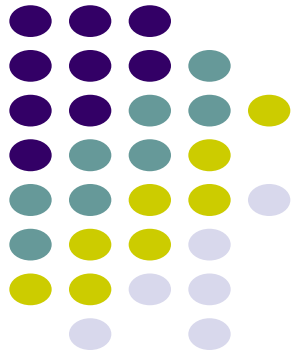
Final Review

ECE 469, April 29

Aravind Machiry



Final

- Online on Brightspace
- Tuesday, 06 May
- Time: 100 min (Approx)
- Available Period: 6:00 am - 11:59 pm

Threads

- Separate the concepts of a “thread of control” (PC, SP, registers) from the rest of the process (address space, resources, accounting, etc.)

- Modern OSes support two entities:
  - the *task* (process), which defines an address space, a resource container, accounting info
  - the *thread* (lightweight process), which defines a single sequential execution stream within a task (process)

Programming with Threads

- Flexible, but error-prone, since there no protection between threads
  - In C/C++,
    - automatic variables are private to each thread
    - global variables and dynamically allocated memory (malloc) are shared
- Need synchronization!

The need for synchronization!

- Cooperating processes may share data via
  - shared address space (code, data, heap) by using threads
  - Files
  - (Sending messages)
- What can happen if processes try to access shared data (address) concurrently?
  - Sharing bank account with sibling:

    At 3pm: If (balance > $10) withdraw $10
- How hard is the solution?

“Too much milk” Problem

Person A

1. Look in fridge: out of milk
2. Leave for Walmart
5. Arrive at Walmart
6. Buy milk
7. Arrive home

Person B

3. Look in fridge: out of milk
4. Leave for Walmart
8. Arrive at Walmart
9. Buy milk
10. Arrive home

- How to put in a locking mechanism?

How can we solve this?

- Root cause: Data Race

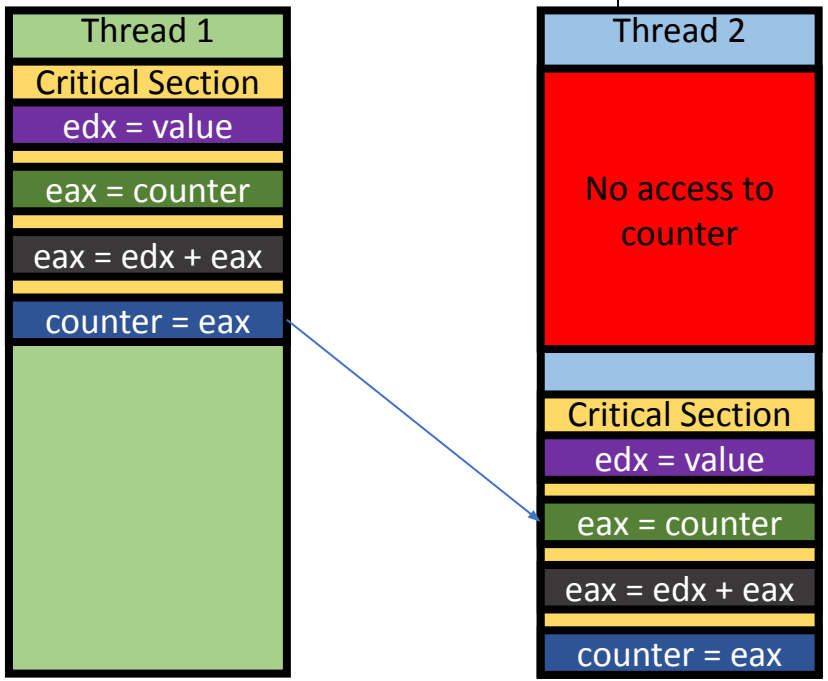
- A thread's execution result could be inconsistent if other threads intervene its execution…

- counter += value
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`

```
mov     0x20087b(%rip),%edx        # 0x201010 <value>
mov     0x20087d(%rip),%eax        # 0x201018 <counter>
add     %edx,%eax
mov     %eax,0x200875(%rip)        # 0x201018 <counter>
```

How can we prevent data races?

- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



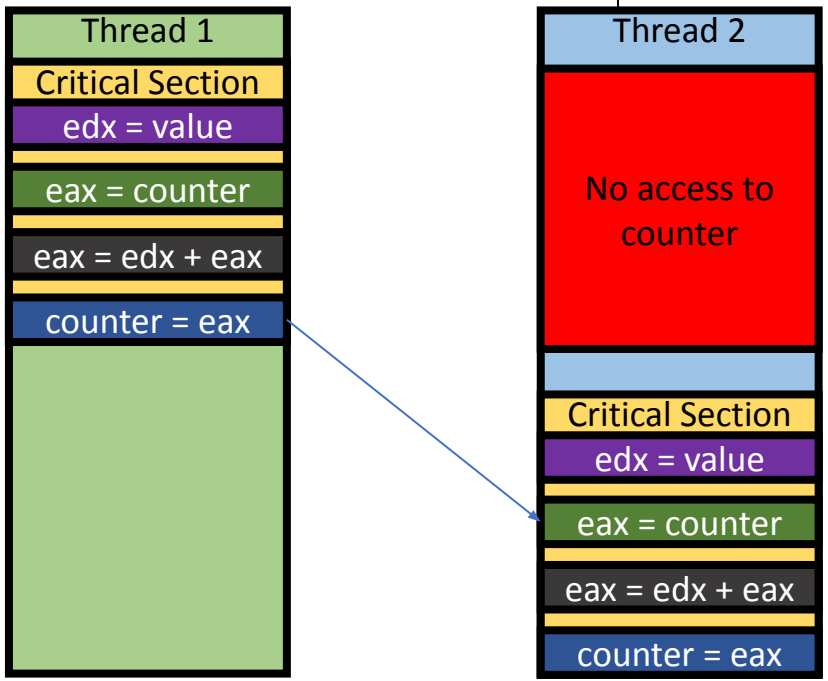
Recap: How can we prevent data races?

- *Critical section* – a section of code, or collection of operations, in which only one process shall be executing at a given time

- *Mutual exclusion (Mutex)* - mechanisms that ensure that only one person or process is doing certain things at one time (others are excluded)

Recap: How can we prevent data races?

- Mutual Exclusion / Critical Section
  - Combine multiple instructions as a chunk
  - Let only one chunk execution runs
  - Block other executions



Recap: Mutual Exclusion through locks

- Lock
  - Prevent others enter the critical section
- Unlock
  - Release the lock, let others acquire the lock

- counter += value
  - `lock()`
  - `edx = value;`
  - `eax = counter;`
  - `eax = edx + eax;`
  - `counter = eax;`
  - `unlock()`

Recap: Manual Spinlock (bad_lock)

- What will happen if we implement lock
  - As bad_lock / bad_lock?

- bad_lock
  - Wait until lock becomes 0 (loops if 1)
  - And then, set lock as 1
    - Because it was 0, we can set it as 1
  - Others must wait! **Can pass this if lock=0**

Sets lock=1 to block others

- bad_unlock
  - Just set *lock as 0

Sets lock=0 to release

```
void *
count_bad_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        bad_lock(&lock);
Critical    sched_yield();
Section     count += 1;
        bad_unlock(&lock);
    }
}
```

```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}

void
bad_unlock(volatile uint32_t *lock) {
    *lock = 0;
}
```

Recap: Why does bad_lock doesn't work?

- There is a room for race condition!

LOAD

```
mov    (%rdi),%eax
cmp    $0x1,%eax
je     0x400b60 <bad_lock>
movl   $0x1,(%rdi)
```

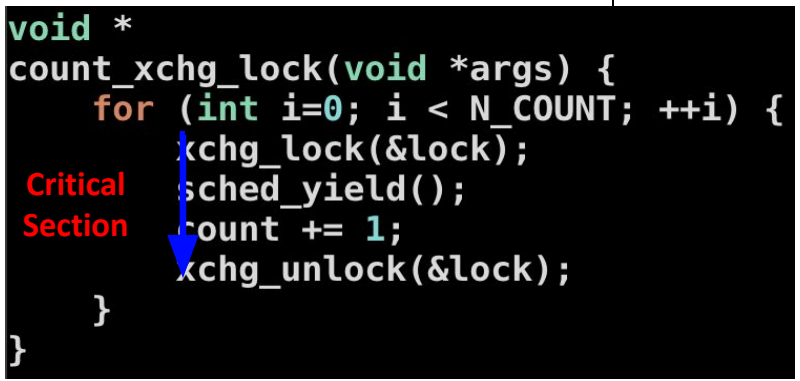
STORE

Race condition may happen

```
void
bad_lock(volatile uint32_t *lock) {
    while (*lock == 1);
    *lock = 1;
}
```

Recap: Lock using xchg

- xchg_lock
  - Use atomic ‘xchg’ instruction to
  - Load and store values atomically
  - Set value to 1, and compare ret
    - If 0, then you can acquire the lock
    - If 1, lock as 1, you must wait
- xchg_unlock
  - Use atomic ‘xchg’
  - Set value to 0
    - Do not need to check
    - You are the only thread that runs in the
    - Critical section..




```
void *
count_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

```
void
xchg_lock(volatile uint32_t *lock) {
    while(xchg(lock, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

Recap: Lock using test and set

- tts_xchg_lock
- Algorithm
  - Wait until lock becomes 0
  - After lock == 0
    - xchg (lock, 1)
    - This only updates lock = 1 if lock was 0
- Why xchg, why not *lock = 1 directly
  - while and xchg are not atomic
  - Load/Store must happen at
    - The same time!

```
void *
count_tts_xchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        tts_xchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

Critical Section

```
void
tts_xchg_lock(volatile uint32_t *lock) {
    while (1) {
        while(*lock == 1);
        if (xchg(lock, 1) == 0) {
            break;
        }
    }
}
```

Recap: Lock using cmpxchg_lock

- Cmpxchg_lock
  - Use cmpxchg to set lock = 1
    - Do not update if lock == 1
    - Only write 1 to lock if lock == 0

```
void *
count_cmpxchg_lock(void *args) {
    for (int i=0; i < N_COUNT; ++i) {
        cmpxchg_lock(&lock);
        sched_yield();
        count += 1;
        xchg_unlock(&lock);
    }
}
```

Critical Section

- Xchg_unlock
  - Use xchg_unlock to set lock = 0
  - Because we have 1 writer and
  - This always succeeds…

```
void
cmpxchg_lock(volatile uint32_t *lock) {
    while(cmpxchg(lock, 0, 1));
}

void
xchg_unlock(volatile uint32_t *lock) {
    xchg(lock, 0);
}
```

Recap: Using hardware features smartly

- backoff_cmpxchg_lock(lock)
- Try cmpxchg
  - If succeeded, acquire the lock.
  - If failed
    - Wait 1 cycle (pause) for 1st trial
    - Wait 2 cycles for 2nd trial
    - Wait 4 cycles for 3rd trial
    - …
    - Wait 65536 cycles for 17th trial..
    - Wait 65536 cycles for 18th trial..

