Lecture 17: Mass-Storage Systems (cont.)

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

"Those parts of the system that you can hit with a hammer are called hardware;

those program instructions that you can only curse at are called software."

-- Anonymous

Outline – Chapter 10/12

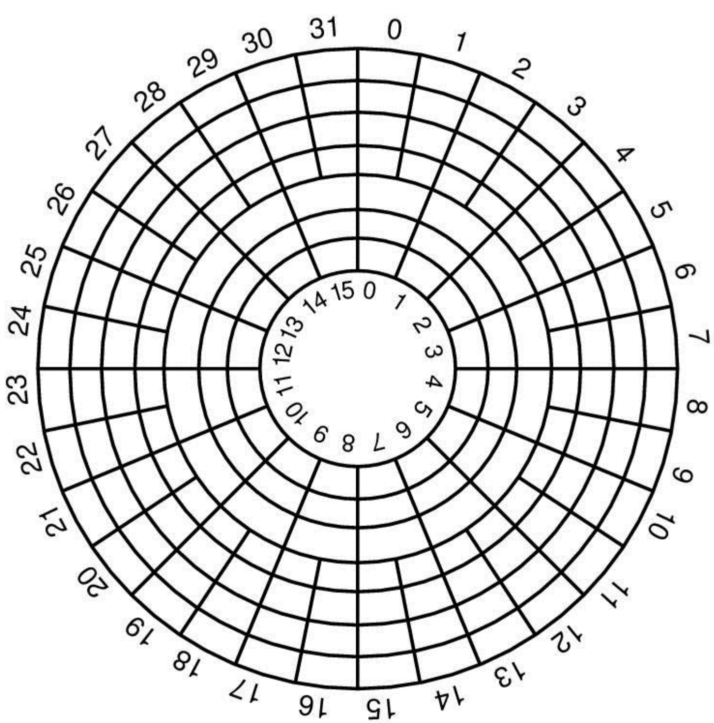
- Overview of Mass Storage Structure
- Disk Structure
- Disk Attachment
- Disk Scheduling
- **Disk Management**
- **Swap-Space Management**
- **RAID Structure**
- **Stable-Storage Implementation**

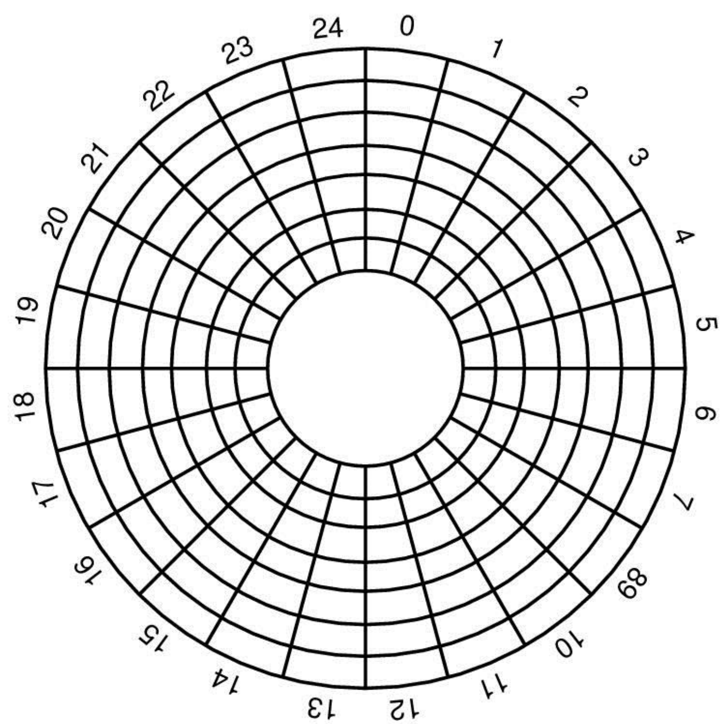
Disk Structure

- Disk drives addressed as 1-dim arrays of *logical blocks*
  - The logical block is the smallest unit of transfer
- This array mapped sequentially onto disk sectors
  - Address 0 is 1^{st} sector of 1^{st} track of the outermost cylinder
  - Addresses incremented within track, then within tracks of the cylinder, then across cylinders, from innermost to outermost
- Translation is theoretically possible, but usually difficult
  - Some sectors might be defective
  - Number of sectors per track is not a constant

Non-uniform #sectors / track

- Reduce bit density per track for outer layers (Constant Linear Velocity, typically HDDs)
- Have more sectors per track on the outer layers, and increase rotational speed when reading from outer tracks (Constant Angular Velcity, typically CDs, DVDs)





