Week-1: Recap

OS: A control program to manage computer resources

OS Types: Resident Monitor, Batch

Buffering Vs Spooling   Program Vs Process

Compute/IO bound Process

OS Types: Pure Multiprogramming and With Priorities — Static / Dynamic

Preemptive and Non-Preemptive Scheduling

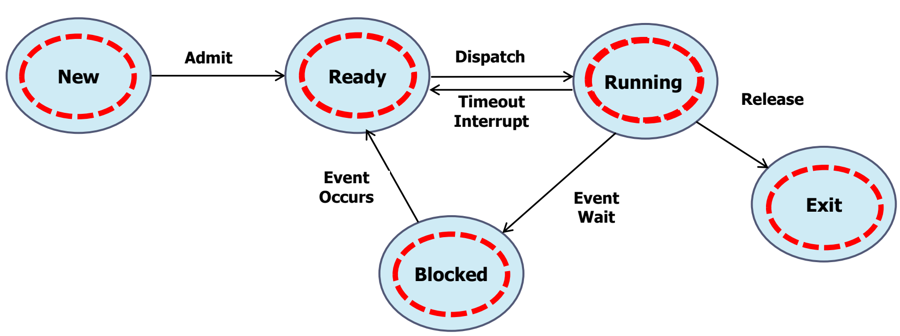
Time-Sharing OS, General Purpose OS

Process States: Creation, Ready, Running, Blocked, Termination

Process Model

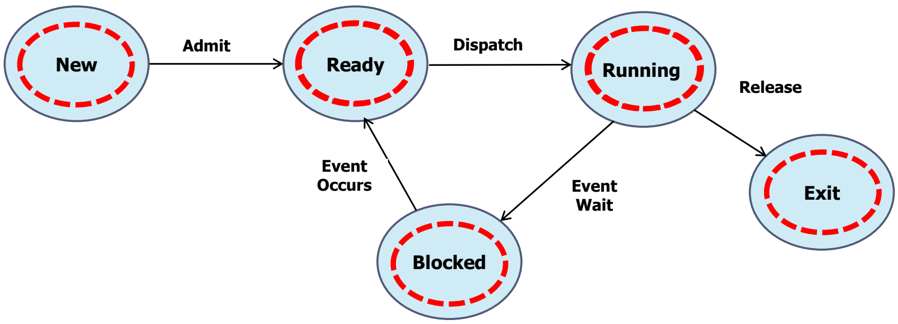
1

Process Model



2

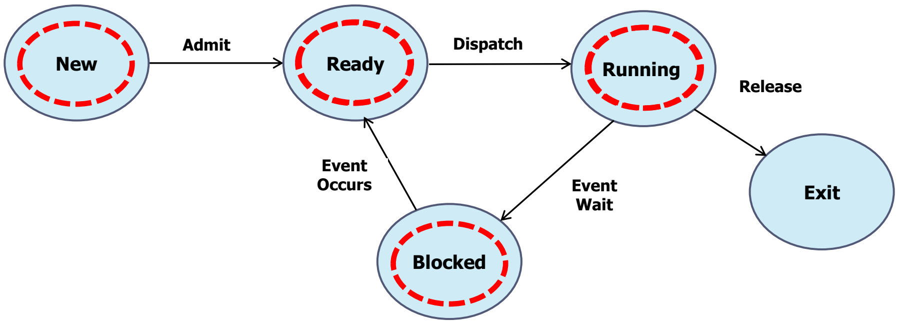
Batch OS



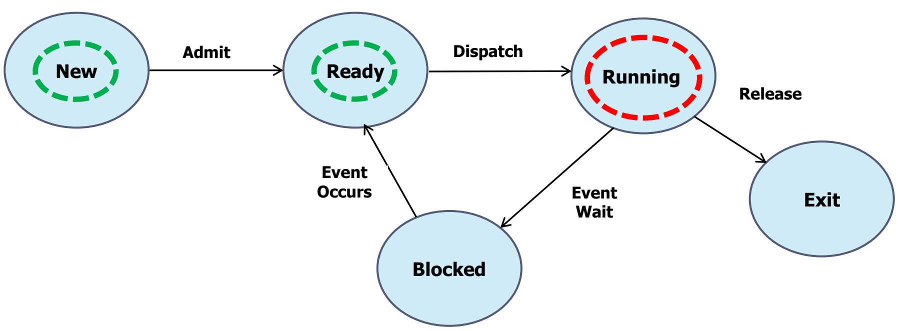
Pure Multiprogramming OS

- Non-Preemptive CPU Scheduling
  - Process in Running state voluntarily leaves
    - Running to Exit state
    - Running to Blocked state
  - Context Switch
    - Running to Exit/Blocked
    - Ready to Running