```
void
backoff_cmpxchg_lock(volatile uint32_t *lock) {
    uint32_t backoff = 1;
    while(cmpxchg(lock, 0, 1)) {
        for (int i=0; i<backoff; ++i) {
            __asm volatile("pause");
        }
        if (backoff < 0x10000) {
            backoff <<= 1;
        }
    }
}
```

- https://en.wikipedia.org/wiki/Exponential_backoff

Recap: Summary

- Mutex is implemented with Spinlock
  - Waits until lock == 0 with a while loop (that's why it's called spin)
- Naïve code implementation
  - Load/Store must be atomic
- xchg is a "test and set" atom
  - Consistent, however, many c
- Lock cmpxchg is a "test and
  - But Intel implemented this a
- We can implement test-and-
  - Faster!
- We can also implement expo
  - Much faster! **Faster Than p**

```
./lock no
Running 30 threads each counting to 50 using no lock
Result:1400, Time taken: 3.913000 ms
./lock bad
Running 30 threads each counting to 50 using bad lock
Result:1465, Time taken: 2.256000 ms
./lock xchg
Running 30 threads each counting to 50 using xchg lock
Result:1500, Time taken: 853.585000 ms
./lock cmpxchg
Running 30 threads each counting to 50 using cmpxchg lock
Result:1500, Time taken: 12997.561000 ms
./lock tts
Running 30 threads each counting to 50 using tts lock
Result:1500, Time taken: 1.779000 ms
./lock backoff
Running 30 threads each counting to 50 using backoff lock
Result:1500, Time taken: 0.939000 ms
./lock mutex
Running 30 threads each counting to 50 using mutex lock
Result:1500, Time taken: 5.313000 ms
```

Semaphore

A semaphore is like an **integer**, with three differences:

When you create the semaphore, you can initialize its value to any integer, but after that the only operations you are **allowed to perform** are **increment** (increase by one) and **decrement** (decrease by one). *You cannot read the current value of the semaphore.*

When a thread **decrements** the semaphore, if the **result is negative**, the **thread blocks itself** and cannot continue until another thread increments the semaphore.

When a thread **increments** the semaphore, if there are **other threads waiting, one of the waiting threads gets unblocked**.

Recap: Producers/consumers using Semaphores

Producer

```
while (1) {

    produce an item;
    wait(EMPTY);
    lock();
    insert(item to pool);
    unlock();
    signal(FULL);
}
```

Consumer

```
While (1) {

    wait(FULL);
    lock();
    remove(item from pool);
    unlock();
    signal(EMPTY);
    consume the item;
}
```

Init: FULL = 0; **EMPTY = N**;

Is Semaphore good for producers/consumers?

Need to know the size of buffer!

How to accommodate dynamic pool size?

Release, wait and reacquire

Release lock, waiting for a condition and acquire lock

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        unlock(m);
        We need to wait for consumer
        lock(m);
    }
    insert(item to pool);
    unlock(m);
    tell a waiting consumer
}
```

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        unlock(m);
        We need to wait for producer
        lock(m);
    }
    remove(item from pool);
    unlock(m);
    tell a waiting producer
    consume the item;
}
```

Condition Variable (CV)

CV full; full->lock = m;
CV empty; empty->lock = m;

Producer

```
while (1) {

    produce an item;
    lock(m);
    if (!pool.has_space) {
        wait(full);
    }
    insert(item to pool);
    signal(empty);
    unlock(m);
}
```

unlock(m);
We need to wait for consumer
lock(m);

Consumer

```
While (1) {

    lock(m);
    if (pool.is_empty) {
        wait(empty);
    }
    remove(item from pool);
    signal(full);
    unlock(m);
    consume the item;
}
```

Condition Variable operations

```
wait(S) {
 unlock(s->lock);
 block and add into s->queue
 lock(s->lock);
}
```

```
signal(S) {
 unlock(s->lock);
 p = remove process from s->queue
 unblock process p
 lock(s->lock);
}
```

Condition Variables

- Wait (condition)
  - Block on “condition”
- Signal (condition)
  - Wakeup one or more processes blocked on “condition”
- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!

Wait free synchronization

- Design data structures in a way that allows safe concurrent accesses
  - no mutual exclusion (lock acquire & release) necessary
  - no possibility of deadlock

- Approach: use a single *atomic* operation to
  - commit all changes
    - move the shared data structure from one consistent state to another

Concurrency Bugs

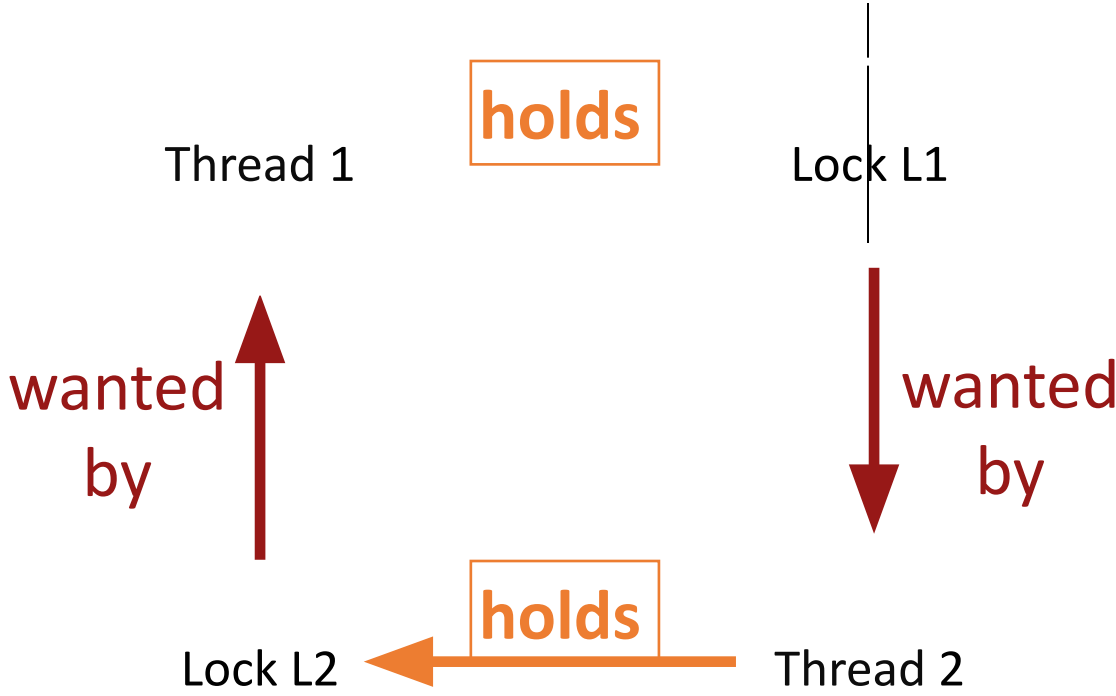
- TOCTOU:
  - Time of check to time of use

Concurrency Bugs

- Deadlock:
  - Two or more threads are waiting for the other to take some actions thus neither make any progress

Deadlocks

- Two or more threads are waiting for the other to take some actions thus neither make any progress




```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Modelling Deadlock

- Resources
  - Resource types R_1, R_2, . . ., R_m
    - *CPU cycles, memory space, I/O devices, mutex*
  - Each resource type R_i has W_i instances
  - *Preemptable:* can be taken away by scheduler, e.g. CPU
  - *Non-preemptable:* cannot be taken away, to be released voluntarily, e.g., mutex, disk, files, ...

- Each process utilizes a resource as follows:
  - request
  - use
  - release

Modelling Deadlock

- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Modelling Deadlocks Using Resource allocation graphs

- If graph contains no cycles ⇒ no deadlock

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

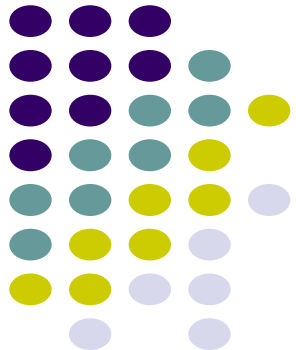
Necessary conditions for a deadlock

- *Mutual exclusion*
  - Each resource instance is assigned to exactly one process
- *Hold and wait*
  - Holding at least one and waiting to acquire more
- *No preemption*
  - Resources cannot be taken away
- *Circular chain of requests*

Handling deadlock

1. Ignore the problem
   - It is user's fault
   - used by most operating systems, including UNIX
2. Detection and recovery (by OS)
   - Fix the problem afterwards
3. Dynamic avoidance (by OS & programmer)
   - Careful allocation
4. Prevention (by programmer & OS)
   - Negate one of the four conditions

