# CSCE 313-200
# Introduction to Computer Systems
# Spring 2024

## File System III

Dmitri Loguinov
Texas A&M University

March 27, 2024

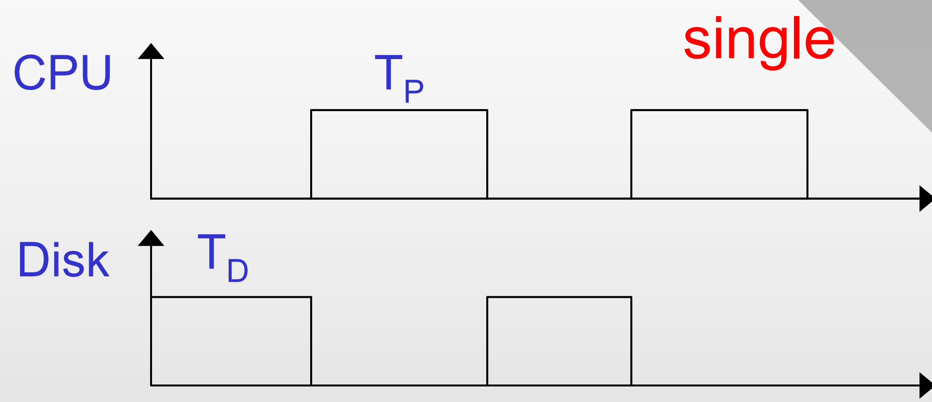
# Chapter 11: Roadmap

# I/O Function

- Programmed I/O (PIO)
  - CPU directly reads device, transferring data to RAM or CPU registers
  - Slow legacy devices (e.g., serial/parallel ports, PS/2 keyboard or mouse)
  - PIO mode 0 to 6: speed range 3.3-25 MB/s
- Not used for high-rate I/O
  - But appropriate for loading config registers from a device or initializing it
- Direct Memory Access (DMA)
  - DMA controller responsible for data transfer between device and RAM
- While PIO keeps the CPU occupied during entire I/O transaction, DMA is fully independent of the CPU
- Zero-copy transfer
  - Data bypasses intermediate buffers and gets to application through DMA

# Chapter 11: Roadmap

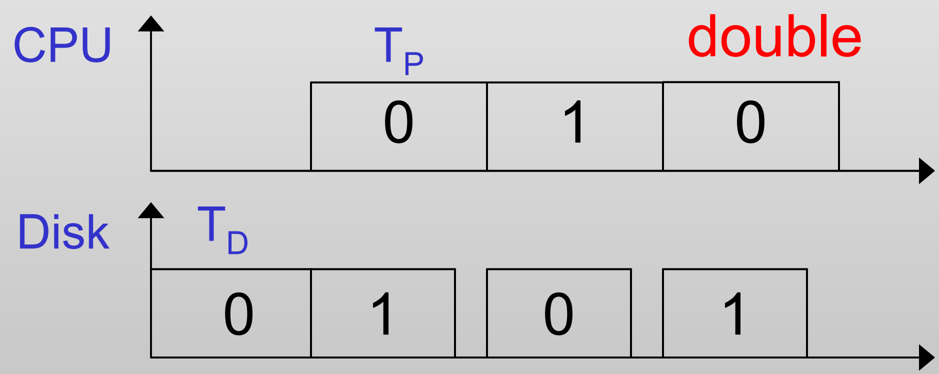
# App Buffering




- Consider application that processes data
- Single buffering

```
while (true) {
    ReadData (buf);
    ProcessData (buf);
}
```

  - Per-buffer delay $T_P+T_D$




- Double buffering requires at least two threads
  - Per-buffer delay $\max(T_P, T_D)$

```
semaFull = {0,2}; semaEmpty = {2,2}
int curDisk = 0;
while (true) {
    semaEmpty.Wait();
    ReadData (buf[curDisk]);
    semaFull.Release();
    curDisk ^= 1;
}
```

