Chapter 33: Event-based Concurrency (Advanced)

- Using threads to achieve concurrency is not the only method to build concurrent applications
- Event-based concurrency used in frameworks such as node.js are now popular
- Problems with event-based concurrency
    - Managing concurrency in multi-threaded applications can be challenging -> missing locks, deadlocks, etc.
    - The developer has little or no control over what is scheduled at a given moment in time
        - programmer can simply create threads and then hope that the underlying OS chedules them in a reasonable manner
- Therefore, it would be nice to build concurrent servers without threads

33.1: The Basic Idea: An Event Loop

- **Event based concurrency:** Wait for something to occur
    - when it does, you check what type of event is is and do the small amount of work it requires (I/O requests, scheduling other events, handling errors, etc.)
- When a handler processes an event, it is the only activity taking place in the system; thus deciding which event to handle next is equivalent of scheduling

33.2: An Important API: select() (or poll())

- select() and poll() interfaces enables a program to check whether is any incoming I/O available that should be attended to
    - ex: a network application that wishes to check whether any packet messages has been sent to the server/arrived
- select() lets you check whether descriptors can be read from as well as written to
    - checking whether a descriptor can be read allows a server to determine that a new packet has arrived while if it can be written to, it tells the server whether it is okay to send a response/reply
- Basic primitives give us a way to build a non-blocking event loop
- Aside: Blocking vs Non-blocking
    - **Blocking/Synchronous interfaces**: Do all of the work before returning to the caller
    - **Non-blocking/Asynchronous:** Begin some work but return immediately, letting whatever work that needs to be done get done in the background
    - Blocking interface example: I/O requests and disk reads
    - A non-blocking approach is very important in an event-based approach because a program that blocks can halt progress

33.3: Using select()

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    // open and set up a bunch of sockets (not shown)
    // main loop
    while (1) {
        // initialize the fd_set to all zero
        fd_set readFDs;
        FD_ZERO(&readFDs);

        // now set the bits for the descriptors
        // this server is interested in
        // (for simplicity, all of them from min to max)
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            FD_SET(fd, &readFDs);

        // do the select
        int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