Disk Formatting

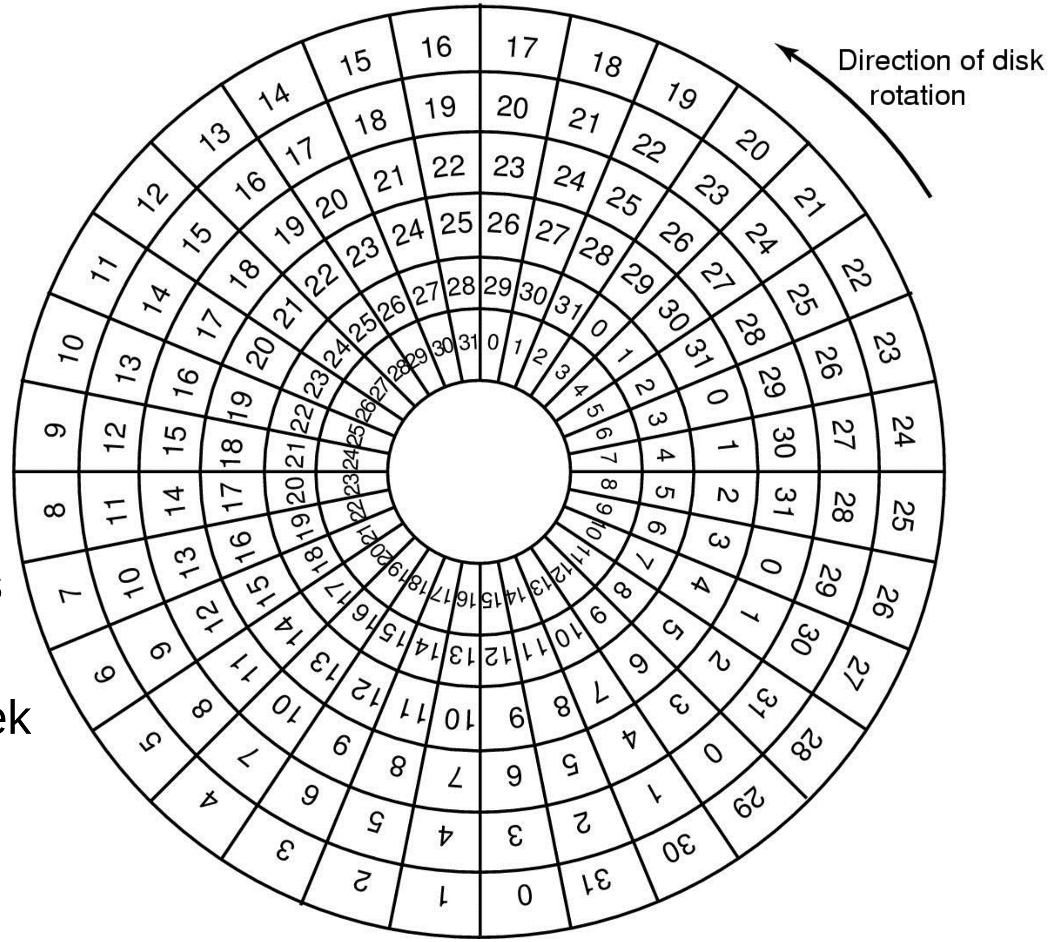
- After manufacturing disk has no information
  - Is stack of platters coated with magnetizable metal oxide
- Before use, each platter receives low-level format
  - Format has series of concentric tracks
  - Each track contains some sectors
  - There is a short gap between sectors

- Preamble allows h/w to recognize start of sector
  - Also contains cylinder and sector numbers
  - Data is usually 512 bytes
  - ECC field used to detect and recover from read errors

Cylinder Skew

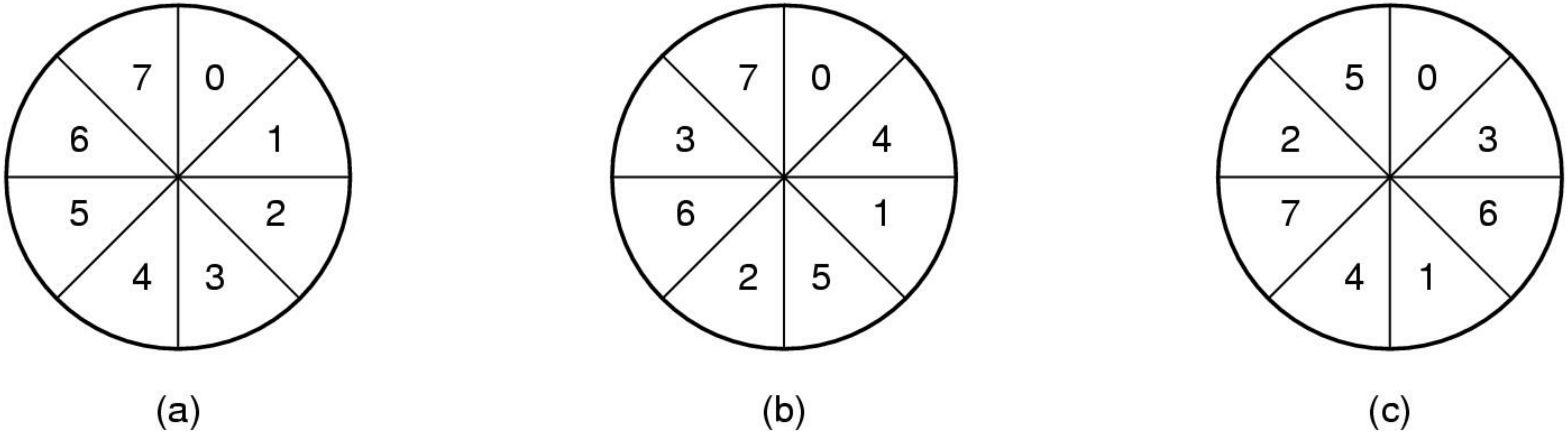
- Why cylinder skew?
- How much skew?
- Example, if
  - 10,000 rpm
    - Drive rotates in 6 ms
  - Track has 300 sectors
    - New sector every 20 µs
  - If track seek time 640 µs
    - $\Rightarrow$ 32 sectors pass on seek

Cylinder skew: 32 sectors



Formatting and Performance

- If 10K rpm, 300 sectors of 512 bytes per track
  – 153600 bytes every 6 ms $\Rightarrow$ 24.4 MB/sec transfer rate
- If disk controller buffer can store only one sector
  – For 2 consecutive reads, 2^{nd} sector flies past during memory transfer of 1^{st} track
  – Idea: Use single/double interleaving