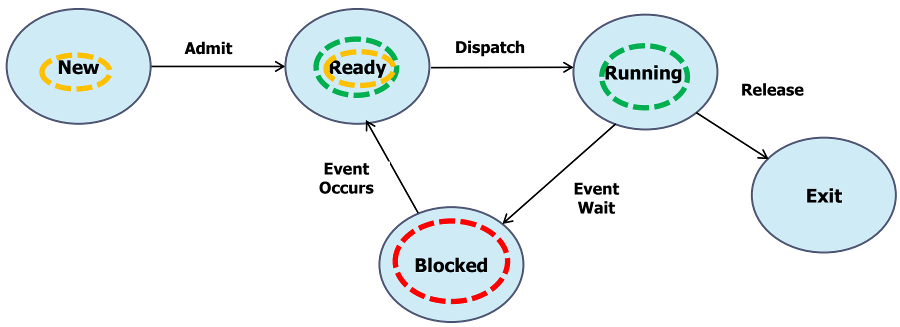
Pure Multiprogramming OS



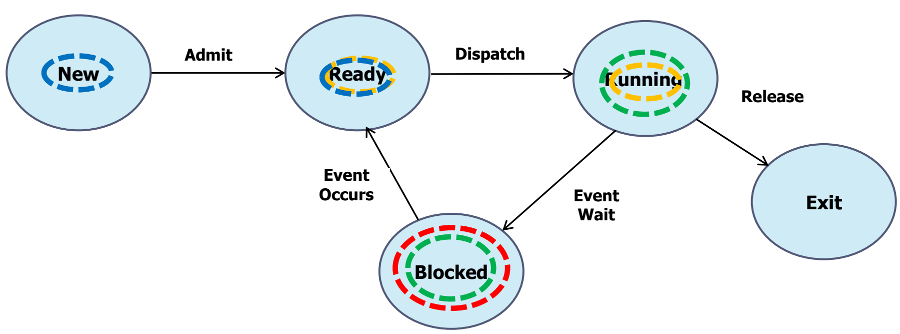
Pure Multiprogramming OS



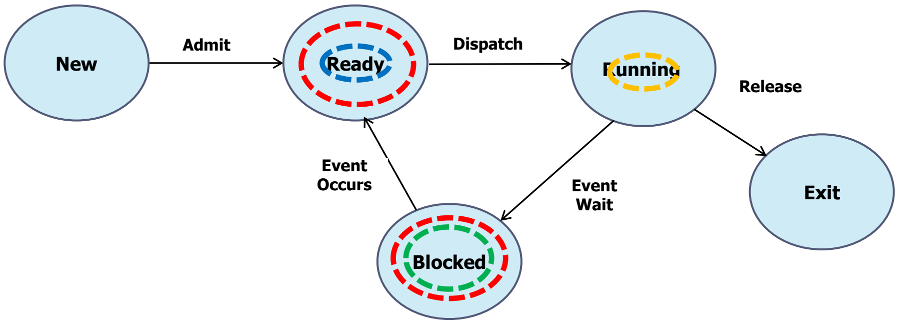
Pure Multiprogramming OS



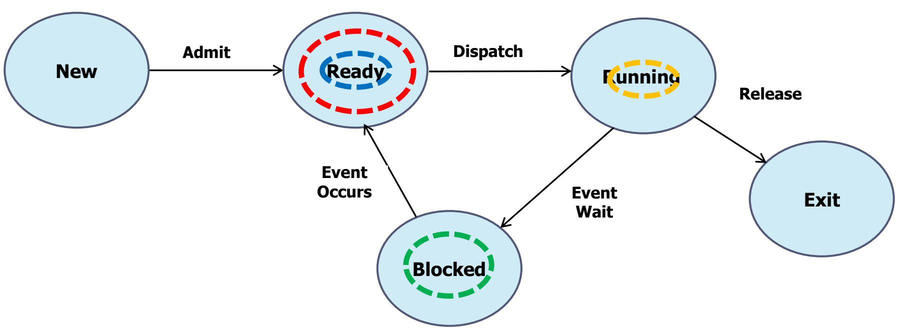
Pure Multiprogramming OS



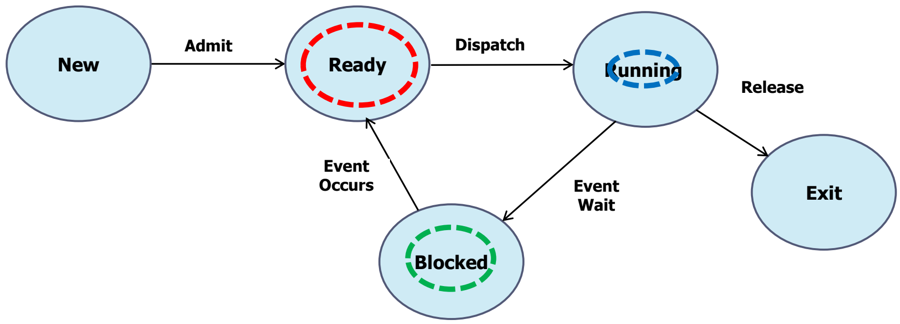
Pure Multiprogramming OS



Pure Multiprogramming OS

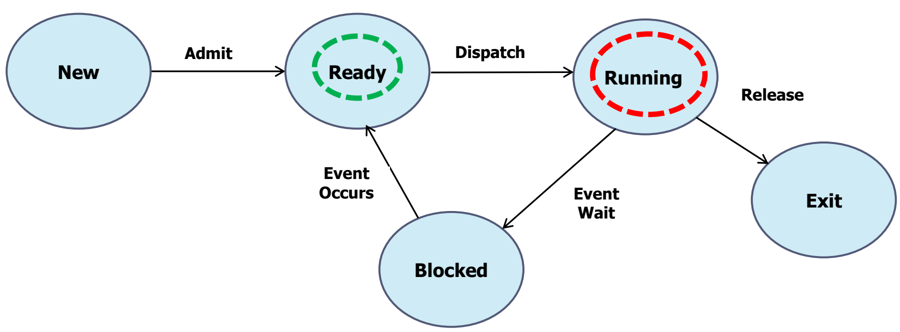


Pure Multiprogramming OS



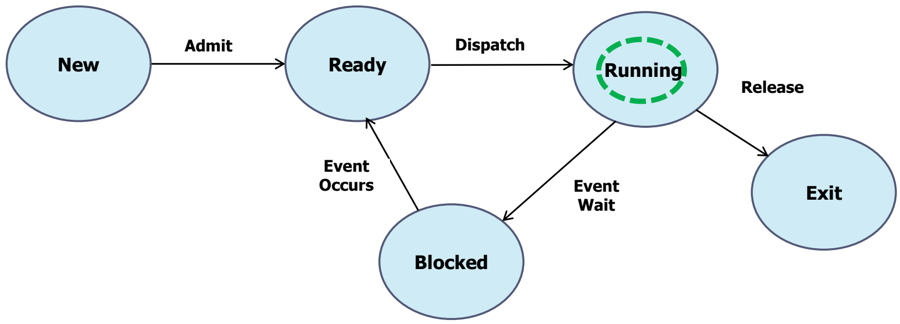
11

Pure Multiprogramming OS



12

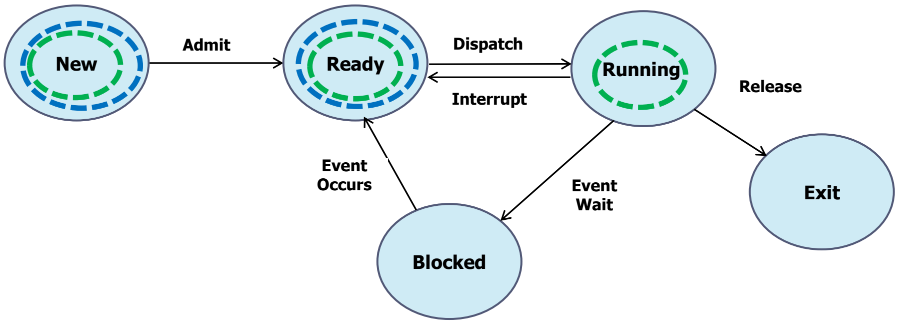
Pure Multiprogramming OS



Multiprogramming with Priorities

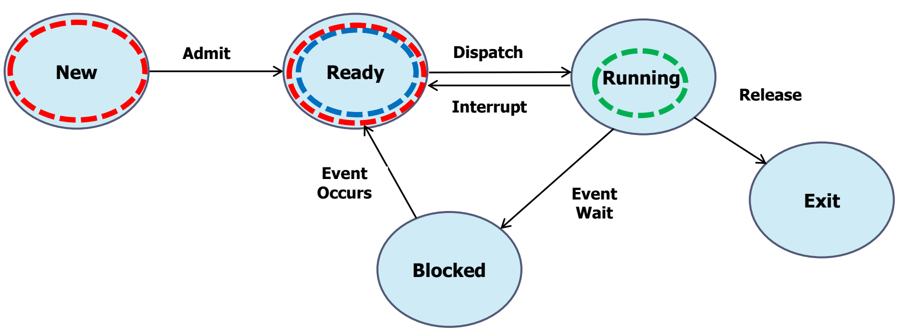
- Preemptive CPU Scheduling
  - Low priority process in Running state
  - High priority process enters Ready state
  - Context Switch
    - Running (LP) to Ready
    - Ready (HP) to Running

