System Calls Types

- System calls can be grouped roughly into six major categories:
  - Process control
  - File manipulation
  - Device manipulation
  - information maintenance
  - Communications
  - Protection

1

1

System Calls Types

- 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

2

2

System Calls Types

- 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

System Calls Types

- 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

System Calls Types

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

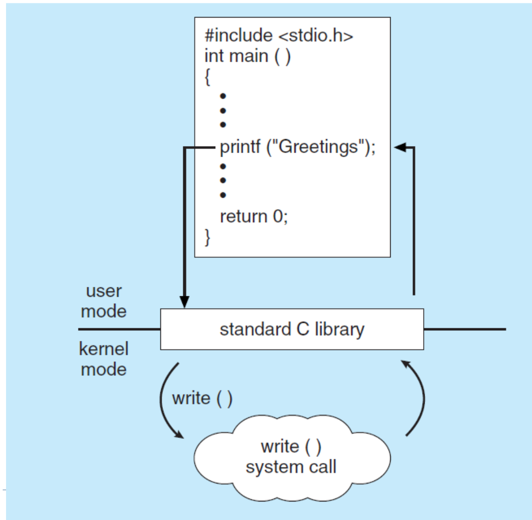
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

The following illustrates various equivalent system calls for Windows and UNIX operating systems.

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

System Calls Types

- C program invoking printf() library call, which calls write() system call



7

System Calls: How Many ?

- Linux exports about 350 system calls
- Windows exports about 400 system calls for core APIs, and another 800 for GUI methods

8

System Calls

- System calls are the "public" OS APIs
- Kernel leverages interrupts to restrict applications to specific functions

Interrupts ?

- An irregular control-flow from one context of execution and back
  - Usually, user to kernel and back, can also happen within kernel
- External interrupt: caused by a hardware device, e.g., timer ticks, network card interrupts
- Trap: Explicitly caused by the current execution, e.g., a system call
- Exception: Implicitly caused by the current execution, e.g., a division-by-zero fault

Why Use Interrupts

- For protection
- Forces applications to call well-defined "public" functions
  - Rather than calling arbitrary internal kernel functions
- Example:
- public foo() {
- if (!permission_ok()) return –EPERM;
- return _foo(); // no permission check
- }
- Calling _foo() directly would circumvent permission check

Interrupts : Handling

- Save current execution context
- Transfer control to a well-defined location in the kernel code
  - Switching privilege levels as needed
- Handle the interrupt
- Return to the previous context after handling the interrupt
- Should restore the saved state

Computer I/O Operation

- I/O devices and the CPU can execute concurrently
- Each device controller is in charge of a particular device type
- Each device controller has a local buffer
- CPU moves data from/to main memory to/from local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an interrupt

IO Operation Structure

- After I/O starts, control returns to user program only upon I/O completion
- Wait instruction idles the CPU until the next interrupt
- Wait loop (contention for memory access)
- At most one I/O request is outstanding at a time, no simultaneous I/O processing
- After I/O starts, control returns to user program without waiting for I/O completion
  - System call – request to the OS to allow user to wait for I/O completion
  - Device-status table contains entry for each I/O device indicating its type, address, and state
- OS indexes into I/O device table to determine device status and to modify table entry to include interrupt

Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- Interrupt architecture must save the address of the interrupted instruction
- A trap or exception is a software-generated interrupt caused either by an error or a user request
- An operating system is interrupt driven

15

Interrupts : Handling

- The state of a program's execution is concisely and completely represented by CPU register state
- Pause the program: dump the registers in memory
- Service the Interrupt
- Resume the program: copy the registers back into CPU

16

Interrupts: Handling

- Control jumps to the kernel
  - At a prescribed address (the interrupt handler)
- The register state of the program is dumped on the kernel's stack
  - Sometimes, extra info is loaded into CPU registers
  - E.g., page faults store the address that caused the fault in the cr2 register
- Kernel code runs and handles the interrupt
- When handler completes, resume the program

Interrupts: Handling

- Kernel creates an array of Interrupt descriptors in memory, called Interrupt Descriptor Table, or IDT
  - Can be anywhere in memory
  - Pointed to by special register (idtr)
    - segment registers and gdtr and ldtr
- Entry 0 configures interrupt 0, and so on
- Most interrupt handling hardware state set during boot
- Each interrupt has an IDT entry specifying:
  - What code to execute, privilege level to raise the interrupt