        // check which actually have data using FD_ISSET()
        int fd;
        for (fd = minFD; fd < maxFD; fd++)
            if (FD_ISSET(fd, &readFDs))
                processFD(fd);
    }
}
```

Figure 33.1: **Simple Code Using `select()`**

- The above code, while in an infinite loop
    - looks at all of the file descriptors from minFD to maxFD
        - this could represent all of the available network sockets in a system
    - The server calls select to see which of the connections have data available upon them
    - By using FD_ISSET in a loop, the server can see which of the descriptors have data ready and process the incoming data

33.4: Why Simpler? No Locks Needed

- With a single CPU and an event-based application, the problems found in concurrent programs are no longer present
    - Because only one event is being handled at a time, there is no need to acquire/release locks
    - cannot be interrupted by another thread because it is decidedly single threaded
    - However, event-based applications must not have blocking operations

33.5: A Problem: Blocking System Calls

- What do you do if an event makes a system that call that requires blocking?
- Ex: Simple HTTP request
    - request comes from the client into a server that calls to read a file from disk and return its contents back to the client
    - An event handler must issue the open() system call to open the file, followed by a series of read() calls to read the file. The file is read into memory and the server then begins to send it over to the client

- The open() and read() system calls need to make I.O requests to the storage system so they may take a long time
    - with a thread based system, this is not an issue because we can run other tasks while handling these I/O requests
    - The server is unable to make progress in event-based approach while a blocking operation occurs
    - Therefore, blocking operations must not be used in an event-based application

33.6: A Solution: Asynchronous I/O

- Many OS's have now introduced solutions to the above problem through creating new ways to issue I/O requests to the disk system called **asynchronous I/O**
- Example: aio_read on mac
    - This call tries to issue the I/O
        - if successful, it simply returns right away and the application can continue its work
        - For every outstanding asynchronous I/O, an application can periodically poll the system via a call to aio_error() to determine whether the I/O has been completed or not
- Problem: What if a program has hundreds of I/Os issued at a given point in time. Should it simply keep checking them repeatedly or wait a little….
    - Some systems provide an approach based on interrupts
    - Uses UNIX signals to inform applications when an asynchronous I/O completes

Aside: UNIX Signals

- At its simplest, signals provide a way to communicate with a process
- Stops the application from whatever it is doing to run a signal handler, i.e. some code in the application to handle the signal -> once that is finished, it resumes execution
- Sometimes it is the kernel itself that sends the signal
    - ex: If we encounter a segmentation fault, the OS itself sends a SIGSEGV
    - if there is no signal handler, default behavior is executed -> for SIGSEGV it is to kill the process
    - We can use the kill command to explicitly send a signal so that it can be handled

Chapter 36: I/O Devices

- Crux: How should I/O be integrated into systems? What are the general mechanisms? How can we make them efficient?

36.1 System Architecture

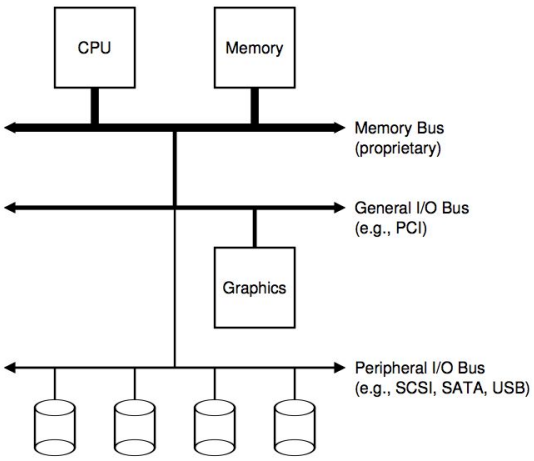


Figure 36.1: **Prototypical System Architecture**

- Single CPU attached to the main memory of the system via some kind of **memory bus**
- Some devices are connected to the system via a general I/O bus -> in modern systems would be PCI (graphics and other high-performance I/O devices are here)
- **Peripheral bus:** SCSI, SATA, USB -> connect the slowest devices to the system such as disks and mice
- The faster a bus is, the shorter it must be -> higher performance memory bus does not have much room to plug devices and such into it

36.2: A Canonical Device

- Two important components in a device
    - **Hardware interface:** hardware must present some kind of interface that allows the system software to control the operation
    - **Internal structure:** Implementation specific and is responsible for implementing the abstraction the device presents to the system
    - Simple devices will have one or a few hardware chips to implement their functionality -> more complex will include a CPU, some general purpose memory, and other device-specific chips

36.3: The Canonical Protocol

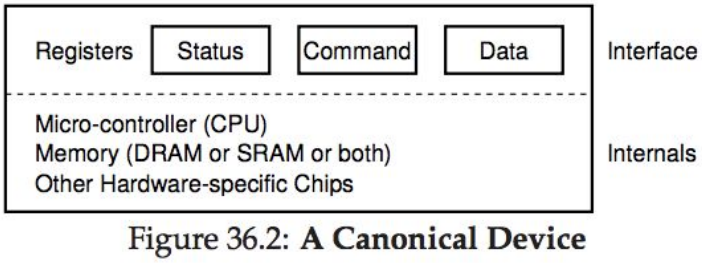


Figure 36.2: **A Canonical Device**

- Simplified interface comprised of three registers
    - **status:** current status of the device

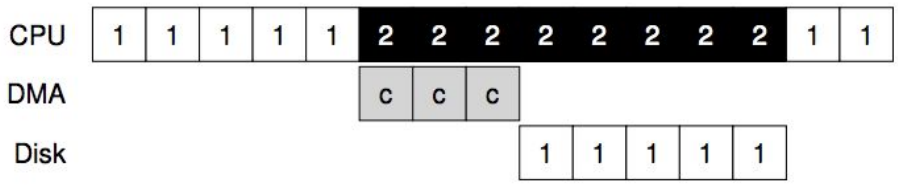
    - **command:** tells the device to perform a certain task
    - **data:** pass data to the device, or get data from the device
- The OS waits until the device is ready to receive a command by repeatedly polling in an infinite loop (like we saw in event-based applications)
- Second, the OS sends some data down to the data register and the multiple writes would need to take place to transfer a disk block to the device
    - When the main CPU is involved with the data movement, we refer to it as programmed I/O (PIO)
- Third, the OS writes a command to the command register -> lets the device know that both the data is present and that it should begin working on command
- OS waits for the device to finish by polling it in an infinite loop once again

36.4: Lowering CPU Overhead with Interrupts

- Because looping to repeatedly check whether data is available and can be read/written to can be inefficient since it wastes CPU cycles, we need to find a way to check device status without frequent polling and reduce CPU overhead
- One solution is an **interrupt**
    - OS issues a request, put the calling process to sleep, and then context switch to another task
    - When the device is finally finished with the operation, it will raise a hardware interrupt which causes the CPU to just to a predetermined interrupt handler
    - Interrupts thus allow for overlap of computation and I/O
    - With interrupts, the OS can run another task instead of spinning until an I/O request is completed
- However, **interrupts are not always the best solution**
    - If a device is able to perform its task very quickly, interrupts could possibly slow down the program
    - For fast tasks we should use polling while for slower tasks
    - It may be useful to use a hybrid of these two approaches when the speed of the task
- Another reason not to use interrupts arises in networks
    - When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to livelock
        - Beacuse the OS would be handling interrupts very often, it would not actually make any progress because the user-level process would not be run
- Another interrupt based optimization: **Coalescing**
    - a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU
    - Multiple requests many soon complete and thus multiple requests can be coalesced into a single interrupt delivery and reduce the overhead of interrupt processing
    - However, waiting too long would increase the latency of the process

36.5: More Efficient Data Movement with DMA

- We must use PIO to transfer a large chunk of data to a device
- The CPU is once again over burdened with a rather trivial task and thus wastes a lot of time and effort that could be spent running other processes
- For example, when a process wishes to write some data to the disk, it must copy the data from memory to the device explicitly, one word at a time
    - this wastes CPU cycles because the CPU could be doing something else
- **Direct Memory Access (DMA)**
    - A DMA engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention
    - To transfer data to the device, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to.
    - At that point, the OS is done with the transfer and can proceed with other work
    - When the DMA is complete, the DMA controller raises an interrupt -> now the OS knows that the transfer is complete



From the timeline, you can see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again.

    -
    - In Summary, without DMA the CPU is unable to perform other tasks while handling I/O requests such as accessing data from the disk. However, by having a DMA control the memory access, the process is able to perform other tasks while initiating I/O

36.6: Methods of Device Interaction

- How does the OS actually communicate with the devices?
- Have explicit I/O instructions as implemented by IBM
    - specify ways for the OS to send data to specific device registers and thus allow the construction of protocols
- In x86, the **in** and **out** instructions can be used to communicate with devices
    - to send data to a device, the caller specifies a register with the data in it, and a specific port which names the device. Thes instructions are usually privileged because the OS controls devices
- **Memory-mapped I/O**: The hardware makes device registers available as if they were memory locations

    - To access a particular register, the OS makes a load or a store to the address
    - the hardware then routes the load/store to the device instead of main memory

36.7: Fitting into the OS: The Device Driver

- Example: We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drives, etc.
- Therefore, we must be able to develop a device-neutral OS in order to hide the details of device interactions from OS subsystems
- At the lowest level, a piece of software in the OS must know in details how a device works. This piece of software is known as the **device driver** and any specifics of the device interactions are encapsulated within it

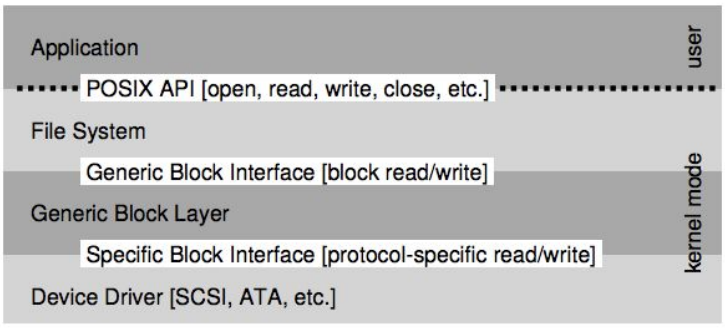


Figure 36.3: **The File System Stack**

- Above shows an approximation of the Linux File System
    - A file system is completely oblivious to the specifics of which disk class it is using; It simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request
    - Once again, this displays good modularity and information hiding as each subsystem of the OS is oblivious from the details of the other parts of the OS
- Because device drivers are needed for any device you might plug into your system, over time, they have come to represent the majority of the kernel code -> most of the code may not be active

36.8: Case Study: A Simple IDE Disk Driver

- Example: IDE disk driver
    - An IDE disk presents a simple interace to the system, consisting of four types of registers; control, command block, status, and error
    - These registers are available through reading or writing to the specific I/O address by using the in and out commands on x86

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6    5      4    3    2    1     0
   BUSY  READY FAULT SEEK  DRQ  CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when Status ERROR==1)
    7      6    5      4    3    2    1    0
   BBK    UNC   MC   IDNF  MCR  ABRT T0NF AMNF

   BBK  = Bad Block
   UNC  = Uncorrectable data error
   MC   = Media Changed
   IDNF = ID mark Not Found
   MCR  = Media Change Requested
   ABRT = Command aborted
   T0NF = Track 0 Not Found
   AMNF = Address Mark Not Found
```