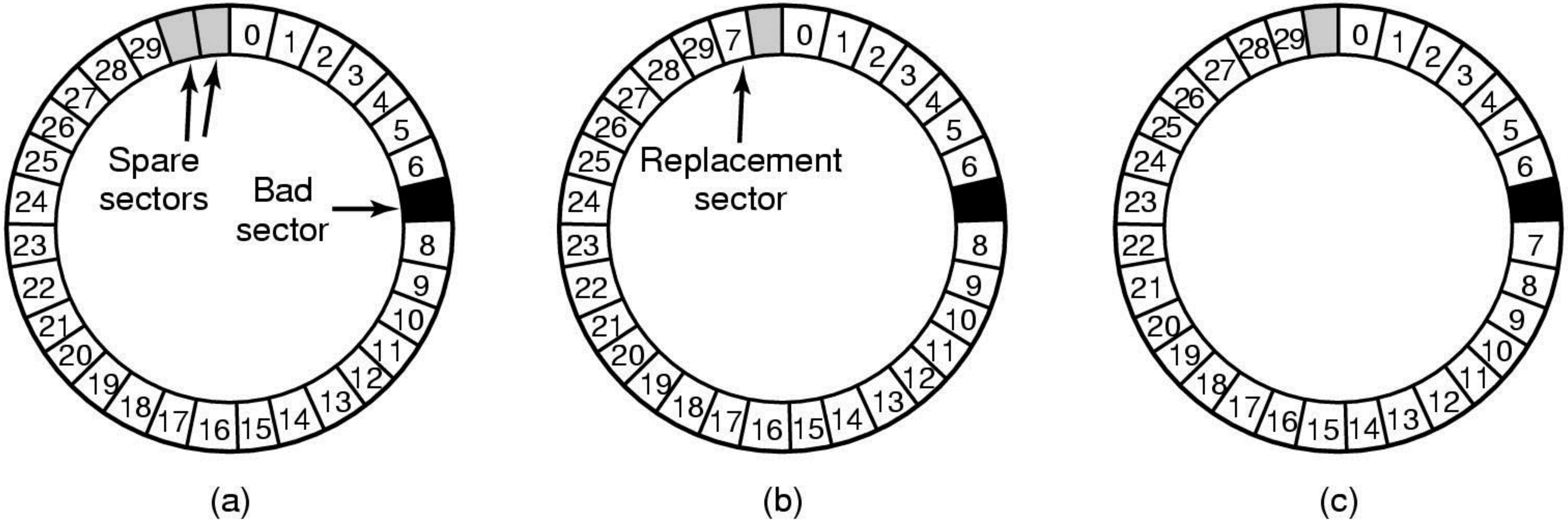


Disk Partitioning

- Each partition is like a separate disk
- Sector 0 is MBR
  - Contains boot code + partition table
  - Partition table has starting sector and size of each partition
- High-level formatting
  - Done for each partition
  - Specifies boot block, free list, root directory, empty file system
- What happens on boot?
  - BIOS loads MBR, boot program checks to see active partition
  - Reads boot sector from that partition that then loads OS kernel, etc.

Handling Errors

- A disk track with a bad sector
- Solutions:
  - Substitute a spare for the bad sector (sector sparing)
  - Shift all sectors to bypass bad one (sector forwarding)



RAID Motivation

- Disks are improving, but not as fast as CPUs
  - 1970s seek time: 50-100 ms.
  - 2000s seek time: <5 ms.
  - Factor of 20 improvement in 3 decades
- We can use multiple disks for improving performance
  - By **striping** files across multiple disks (placing parts of each file on a different disk), parallel I/O can improve access time
- Striping reduces reliability
  - 100 disks have 1/100th mean time between failures of one disk
- So, we need **striping for performance**, but we need something to help with reliability / availability
- To improve reliability, we can **add redundant data** to the disks, in addition to striping

RAID

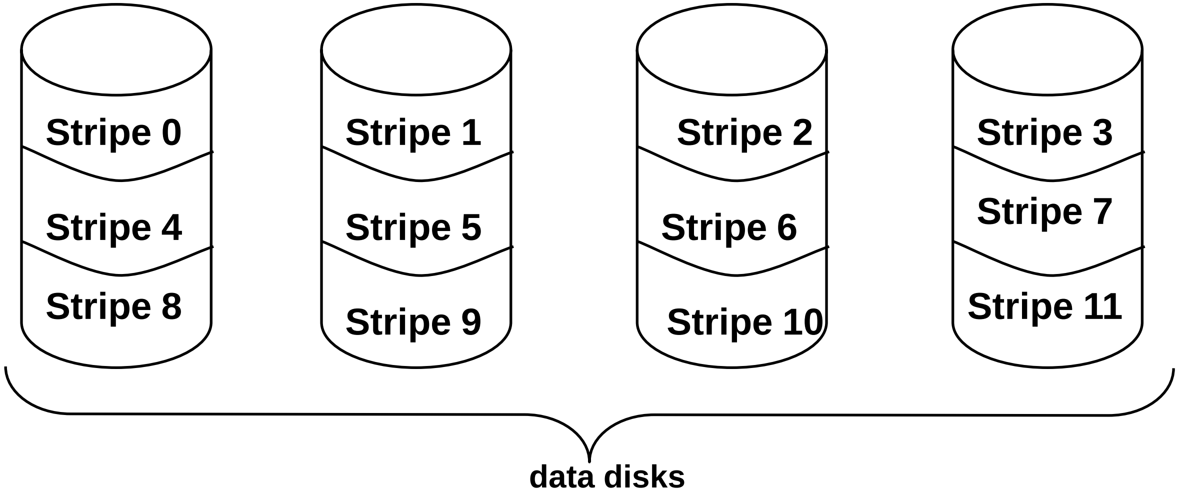
- A RAID is a Redundant Array of Inexpensive Disks
  - In industry, “I” is for “Independent”
  - The alternative is SLED, single large expensive disk
- Disks are small and cheap, so it’s easy to put lots of disks (10s to 100s) in one box for increased storage, performance, and availability
- The RAID box with a RAID controller looks just like a SLED to the computer
- Data plus some redundant information is striped across the disks in some way
- How that striping is done is key to performance and reliability.

Some RAID Issues

- **Granularity**
  - fine-grained: Stripe each file over all disks. This gives high throughput for the file, but limits to transfer of 1 file at a time
  - coarse-grained: Stripe each file over only a few disks. This limits throughput for 1 file but allows more parallel file access
- **Redundancy**
  - uniformly distribute redundancy info on disks: avoids load-balancing problems
  - concentrate redundancy info on a small number of disks: partition the set into data disks and redundant disks