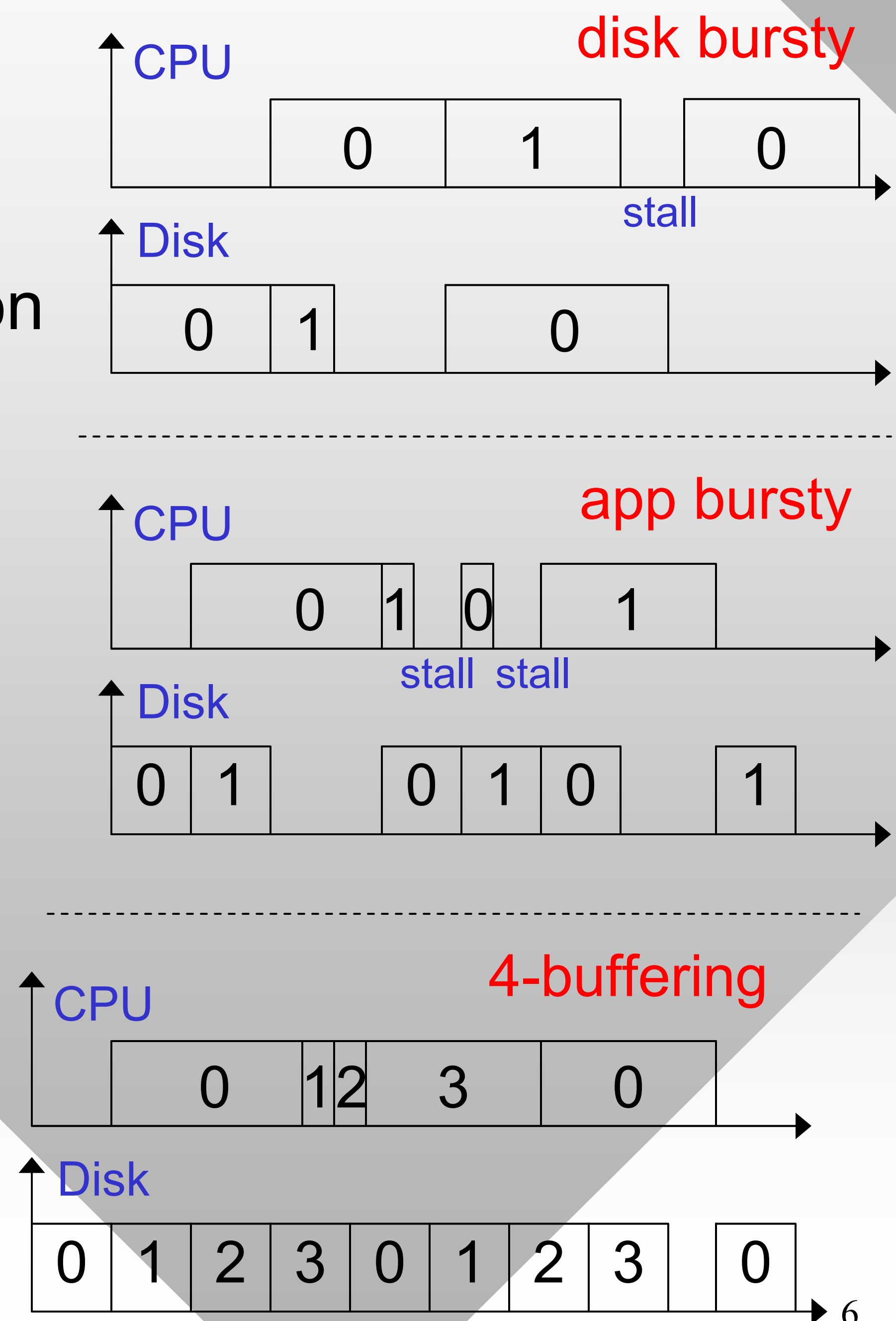
disk thread

```
int curProc = 0;
while (true) {
    semaFull.Wait();
    ProcessData (buf[curProc]);
    semaEmpty.Release();
    curProc ^= 1;
}
```

