CSCE 313-200
Introduction to Computer Systems
Spring 2025

Processes

Dmitri Loguinov
Texas A&M University

January 23, 2025

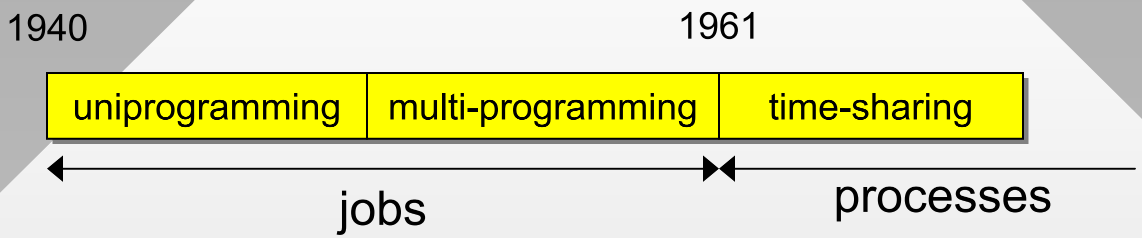
Chapter 3: Roadmap

- ➡ 3.1 What is a process?
- 3.2 Process states
- 3.3 Process description
- 3.4 Process control
- 3.5 Execution of the OS
- 3.6 Security issues
- 3.7 Unix process management

Part II

- Chapter 3: Processes
- Chapter 4: Threads
- Chapter 5: Concurrency
- Chapter 6: Deadlocks

Processes



1940

1961

uniprogramming

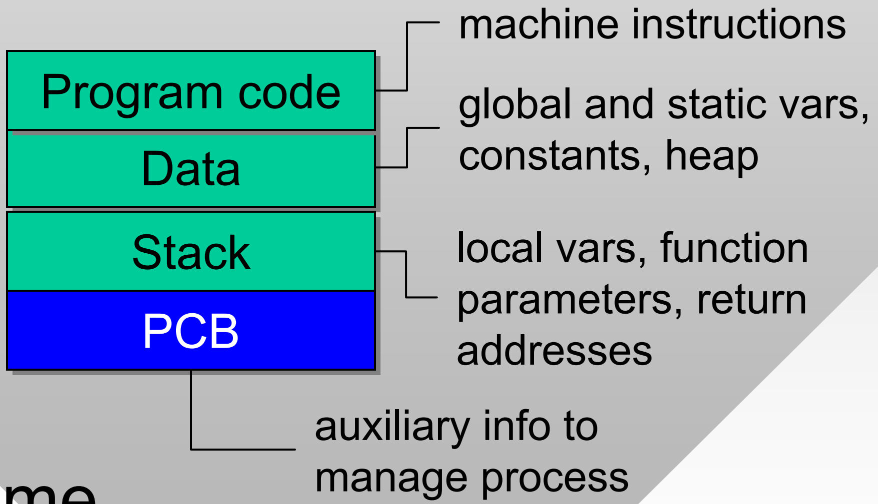
multi-programming

time-sharing

jobs

processes

- From the 1960s, jobs were described by a special data structure that allowed the OS to systematically monitor, control, and synchronize them
- This became known as a process, which consists of:
  - Program in execution
  - Data
  - Stack
  - Process Control Block (PCB)



Program code

Data

Stack

PCB

machine instructions

global and static vars, constants, heap

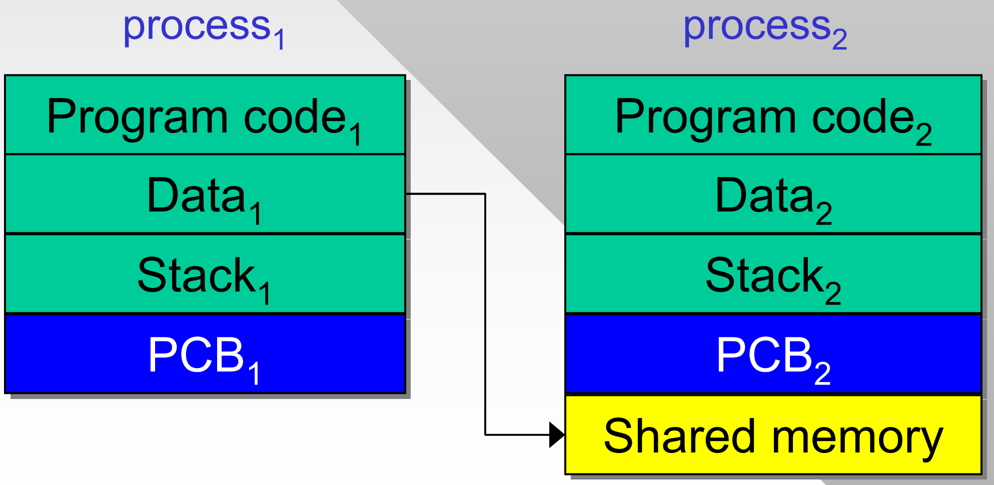
local vars, function parameters, return addresses

auxiliary info to manage process

- Note that programs stored on disk do not become processes until they are started

Processes

- Processes with shared memory
  - If shared memory is created by a process, it can be accessed in other processes in the system
  - This is called *memory mapping*
  - Just like named pipes, shared memory in Windows is addressable using some unique name