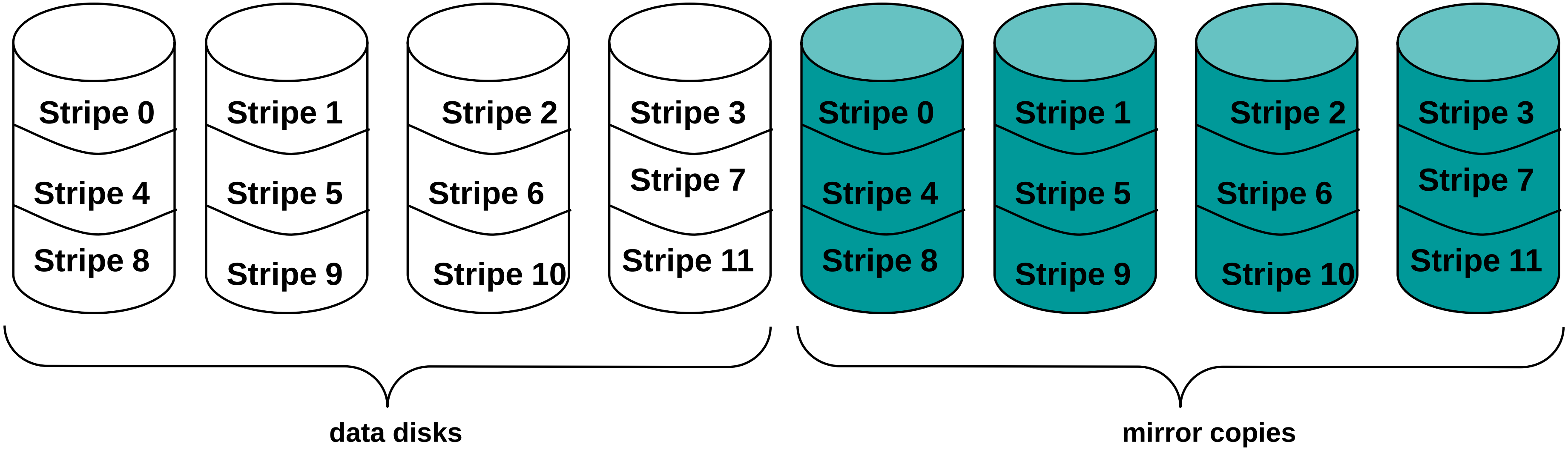
Raid Level 0

- Level 0 is **non-redundant** disk array
- Files are striped across disks, **no redundant info**
- High read throughput
- Best write throughput (no redundant info to write)
- **Any disk failure results in data loss**
  - Reliability worse than SLED



Raid Level 1

- **Mirrored Disks**
- **Data is written to two places**
  - On failure, just use surviving disk
- On read, choose fastest to read
  - Write performance is same as single drive
  - Read performance is 2x better
- Expensive



Parity and Hamming Codes

- What do we need to do to detect and correct a one-bit error ?
  - Suppose you have a binary number, represented as a collection of bits: <b3, b2, b1, b0>, e.g. **0110**
- Detection is easy
- Parity:
  - Count the number of bits that are on, see if it's odd or even
    - EVEN parity is 0 if the number of 1 bits is even
  - Parity(<b3, b2, b1, b0 >) = P0 = $b0 \otimes b1 \otimes b2 \otimes b3$
  - Parity(<b3, b2, b1, b0, p0>) = 0 if all bits are intact
  - Parity(0110) = 0, Parity(01100) = 0 => No Error
  - Parity(11100) = 1 => ERROR!
  - Parity can detect a single error, but can't tell you which of the bits got flipped

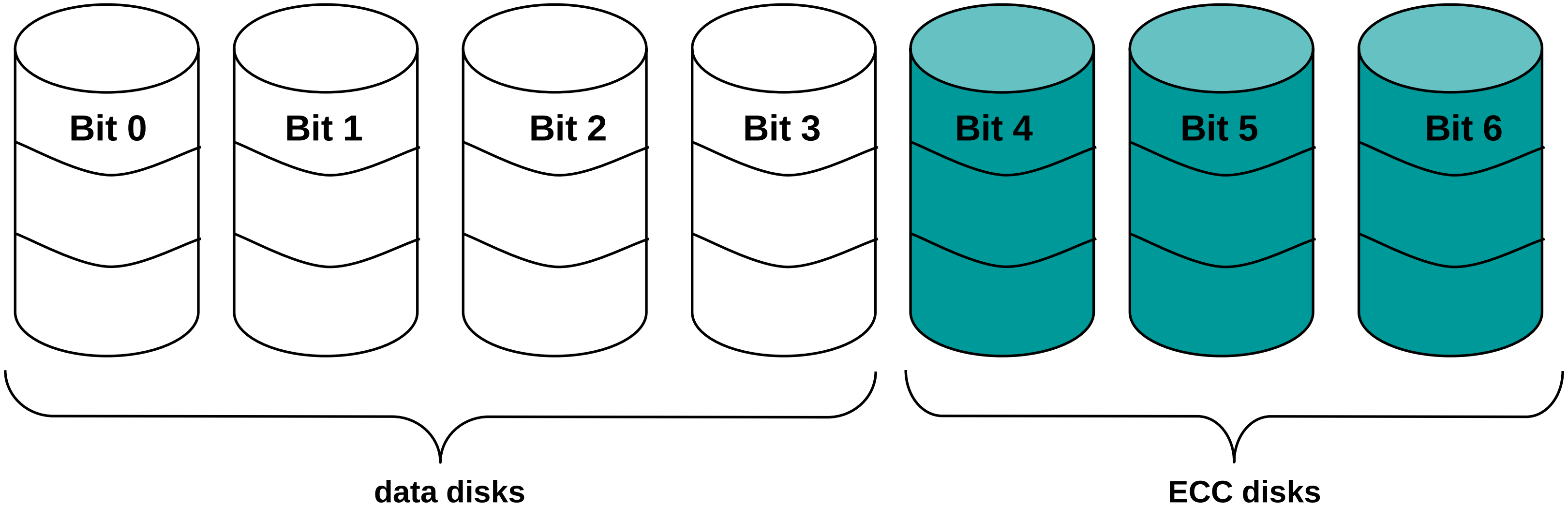
Parity and Hamming Code

- Detection and correction require more work
- Hamming codes can detect double bit errors and detect & correct single bit errors
- 7/4 Hamming Code
  - $h0 = b0 \otimes b1 \otimes b3$, 3 = 1+2, 5 = 1+4, 7 = 1+2+4
  - $h1 = b0 \otimes b2 \otimes b3$, 3 = 2+1, 6 = 2+4, 7 = 2+4+1
  - $h2 = b1 \otimes b2 \otimes b3$, 5 = 4+1, 6 = 4+2, 7 = 4+2+1
  - H0(<1101>) = 0
  - H1(<1101>) = 1
  - H2(<1101>) = 0
  - Hamming(<**1101**>) = <**b3, b2, b1, h2, b0, h1, h0**> = <**1100110**>

    7 6 5 4 3 2 1 = bit position
  - If a bit is flipped, e.g. <**1110110**>
  - Hamming(<1111>) = <h2, h1, h0> = <111> compared to <010>, <101> are in error. Error occurred in bit 5 = 4+1.