Multiprogramming with Priorities



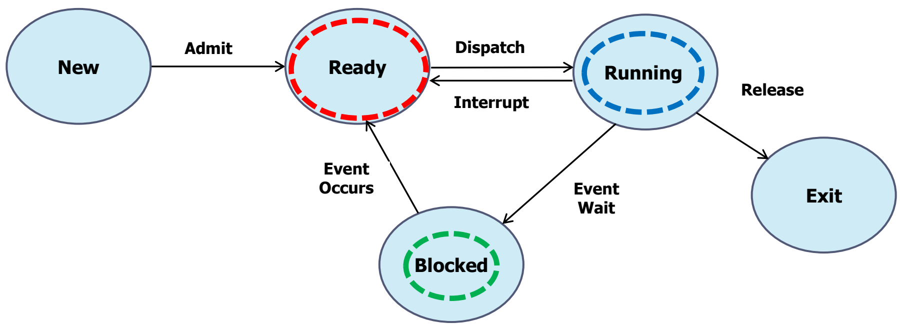
15

Multiprogramming with Priorities



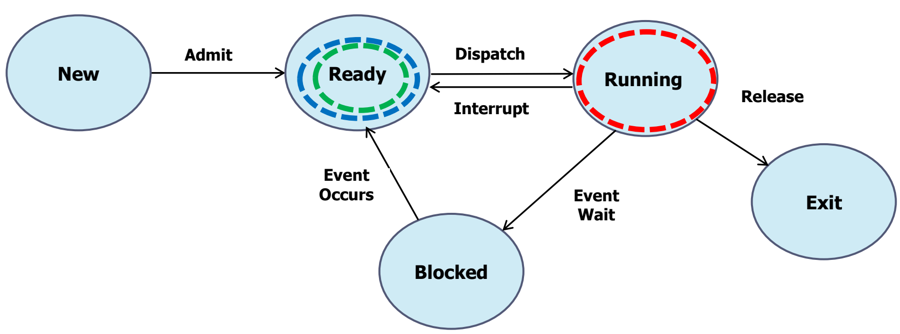
16

Multiprogramming with Priorities



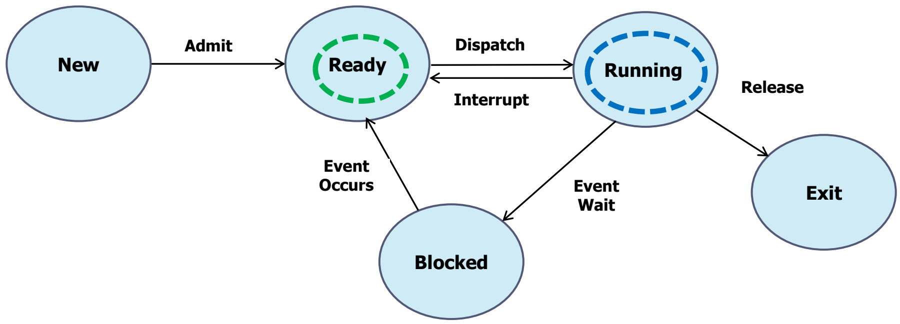
17

Multiprogramming with Priorities

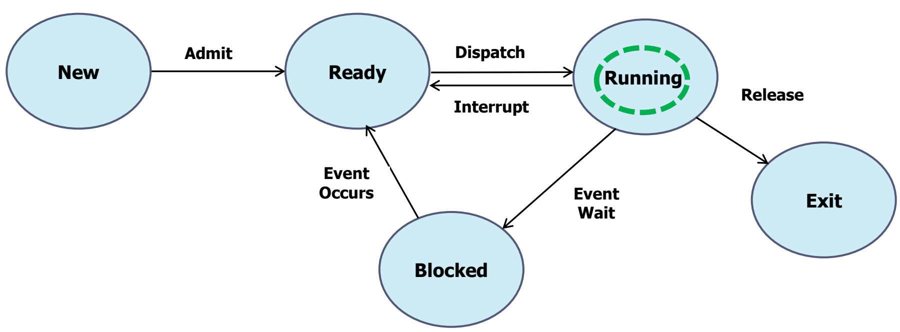


18

Multiprogramming with Priorities



Multiprogramming with Priorities



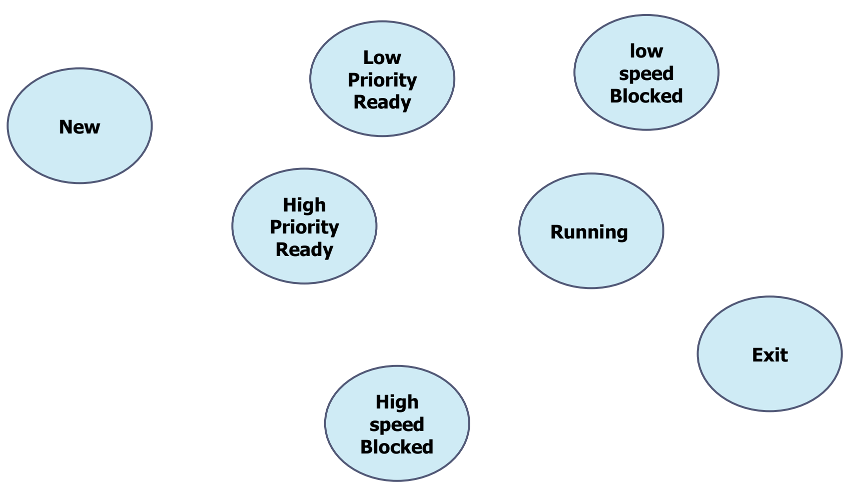
Multiprogramming with Priorities

- Multiprogramming with priorities OS, priorities are assigned statically (at process creation time)
- Two types of priorities (Low and High)
- IO devices with Two speeds (Low and High)
- Low/High priority processes use Low/High speed IO devices respectively
- Different lists for Ready and Blocked states of different priorities and different speed IO devices
- Process Model

