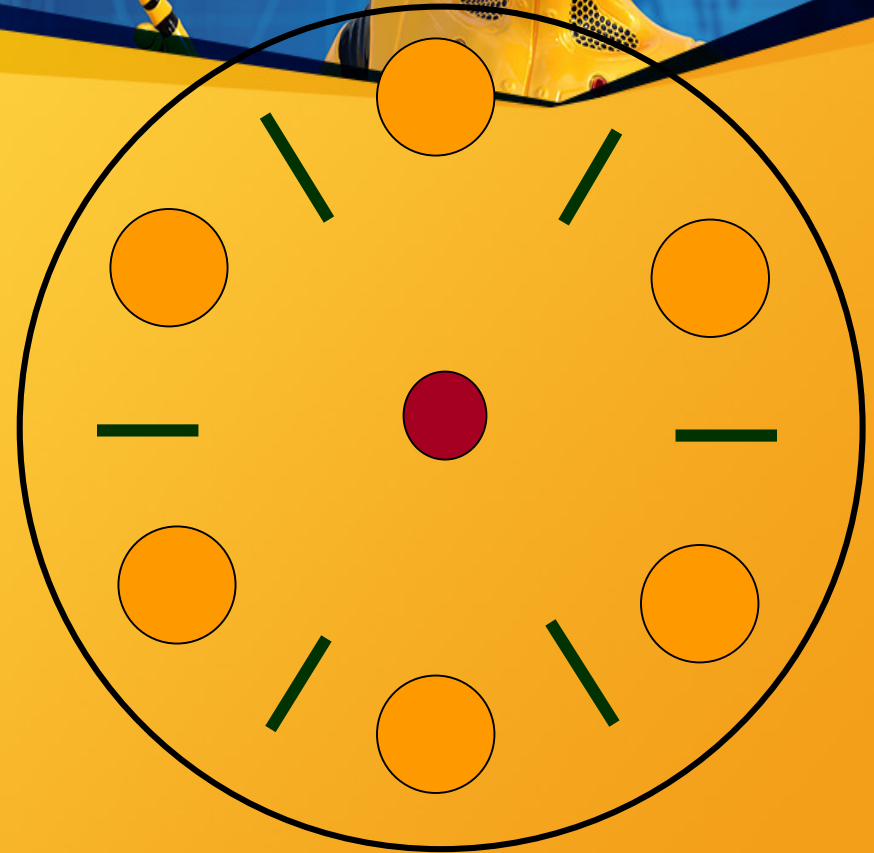
# *An incorrect naïve solution*



( means "waiting for this fork")

# Dining philosophers: textbook solution

- The solution
  - A philosopher first  gets
  - only then it tries to take the 2 forks.



# Process and Threads (cont.)



- Process have a process id (pid) and parent process id (ppid)
  - The main process does not have a parent (i.e. ppid = 0)
  - Multiple processes share the same memory space
    - Pros: easy to communicate
    - Cons: hard to sync (problem when writing)
  - Threads
    - Local sub-process inside of a process
    - May be known or unknown to the OS
  - Multiple processes have their own memory space:
    - Pros: not much syncing required
    - Cons: hard to intercommunicate them



# Writing Process and Threads in C/C++



- Fork() creates a process (child) and invokes it
  - Returns the pid of child to parent
    - Returns pid to child
    - A negative number upon error
- Threads are implemented via pthread.h
  - Simply call the create function and pass the function that will be executed by it

# Some shells



- Unix
  - bash
  - sh
  - csh
- Windows
  - cmd

# Useful Shell Commands (Unix)



- cd: change directory
- ls: list files in directory
- pwd: shows current path
- ifconfig: show/configure network interfaces
  - cat: display content of a file
  - pico/nano/vim: some text editors
    - ps: shows process list
    - top: shows shell task manager
    - ping: test network connectivity
- traceroute: trace hops in a connection
  - ssh: opens ssh connection
    - Help:
      - --help flag on commands
      - Man pages (e.g. man man)

# Useful Shell Commands (Windows)



- cd: change directory
  - dir: list files in directory
- cd with no params: shows current path
- ipconfig: show/configure network interfaces
  - edit: some text editors
  - taskmgr: task manager
  - regedit: OS register modification
- msconfig: shows startup/boot configuration
  - ping: test network connectivity
  - tracert: trace hops in a connection
    - ssh: opens ssh connection
      - Help:
        - Typically /? on commands

# Programming with C/C++ and ASM



- Notation used is AT&T ASM (more common among platforms)
  - Biggest change are explicit size of operand
    - Addb, addw, instead of add
  - Destination and source are exchanged
    - $i = i + 3$
    - add ax, 3 ; INTEL
    - add 3, ax ; AT&T
  - Some notations
    - Binary
    - Decimal
    - Hex
    - Registers
    - Memory

# Basic ASM



- `asm` or `__asm__` directive
  - `asm("command") //in C`
  - E.g. `asm("add ax, 9"); //in C`
    - From ASM
      - `.global myfunction`
    - Then in C invoke `myfunction`



# Extended ASM



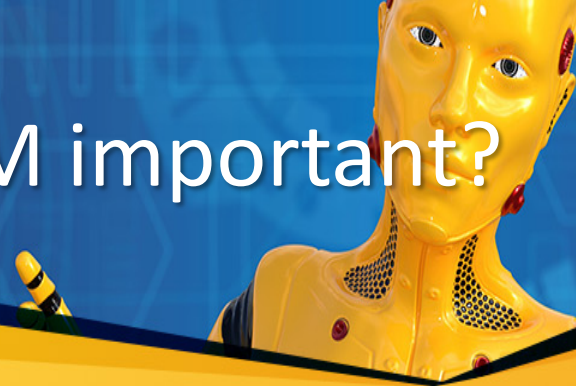
- Allows better communication between C and ASM

- `Asm("command" :`  
output separated by commas :  
input separated by commas :  
optional registers (avoid them));

e.g.

```
int x = 2;
int result = 0;
int c = 3;
__asm__("movl %1, %%eax;"
        "addl $3, %%eax;" : "=a" (result) : "r" (x));
printf("%d\n", result);    //what does this line print?
__asm__("imull %%ebx, %%eax;" : "=a" (result) : "a" (result), "b" (c));
printf("%d\n", result);    //how about this one
```

# Why is C and ASM important?



- Low level programming gives you better control of the machine
  - Allows specifying efficiency in instruction
    - Compiler doesn't throw surprises
      - Faster execution
        - Memory mapping
          - Parts/devices on the machine are mapped to a specific memory

# Operating Systems

## Introduction to Processes



# Process Concept



- Process is a program in execution; forms the basis of all computation; process execution must progress in sequential fashion.
- Program is a passive entity stored on disk (executable file), Process is an active entity; A program becomes a process when executable file is loaded into memory.
- Execution of program is started via CLI entry of its name, GUI mouse clicks, etc.
- A process is an instance of a running program; it can be assigned to, and executed on, a processor.
- Related terms for Process: Job, Step, Load Module, Task, Thread.