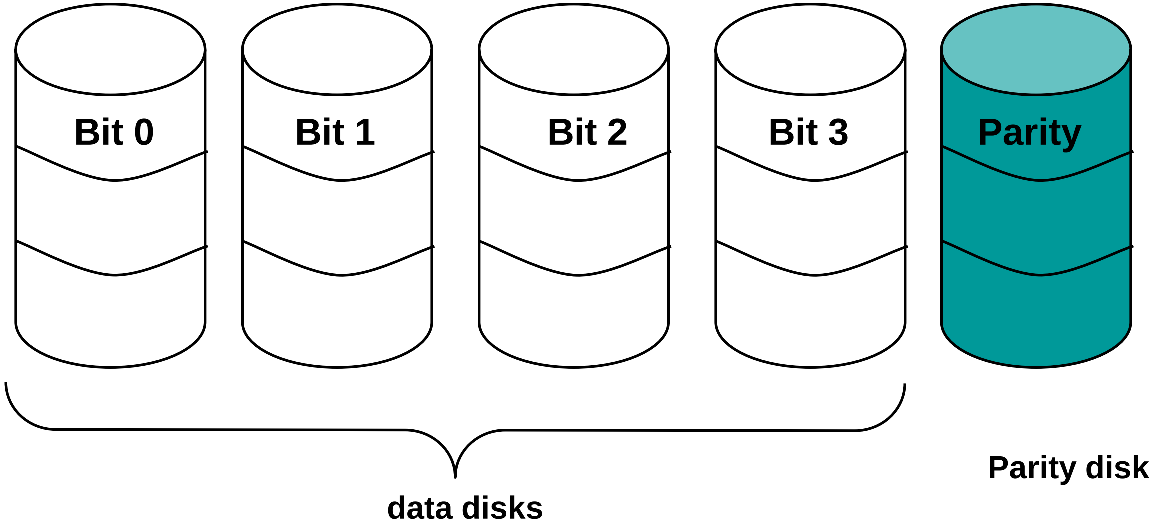
Raid Level 2

- Bit-level striping with Hamming (ECC) codes for error correction
- All 7 disk arms are synchronized and move in unison
- Complicated controller
- Single access at a time
- Tolerates only one error, but with no performance degradation



Raid Level 3

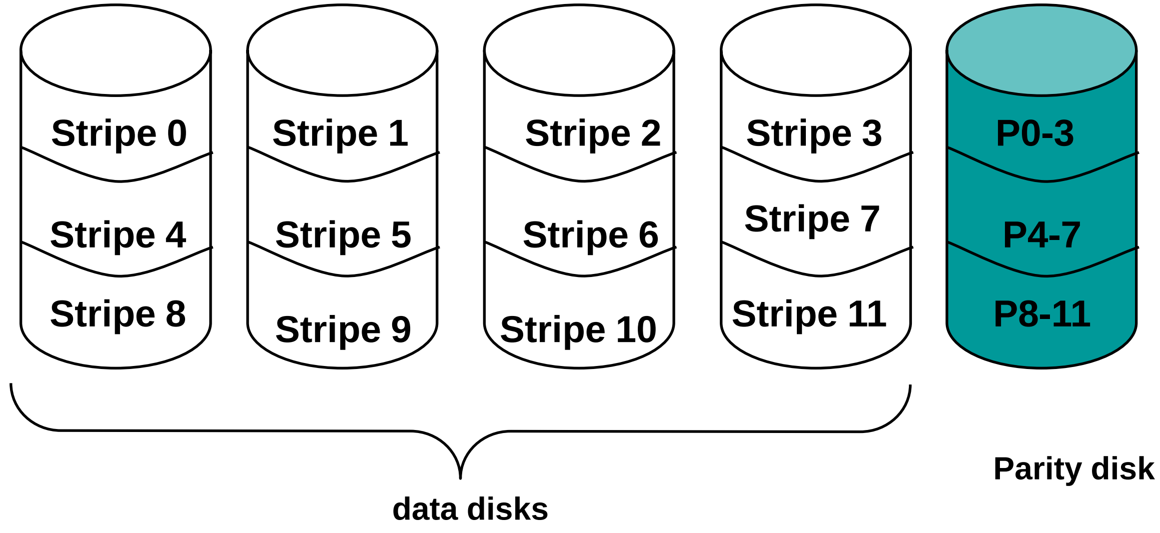
- Use a parity disk
  - Each bit on the parity disk is a parity function of the corresponding bits on all the other disks
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- On disk failure, read remaining disks plus parity disk to compute the missing data



Single parity disk can be used to detect and correct errors

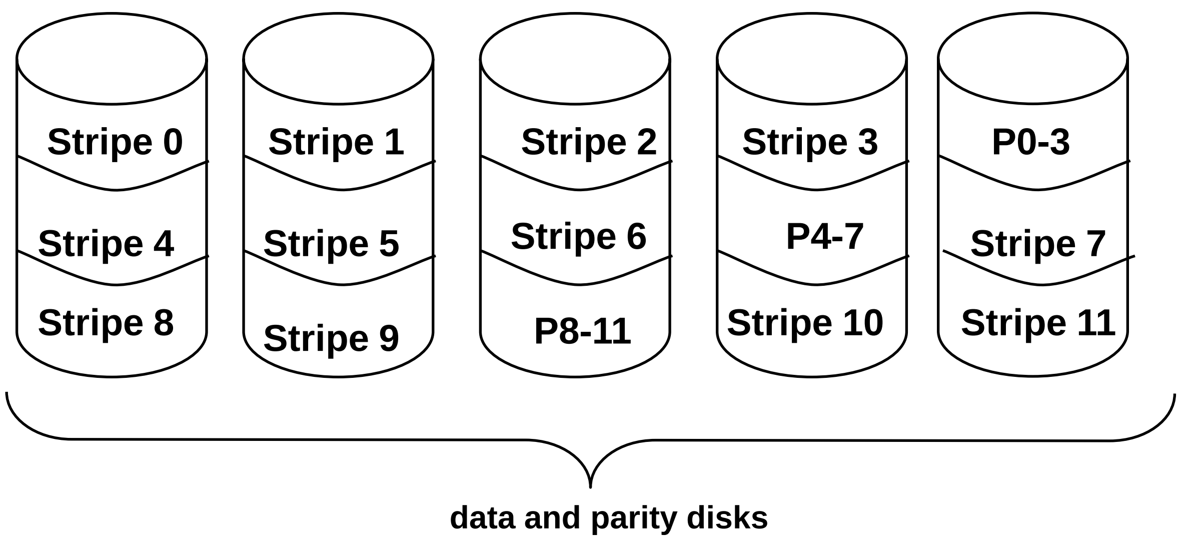
Raid Level 4

- Combines Level 0 and 3 – block-level parity with stripes
- A read accesses all the data disks
- A write accesses all data disks <u>plus</u> the parity disk
- Heavy load on the parity disk



