Topics to be covered

- Principles of I/O Hardware: I/O devices
- Device controllers
- Direct memory access
- Principles of I/O Software: Goals of Interrupt handlers
- Device drivers
- Device independent I/O software
- Secondary-Storage Structure: Disk structure
- Disk scheduling algorithm

Components of I/O devices

- I/O devices have two components
  1. **Mechanical component**
  2. **Electronic component**
- The mechanical component is device itself.
- Electronic component of devices is called the **Device Controller**.

Device Controller

- **Electronic component** which **controls the device**.
- It may handle multiple devices.
- There **may be more than one controller** per mechanical component (example: hard drive).
- Controller's tasks are:
  - It **converts serial bit stream to block of bytes**
  - **Perform error correction** if necessary
  - **Block of bytes is** first **assembled bit by bit** in buffer inside the controller
  - After verification, the block has been declared to be error free, and then it can be **copied to main memory**

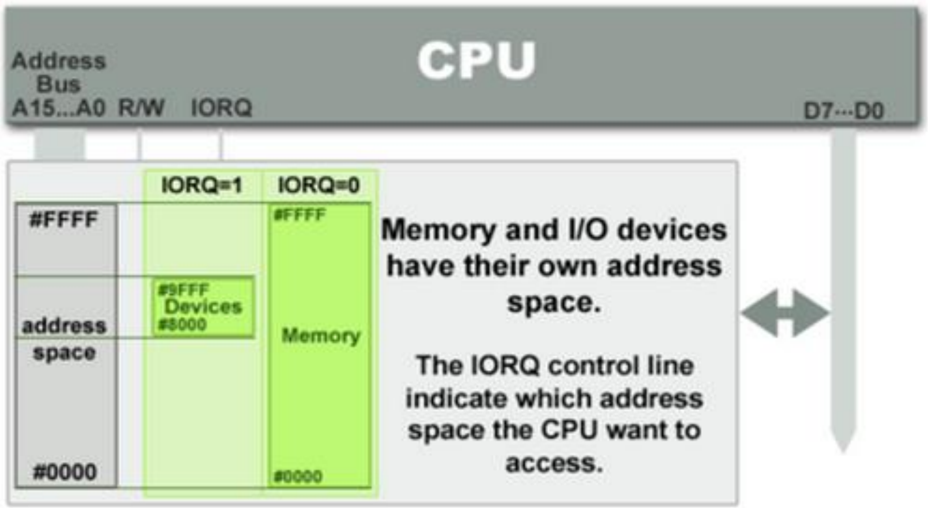
Memory-Mapped I/O

- Each device controller **has a few registers** that are **used for communicating with the CPU**.
- By **writing into these registers**, the OS can command the device to **deliver data, accept data, switch itself on or off**, or perform some action.
- By **reading from these registers** OS can learn **what the device's status is**, whether it is prepared to accept a new command and so on.
- There are two ways to communicate with control registers and the device buffers:
  1. I/O Port
  2. Memory mapped I/O

I/O Port Vs Memory Mapped I/O

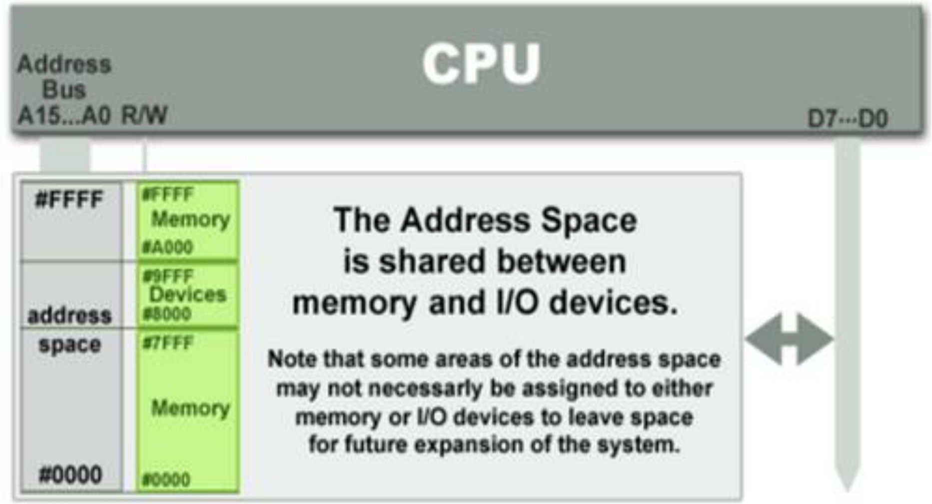
I/O Port

- **Uses different address spaces** for memory and I/O devices
- **Uses a special class** of CPU instructions to access I/O devices



Memory mapped I/O

- **Uses same address space** to address memory and I/O devices
- Access to the I/O devices using **regular instructions**



Direct Memory Access

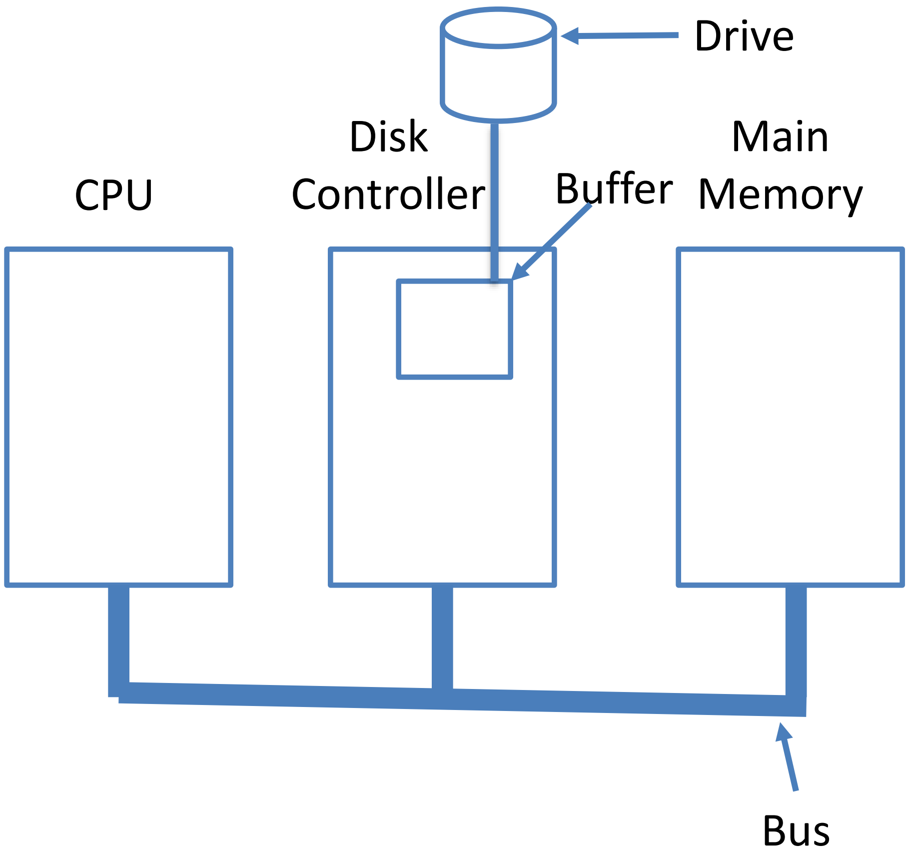
- Feature of computer systems that **allows certain hardware subsystems to access main memory (RAM), independent of the central processing unit (CPU)**.
- Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work.

Direct Memory Access

- With DMA, the **CPU first initiates the transfer, then it does other operations while the transfer is in progress**, and it finally receives an interrupt from the DMA controller when the operation is done.
- This feature is **useful when the CPU needs to perform useful work while waiting** for a relatively slow I/O data transfer.
- Many hardware systems such as disk drive controllers, graphics cards, network cards and sound cards use DMA.

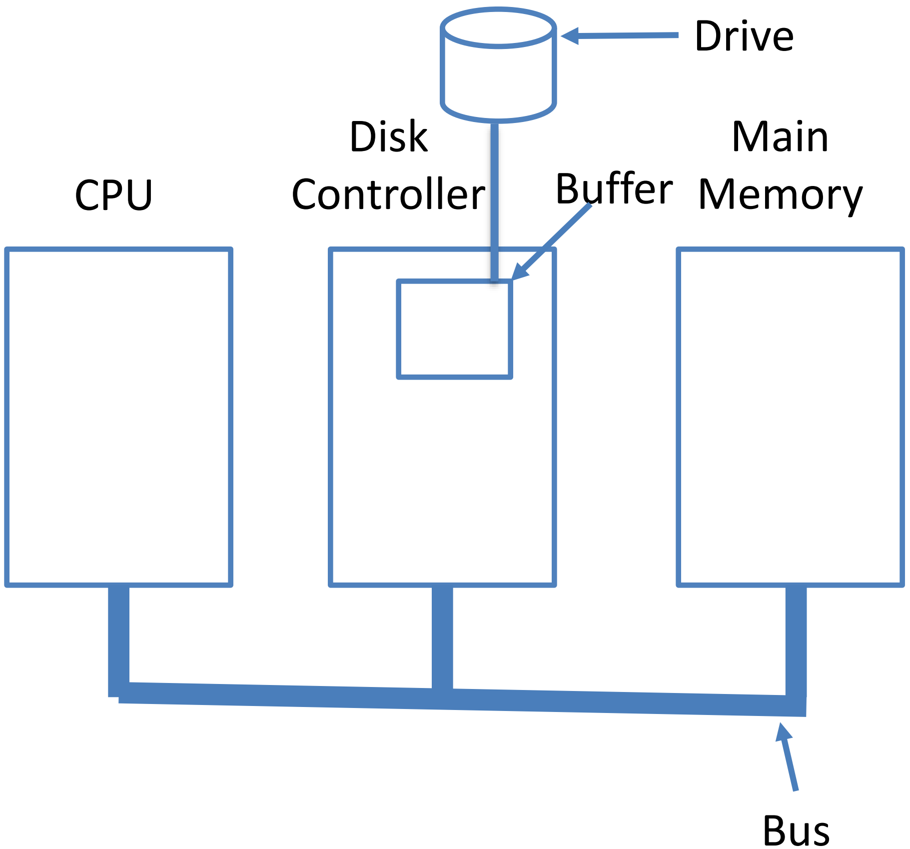
Disk read-write without a DMA

- The disk controller **reads the block from the drive serially**, **bit by bit**, until the entire block is in the controller's buffer.
- Next, it **computes the checksum** to verify that no read errors have occurred.