Figure 36.4: **The IDE Interface**

- The basic protocol to interact with the device assuming it has already been initialized
    - **Wait for drive to be ready**: Read Status Register until drive is READY and not BUSY
    - **Write parameters to command registers:** Write the sector count, logical block address of the sectors to be accessed, and drive number to command registers
    - **Start the I/O:** Issue read/write to command register. Write READ--WRITE command to command register (shown above in 0x1F7)
    - **Data Transfer (for writes):** Wait until drive status is READY and DRQ (drive request for data) -> then write data to data port
    - **Handle interrupts:** In the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is completed
    - **Error handling:** After each operation, read the status register. If the ERROR bit is on, read the error register

Chapter 37: Hard Disk Drives

- Hard disk drives have been the main form of persistent data storage in computer systems for many decades of file system development

37.1: The Interface

- The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written
- Thus, we can view the disk as as an array of sectors; 0 to n-1 is thus the address space of the drive
- Multi-sector operations are still possible

    - However, a single 512-byte is always atomic meaning that if there is a loss of power, only a portion of the larger write may complete (torn write)
- Temporal and spatial locality apply in disk drives as a part of the “unwritten contract”
    - two blocks that are near one-another within the drive’s address space will be faster than accessing two blocks that are far apart

37.2: Basic Geometry

- **Platter:** a circular hard surface on which data is stored persistently by inducing magnetic changes to it
    - A disk may have one or more platters; each platter has two sides; each of which is called a surface
- Platters are all bounded together through a spindle which rotates at a constant rate measured by RPM
- Data is encoded on each surface in concentric circles of sectors -> each called a track
    - A single surface contains many thousand tracks
- To read and write from the surface, we have a mechanism that allows us to either sense the magnetic patterns on the disk or to induce a charge in them
- The process of reading and writing is accomplished by the **disk head** and the disk head is attached to the **disk arm,** which moves across the surface to position the head over the desired track

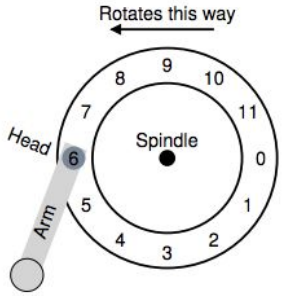


Figure 37.2: **A Single Track Plus A Head**

-

37.3: A Simple Disk Drive

- Single track Latency: The Rotational Delay
    - Example: we receive a request to read block 0
        - The disk must wait for the desired sector to rotate under the disk head
        - This delay is called a **rotational delay**
- Multiple Tracks: Seek Time
    - The disk arm must move to the correct corresponding track in a process known as **seek**
    - **Seeks and rotations are the most costly disk operations**
    - The seek has many phases