Demand Paging and Page Replacement Algorithms



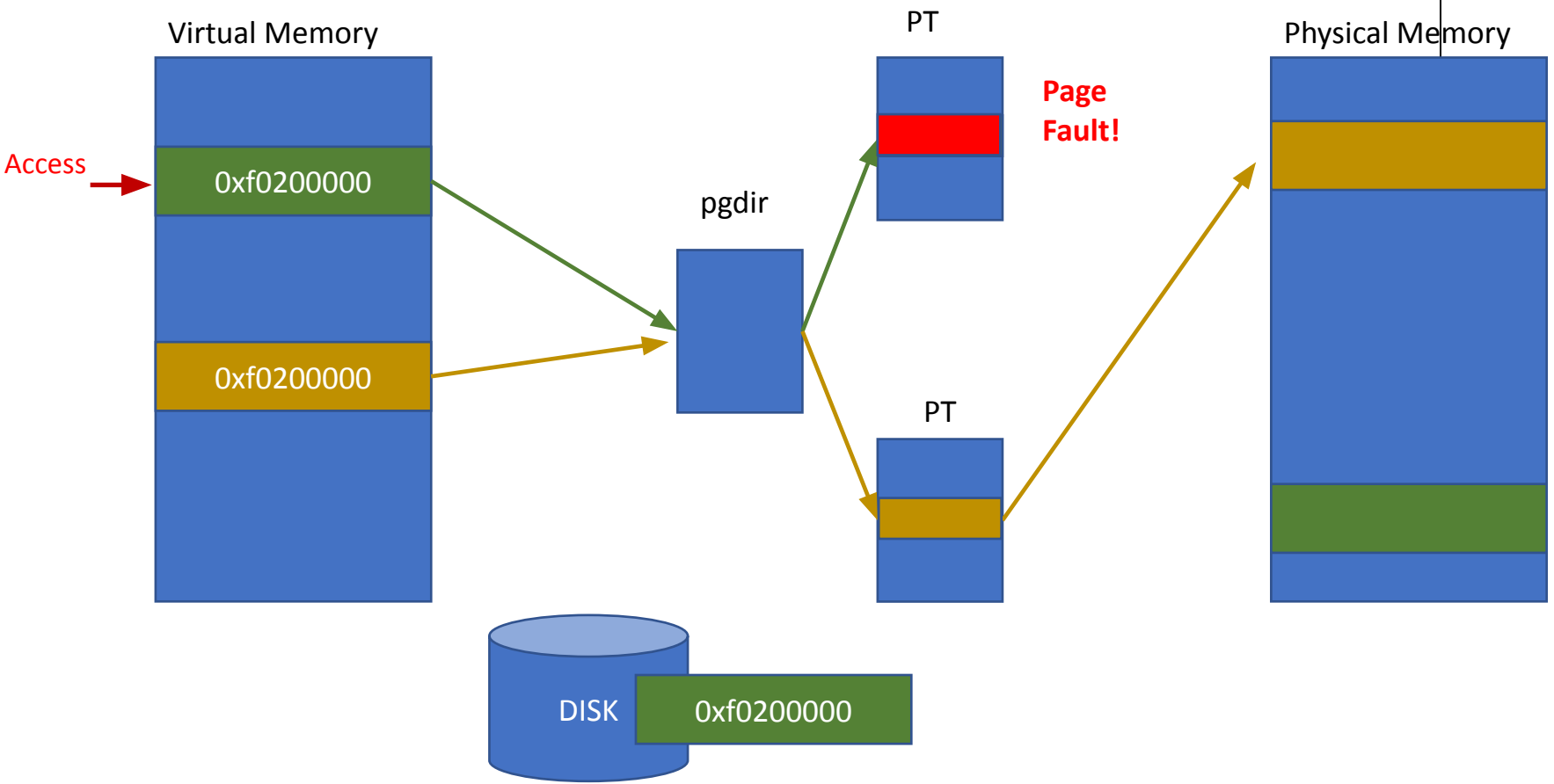
Handling low memory

- Suppose you have 8GB of main memory

- Can you run a program that its program size is 16GB?
  - Yes, you can load them part by part
  - This is because we do not use all of data at the same time

- Can your OS do this execution seamlessly to your application?

Memory Swapping - Removing a page



Swapping - Transparently load page from disk

- Page fault handler
  - Read CR2 (get address, `0xf0200000`)
  - Read error code
- If error code says that the fault is caused by non-present page and
- The faulting page of the current process is stored in the disk
  - Lookup disk if it swapped put 0xf0200000 of this environment (process)
    - This must be per process because virtual address is per-process resource
- Load that page into physical memory
- Map it and then continue!

Finding Least Useful Page is Hard

- Don’t know future!

Finding Least Useful Page is Hard

- Temporal Locality:
  - Past behavior is a good indication of future behavior! (e.g. LRU)

- Perfect (past) reference stream hard to get
  - Every memory access would need bookkeeping
  - Is this feasible (in software? In hardware?)

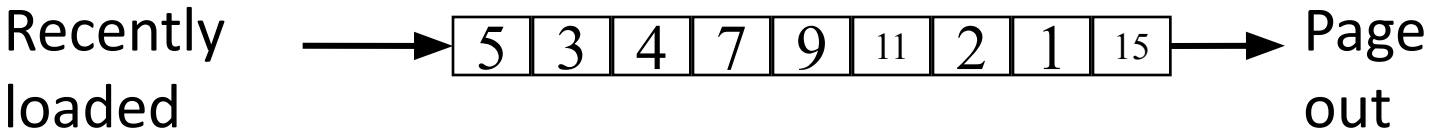
Finding Least Useful Page is Hard

- Temporal Locality:
  - Past behavior is a good indication of future behavior! (e.g. LRU)

- Perfect (past) reference stream hard to get
  - Every memory access would need bookkeeping
  - Is this feasible (in software? In hardware?)

- Minimize overhead
  - If no memory pressure, ideally no bookkeeping
  - In other words, make the common case fast (page hit)

First In First Out (FIFO)



- Algorithm
  - Throw out the oldest page
- Pros
  - Low-overhead implementation
- Cons
  - No frequency/no recency may replace the heavily used pages

Belady’s anomaly

Belady's anomaly: Laszlo Belady states that it is possible to have more page faults when increasing the number of page frames.

Previously, it was believed that an increase in the number of page frames would always provide the same number or fewer page faults.

Example (Page Faults in Red)

Page Requests – 3 frames

Total Page Faults: 9

	3	2	1	0	3	2	4	3	2	1	0	4
Frame 1	3	3	3	0	0	0	4	4	4	4	4	4
Frame 2		2	2	2	3	3	3	3	3	1	1	1
Frame 3			1	1	1	2	2	2	2	2	0	0

Example (Page Faults in Red)

Page Requests – 4 frames

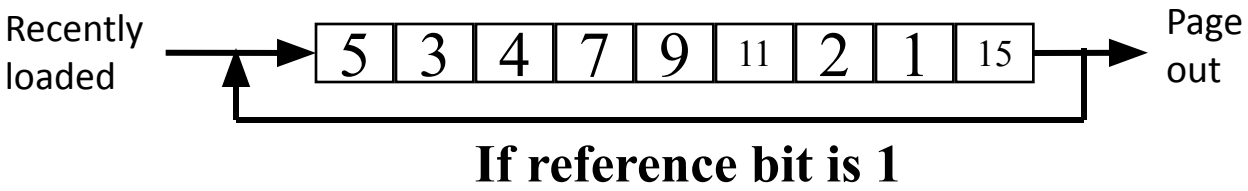
Total Page Faults: 10

	3	2	1	0	3	2	4	3	2	1	0	4
Frame 1	3	3	3	3	3	3	4	4	4	4	0	0
Frame 2		2	2	2	2	2	2	3	3	3	3	4
Frame 3			1	1	1	1	1	1	2	2	2	2
Frame 4				0	0	0	0	0	0	1	1	1

Optimal or MIN

- Algorithm (also called Belady's Algorithm)
  - Replace the page that won't be used for the **longest** time
- Pros
  - Minimal page faults (can you prove it?)
  - Used as an **off-line** algorithm for perf. analysis
- Cons
  - **No on-line** implementation
- What was the CPU scheduling algorithm of similar nature?

FIFO with Second Chance



- Algorithm
  - Check the reference-bit of the oldest page (first in)
  - If it is 0, then replace it
  - If it is 1, clear the referent-bit, put it to the end of the list, and continue searching
- Pros
  - Fast
  - Frequency □ do not replace a heavily used page
- Cons
  - The worst case may take a long time

Least Recently Used (LRU)

- Algorithm
  - Replace page that hasn’t been used for the longest time

- Advantage: with locality, LRU approximates Optimal

Least Recently Used (LRU)

- What hardware mechanisms are required to implement LRU?

Approximate LRU

Initial	Interval 1	Interval 2	Interval 3	Interval 4	Interval 5
00000000	00000000	00000000	10000000	01000000	10100000
00000000	00000000	10000000	01000000	10100000	01010000
00000000	10000000	11000000	11100000	01110000	01110000
00000000	00000000	00000000	00000000	10000000	01000000

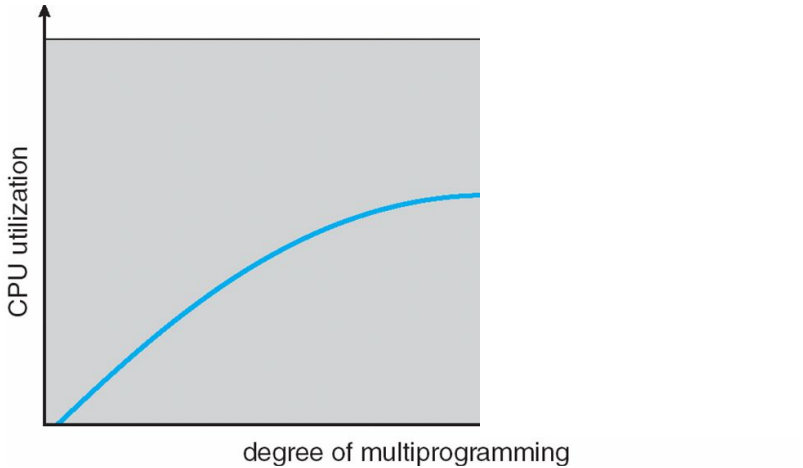
- Algorithm
  - At regular interval, OS shifts reference bits (in PTE) into counters (and clear reference bits)
  - Replacement: Pick the page with the "smallest counter"
- How many bits are enough?
  - In practice 8 bits are quite good
- Pros: **Require one reference bit, small counter/page**
- Cons: **Require looking at many counters (or sorting)**

Enhanced FIFO with 2nd Chance

Same as the basic FIFO with 2nd chance, except that it considers both (reference bit, modified bit)

Ref, Mod	Needed Soon?	Replacement Cost?	Preference
0, 0	Unlikely	Low (Drop the page)	😍
0, 1	Unlikely	High (Write to disk)	😄
1, 0	Likely	Low (Drop the page)	😊
1, 1	Likely	High (Write to disk)	😟

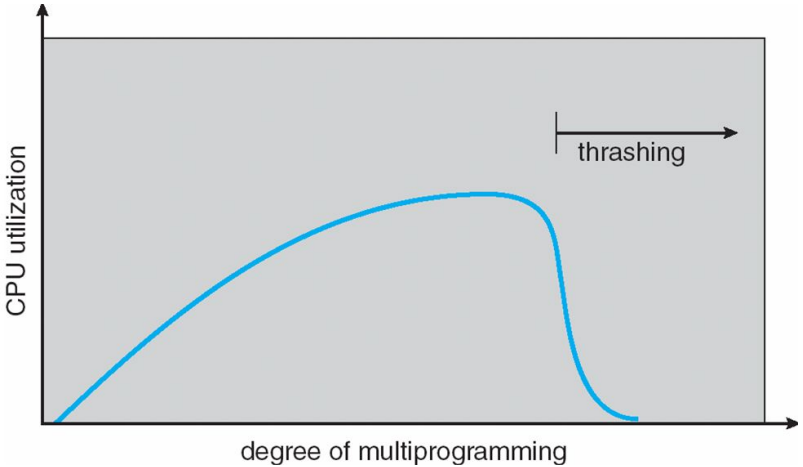
Increasing multiprogramming increases CPU utilization!!?



Thrashing can lead to vicious cycle

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming (actual behavior of early paging systems)
  - another process added to the system
  - page fault rate goes even higher

Thrashing!!



What causes Thrashing!?

