Principles of Deadlock

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set

Example

System has 2 disk drives

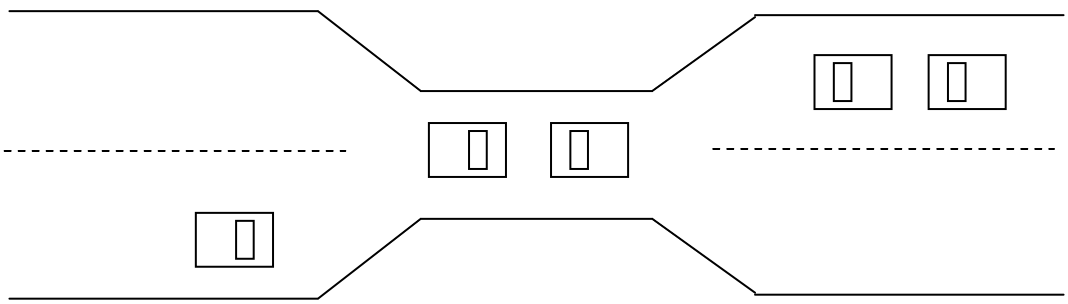
P_1 and P_2 each hold one disk drive and each needs another one

Example

semaphores *A* and *B*, initialized to 1

P_0	P
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- Note – Most OSes do not prevent or deal with deadlocks

System Model:

In any system, there are number of processes which are competing for the resources available in the system. Memory, CPU, files, input/output devices (like printer, scanner, etc.) are called resource types. Each resource type may have many instances i.e., the number of resource of the same type.

For example, printer is a particular resource type and if we have two printers in the system, we say that, we have two instances of resource type printer. If a process requests same resource, then any instance of that resource type should satisfy the need of process, only then they belong to same class of resource type.

Resource types R_1, R_2, . . ., R_m
CPU cycles, memory space, I/O devices
Each resource type R_i has W_i instances.

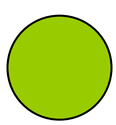
Each process utilizes a resource as follows:
request
use
release

Deadlock Characterization:

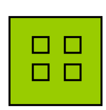
Deadlock can arise if four conditions hold simultaneously
Mutual exclusion: only one process at a time can use a resource
Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes
No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
Circular wait: there exists a set {P_0, P_1, …, P_0} of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, $P_{n–1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.
n

Resource-Allocation Graph
A set of vertices *V* and a set of edges *E*
V is partitioned into two types:
P = {P_1, P_2, …, P_n}, the set consisting of all the processes in the system
R = {R_1, R_2, …, R_m}, the set consisting of all resource types in the system
request edge – directed edge P_1 ® R_j
assignment edge – directed edge R_j ® P_i

