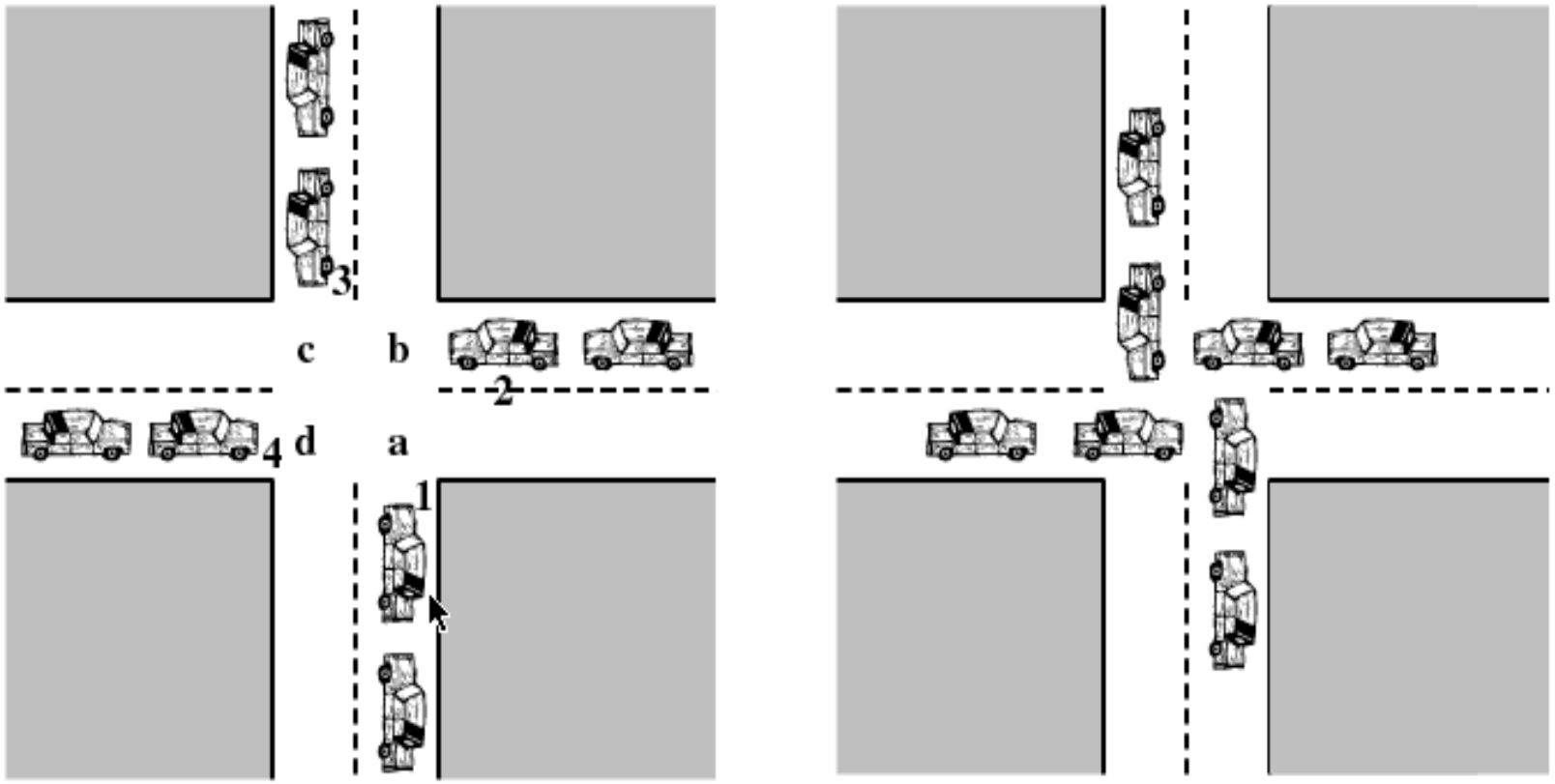
System model

- Deadlock:- A process requests resources; if the resources are not available at that time , the process enters a waiting state. Waiting process may never again change state, because the resources they have requested are held by other waiting process. This situation is called a ***Deadlock.***

OR

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.
- A system consists of a finite number of resources and distributed all the processes.
- The resources are Memory, CPU cycles, files, and IO devices (like printers fax, tape drives).



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

- A process must request a resource before using it, and must release the resource after using it.
- Every process may utilize resource some sequence of operations

1. Request:- If the request cannot be granted immediately then the requesting process must wait until it can acquire the resource.
2. Use:- The process uses/can operate on the resource.
3. Release:- The process release the resource.

- 1. Reusable Resources:-

- Used by only one process at a time and not depleted by that use.
- Processes obtain resources that they later release for reuse by other processes.
- Examples:- Processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores.
- Deadlock occurs if each process *holds one resource and requests the other*.

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

2. Consumable Resources/Constant resources :-

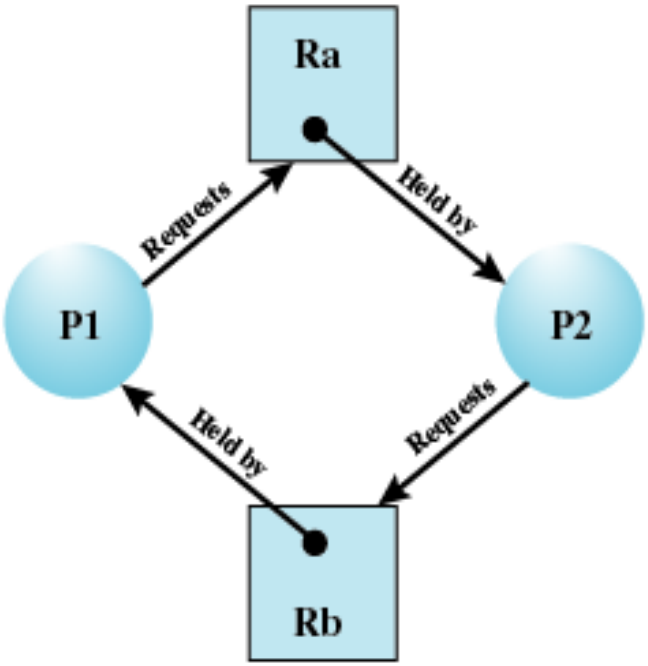
- Created (produced) and destroyed (consumed)
- Examples :-

  Interrupts, signals, messages, and information in I/O buffers.
- Deadlock may occur if a *Receive message* is blocking
- May take a rare combination of events to cause deadlock
- Deadlock occurs if receive is blocking

Deadlock Characteristics

- 1. Necessary Conditions/Conditions for Deadlock :-

  1. Mutual exclusion:-
     - At least one resources must be held in a *non-sharable* mode.
     - i.e.; Only one process may use a resource at a time
     - If another process request that resource, the requesting process must be delayed until the resource has been released.
  2. Hold-and-wait:-
     - A process holds one resource and wait for other resource.
     - A process may hold allocated resources while a waiting assignment of others. Example: R1 → P1 →R2
  3. No preemption:-
     - Resources cannot be preempted;
     - No resource can be forcibly removed form a process holding it.
  4. Circular wait:-
     - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.



(c) Circular wait

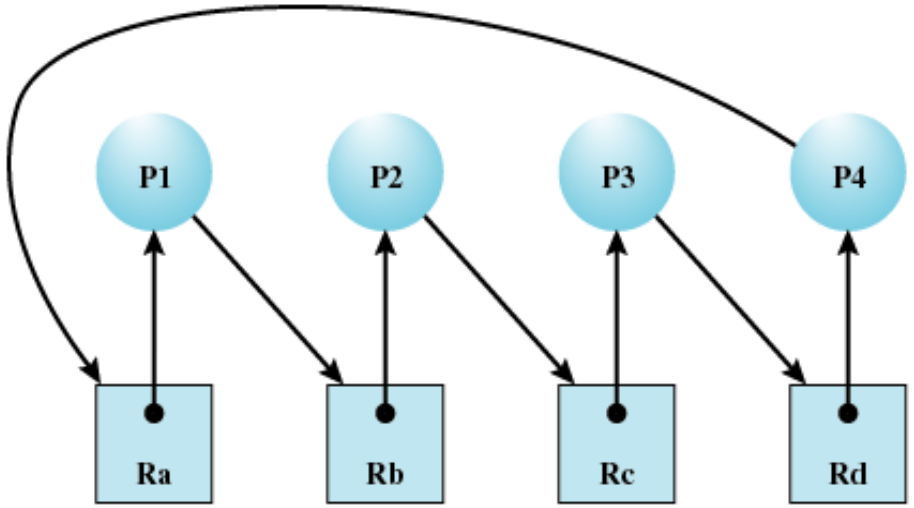
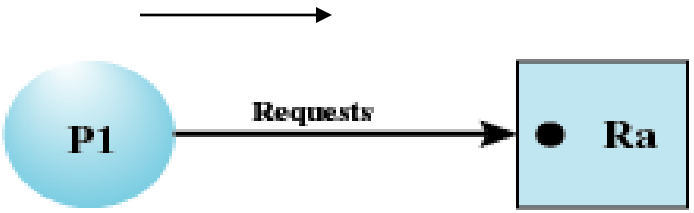


Figure 6.6 Resource Allocation Graph for Figure 6.1b

- 2. Resource Allocation Graphs:-
- Directed graph that describes a state of the system of resources and processes
- The graph G consists of a set of *vertices* V and a set of *edges* E.
- Vertices are 2 types of nodes:

  1. The set of active processes in the system P = { P1, P2, …, Pn }

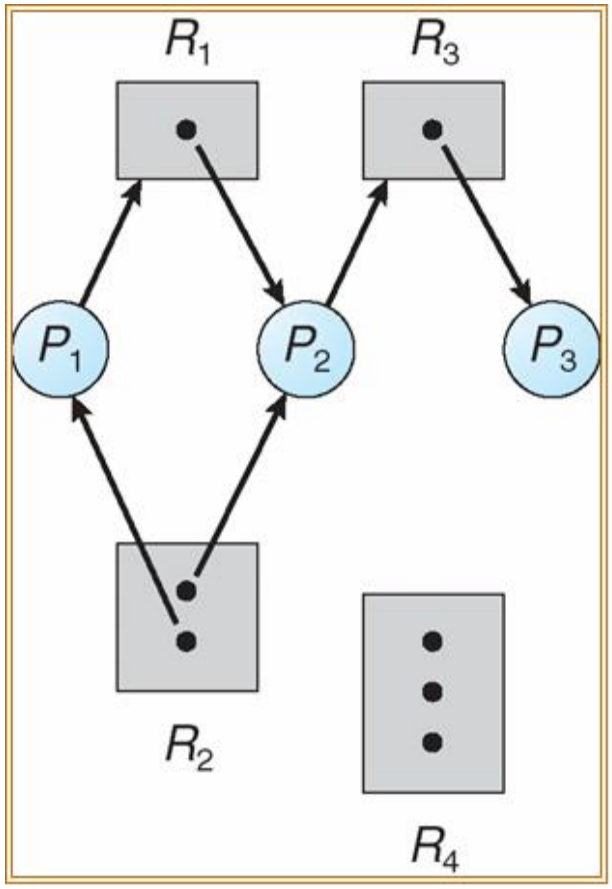
  2. The set of resource types in the system R= {R1, R2, …, Rm }
- A directed edge from process P1 to resource type Ra is denoted as P1 → Ra called as *request edge*.
- A directed edge from resource Ra to type process P1 is denoted as Ra → P1 called as *assignment* edge.



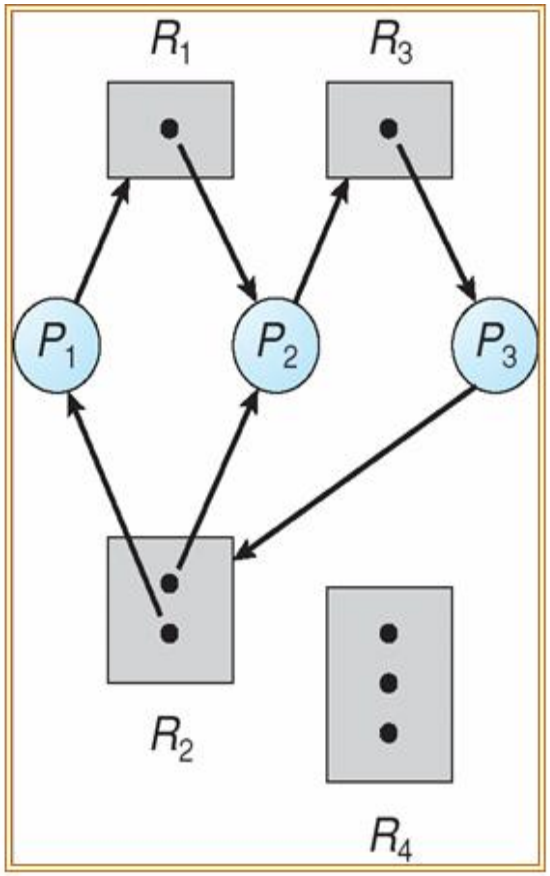
(a) Resouce is requested

(b) Resource is held

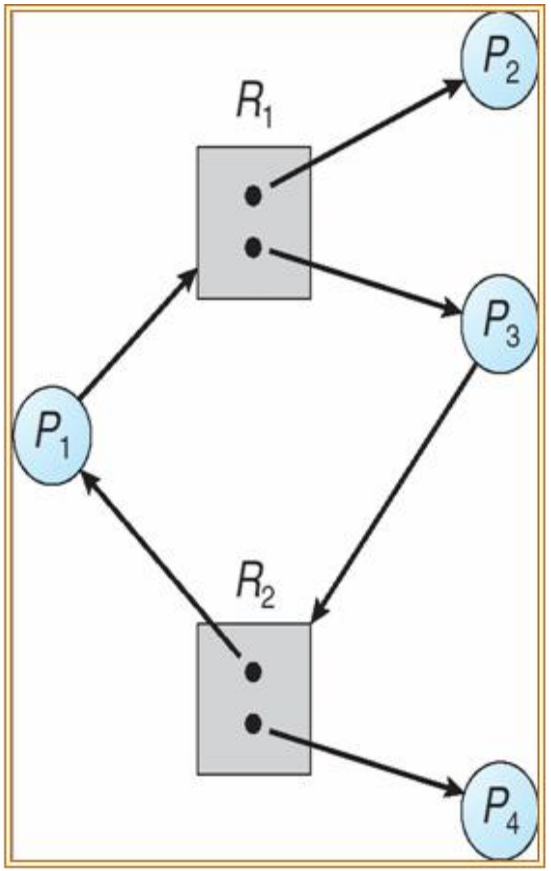
Resource allocation graphs



Resource Allocation Graph



Resource Allocation graph With A Deadlock



Graph With A Cycle But No Deadlock

- The sets P,R, and E:

  P = { P1, P2, P3 } R = {R1, R2, R3, R4}

  E = { P1 →R1, P2 → P3, R1 → P2, R2 →P2, R2 →P1, R3 → P3 }
- Resource instances:-

  One instance of resource type R1

  Two instance of resource type R2

  One instance of resource type R3

  Three instance of resource type R4
- Process states:

  1. R2 → P1 → R1
  2. R1,R2 → P2 → R3
  3. R3 →P3

Methods for handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.

1. Deadlock prevention
2. Deadlock avoidance

- We can deal with the deadlock problem in one of three ways:

1. We can *use a protocol* to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2. We can allow the system to *enter a deadlock state, detect it, and recover.*
3. We can ignore the problem altogether, and assume that deadlocks never occur in the system.

Deadlock Prevention

- We know the 4 conditions of occurring deadlock.
- Among the 4 conditions if we able to prevent any one of them automatically.

1. Mutual exclusion:-
   - We know that mutual exclusion as applicable for *non-sharable resource*. If we able to use sharable resource which can be shared by multiple process at a time then automatically it leads to prevents the mutual exclusion.
   - Must be supported by the operating system

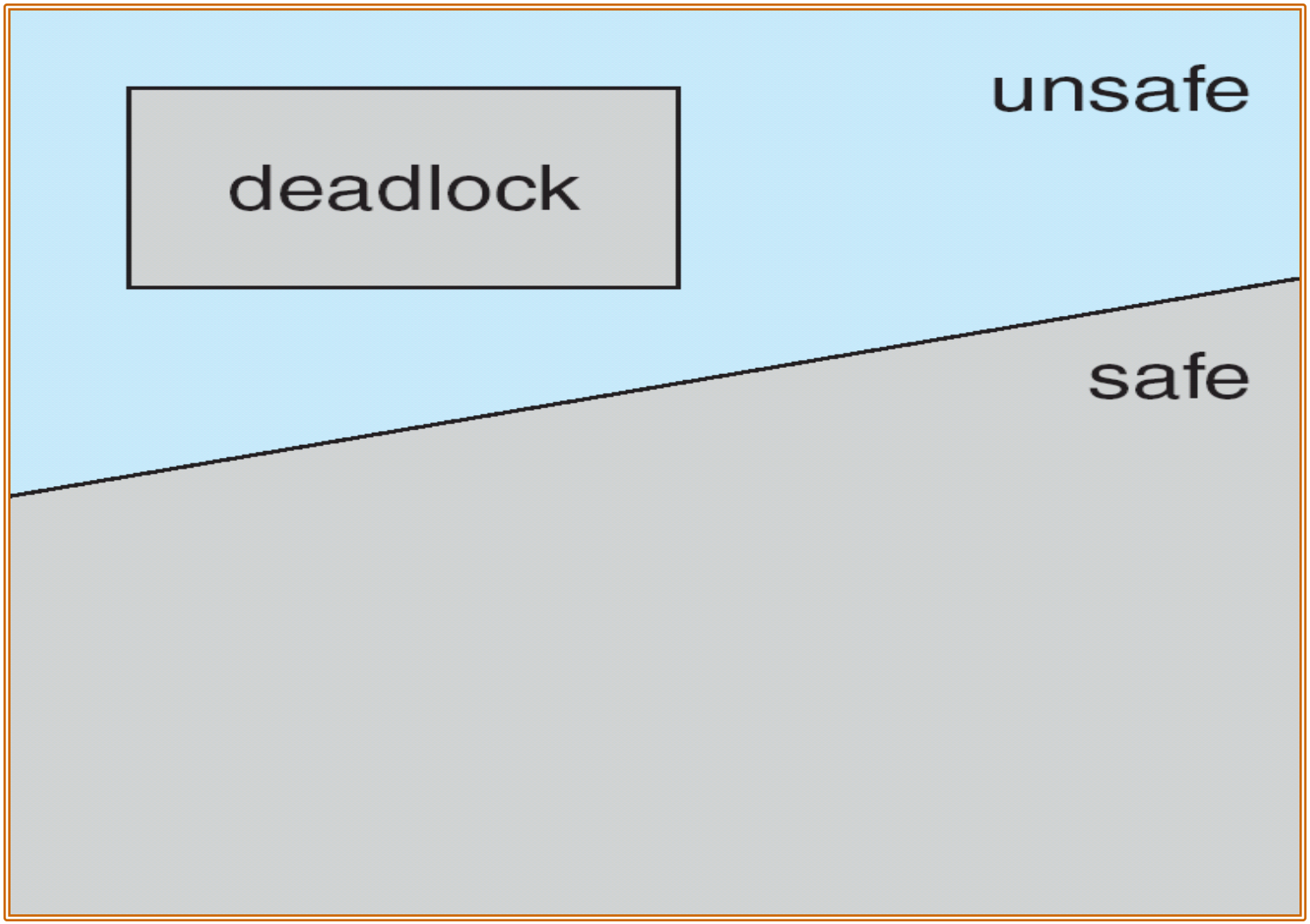
2. Hold and Wait:-
   - *Require a process request all of its required resources at one time.*
   - In order to prevent the hold and wait condition if a process requesting some resources, means the process doesn't hold any resources before requesting.
   - Due to this *first protocol* can be used by the processes requesting all resources needed before the execution being.
   - If the requesting resource is not available means it must wait until they are available after that the enters into execution.
   - *Another protocol* when a process to request the resource only when the process *has none*.
   - These protocols have 2 disadvantages
   - 1. Resource utilization problem
   - 2. Starvation problem.