Raid Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better read performance, large write performance
  - Reads can outperform SLEDs and RAID-0



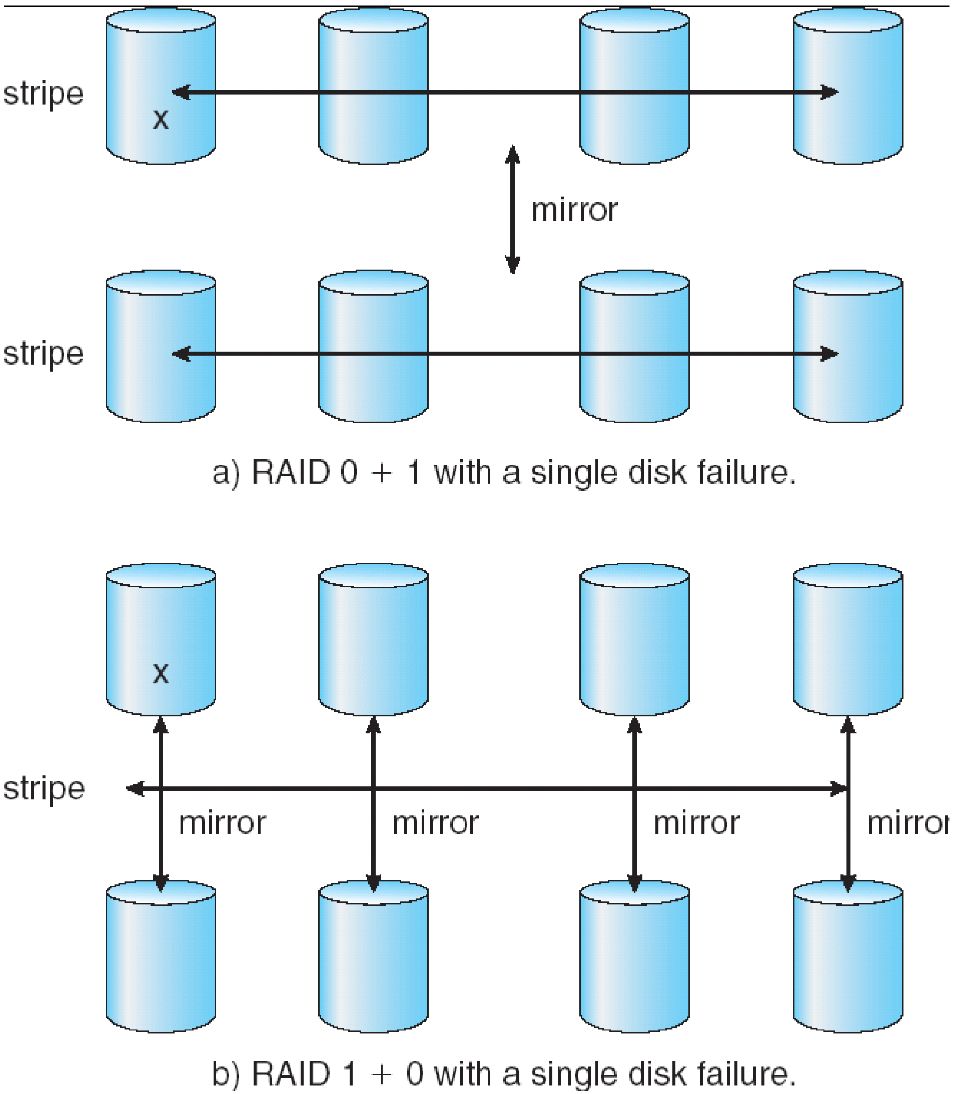
Raid Level 6

- Level 5 with an extra parity bit
- Can tolerate two failures
  - What are the odds of having two concurrent failures?
- May outperform Level-5 on reads, slower on writes

RAID 0+1 and 1+0

RAID 0+1
Stripes are mirrored

RAID 1+0
Mirrored pairs are striped



Stable Storage

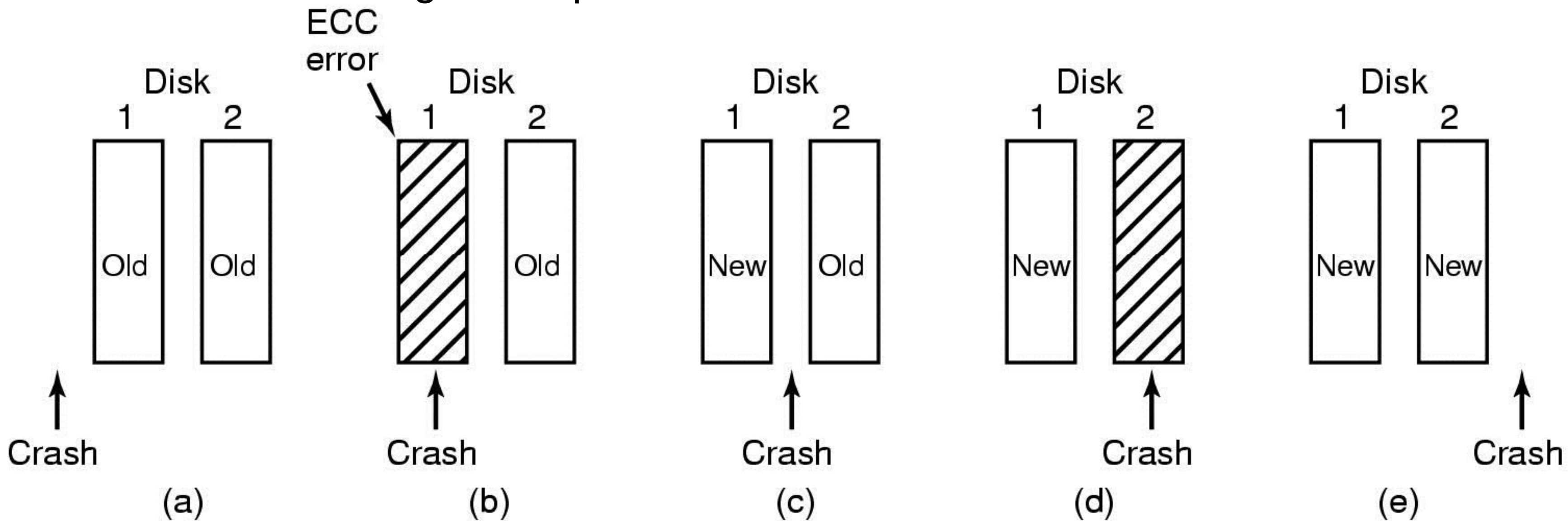
- Handling disk write errors:
  - Write lays down bad data
  - Crash during a write corrupts original data