        - acceleration: when the disk gets going
        - coasting: when the arm is moving at its full speed
        - deceleration: when the arm slows down
        - settling: the head is carefully positioned over the correct track
    - When the last sector is passed over, the I/O transfer finally begins
- Some Other Details
    - Many drives employ some kind of track skew to make sure that sequential reads can be properly serviced even when crossing track boundaries
    - Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head
    - Outer tracks tend to have more sectors than inner tracks
    - Modern disk drives also have **cache**, also known as the track buffer
        - This cache is some small amount of memory which the drive can use to hold data read from or written to the disk
        - Reading a sector from the disk, the drive might decide to cache all of the sectors on one track, making subsequent calls faster
    - **Write back caching:** When the drive acknowledges that a write has completed when the data is in memory
    - **Write through caching:** When the drive acknowledges that a write has completed when the write has actually been written to disk
    - Write back caching makes the drive appear faster but can be dangerous if the file system or applications require that the data be written to disk in a certain order

37.4: I/O Time: Doing the Math

- T(I/O) = T(seek) + T(rotation) + T(transfer)
    - The rate of I/O is more easily use for comparison between drives and is easily computed from the time
    - R(I/O) = Size(transfer)/T(I/O)
- Random vs Sequential Workloads: Reads to random locations on disk vs consecutive sectors on disk
- “High Performance” drive market: Drives are engineered to spin as fast as possible, deliver low seek times, and transfer data quickly
- “Capacity” drive market: cost per byte is the most important aspect
- T(seek) = average time reported by the manufacturer
- T(rotation) = RPM -> RPS/2 for average amount of rotations per ms
- T(transfer) = size of transfer/peak transfer rate

37.5: Disk Scheduling

- Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/O’s issued to the disk
- Unlike job scheduling, we can make a good guess at how long a disk request will take by estimating the seek and possible rotational delay of the request
- Therefore, the disk scheduler will usually try to pick the shortest job first

- SSTF: Shortest Seek Time First
    - SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first
    - Example: If the head is on the inner track, and If we have requests for sectors 21 (middle track) and 2 (outer track), we must issue 21 first and then the request to 2 because 21 is the closest
    - Problems
        - Drive geometry is not available to the OS as it is seen as a simple array of blocks
            - OS can fix this problem through using a nearest-block-first algorithm which schedules the request with the nearest block address next
        - **Starvation:** If there was a steady stream of requests to the inner track, where the head is currently positioned, requests to any other tracks would then be ignored in a pure SSTF approach
- Elevator (a.k.a SCAN or C-SCAN)
    - The algorithm simply moves back and forth across the disk servicing requests in order across the tracks
    - Single pass across the disk is called a sweep
        - If a request comes for a block on a track that has already been serviced on this sweep, it is not handled immediately but rather queued until the next sweep
    - F-SCAN: freezes the queue to be serviced when it is doing a sweep
        - this action places requests that come in during that sweep into a queue to be serviced later
        - Avoids starvation through avoiding servicing requests that occur during this sweep
    - C-SCAN: Circular Scan
        - Instead of sweeping in both directions across the disk, the algorithm sweeps from outer to inner, and then resets the outer track to begin again
        - Other back-and-forth scans favors the middle tracks because it scans twice before checking the inner/outer tracks again
    - However, SCAN algorithms do not take into account both rotation and seek
- SPTF: Shortest Positioning Time First
    - Example: Head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide whether it should schedule sector 16 (middle track) or sector 8 (outer track) for its next request
    - The answer depends on the relative time of seeking as compared to rotation
        - If seek time is much higher than rotational delay, then SSTF is fine
        - If seek time is faster than rotational delay, then it should most likely service the outer track sector first

Other Scheduling Issues

- Where is the disk scheduling performed on modern systems?
    - Disks can accommodate multiple outstanding requests, and have sophisticated internal schedulers themselves
    - The OS scheduler usually picks what it thinks the best few requests are and issues them all to disk
        - The disk then uses the internal knowledge of disk head position, seek time, etc. to use SPTF to process the requests in the best possible order
- **I/O Merging**
    - Example: series of requests to read blocks 33, then 8, then 34 -> can merge requests 33 and 34 into one request to reduce overhead
- How long should the system wait before issuing an I/O to disk?
    - **Work-conserving:** Immediately issue the request to drive
    - **anticipatory disk scheduling/non-work conserving:** Wait for a bit before issuing the request to drive
        - By waiting, a new and better request may arrive and it may be possible to coalesce/merge the requests before sending them to disk, lowering overhead and increasing efficiency

Chapter 38: Redundant Arrays of Inexpensive Disks (RAIDs)

- A technique to use multiple disks in concert to build a faster, bigger, and more reliable disk system
- Externally, a RAID looks like a disk; a group of blocks one can read or write to
    - Internally, the RAID consists of multiple disks, memory, and one or more processors to manage the system
- RAIDS increase performance, capacity, and reliability
- RAIDS look just like a big disk to the host system, meaning that is transparent to the systems that use them

