Lecture 14

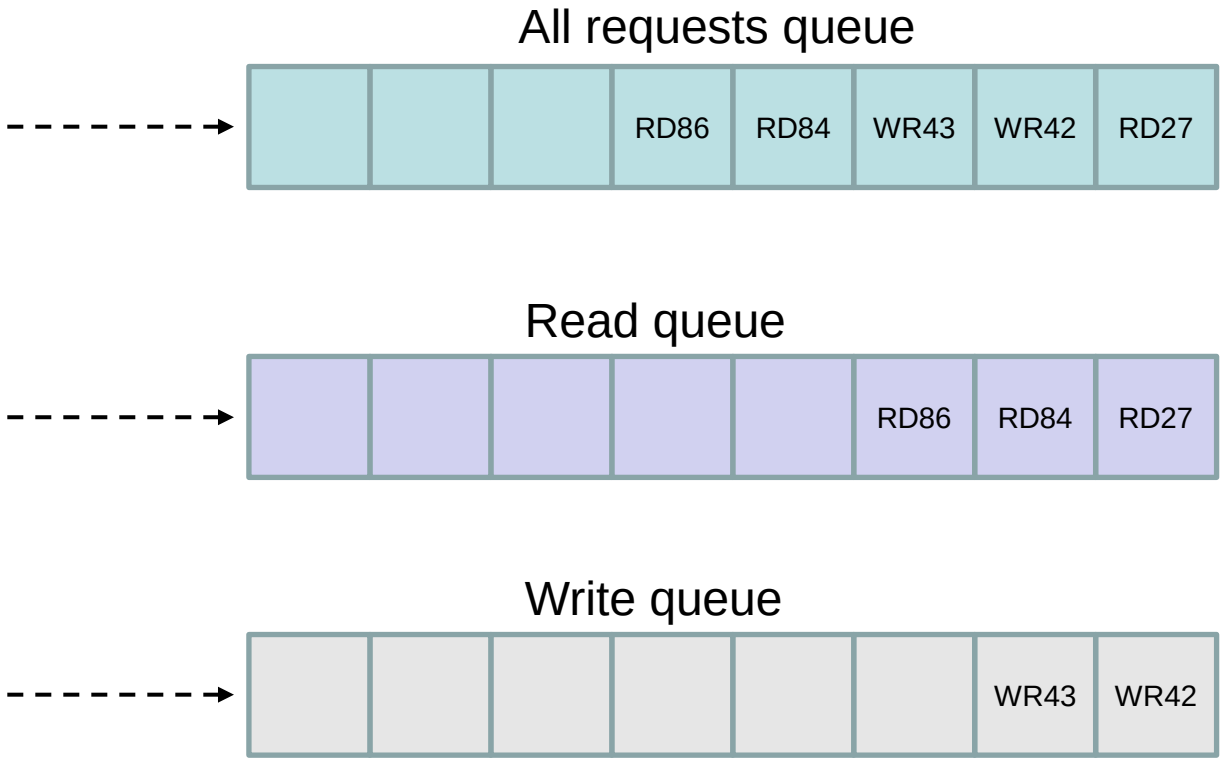
Disk scheduling and examples of I/O Device Management

The disk

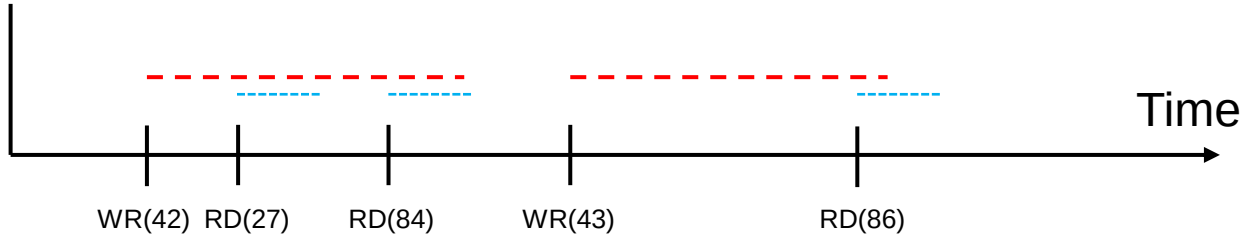
- Computer storage is still largely provided by disks. The disk is rotating at constant speed. To read or write, the head needs to be positioned on the track, at the beginning of the sector.
- Track positioning involves moving the head. The time taken by this operation is called *seek* time. Once on top of the track, the controller waits until the targeted sector arrives. The waiting time is called the *rotational delay* (latency).
- The access time is the sum of the seek time and rotational delay.
- The time taken for data read/write is the *transfer* time. Generally, there are additional delay times associated with disk I/O operations.
- If disk operations involve selection of track at random, the performance will be poor – the seek time contribution is high (in the order of ms). *Operations can be scheduled for a better result.*