Chapter 3: Roadmap

3.1 What is a process?

➡ 3.2 Process states

3.3 Process description

3.4 Process control

3.5 Execution of the OS

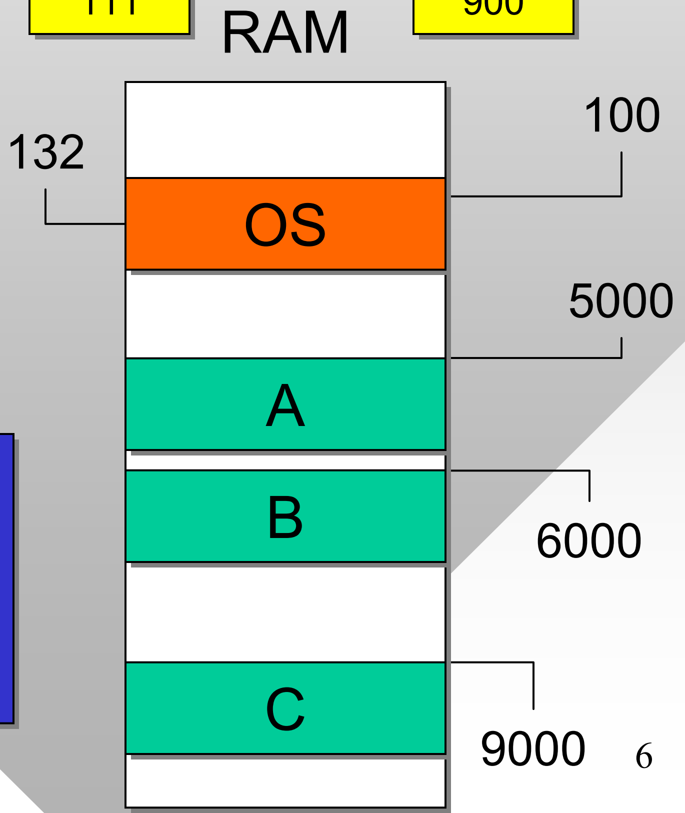
3.6 Security issues

3.7 Unix process management

Process States

- Process trace
  - Offsets (i.e., relative addresses) of instructions executed by a process
- CPU trace
  - Sequence of absolute addresses executed by the CPU
  - Suppose OS allows 6 CPU instructions in a slice, needs 3 to perform a process switch

A	B	C
0	0	0
1	1	900
2	2	901
503	3	902
504		903
505		904
506		900
507		901
108		902
109		903
110		904
111		900



CPU
5000
5001
5002
5503
5504
5505

100
101
102
6000
6001
6002

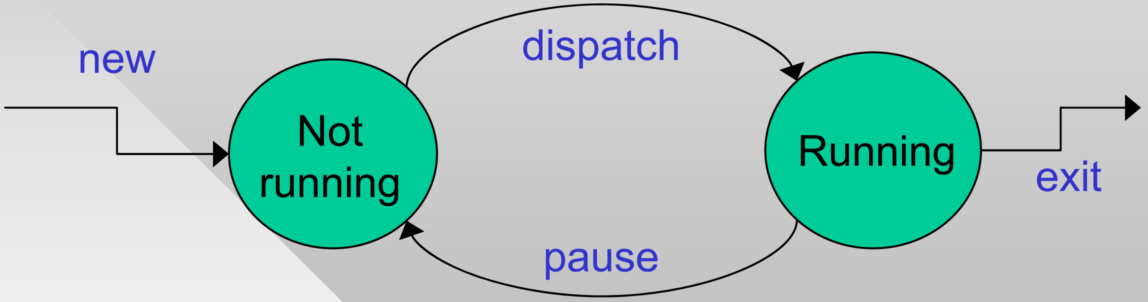
6003
100
101
102
9000
9900

9901
9902
9903
9904
100
101

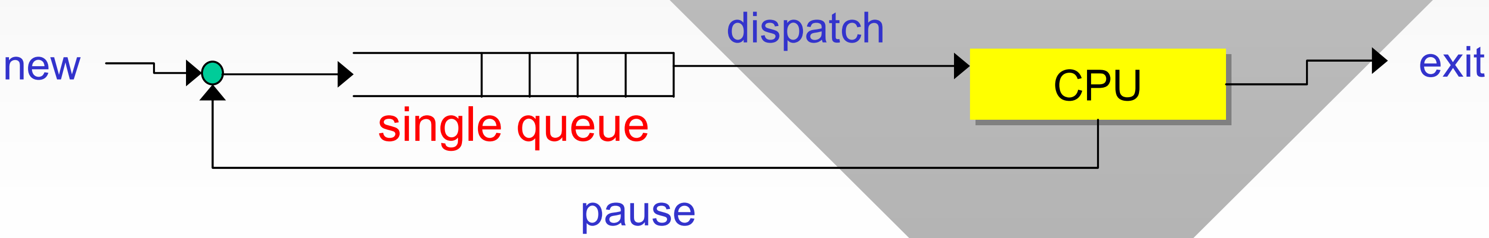
102
5506
5507
5108
5109
5110

Process States

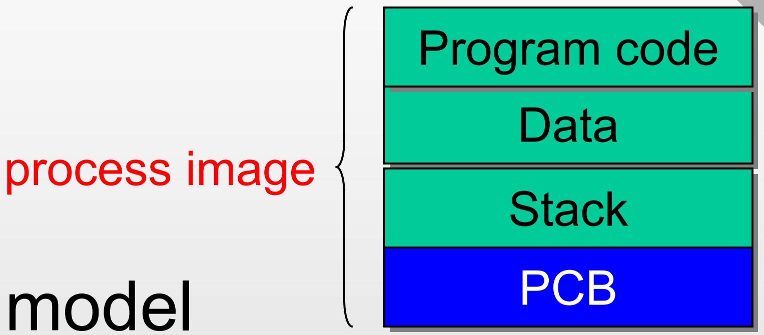
- This brings us to the issue of how the OS keeps track of processes and what runs next
- Simple *2-state model*:



- Implementation:



Process States

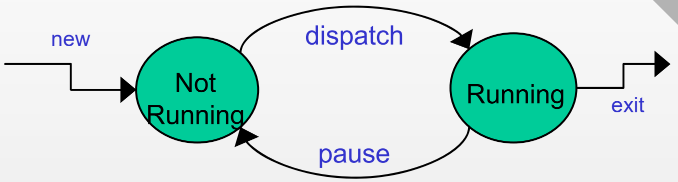


- Process creation in 2-state model
  - OS creates a PCB, loads necessary code and data in RAM, and moves process to the Not Running state
- Possible reasons for creation
  - Ready for next job in batch mode (old supercomputers)
  - User demand (command-line, login-related)
  - Needed by OS to serve a request
  - Explicitly spawned by a user program (e.g., CC.exe in hw #1)
- Original process is *parent*, spawned process *child*
  - Child may inherit access to certain open handles
  - Parent usually has full access rights to control the child (e.g., set its priority/affinity or terminate it)

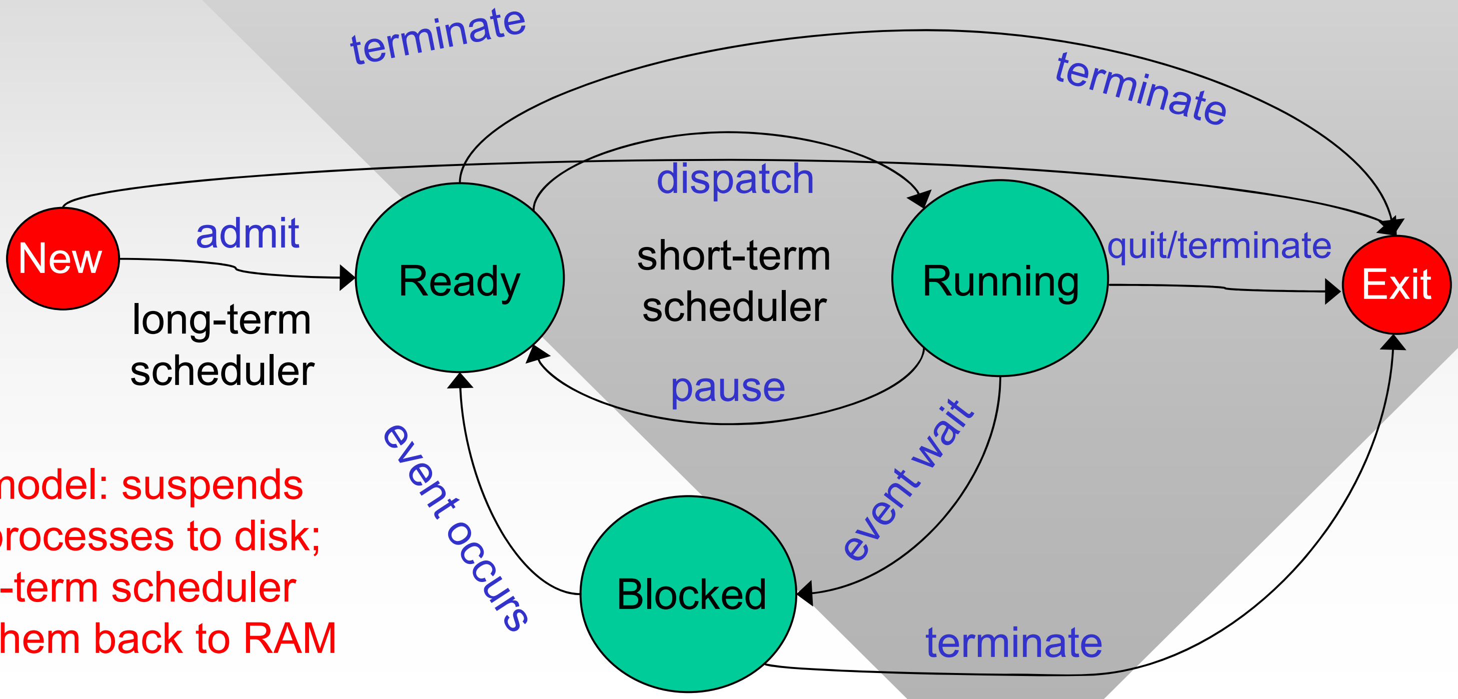
Process States

- Process termination
  - Normal completion
  - User request (e.g., Ctrl-C)
  - Request from another process
  - Access violation
  - Arithmetic error (division by zero)
  - Invalid instruction
  - Privileged instruction
  - Not enough RAM (bad_alloc exception)
- Stealthy crashes
  - Severe stack corruption may cause program to quit without any warning or error
- If code crashes in Release mode, will it crash in Debug?
  - Not necessarily
  - Some bugs can be seen only in release mode
  - Reasons?
- What about vice versa?

Process States



- Notice that 2-state model has no simple way of selecting the next ready process
  - Some might be blocked on I/O or events
- Next version, called *5-state model*, solves this:

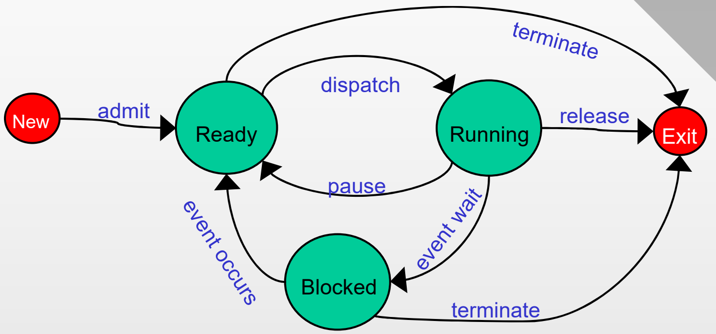


7-state model: suspends blocked processes to disk; medium-term scheduler activates them back to RAM