proc thread

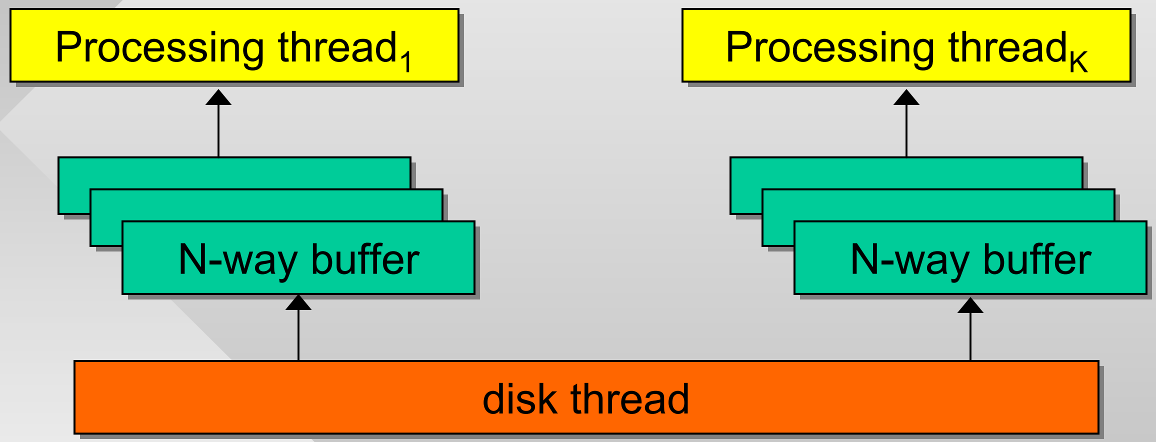
# App Buffering

- Suppose disk or application is bursty, but on average ReadData() is faster than ProcessData()
  - Even double-buffering may stall processing
- Multi-buffering
  - N ≥ 3 buffers, circular array
  - Solves the problem by reading ahead, smoothes out any fluctuations
- Easy for single thread, what about K threads?



# App Buffering

- Naïve approach: give each thread its own N-buffering




- Optimal management of buffers (load-balancing) requires a different architecture
  - See homework #3
- Why not make K independent disk threads?
  - Leads to disk-seek thrashing; no benefit to parallelization if there is only 1 disk and it's the bottleneck

# Inside the OS

- Single OS buffering is normal operation of ReadFile
  - ProcessData() is just a memcpy to user space →
- No OS buffering is used for extreme I/O rates (GB/s and faster) →
  - Earlier we called this zero-copy
- Note that the OS treats data in OSbuf as a cache
  - Makes it available on the next read through the file
  - Data that fits entirely in RAM can be served from the cache

```
// single buffer: T_D+T_copy
ReadFile (char *userBuf) {
    SetupDMA (OSbuf);
    WaitForDMA (OSbuf);
    memcpy (userBuf, OSbuf);
}
```

```
// no buffering: T_D
ReadFile(char *userBuf) {
    SetupDMA (userBuf);
    WaitForDMA (userBuf);
}
```