Interrupts

- External interrupts are asynchronous interrupts
  - will happen every time an instruction executes (with a given program state)
    - Divide by zero
    - System call
    - Bad pointer dereference
- Traps and exceptions are synchronous interrupts
  - Caused by an external event
    - Usually device I/O
    - Timer ticks (well, clocks can be considered a device)

Interrupts

Intel nomenclature

- Interrupt – only refers to asynchronous interrupts
- Exception – synchronous control transfer

- Each interrupt or exception includes a number indicating its type
  - E.g., 14 is a page fault, 3 is a debug breakpoint
  - The number is the index into an Interrupt Descriptor Table

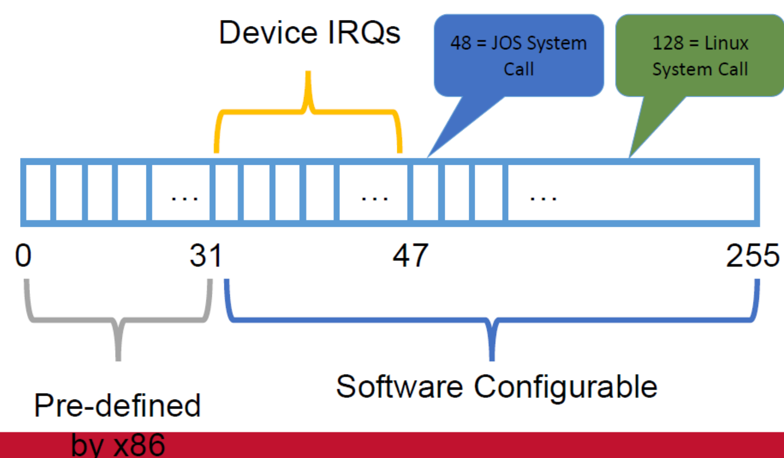
Interrupts: Handling

- How interrupts work in hardware
- How interrupt handlers work in software
- How system calls work

- Respond to some event, return control to the appropriate process
- What to do on:
  - Network packet arrives
  - Disk read completion
  - Divide by zero
  - System call

Interrupts: Handling

- Hx86 Interrupts



Interrupts: Handling

Hx86 Interrupt Overview

- Each interrupt is assigned an index from 0-255
- 0-31 are for processor interrupts; generally fixed by
- Intel
  - E.g., 14 is always for page faults
- 32-255 are software configured
- 32-47 are for device interrupts (IRQs) in JOS
- Most device's IRQ line can be configured
- 128 (0x80) and 48 (0x30) issue system calls in Linux and JOS respectively

Interrupts: Handling

Software Interrupts

- The int <num> instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention. JOS uses 0x30
- OS sets ring level required to raise an interrupt
  - Generally, user programs can't issue an int 14 (page fault manually)
  - An unauthorized int instruction causes a General Protection (#GP) fault
    - Interrupt 13

OS Design and Implementation

- Design and Implementation of OS IS not easily "solvable", but some approaches have proven successful
- Internal structure of different Operating Systems can vary widely
- Start the design by defining goals and specifications
- Affected by choice of hardware, type of system

OS Design and Implementation

- User goals
  - operating system should be convenient to use, easy to learn, reliable, safe, and fast
- System goals
  - operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

OS Design and Implementation

Important principle to separate

- Policy
  - What will be done?
- Mechanism
  - How to do it?
- The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)
- Specifying and designing an OS is highly creative task of software engineering

27

27

OS Implementation

- Much variation
  - Early OSes in assembly language
  - Now C, C++
- Actually usually a mix of languages
  - Lowest levels in assembly
  - Main body in C
  - Systems programs in C, C++, scripting languages like PERL, Python, shell scripts
- More high-level language easier to port to other hardware
  - But slower