38.1: interface and RAID internals

- Just as a single disk, it presents itself as a linear array of blocks, each of which can be written or read by the file system
- When a file system issues a logical I/O request to the RAID, the RAID must internally calculate which disk to access in order to complete the request, and then issue one or more physical I/Os to do so
- Ex: a RAID that keeps two copies of each block
    - the RAID will have to perform two physical I/Os and one logical I/O
- RAID system is often built as a separate hardware box, with a standard connection as a host

38.2: Fault Model

- RAIDs are designed to detect and recover from certain kinds of disk faults
- **Fail-stop:** Disk can be in exactly two states: working or failed
    - With a working disk, all blocks can be read or written to

38.3: How to Evaluate a RAID

- **Capacity:** Given a set of N disks each with B blocks, how much useful capacity is available to clients of the RAID?
    - Without redundancy, N*B
    - With **mirroring**: (keeping two copies of the block) -> (N*B)/2
- **Reliability:** How many disk faults can that given design tolerate
- **Performance:** Depends on the workload that is presented to the disk array

38.4: RAID Level 0: Striping

- Serves an excellent upper-bound on performance and capacity -> no redundancy
- Striping will stripe blocks across the disks of the system
    - Spread the blocks of the array across the disks in a round-robin fashion
    - This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

-
- Blocks in the same row are called a **stripe (i.e. 0,1,2,3)**
- It is also possible to place multiple blocks on a single disk
    - we need to know the chunk size of a RAID array (you can place two 4KB blocks on one array entry, meaning that a chunk size could be 8KB*4 = 32KB per stripe)
- Aside: The RAID mapping problem
    - How does the RAID know which physical disk and offset to access?
    - Disk = block address%number_of_disks
    - Offset = A/number_of_disks
- **Chunk Sizes**
    - A small chunk size implies that many files will get striped across many disk, thus increasing parallelism of reads and writes to a single file
        - However, the positioning time to access blocks across multiple disks increase, because the positioning time for the entire request is determined by the maximum of the positioning time for the requests across all drives
- **RAID-0 Analysis**
    - Performance: Excellent - all disks are utilized, often in parallel, to service user I/O requests
    - Reliability: Any disk failure will lead to data loss

    - Capacity: given N disks each of size B blocks, striping delivers N*B blocks of useful capacity
- **Evaluating RAID Performance**
    - **Single request latency:** Latency of a single I/O request to a RAID -> reveals how much parallelism can exist during a single logical I/O operation
    - **Steady-state throughput:** total bandwidth of many concurrent requests
    - For a sequential workload, we will assume that requests come in large continuous chunks
    - For a random workload, we assume that each request is rather small, and that each request is to a different random location on disk
        - Used in database management systems
    - With sequential workloads, the disk operates efficiently, spending little time seeking and waiting for disk rotation and most of its time transferring data
    - Exercise: Calculate data transfer rate
        - Rate = Amount of Data/Time to access
- RAID-Analysis, again
    - From a latency perspective, the latency of a single-block request should be just about identical to that of a single disk
    - Throughput = N*S for sequential bandwidth of a single disk
    - Throughput = N*R(MB/s) for random I/Os
    - Both of the above values are both the simplest to calculate and will serve as an upper bound in comparison with other RAID levels
- In summary, RAID-Level 0/Striping presents upper bounds on capacity and performance (both latency and throughput) while reliability is a bit of an issue due to the fact that when a disk fails, data loss is guaranteed

38.5: RAID Level 1: Mirroring

- In a mirrored system, we simply make more than one copy of each block in the system, and the copy is placed in a separate so that we can tolerate disk failures

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

- In the above example, disk 0 and disk 1 have identical contents, and disk 2 and 3 do as well
- When reading a block from a mirrored array, the RAID has a choice
    - it can read either copy
- However, when writing a block, no such choice exists as the RAID must update both copies of data in order to preserve reliability
- **RAID-1 Analysis**

    - From a capacity standpoint, RAID-1 is expensive because we only obtain half of the data
    - **Reliability:** RAID-1 can tolerate the failure of any one disk
        - A mirrored system can tolerate between 1 and N/2 failures depending on which disks fail
        - Can fail without data loss
    - Performance
        - **Latency:** It is the same as when you have a single disk when reading because it only needs to read one of the two locations
            - Writes occur in parallel so they will roughly be the same as a single write
            - However, because the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay
        - **Throughput**
            - **Sequential Workload:** Each logical write must result in two physical writes, so the maximum bandwidth would be half of the peak bandwidth (B/2)
                - Performance not the same for a sequential read. although it seems like it is -> it is actually worse
                - The actual sequential read bandwidth is only half of the peak bandwidth as well
            - **Random Workload**: The best case for mirrored RAID because we can distribute the requests across a multiple disks and achieve full bandwidth
                - Random writes also achieve half the peak bandwidth, as expected
- Aside: The RAID Consistent-update problem
    - This problem occurs on a write to any RAID that has to update multiple disks during a single logical operation
    - If we need to write to both disk 0 and 1, but during our write to disk 0, a power loss occurs and the request to disk 0 is completed but the request to disk 1 is not.
    - This means that it could be possible that we end up with a new version in disk 0 and an old version in disk 1
        - Therefore, we want to execute everything atomically, meaning that the write is either completed or it is not
    - **Write-ahead log:** record what the RAID is about to do (update two disks with a certain piece of data) before doing it
        - We can run a recovery procedure based on the logging
    - Because logging every write is expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g. battery-backed) where it performs the logging