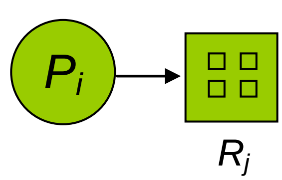
Process



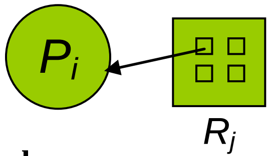
Resource Type with 4 instances



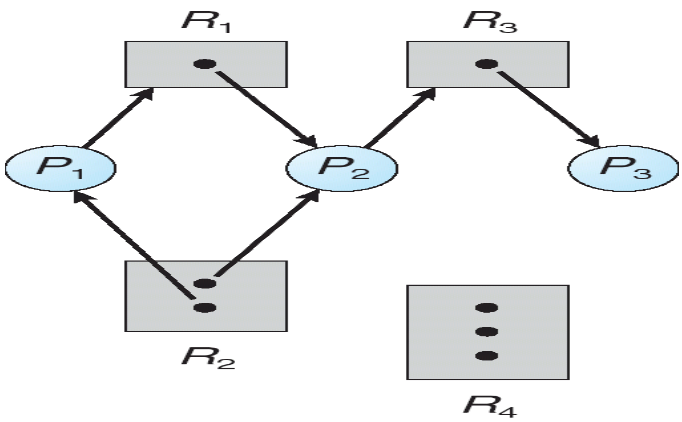
P_i requests instance of R_jn



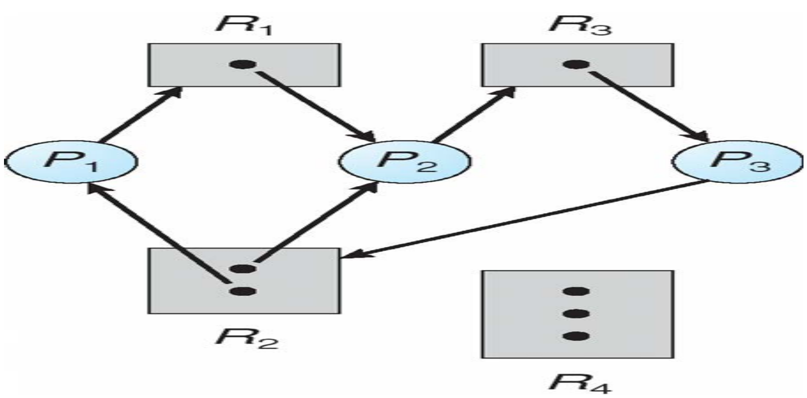
P_i is holding an instance of R_j



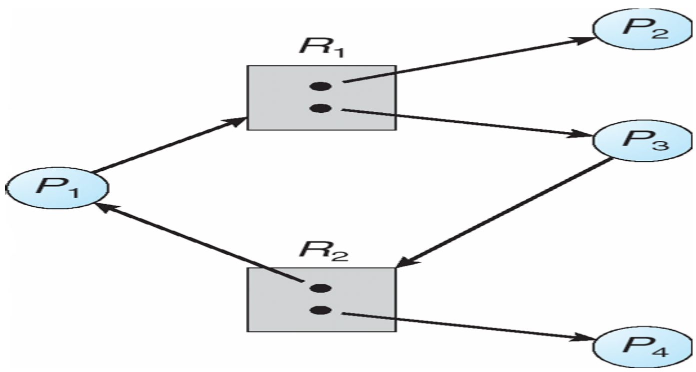
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts
If graph contains no cycles Þ no deadlocknIf graph contains a cycle Þlif only one instance per resource type, then deadlock
if several instances per resource type, possibility of deadlock
Methods for Handling Deadlocks
Ensure that the system will *never* enter a deadlock statenAllow the system to enter a deadlock state and then recovernIgnore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention:

Restrain the ways request can be made
Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources
Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources
Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none
Low resource utilization; starvation possible
No Preemption – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
Preempted resources are added to the list of resources for which the process is waiting
Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance:

Requires that the system has some additional *a priori* information available
Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need
The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State
When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes is the systems such that for each P_i, the resources that P_i can still request can

be satisfied by currently available resources + resources held by all the P_j, with $j < i$That is:

If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished

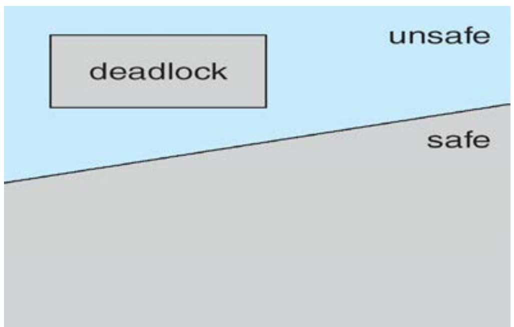
When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate

When P_i terminates, P_i +1 can obtain its needed resources, and so on

Basic Facts:

nIf a system is in safe state Þ no deadlocksnIf a system is in unsafe state Þ possibility of deadlocknAvoidance Þ ensure that a system will never enter an unsafe state.

Safe, Unsafe , Deadlock State



Avoidance algorithms:

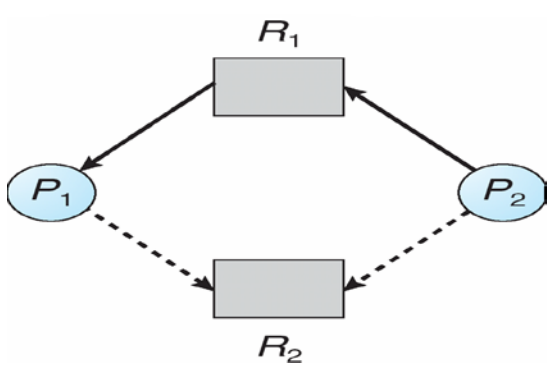
- Single instance of a resource type
- Use a resource-allocation graph
- Multiple instances of a resource type
- Use the banker's algorithm