28

28

OS Structure

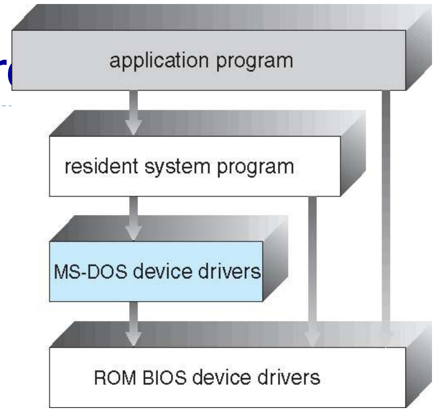
- OS must be engineered carefully if it is to function properly and be modified easily
- A common approach is to partition the task into small components, or modules
- Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions

Simple Structure

- Many operating systems do not have well-defined structures
- Such systems started as small, simple, and limited systems and then grew beyond their original scope like MS-DOS
  - Written to provide the most functionality in the least space

Simple Structure

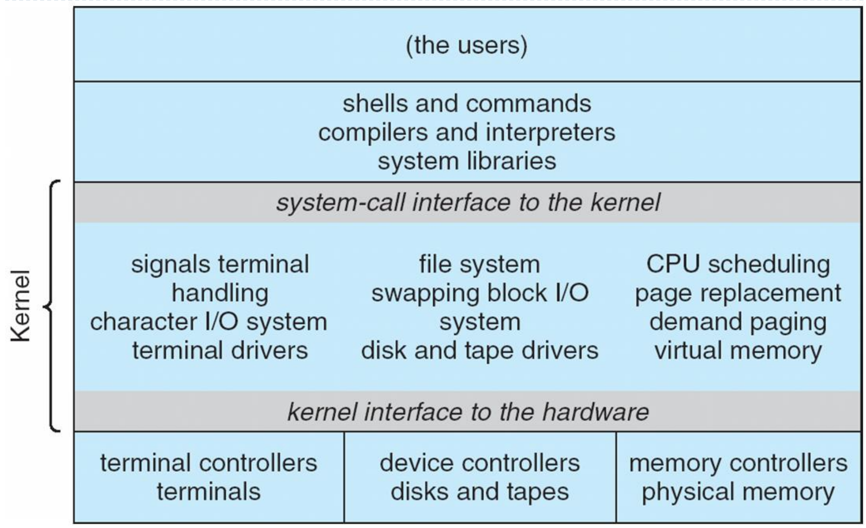
- Not divided into modules
- Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



Limited Structure

- Original UNIX OS, initially was limited by hardware functionality
- Consists of two separable parts:
  - Kernel
    - A series of interfaces
    - device drivers
  - System Programs

Limited Structure



33

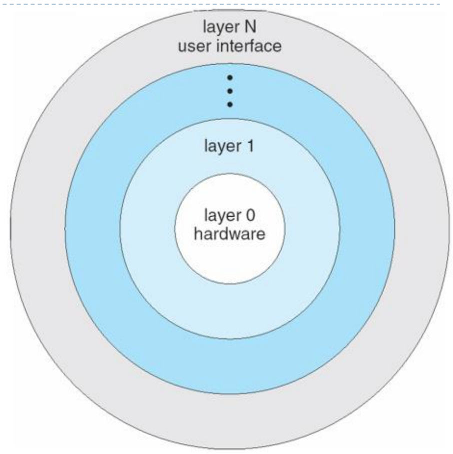
Layered Approach

- With proper hardware support, OS can be broken into pieces that are smaller and more appropriate
- OS can then retain much greater control over the computer and over the applications that make use of that computer
- Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems
- Multiple ways to Modular Design → Layered

34

Layered Approach

- OS is divided into a number of layers (levels), each built on top of lower layers
- The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