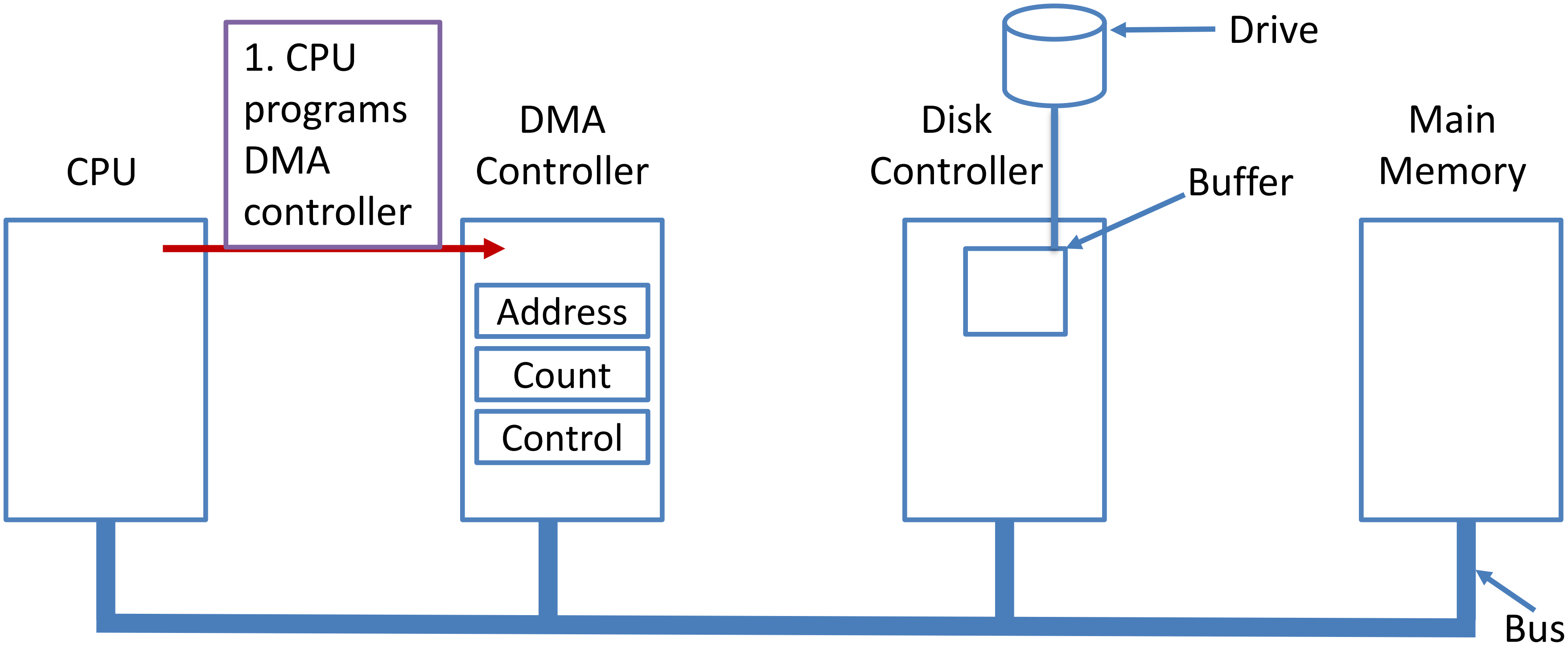


Disk read-write without a DMA

- Then the **controller causes an interrupt**, so that OS can read the block from controller's buffer (a byte or a word at a time) by executing a loop.
- After reading every single part of the block from controller device register, the operating system will store them into the main memory.



Disk read-write using DMA

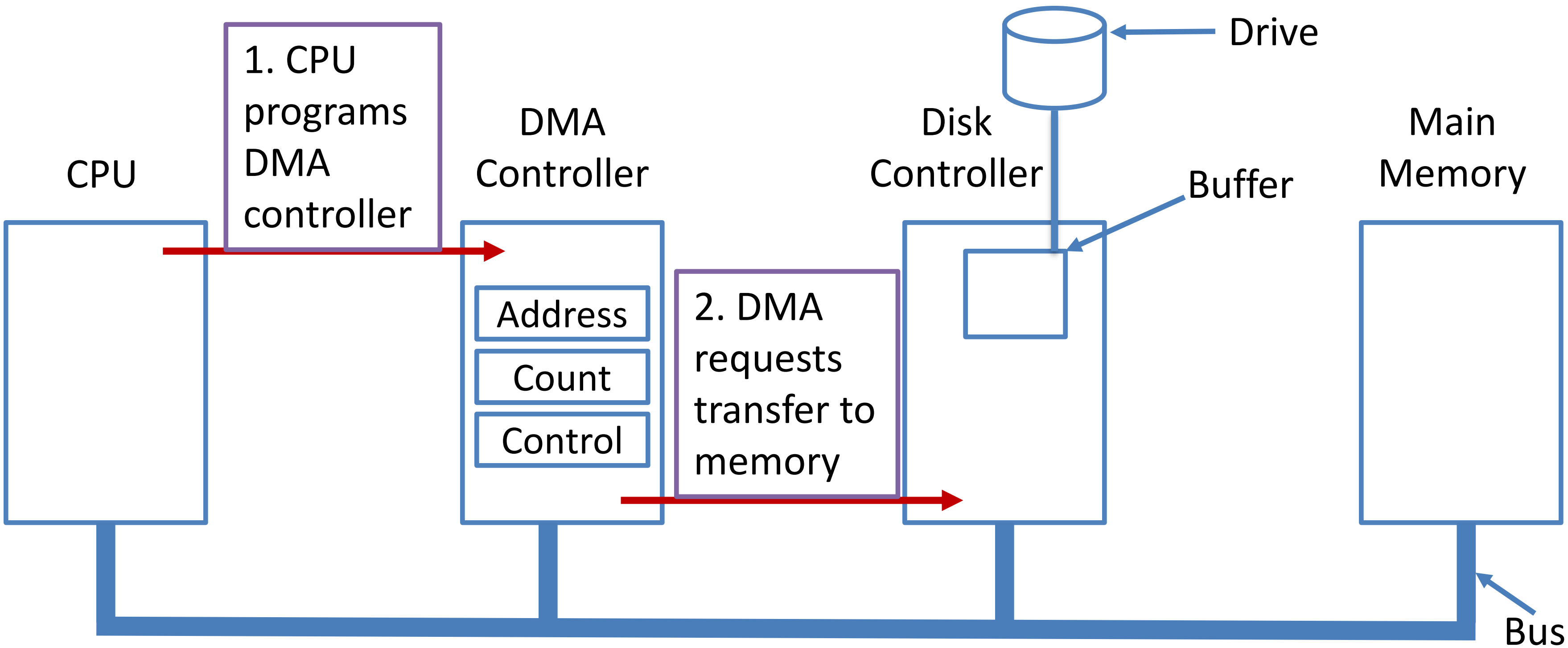


Step 1: First the CPU programs the DMA controller by setting its registers so it knows what to transfer where.

It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum.

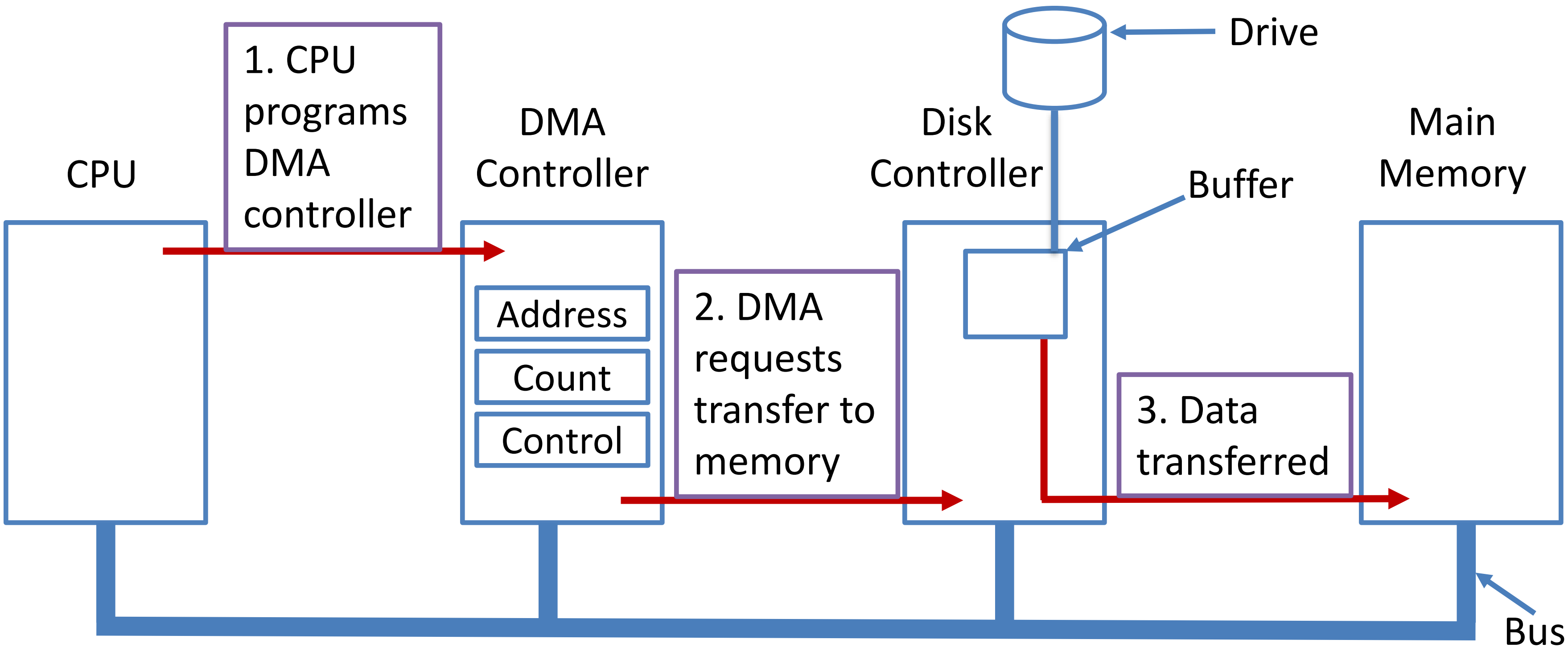
When valid data are in the disk controller's buffer, DMA can begin.