Process States

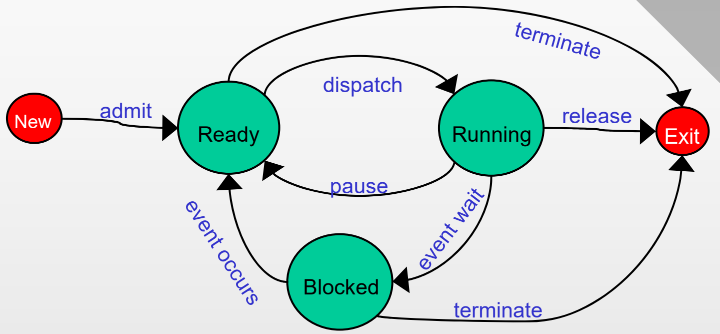
- Process creation in 5-state model
  - When the OS creates a PCB, it moves the process to New
  - However, data/code may still be on disk
- Given enough RAM, process is admitted to Ready
  - Code/data is loaded (fully or partially depending on whether virtual memory is available)
- Upon termination
  - Process memory is released, PCB is moved to the Exit state
  - May be beneficial to retain some PCB information (e.g., process exit code, PID, process handle)
  - Queries about a terminated process can be resolved using the PCB in the Exit state

Process States

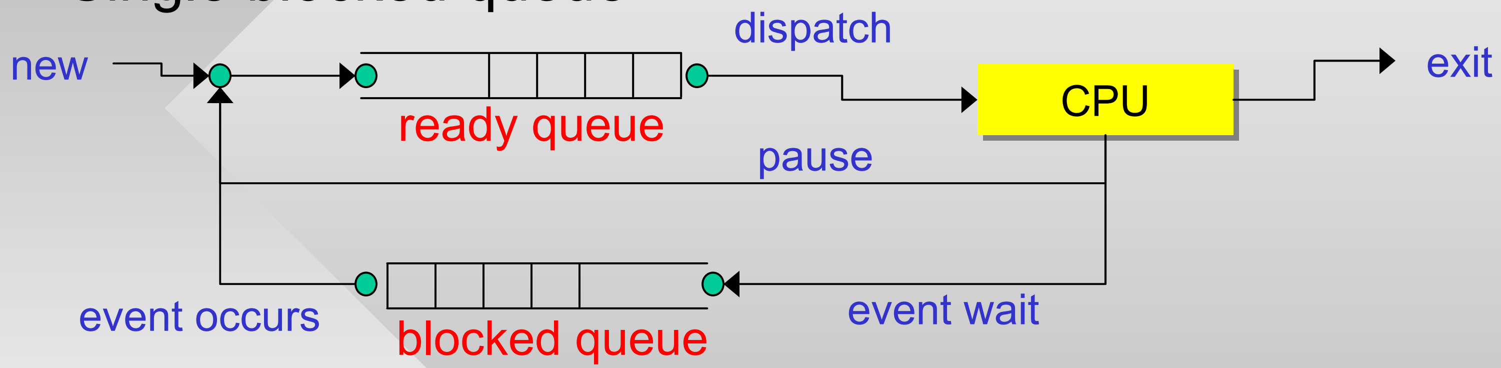


- Common transitions
  - Ready → Running: scheduler decides based on its policy (e.g., round-robin, strict priority, weighted round-robin)
  - Running→ Ready: either 1) time slice is over or 2) pre-empted by a higher-priority process in the Ready state
  - Running → Blocked: one of three options: process 1) voluntarily sleeping; 2) waiting for other processes (i.e., IPC); 3) waiting for I/O devices
  - Blocked → Ready: event signaled
  - Running → Exit: quits normally, crashes, or forced to quit
- Rarer cases
  - Ready → Exit, New→Exit, or Blocked → Exit: forced termination by user, OS, or another process