38.6: RAID Level 4: Saving Space with Parity

- Add redundancy to a disk array known as parity
- Parity based approaches attempt to use less capacity and thus overcome the huge **space penalty** paid by mirrored systems
    - At the cost of performance, however

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: **RAID-4 with Parity**

- Parity block P0 has redundant information calculated from 0,1,2, and 3, etc.
    - A simple function (XOR) is used and therefore there must be an even number of ones in any row in order to maintain correct parity

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

- If data in column 3 (C3) is lost, we know that each row must contain an even number of 1s, so we can retrieve information when data is lost
- **RAID-4 Analysis**
    - **Capacity:** RAID-4 uses one disk for parity so the capacity is therefore (N-1)*B
    - **Reliability:** RAID-4 tolerates at most one disk failure, otherwise it is simply impossible to reconstruct
    - **Performance**
        - **Steady-state throughput:**
            - **Sequential Read:** can utilize all of the disks except for parity so therefore has an effective bandwidth of (N-1)*B
            - **Sequential Writes:** are done through a simple optimization called a **full-stripe write**.
                - ex: blocks 0,1,2, and 3 sent to the RAID as a part of a write request
                - RAID can simply calculate the new value of P0 (by performing XOR on blocks 0,1,2,3) and then write all of the blocks (including the parity block) tot the five disks above in parallel
                - Effective bandwidth for sequential writes are also (N-1)*S MB/s
            - **Random Reads:** A set of random reads will be spread across the data disks of the system but not the parity disk. Thus, the effective performance is (N-1)*B

- **Random Writes:** Most interesting case in RAID-4

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

-
  - ex: We wish to overwrite block 1 in the example above
  - If we overwrite block 1, then the parity block would not accurately represent the data in the blocks
    - Therefore, we must update P0 both correctly and efficiently
  - Two methods
    - **additive parity:** read in all of the other data blocks in the stripe in parallel and XOR those with the new block. The result of this is the new parity block. To complete the write, you can then write the new data and new parity to their respective disks in parallel
      - problem: scales with the number of disks and thus larger RAIDs require a high number of reads to compute parities
    - **Subtractive Parity**

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0

      -
      - Ex: we want to overwrite bit C2 with a new value
      - Step 1: Read in the old data at C2 and the old parity
      - Step 2: Compare the old data and the new data
        - If they are the same, then the parity bit can remain the same
        - If they are not, we must flip the parity bit to the opposite of the current bit

$$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old} \quad (38.1)$$

      -
      - For each write using subtractive parity, the RAID must perform 4 physical I/O's two reads and two writes

        - RAID-4's throughput under random small writes are terrible and does not improve as you add disks to the system
    - Latency
        - Latency of a single read is just mapped to a single disk, and thus is equivalent to the latency of a single disk request
        - Latency of a single write requires two reads and then two writes; the reads can happen in parallel, as can the writes and thus the total latency is about twice that of a single disk

38.7: RAID Level 5: Rotating Parity

- Small-write problem: Parity disk is a bottleneck under a small random workload

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

-
- **RAID-5 Analysis**
    - **Effective capacity and failure tolerance is essentially the same as RAID-4**
    - **Random Read:**
        - Random read performance is slightly better because we can now utilize all disks instead of leaving out the parity disk
    - **Random Write**
        - improves noticeably over RAID-4 because it allows for parallelism across requests
    - When would you want to use RAID-4 over 5?
        - when a system knows that it will not perform anything other than a large write

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: **RAID Capacity, Reliability, and Performance**

-

Lecture Notes

Memory type busses

- Initially back-plan memory-to-CPU interconnects
    - a few “bus masters”, and many “slave devices”
    - arbitrated multi-cycle bus transactions
- Originally most busses were of this sort
    - ISA, EISA, PCMCIA, PCI, video busses, etc.
    - distinguished by
        - form factor, speed , data width, hot-plug
        - bridging, self-identifying, dynamic resource allocation

TERMS: Bus Arbitration & Mastery (SOMETHING ABOUT BUSSES ON EXAM)

- bus master
    - any device or CPU that can request the bus
    - one can also speak of the current bus master
- bus slave
    - a device that can only respond to bus requests
- bus arbitration
    - process of deciding to whom to grant the bus
        - may be based on time, geography, or priority
        - may also clock/choreograph steps of bus cycles
        - bus arbitrator may be part of the CPU or separate

Network type busses

- evolved as peripheral device interconnects
    - SCSI, USB, 1394, infiniband
    - cables and connectors rather than back-planes
    - designed for easy and dynamic extensibility

    - originally slower than back-plane, but no longer
- much more similar to a general purpose network
    - packet switched, topology, routing, node identity
    - may be master/slave (USB) or peer-to-peer (1394)
    - may be implemented by controller or by host

I/O architectures: devices and controllers

- I/O devices
    - peripheral devices that interface between the computer and other media (disks, tapes, networks, serial ports, keyboards, etc.)
- device controllers connect a device to a bus
    - communicate control operations to device
    - relay status info back to the bus
    - manage DMA transfers for the device
    - generate interrupts for the device
- controller usually specific to a device and a bus