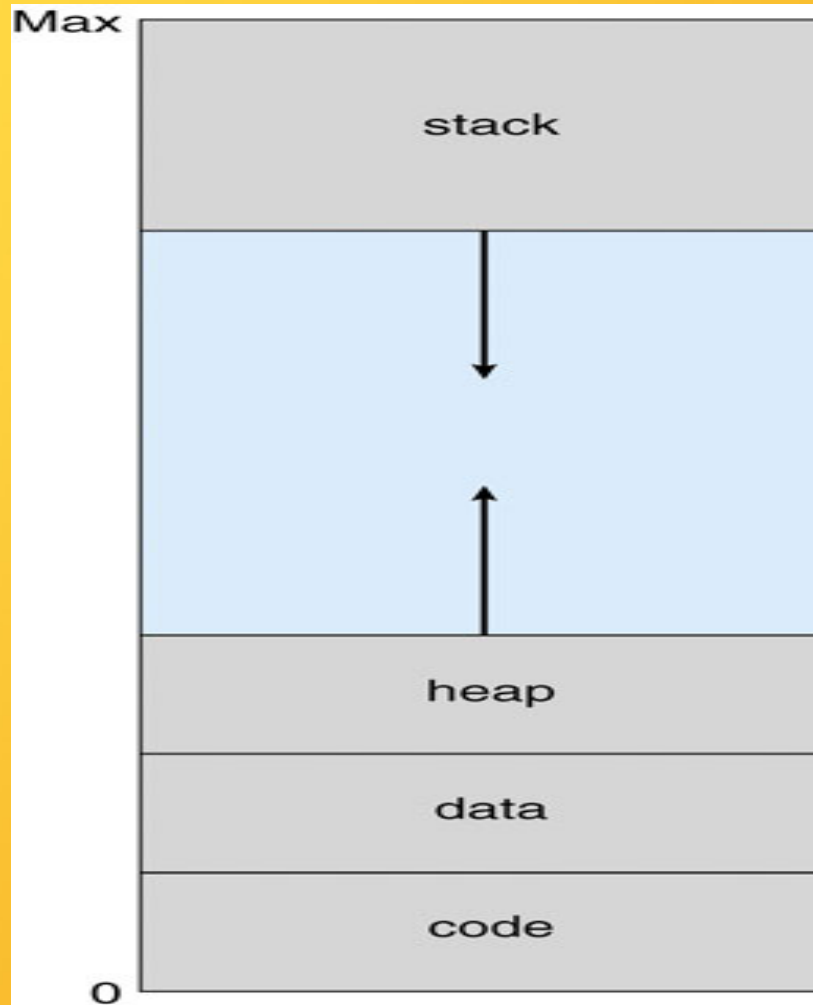


# Process Parts



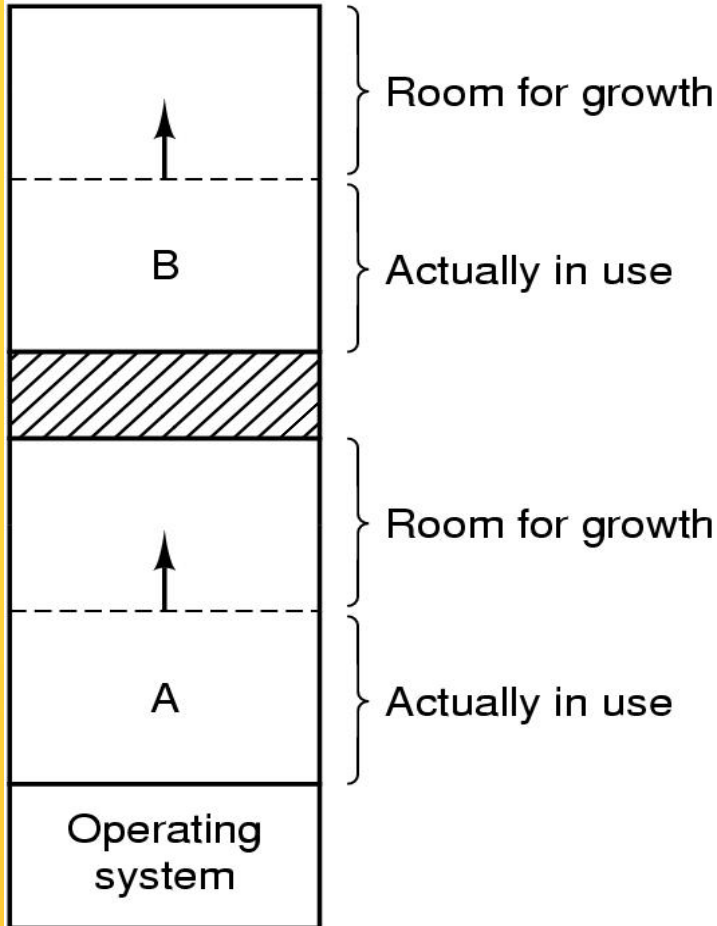
- A process includes three segments/sections:
  1. **Program:** code/text.
  2. **Data:** global variables and heap
    - Heap contains memory dynamically allocated during run time.
  3. **Stack:** temporary data
    - Procedure/Function parameters, return addresses, local variables.
- Current activity of a program includes its **Context:** program counter, state, processor registers, etc.
  - One program can be several processes:
    - Multiple users executing the same Sequential program.
    - Concurrent program running several process.

# Process in Memory (1)

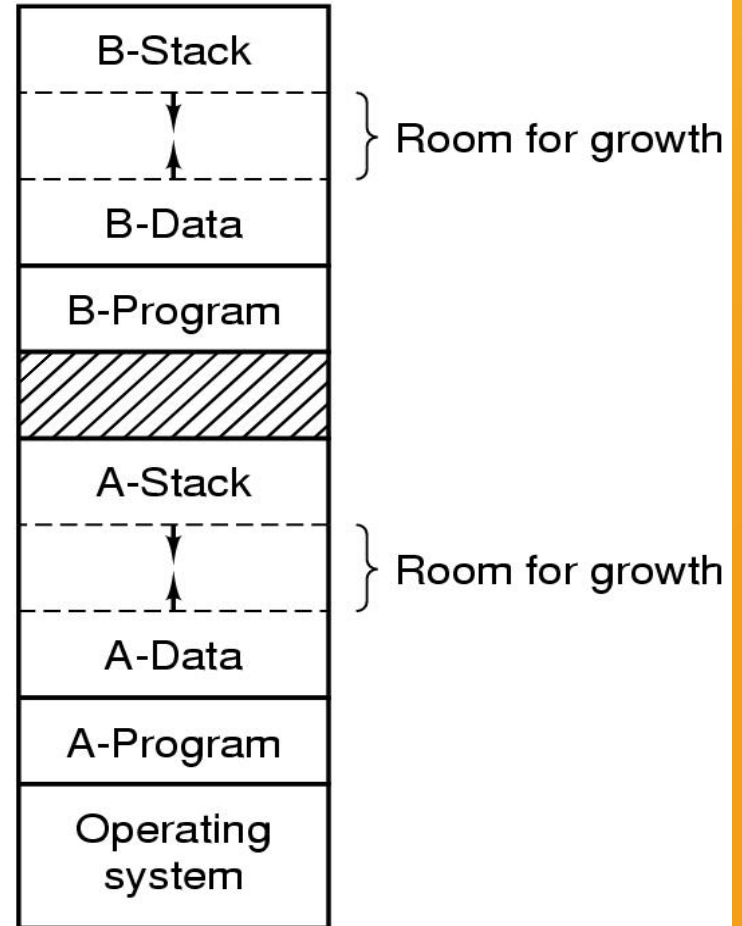




# Processes in Memory (2)

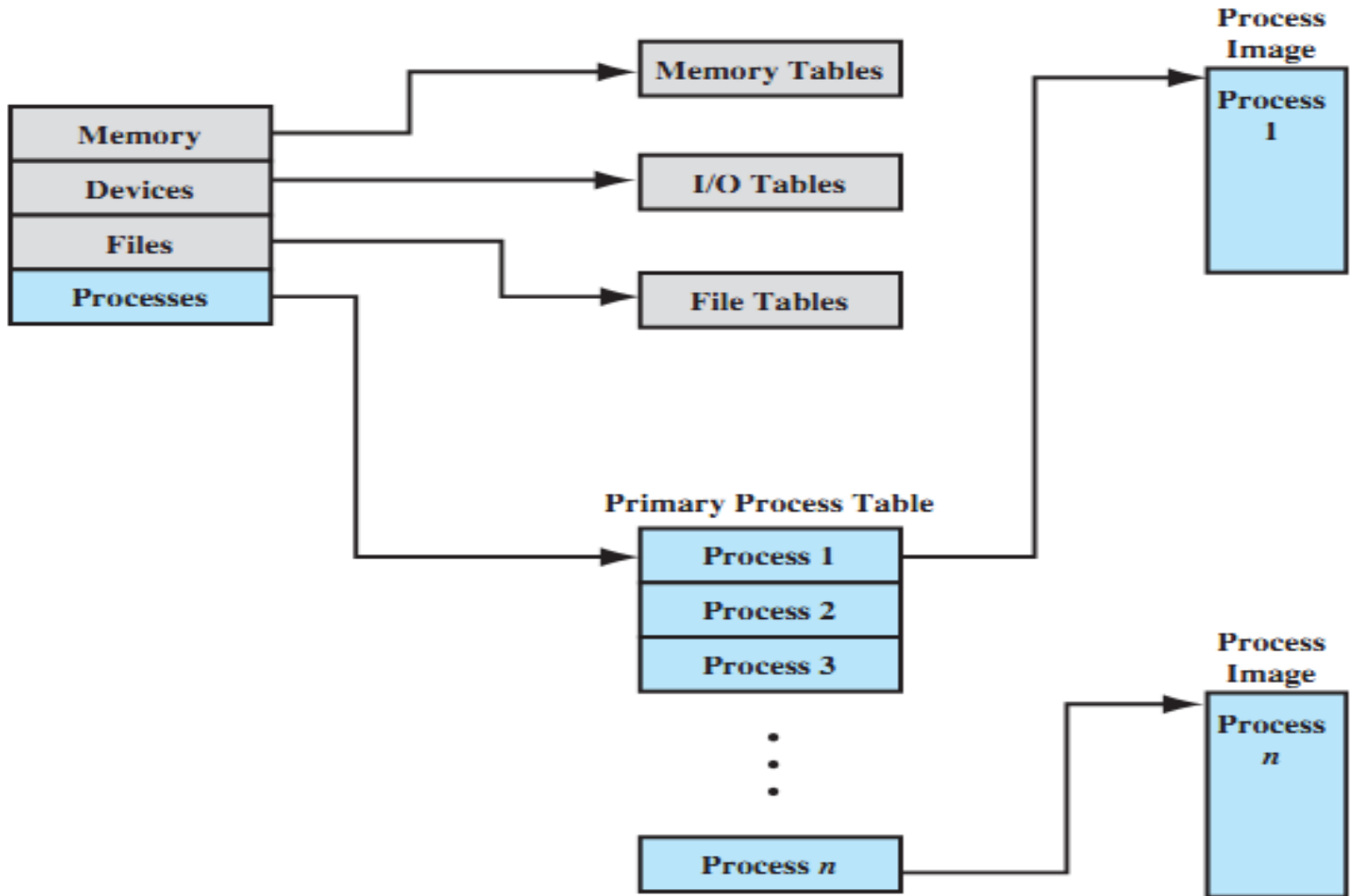


(a)