- What we want to achieve? Stable Storage
  - When a write is issued, the disk either correctly writes data, or it does nothing, leaving existing data intact

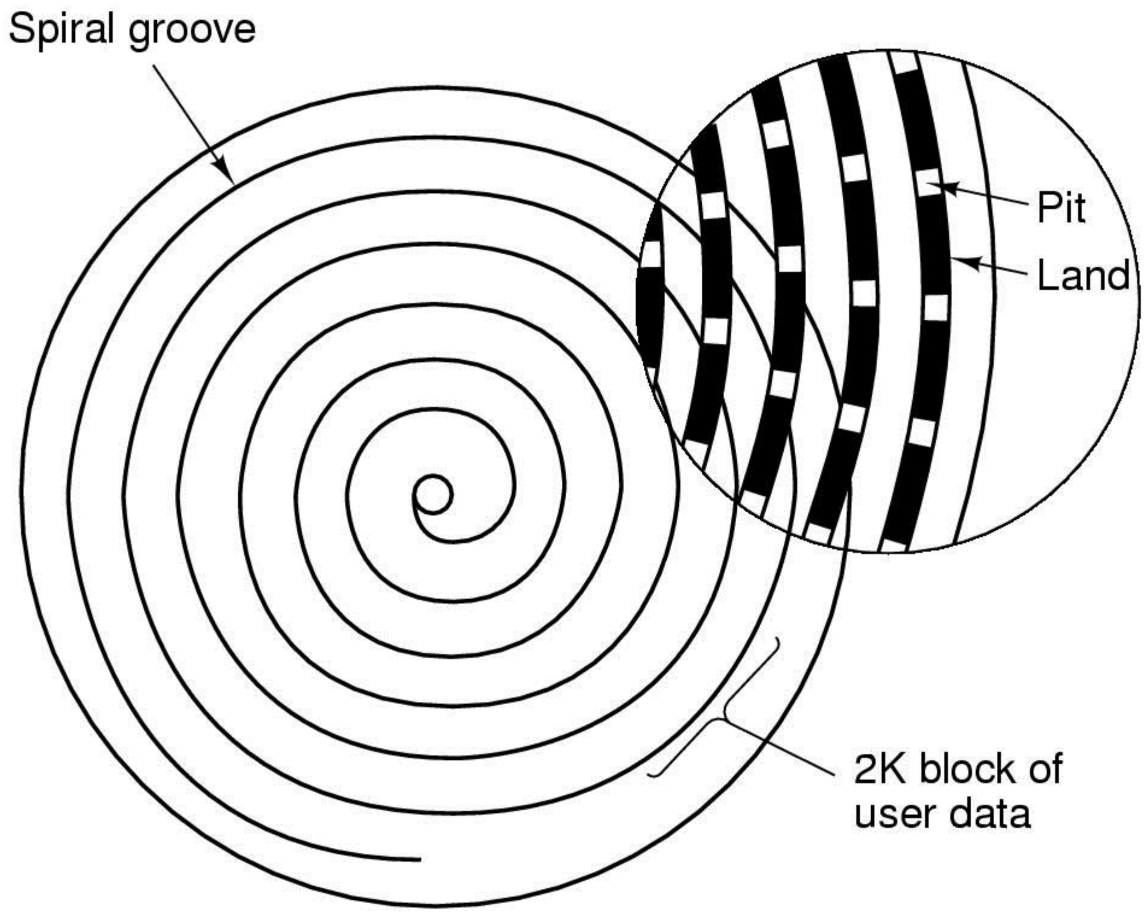
- Model:
  - An incorrect disk write can be detected by looking at the ECC
  - It is very rare that same sector goes bad on multiple disks
  - CPU is fail-stop

Approach

- Use 2 identical disks
  - corresponding blocks on both drives are the same
- 3 operations:
  - Stable write: retry on 1st until successful, then try 2nd disk
  - Stable read: read from 1st. If ECC error, then try 2nd
  - Crash recovery: scan corresponding blocks on both disks
    - If one block is bad, replace with good one
    - If both are good, replace block in 2nd with the one in 1st