Disk read-write using DMA



Step 2: The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.
This read request looks like any other read request, and the disk controller does not know (or care) whether it came from the CPU or from a DMA controller.

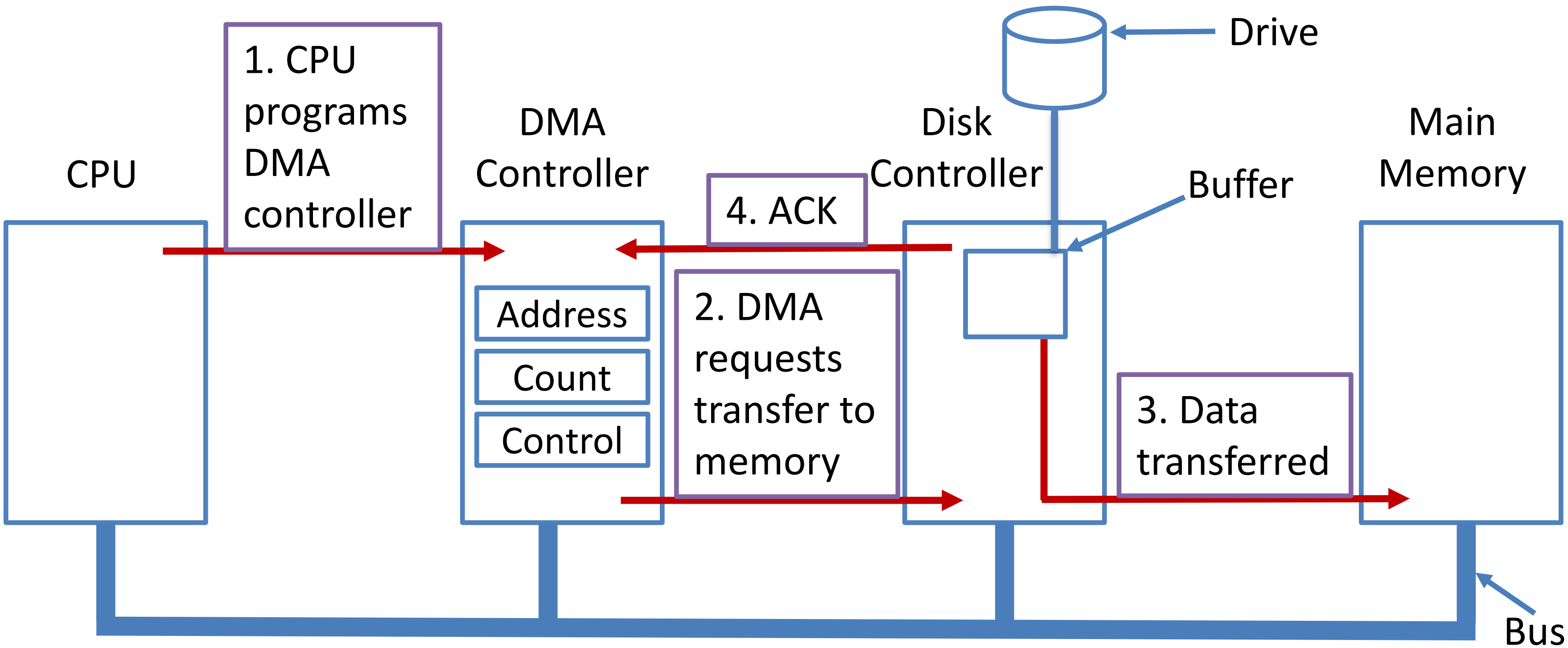
Disk read-write using DMA



Typically, the memory address to write to is on the bus' address lines, so when the disk controller fetches the next word from its internal buffer, it knows where to write it.

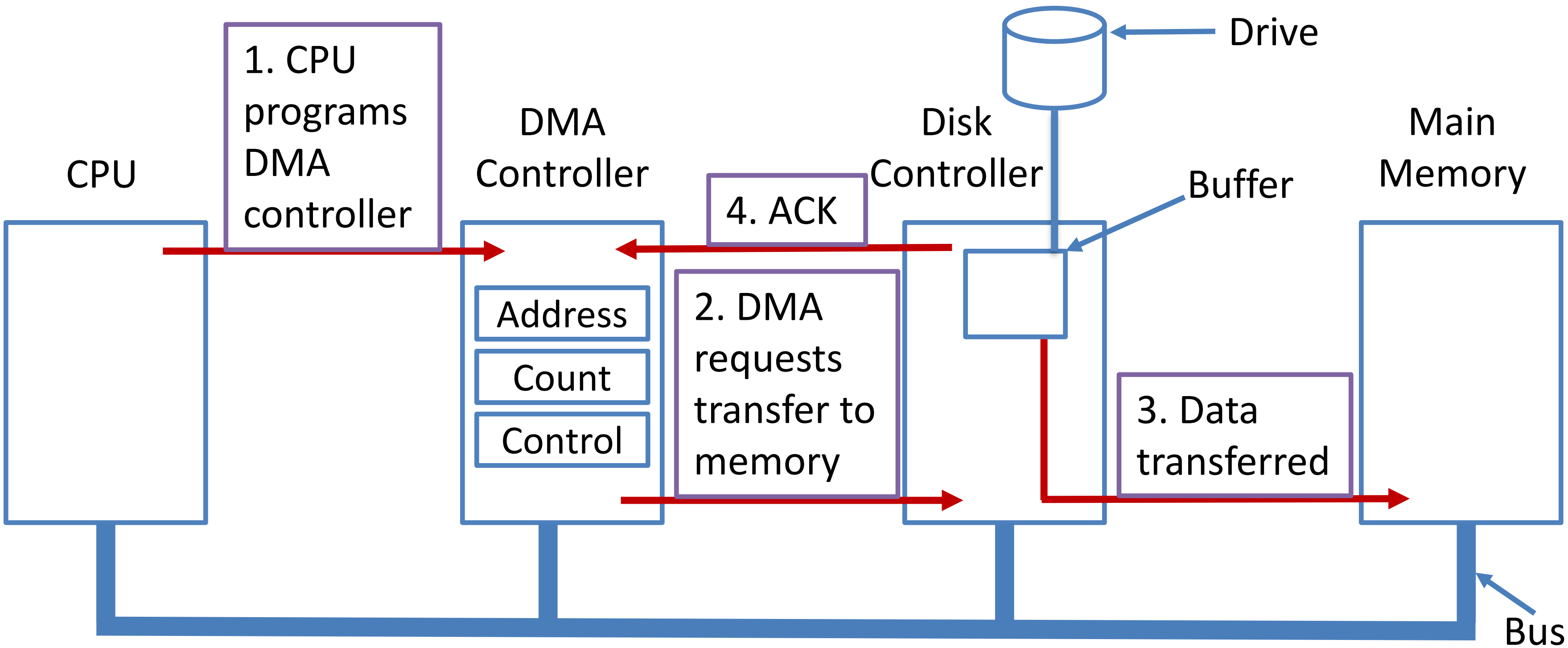
Step 3: The write to memory is another standard bus cycle.

Disk read-write using DMA



Step 4: When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus.
The DMA controller then increments the memory address to use and decrements the byte count.
If the byte count is still greater than 0, steps 2 to 4 are repeated until it reaches 0.

Disk read-write using DMA



At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete.
When the OS starts up, it does not have to copy the disk block to memory; it is already there.

Modes of bus operation

- The buses can be operated in two modes

1. **Word-at-a-time mode**: Here the DMA requests for the **transfer of one word** and gets it.
   - If CPU wants the bus at same time then it has to wait.
   - This mechanism is known as Cycle.
2. **Block mode**: Here the DMA controller tells the device to acquire the bus, issues a **series of transfer** and then releases the bus.
   - This form of the operation is called Burst mode.
   - It is more efficient then cycle stealing.

Goals of I/O Software

1. Device independence
    - It should be possible to write **programs that can access any I/O devices** without having to specify device in advance.
    - For example, a program that reads a file as input should be able to read a file on a floppy disk, on a hard disk, or on a CD-ROM, without having to modify the program for each different device.
2. Uniform naming
    - Name of file or device should be some specific string or number. It **must not depend upon device** in any way.
    - All files and devices are addressed the same way: by a path name.