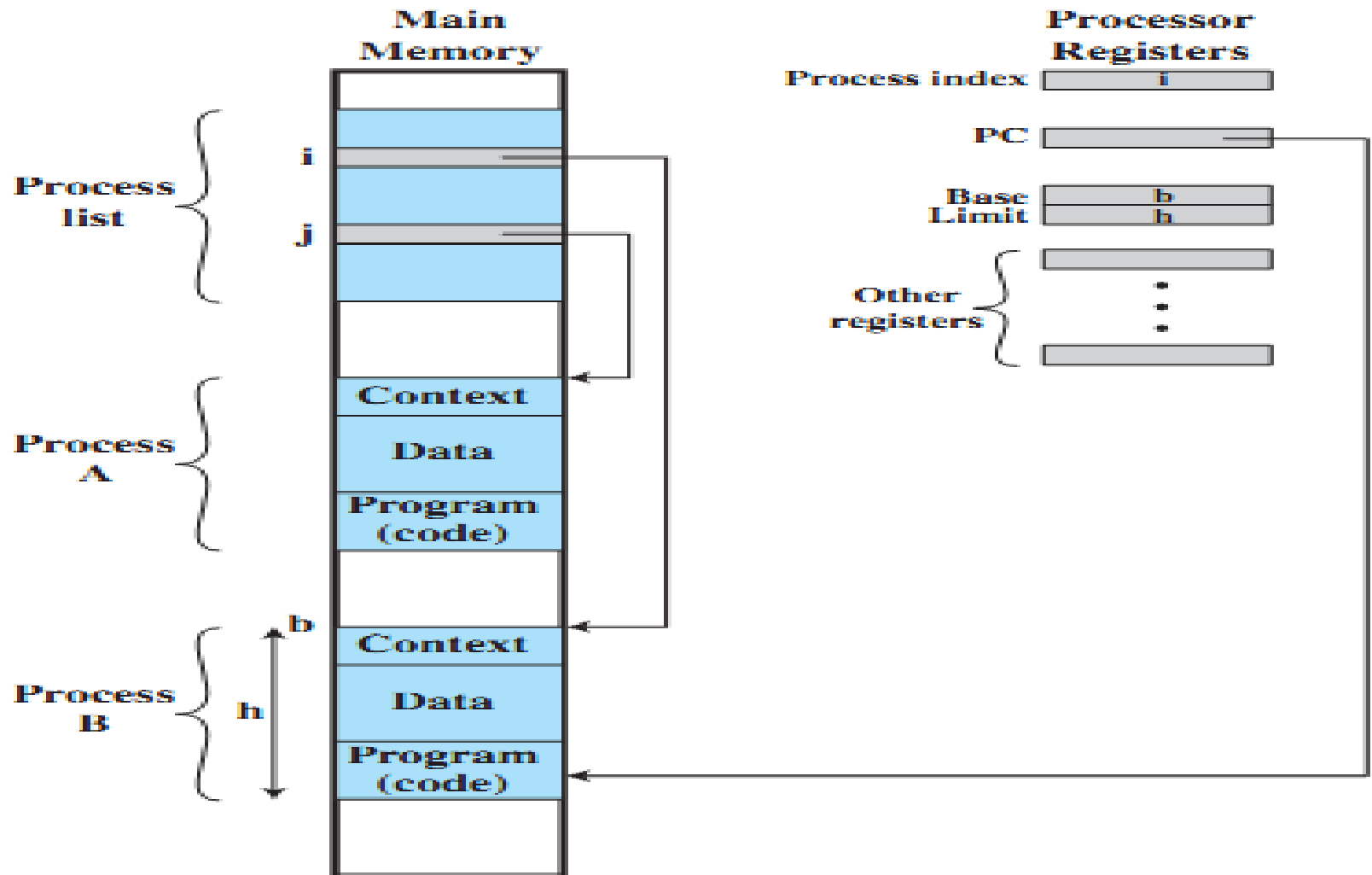


(b)

# General Structure of OS Control Tables



# Typical Process Table Implementation



# Process Attributes

- Process ID
- Parent process ID
- User ID
- Process state/priority
- Program counter
- CPU registers
- Memory management information
- I/O status information
- Access Control
- Accounting information



# Typical process table entry

process state

process number

program counter

registers

memory limits

list of open files

• • •





# Fields of a typical process table entry

## **Process management**

Registers  
Program counter  
Program status word  
Stack pointer  
Process state  
Priority  
Scheduling parameters  
Process ID  
Parent process  
Process group  
Signals  
Time when process started  
CPU time used  
Children's CPU time  
Time of next alarm

## **Memory management**

Pointer to text segment info  
Pointer to data segment info  
Pointer to stack segment info

## **File management**

Root directory  
Working directory  
File descriptors  
User ID  
Group ID




# Components of Process Control Block (PCB)



- Process Control Block (PCB) – IBM name for information associated with each process – its context!
- PCB (execution context) is the data needed (process attributes) by OS to control process:
  1. Process location information
  2. Process identification information
  3. Processor state information
  4. Process control information

# Process Control Block (PCB)



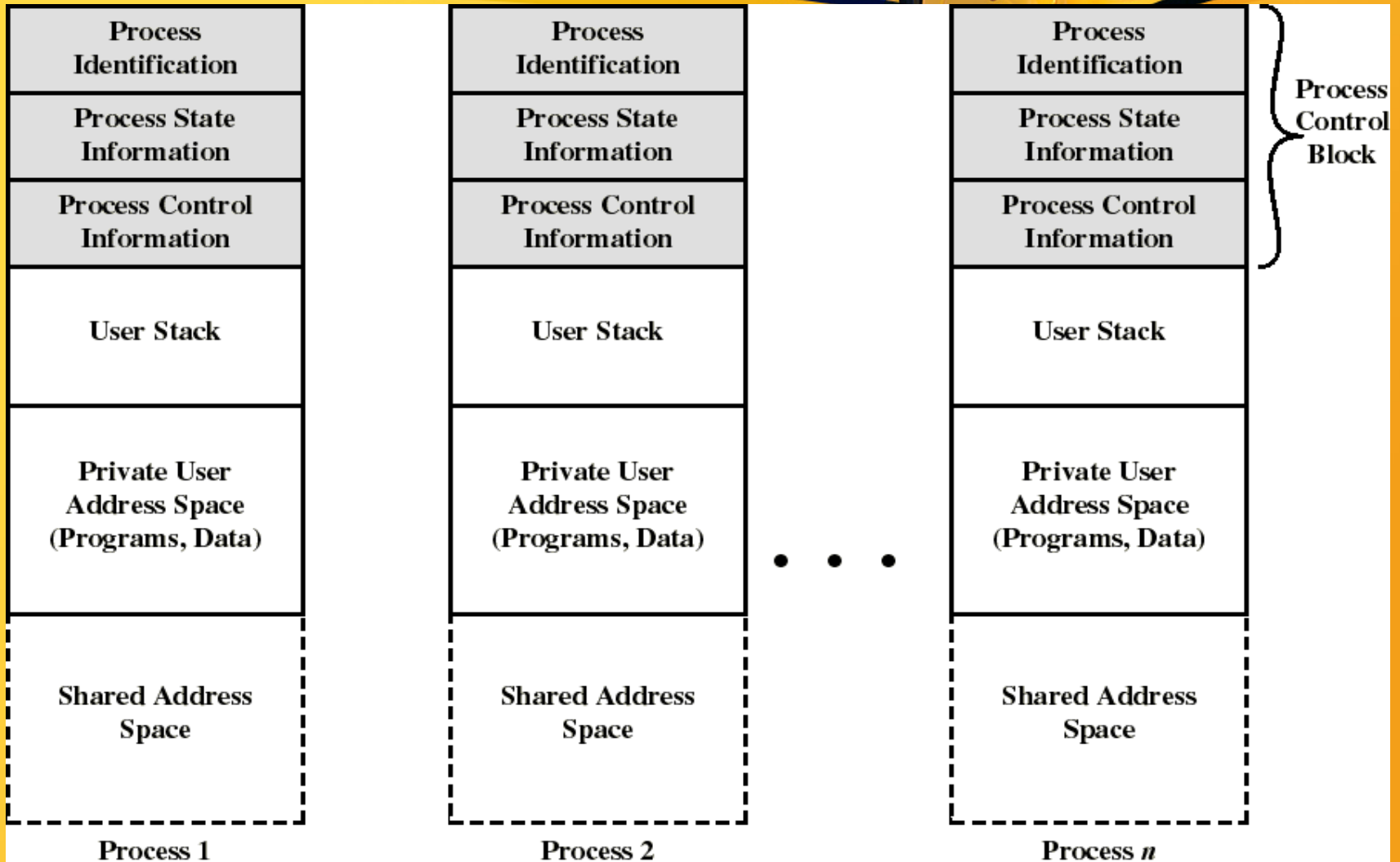
<b>Identifier</b>
<b>State</b>
<b>Priority</b>
<b>Program counter</b>
<b>Memory pointers</b>
<b>Context data</b>
<b>I/O status information</b>
<b>Accounting information</b>
<b>⋮</b>



## Process Location Information

- Each process image in memory:
  - may not occupy a contiguous range of addresses (depends on memory management scheme used).
  - both a private and shared memory address space can be used.
- The location of each process image is pointed to by an entry in the process table.
- For the OS to manage the process, at least part of its image must be brought into main memory.

# Process Images in Memory

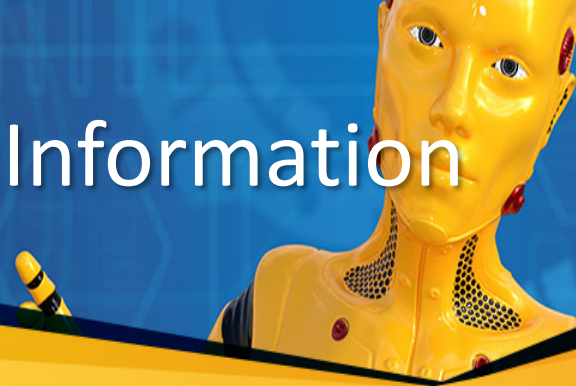


# Process Identification Information



- A few numeric identifiers may be used:
  - Unique process identifier (PID) –
  - indexes (directly or indirectly) into the process table.
    - User identifier (UID) –
      - the user who is responsible for the job.
    - Identifier of the process that created this process (PPID).
- Maybe symbolic names that are related to numeric identifiers.

# Processor State Information



- Contents of processor registers:
  - User-visible registers
  - Control and status registers
  - Stack pointers
- Program Status Word (PSW)
  - contains status information
- Example: the EFLAGS register on Pentium machines.



A yellow humanoid robot with a blue background and a yellow banner. The robot is positioned in the upper right corner, looking towards the viewer. The banner is a curved yellow shape that spans across the top of the slide.

# Process Control Information

- scheduling and state information:
  - Process state (i.e., running, ready, blocked...)
    - Priority of the process
  - Event for which the process is waiting (if blocked).
- data structuring information
  - may hold pointers to other PCBs for process queues, parent-child relationships and other structures.