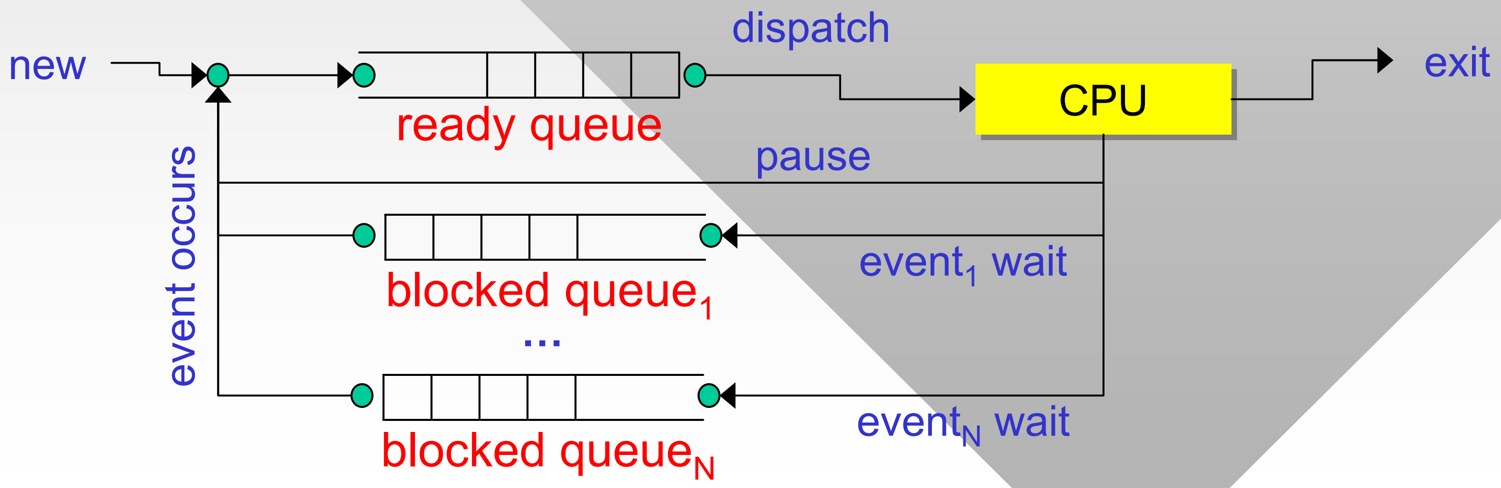
Process States



- Implementing 5-state model
  - Single blocked queue



  - Multiple blocked queues



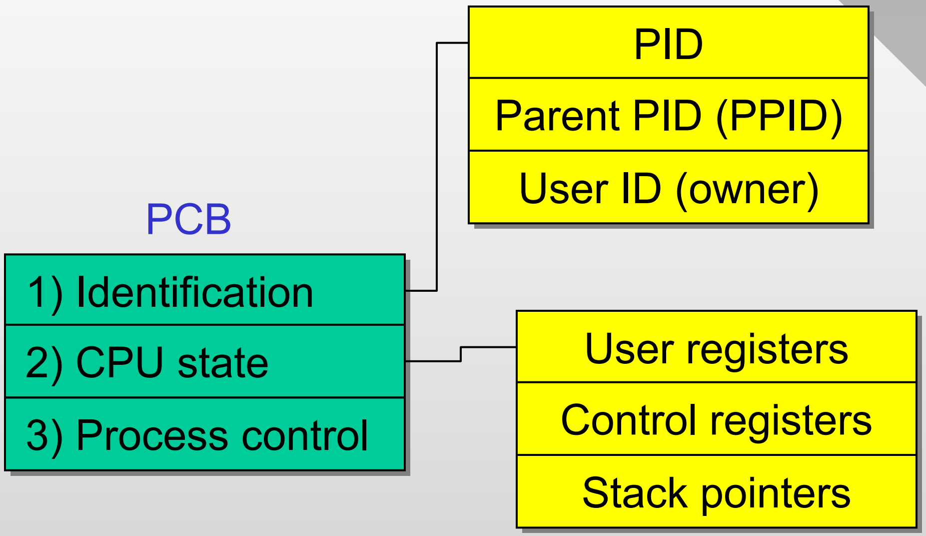
Implementation Notes

- By default, I/O requests are blocking
  - *Non-blocking* (*asynchronous*): APIs return control to the process regardless of whether data is ready or not
- How to know when async requests are finished
  - *Polling*: the process must periodically check on the status of the pending operation (Unix, Windows)
  - *Event-driven*: the API works with a special event handle that gets signaled when the operation is finished (Windows)
  - *Callback*: OS calls a specific function in the process upon event (GUI applications such as MFC)
  - Overlapped: asynchronous model that allows multiple requests to be pending to the same I/O handle
  - *I/O Completion Ports (IOCP)*: OS send notifications into a shared queue that the process can read (Windows)

Chapter 3: Roadmap

- 3.1 What is a process?
- 3.2 Process states
- ➡ 3.3 Process description
- 3.4 Process control
- 3.5 Execution of the OS
- 3.6 Security issues
- 3.7 Unix process management

Process Description


PCB

1) Identification

2) CPU state

3) Process control

PID

Parent PID (PPID)

User ID (owner)

User registers

Control registers

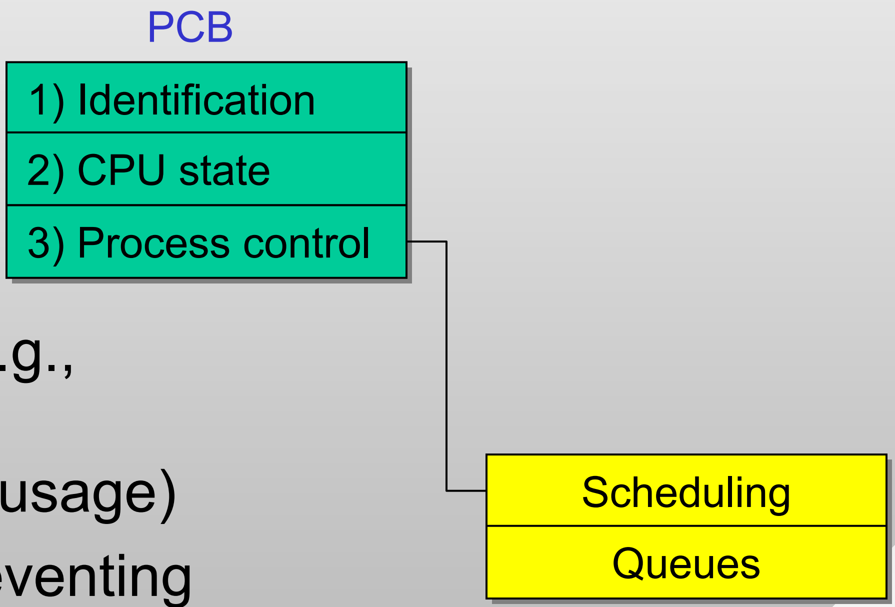
Stack pointers

- Process Control Block split into 3 general parts
- 1) Identification
  - Process ID (PID)
  - PPID sometimes needed to verify inherited rights
  - User/group IDs
- 2) CPU state is used during context (process) switches
  - User-modified registers (30-100 depending on the architecture)
  - Control registers (e.g., PC, flags)
  - Various stack pointers
- Context switch entails
  - Storing all CPU/FPU registers into PCB of running process
  - Deciding which process to run next
  - Loading registers from context of that process