- The system does not know it has taken more work than it can handle
- Virtual memory bites back!
- Mitigating Thrashing:
  - Run fewer programs.
  - Dropping or degrading a course if taking too many than you can handle ☺️

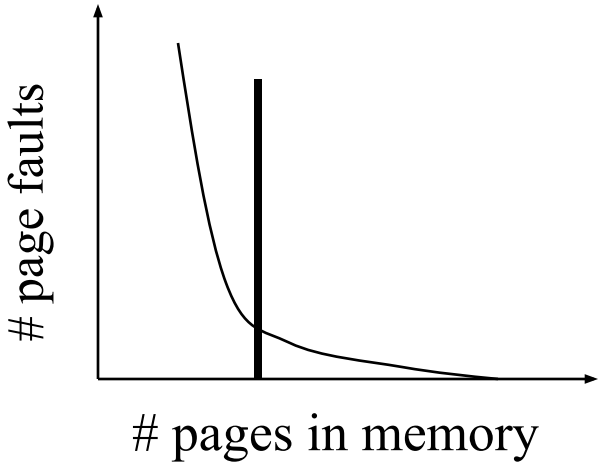
Intuitively, what to do about thrashing?

- If a single process's locality too large for memory, what can OS do?
  - e.g., pin most data (hotter data) in memory, sacrifice the rest

- If the problem arises from the sum of several processes?
  - Figure out how much memory each process needs – "locality"
  - What can we do?
    - Can limit effects of thrashing using local replacement
    - Or, bring a process' working set before running it
    - Or, wait till there is enough memory for a process's need

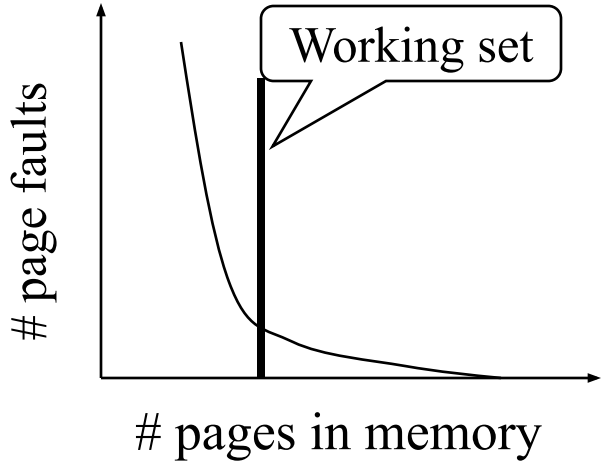
Key Observation

- Locality in memory references
  - Spatial and temporal
- Want to keep a set of pages in memory that would avoid a lot of page faults
  - “Hot” pages
- Can we formalize it?



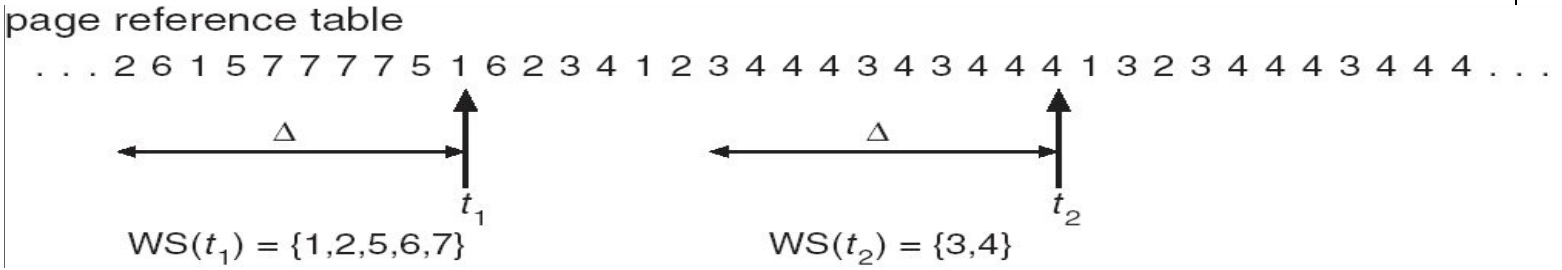
Working Set Model – by Peter Denning (Purdue CS head, 79-83)

- An informal definition:
  - Working set: The collection of pages that a process is working within a time interval, and which must thus be resident if the process is to avoid thrashing



- But how to turn the concept/theory into practical solutions?
  1. Capture the working set
  2. Influence the scheduler or replacement algorithm

Working Sets

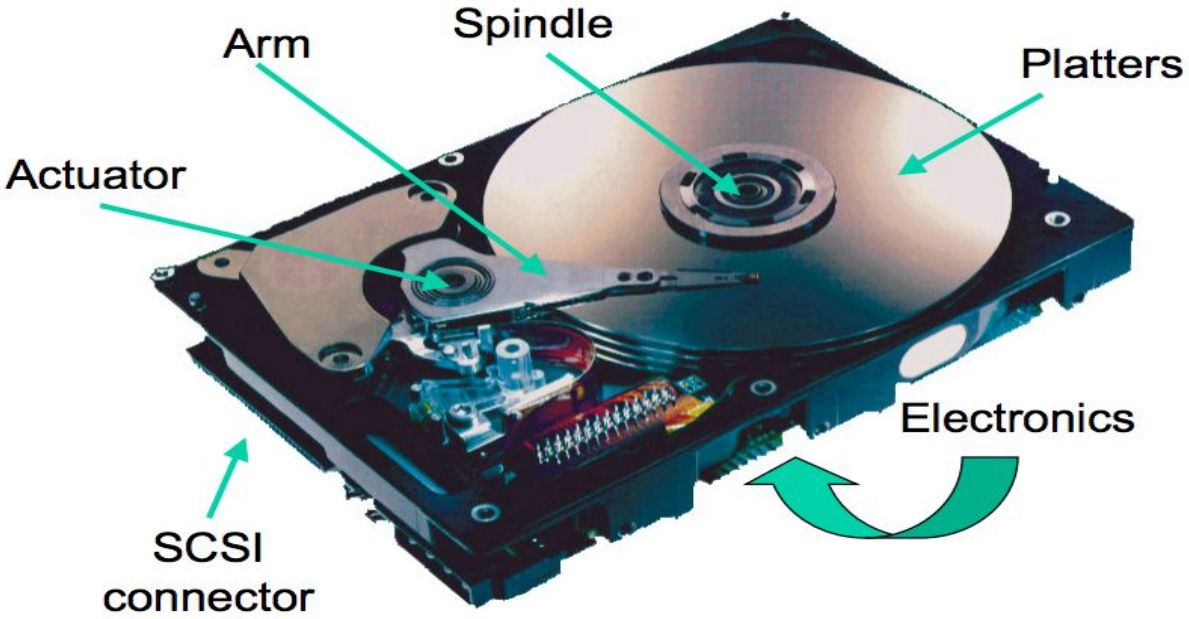


- The <u>working set size</u> is *num of* pages in the working set
  - the number of pages touched in the interval [t-Δ+1..t].
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory.

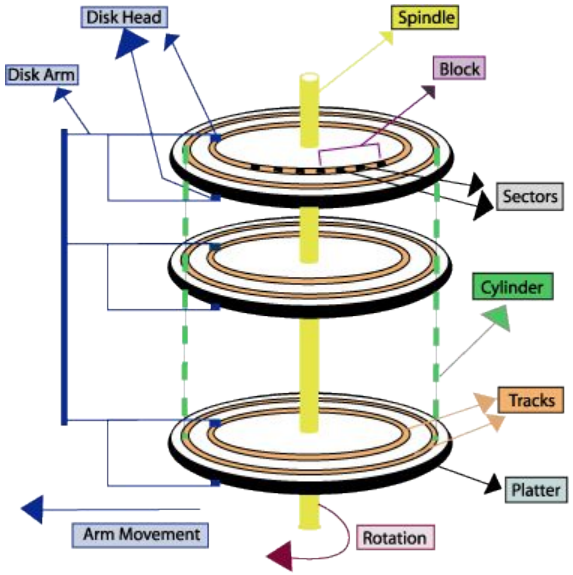
Storage Technologies

- Tapes
- Magnetic Disks
- Flash Memory

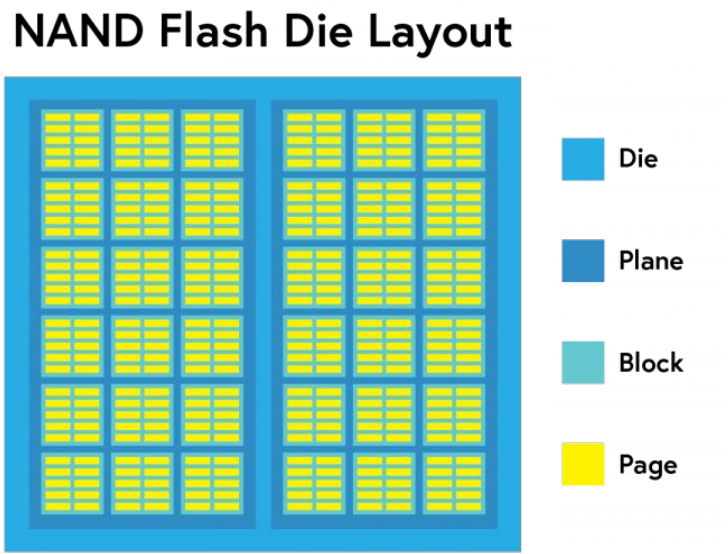
What is inside a disk drive?



More Deeper!


Disk Head
Spindle
Disk Arm
Block
Sectors
Cylinder
Tracks
Platter
Arm Movement
Rotation

Flash Memory



Flash Memory

- Page Size: 512 - 4K bytes.

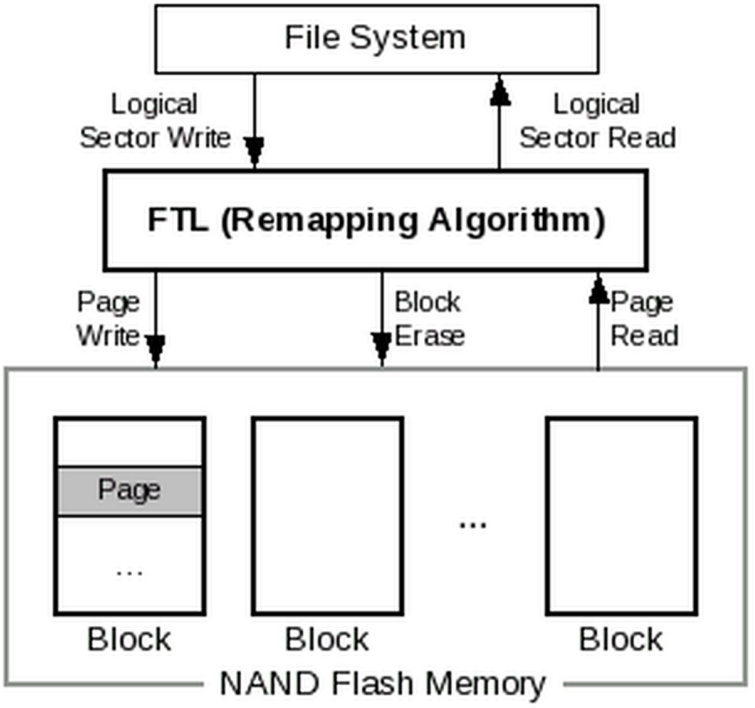
- Write needs complete erasure:
  - Erase before write.
    - Cannot go cell-wise from 0 -> 1

  - Any cells that have been set to 0 (written to) by programming can only be reset to 1 by erasing the entire block.