35

35

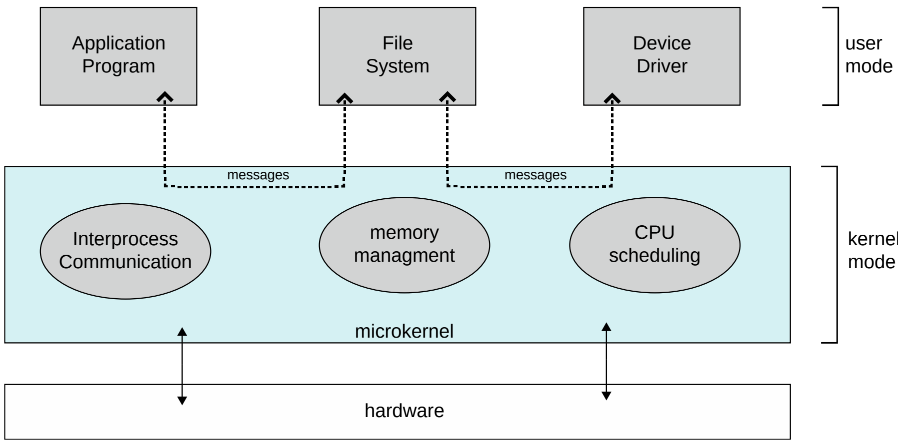
Microkernel System Structure

- Moves as much from the kernel into user space
  - Mach example of microkernel
  - Mac OS X kernel (Darwin) partly based on Mach
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

36

36

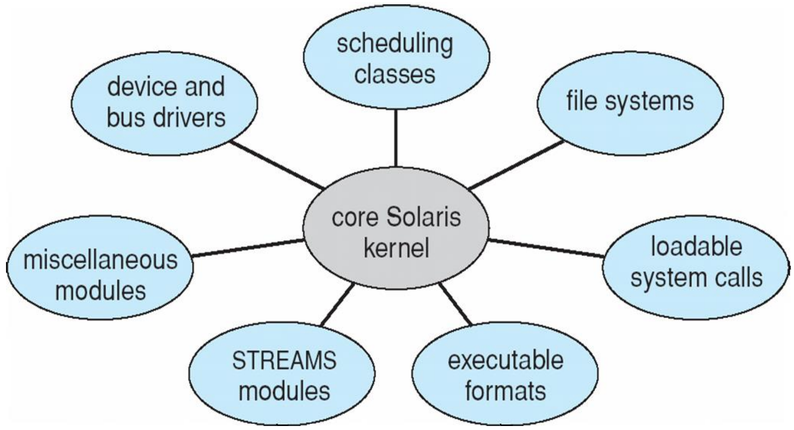
Microkernel System Structure



OS Design: Modular Approach

- Many modern operating systems implement loadable kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, etc

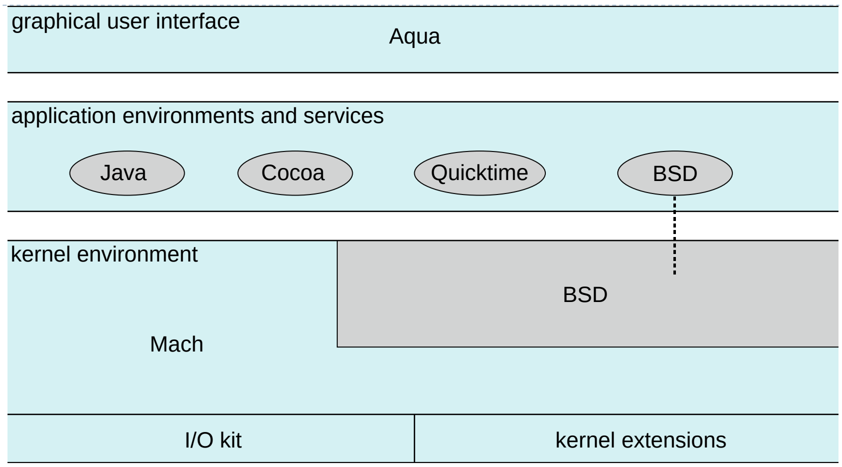
Solaris Modular Approach



Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - Linux and Solaris kernels in kernel address space, so monolithic, plus modular for dynamic loading of functionality
  - Windows mostly monolithic, plus microkernel for different subsystem personalities
- Apple Mac OS X hybrid, layered, Aqua UI plus Cocoa programming environment
  - Below it is kernel consisting of Mach microkernel and BSD Unix parts, plus I/O kit and dynamically loadable modules (called kernel extensions)

Mac OS X Structure



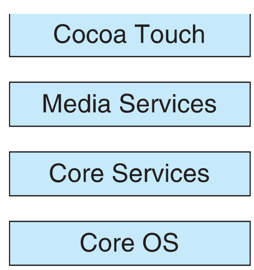
41

41

iOS

Apple mobile OS for iPhone, iPad

- Structured on Mac OS X, added functionality
- Does not run OS X applications natively
  - Also runs on different CPU architecture (ARM vs. Intel)
- Cocoa Touch Objective-C API for developing apps
- Media services layer for graphics, audio, video
- Core services provides cloud computing, databases
- Core OS, based on Mac OS X kernel