Goals of I/O Software

3. Error handling
   - **Error should be handled as close to hardware** as possible.
   - If any controller generates error then it tries to solve that error itself. If controller can’t solve that error then device driver should handle that error, perhaps by reading all blocks again.
   - Many times when error occurs, error is solved in lower layer. If lower layer is not able to handle error then problem should be told to upper layer.
   - In many cases error recovery can be done at a lower layer without the upper layers even knowing about error.

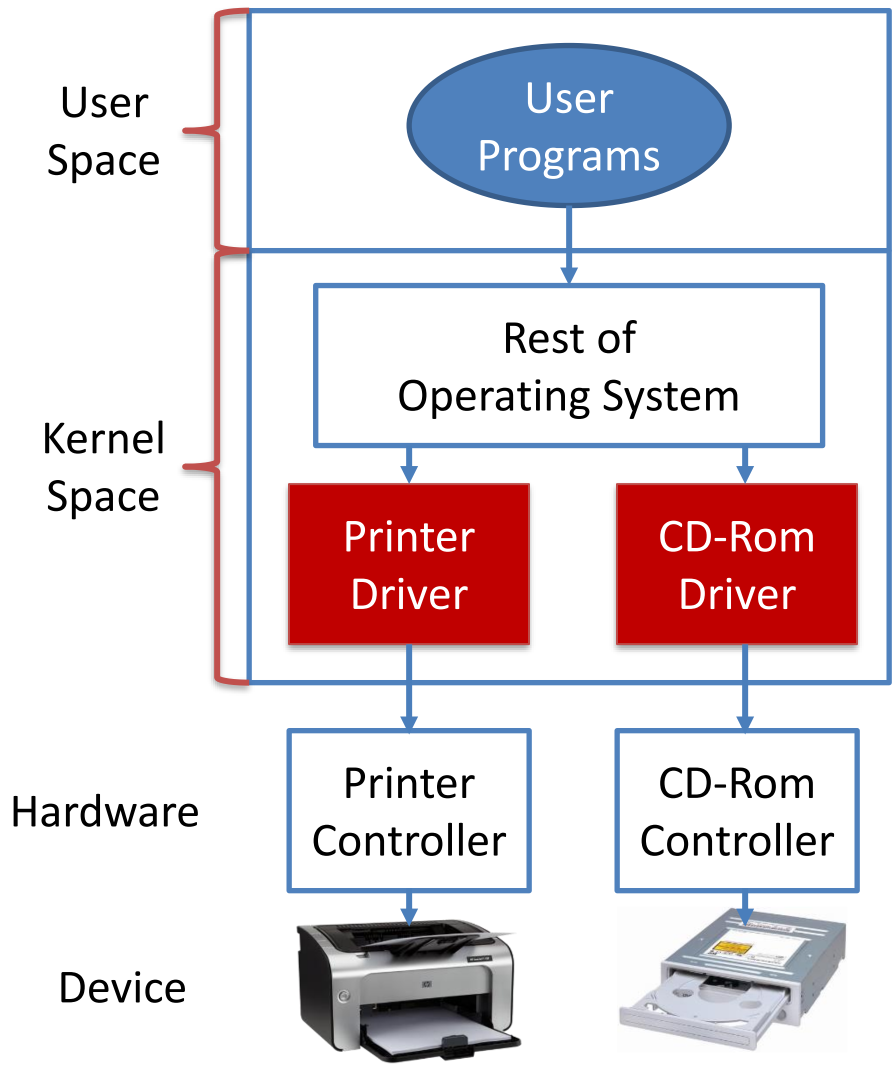
Goals of I/O Software

4. Synchronous vs. asynchronous transfers
   - Most of devices are asynchronous device. CPU starts transfer and goes off to do something else until interrupt occurs.
   - I/O Software **needs to support both the types of devices**.
5. Buffering
   - **Data comes in main memory cannot be stored directly**.
   - For example data packets come from the network cannot be directly stored in physical memory.
   - Packets have to be put into output buffer for examining them.
   - Some devices have several real-time constraints, so data must be put into output buffer in advance to decouple the rate at which buffer is filled and the rate at which it is emptied, in order to avoid buffer under runs.

Device driver

- I/O devices which are plugged with computer have **some specific code for controlling them**. This code is called the **device driver**.
- Each device driver normally handles one device type, or at most one class of closely related devices.
- Generally device driver is delivered along with the device by device manufacturer.
- Device drivers are normally positioned below the rest of Operating System.

Logical positioning of device drivers



Functions of device drivers

1. Device driver **accept abstract read and write requests** from device independent software.
2. Device driver must **initialize the device** if needed.
3. It also **controls power requirement** and **log event**.
4. It also **checks statues of devices**. If it is currently in use then queue the request for latter processing. If device is in idle state then request can be handled now.
5. Pluggable device can be added or removed while the computer is running. At that time the device driver **inform CPU** that the user has suddenly removed the device from system.

Device Independent I/O Software

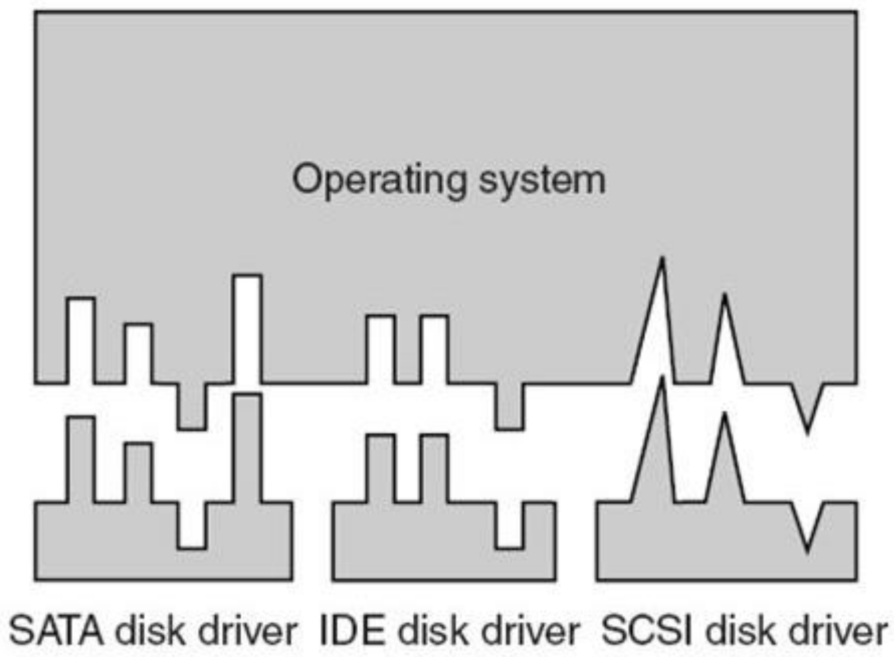
- Exact boundary between the drivers and the device independent I/O software is system dependent.
- Function of device independent I/O Software
  1. Uniform interfacing for device drivers.
  2. Buffering.
  3. Error Reporting.
  4. Allocating and releasing dedicated devices.
  5. Providing a device-independent block size.

Uniform interfacing for device drivers

- A major issue of an OS is how to make all I/O devices and drivers look more or less the same.
- One aspect of this issue is the **interface between the device drivers and the rest of the OS**.

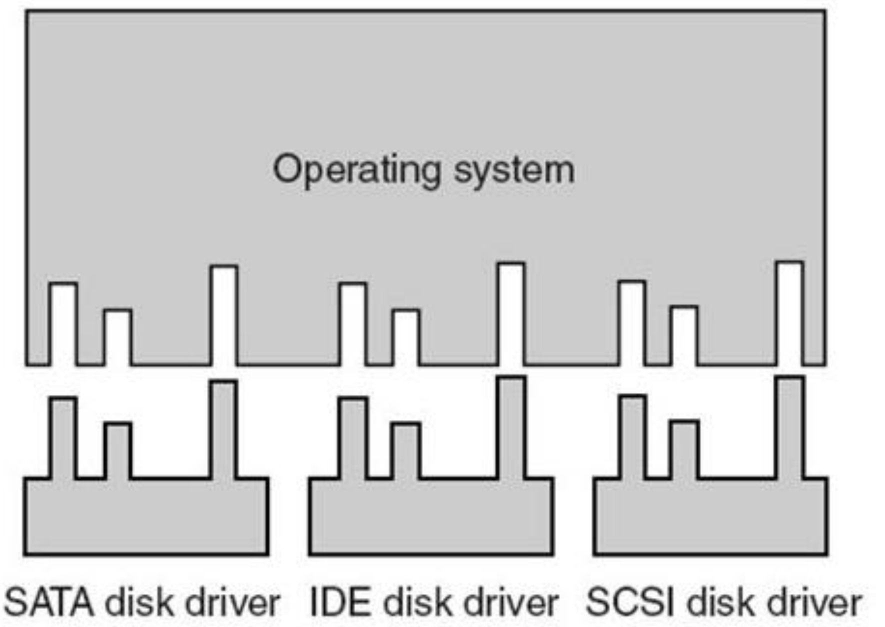
Uniform interfacing for device drivers

- Figure shows situation in which **each device driver has a different interface to OS**, it means that interfacing each new driver requires a lot of new programming effort.



Uniform interfacing for device drivers

- Figure shows a different design in which **all drivers have the same interface**.
- Now it becomes much easier to plug in a new driver.