Device Controller Registers

- device controllers export registers to the bus
    - registers in controller can be addressed from the bus
    - writing into registers control device or sends data
    - reading from registers obtains data/status
- register access method varies with CPU type
    - may require special instructions
        - privileged instructions restricted to supervisor mode
    - may be mapped onto bus like memory
        - accessed with normal (load/store) instructions
        - I/O address space not accessible to most processes

http://www.dtp.fmph.uniba.sk/pchardware/bus.html: good article on busses

Scenario: direct I/O with polling

```
uart_write_char( char c ) {
    while( (inb(UART_LSR) & TR_DONE) == 0);
    outb( UART_DATA, c );
}
char uart_read_char() {
    while( (inb(UART_LSR) & RX_READY) == 0);
    return( inb(UART_DATA) );
}
```

- while the done bit is not set, we want to spin

- store the character/data into the register
- If there is not character available, spin
- otherwise, retrieve the character from the register

Mechanism: Direct Polled I/O

- All transfers happen under direct control of CPU
    - CPU transfers data to/from device controller registers
    - transfers are typically one byte or word at a time
    - may be accomplished with normal or I/O instructions
- CPU polls device until it is ready for data transfer
    - received data is available to be read
    - previously initiated write operations are completed
- advantages
    - very easy to implement (both hardware and software)

Performance of direct I/O

- CPU intensive data transfers
    - each byte/word requires multiple instructions
- CPU wasted while awaiting completion
    - busy-wait polling ties up CPU until the I/O is completed
- devices are idle while we are running other tasks
- How can these problems be dealt with
    - let controller transfer data without attention from CPU
    - let application block pending I/O completion
    - let controller interrupt CPU when I/O is finally done

Importance of Good Device Utilization

- key system devices limit system performance
    - file system I/O, swapping, network communication
- if device sits idle, its throughput drops
    - this may result in lower system throughput
    - longer service queues, slower response times
- delays can disrupt real-time data flows
    - resulting in unacceptable performance
    - possible loss of irreplaceable data
- it is very important to keep key devices busy
    - start request n+1 immediately when n finishes

Direct Memory Access

- bus facilitates data flow in all directions between
    - CPU, memory, and device controllers
- CPU can be the bus master
    - initiating data transfers w/memory, device controllers

- device controllers can also master the bus
    - CPU instructs controller what transfer is desired
        - what data to move to/from what part of memory
    - controller does transfer w/o CPU assistance
    - controller generates interrupt at end of transfer

I/O interrupts

- Primary difference between trap and interrupt: trap comes from an illegal instruction in the CPU
- Device controllers, busses, and interrupts
    - busses have ability to send interrupts to the CPU
    - device signal controller when they are done/ready
    - when device finishes, controller puts interrupt on bus
- CPUs and interrupts
    - interrupts look very much like traps
        - traps come from CPU, interrupts caused externally
    - unlike traps, interrupts can be enabled/disabled
        - a device can be told it can or cannot generate interrupts
        - special instructions can enable/disable interrupts to CPU
        - interrupt may be held pending until s/w is ready for it

Keeping Key Devices Busy

- Allow multiple requests pending at a time
    - queue them, just like processes in the ready queue
    - requesters block to await eventual completions
- use DMA to perform the actual data transfers
    - data transferred, with no delay, at device speed
    - minimal overhead imposed on CPU -> doesn't need help from the CPU
- when the currently active request completes
    - device controller generates a completion interrupt
    - interrupt handler posts completion to requester
    - interrupt handler selects and initiates next transfer
    - interrupt driven I/O

Mechanisms: memory mapped I/O

- DMA may not be the best way to do I/O
    - designed for large contiguous transfers
    - some devices have many small sparse transfers
        - e.g. consider a video game displayer
- Implement as a bit-mapped display adaptor
    - !M pixel display controller, on the CPU memory bus
    - each word of memory corresponds to one pixel
    - application uses ordinary stores to update display

- low overhead per update, no interrupts to service
- relatively easy to program

Trade-Off: Memory Mapped vs DMA

- DMA performs large transfers efficiently
    - better utilization of both the devices and the CPU
        - device doesn't have to wait for CPU to do transfers
    - But there is considerable per transfer overhead
        - like calling an uber to cross the street
        - setting up the operation, processing completion, interrupt
- Memory-mapped I/O has no per-operation overhead
    - but every byte is transferred by a CPU instruction
        - no waiting because device accepts data at memory speed
- DMA better for occasional large transfers
- memory-mapped better frequent small transfers
- memory-mapped devices more difficult to share

I/O mechanisms: smart controllers

- smart controllers can improve on basic DMA
- they can queue multiple input/output requests
    - when one finishes, automatically start the next one
    - reduce completion/start-up delays
    - eliminate need for CPU to service interrupts
- They can relieve CPU of other I/O responsibilities
    - request scheduling to improve performance
    - they can do automatic error handling and retries
- abstract away details of underlying devices

Disk Drive Geometry

- Spindle
    - a mounted assembly of circular patterns
- head assembly
    - read/write head per surface, all moving in unison
- track
    - ring of data readable by one head in one position
- cylinder
    - corresponding tracks on all platters
- sector
    - logical records written within tracks
- disk address = cylinder/head/sector

Disks have Dominated File Systems

- Fast swap, file system, database access

- minimize seek overhead
    - organize file systems into cylinder clusters
    - write-back caches and deep request queues
- minimize rotational latency delays
    - maximum transfer sizes
    - buffer data for full-track reads and writes
- we accepted poor latency in return for IOPS