Resource-Allocation Graph Scheme

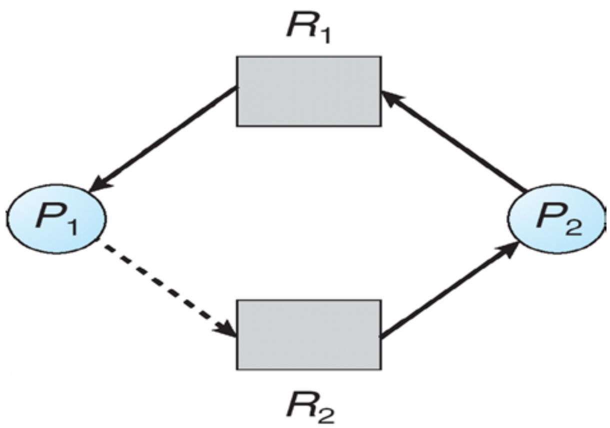
Claim edge P_i ® R_j indicated that process P_j may request resource R_j; represented by a dashed linenClaim edge converts to request edge when a process requests a resourcenRequest edge converted to an assignment edge when the resource is allocated to the process

When a resource is released by a process, assignment edge reconverts to a claim edgenResources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

Suppose that process P_i requests a resource R_j

The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

Multiple instancesnEach process must a priori claim maximum usenWhen a process requests a resource it may have to wait nWhen a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let *n* = number of processes, and *m* = number of resources types.

Available: Vector of length *m*. If available [*j*] = *k*, there are *k* instances of resource type R_j available

Max: *n x m* matrix. If *Max* [*i,j*] = *k*, then process P_i may request at most *k* instances of resource type R_jn**Allocation**: *n* x *m* matrix. If Allocation[*i,j*] = *k* then P_i is currently allocated *k* instances of R_jn**Need**: *n* x *m* matrix. If *Need*[*i,j*] = *k*, then P_i may need *k* more instances of R_j to complete its task

Need [*i,j] = Max*[*i,j*] – *Allocation* [*i,j*]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

Work = *Available*

Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find and *i* such that both:

(a) *Finish* [*i*] = *false*(b) $Need_i$ £ *Work*

If no such *i* exists, go to step 4

3.*Work* = *Work* + $Allocation_i$

Finish[*i*]=*true*
go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Resource-Request Algorithm for Process P_i

1. *Request* = request vector for process P_i. If $Request_i$ [*j*] = *k* then process P_i wants *k* instances of resource type R_j1. If $Request_i$ £ $Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i$ £ *Available*, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

Available = Available – Request;
$Allocation_i = Allocation_i + Request_i$;
$Need_i = Need_i – Request_i$;

If safe Þ the resources are allocated to Pi
If unsafe Þ Pi must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

5 processes P_0 through P_4;
3 resource types:
A (10 instances), *B* (5instances), and *C* (7 instances)
Snapshot at time T_0:

	Allocation	Max	Available
	A B C	ABC	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

The content of the matrix *Need* is defined to be *Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

nThe system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

Check that Request £ Available (that is, (1,0,2) £ (3,3,2) Þ true

	Allocation	Need	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement

Can request for (3,3,0) by P_4 be granted?

Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

nAllow system to enter deadlock state nDetection algorithmnRecovery scheme

Single Instance of Each Resource Type

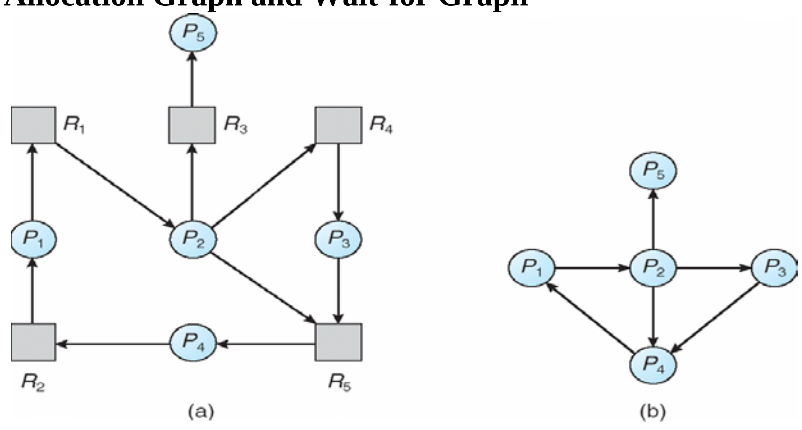
Maintain *wait-for* graph

Nodes are processes

P_i ® P_j if P_i is waiting for P_jnPeriodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock

An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

Available*:* A vector of length *m* indicates the number of available resources of each type.n

Allocation*:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.n**Request***:* An *n* x *m* matrix indicates the current request of each process. If *Request* [i_j] = *k*, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
(a) *Work* = *Available*(b) For *i* = 1,2, …, *n*, if $Allocation_i$ ¹ 0, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*2. Find an index *i* such that both:
(a) *Finish*[*i*] == *false*(b) $Request_i$ £ *Work*If no such *i* exists, go to step 4
3. *Work* = *Work* + $Allocation_i$
Finish[*i*] = *true*
go to step 24. If *Finish*[*i*] == false, for some *i*, 1 £ *i* £ *n*, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
Snapshot at time T_0:

	Available	Allocation	Request
	A B C	*A B C*	*ABC*
P_0	0 1 0	0 0 0	0 0 0
P_1		2 0 0	2 0 2
P_2	3 0 3	0 0 0	
P_3		2 1 1	1 0 0
P_4		0 0 2	0 0 2

Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*

P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

State of system?
Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

When, and how often, to invoke depends on:

How often a deadlock is likely to occur?
How many processes will need to be rolled back?
one for each disjoint cyclenIf detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes "caused" the deadlock

Recovery from Deadlock: Process Termination

Abort all deadlocked processesnAbort one process at a time until the deadlock cycle is eliminatednIn which order should we choose to abort?
Priority of the process
How long process has computed, and how much longer to completion
Resources the process has used
Resources process needs to complete
How many processes will need to be terminated
Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

Selecting a victim – minimize costnRollback – return to some safe state, restart process for that statenStarvation – same process may always be picked as victim, include number of rollback in cost factor

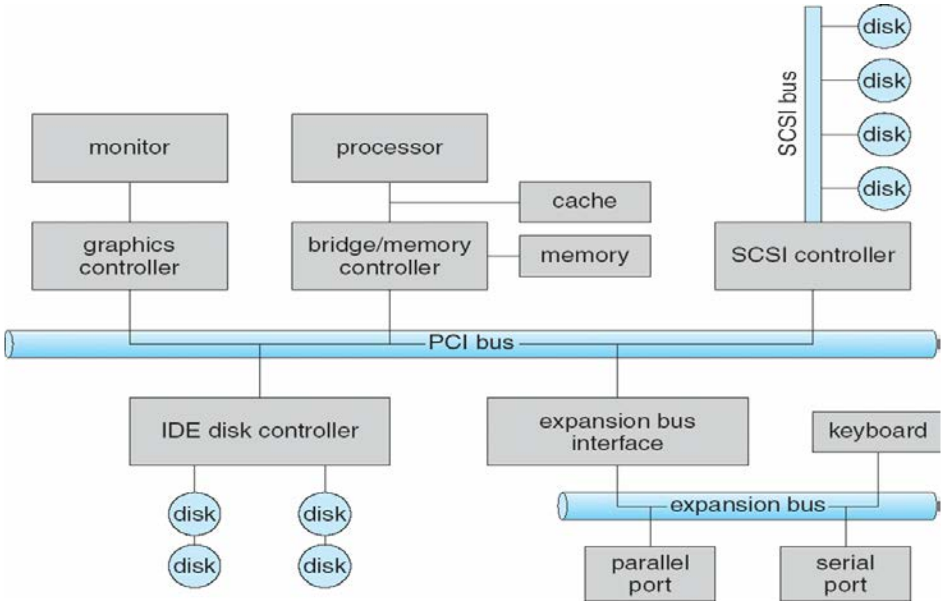
I/O Systems

Explore the structure of an operating system's I/O subsystem
Discuss the principles of I/O hardware and its complexity
Provide details of the performance aspects of I/O hardware and software