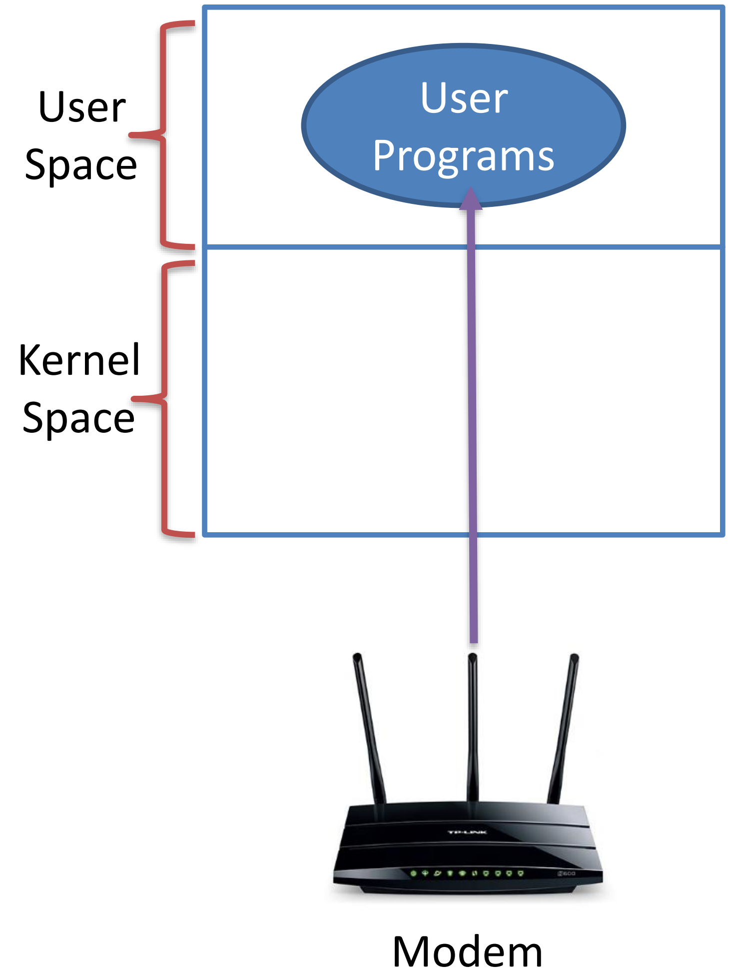


Buffering

- Buffering is also issue, both for block and character devices.
- In case of a process, which reads the data from the modem, **without buffering the user process has to be started up for every incoming character**.
- We have four different options:
  1. Unbuffered input
  2. Buffering in user space
  3. Buffering in kernel followed by user space
  4. Double buffering in kernel space

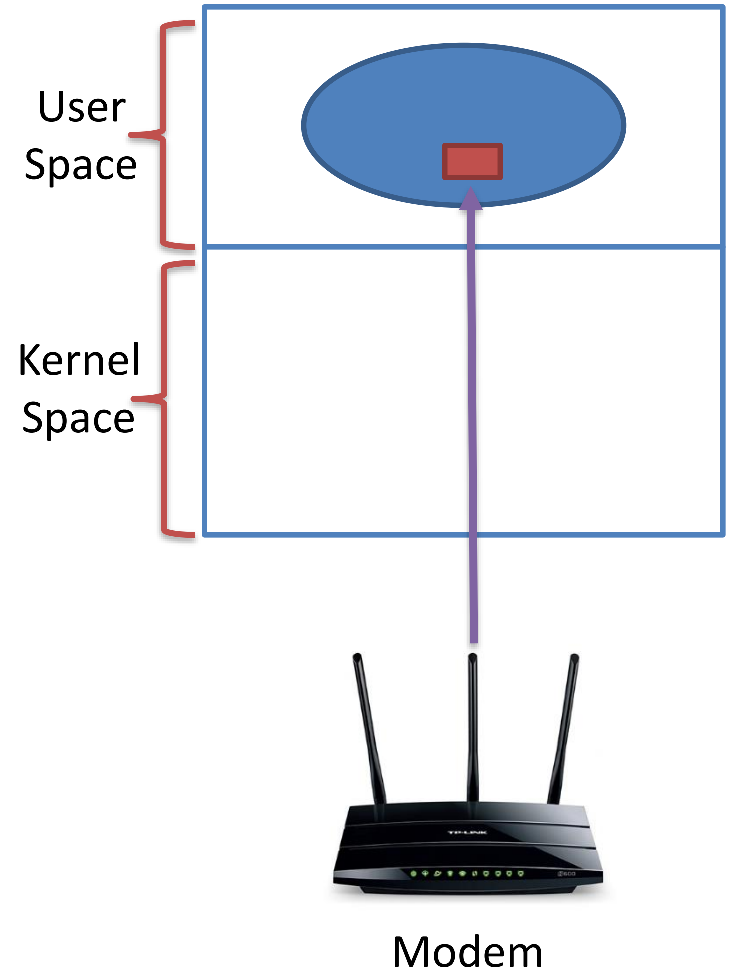
Unbuffered input

- No buffer available in user space or kernel space.
- Allowing a process to run many times for short runs is inefficient, so this design is not a good one.



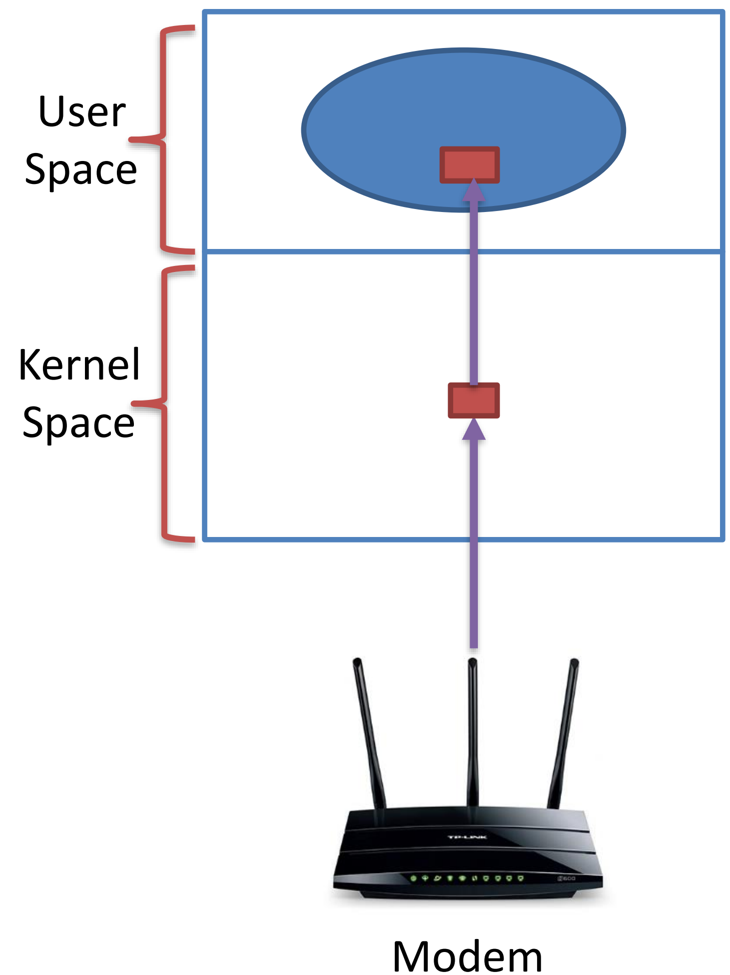
Buffering in user space

- Buffer in users pace: here user process provides an n-character buffer in user space and does a read of n-characters.



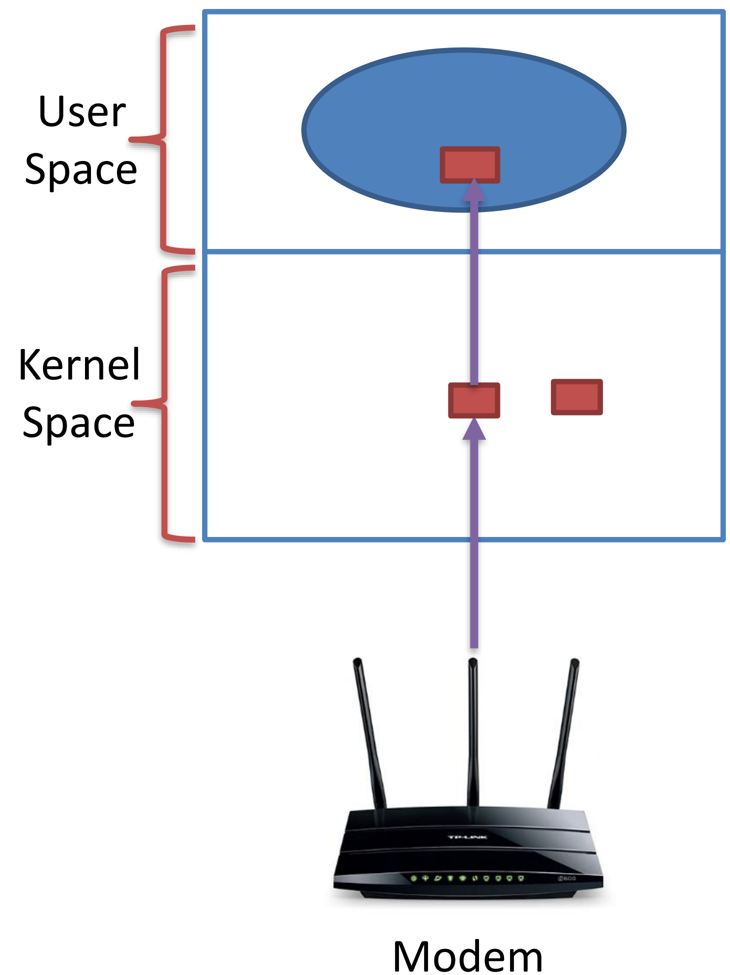
Buffering in kernel space followed by user space

- Buffer inside kernel: to create the buffer inside the kernel and interrupt handler is responsible to put the character there.