3. No Preemption:-
   - Process must release resource and request again
   - Operating system may preempt a process to require it *releases its resources*
4. Circular Wait:-
   - Define a *numerical/linear* ordering of resource types
   - Here resources must be requested in the increasing order.
   - If P1 wants one resource(R1) then holding after P1 wants another(R2) then P1 must release the previous resource (R1) then another will be taken, So reducing the cycles.

Deadlock Avoidance

- If a process requests for a resources, the operating system will follow an algorithm known as **Safety algorithm.**
- The results of this algorithm are *safe state and unsafe state*.
- *State* of the system is the *current allocation* of resources to process
- 1. Safe state
- 2. Resource –allocation Graph Algorithm
- Referred to as the Banker's Algorithm
- Safe state is where there is at least one sequence that does not result in deadlock
- Unsafe state is a state that is not safe / result in Deadlock.

Safe, Unsafe, Deadlock State



Bankers Algorithm:

- **Available:** It deals with number of resources that are *available* in the system.
- **Maximum:** It deals with maximum number of resources that are *required* by a process to complete execution.
- **Allocation:** It deals with number of resources that are *allocated* to each process
- **Need:** It deals with number of resources that are left for each process. In other words, it deals with number of resources that are *required more* in order to complete execution.
- **Request:** It deals with number of resources that are *requested by the process* to complete execution, at a particular instant of time.

- **Algorithm:-**
- **Step 1**: If Request > Need

  then raise an error.
- **Step 2:** If Request < = Available then

  Available = Available – Request

  Allocation = Allocation + Request

  Need = Maximum – Allocation

  else

  Process Pi must wait.

- **Problem:-**
- Let us consider a system consisting of three processes *P1*, *P2 and P3*. The number of resources in the system is *12*. Let us assume that for process P1 to complete execution it requires 10 resources and for process P2 to complete execution it requires 4 resources and for process P3 it requires 9 resources.
- Process Maximum Allocation Need Available

P1	10	5	5	
P2	4	2	2	3
P3	9	2	7	

- **Answer:-**
- **Step 1:**

Process	Maximum	Allocation	Need	Request	Available
P1	10	5	5	5	
P2	4	2	2	2	3
P3	9	2	7	7	

- **Step 2:**
- In case of Step 1, all three processes are requesting operating system for resources. Then operating system will follow a formula,

Request < = Available

- For Process P1: 5 < = 3 → False
- For Process P2: 2 < = 3 → True
- For Process P3: 7 < = 3 → False

- Process Maximum Allocation Request Available
  P1 10 5 5
  P2 4 2+2 0 1
  P3 9 2 7

- **Step 3**
- Process Maximum Allocation Request Available
  P1 10 5 5
  P2 0 0 0 **4**+1
  P3 9 2 7
- **Request < = Available**
- For Process P1: 5 < = 5 → True
- For Process P3: 7 < = 5 → False

- **Step 4:**

Process	Maximum	Allocation	Request	Available
P1	10	5 + 5	0	
				0
P3	9	2	7	

Step 5:

Process	Maximum	Allocation	Request	Available
P1	0	0	0	
				10
P3	9	2	7	

- **Step 6:**

  Request < = Available

  For Process P3: 7 < = 10 → True

- Process Maximum Allocation Request Available

  P1 0 0 0

  3

  P3 9 2 + 7 0

- **Step 7:**

  Process Maximum Allocation Request Available

  P1 0 0 0

  9+3

  P3 0 0 0

  So now, all the 12 resources are free in the system.

Deadlock Detection

- If a system does not employ either *deadlock-detection or a deadlock avoidance* algorithm, then a deadlock situation may occur.
- System provide an algorithm that examines the state of the system to determine if a deadlock has occurred.
- An algorithm to Recover from the deadlock.

  1. Single Instance of Each Resource Type
  2. Several Instances of a Resource Type

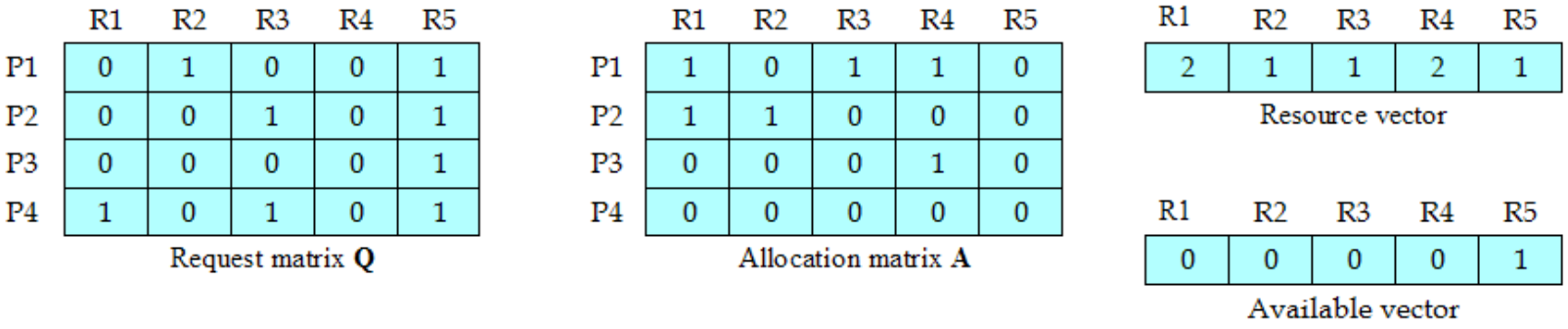
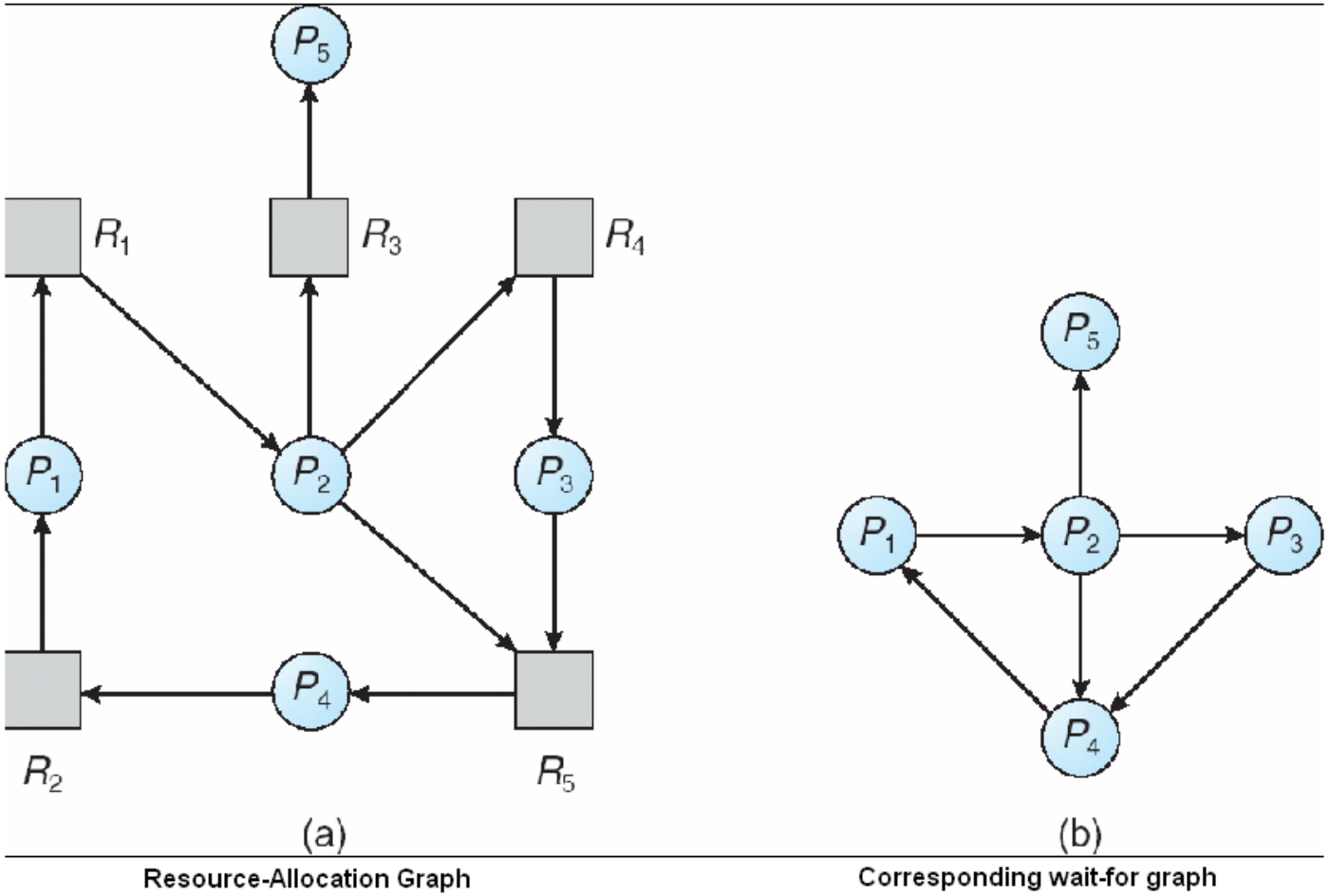


Figure 6.10 Example for Deadlock Detection

1. Single Instance of Each Resource Type:-

- All resources have *only one instance* then we can define a deadlock detection algorithm that uses a different of the resource allocation graph, called as ***Wait-for*** graph.
- We obtain this graph from *resource-allocation* graph by removing the *nodes of type resources* and collapsing the appropriate edges.