# Chapter 11: Roadmap

11.1 I/O devices

11.2 I/O function

11.3 OS design issues

11.4 I/O buffering

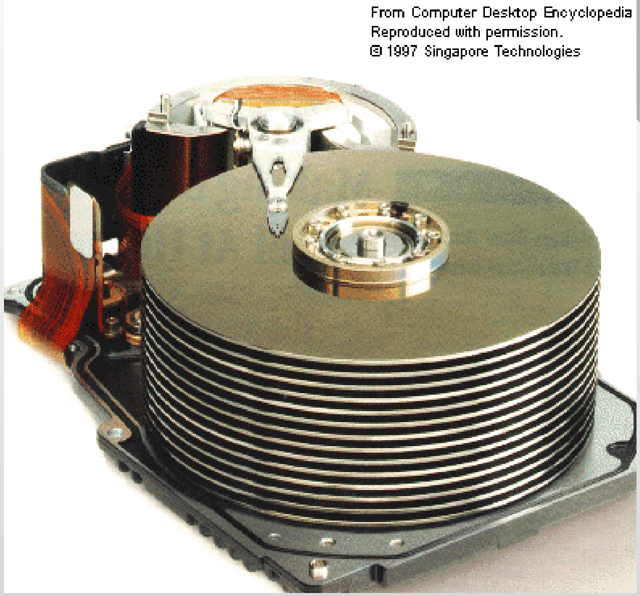
11.5 Disk scheduling

11.6 RAID

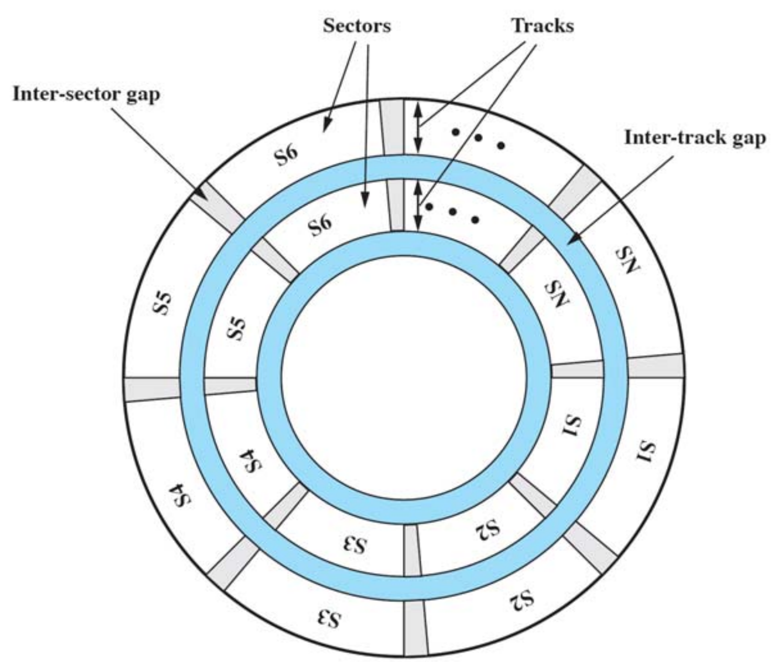
11.7 Disk cache

11.8-11.10 Unix, Linux, Windows

# Disk Internals




- Hard drive consists of P platters, each with two magnetic surfaces
  - Platters spin on a central spindle, rotational speed R is given in RPM
- Data is read using 2P heads, one for each surface
- Surface broken into K concentric circles called tracks
  - Track 0 near the outer edge
- Track consists of N sectors of B bytes each
- The same track on all 2P surfaces comprises a cylinder



# Disk Internals

- Question: how much can a disk read in one rotation?
  - C = 2P*N*B (cylinder size = number of surfaces * track size)
- Question: total disk capacity?
  - 2P*N*B*K = C*K (cylinder size * number of tracks)
- Question: for R=7200 RPM drive, how to figure out cylinder size and how many tracks it has?
  - Assume $\Delta$ is the inter-track delay during sequential read
  - Then, disk read speed S = C / ( 60/R + $\Delta$ )
  - Since $\Delta$ is unknown, we neglect it in our estimates
- Example: 2 TB Hitachi with 150 MB/s sustained read
  - Solving C*R/60 = 150 MB/s, we get C = 1.25 MB
  - Solving C*K = 2TB, we get K = 1.6M

# Disk Internals

Time to obtain b bytes from disk

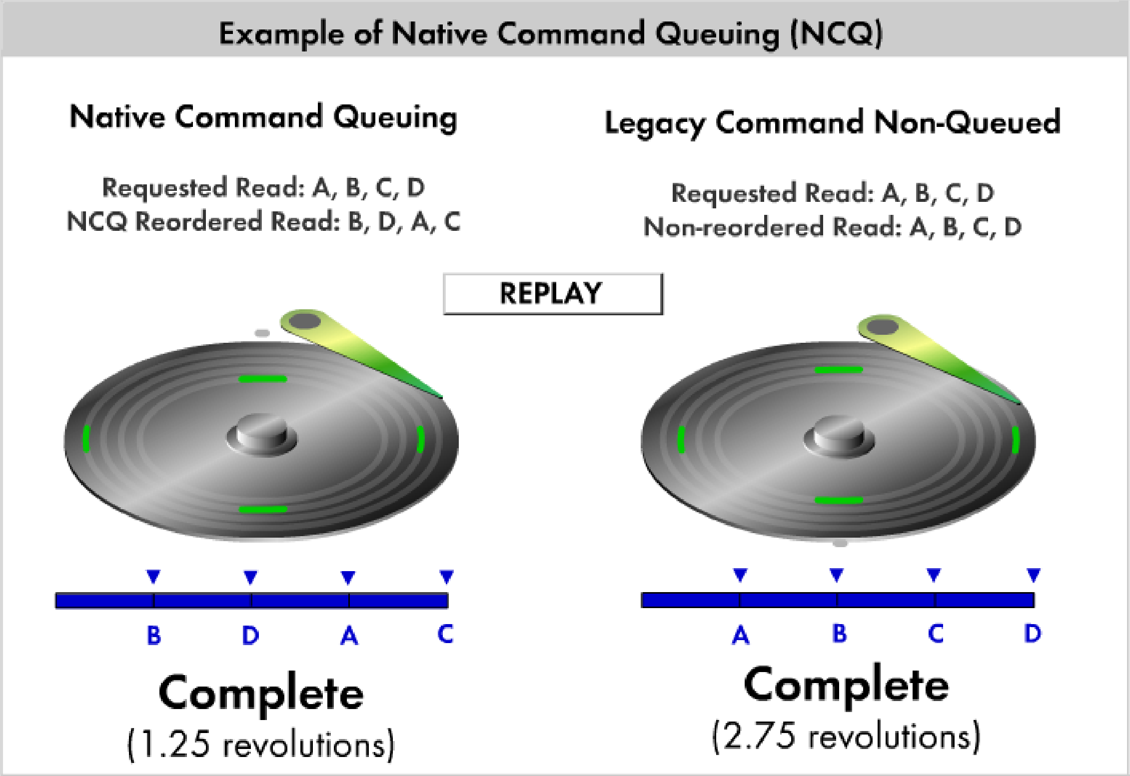
- Seek time $T_S$
  - Delay needed to move the heads to the right track
  - Includes time to start, move, and settle down
  - Average 8 ms for regular HDDs, ~0.1 ms for SSDs
- Rotational delay $T_R = 60 / (2*R)$
  - Time until the right sector passes under head
  - On average ½ revolution; for 7200 RPM, it's 4ms
  - Absent in SSDs
- Transfer delay $T_T = b / S$
  - Time to read a chunk of size b bytes
- Total time $T = T_S + T_R + T_T$