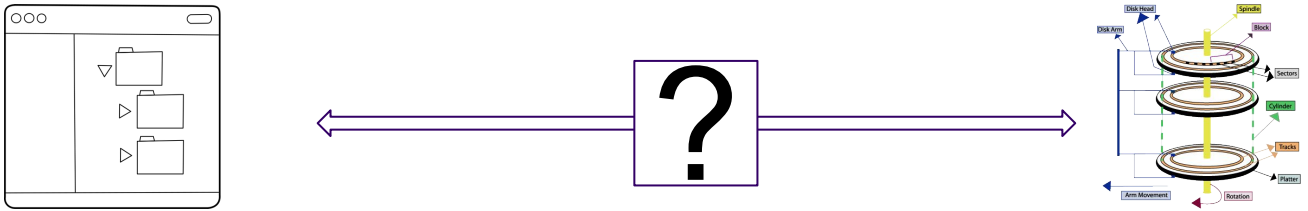
- Writes >> Reads

SSD : Flash Translation Layer

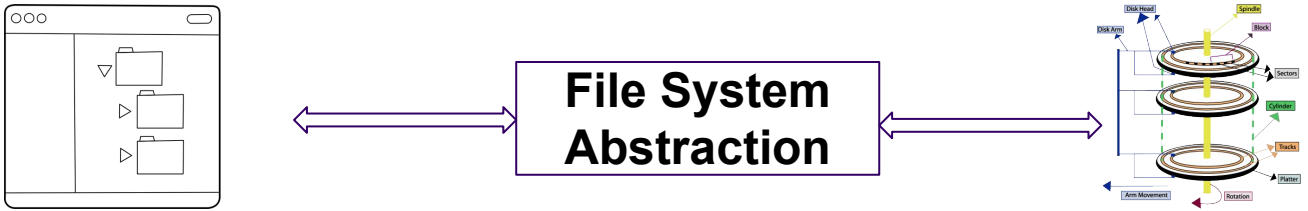
- Maps logical blocks to real blocks.
- Hides erase before write.
- Maintains free blocks.



How do we access storage?



File System Abstraction



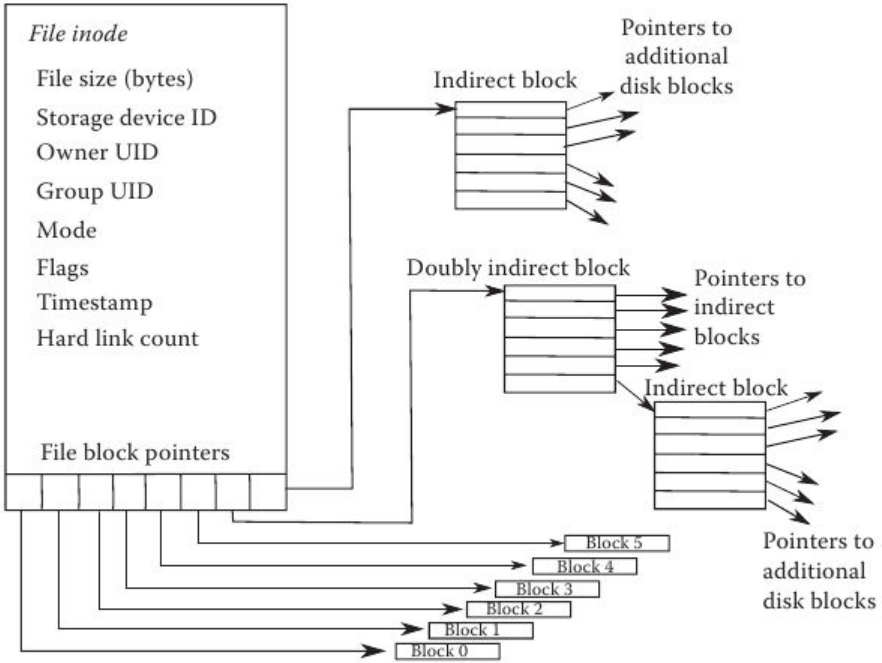
What does a File System Store about a file?

- File Metadata:
  - File Name.
  - Permissions (read or write).
  - owner/group.
  - Type of file.
  - List of disk blocks that contain data of the file.
  - Several others...

inode: Structure of metadata

- Indirect Node.
- Each file has one or more inodes corresponding to it.
- Where are inodes stored?
  - Disk
- A portion of disk is dedicated to inodes.

File inode

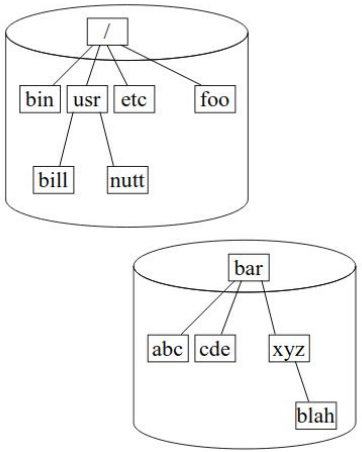

File inode
File size (bytes)
Storage device ID
Owner UID
Group UID
Mode
Flags
Timestamp
Hard link count
File block pointers
Indirect block
Pointers to additional disk blocks
Doubly indirect block
Pointers to indirect blocks
Indirect block
Pointers to additional disk blocks
Block 5
Block 4
Block 3
Block 2
Block 1
Block 0

Directory Permissions

dir permissions	Octal	del rename create files	dir list	read file contents	write file contents	cd dir	cd subdir	subdir list	access subdir files
---	0								
-W-	2								
R--	4		only file names (*)						
RW-	6		only file names (*)						
--X	1			X	X	X	X	X	X
-WX	3	X		X	X	X	X	X	X
R-X	5		X	X	X	X	X	X	X
RWX	7	X	X	X	X	X	X	X	X

Mounting File System

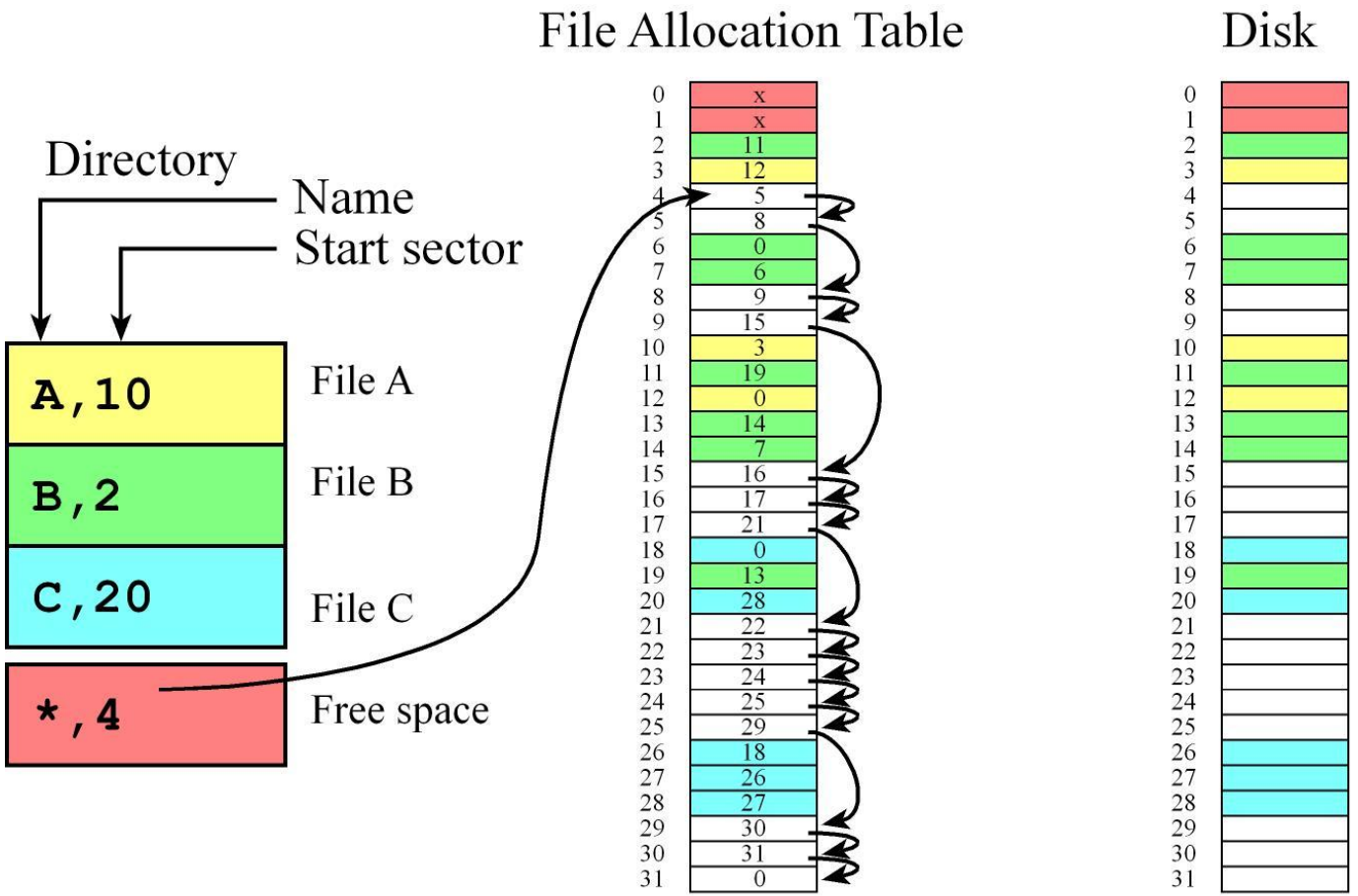
- Mapping from a name in a filesystem to root of another filesystem
- Allows users to manage multiple filesystems through a single filesystem.