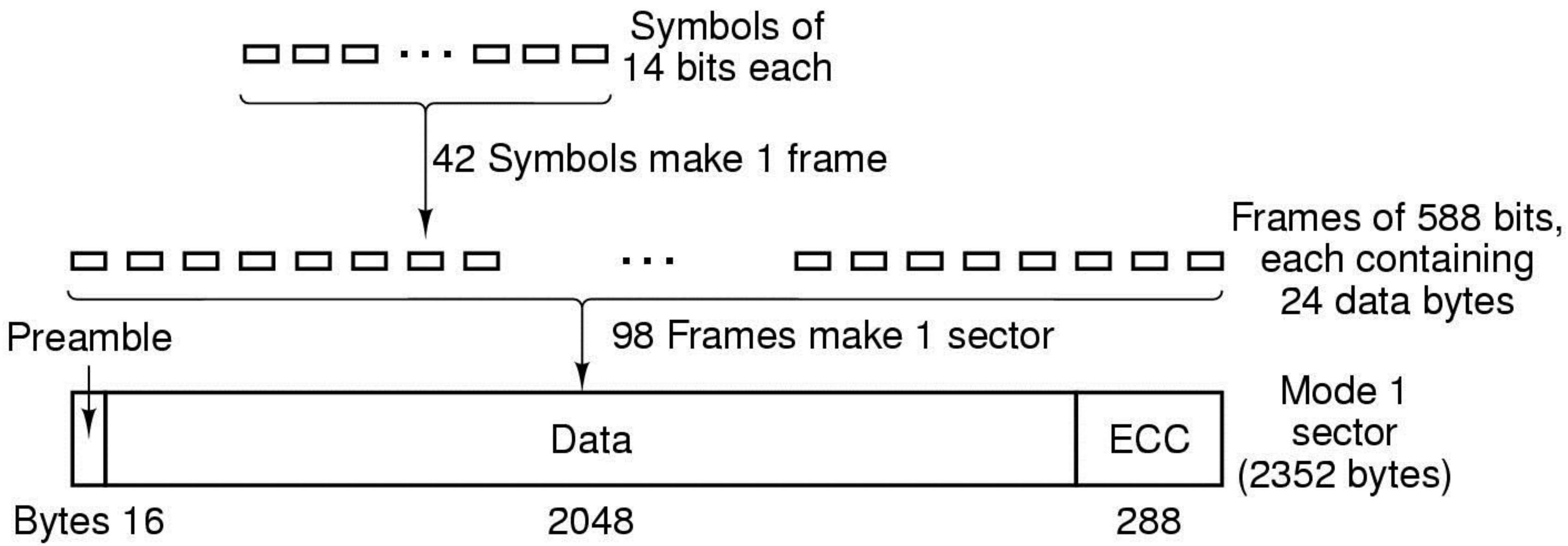


CD-ROMs



Spiral makes 22,188 revolutions around disk (approx 600/mm).
Will be 5.6 km long. Rotation rate: 530 rpm to 200 rpm

CD-ROMs



Logical data layout on a CD-ROM

start_process

```
static void start_process (void *exec_)
{
  struct exec_info *exec = exec_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  memset (&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;
  success = load (exec->file_name, &if_.eip, &if_.esp);

  /* Allocate wait_status. */
  if (success)
  {
      exec->wait_status = thread_current ()->wait_status
        = malloc (sizeof *exec->wait_status);
      success = exec->wait_status != NULL;
  }
```

start_process

```
/* Initialize wait_status. */
 if (success)
 {
     ..
 }
 /* Notify parent thread and clean up. */
 exec->success = success;
 sema_up (&exec->load_done);
 if (!success)
   thread_exit ();
 /* Start the user process by simulating a return from an
    interrupt, implemented by intr_exit (in
    threads/intr-stubs.S).  Because intr_exit takes all of its
    arguments on the stack in the form of a `struct intr_frame',
    we just point the stack pointer (%esp) to our stack frame
    and jump to it. */
 asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
}
```

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- Special assembly code instruction to access
- On (recent model) Intel machines:
  - 64 bit counter.
  - RDTSC instruction sets `%edx` to high order 32-bits, `%eax` to low order 32-bits
- Aside: Is this a security issue?

Cycle Counter Period

- Wrap Around Times for 550 MHz machine
  - Low order 32 bits wrap around every $2^{32} / (550 * 10^6)$ = 7.8 seconds
  - High order 64 bits wrap around every $2^{64} / (550 * 10^6)$ = 33539534679 seconds
    - 1065 years
- For 2 GHz machine
  - Low order 32 bits every 2.1 seconds
  - High order 64 bits every 293 years

Measuring with Cycle Counter

- Idea
  - Get current value of cycle counter
    - store as pair of unsigned's `cyc_hi` and `cyc_lo`
  - Compute something
  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Counter

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Emit assembly with `rdtsc` and two `movl` instructions

Closer Look at Extended ASM

```
asm("Instruction String"
       : Output List
       : Input List
       : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Instruction String
  - Series of assembly commands
    - Separated by "`;`" or "`\n`"
    - Use "`%%`" where normally would use "`%`"

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Output List
  - Expressions indicating destinations for values `%0`, `%1`, …, `%j`
    - Enclosed in parentheses
    - Must be *lvalue*
      - Value that can appear on LHS of assignment
  - Tag `"=r"` indicates that symbolic value (`%0`, etc.), should be replaced by a register

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List)
}
```

```
void access_counter
   (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Input List
  - Series of expressions indicating sources for values `%j+1`, `%j+2`, …
    - Enclosed in parentheses
    - Any expression returning value
  - Tag `"r"` indicates that symbolic value (`%0`, etc.) will come from register

Closer Look at Extended ASM

```
asm("Instruction String"
       : Output List
       : Input List
       : Clobbers List);
}
```

```
void access_counter
   (unsigned *hi, unsigned *lo)
{
   /* Get cycle counter */
   asm("rdtsc; movl %%edx,%0; movl %%eax,%1
        : "=r" (*hi), "=r" (*lo)
        : /* No input */
        : "%edx", "%eax");
}
```

- Clobbers List
  - List of register names that get altered by assembly instruction
  - Compiler will make sure doesn't store something in one of these registers that must be preserved across `asm`
    - Value set before & used after

Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

Timing With Cycle Counter

- Determine Clock Rate of Processor
  - Count number of cycles required for some fixed number of seconds

```
double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);
```

- Time Function P( )
  - First attempt: Simply count cycles for one execution of P

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```

Example – testClock.c

```
#include <stdio.h>
#include "clock.h"


int main()
{

  double cycles, Mhz;

  Mhz = mhz(1);
  start_counter();
  sleep(2);
  cycles = get_counter();

  printf("cycles = %f, MHz = %f, cycles/Mhz = %f\n", cycles, Mhz,
 cycles/Mhz);
  printf("elapsed time = %f seconds \n", cycles/(1.0e6*Mhz));

  return 0;
}
```

Processor Clock Rate ~= 2673.5 MHz
cycles = 5343976388.000000, MHz = 2673.526339, cycles/Mhz = 1998849.351153
elapsed time = 1.998849 seconds

- `cat /proc/cpuinfo -> 2.53 GHz.`

Measurement Pitfalls

- Overhead
  - Calling `get_counter()` incurs small amount of overhead
  - Want to measure long enough code sequence to compensate

Summary

- Read Ch. 1-10
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Virtual Memory (Ch. 9)
- Mass-Storage Structure (Ch. 10)
- Project #2 – System Calls and User-Level Processes