# Process Control Information

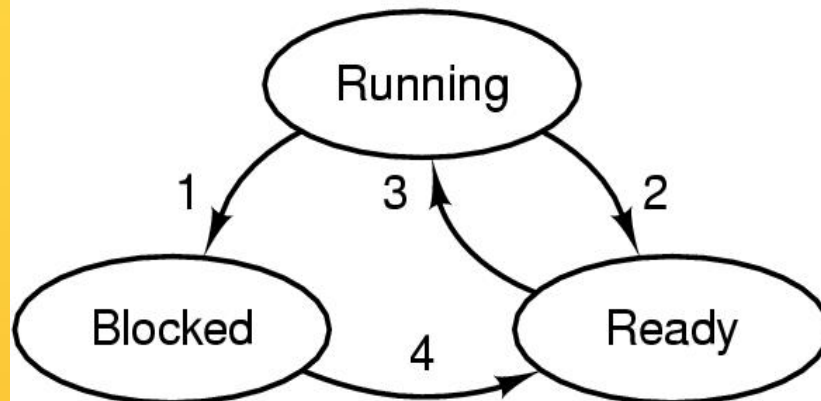
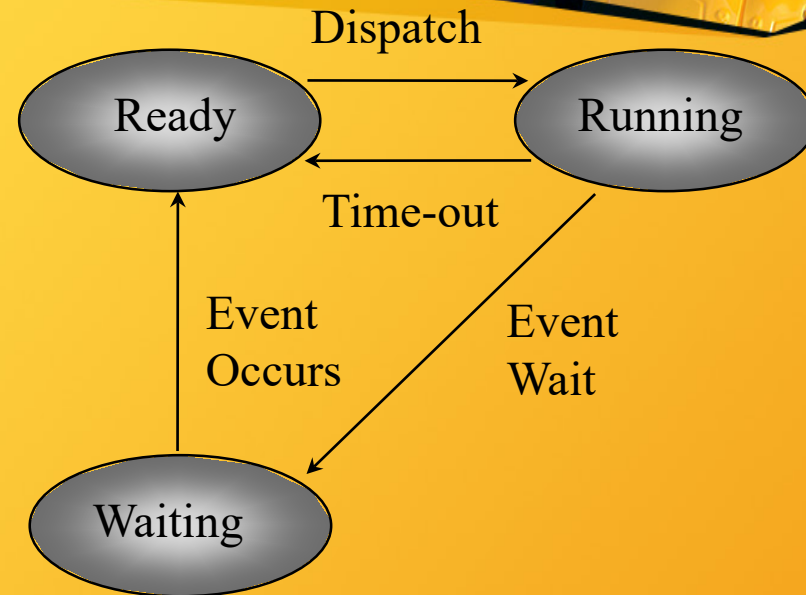
- Inter-process communication –
  - may hold flags and signals for IPC.
- Resource ownership and utilization –
  - resource in use: open files, I/O devices...
  - history of usage (of CPU time, I/O...).
- Process privileges (Access control) –
  - access to certain memory locations, to resources, etc...
    - Memory management –
      - pointers to segment/page tables assigned to this process.

# Process States



- Let us start with three states:
  - 1) Running state –
    - the process that gets executed (single CPU); its instructions are being executed.
  - 2) Ready state –
    - any process that is ready to be executed; the process is waiting to be assigned to a processor.
  - 3) Waiting/Blocked state –
    - when a process cannot execute until its I/O completes or some other event occurs.

# A Three-state Process Model



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

# Process Transitions (1)



- Ready → Running
  - When it is time, the dispatcher selects a new process to run.
- Running → Ready
  - the running process has expired his time slot.
  - the running process gets interrupted because a higher priority process is in the ready state.





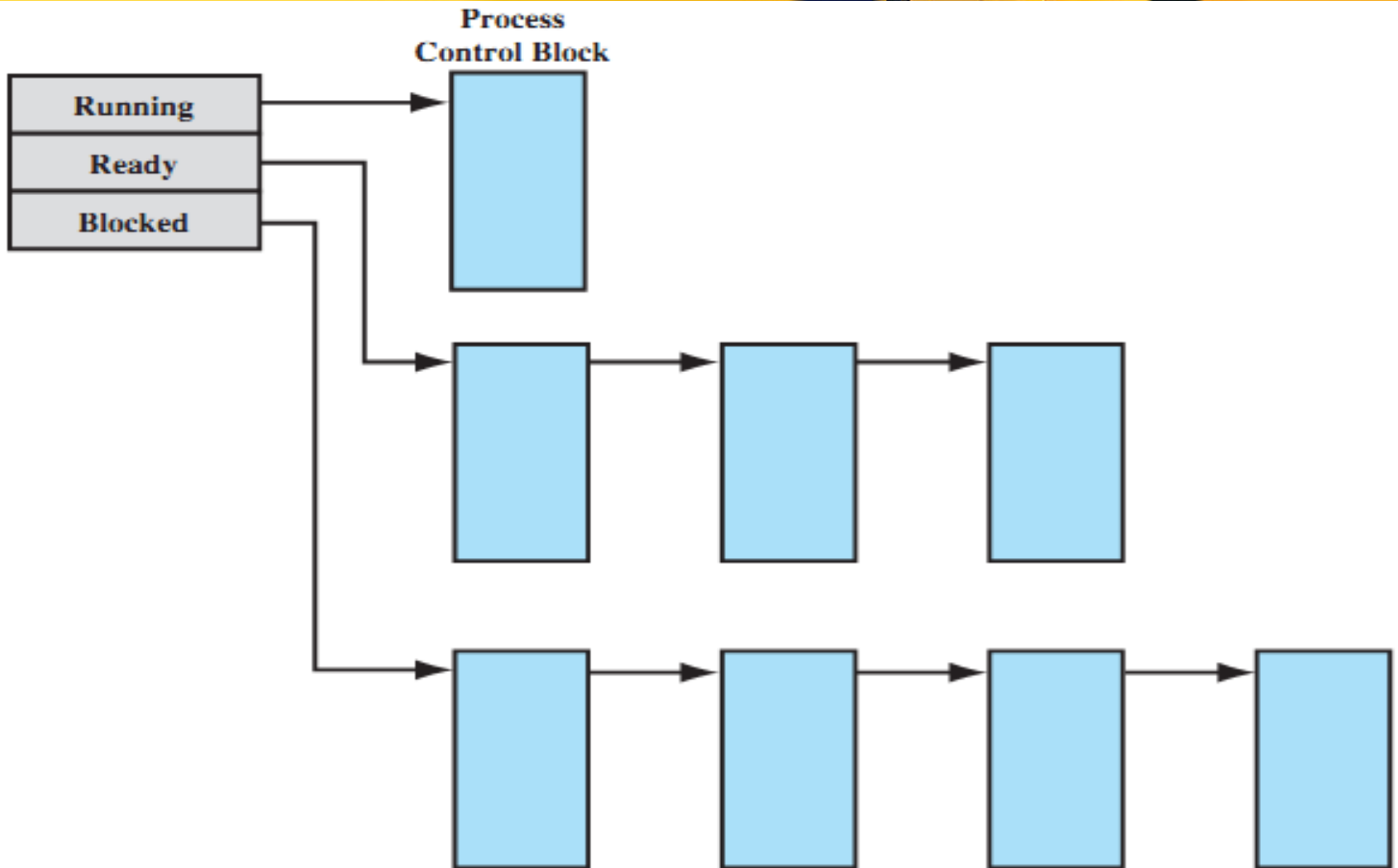
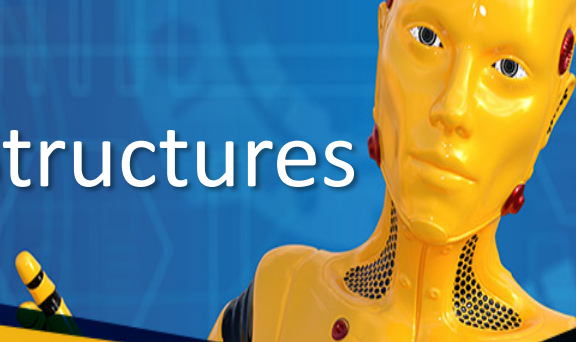
## Process Transitions (2)



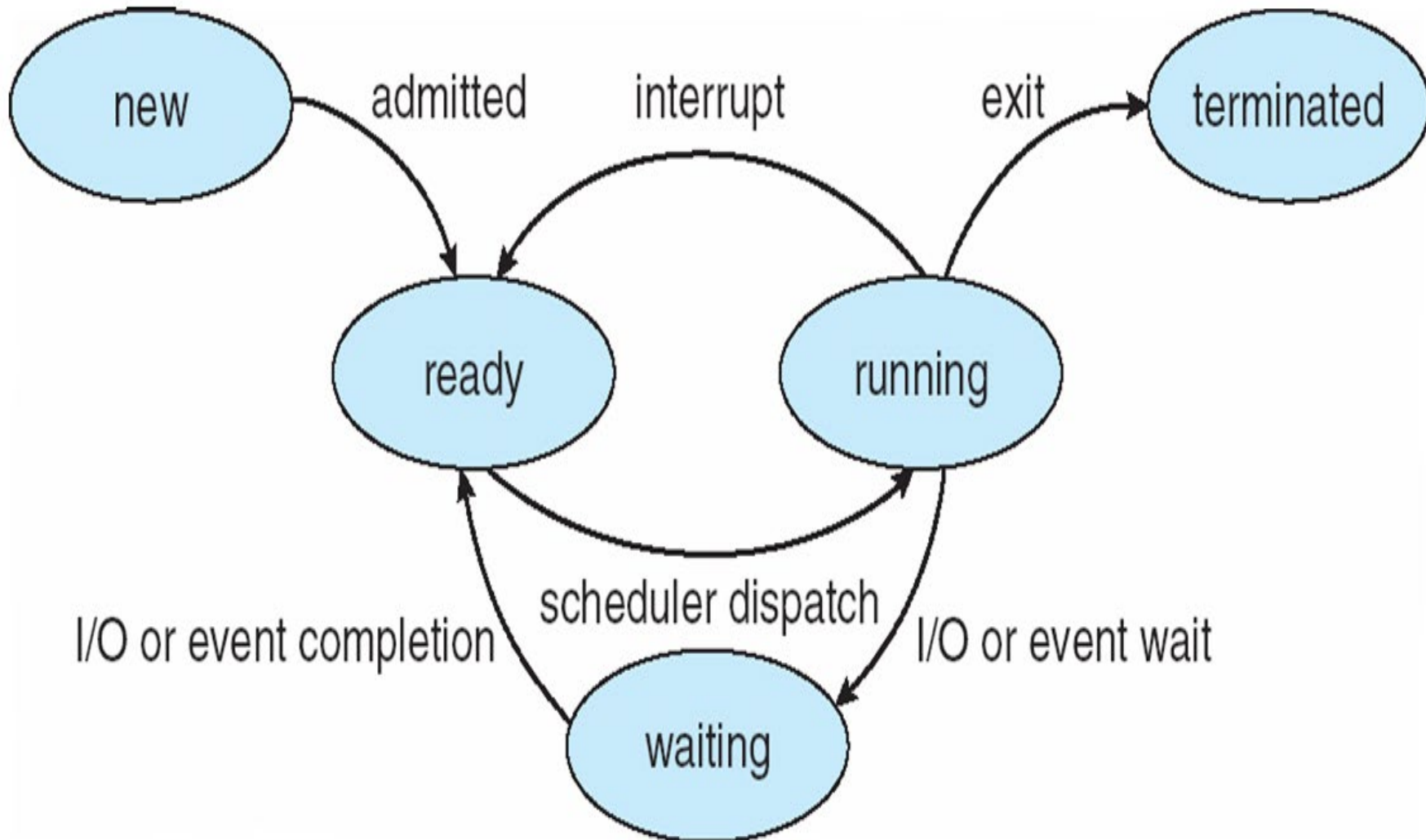
- Running → Waiting
  - When a process requests something for which it must wait:
    - a service that the OS is not ready to perform.
    - an access to a resource not yet available.
    - initiates I/O and must wait for the result.
    - waiting for a process to provide input.
  - Waiting → Ready
    - When the event for which it was waiting occurs.