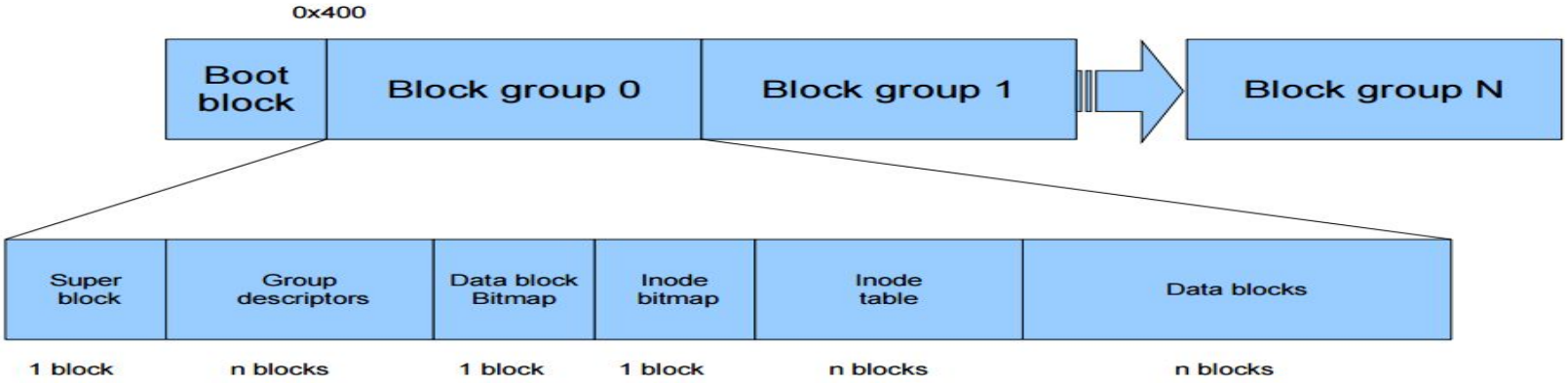
File Allocation Table (FAT)

- Simple.
- Easy to implement.
- Still used in Phones and Thumb drives.

- Key data structure: File Allocation Table
  - List of all disk blocks.
  - File: Linked list of blocks.


File Allocation Table
Disk
Directory
Name
Start sector
A,10
B,2
C,20
*,4
File A
File B
File C
Free space
0 x
1 x
2 11
3 12
4 5
5 8
6 0
7 6
8 9
9 15
10 3
11 19
12 0
13 14
14 7
15 16
16 17
17 21
18 0
19 13
20 28
21 22
22 23
23 24
24 25
25 29
26 18
27 26
28 27
29 30
30 31
31 0

EXT2 File System



Exploring EXT2

gzip -d disk.img.gz

$ ls -lh disk.img
-rw-rw-r-- 1 machiry machiry 2.0M Apr 6 2020 disk.img

mkdir /tmp/myfs

# Mount the file system
$ sudo mount disk.img /tmp/myfs

$ mount | grep myfs
/home/machiry/Desktop/lec22/disk.img on /tmp/myfs type ext2 (rw,relatime)

Directory Structure

```
$ ls -lh /tmp/myfs/
total 14K
-rw-r--r-- 1 root root   10 Apr  6  2020 aa
lrwxrwxrwx 1 root root    2 Apr  6  2020 aa-symlink -> aa
drwx------ 2 root root  12K Apr  2  2020 lost+found
drwxr-xr-x 2 root root 1.0K Apr  6  2020 testdir
```

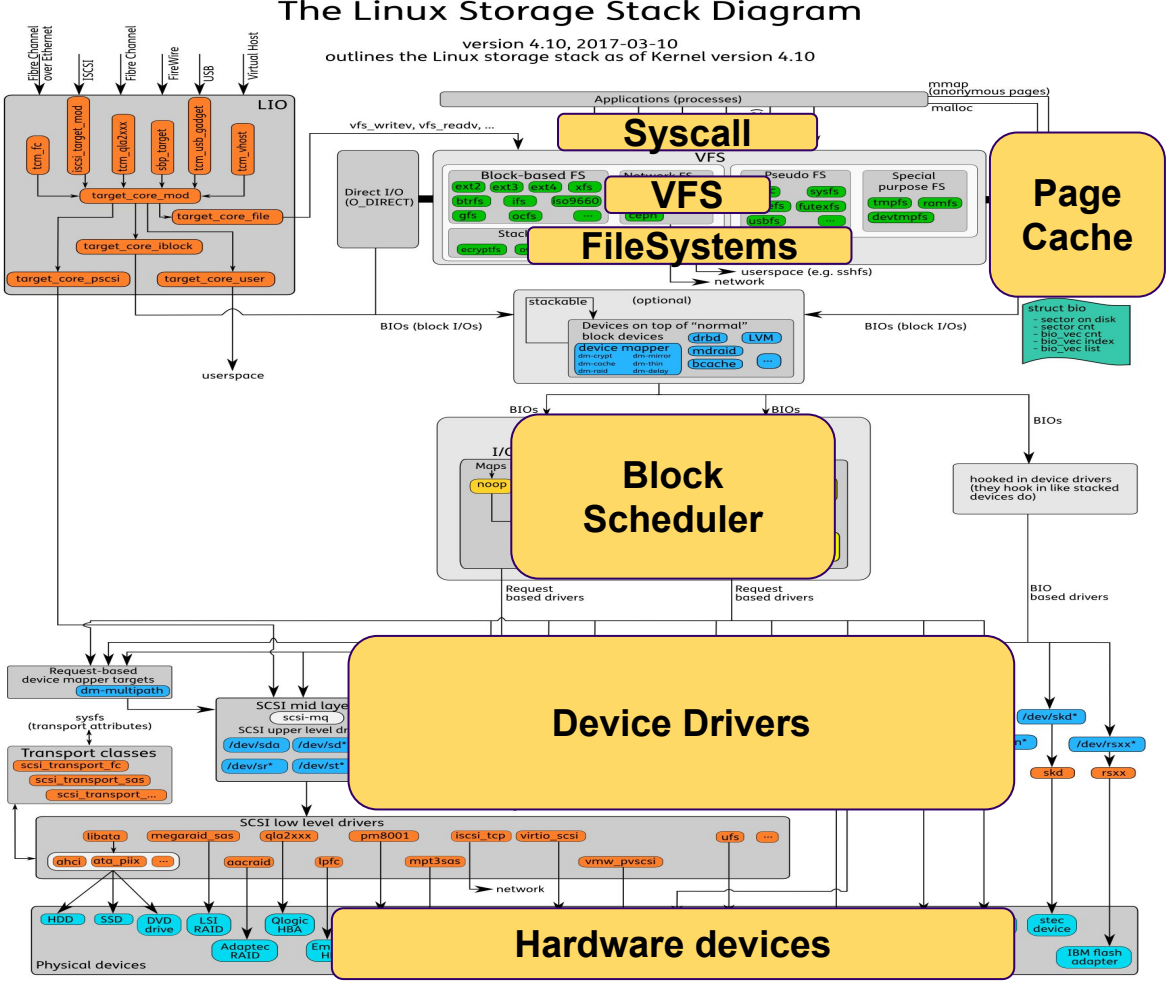
Checking Super Block

```
$ df -h | grep myfs
/dev/loop6                      2.0M   23K  1.9M   2% /tmp/myfs

$ sudo dumpe2fs /dev/loop6
dumpe2fs 1.45.5 (07-Jan-2020)
Filesystem volume name:   <none>
Last mounted on:          /tmp/myfs
Filesystem UUID:          6430eccd-11d0-4ea9-bd26-ce6e946dc02b
Filesystem magic number:  0xEF53
…
Inode count:              256
Block count:              2048
Reserved block count:     102
Free blocks:              1988
Free inodes:              242
First block:              1
Block size:               1024
Fragment size:            1024
Reserved GDT blocks:      7
Blocks per group:         8192
Fragments per group:      8192
Inodes per group:         256
Inode blocks per group:   32

Group 0: (Blocks 1-2047)
  Primary superblock at 1, Group descriptors at 2-2
  Reserved GDT blocks at 3-9
  Block bitmap at 10 (+9)
  Inode bitmap at 11 (+10)
  Inode table at 12-43 (+11)
  1988 free blocks, 242 free inodes, 3 directories
  Free blocks: 59-512, 514-2047
  Free inodes: 15-256
```

Only one block group



THOMAS KRENN®

Handling loss of data

- Accidental or malicious deletion of data?
  - Backup
- Media (disk) failure?
  - Data Replication (e.g., RAID)
- System crash during file system modifications?
  - **Crash Recovery**

Traditional Solution: fsck

- FSCK: “file system checker”
- When system boots:
  - Make multiple passes over file system, looking for inconsistencies
    - e.g., inode pointers and bitmaps, directory entries and inode reference counts
  - Either fix automatically or punt to admin
  - How to recover?

Traditional Solution: fsck

- Main problem with fsck: Performance
  - Sometimes takes hours to run on large disk volumes

Solution 2: Logging file system updates

- We need to ensure a “copy” of consistent state can always be recovered

- Either the old consistent state (before updates)
- Undo Log
  - Make a copy of the old state to a different place
  - Update the current place

- Or the new consistent state (after updates)
- Redo Log
  - Write to a new place, leave the old place intact

Redo Log

- Idea: Write something down to disk at a different location from the data location
  - Called the “write ahead log” or “journal”
- When all data is written to redo log, write it back to the data location, and then delete the data on redo log

- When crash occurs, look through the redo log and see what was going on
  - Replay complete data, discard incomplete data
  - The process is called “recovery”

Journaling File Systems

- Basic idea
  - update metadata, or all data, *transactionally*
    - *“all or nothing”*
    - *Failure atomicity*
  - if a crash occurs, you may lose a bit of work, but the disk will be in a consistent state
    - more precisely, you will be able to quickly get it to a consistent state by using the transaction log/journal – rather than scanning every disk block and checking sanity conditions

Journal

- Journal: an append-only file containing log records
  - <start t>
    - transaction t has begun
  - <t,x,v>
    - transaction t has updated block x and its new value is v
      - Can log block "diffs" instead of full blocks
      - Can log *operations* instead of data (operations must be idempotent and undoable)
  - <commit t>
    - transaction t has committed – updates will survive a crash
    - Only after the commit block is written is the transaction final
    - The commit block is a single block of data on the disk
- Committing involves writing the records – **the home data doesn't need to be updated at this time**

If a crash occurs

- Open the log and parse
  - <start>, data, <commit> => committed transactions
  - <start>, ~~no <commit>~~ => uncommitted transactions
- Redo committed transactions
  - Re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any positive number of times with the same result.
- Undo uncommitted transactions
  - Undo updates from all uncommitted transactions
  - Write “compensating log records” to avoid work in case we crash during the undo phase

Ext3 and Journaling

- Journal location
  - EITHER on a separate device partition
  - OR just a “special” file within ext2

- Three separate modes of operation:
  - Data: All data is journaled
  - Ordered, Writeback: Just metadata is journaled

Data Journaling Mode

- Same example: Update Inode (I), Bitmap (B), Data (D)
- First, write to journal:
  - Transaction begin (Tx begin)
  - Transaction descriptor (info about this Tx)
  - I, B, and D blocks (in this example)
  - Transaction end (Tx end)
- Then, “checkpoint” data to fixed ext2 structures
  - Copy I, B, and D to their fixed file system locations
- Finally, free Tx in journal
  - Journal is fixed-sized circular buffer, entries must be periodically freed

Metadata only journaling: Writeback mode

- Writeback mode
  - Just journals metadata
  - Data is not journaled. Writes data to final location directly
  - Better performance than data journaling (data written once)
  - The contents might be written at any time (before or after the journal is updated)
- Problems?
  - If a crash happens, metadata can point to old or even garbage data!