I/O Hardware

Incredible variety of I/O devices
Common concepts
lPort
lBus (daisy chain or shared direct access)
lController (host adapter)
I/O instructions control devices
Devices have addresses, used by
lDirect I/O instructions
lMemory-mapped I/O

A Typical PC Bus Structure



Device I/O Port Locations on PCs (partial)

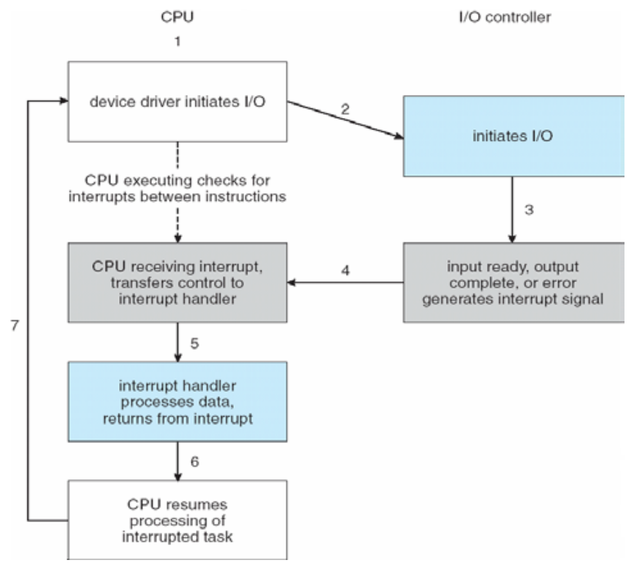
I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Polling

Determines state of device
command-ready

busy
ErrornBusy-wait cycle to wait for I/O from device
Interrupts
CPU Interrupt-request line triggered by I/O devicenInterrupt handler receives interruptsnMaskable to ignore or delay some interruptsnInterrupt vector to dispatch interrupt to correct handler
Based on priority
Some nonmaskablenInterrupt mechanism also used for exceptions

Interrupt-Driven I/O Cycle



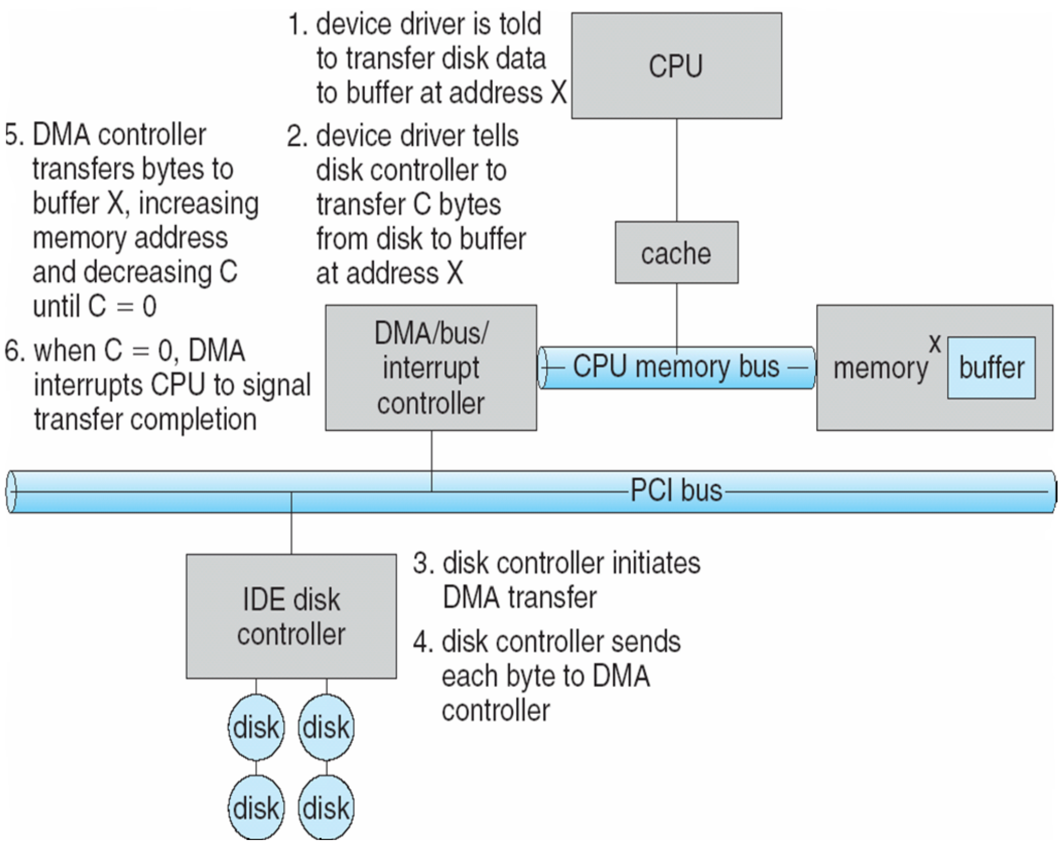
Intel Pentium Processor Event-Vector Table