Resource allocation graph → wait-for graph



2. Several Instances of a Resource Type:-

- The *wait-for a graph* is not applicable to a resource allocation system with multiple instances of each type.
- The algorithm employs several time-varying data structures that are similar to those used in *Banker's algorithm*.
- **Available**:- a vector of length *m* indicates the number of *available* resources of each type.
- **Allocation**:- An *n* x *m* matrix defines the number of resources of each type *currently allocated* to each process.
- **Request**:- An *n* x *m* matrix indicates the *current request* of each process. If Request [i, j] =k then process is requesting *k* more instances of resource type *Rj*.

- The *Allocation* i and *Request* i are the row matrix of Allocation and Request vectors.

  1. Let *work and finish* be vectors of length *m and n* respectively.

Initialize Work:=Available. For i=1, 2, …, n, if *Allocation* 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 2.

4. If *finish [i]* = false, for some *i*, *1<=i<=n*, then the system is in a deadlock state. More over , if *Finish [i]=false*, then process Pi is deadlocked.

Deadlock Detection Algorithm:

- The operating system uses this algorithm to perform deadlock detection in a system containing *n processes* and *r resources*. This algorithm is used by operating system to see whether there is a deadlock or not.

- **Data Structures**
- **Free:** It is defined as an array that contains information about number of resources that are currently free.
- **Allocated:** It is defined as an array that contains information about number of resources that are allocated to each process**.**
- **Requested:** It is defined as an array that contains information about number of resources the process is requesting.
- **Running :** set of processes
- **Blocked:** set of processes.
- **Finished :** set of processes
- **n :** number of processes
- **r:** number of resources.

- **Step1: repeat until** set *Running* is empty
  - Select a process Pj from the set Running
  - After selecting, delete it from set Running and add it to set Finished.
  - for k=1..r
    Free[k] = Free[k] + Allocated[j,k]
  - while Blocked contains process Pl such that for k=1..r,
    Requested[l,k] < = Free[k]
- i) for k=1,r
  Free[k] = Free[k] – Requested[l,k];
- Allocated[l,k] = Allocated[l, k] + Requested[l,k];
- ii) Delete Pl from set Blocked and add it to set Running
- **Step 2:** if set Blocked is not empty then
- Declare processes in Blocked to be deadlocked

Strategies once Deadlock Detected

- Abort all deadlocked processes
- *Back up* each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Recovery from Deadlock

- Whenever there is a deadlock, the operating system recovers from a deadlock using two methods:

  1. Process Termination

  2. Resource Preemption

1. Process Termination:-

- In case of process termination, the operating system will follow two protocols as explained below:

  i). Kill all the deadlock processes:-

  This method clearly will break the deadlock cycle, then all resources belonging to above processes are free, but at a time great expense; these processes may have completed for a long time, and result of these partial computations must be discarded and probably recomputed later.

ii). <u>Kill one process at a time until the deadlock cycle is terminated</u>:-

In this method Kill one process at a time, after each process is terminated, a deadlock-detection algorithm must be invoked to determine if any processes are still deadlocked.

- Killing of a process is *not easy*, that is if the process is *updating a file*, terminating it will leave that file in an *incorrect state*.
- If the *partial termination* method is used , then given a set of deadlocked processes, we determine which process should be terminated in an attempt to break the deadlock.

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
- Many factors may determine which process is chosen, including:-

1. *Priority* of the process.
2. How long process has computed, and how much longer to completion.
3. How many and what type of resources the process has used.
4. How many more resources the process needs in order to complete.
5. How many processes will need to be terminated.
6. Whether the process is interactive OR batch.

2. **Resource Preemption:-**

- To eliminate deadlock using resource preemption, we successfully preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- If the preemption is required to deal with deadlocks, then 3 issues need to be addressed:

1. Selecting a Victim:-

- Which resource and which processes are to be preempted? As in the process termination, we must determine the order of preemption to minimize cost.

2. Rollback:-

If we preempt a resource from a process, what should be done with that process? It is missing some needed resource. We must rollback the process to some safe state, and restart from that state.

3. Starvation:-

How can we guarantee that resources will not always be preempted from the same process?

- Here the victim process is selected based on cost factor. As a result, this process never completes its designed task, a starvation situation will occur. And the most common situation solution is to include the number of rollbacks in the cost factor

Some More Examples of Bankers Algorithm and Deadlock Detection

Another 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		*A B C*	*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

- The 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 $\leq$ Available (that is, (1,0,2) $\leq$ (3,3,2) $\Rightarrow$ 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?

Another 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:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
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

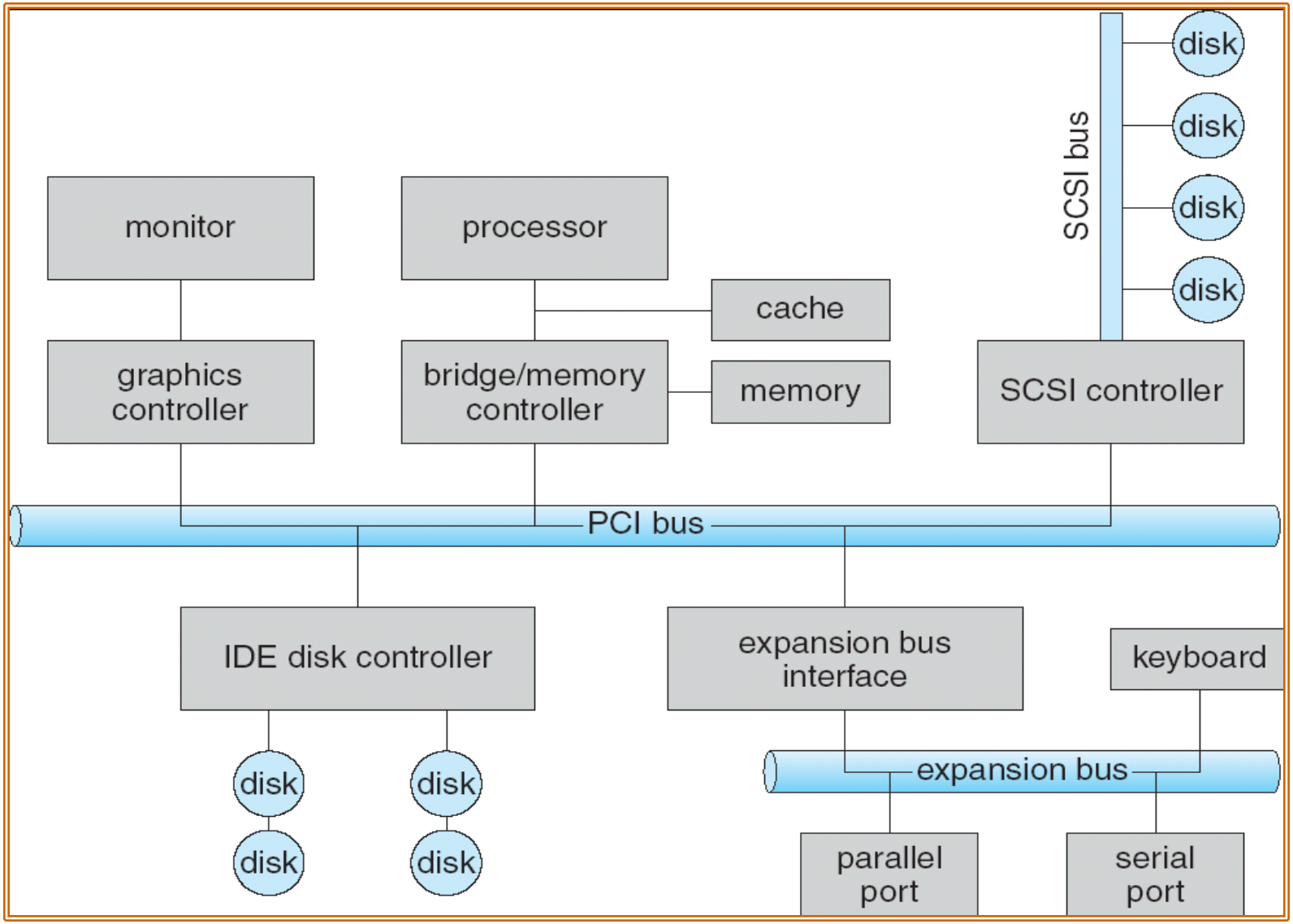
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

I/O Hardware

- Categories of I/O Devices:-
- Human readable:-
  - Used to communicate with the user
  - Printers
  - Video display terminals
    - Display
    - Keyboard
    - Mouse
- Machine readable:-
  - Used to communicate with electronic equipment
  - Disk and tape drives
  - Sensors
  - Controllers

- Communication:-
  - Used to communicate with remote devices
  - Digital line drivers
  - Modems
- Common concepts:-
  - Port
  - Bus (daisy chain or shared direct access)
  - Controller (host adapter)
- Devices have addresses, used by:-
  - Direct I/O instructions
  - Memory-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)

- A Controller
  - Host adapter
  - memory mapped IO
  - IO Port
- An IO port typically consists of 4 registers called as
  - status register :- contains a bit read by the host to get output
    The bit indicates states, i.e; current command completed, error,
  - control register :- written by the host to start a command/ change the mode of a device.
  - data-in register :- read by the host to get input
  - data-out register :- written by the host to send output

- Daisy chain:- When device A has a cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain.
- A daisy chain usually operates as a bus.