Metadata only journaling: Ordered mode

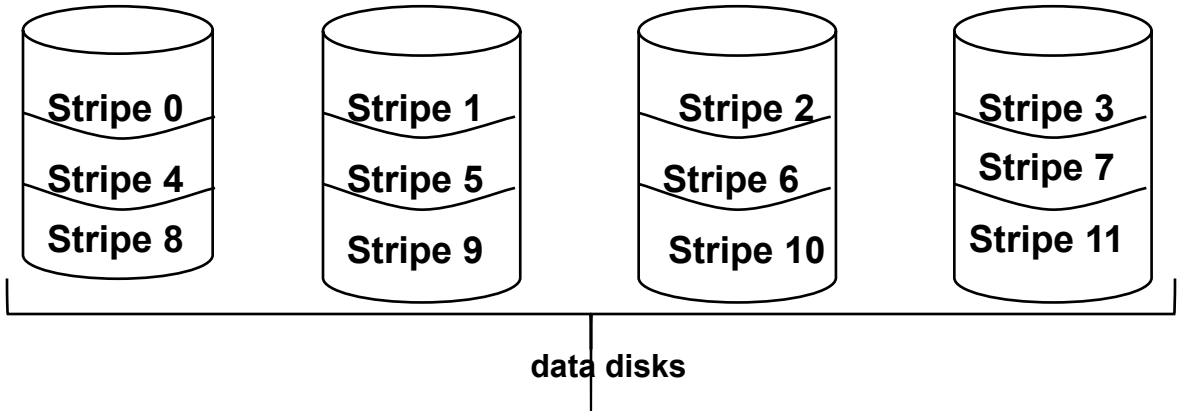
- Ordered mode
  - Only metadata is journaled, file contents are not (like writeback mode)
  - But file contents guaranteed to be written to disk before associated metadata is marked as committed in the journal
  - Default ext3 journaling mode

RAID

- Mirroring or shadowing (for reliability)
  - Local disk consists of 2 physical disks in parallel
  - Every write performed on both disks
    - Can read from either disk
    - Probability of both fail at the same time?

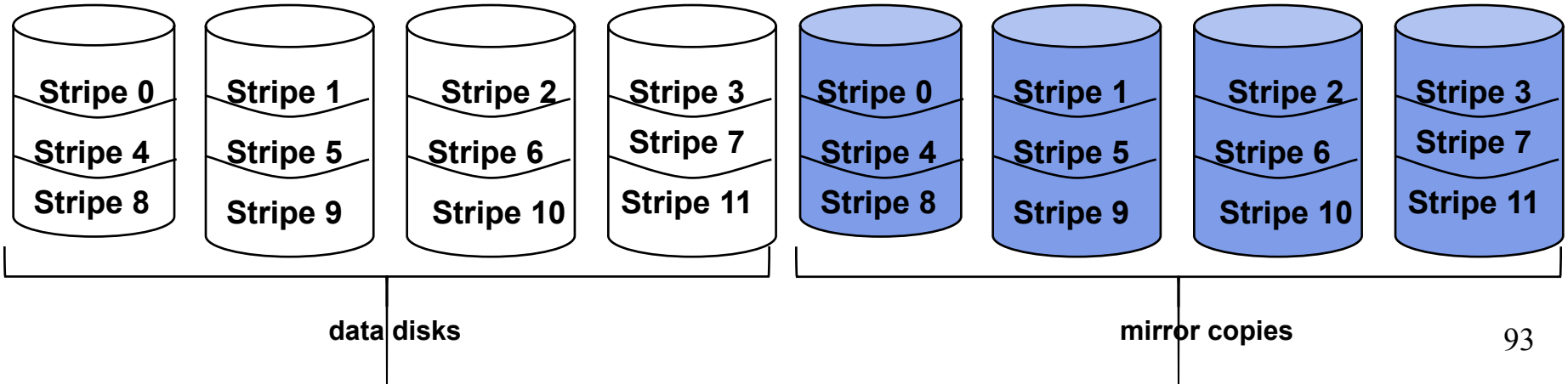
RAID Level 0

- Level 0 is non-redundant disk array
- Files are Striped across disks, no redundant info
- High read throughput
- Best write throughput among RAID levels (no redundant info to write)
- Any disk failure results in data loss
  - What's the MTTF (mean time to failure) of the whole system?



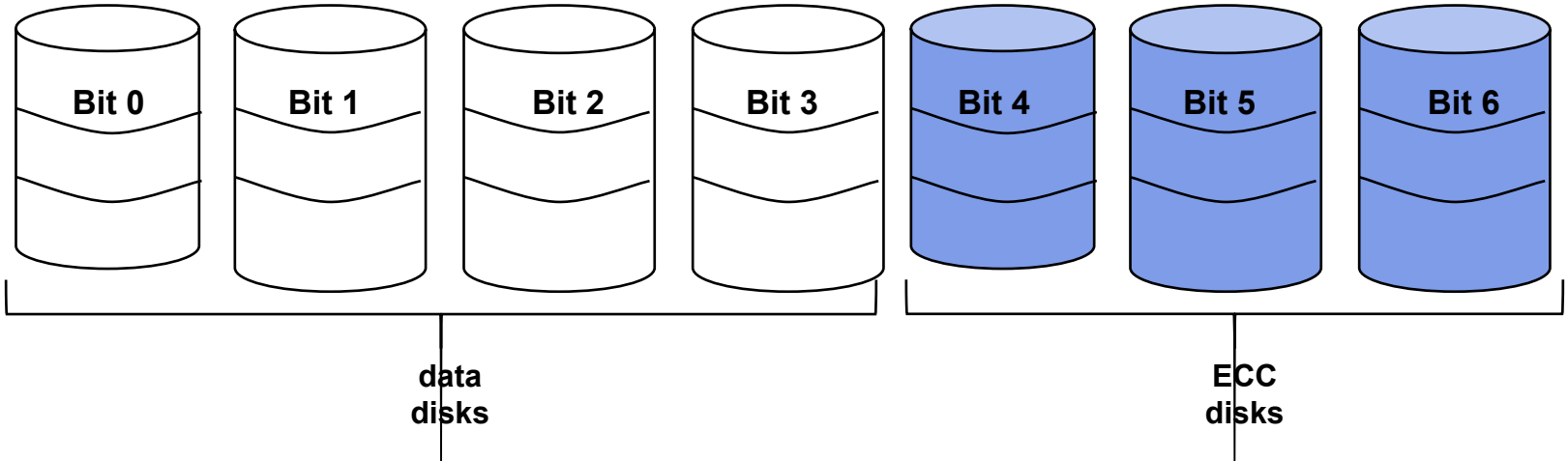
RAID Level 1

- 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



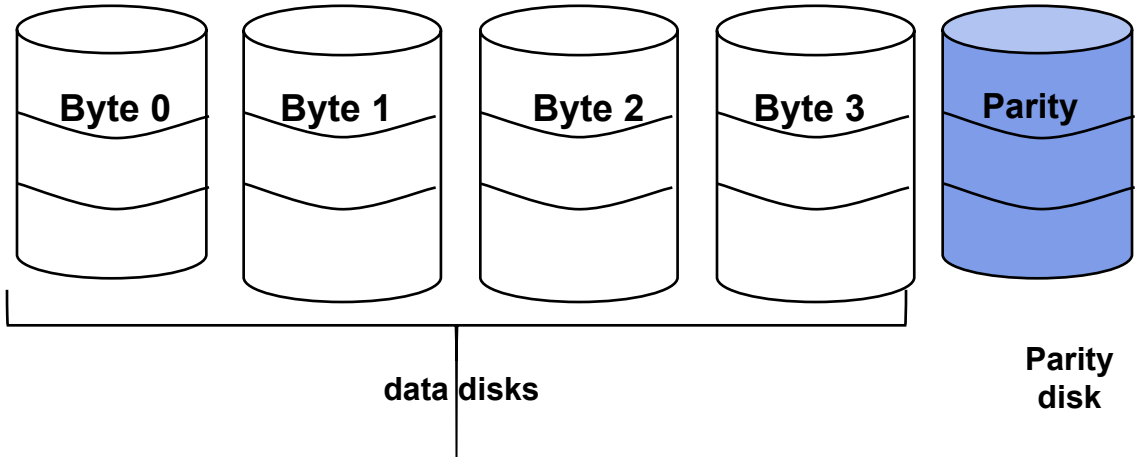
RAID Level 2

- 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
- Not used in real world



RAID Level 3

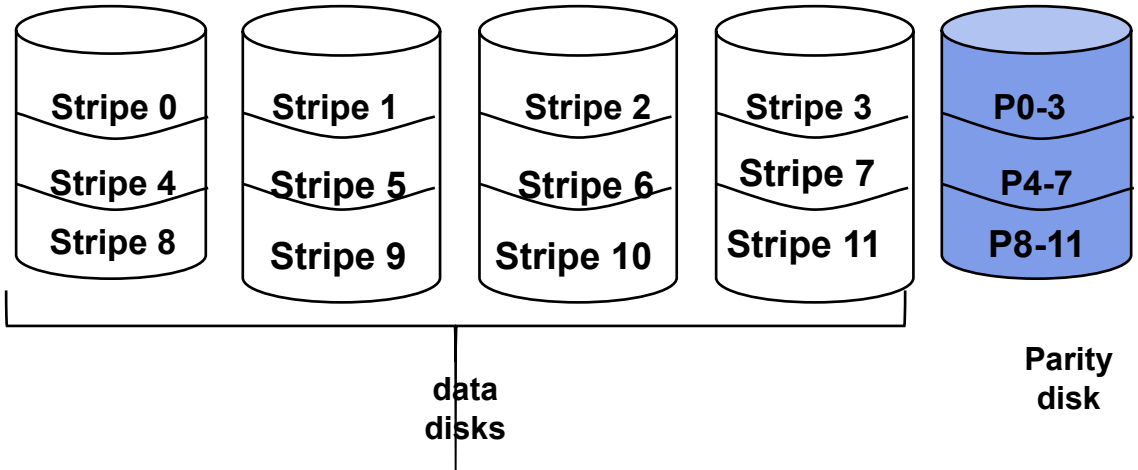
- Use a parity disk
  - Each byte on the parity disk is a parity function of the corresponding bytes on all the other disks
- A read accesses all the data disks
- A write accesses all data disks plus the parity disk
- **On disk failure, read remaining disks plus parity disk to compute the missing data**