A. I/O schedulers

- *Noop scheduler* merges adjacent requests: when a request addresses sectors adjacent to sectors accessed by a request already in the queue, the two can be combined into a single request.
- *Deadline scheduler* merges requests as above and maintains three queues: one for all read requests in FIFO order, one for write requests in FIFO order and one for all requests sorted by sector number. When being inserted, each request has an expiration time attached (e.g., read – 500 ms, write – 5 sec). The expiration times are selected to give preference to reads over writes (processes must block until reads are complete, writes are buffered, so that processes may continue…).
- The scheduler dispatches from the sorted queue. When a request is satisfied, it is removed from both the sorted queue and read or write queue. When a request at the head of a queue is older than the expiration time, the scheduler next dispatches from that queue (the expired plus some other requests).



Example: arrival of requests, WR(42), RD(27), RD(84), WR(43), RD(86)



- *Anticipatory scheduler* is a variation of the deadline scheduler; it introduces a delay of few msec at the end of each request service. If another request arrives for the same area of the disk during that time, it will handle it immediately. Otherwise, the delay expires and the scheduler goes back to processing the queues normally.

- *Complete fair queuing scheduler* maintains a separate queue for each process. Requests are merged and inserted in order of sector number. It schedules among the queues in a round-robin order. By default, it processes up to four requests from a process's queue before moving on to the next. In this way, no single process can starve other processes of disk access.

B. UNIX

- All input and output operations are carried out through the same *read()* and *write()* system calls that are used for file operations.
- Devices are also opened in the same way as files, using names managed by the file system.
- Querying and setting serial port parameters is handled through the *gtty()* and *stty()* system calls, respectively.
- Device drivers are included as part of the kernel image loaded at boot time.
- In UNIX, devices are either *block* or *character*. Applications directly open, read, write and close only character devices. Access to block devices is provided by the file system. However, most block devices drivers also support a *raw interface* similar to that for character devices.

Representation of devices

- When a device name, such that /dev/tty, is referenced, the file system translates it to three data items: a *major device number*, a *minor device number* and a *flag* indicating if the devices is a character or block one.
- In handling a system call, the flag is used to select one of two tables, *bdevsw* and *cdevsw*, that determine how the call is processed.
- The *major device* number is used to index the selected table. Each entry is a structure containing pointers to functions for handling specific operations.
- The *minor device* number is passed to the driver which may use it as it sees fit (i.e. indicates which device among several connected to a controller is selected). In some other cases, this number can be interpreted differently – e.g., rewound a tape when closed.