Double buffering in kernel space

- Two buffers in kernel: the first buffer is used to store characters. When it is full, it is being copied to user space. During that time the second buffer is used.
- In this way, two buffers take turns.
- This is called double buffering scheme.



Error Reporting

- Errors are far more common in the context of I/O than in other context. When they occur, the OS must handle them as best it can.
- One class of I/O errors is programming errors. These occur when a process asks for something impossible, such as writing to an input device or reading from an output device.
- The action taken for these errors is, to report an error code back to the caller.
- Another class of error is the class of actual I/O errors, for example trying to write a disk block that has been damaged.
- In this case, **driver determines what to do** and **if it does not know the solution then the problem may be passed to the device independent software**.

Allocating and releasing dedicated devices

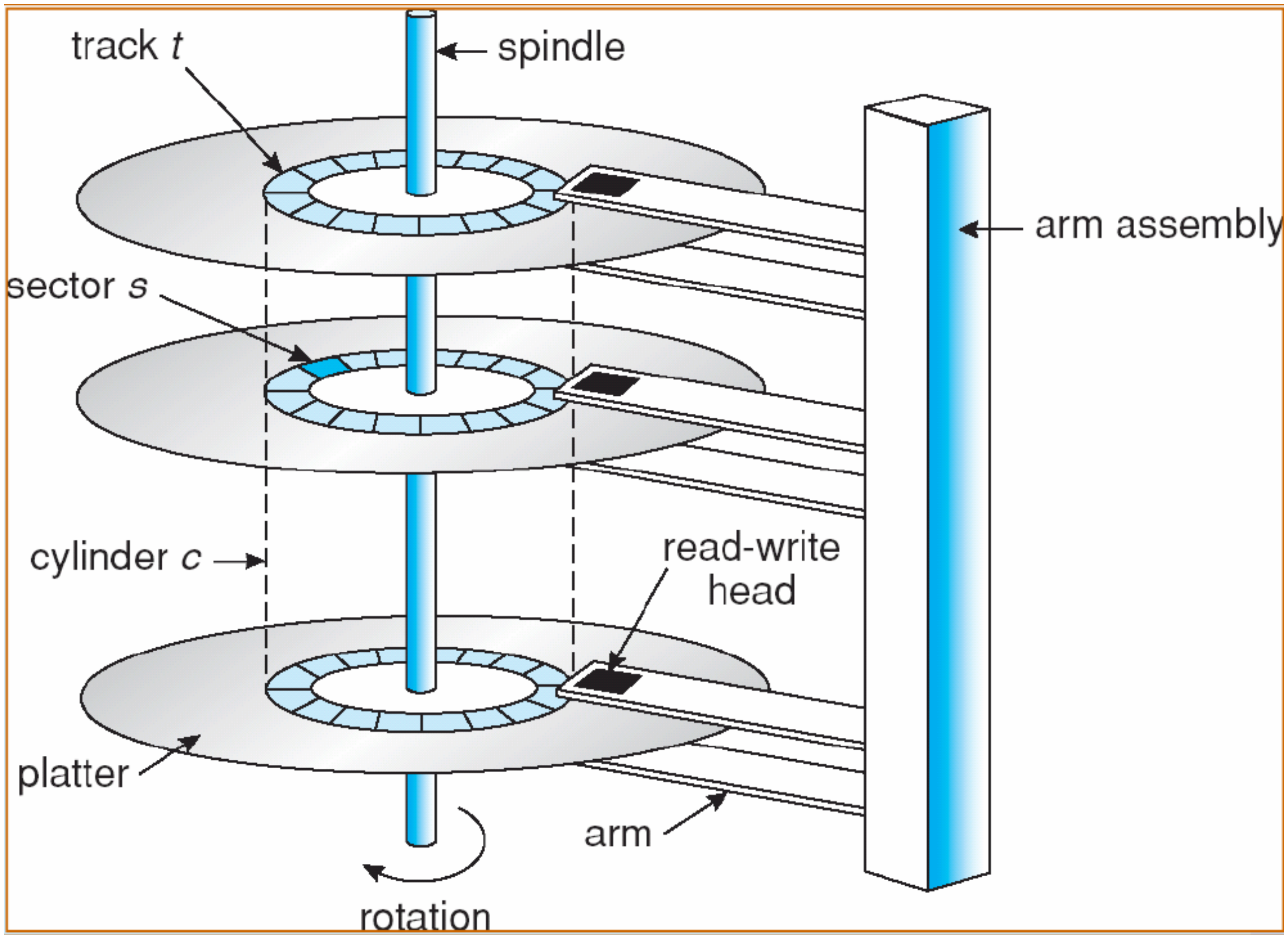
- Some devices such as CD-ROM recorders can be used only by a single process at any given moment.
- A mechanism for requesting and releasing dedicated devices is required.
- An attempt to acquire a device that is not available blocks the caller instead of failing.
- Blocked processes are put on a queue, sooner or later the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

Providing a device-independent block size

- Different disks may have different sector sizes.
- It is up to the device independent I/O software to hide this fact and provide a uniform block size to higher layers.

Definitions

- Seek time: The time to move the arm to the proper cylinder.
- Rotational delay: The time for the proper sector to rotate under the head.



Disk Arm Scheduling Algorithm

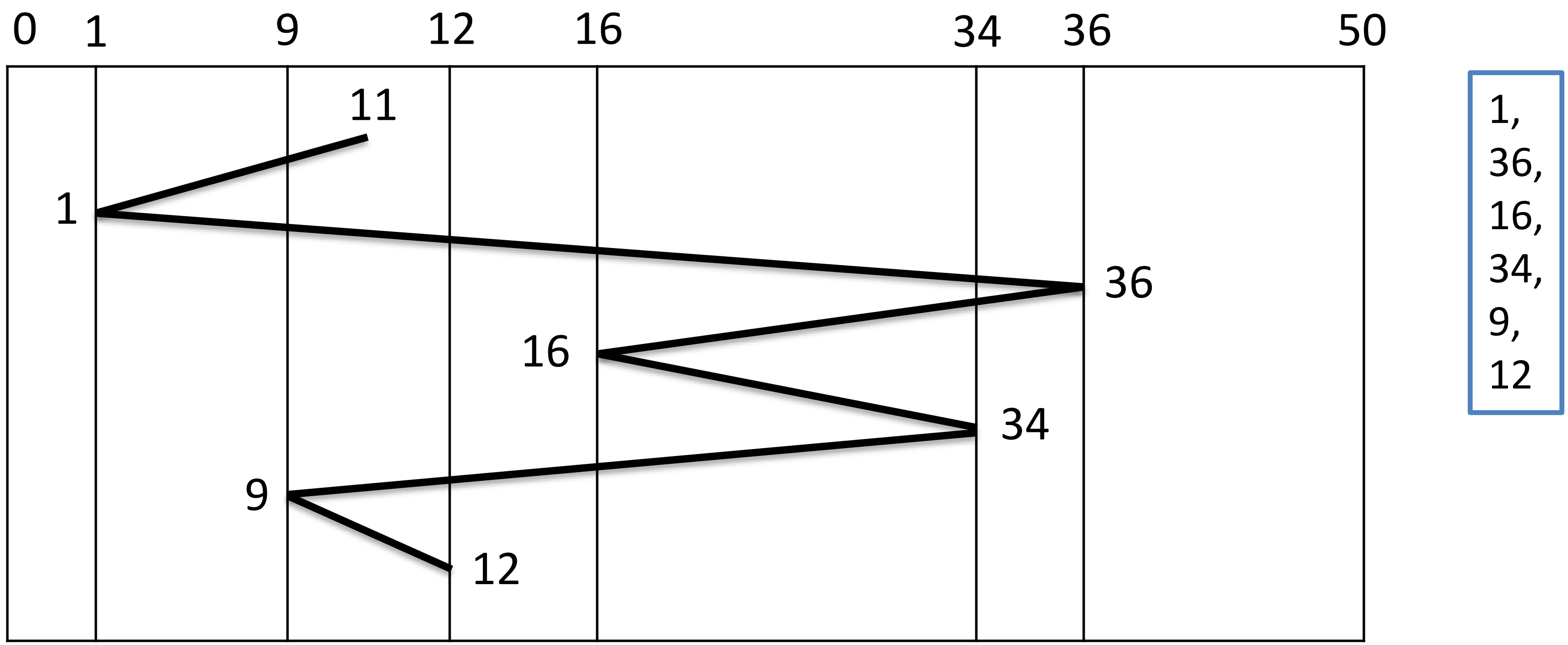
- Various types of disk arm scheduling algorithms are available to decrease mean seek time.
    1. FCFS (First come first serve)
    2. SSTF (Shorted seek time first)
    3. SCAN
    4. C-SCAN
    5. LOOK (Elevator)
    6. C-LOOK

Example for Disk Arm Scheduling Algorithm

- Consider an imaginary disk with 51 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order.
- Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests, for each of the following disk scheduling Algorithms?