21

21

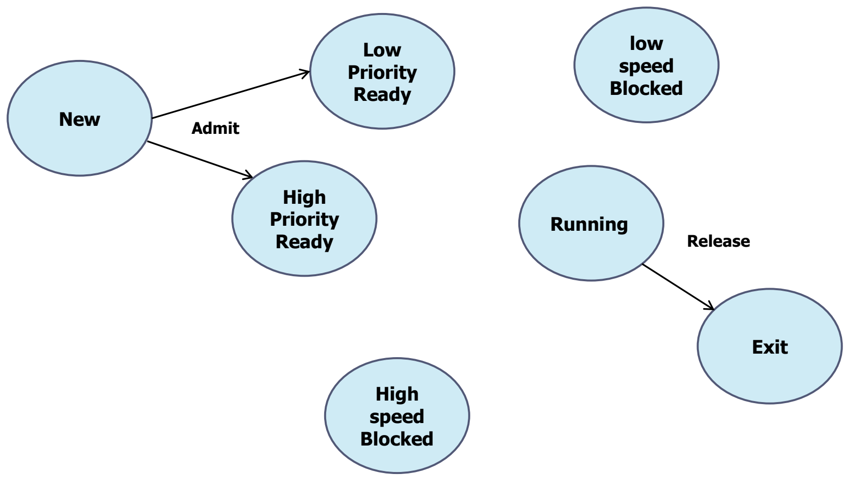
Multiprogramming with Priorities



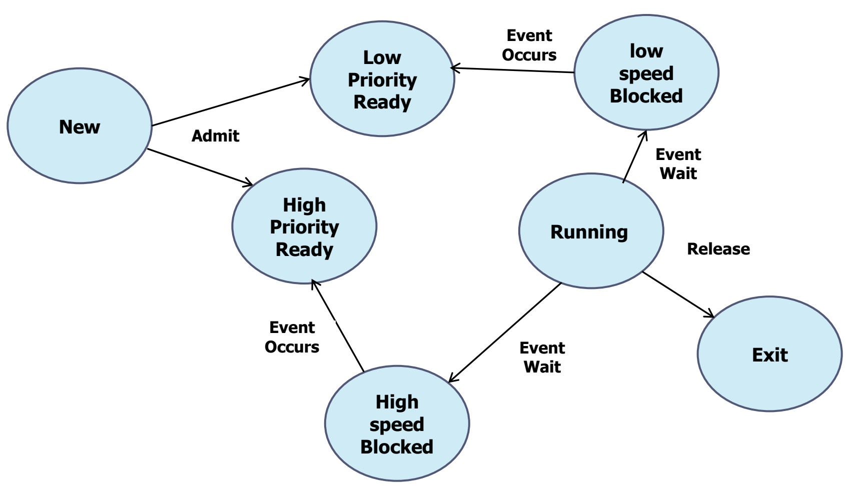
22

22

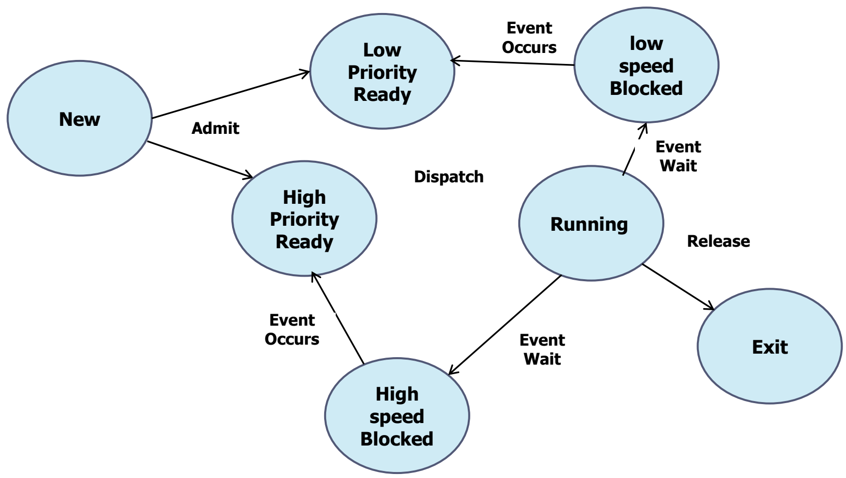
Multiprogramming with Priorities



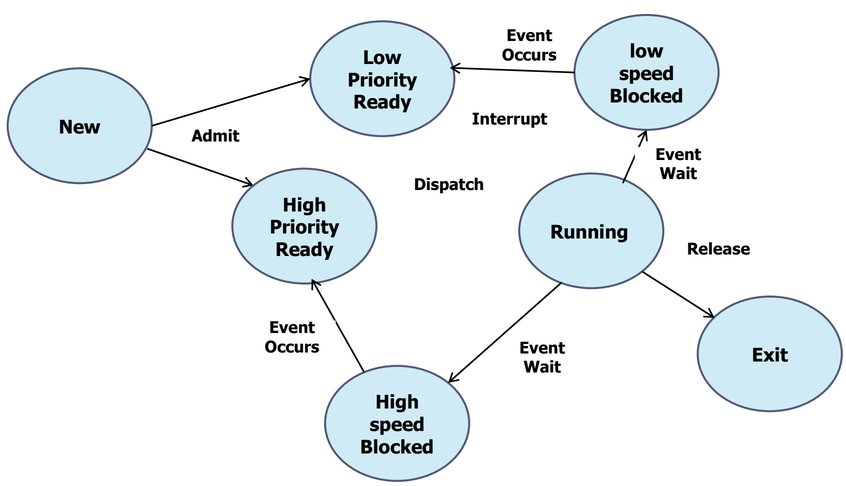
Multiprogramming with Priorities



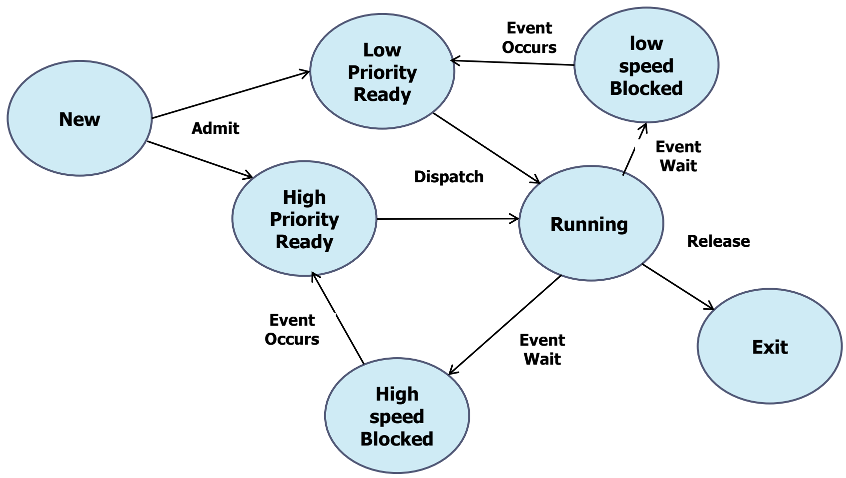
Multiprogramming with Priorities



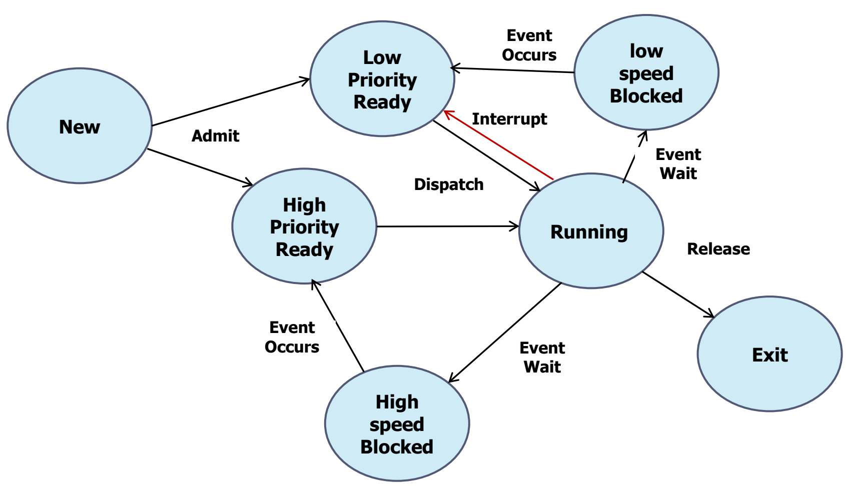
Multiprogramming with Priorities



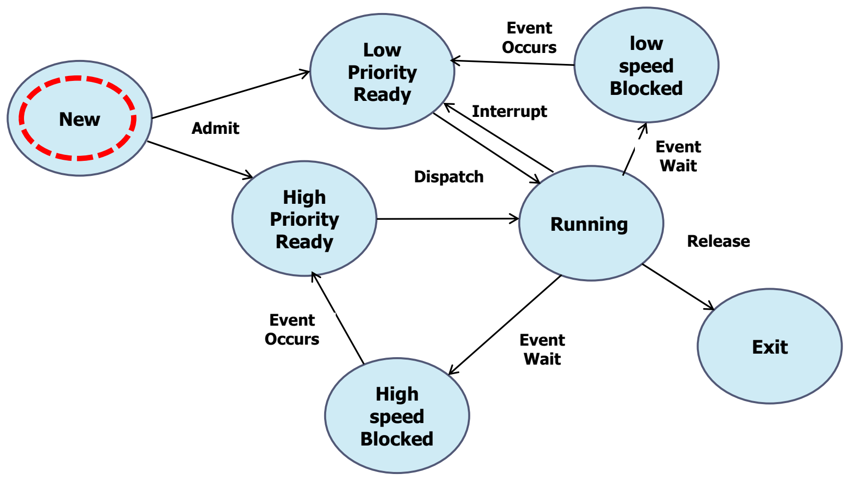
Multiprogramming with Priorities



Multiprogramming with Priorities

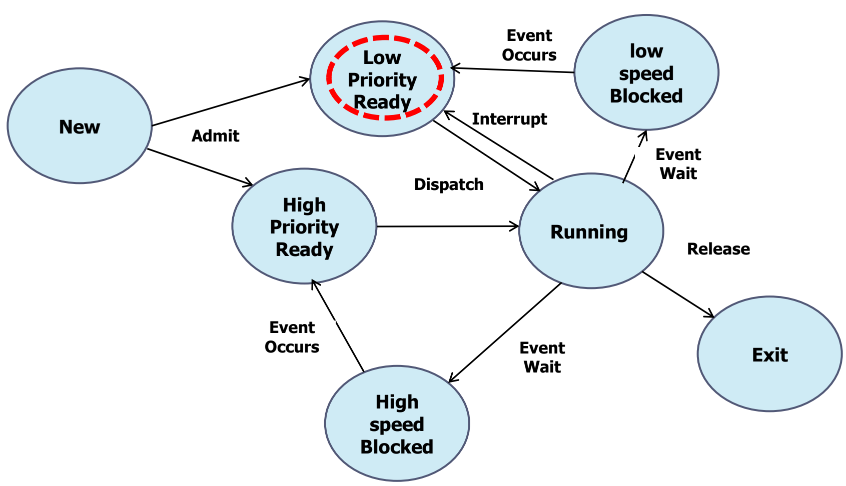


Multiprogramming with Priorities



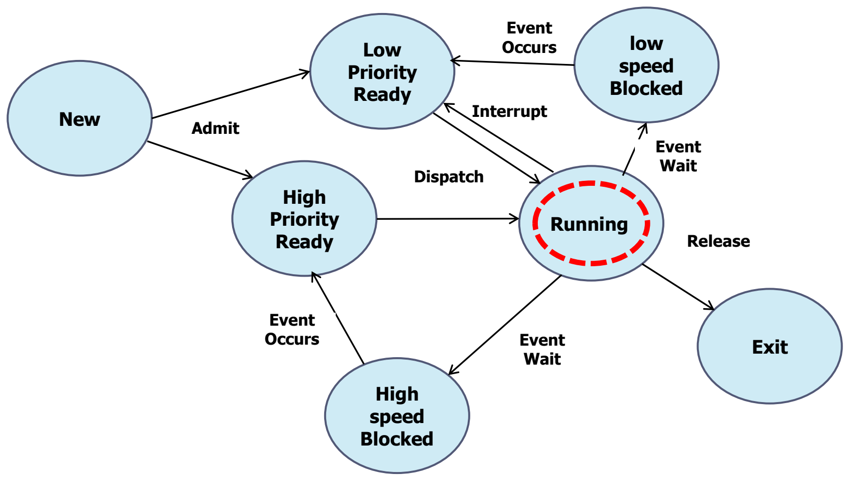
29

Multiprogramming with Priorities



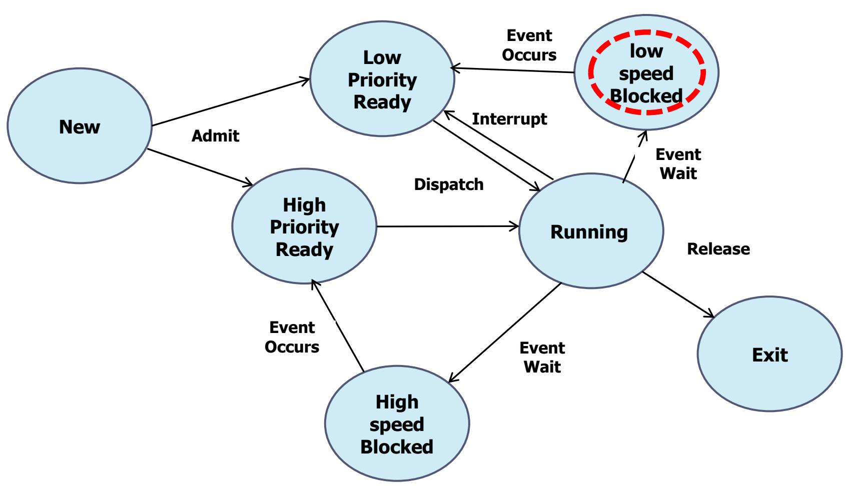
30

Multiprogramming with Priorities