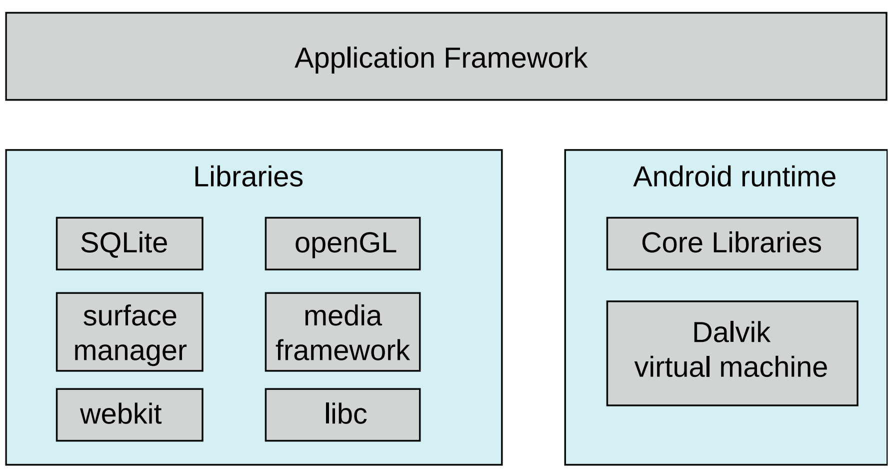
42

42

Android

- Developed by Open Handset Alliance (mostly Google)
  - Open Source
- Similar stack to IOS
- Based on Linux kernel but modified
  - Provides process, memory, device-driver management
  - Adds power management
- Runtime environment includes core set of libraries and Dalvik virtual machine
  - Apps developed in Java plus Android API
    - Java class files compiled to Java bytecode then translated to executable than runs in Dalvik VM
- Libraries include frameworks for web browser (webkit), database (SQLite), multimedia, smaller libc

Android Architecture



Process

- Process: A program in action
- Code, data, and stack
  - Usually (but not always) has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- Only one process can be running in the CPU at any given time!

45

45

Process Creation

Processes can be created in two ways

1. System initialization: one or more processes created when the OS starts up
2. Execution of a process creation system call: something explicitly asks for a new process

System calls can come from

- User request to create a new process (system call executed from user shell)
- Already running processes
  - User programs
  - System daemons

46

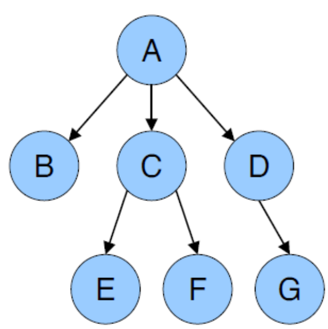
46

Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

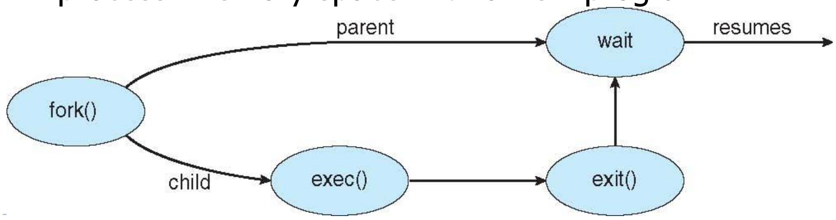
Process Creation

- Processes can create other processes
- Process tree tracks these relationships
  - A is the root of the tree
  - A created three child processes: B, C, and D
  - C created two child processes: E and F
  - D created one child process: G



Process Creation

- 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 a fork() to replace the process' memory space with a new program



Process Termination

Termination of processes can be

- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

Process Termination

- Process executes last statement and then asks the operating system to delete it using the exit() system call
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call
- Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - Cascading termination. All children, grandchildren, etc. are terminated
  - Termination is initiated by the operating system

Process Termination

- The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process

  pid = wait(&status);
- If no parent waiting (did not invoke wait()) process is a zombie
- If parent terminated without invoking wait, process is an orphan