# Disk Internals

- Examples: total time to read one sector of Hitachi
  - T = 8 + 4 + 512 / 150e6 = 12.003 ms
- If we read sectors randomly across the disk?
  - Speed dominated by $T_S+T_R$, approx 41.6 KB/s
- Want 100 randomly scattered records in 15-MB file?
  - Seeking takes 1.2 seconds, reading the whole file 112 ms
- Lesson #1: disk seeking should be minimized
- If we read data sequentially, but one sector at a time?
  - One sector per revolution, i.e., 120 sectors/s, 60 KB/s
  - Usually speed isn't this bad due to internal HDD caching
- Lesson #2: sequential reads must be in large chunks

# Disk Internals

- Overlapped I/O sends multiple requests to HDD
  - Beneficial if supported by the underlying HDD protocol such as SATA NCQ (Native Command Queuing)



# Disk Internals

- Lessons
  - If data is sequential, reading small chunks not only creates a huge amount of kernel transitions, but also makes the disk inefficient at reading sectors
  - Should ask for at least a full cylinder per call
- NCQ/overlapped has several benefits:
  - Allows the drive to pull data out of order
  - Keeps the drive always reading ahead even when the OS is processing previous chunks (e.g., completing DMA housekeeping) or copying them to application buffers

# Overlapped I/O Example

- Demonstrate using N buffers, no data processing
  - Buffers are used sequentially

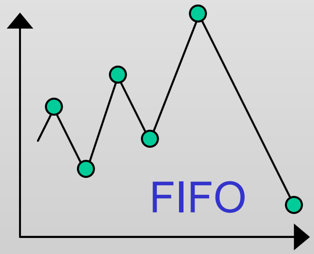
| | | |
|---|---|---|
| $buffer_0$ | … | $buffer_{N-1}$ |
| $ol_0$ | | $ol_{N-1}$ |

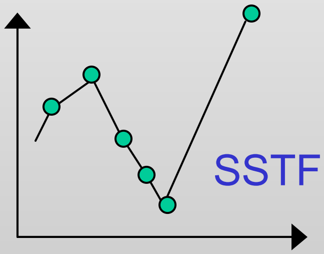
- This example just reads data in order, throws it away:
  - Obviously need to handle errors/EOF
  - If data is processed elsewhere, need to wait for buffer to be released before attempting a refill

```
OVERLAPPED ol[N];
memset (ol, 0, sizeof(OVERLAPPED) * N);
// create ol[i].hEvent
issue N overlapped requests to buf[0] ... buf[N-1]
int cur = 0;         // current buffer
while (true) {
    WaitForSingleObject (ol[cur].hEvent, INFINITE);
    GetOverlappedResult (..., ol + cur, ...);
    // process buffer[cur] and refill
    ReadFile (hFile, buffer[cur], ..., ol + cur);
    cur = (cur + 1)%N;
}
```

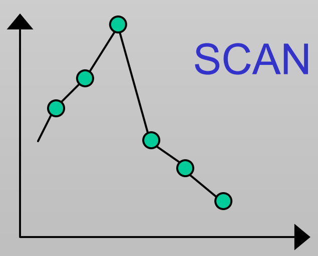
# Disk Scheduling

- When future requests are known, OS or HDD may optimize overall seek distance to reduce delay
- FIFO serves them in order
  - Main benefit is that it's fair




- Priority-based (OS decides)
- Shortest Service Time First (SSTF)
  - Nearest track from current location




- SCAN (elevator algorithm)
  - Serves tracks in increasing order until max, then scans back
- C-SCAN
  - Always scans upward until max, then returns to track 0
  - Reduces the worst wait delay compared to SCAN