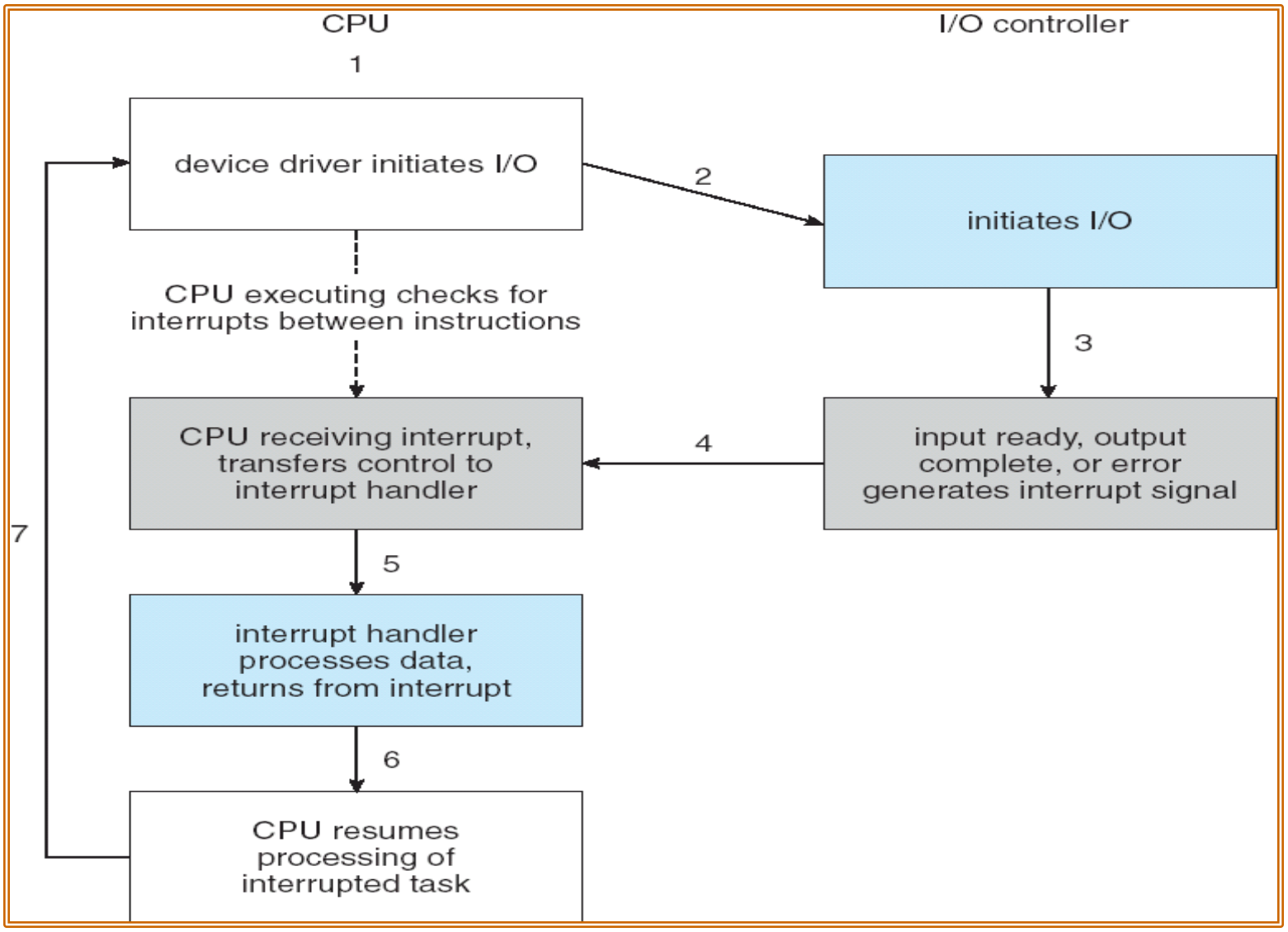
1. Polling:-

- The complete protocol for interaction between the *host and a controller*, but the basic handshaking notion is simple.
- Determines state of device
  - command-ready
  - busy
  - Error
- **Busy-wait** cycle to wait for I/O from device
- In Host is busy waiting/ polling:- It is in a loop, reading the status register over and over until the busy bit becomes clear.

2. Interrupts:-

- The CPU hardware has *wired* called the *interrupt-request line* that the CPU senses after executing every instruction
- interrupt handler:- receives interrupts
- interrupt controller:-
- nonmaskable interrupt :- Those interrupts which is reserved for events. (High priority)
  - Those are interrupts at the middle of execution.
  - Such as un recoverable errors
- maskable :- to ignore or delay some interrupts (low priority)
  - Those are (masked) *turned off by the CPU* before the execution of critical instruction sequences that must not be interrupted.
- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority:- High priority interrupts CPU handles without masking, Low priority interrupts makes possible to masking.
  - Some **nonmaskable**
- software interrupt (trap)
- Interrupt mechanism also used for exceptions

Interrupt driven IO cycle



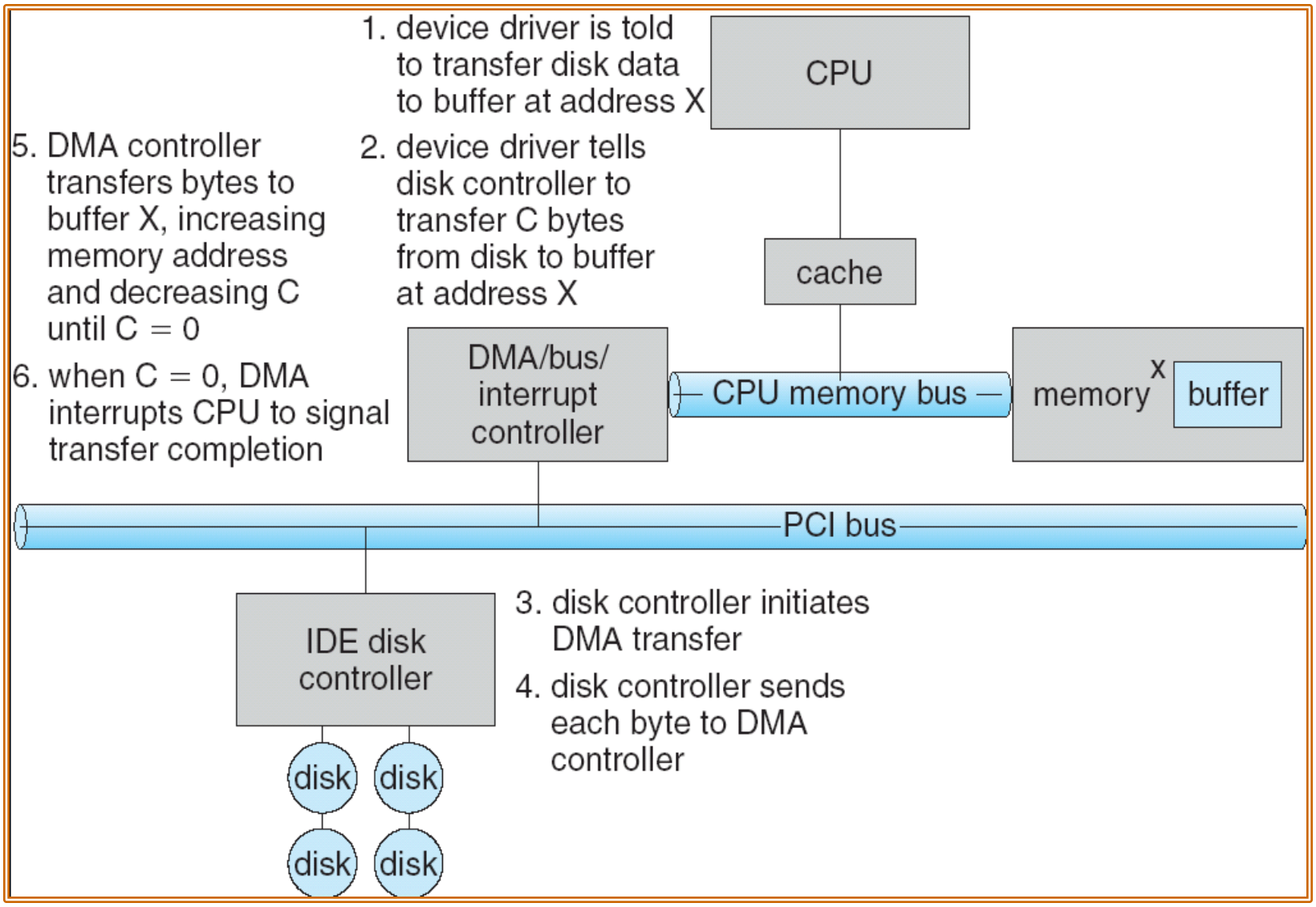
Intel Pentium Processor Event-Vector Table

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

3. Direct Memory Access (DMA):-

- Programmed I/O
  - Process is busy-waiting for the operation to complete
- Interrupt-driven I/O
  - I/O command is issued
  - Processor continues executing instructions
  - I/O module sends an interrupt when done
- Direct Memory Access (DMA)
  - DMA module controls exchange of data between main memory and the I/O device
  - Processor interrupted only after entire block has been transferred
- Direct Virtual memory access (DVMA)