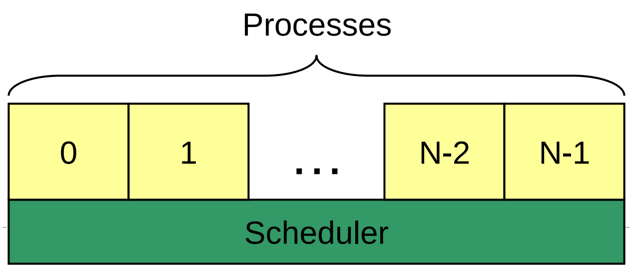
Process Hierarchies

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
- UNIX calls this a "process group"
  - If a process exits, its children are "inherited" by the exiting process's parent
- Windows has no concept of process hierarchy
  - All processes are created equal

OS Processes

Two "layers" for processes

- Lowest layer of processes in a structured OS, handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the process table
  - Each process has a process table entry



CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
- Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

CPU Scheduler

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- Dispatch latency: time it takes for the dispatcher to stop one process and start another running

57

Process/Processor Scheduling

- In multiprogramming/timesharing operating systems CPU is multiplexed among processes to improve CPU utilization

- Several CPU-scheduling algorithms

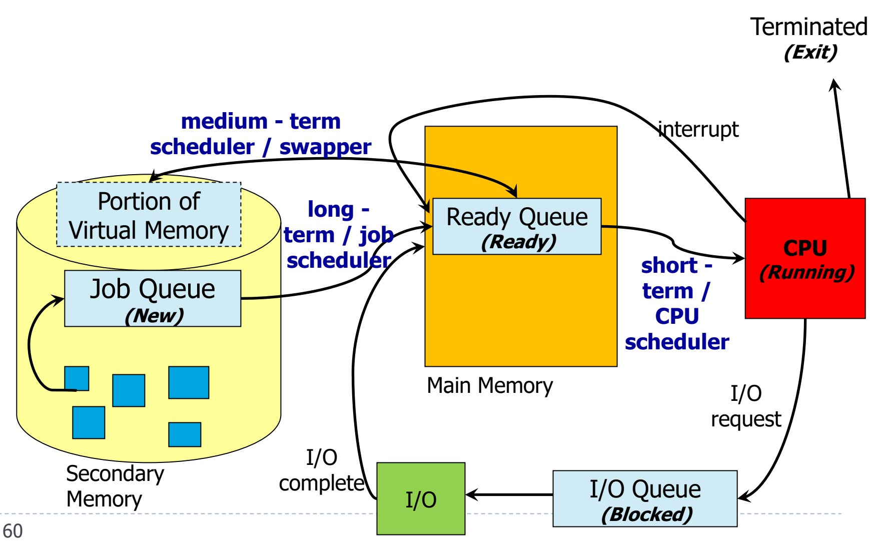
58

Process Scheduling Levels

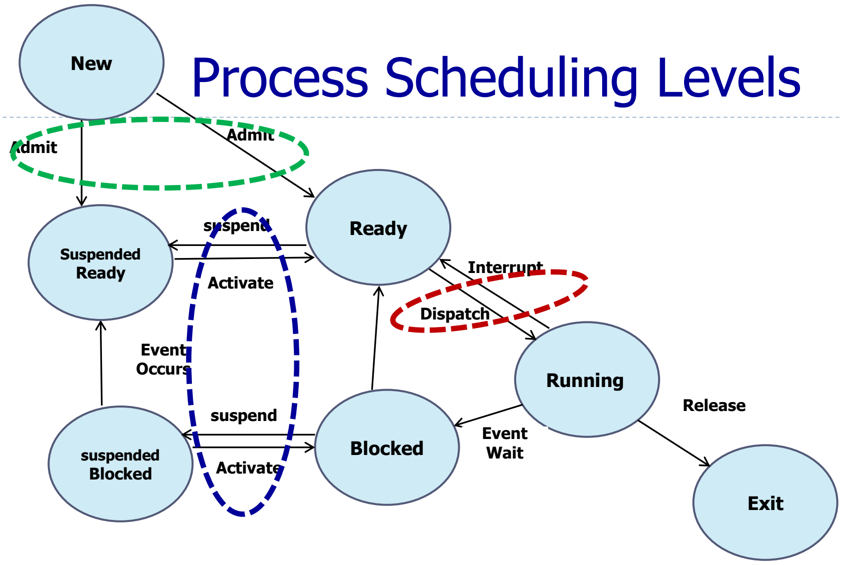
- Three levels of scheduling on frequency of execution