FCFS (First come first serve)

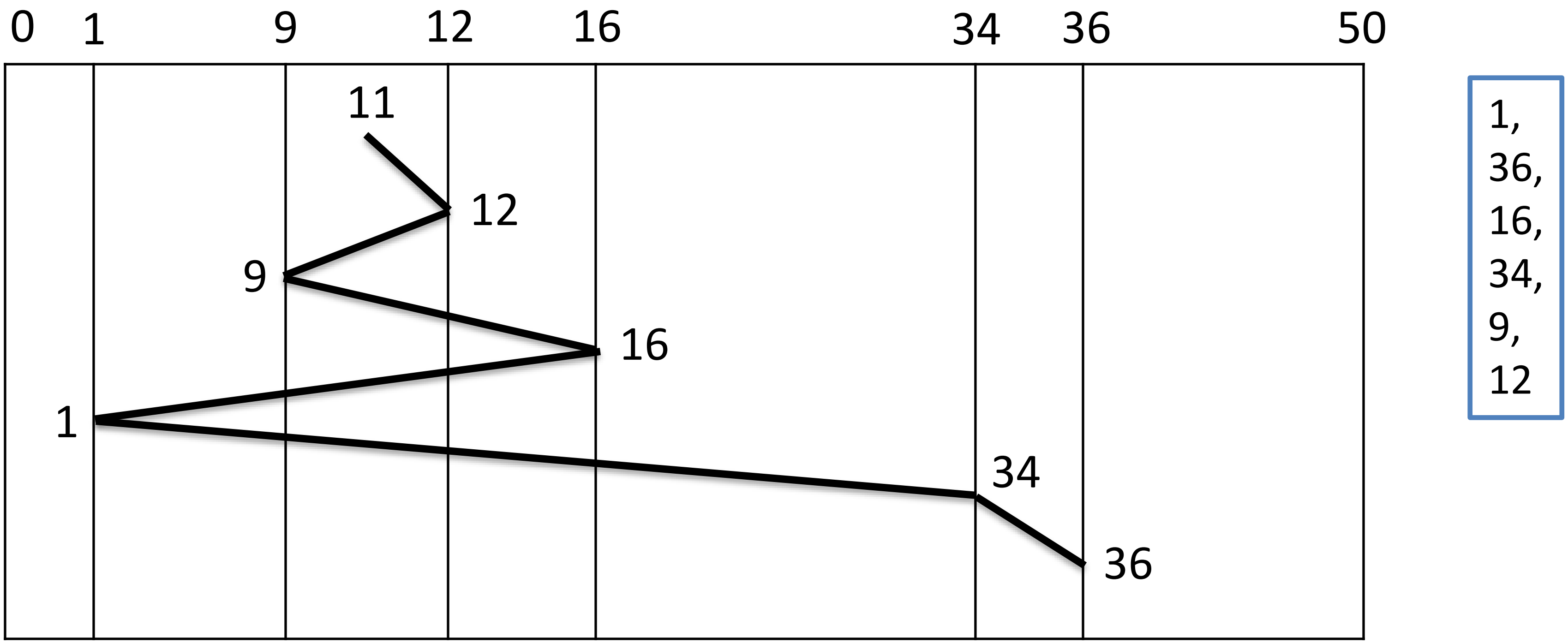
- Here **requests are served in the order of their arrival**.



- Disk movement will be 11, 1, 36, 16, 34, 9 and 12.
- Total cylinder movement: (11-1) + (36-1) + (36-16) + (34-16) + (34-9) + (12-9) = 111

SSTF (Shortest seek time first)

- We can minimize the disk movement by **serving the request closest to the current position** of the head.

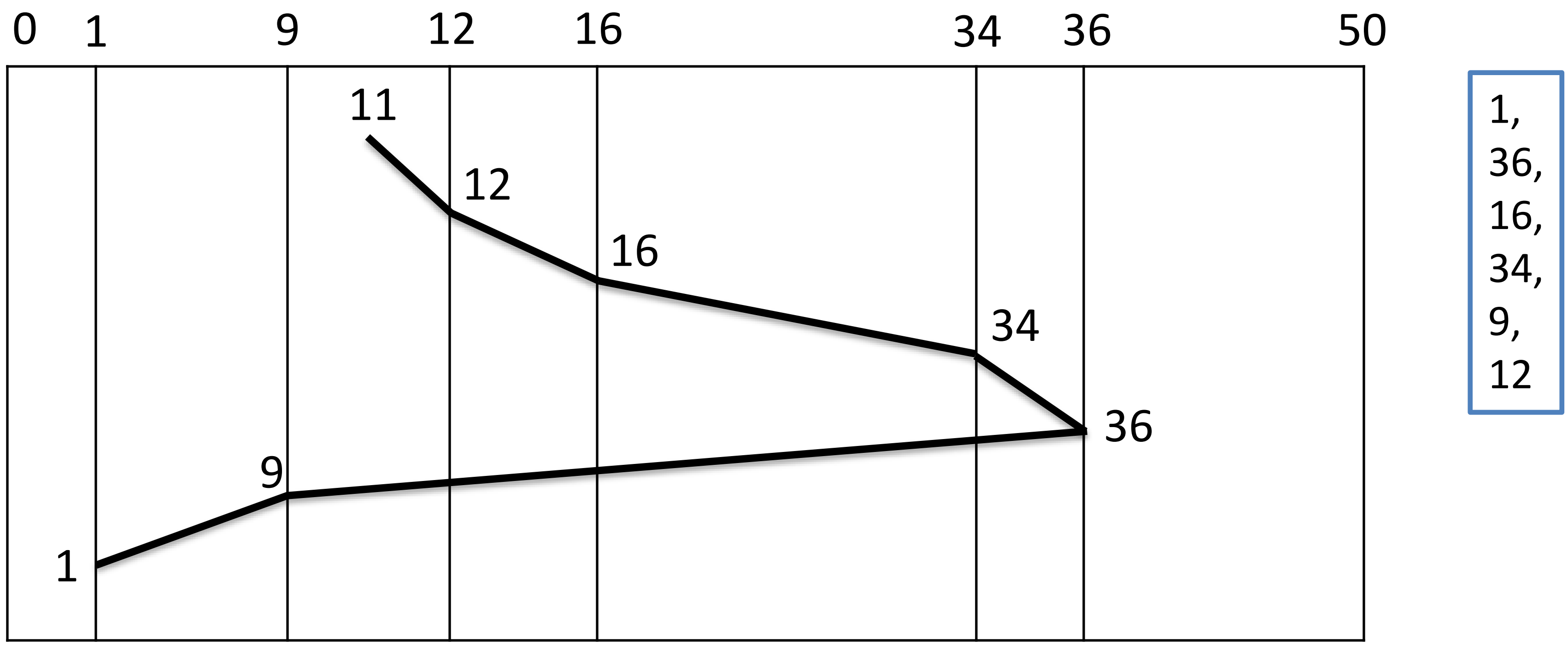


- Disk movement will be 11, 12, 9, 16, 1, 34, 36.
- Total cylinder movement: (12-11) + (12-9) + (16-9) + (16-1) + (34-1) + (36-34) = 61

LOOK (Elevator)

- **Keep moving in the same direction until there are no more outstanding requests pending in that direction, then algorithm switches the direction**.
- After switching the direction the arm will move to handle any request on the way. Here **first go it moves in up direction then goes in down direction**.
- This is also called as **elevator algorithm**.
- In the elevator algorithm, the **software maintains 1 bit: the current direction bit, which takes the value either UP or DOWN**.

LOOK (Elevator)

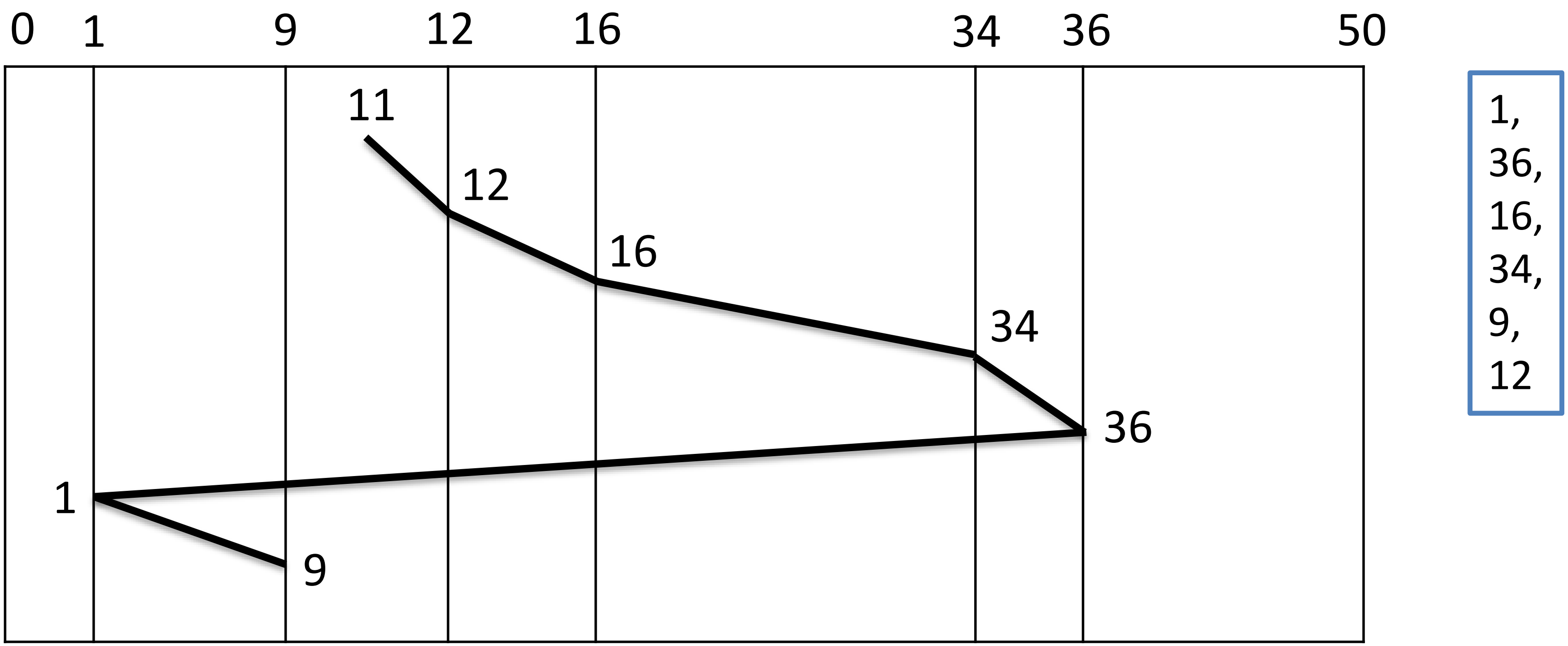


- Disk movement will be 11, 12, 16, 34, 36, 9, 1.
- Total cylinder movement: (12-11) + (16-12) + (34-16) + (36-34) + (36-9) + (9-1)=60

C-LOOK

- **Keep moving in the same direction until there are no more outstanding requests pending in that direction, then algorithm switches direction**.
- **When switching occurs the arm goes to the lowest numbered cylinder** with pending requests and **from there it continues moving in upward direction again**.