Optimizing Disk Performance

- Don't start I/O until disk in on cylinder/near sector
    - I/O ties up the controller, looking out other operations
    - other drives seek while one drive is doing I/O (seeks in parallel)
- minimize head motion
    - do all possible reads in current cylinder before moving
    - make minimum number of trips in small increments
- encourage efficient data requests
    - have lots of requests to choose from
    - encourage cylinder locality
    - encourage largest possible block sizes

The Changing I/O Landscape

- Storage Paradigms
    - Old: swapping, paging, file systems, databases
    - New: NAS (network address storage), distributed object/key-value stores
- i/O traffic
    - old: most I/O was disk I/O
    - new: network and video dominate many systems
- Performance goals
    - old: maximize throughput, IOPS
    - new: low latency, scalability, reliability, availability

Bigger Transfers are Better

- disks have high seek/rotation overheads
    - larger transfers amortize down the cost/byte
- all transfers have per-operation overhead
    - instructions to set up operation
    - device time to start new operation
    - time and cycles to service completion interrupt
- larger transfers have lower overhead/byte
    - this is not limited to s/w implementations

I/O Buffering

- Fewer/large transfers are more efficient

    - they may not be convenient for applications
    - natural record sizes tend to be relatively small
- Operating system can buffer process I/O
    - maintain a cache of recently used disk blocks
    - accumulate small writes, flush out as blocks fill
    - read whole blocks, deliver data as requested
- Enables read-ahead
    - OS reads/caches blocks not yet required

Deep Request Queues

- Having many I/O operations queued is good
    - maintains high device utilization (little idle time)
    - reduces mean seek distance/rotational delay
    - may be possible to combine adjacent requests
- Ways to achieve deep queues
    - many processes making requests
    - individual processes making parallel requests
    - read-ahead for expected data requests
    - write-back cache flushing

Double-buffered Output

- multiple buffers queued up, ready to write
    - each write completion interrupt starts next write
- application and device I/O proceed in parallel
    - application queue successive writes
        - don't bother waiting for previous operation to finish
    - device picks up next buffer as soon as it is ready
- If we're CPU-bound (more CPU than output)
    - application speeds up because it doesn't wait for I/O
- If we're I/O -bound (more output from CPU)
    - device is kept busy, which improves throughput
    - but eventually we have to block the process

Double Buffered Input

- have multiple reads queued up, ready to go
    - read completion interrupt starts read into next buffer
- filled buffers wait until application asks for them
    - application doesn't have to wait for data to be read
- when can we do chain-scheduled reads?
    - each app will probably block until its read completes
        - so we don't get multiple reads from one application
    - we can queue reads from multiple processes
    - we can do predictive read-ahead

Data Striping for Bandwidth

- spread requests across multiple targets
    - increased aggregate throughput
    - fewer operations per second per target
- used for many types of devices
    - disk or server striping
    - NIC bonding
- potential issues
    - more/shorter requests will be less efficient
    - source can generate many parallel requests
    - striping agent throughput is the bottleneck

Data Mirroring for Reliability

- mirror writes to multiple targets
    - redundancy in case a target fails
    - spread reads across multiple targets
        - increased aggregate throughput, reduced ops/target
- used for all types of persistent storage
    - disks, NAS, distributed key-value stores
- potential issues
    - added write traffic on the source
    - 2x-3x storage requirements on targets
    - deciding which (conflicting) copy is correct

Parity/Erasure Coding for Efficiency

- N out of M encoding (with M/N overhead)
    - accumulate N writes from a source
    - compute M versions of that collection
    - send a version to each of M targets
- Commonly used for archival storage
- Potential issues
    - greatly increased source computational load
    - deferred writes for parity block accumulation
    - expensive updates, recovery (and EC reads)
    - choosing the right ratio

Error Detection/Correction Terms

- Parity
    - typically one bit per byte, detect single-bit errors
    - used as redundancy, it can recover one lost bit/block
- Cyclical Redundancy Check (CRC)
    - multiple bits per record, detect multi-bit burst errors
- Error Correcting Coding (ECC)

    - fixed ratio, capable of detection and correction
    - e.g. Reed-Soloman can correct 8 bad bits
- Erasure Coding (distributed Reed-Soloman)
    - transforms K blocks into N (>K)
    - all can be recovered from any K (of those N) blocks

Parallel I/O Paradigms

- Busy, but periodic checking just in case
    - new input might cause a change of plans
- Multiple semi-independent streams
    - each requiring relatively simple processing
- Multiple semi-independent operations
    - each requiring multiple, potentially blocking steps
- Many balls in the air at all times
    - numerous parallel requests from many clients
    - keeping I/O queues full to improve throughput

Enabling Parallel Operations

- Threads are an obvious solution
    - one thread per-stream or per-request
    - streams or requests are handled in parallel
    - when one thread blocks, others can continue
- There are other parallel I/O mechanisms
    - non-blocking I/O
    - multi-channel poll/select operations
    - asynchronous I/O

Non-blocking I/O

- check to see if data/room is available
    - but do not block to wait for it
    - this enables parallelism, and prevents deadlocks
- a file can be opened in a non-blocking mode
    - open(name, flags | O_NONBLOCK)
- if data is available, read(2) will return it
    - otherwise it fails with EWOULDBLOCK
- can also be used with write(2) and open(2)