- Long-term scheduling
  - Performed when a new process is created
  - Executes relatively infrequently
- Medium-term scheduling
  - Performed when a process is swapped-in/out
  - Executes somewhat more frequently
- Short-term scheduling (Dispatcher/Scheduler)
  - Performed to decide which ready process to execute next
  - Executes most frequently

Scheduling Levels

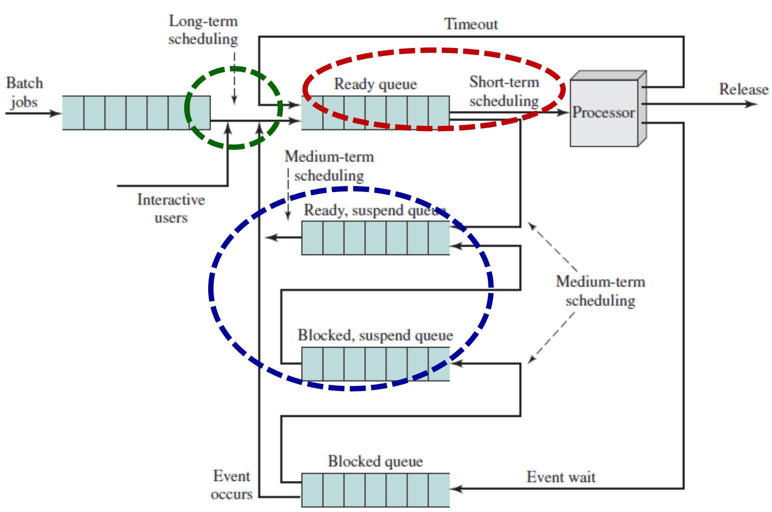


Process Scheduling Levels



61

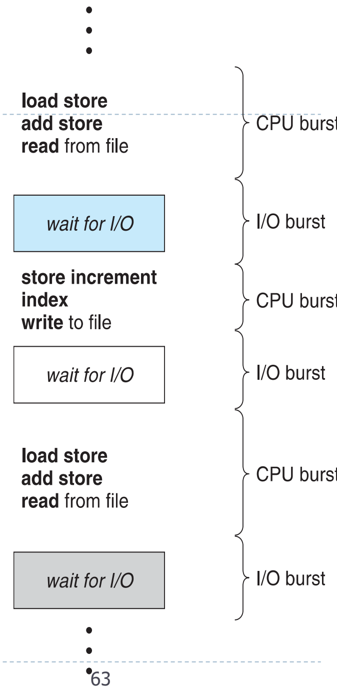
Process Scheduling Levels



62

CPU Scheduling

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle
- Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



Process/Processor Scheduling

- Process: a program in action
- Process execution consists of a cycle of CPU execution (CPU burst) and I/O (burst) wait
- Compute Bound

- IO Bound

Scheduling Criteria

- CPU utilization
  - keep the CPU as busy as possible
- Throughput
  - # of processes that complete their execution per time unit
- Turnaround time
  - amount of time to execute a particular process
- Waiting time
  - amount of time a process has been waiting in the ready queue
- Response time
  - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algo Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

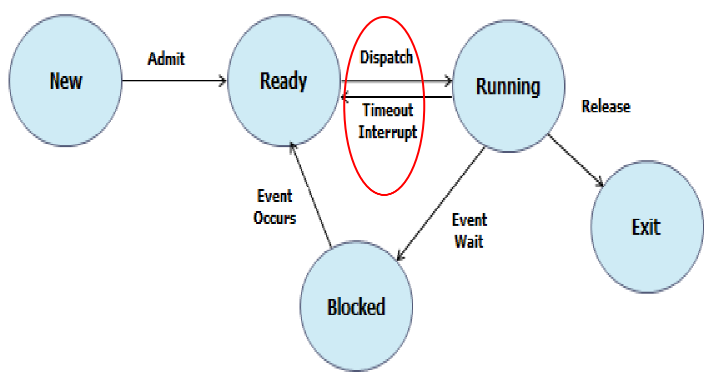
Process Scheduling Criteria

- CPU Utilization
- Throughput

- Turnaround Time
  - Completion Time - Arrival Time
- Waiting Time
  - Completion Time - Arrival Time - CPU Time
  - Turnaround Time - CPU Time
- Response Time