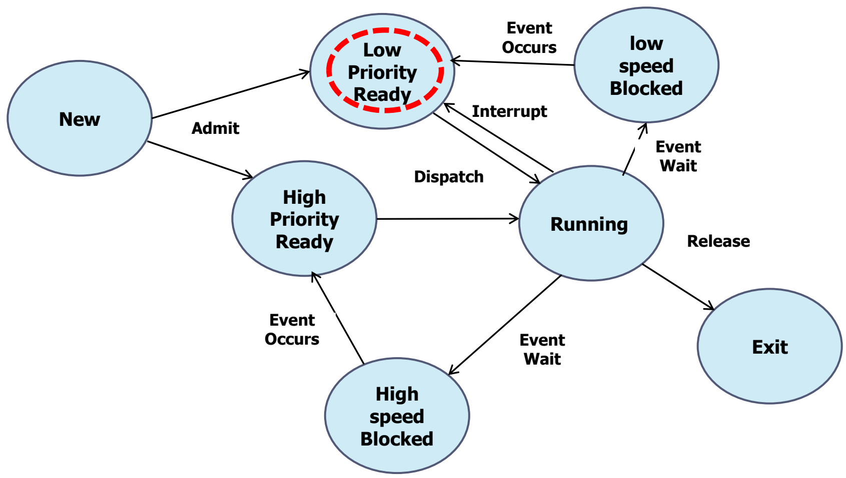
31

Multiprogramming with Priorities



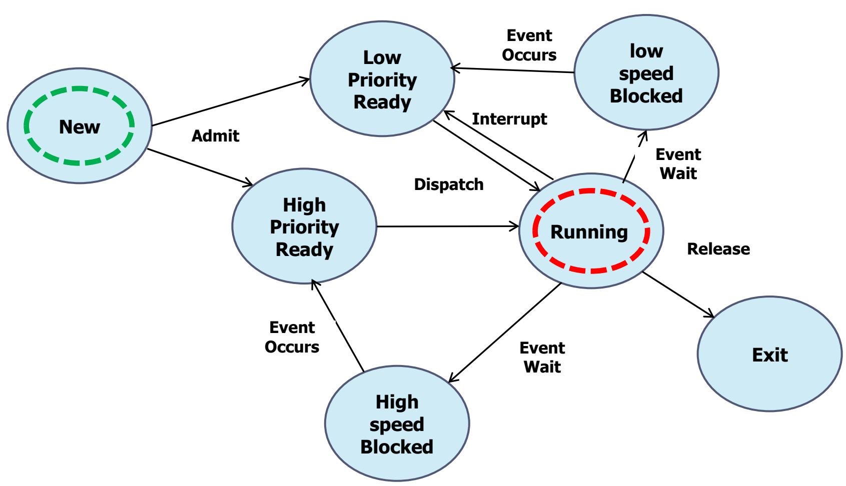
32

Multiprogramming with Priorities



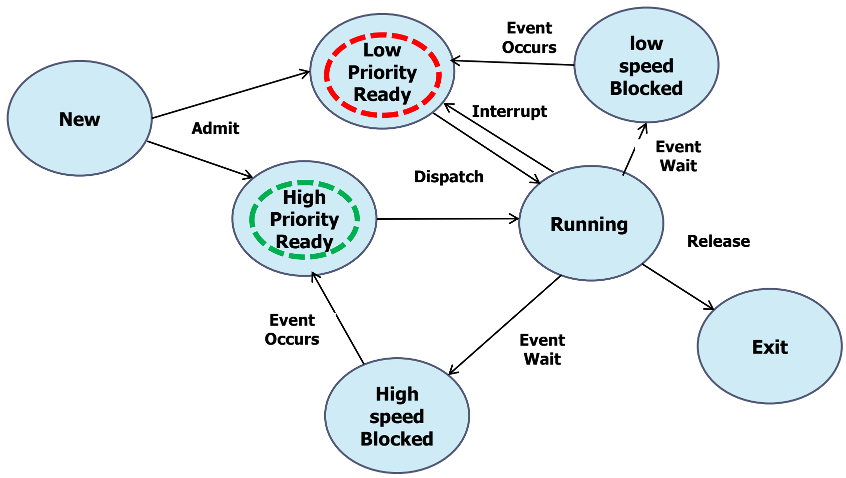
33

Multiprogramming with Priorities



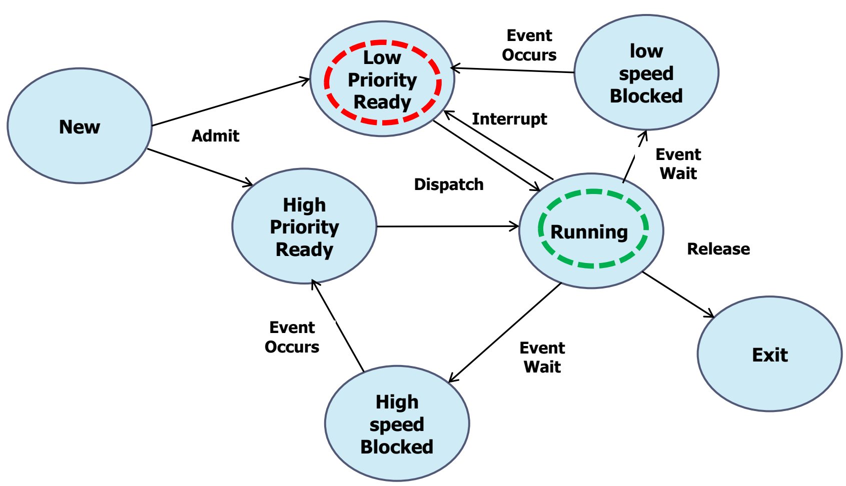
34

Multiprogramming with Priorities



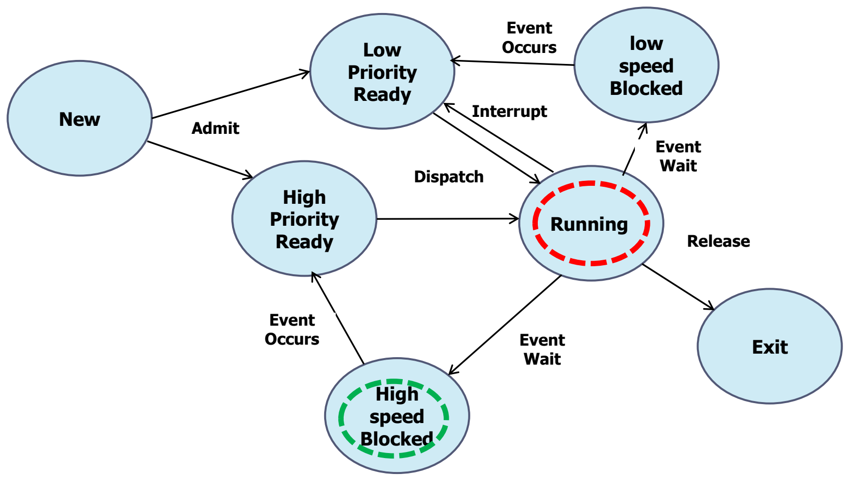
35

Multiprogramming with Priorities

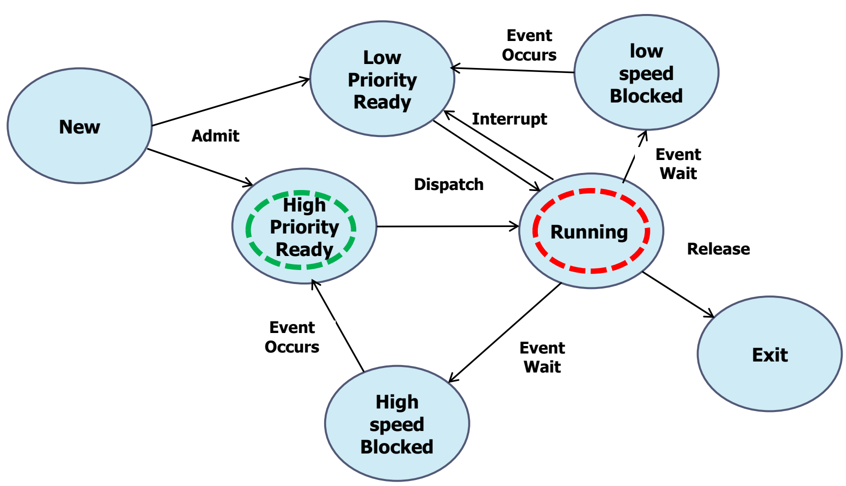


36

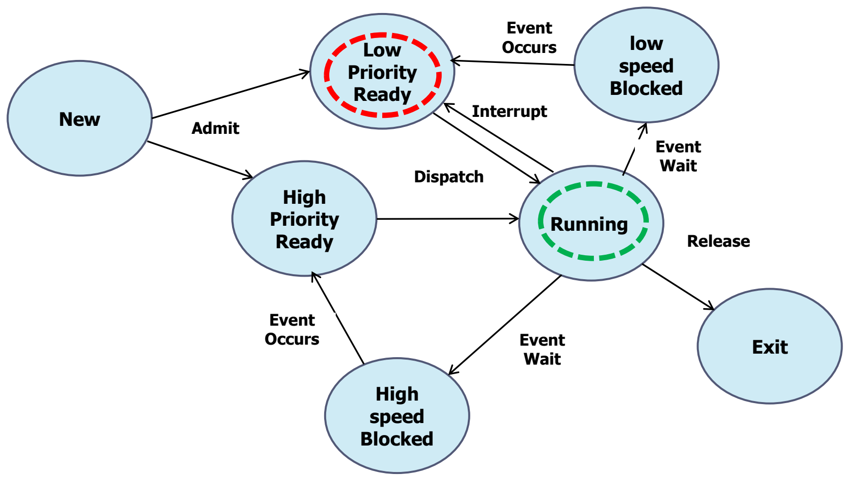
Multiprogramming with Priorities



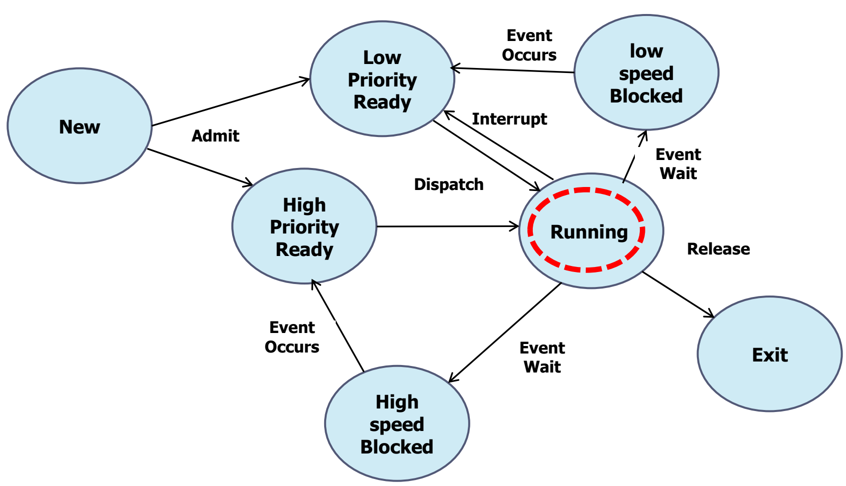
Multiprogramming with Priorities



Multiprogramming with Priorities



Multiprogramming with Priorities



Multiprogramming with Priorities

- Multiprogramming with priorities OS, priorities are assigned statically (at process creation time)