C-LOOK

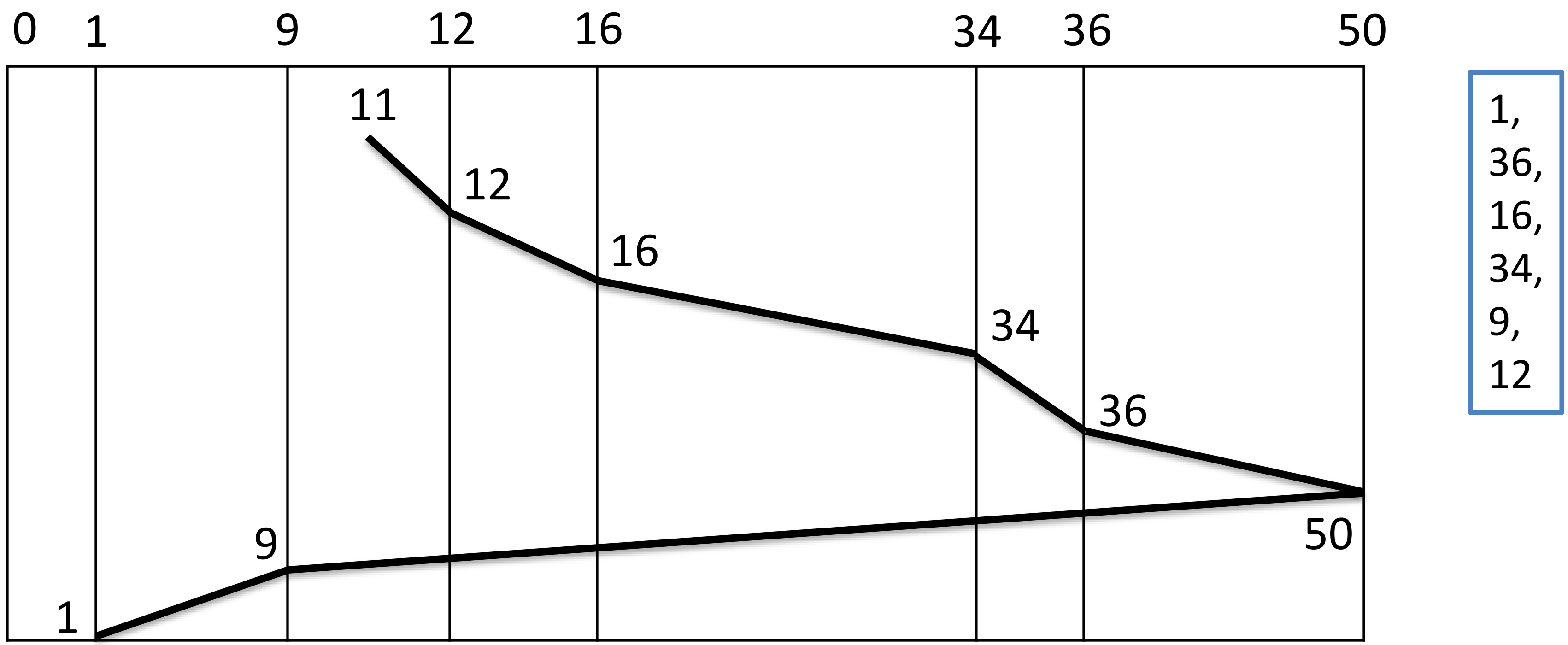


- Disk movement will be 11, 12, 16, 34, 36, 1, 9.
- Total cylinder movement: (12-11) + (16-12) + (34-16) + (36-34) +(36-1)+(9-1)=68

SCAN

- **From the current position disk arm starts in up direction and moves towards the end, serving all the pending requests until end**.
- At that end arm **direction is reversed (down) and moves towards the other end serving the pending requests on the way**.

SCAN

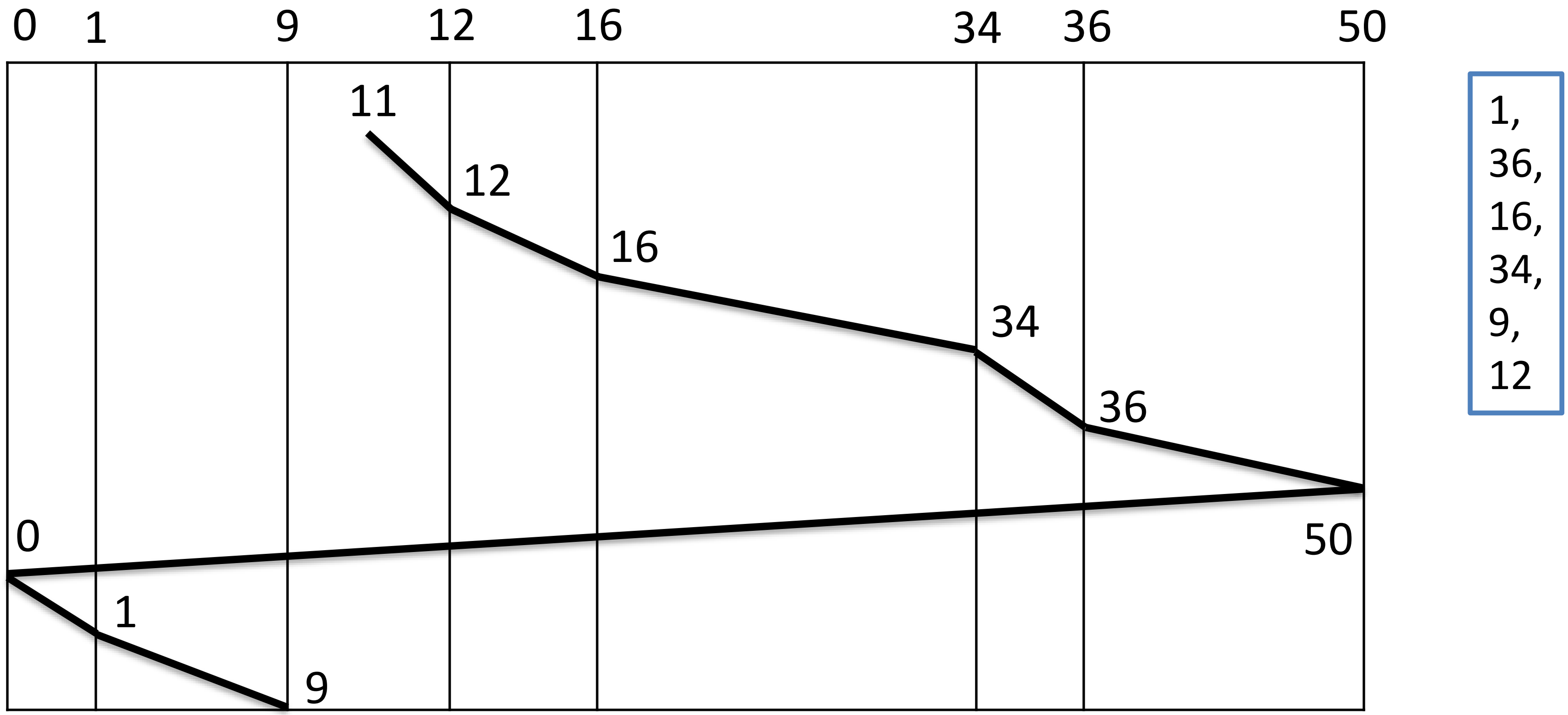


- Disk movement will be 11, 12, 16, 34, 36, 50, 9, 1.
- Total cylinder movement: (12-11) + (16-12) + (34-16) +(36-34) +(50-36) + (50-9) + (9-1) = 88

CSCAN

- From the **current position disk arm starts in up direction and moves towards the end, serving request until end**.
- At the end the arm **direction is reversed (down), and arm directly goes to other end and again continues moving in upward direction**.

CSCAN



- Disk movement will be 11, 12, 16, 34, 36, 50, 0, 1,9.
- Total cylinder movement: (12-11) + (16-12) + (34-16) +(36-34) +(50-36) + (50-0) + (1-0)+ (9-1) = 98

Examples for Disk Arm Scheduling Algorithm

- Consider an imaginary disk with 45 cylinders. A request comes in to read a block on cylinder 20. While the seek to cylinder 20 is in progress, new requests come in for cylinders 10, 22, 20, 2, 40, 6, and 38 in that order.
- Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests and how much seek time is needed for, for each of the disk scheduling algorithms if a seek takes 6 msec per cylinder moved?