Six Step Process to Perform DMA Transfer



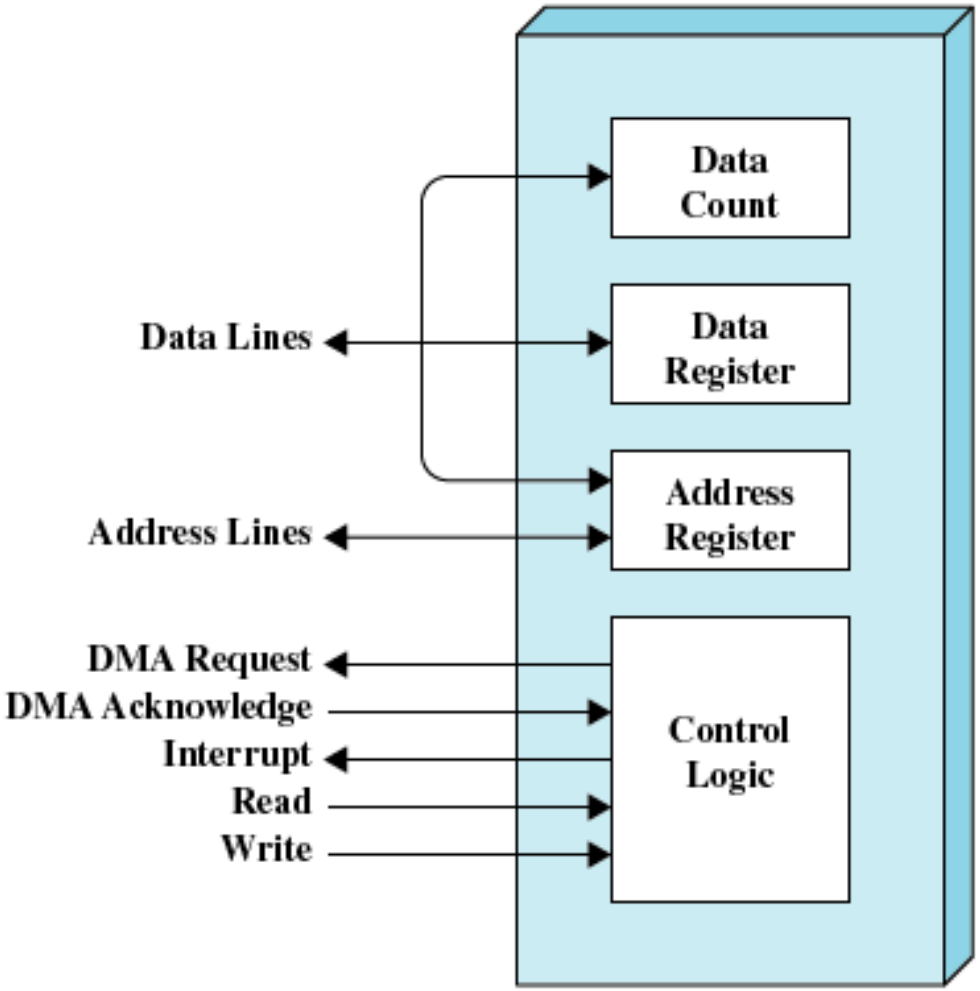


Figure 11.2 Typical DMA Block Diagram

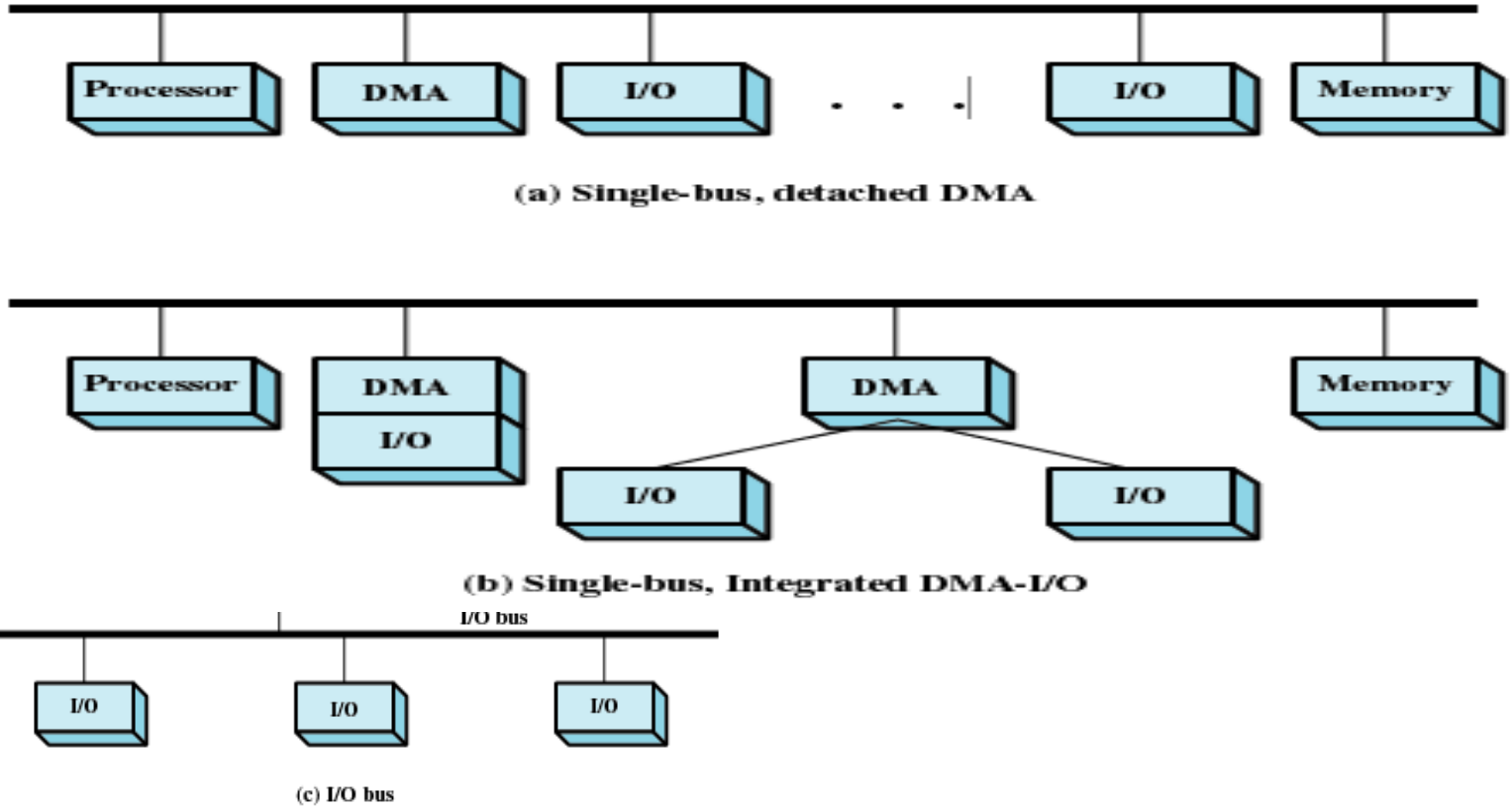
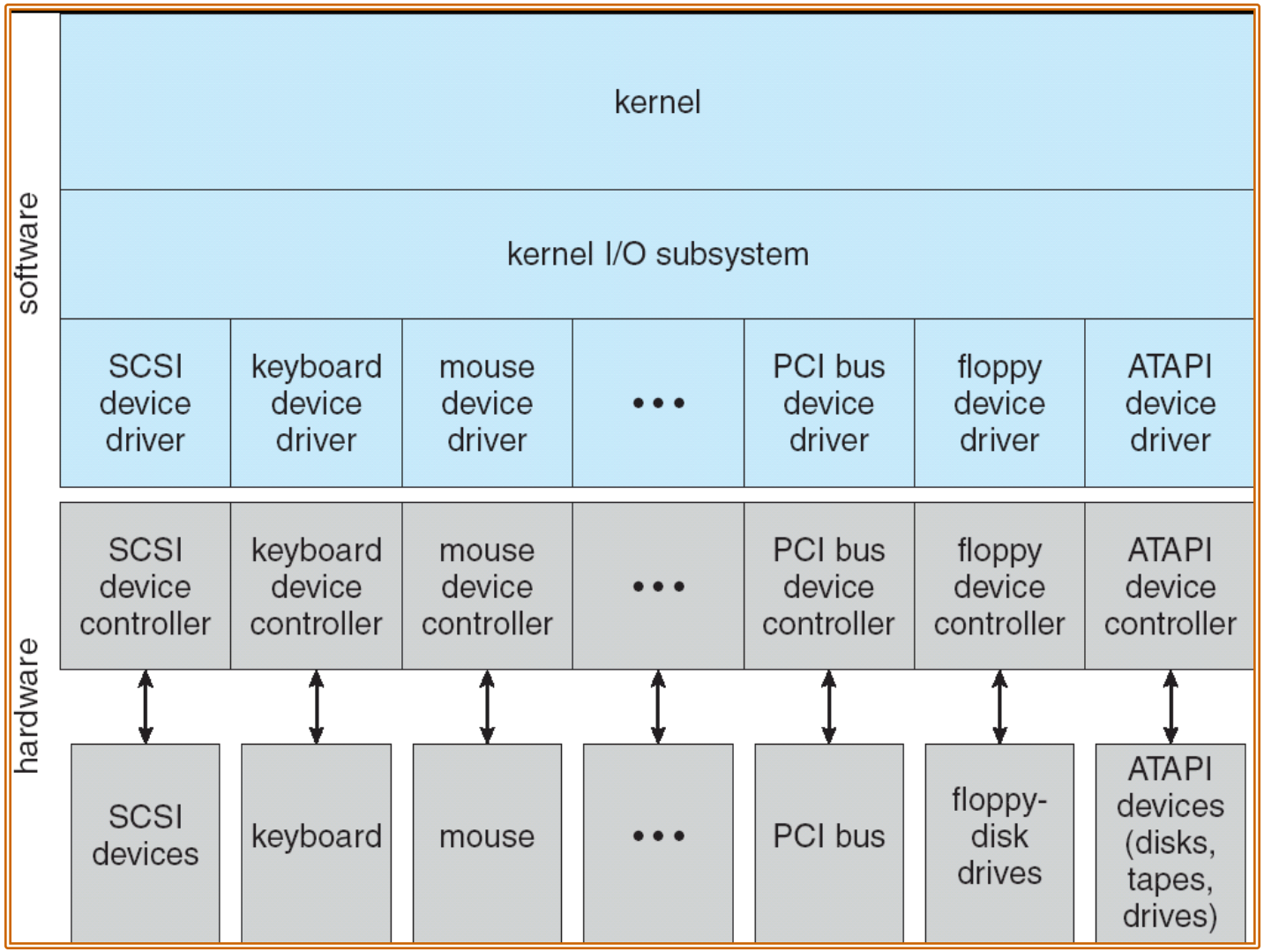