- Three types of priorities (Low, Medium and High)
- IO devices with Three speeds (Low, Medium and High)
- Low/Medium/High priority processes use Low/Medium/High speed IO devices respectively
- Different lists for Ready and Blocked states of different priorities and different speed IO devices

- Draw the Process Model

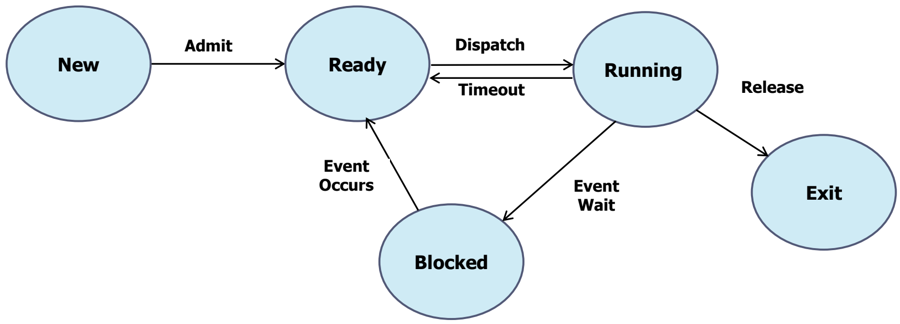
Time-sharing OS

- Preemptive CPU Scheduling

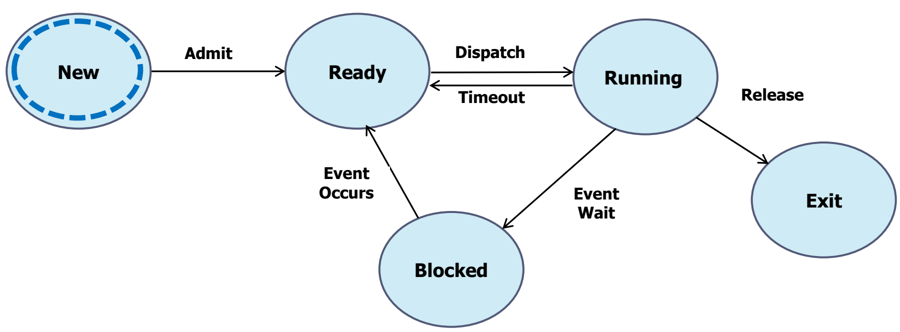
- FIFO Ready List

- Context Switch at time-out (timeslot completion)
  - Running to Ready (at rear pointer of list)
  - Ready (from front pointer) to Running