# Process List Structures



# Five-state Process Model



# Other Useful States (1)



- New state –
  - OS has performed the necessary actions to create the process:
    - has created a process identifier.
    - has created tables needed to manage the process.
  - but has not yet committed to executing the process (not yet admitted):
    - because resources are limited.



## Other Useful States (2)



- Terminated state –
  - Program termination moves the process to this state.
    - It is no longer eligible for execution.
  - Tables and other info are temporarily preserved for auxiliary program –
    - Example: accounting program that cumulates resource usage for billing the users.
- The process (and its tables) gets deleted when the data is no more needed.

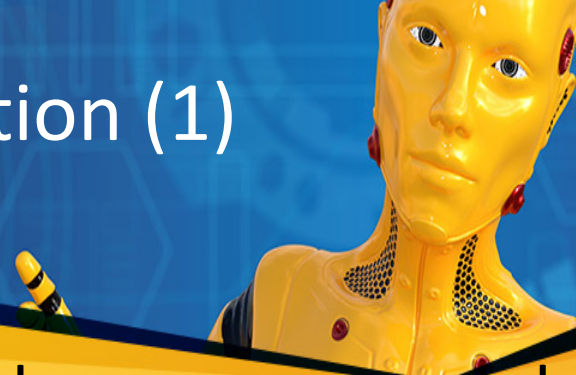
# Reasons for Process Creation

A yellow robot head with a blue background and a yellow banner. The robot has a human-like face with blue eyes and red accents. It is holding a yellow and black object, possibly a pen or a small tool. The background is a blue grid pattern.

- System initialization.
- Submission of a batch job.
  - User logs on.
- Created by OS to provide a service to a user (e.g., printing a file).
  - A user request to create a new process.
    - Spawned by an existing process
- a program can dictate the creation of a number of processes.



# Process Creation (1)

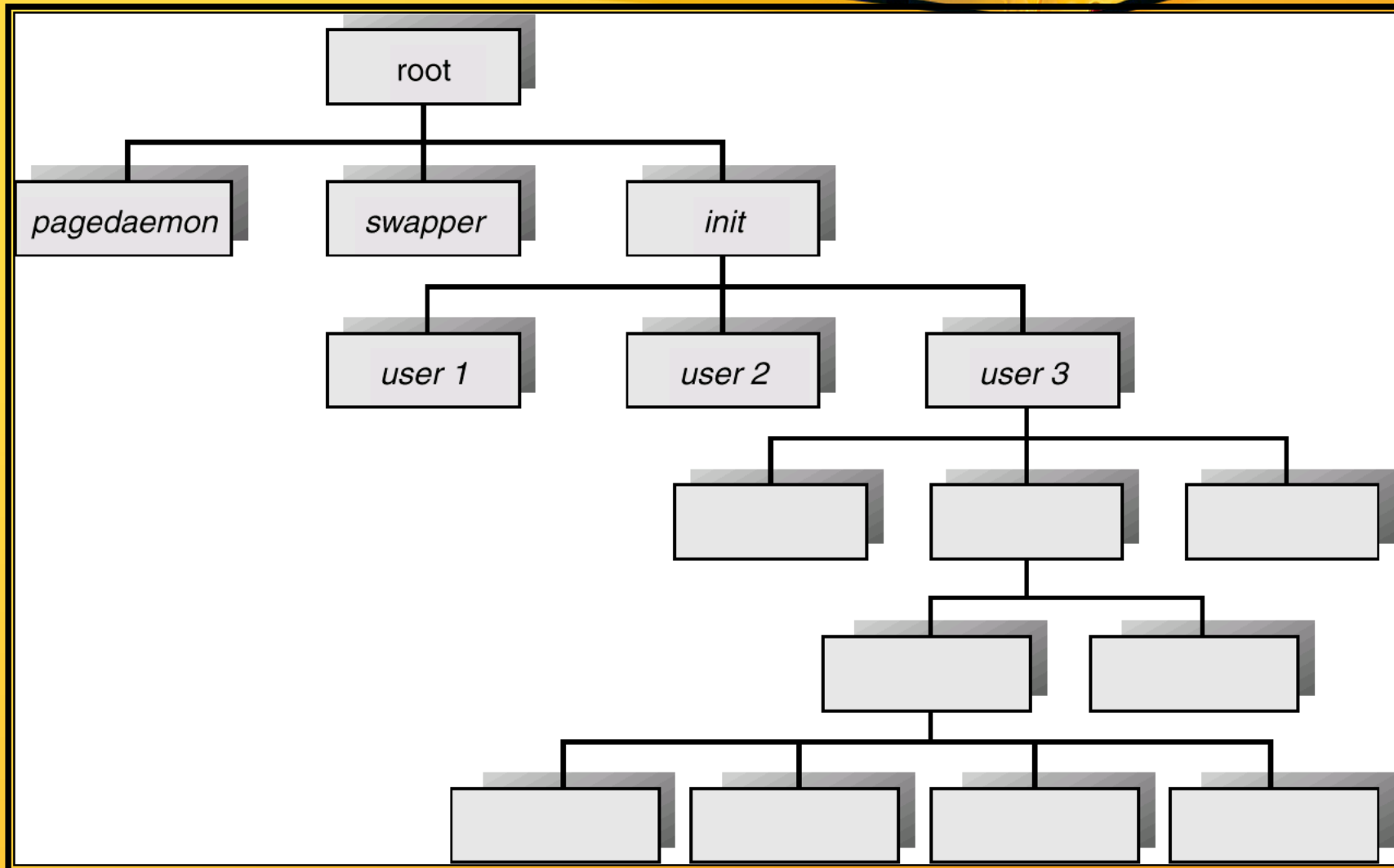


- Parent process create children processes, which, in turn create other processes, forming a tree of processes.
- Possible resource sharing:
  - Parent and children share all resources.
  - Children share subset of parent's resources.
  - Parent and child share no resources.
- Possible execution:
  - Parent and children execute concurrently.
  - Parent waits until children terminate.

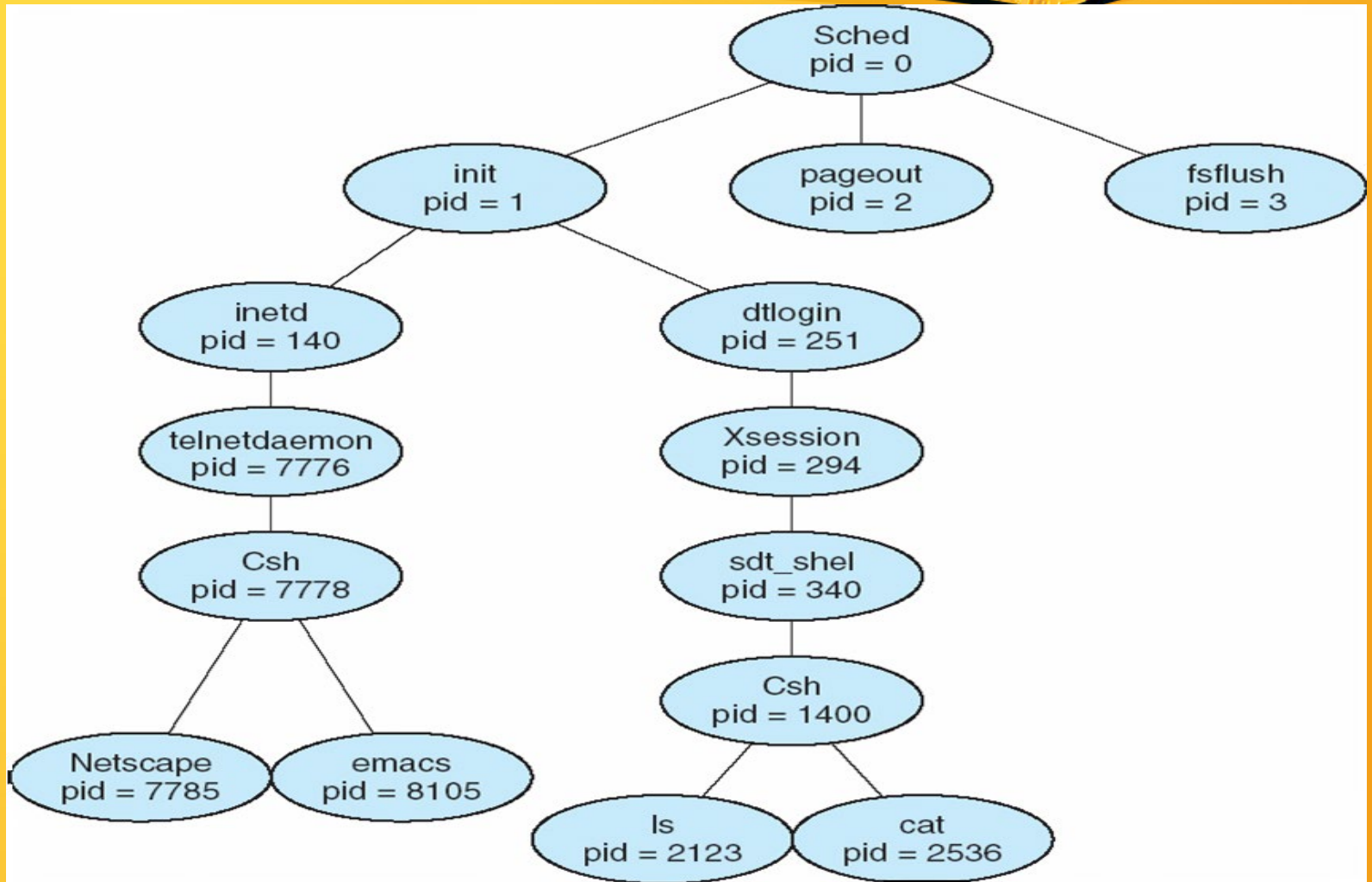




# A tree of processes on UNIX



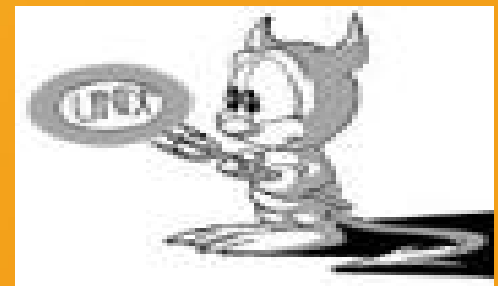
# A tree of processes on typical Solaris