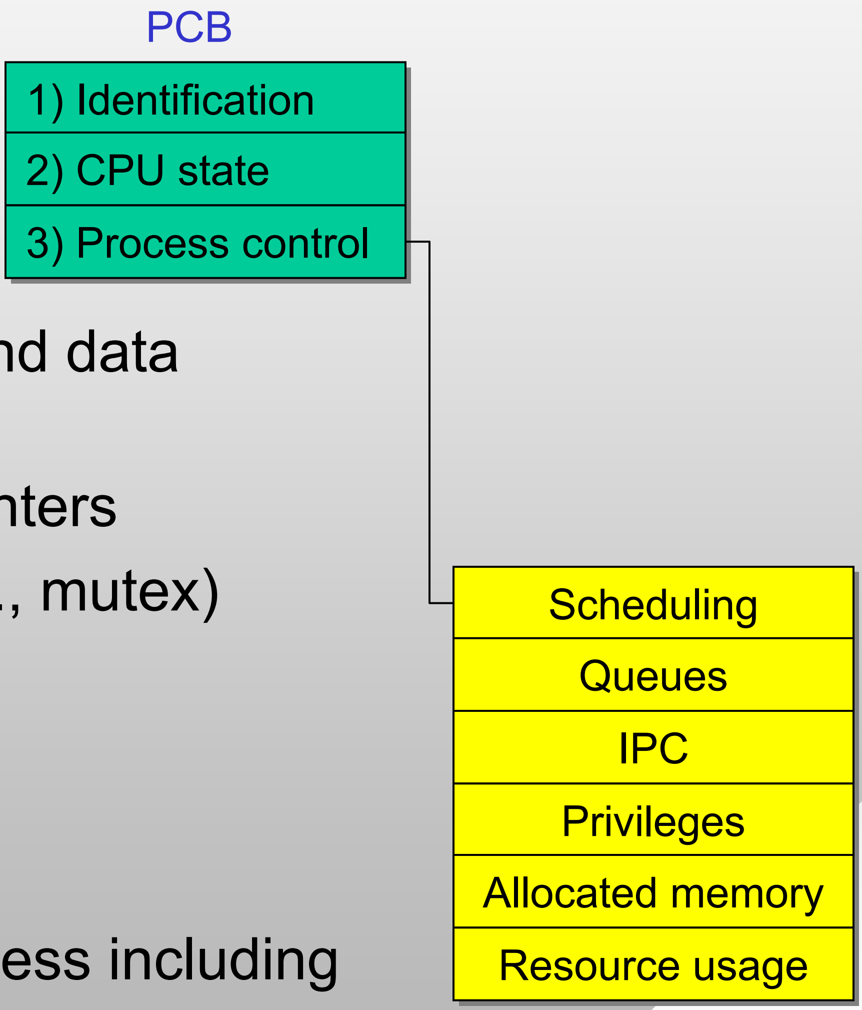
Process Description

3) Process control information

- Scheduling
  - Process state (e.g., ready, running, blocked)
  - Priority class
  - Info that helps scheduler (e.g., current wait time, estimated completion time, past CPU usage)
  - Events (if any) currently preventing the process from being ready
- Queues
  - Various wait queues the process is part of (e.g., scheduler, device I/O)



Process Description


PCB

1) Identification

2) CPU state

3) Process control

Scheduling

Queues

IPC

Privileges

Allocated memory

Resource usage

- Inter-process communication (IPC)
  - Message-passing handles and data (e.g., pipes, mailslots)
  - Shared memory handles/pointers
  - Synchronization objects (e.g., mutex)
- Privileges
  - Various system permissions
- Allocated memory
  - Virtual memory used by process including pages in pagefile
- Resource usage
  - Other open handles and various accounting