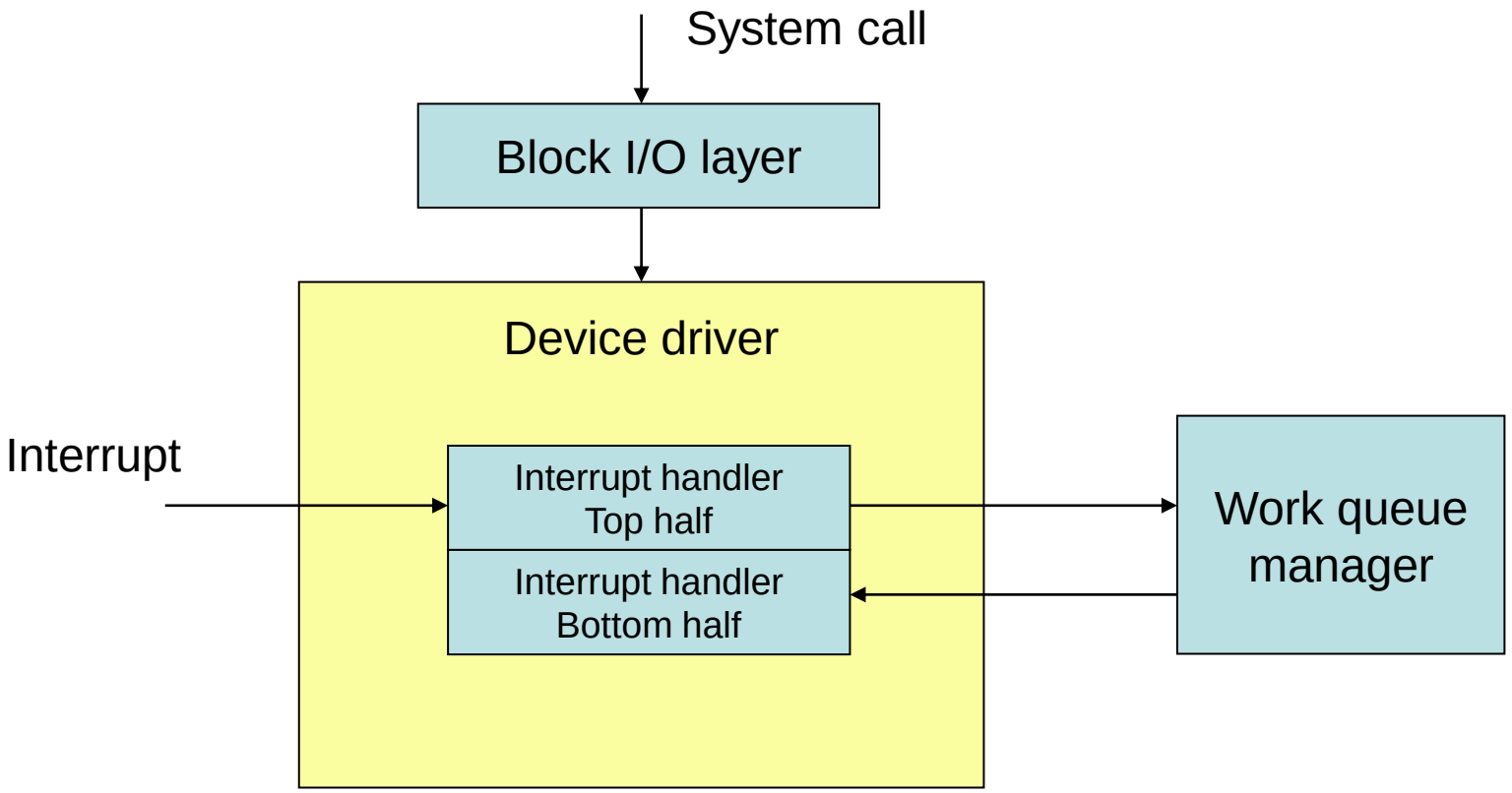
C. I/O devices in Linux

- As most of the details of managing I/O requests for block devices are independent of the device, Linux abstracts those details into a block I/O layer.
- This layer provides functions that manage request queues. Requests of type *struct request* are added to a queue by the file system. Then, they are broken into one or more structures of type *struct bio*, each of them corresponding to one I/O operation.
- Block device drivers specify a function to be called each time there is a block I/O operation that becomes available for processing. This happens either when a new request is added to an empty queue or when a request has been completed and is removed from the queue.

The elevator scheduler

- The default Linux disk scheduler is known as the Linux elevator.
- This scheduler maintains a single queue for read and write requests and performs both sorting (by block number) and merging. The head drive moves in one direction, satisfying each request as it is encountered.
- In Linux, the anticipatory scheduler is superimposed on the deadline scheduler:
  - When a read request is dispatched, the anticipatory component causes the scheduler to delay for up to 6 ms, waiting for a possible new read operation issued for the same region of the disk. If that is the case, the new read operation is served immediately.
  - If there is no other read operation, the scheduler resumes by using the deadline component.

D. Two-half interrupt handler



The top half is invoked by the interrupt; it does minimum work, ack the interrupt and retrieves the controller status. Then it schedules the bottom half to run at some time later.

E. Parallel port driver

- This is a simple *character driver* – the printer.

- When an application issues a write() on the file descriptor attached to a parallel port, the lp_write() function of the driver is called. It takes as arguments a pointer to a structure that describes the open file, a pointer to the data in the user process's memory space, and a count giving the number of bytes to write.
- Preparing to service the request: the size of operation is limited to the available buffer (the size of a page); the request is broken into a series of write operations, each one page in size.
- Fetching the data for the write: data is copied from the process space into the buffer.

- Setting up the hardware: if multiple processes/threads attempt to access the port simultaneously, they can interfere with each other and cause improper function. Therefore, the process needs *exclusive access* to the hardware port – critical resource. The next step is to determine which mode of operation is the proper one. The last step is setting a time out.

- Writing pages of data: call parport_write() for each page; this is followed by fetching the next page,…
- Cleaning up: the port is released by unlocking the mutex lock of this port data structure.

- The main function of parport_write() is to identify and call the appropriate function for the specific hardware and operating mode.