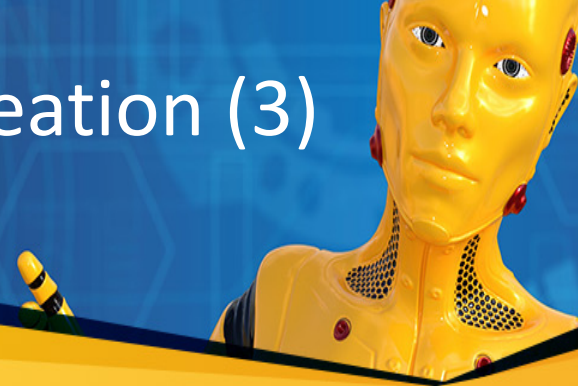
## Process Creation (2)



- Assign a unique process identifier (PID).
- Allocate space for the process image.
- Initialize process control block
  - many default values (e.g., state is New, no I/O devices or files...).
- Set up appropriate linkages
  - Ex: add new process to linked list used for the scheduling queue.



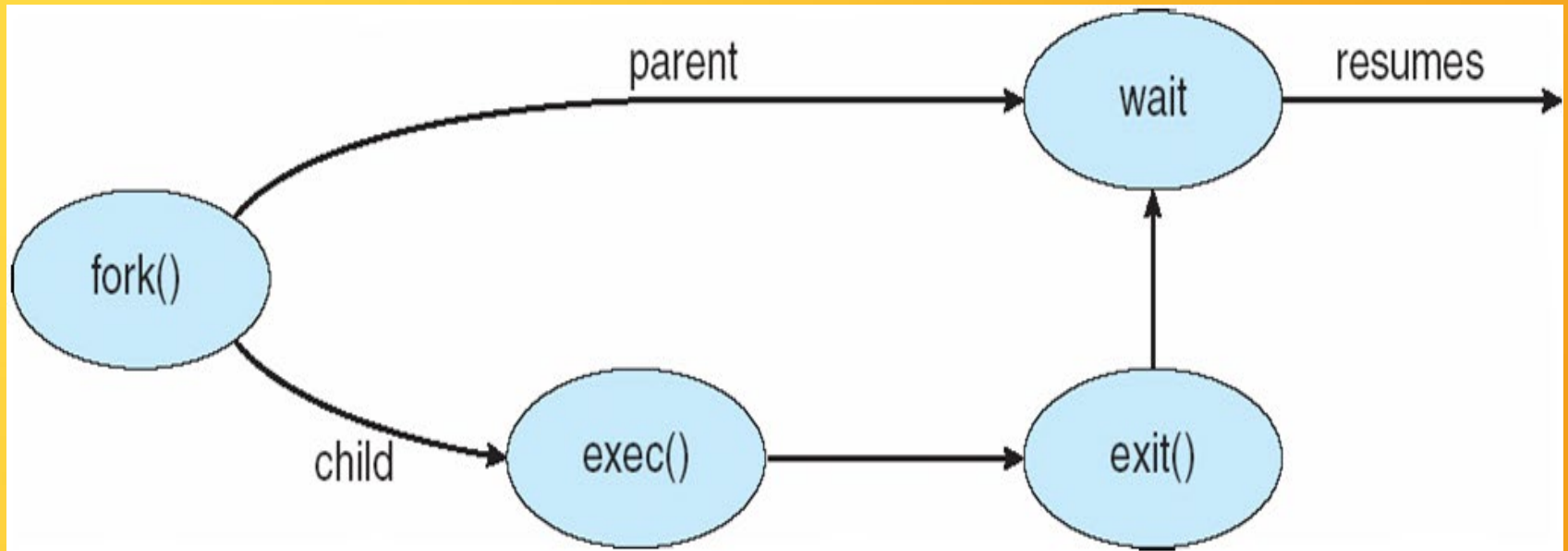
# Process Creation (3)



- Address space
  - Child duplicate of parent.
  - Child has a program loaded into it.
- UNIX examples
  - **fork** system call creates new process.
  - **exec** system call used after **fork** to replace the memory space new program.



# Process Creation (3)





# C Program Forking Separate Process

```
int main()
{
    pid_t pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/lS", "lS", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait (NULL);
        printf ("Child Complete");
        exit(0);
    }
}
```



# When does a process get terminated?



- Batch job issues *Halt* instruction.
- User logs off.
- Process executes a service request to terminate.
- Parent kills child process.
- Error and fault conditions.



# Reasons for Process Termination (1)



- Normal/Error/Fatal exit.
- Time limit exceeded
- Memory unavailable
- Memory bounds violation
- Protection error
  - example: write to read-only file
- Arithmetic error
- Time overrun
  - process waited longer than a specified maximum for an event.



# Reasons for Process Termination (2)

- I/O failure
- Invalid instruction
  - happens when trying to execute data
  - Privileged instruction
  - Operating system intervention
    - such as when deadlock occurs.
  - Parent request to terminate one child.
- Parent terminates so child processes terminate.



# Process Termination



- Process executes last statement and asks the operating system to terminate it (**exit**):
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of child processes (**abort**):
  - Child has exceeded allocated resources.
  - Mission assigned to child is no longer required.
    - If Parent is exiting:
- Some OSs do not allow child to continue if its parent terminates.
  - Cascading termination – all children terminated.

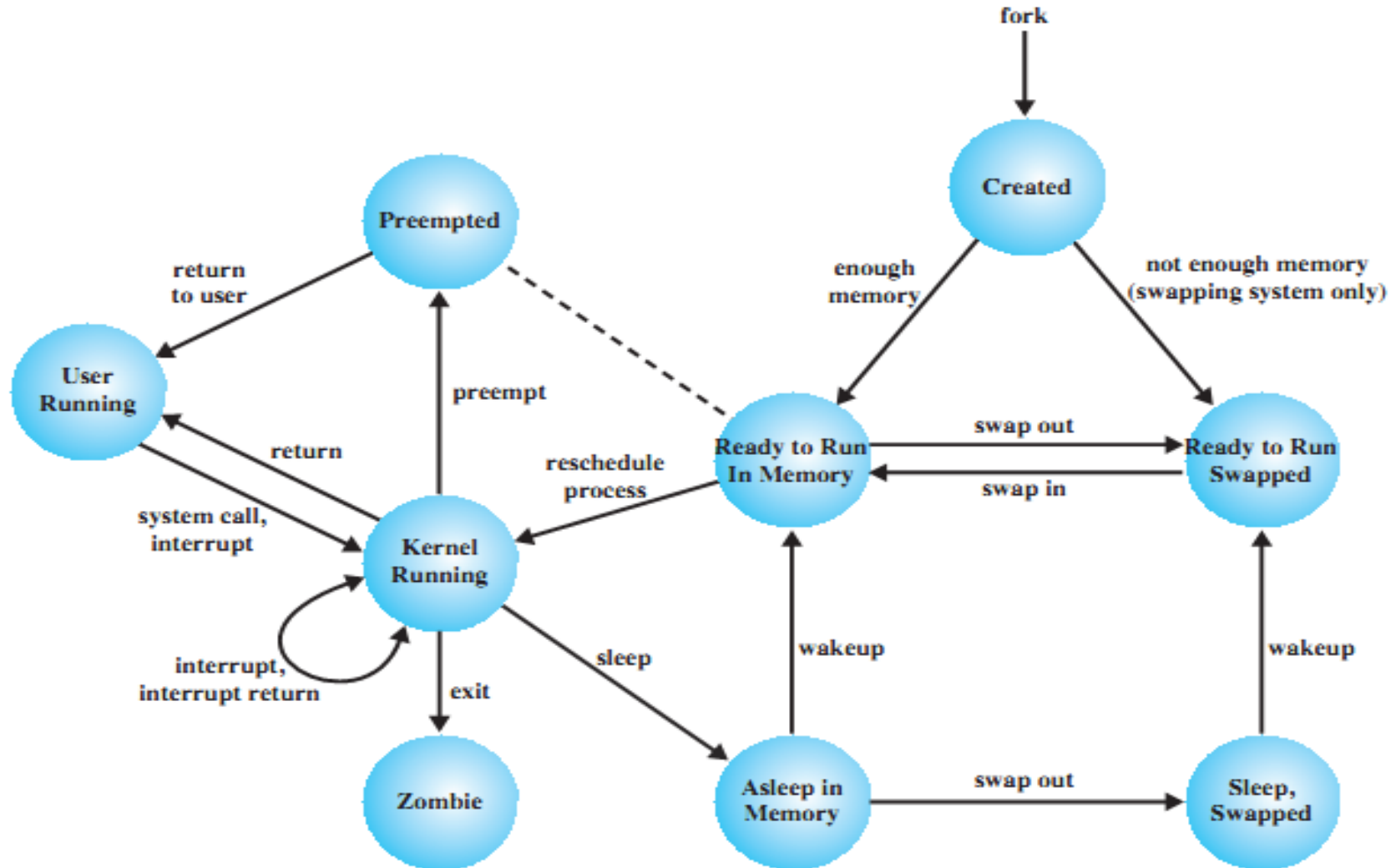
# UNIX SVR4 Process States



- **User Running** Executing in user mode.
- **Kernel Running** Executing in kernel mode.
- **Ready to Run, in Memory** Ready to run as soon as the kernel schedules it.
- **Asleep in Memory** Unable to execute until an event occurs; process is in main memory (a blocked state).
- **Ready to Run, Swapped** Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
- **Sleeping, Swapped** The process is awaiting an event and has been swapped to secondary storage (a blocked state).
- **Preempted** Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
  - **Created** Process is newly created and not yet ready to run.
- **Zombie** Process no longer exists, but it leaves a record for its parent process to collect.