Chapter 3: Roadmap

3.1 What is a process?

3.2 Process states

3.3 Process description

3.4 Process control

3.5 Execution of the OS

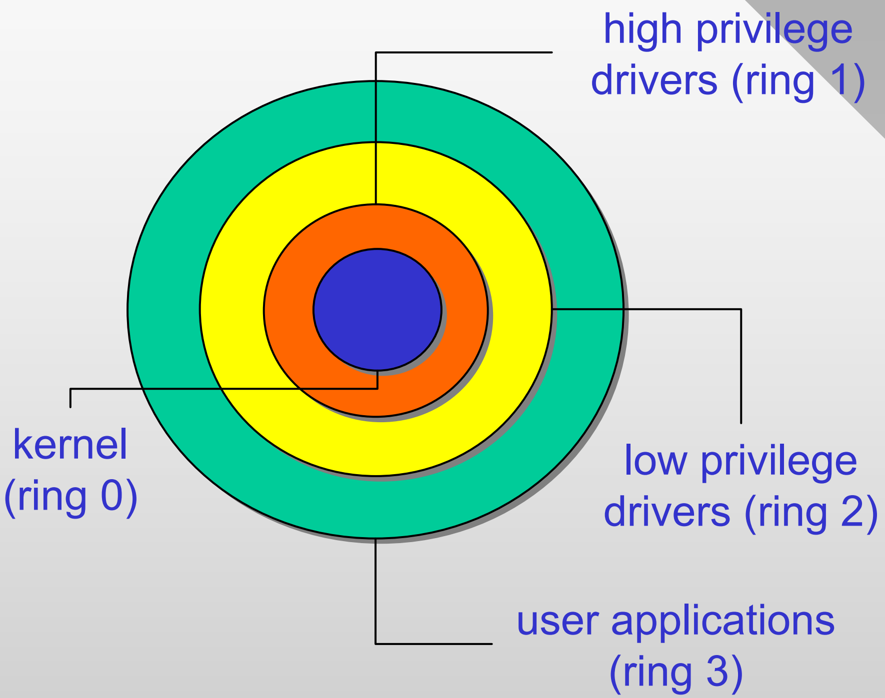
3.6 Security issues

3.7 Unix process management

Execution Modes

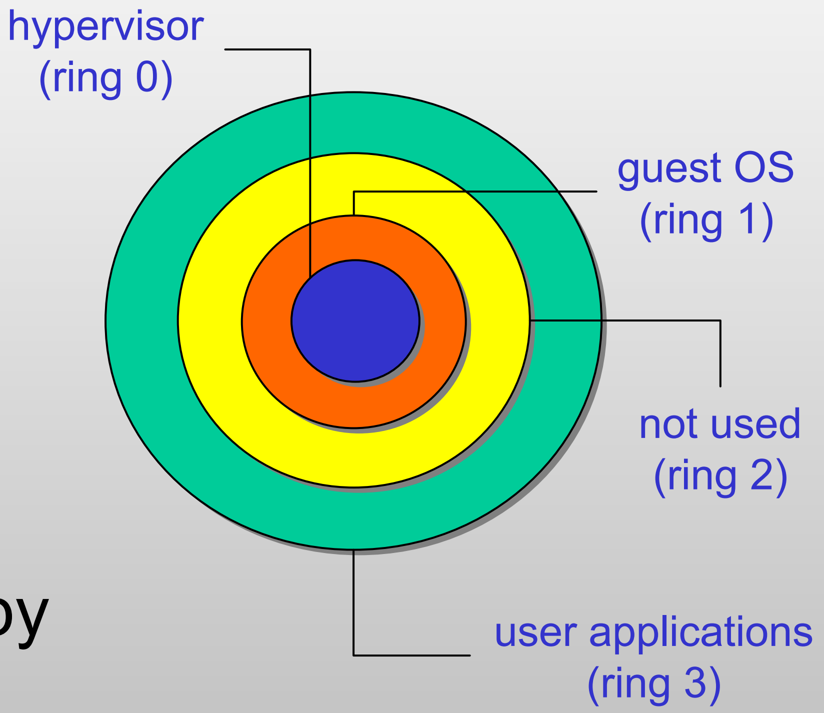
- CPU provides at least 2 execution modes
  - User mode prohibits all I/O instructions, virtual table manipulation, access to blocks of RAM not owned by process, and modification of certain registers
  - Kernel mode has no restrictions
- Some architectures allow more than 2 modes
  - These are often called protection rings
  - More granularity to allow “intermediate” privileges to certain processes (e.g., printer driver should be able to perform I/O, but not modify virtual-memory tables)
- Intel/AMD CPUs support 4 execution levels
  - Some older supercomputers had 8

Execution Modes



- Consider a hypothetical 4-ring system:
  - Ring 3 always user mode
  - Ring 0 always kernel
  - Rings 1 and 2 depend on the implementation
- Windows and Linux support only rings 0 and 3
  - Partly because other architectures these can run on (e.g., PowerPC and MIPS) traditionally had only 2 modes
  - Partly to reduce complexity
- Main drawback of 2-level systems
  - Any driver crash bluescreens the system and forces a reboot

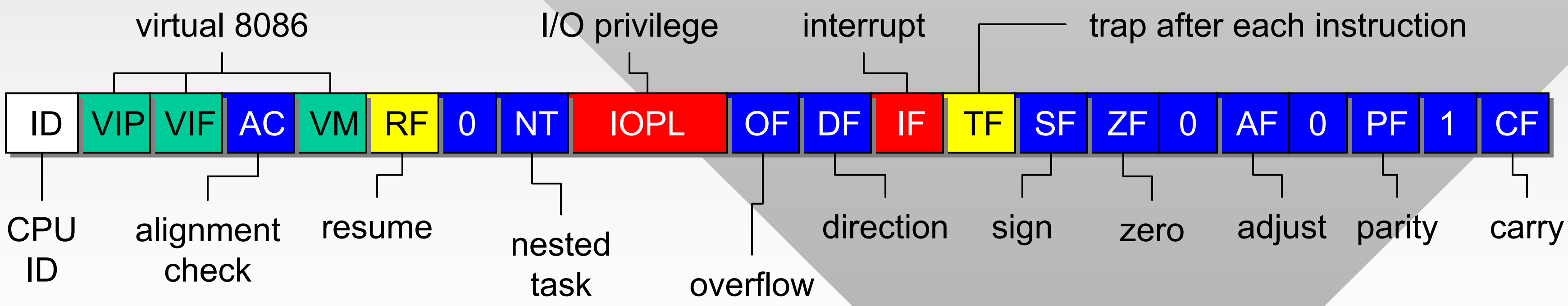
Execution Modes



- Microsoft virtualization server (Hyper-V) is an exception
  - Virtual machines (VM) allow multiple guest OSes to run transparently on the CPU
- Guest OSes are managed by the virtual machine monitor (VMM) called hypervisor
  - In contrast to normal kernels that are called supervisors
- Hypervisor runs in ring 0, guest OS in ring 1
  - AMD-V was supported starting with Athlon 64 (2006) and Intel VT-x starting with Pentium 4 (2005)