Figure 11.3 Alternative DMA Configurations

Application IO Interface

- 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:
    - Synchronous or asynchronous:
    - Sharable or dedicated:
    - Read-Write, read only, or write only:

A Kernel I/O Structure



Characteristics of IO 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 possible
- Character devices include keyboards, serial ports
  - Commands include *get(), put()*
  - Libraries layered on top allow line editing

Network Devices

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

Clocks and Timers

- Provide current time, elapsed time, timer
- **Programmable interval timer** used for timings, periodic interrupts
- ioctl (on UNIX) covers odd aspects of I/O such as clocks and timers

Blocking and Non-blocking I/O

- **Blocking** - process suspended until I/O completed
  - Easy to use and understand
  - Insufficient for some needs

- **Non-blocking** - 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 written

- **Asynchronous** - process runs while I/O executes
  - Difficult to use
  - I/O subsystem signals process when I/O completed

Kernel I/O Subsystem

1. I/O Scheduling
2. Buffering
3. Caching
4. Spooling and Device Reservation
5. Error Handling
6. Kernel Data Structures

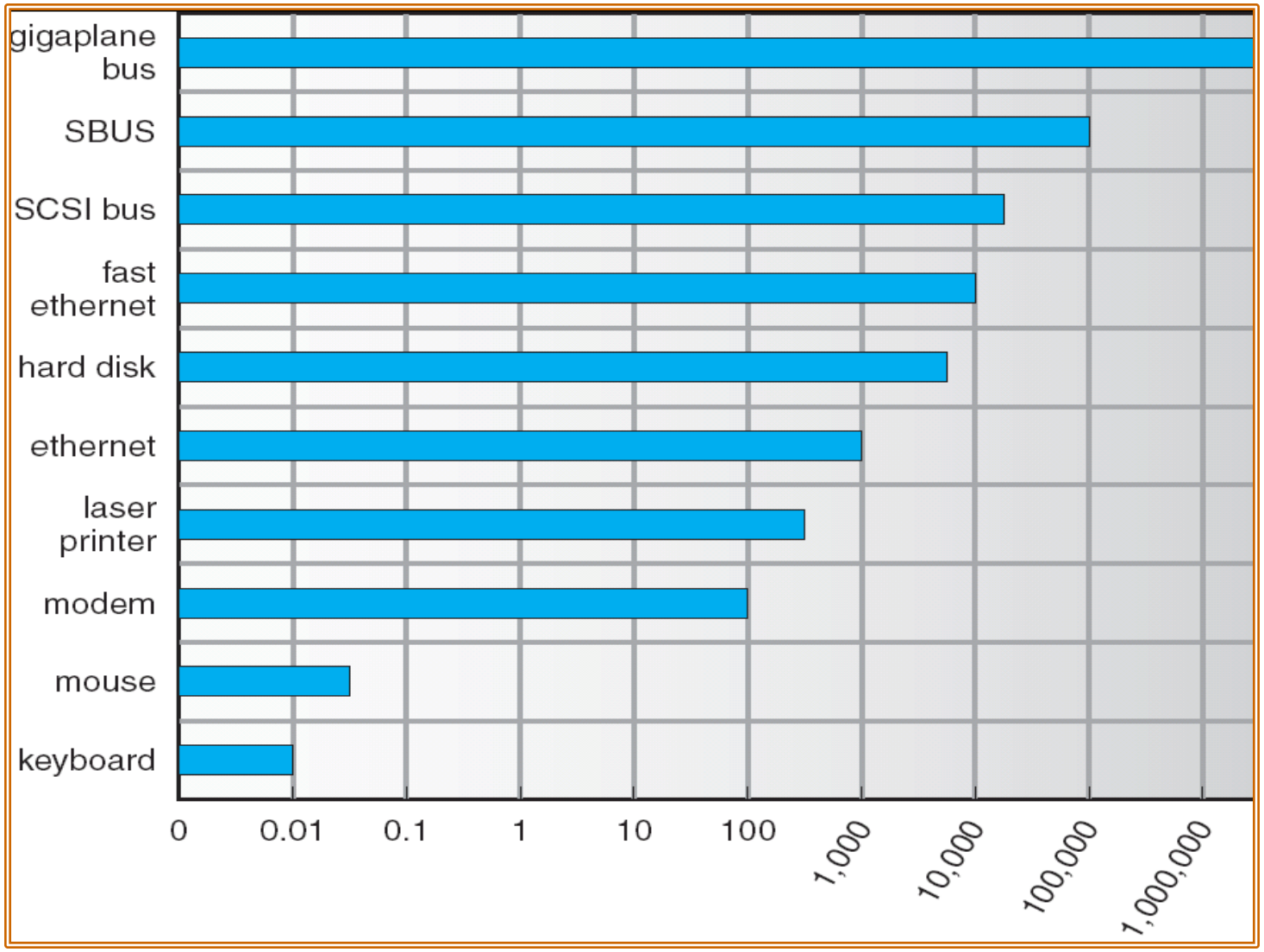
1. I/O Scheduling:-

- Some I/O request ordering via per-device queue
- Some OSs try fairness
- Schedule all the I/O devices

2. Buffering :-

- store data in memory while transferring between devices
  - 1.To cope with device speed mismatch:
    Ex:-modem to hard disk.
  - 2. To cope with device transfer size mismatch
    Ex:- Fragmentation and reassembling messages in the Network..
  - 3. To maintain “copy semantics”
    Ex:- Copy data from and write in to Hard disk.

Sun Enterprise 6000 Device-Transfer Rates



3. Caching :- fast memory holding copy of data

- Always just a copy
- Key to performance

4. Spooling :- hold output for a device

- If device can serve only one request at a time
- i.e., Printing

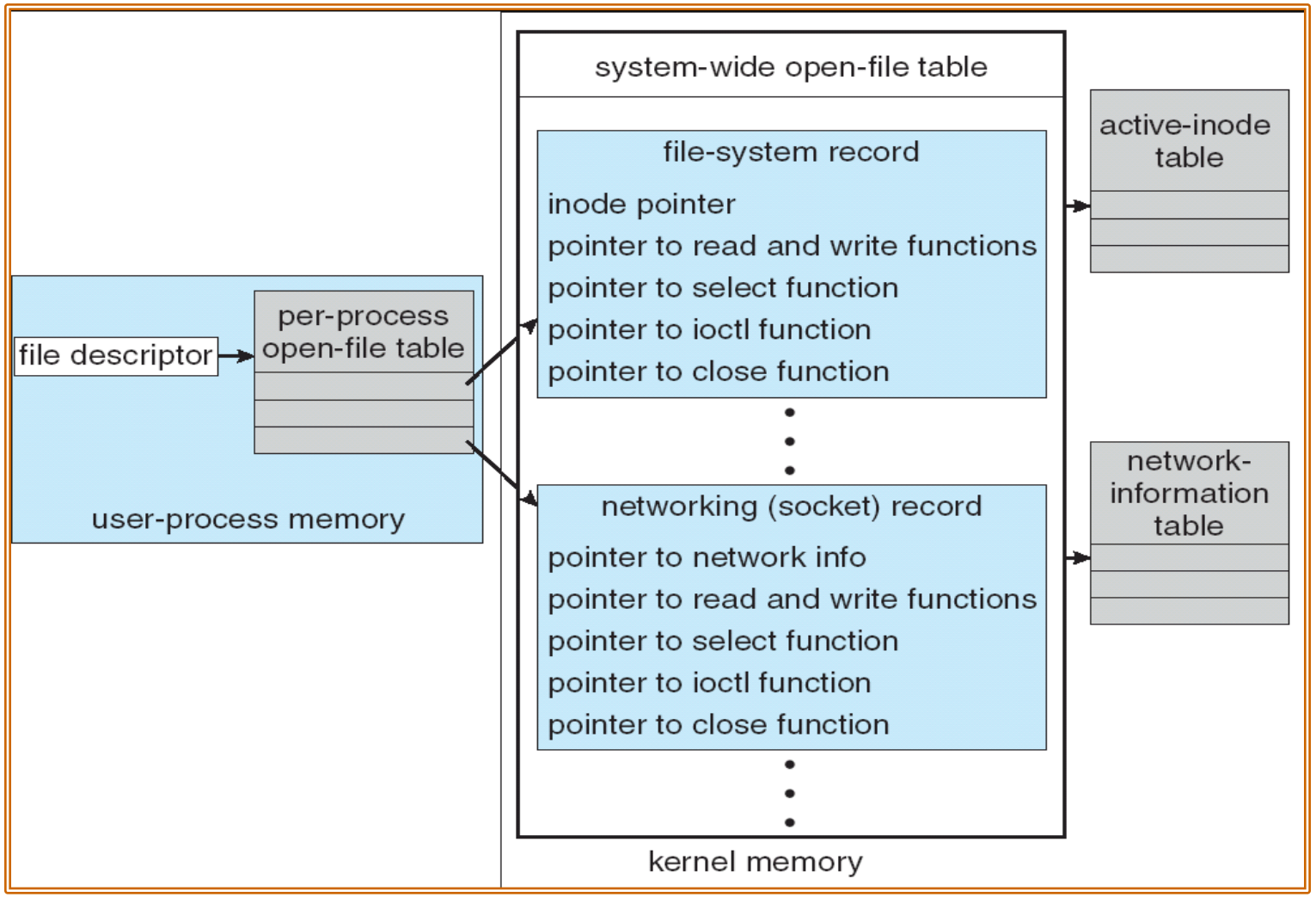
5. Device reservation :- provides exclusive access to a device