- Preemptive Vs Non-preemptive

Process Scheduling



- Preemptive Vs Non-preemptive

First-Come, First-Served Scheduling

- Simplest CPU-scheduling algorithm
- Process requesting CPU first is allocated CPU first
- Implementation is easily managed with a FIFO queue

Process	Burst Time	Arrival	Finish	Turn-Around	Wait
P1	23	1	24	23	0
P2	3	1	27	26	23
P3	4	1	31	30	26

- Average Waiting time = (0 + 23 + 36)/3 =16.3 milliseconds

First-Come, First-Served Scheduling

- Processes enter queue P2(3), P3(4), P1(23)

Process	CPU Burst	Arrival	Finish	Turn-Around	Wait
P2	3	1	4	3	0
P3	4	1	8	7	3
P1	23	1	31	30	6

- Average Waiting time = (0 + 3 + 6)/3 = 3 milliseconds

First-Come, First-Served Scheduling

- Average waiting time may vary substantially if the processes' CPU burst times vary greatly
- Non-Preemptive Scheduling
- Convoy effect
  - All the other processes wait for the one big process to get off the CPU
  - Lower CPU and device utilization

71

71

Shortest-Job-First Scheduling

- Process's next CPU burst (if available)
  - Assigned CPU to the process with smallest next CPU burst
  - Next CPU bursts of 2 processes same → FCFS scheduling
- Gives minimum average waiting time for a given set of processes
- Knowing the length of the next CPU request ?

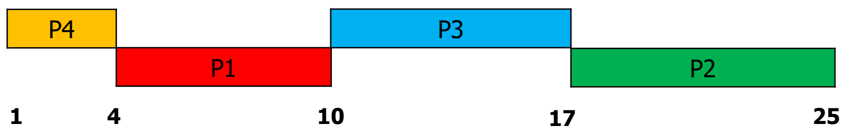
72

72

Shortest-Job-First Scheduling

- Non-Preemptive
- All processes enter at time 1

Process	Burst Time
P1	6
P2	8
P3	7
P4	3



SRTF Scheduling

- Shortest-Remaining-Time-First (SRTF) scheduling is Preemptive version of SJF algorithm
- When a new process arrives at the ready queue while a previous process is still executing
  - If the next CPU burst of the newly arrived process is shorter than what is left of the currently executing process, it will preempt the currently executing process

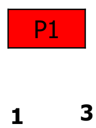
SRTF Scheduling

Ready	Running
P2(3)	P1(4)

➡

Ready	Running
P1(4)	P2(3)

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2



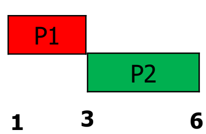
SRTF Scheduling

Ready	Running
P1(4)	P2(3)

➡

Ready	Running
-	P1(4)

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2



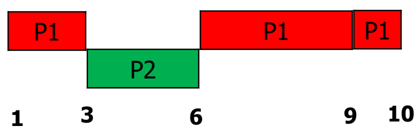
SRTF Scheduling

Ready	Running
-	P1(4)

➡

Ready	Running
P3(5)	P1(1)

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2



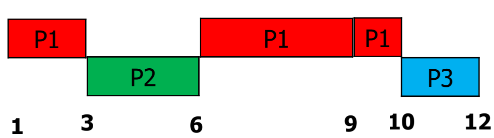
SRTF Scheduling

Ready	Running
-	P3(5)

➡

Ready	Running
P4(2)	P3(3)

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2



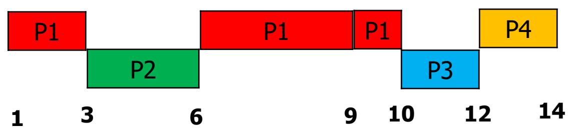
SRTF Scheduling

Ready	Running
P3(3)	P4(2)

➡

Ready	Running
-	P3(3)

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2



SRTF Scheduling

Ready	Running
-	P3(3)

➡

Ready	Running
-	-

Process	Arrival Time	Burst Time
P1	1	6
P2	3	3
P3	9	5
P4	12	2