# UNIX SVR4 States Process Model







# UNIX Process Control

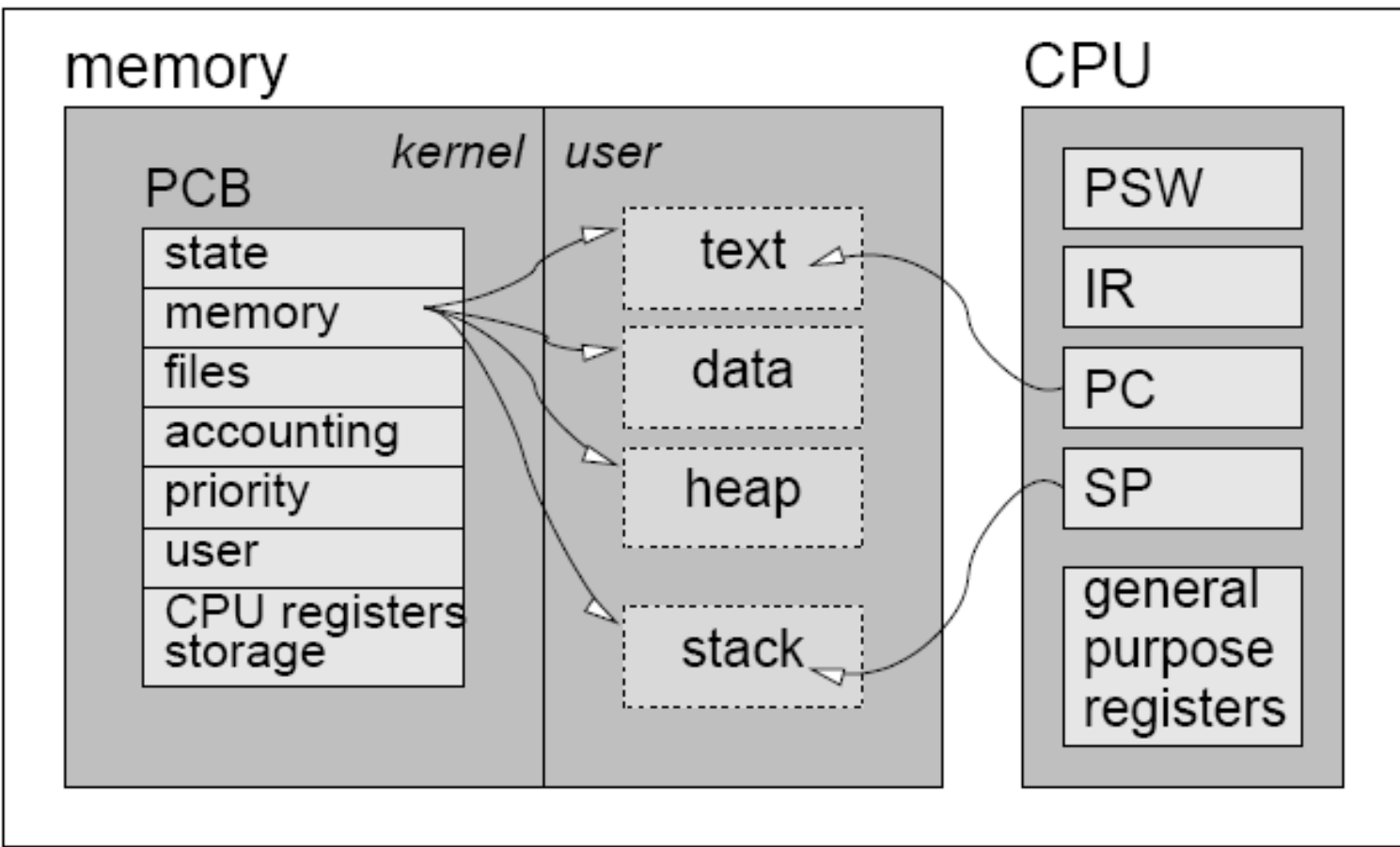
# Process: A Context for Computation



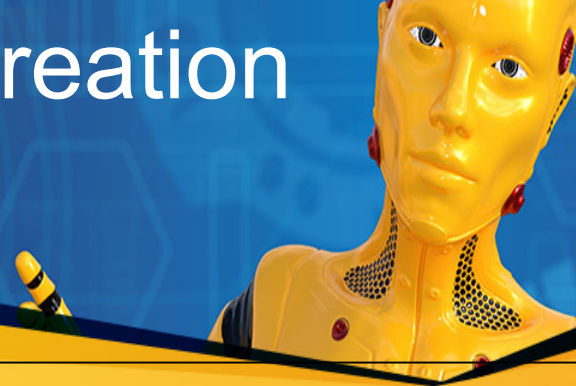
A process is largely defined by:

- Its CPU state (**register** values).
- Its address space (**memory** contents).
- Its **environment** (as reflected in operating system tables).

# Process layout



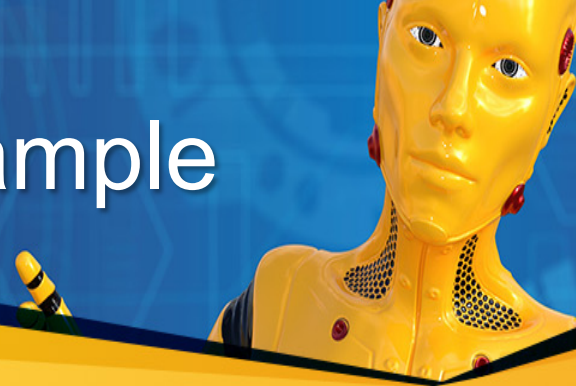
# Process Creation



```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- System call:
  - Child - 0.
  - Parent - PID of the child.
- System call algorithm:
  - Allocate a slot in process table.
  - Allocate PID.
  - Create a logical copy of the parent context.
  - Return values.

# Fork Example



```
if ( (pid = fork()) < 0 )  
    error  
    If (pid == 0)  
    { code for child }  
    else  
    { code for parent }
```

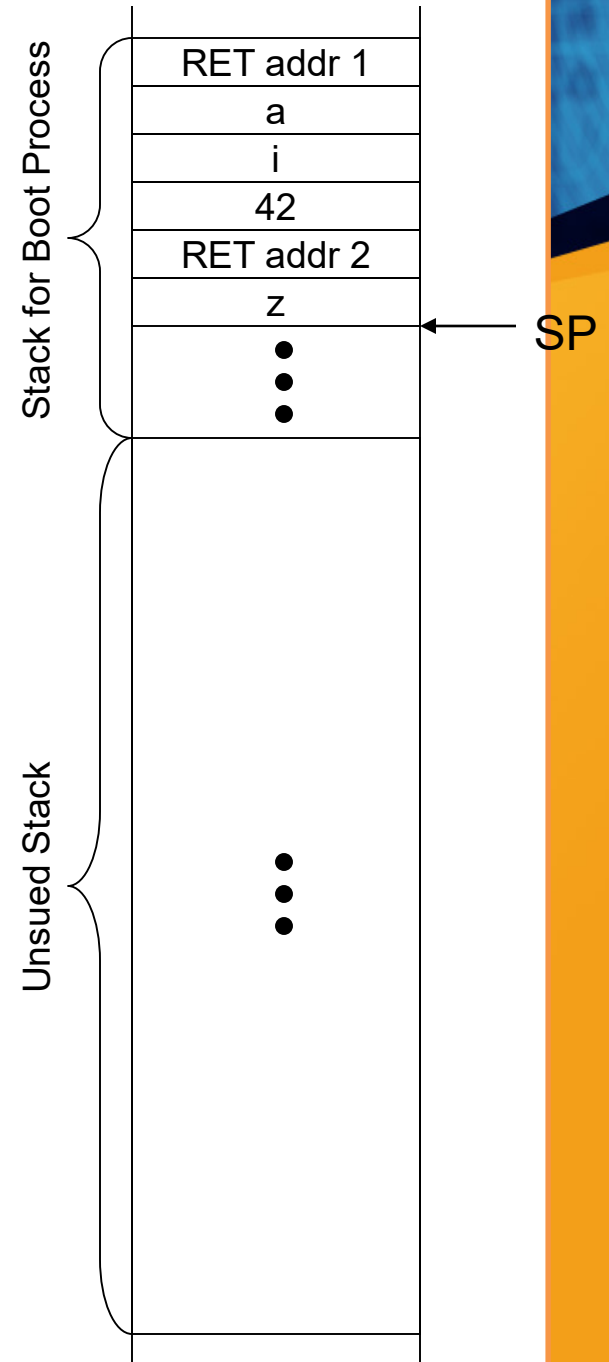


# Before calling fork()