Single parity disk can be used to detect and recover errors

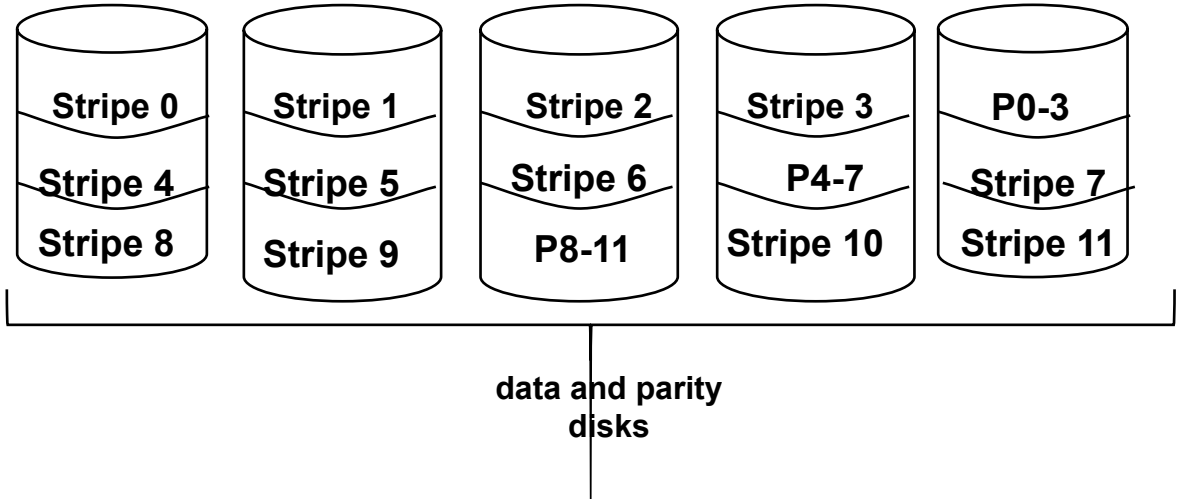
RAID Level 4

- Combines Level 0 and 3 – block-level parity with Stripes
- Lower transfer rate for each block (by single disk)
- Higher overall rate (many small files, or a large file)
- Large writes □ parity bits can be written in parallel
- Small writes □ 2 reads + 2 writes !
- Heavy load on the parity disk



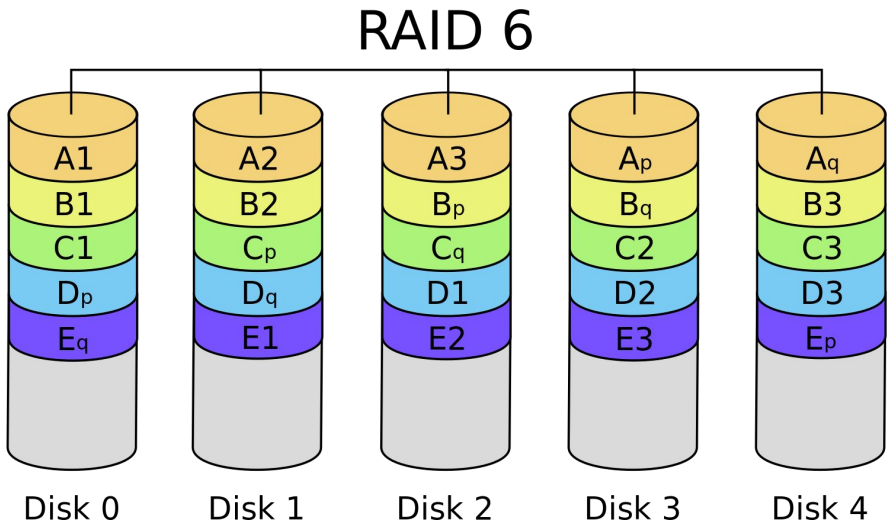
RAID Level 5

- Block Interleaved Distributed Parity
- Like parity scheme, but distribute the parity info over all disks (as well as data over all disks)
- Better (large) write performance
  - **No single disk as performance bottleneck**



RAID Level 6

- Level 5 with an extra parity bit
- **Can tolerate two failures**
  - What are the odds of having two concurrent failures ?
- No performance penalty on reads, slower on writes (compared to RAID5)



NFS Overview

- Remote Procedure Calls (RPC) for communication between client and server

- Client Implementation
  - Provides transparent access to NFS file system
    - UNIX contains Virtual File system layer (VFS)
    - Vnode: interface for procedures on an individual file
  - Translates vnode operations to NFS RPCs

- Server Implementation
  - Stateless: Must not have anything only in memory
  - Implication: All modified data written to stable storage before return control to client
    - Servers often add NVRAM to improve performance

Identifying Files in NFS

- Can we still use inode?
- NFS use File handles
- **File handle** consists of
  - ***Filesystem id*** identifying disk partition
  - ***i-node number*** identifying file within partition
  - ***i-node generation number*** changed every time i-node is reused to store a new file

Filesystem id	i-node number	i-node generation number

Performance

- Evey read and write requires a network access
- How can we avoid this frequent network access?

Client-Side Caching

- Caching needed to improve performance
  - Reads: Check local cache before going to server
  - Writes: Only periodically write-back data to server
  - Why avoid contacting server
    - Avoid slow communication over network
    - Server becomes scalability bottleneck with more clients
- Two types of client caches
  - data blocks
  - attributes (metadata)

Cache Consistency

- Problem: Consistency across multiple copies (server and multiple clients)
  - How to keep data consistent between client and server?
    - If file is changed on server, will client see update?
    - Determining factor: Read policy on clients
  - How to keep data consistent across clients?
    - If write file on client A and read on client B, will B see update?
    - Determining factor: Write and read policy on clients

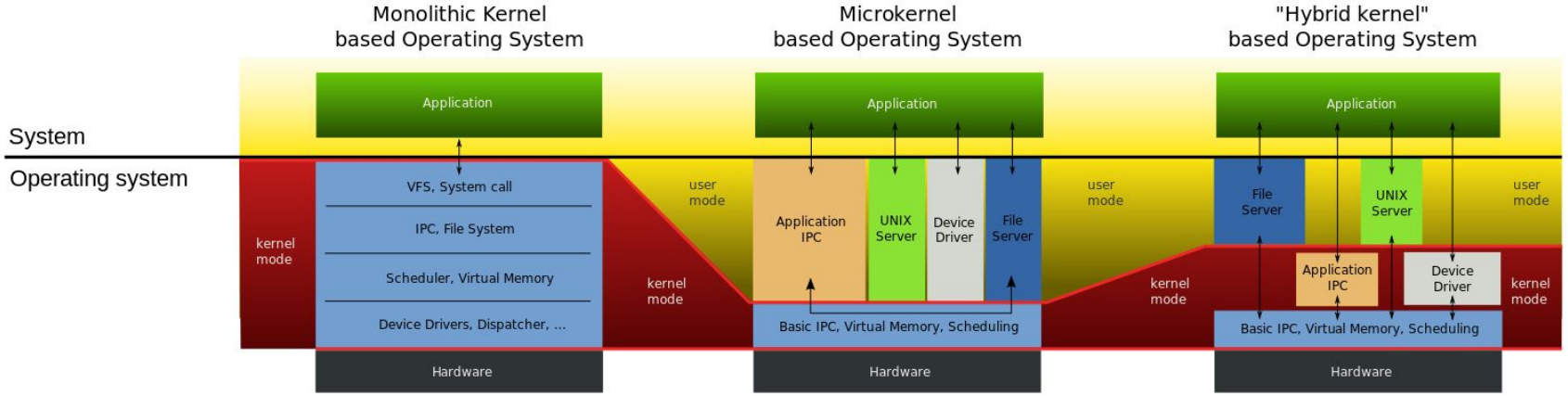
NFS Consistency: Reads

- Reads: How does client keep current with server state?
  - Attribute cache: Used to determine when file changes
    - File open: Client checks server to see if attributes have changed
      - If haven't checked in past T seconds (configurable, T=3)
    - Discard entries every N seconds (configurable, N=60)
  - Data cache
    - Discard all blocks of file if attributes of the file has been modified

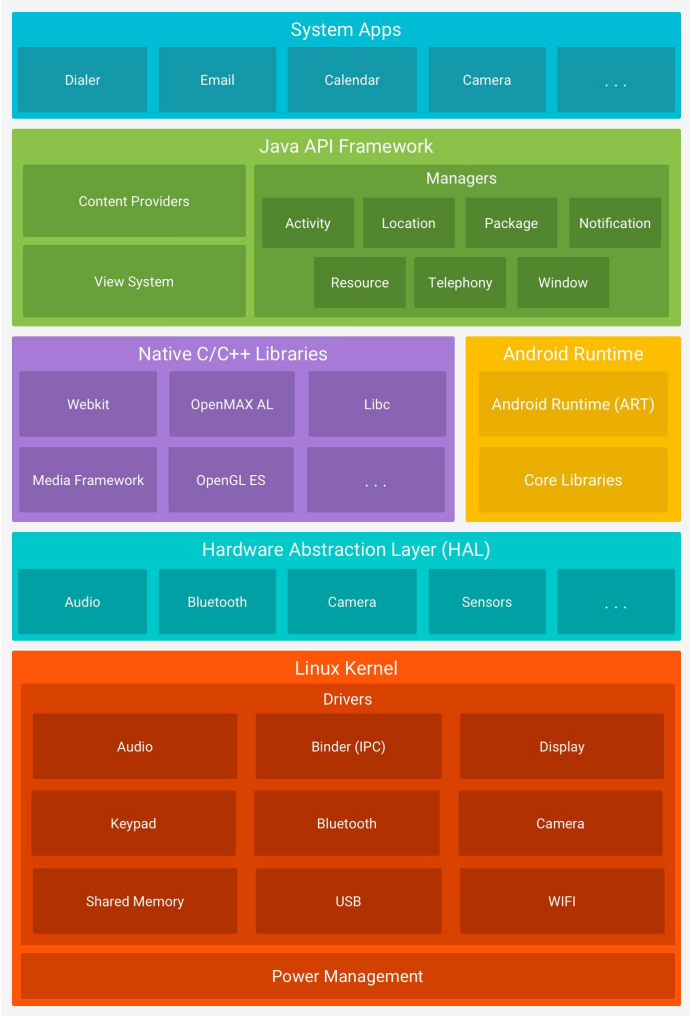
NFS Consistency: Writes

- Writes: How does client update server?
  - Files
    - Write-back from client cache to server every 30 seconds
    - Also, Flush (write all dirty data) on close() (AKA *flush-on-close*)
  - Directories
    - Synchronously write to server (write through)

Monolithic/Micro/Hybrid kernel



Case Study: Android



Case Study: FreeRTOS

- A Real Time Operating System
- Written by Richard Barry & FreeRTOS Team
- Owned by Real Time Engineers Ltd but free to use
- Huge number of users all over the world
    - 6000 Download per month
- Simple but very powerful