- System calls for allocation and deallocation
- Watch out for deadlock

6. Kernel Data Structures:-

- Kernel keeps state info for I/O components, including open file tables, network connections, character device state
- Many, many complex data structures to track buffers, memory allocation, “dirty” blocks
- Some use object-oriented methods and message passing to implement I/O

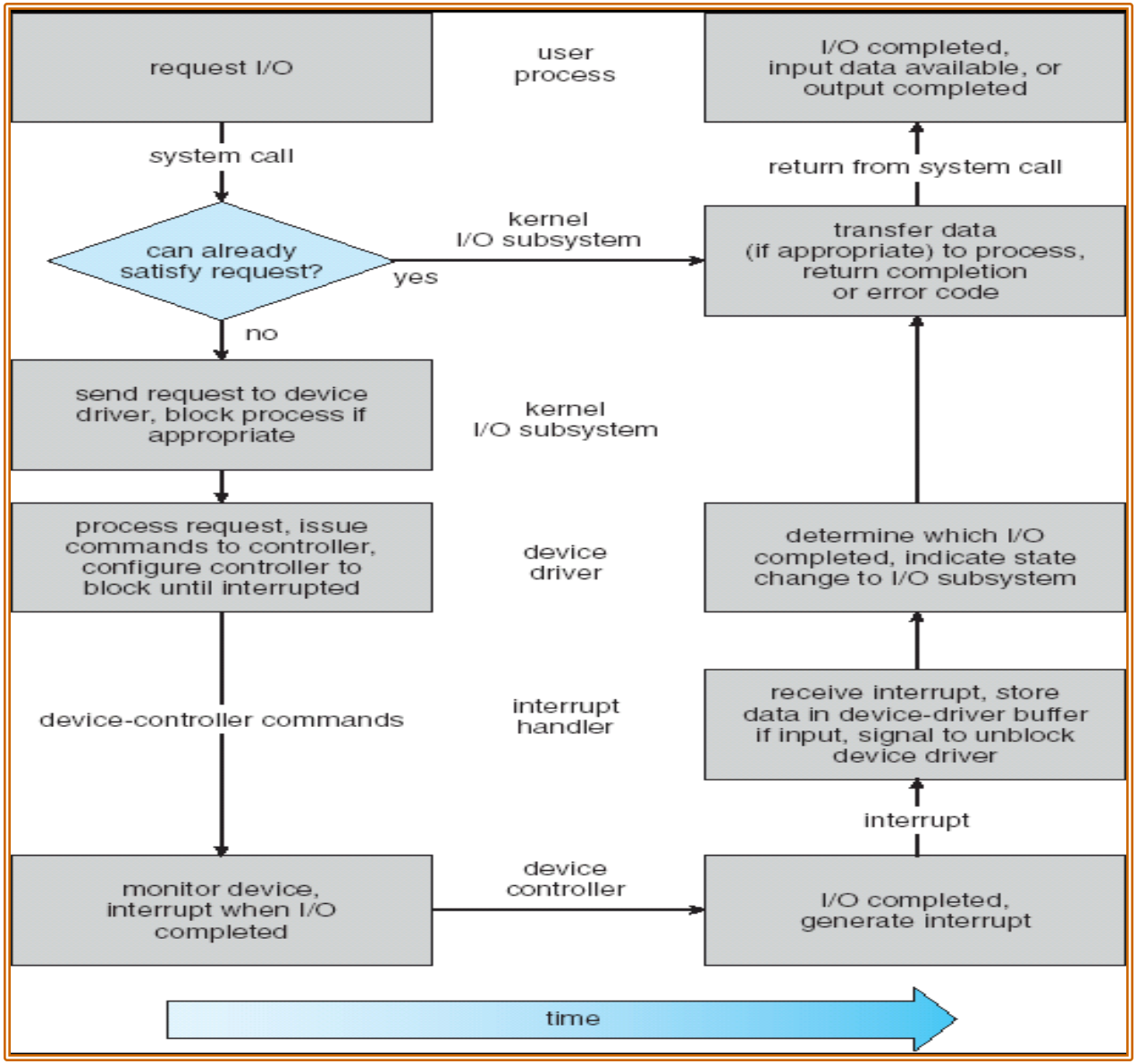
- # UNIX I/O Kernel Structure



Transforming IO Request to Hardware Operations

- Consider reading a file from disk for a process:
  - Determine device holding file
  - Translate name to device representation
  - Physically read data from disk into buffer
  - Make data available to requesting process
  - Return control to process

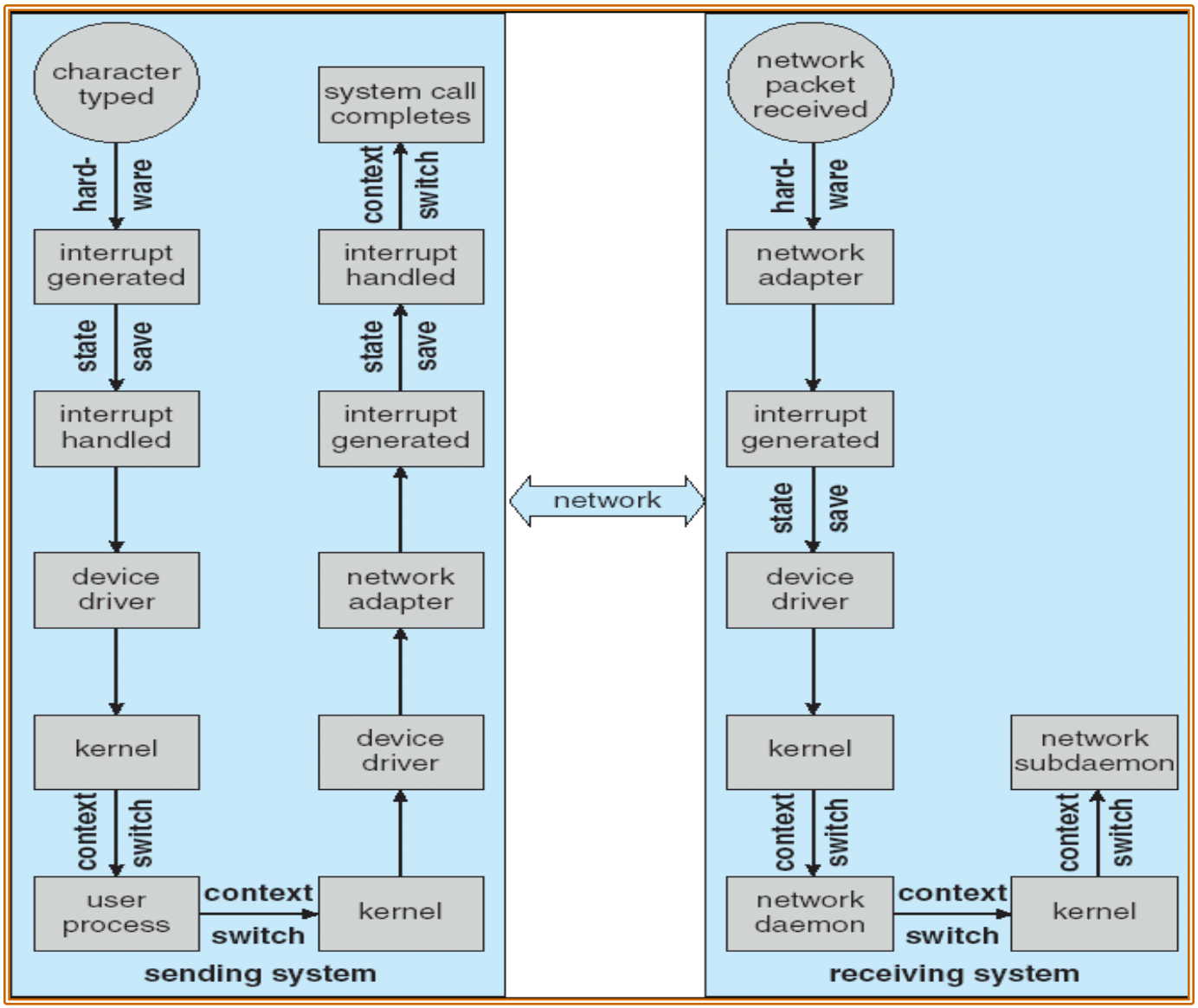
Life Cycle of An I/O Request



Performance

- I/O a major factor in system performance:-
  - Demands CPU to execute device driver code, kernel I/O code
  - The resulting Context switches stress the CPU and its hardware caches.
  - It exposes interrupt-handling mechanism in the kernel.
  - I/O loads memory bus during Data copy between controller and physical memory and between kernel buffer and application data space.
  - Network traffic can also cause a high context-switching rate. Example:- Remote login form one machine to another.

Inter-computer Communications



Improving Performance

- Reduce number of context switches
- Reduce data copying
- Reduce interrupts by using large transfers, smart controllers, polling
- Use DMA
- Balance CPU, memory, bus, and I/O performance for highest throughput

Device-Functionality Progression