```
void f (int x)
{
    int z;
    fork(); ← Before fork()
}
```

```
void g()
{
    char a;
    int i;
    f (42); ← RET addr 2
}
```

```
int main()
{
    g(); ← RET addr 1
    return 0;
}
```



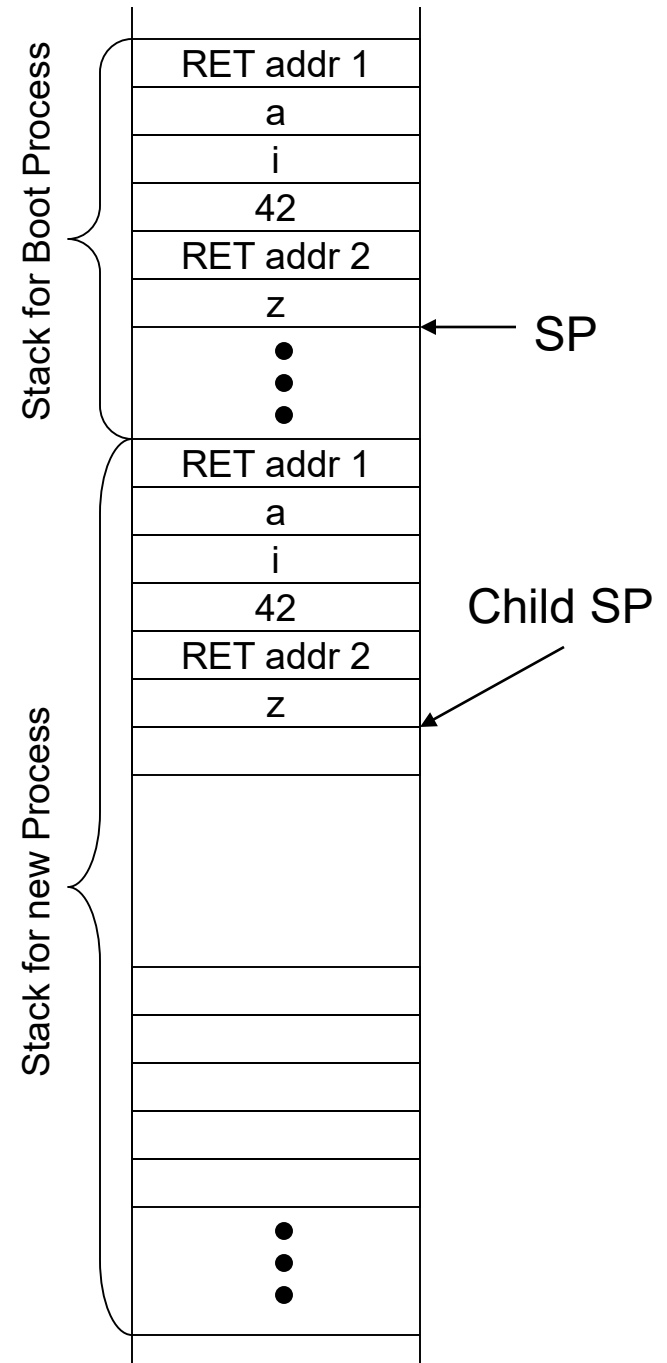


# After calling fork()

```
void f (int x)
{
    int z;
    fork(); ← Location 3
}

void g()
{
    char a;
    int i;
    f (42); ← RET addr 2
}

int main ()
{
    g(); ← RET addr 1
    return 0;
}
```



# Example: Race Conditions



```
#include <iostream>
#include <unistd.h>
#include <sys/types.h>

int main()
{
    pid_t pid;
    if ((pid = fork()) < 0 )
        exit(1); //error
    if(pid != 0) {
        for(int i = 0; i < 100; i++)
            cout<<"Parent process " << i <<endl;
    } else {
        for(int i = 100; i < 200; i++)
            cout <<"Child process " << i << endl;
    }
    return 0;
}
```

# Process Termination



- Normal Termination:
  - Executing a return from the main function.
    - Calling the exit function. (ANSI C)
    - Calling the \_exit function. (Sys call)
  - Abnormal Termination:
    - When a process receives certain signals.

# Normal Process Termination



```
#include <stdlib.h>
void exit (int status);
```

- System call:
  - Status: IPC.
  - NEVER returns.
- System call algorithm:
  - Mask Signals.
  - Close open files.
  - Release memory.
  - Save the process exit status.
  - Set the process state to ZOMBIE.

# Awaiting Process Termination Bach



```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- **System call:**
  - Returns the PID of a zombie child -1 when no children exist.
  - Status is the exit status of the child process.
  - Wait can block the caller until a child process terminates.
- **System call algorithm:**
  - Search for a zombie child of the process.
  - Extract PID and status of the zombie child.
  - Release the process table slot.

# Orphans and Zombies



- When a child exits when its parent is not currently executing a `wait()`, a *zombie* emerges.
  - A zombie is not really a process as it has terminated but the system retains an entry in the process table.
  - A zombie is put to rest when the parent finally executes a `wait()`.
- A child process whose parent has terminated is referred to as *orphan*.
- When a parent terminates, orphans and zombies are adopted by the *init* process (`process-id:0`) of the system.



# Invoking other programs



- The *exec* invokes another program, overlaying the memory space with a copy executable file.
  - The contents of the user-level context is accessible through *exec* parameters.

```
exec(filename, argv, envp);
```

# notable `exec` properties



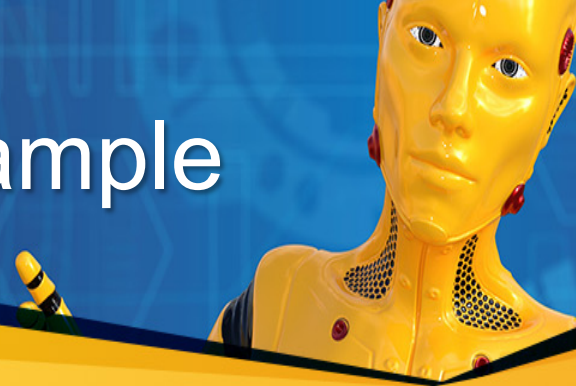
- an `exec` call transforms the calling process by loading a new program in its memory space.
- the `exec` does not create a new sub-process.
  - unlike the *fork* there is no return from a successful `exec`.

# System call algorithm



- Determine the file properties
  - Determine If the file is an executable.
  - Determine if the user has permissions.
    - Determine the file's layout.
- Copy exec arguments to system space.
  - Detach old memory regions.
    - Allocate new regions.
    - Copy exec arguments.

# Exec example



```
If((pid = fork()) < 0)
    error;
if (pid== 0 ) {
    exec( arguments );
    exit(-1);
}
// parent continues here
```

# fork/wait/execv Example

```
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;
int main()
{
    int x = 42;
    pid_t pid;
    if ((pid = fork()) < 0 )
        exit(1);
    if(pid != 0) {
        int status;
        cout << "Parent process. x=" << x << endl;
        wait (&status);
    } else {
        cout << "Child process. x=" << x << endl;
        char* args[] = {"ls", NULL};
        execv ("/bin/ls", args);
        cout << "Never reached" << endl;
    }
    return 0;
}
```

# The Shell



- The shell read a command line from STDIN and execute.
  - The shell has to main commands:
    - Internal commands. (cd)
    - External commands. (cp)
  - External commands may run foreground/background.



# Signals



- Signals are notifications sent to a process in order to notify the process of events.
  - Kernel.
  - Processes (system call kill)
- The kernel send a signal by setting a bit in the field of the process table entry.
  - A process can remember different types of signals, but not the number of signals from each type.

# Sending Signals



Using the keyboard:

- Ctrl-C: Causes the system to send an `INT` signal (`SIGINT`) to the running process.

Using shell kill command:

- The *kill* command has the following format:  
*kill [options] pid*

# Handling Signals



- The kernel handles signals in the context of the process that receives them so process must run to handle signals.
- There are three case for handling signals:
  - The process exits. (default action)
  - The process ignores.
  - The process execute particular function.  
`oldfun = signal(signum,newfun);`

# Non-Catchable Signals



- Most signals may be caught by the process, but there are a few signals that the process cannot catch, and cause the process to terminate.
  - For example: `KILL` and `STOP`.
- If you install no signal handlers of your own the runtime environment sets up a set of default signal handlers.
  - For example:
    - The default signal handler for the `TERM` signal calls the `exit()`.
    - The default handler for the `ABRT` is to dump the process's memory image into a file, and then exit.

# Summary



1. Each signal may have a **signal handler**, which is a function that gets called when the process receives that signal.
2. When the signal is sent to the process, the operating system stops the execution of the process, and "**forces**" it to call the signal handler function.
3. When that signal handler function **returns**, the process continues execution from wherever it happened to be before the signal was received, as if this interruption never occurred.

# Signal Handlers - Example



```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void catch_int(int sig_num) {
    signal(SIGINT, catch_int); //install again!
    printf("Don't do that\n");
    fflush(stdout);
}

int main(int argc, char* argv[]) {
    signal(SIGINT, catch_int);
    for ( ;; )
        pause(); //wait till receives a signal.
}
```



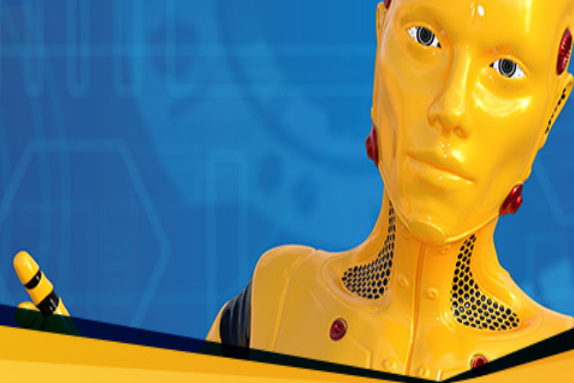
# Avoiding Signal Races - Masking Signals



- The occurrence of a second signal while the signal handler function executes.
  - The second signal can be of different type than the one being handled, or even of the same type.
- The system also contains some features that will allow us to block signals from being processed.
  - A global context which affects all signal handlers, or a per-signal type context.



- `#include <stdio.h>`
- `#include <stdlib.h>`
- `#include <unistd.h>`
  
- `int main(int argc, char *argv[]) {`
- `int pipefd[2];`
- `pid_t ls_pid, wc_pid;`
  
- `pipe(pipefd);`
  
- `if ((ls_pid = fork()) == 0) {`
  
- `dup2(pipefd[1],STDOUT_FILENO);`
- `close(pipefd[0]);`
  
- `execl("/bin/l", "l", 0);`
- `perror("exec l failed");`
- `exit(EXIT_FAILURE);`
- `}`



```
close(pipefd[0]);  
close(pipefd[1]);  
int status;  
int pid = waitpid(ls_pid, &status, 0);  
pid = waitpid(wc_pid, &status, 0);
```