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Direct Memory Access

Used to avoid programmed I/O for large data movement nRequires DMA controllernBypasses CPU to transfer data directly between I/O device and memory

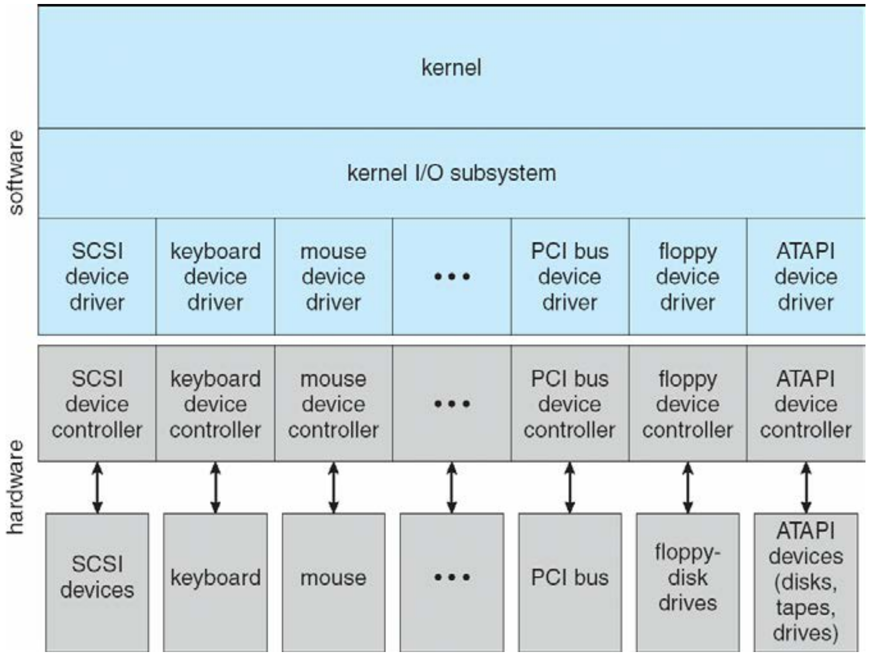
Six Step Process to Perform DMA Transfer



Application I/O Interface

I/O system calls encapsulate device behaviors in generic classes
Device-driver layer hides differences among I/O controllers from kernel
Devices vary in many dimensions
Character-stream or block
Sequential or random-access
Sharable or dedicated
Speed of operation
read-write, read only, or write only