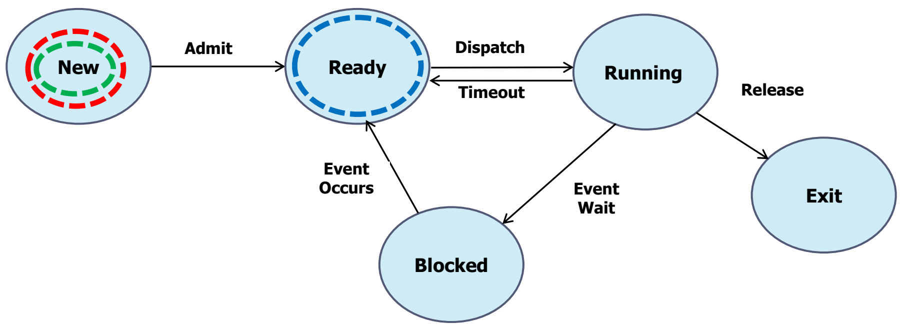
Time-sharing OS



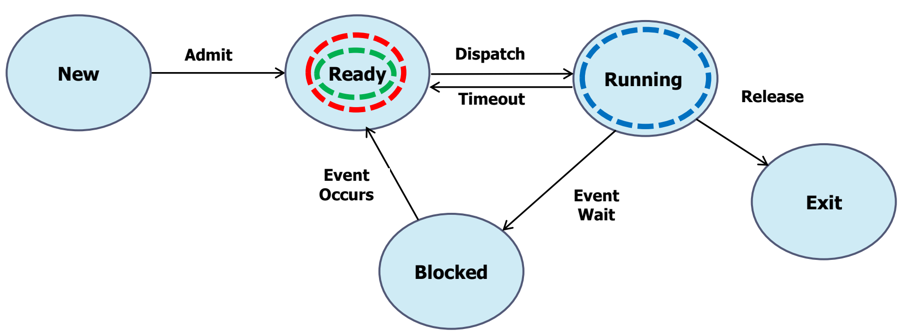
Time-sharing OS



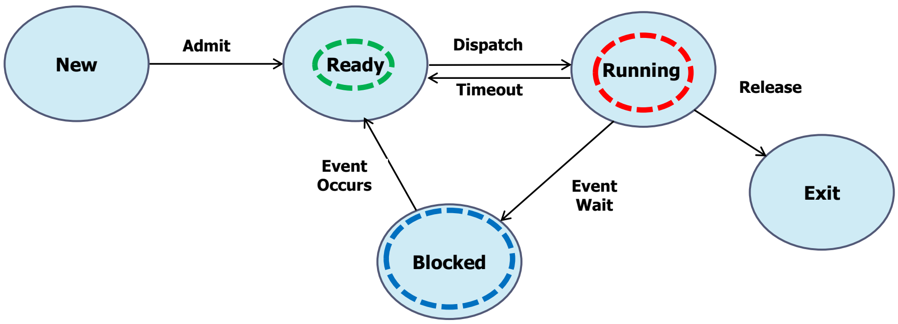
Time-sharing OS



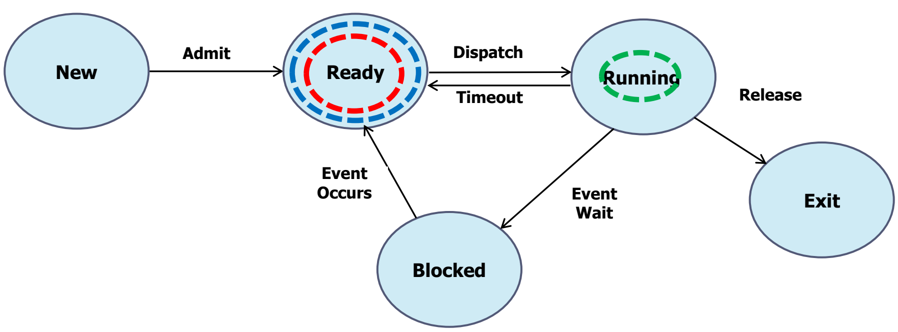
Time-sharing OS



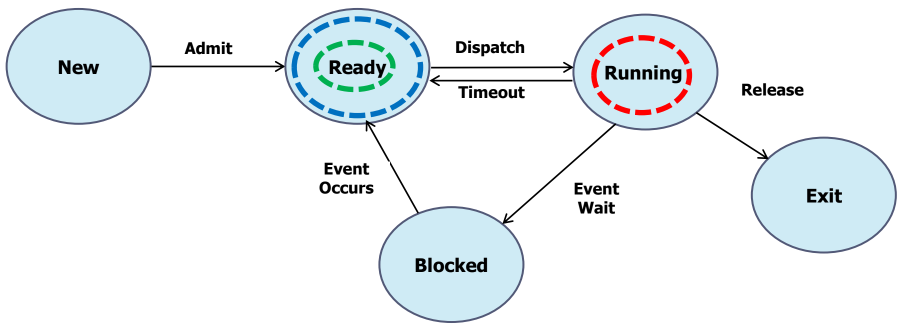
Time-sharing OS



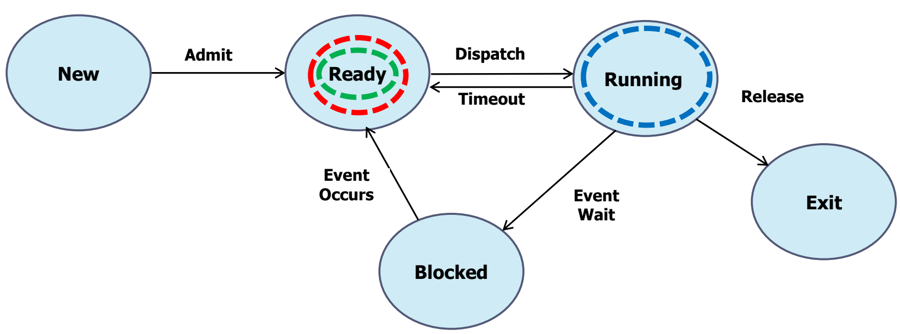
Time-sharing OS



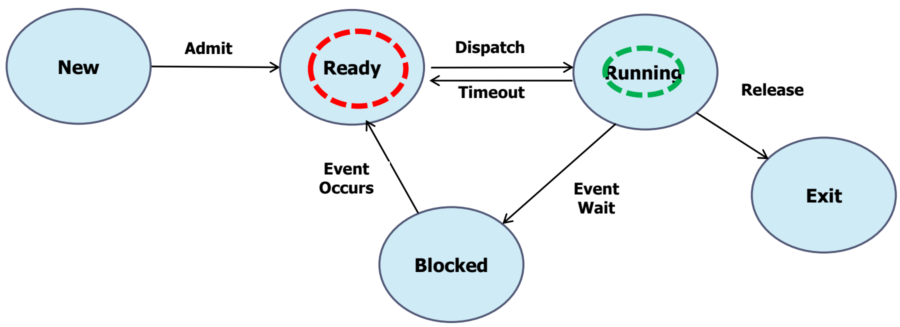
Time-sharing OS



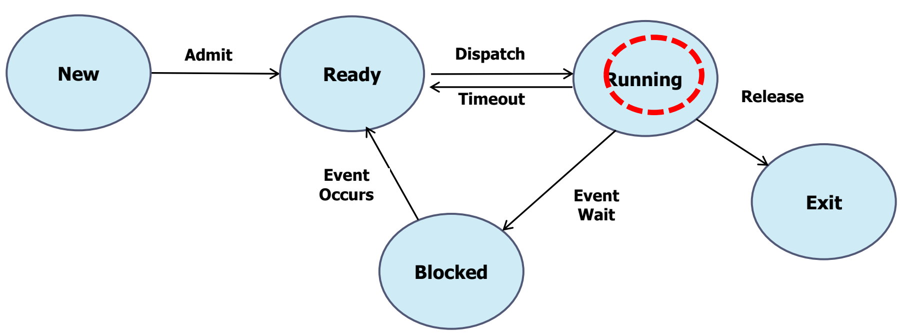
Time-sharing OS



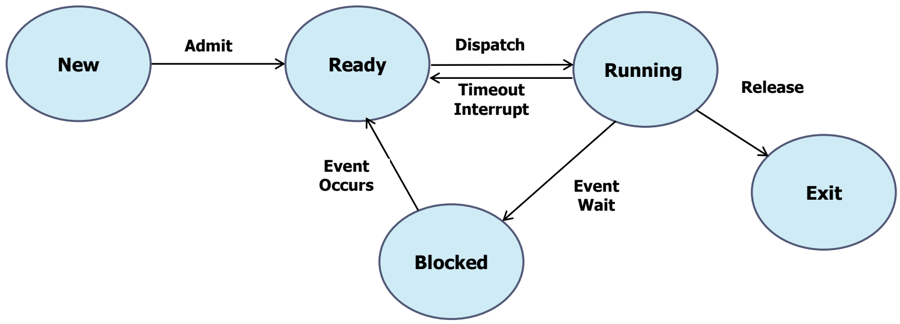
Time-sharing OS



Time-sharing OS



General Purpose OS



Computer/Operating System: Basics

CPUs have 2 modes

- Supervisor mode
  - Issue commands to hardware devices
  - Power off, Reboot, Suspend
  - Launch missiles, Do awesome stuff
- User mode
  - Run other code, hardware snitches if you try anything reserved for the supervisor

Computer/Operating System: Basics

Supervisor (privileged) mode

- All processor instructions are available including control instructions
- E.g., enable/disable interrupts, change the page table, performance counters, …
- All general-purpose as well as control registers are accessible

User (non-privileged) mode

  - Only a subset of "harmless" processor instructions are available
  - Arithmetic and logic operations, branches, memory load/store
  - Only a few general-purpose registers accessible

Computer/Operating System: Basics

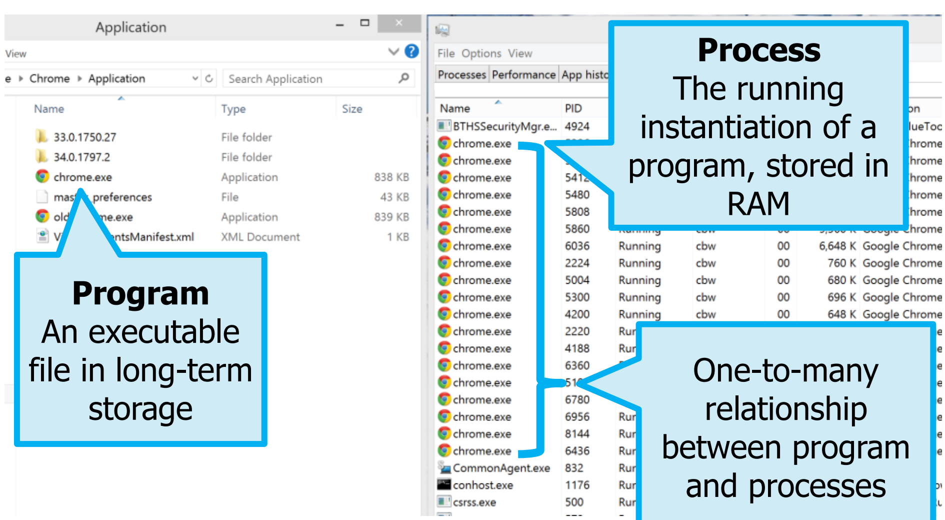
User mode

- Applications exclusively run in the non-privileged mode
- Can do whatever permitted in that mode without OS intervention
  - Change register values, read/write their own stack or heap, do ALU operations, take branches, call functions in their code segment, etc.
- Anything else requires switching to privileged mode (i.e., making a syscall) at which point the kernel takes over
- How is this mode transfer (user-to-supervisor and vice versa) implemented ?
- interrupts

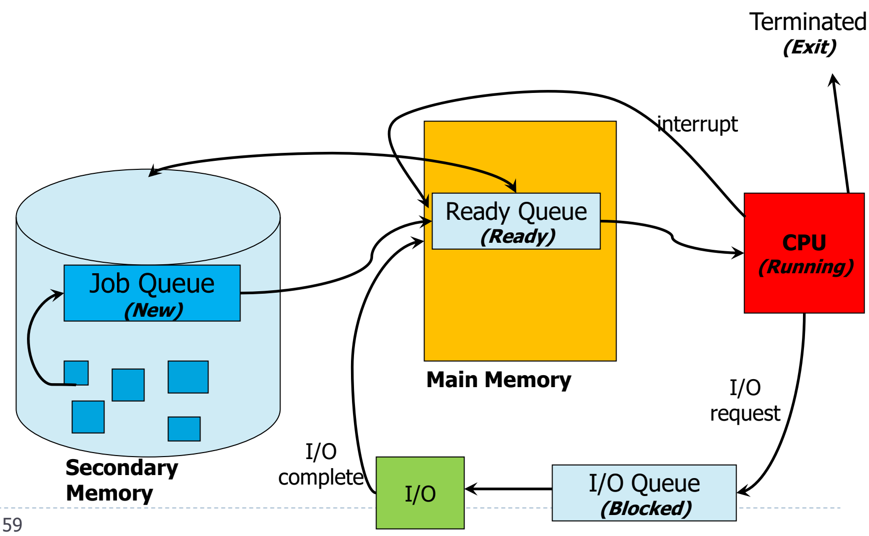
Processes

- Process: program in execution
  - Address space (memory) the program can use
  - State (registers, including PC & stack pointer)
  - OS keeps track of all processes in a process table

Program → Process

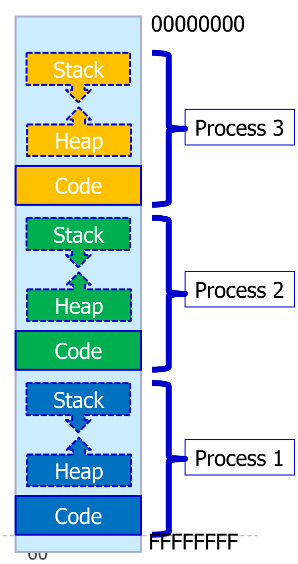


Process States: Hardware View



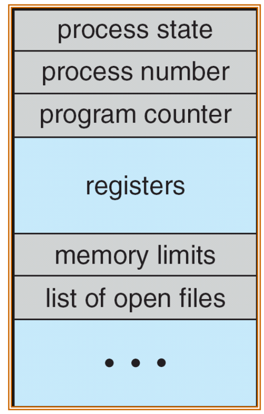
Program → Process

- OS functionality that loads programs into memory, creates processes
- Addresses are loaded → Address space