Mode Switch

- CPU support for changing execution mode
  - On some architectures special register called Program Status Word (PSW) tracks current mode
- On Intel, protection is scattered across many registers
  - CPL (current privilege level): 2 bits in CS (code segment) reg
  - DPL (data privilege level): 2 bits in virtual table of the segment
  - IOPL (I/O privilege level): 2 bits in EFLAGS register



virtual 8086

I/O privilege

interrupt

trap after each instruction

ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

CPU ID

alignment check

resume

nested task

overflow

direction

sign

zero

adjust

parity

carry

- I/O requires CPL $\leq$ IOPL; data access CPL $\leq$ DPL

Mode Switch

- Upon interrupt or kernel call (syscall)
  - CPL cleared to 0
  - Old values of registers are stored in stack (and later in PCB if a context switch occurs)
  - Execution passed to kernel address
  - Interrupt return (iret) causes old values to be restored
- Violations of current execution mode must be supported by the CPU
  - Throws a general protection fault if it detects attempts to circumvent kernel defenses (e.g., read/write or execute parts of memory with insufficient CPL, modify certain flags, execute I/O instructions, exceed allocated segment size)
  - OS intercepts these interrupts and terminates the process

Context (Process) Switch

- OS can switch processes whenever it gains control
- When does the OS actually execute?
- Three main instances:
  - External interrupt
  - CPU exception/fault/trap
  - System call
- Interrupts
  - Timer (e.g., slice over)
  - I/O (e.g., device ready)
- CPU Traps
  - Invalid instructions
  - Protection violations
  - Memory faults (e.g., virtual page not in RAM)
  - Arithmetic errors
- System calls
  - Kernel-level APIs invoked by user process
- Kernel may return control to current process, let it continue

Context (Process) Switch

- In fact, most non-timer interrupts do not switch processes
  - Short routines record interrupt conditions, reset the device, and return to user mode quickly
  - Later, other parts of the kernel (e.g., svchost.exe) perform full handling of the interrupt
  - Implemented via Deferred Procedure Calls (DPC) in Windows
- Process switch typically occurs only when either:
  - Time slice expires or process blocks on API
- Note that process switch requires mode switch, but not vice versa!
  - Q: Which of the two is more expensive?
- A: Process switch
  - Transition to kernel mode, selection of task to run, saving/restoring registers

Chapter 3: Roadmap

3.1 What is a process?

3.2 Process states

3.3 Process description

3.4 Process control

➡ 3.5 Execution of the OS

3.6 Security issues

3.7 Unix process management

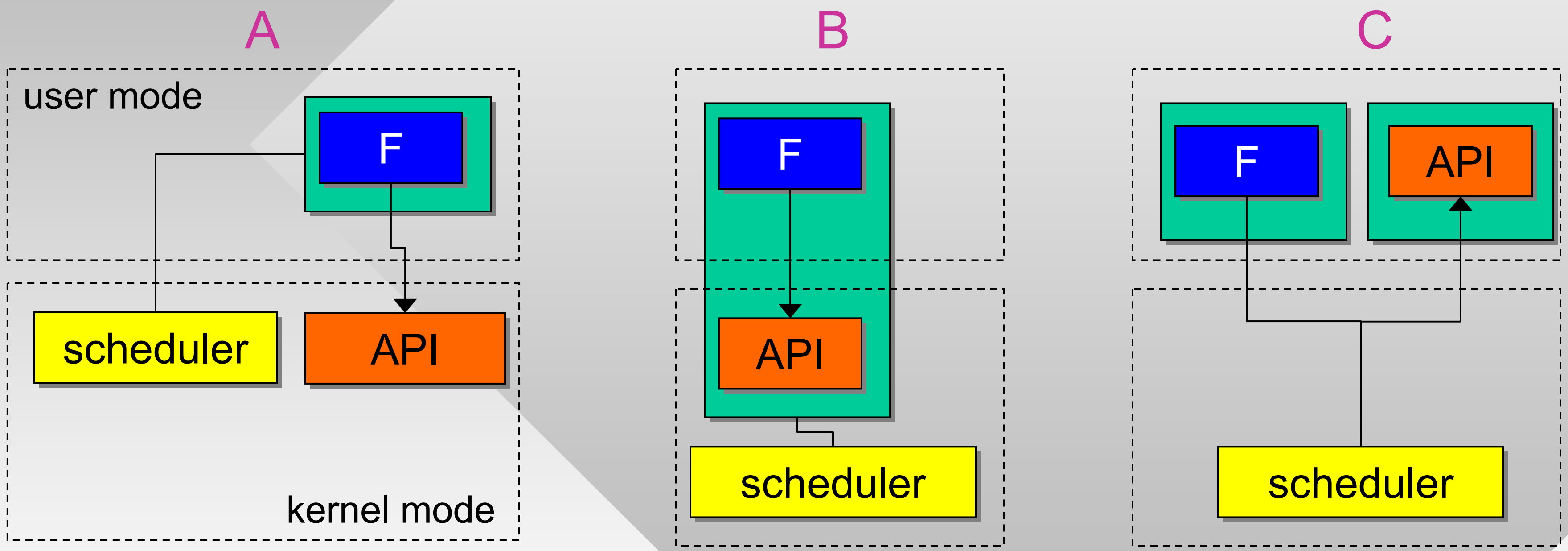
Execution of the OS

🟩 process

App user function

API OS API

- Three ways to execute calls to OS



API executes in kernel outside any process

2 user-OS context switches and 2 mode switches

old monolithic Unix

API executes in kernel mode as part of process

2 mode switches

Windows/Linux

API executes as separate user process

2 process context switches and 4 mode switches

micro-kernels

Execution of the OS

- Method A
  - Scheduler cannot interrupt the API when its running
  - 2 extra context switches per call compared to method B
- Method C (micro-kernels)
  - High switching overhead, but allows rapid user-mode API development
  - Better security as untrusted components (e.g., drivers) run in user mode
  - Certain high-security (e.g., military) applications
- Method B
  - Fastest switch to APIs, but less secure and more complex to develop
  - APIs must be *re-entrant*
  - Kernel attaches its own stack to each process image

process image

- Data
- Program code
- User stack
- PCB
- Kernel stack
- Shared memory