A Kernel I/O Structure



Characteristics of I/O Devices

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Block and Character Devices

Block devices include disk drives

Commands include read, write, seek

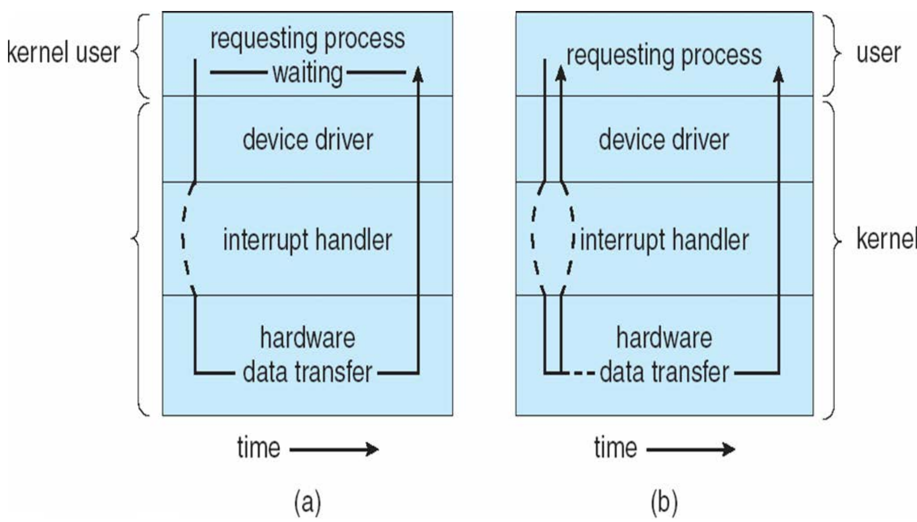
Raw I/O or file-system access

Memory-mapped file access possiblenCharacter devices include keyboards, mice, serial ports
Commands include get(), put()lLibraries layered on top allow line editing

Network Devices
Varying enough from block and character to have own interfacenUnix and Windows NT/9*x*/2000 include socket interface
Separates network protocol from network operation
Includes select() functionalitynApproaches vary widely (pipes, FIFOs, streams, queues, mailboxes)

Clocks and Timers
Provide current time, elapsed time, timernProgrammable interval timer used for timings, periodic interruptsnioctl() (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Nonblocking I/O
Blocking - process suspended until I/O completed
Easy to use and understand
Insufficient for some needsnNonblocking - I/O call returns as much as available
User interface, data copy (buffered I/O)
Implemented via multi-threading
Returns quickly with count of bytes read or writtennAsynchronous - process runs while I/O executes
Difficult to use
I/O subsystem signals process when I/O completed
Two I/O Methods

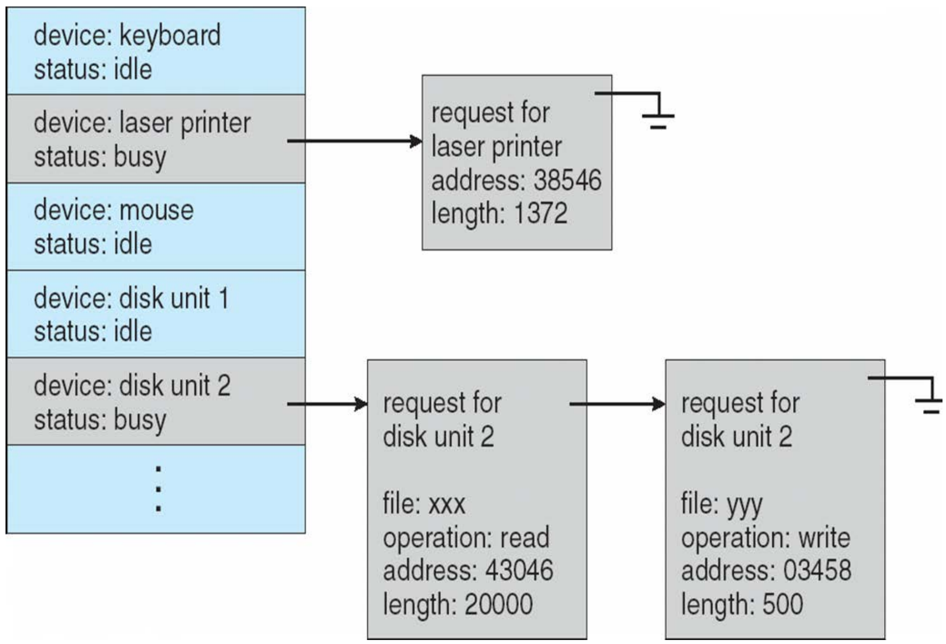


Synchronous Asynchronous

Kernel I/O Subsystem
Scheduling
Some I/O request ordering via per-device queue

Some OSs try fairnessnBuffering - store data in memory while transferring between devices
To cope with device speed mismatch
To cope with device transfer size mismatch
To maintain “copy semantics”

Device-status Table



Sun Enterprise 6000 Device-Transfer Rates