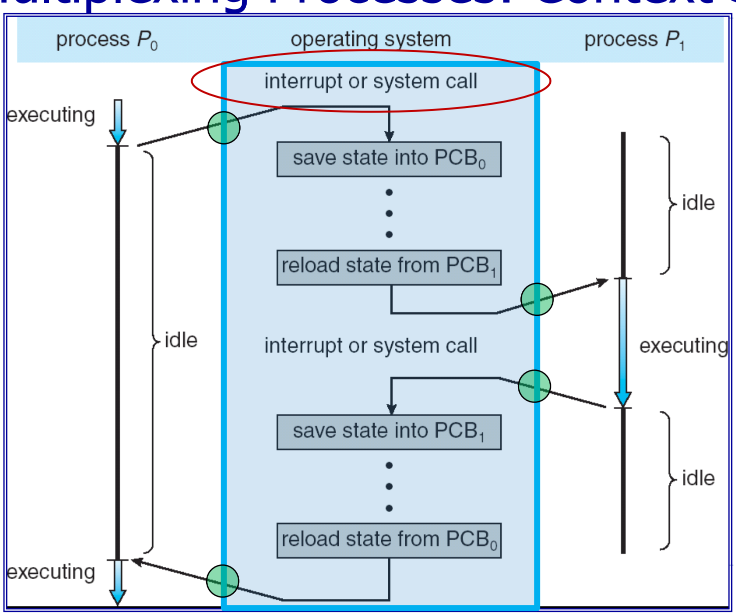


Process

- Process is represented as a process control block (PCB) in kernel
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …



Multiplexing Processes: Context Switch



System Call

- System Call
  - Special instruction to switch from user to supervisor mode
  - Transfers CPU control to the kernel
    - One of a small-ish number of well-defined functions

- How many system calls does Windows or Linux have?
  - Windows ~1200
  - Linux ~350

63

System Calls

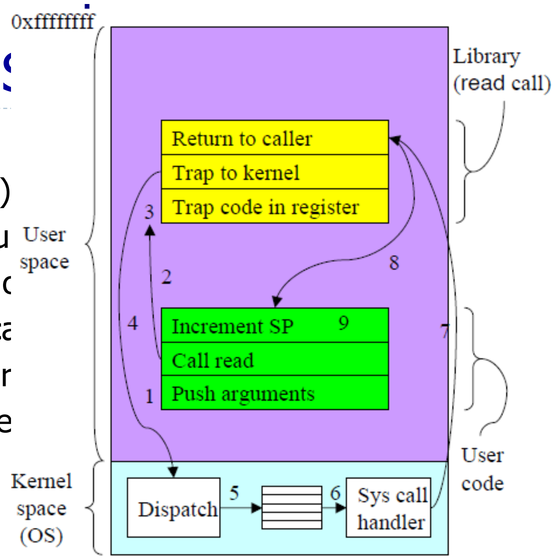
- Programs want the OS to perform a service
  - Access a file
  - Create a process
  - Others…
- Accomplished by system call
- Program passes relevant information to OS
- OS performs the service if
  - the OS is able to do so
  - the service is permitted for this program at this time
- OS checks information passed to make sure it's OK
  - Don't want programs reading data into other programs' memory!

64

Making a S

System call:

- read(fd,buffer,length)
- Program pushes argu
- Library sets up trap, c
- OS handles system ca
- Control returns to libr
- Library returns to use



System Calls for Files-Directories

Call	Description
fd = open(name,how)	Open a file for reading and/or writing
s = close(fd)	Close an open file
n = read(fd,buffer,size)	Read data from a file into a buffer
n = write(fd,buffer,size)	Write data from a buffer into a file
s = lseek(fd,offset,whence)	Move the "current" pointer for a file
s = stat(name,&buffer)	Get a file's status information (in *buffer*)
s = mkdir(name,mode)	Create a new directory
s = rmdir(name)	Remove a directory (must be empty)
s = link(name1,name2)	Create a new entry (*name2*) that points to the same object as *name1*
s = unlink(name)	Remove *name* as a link to an object (deletes the object if *name* was the only link to it)

Systems Calls for Process

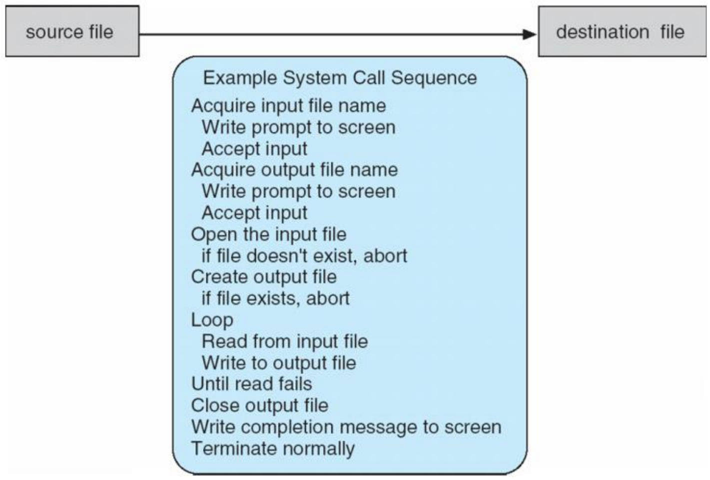
Call	Description
fd = open(name,how)	Open a file for reading and/or writing
s = close(fd)	Close an open file
n = read(fd,buffer,size)	Read data from a file into a buffer
n = write(fd,buffer,size)	Write data from a buffer into a file
s = lseek(fd,offset,whence)	Move the "current" pointer for a file
s = stat(name,&buffer)	Get a file's status information (in *buffer*)
s = mkdir(name,mode)	Create a new directory
s = rmdir(name)	Remove a directory (must be empty)
s = link(name1,name2)	Create a new entry (*name2*) that points to the same object as *name1*
s = unlink(name)	Remove *name* as a link to an object (deletes the object if *name* was the only link to it)

System Calls

- System calls provide an interface to the services made available by an OS
- Generally available as routines written in C and C++, although certain low-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly-language instructions

System Call Example

- Copy the contents of one file to another file



System Calls

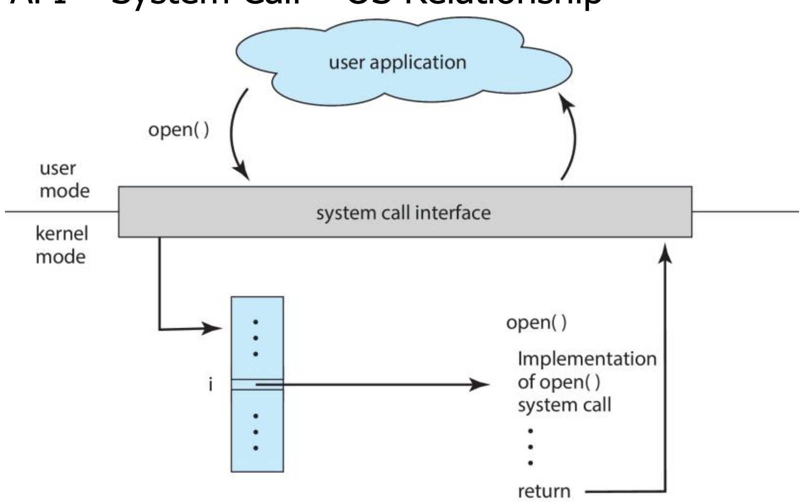
- Even simple programs may make heavy use of OS frequently, systems execute thousands of system calls per second
- Most programmers never see this level of detail, however
- Typically, application developers design programs according to an application programming interface (API)
- The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function, and the return values the programmer can expect

System Calls

- A programmer accesses an API via a library of code provided by the operating system
- Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer
  - For example, the Windows function CreateProcess() (which unsurprisingly is used to create a new process) actually invokes the NTCreateProcess() system call in the Windows kernel

System Calls

- API – System Call – OS Relationship



System Calls Types

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

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

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

75

75

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

76

76

System Calls Types

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

77

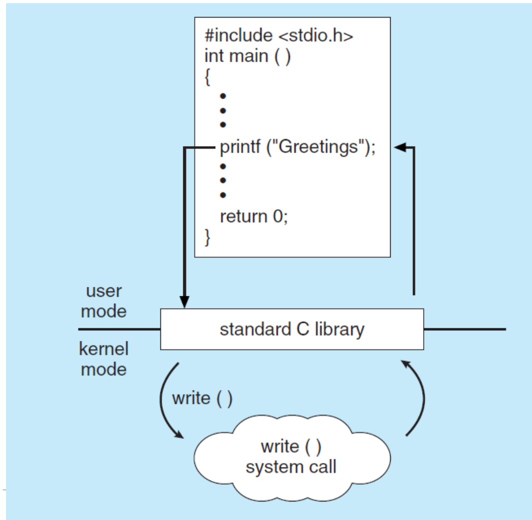
EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

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



79