State transition limitations

| **Embryro** -> Ready; | **Ready** -> Ready/ Running; |
| **Running** -> Running/Blocked/ Ready/ Zombie; |
| **Blocked** -> Blocked/Ready; |
| **Zombie** -> NONE |

Turnaround time = T end – T arrive

Response time = T first run – T arrive

Quantum length = t time unit; determines time length of 1 job slice.

Allotments = # of job/time slice ran for 1 job before demotion

Size

2^5 = 32	18 = 262,144
6 = 64	19 = 524,288
7 = 128	20 = 1,048,576
8 = 256	21 = 2,097,152
9 = 512	22 = 4,194,304
10 = 1024	23 = 8,388,608
11 = 2048	24 = 16,777,216
12 = 4096	25 = 33,554,432
13 = 8192	26 = 67,108,864
14 = 16,384	27 = 134,217,728
15 = 32,768	28 = 168,435,456
16 = 65,536	29 = 536,870,912
17 = 131,072	30 = 1,073,741,824

Prefix

Kilo (K) – 10^3

Mega (M) – 10^6

Giga (G) – 10^9

1kb = 1023 b

1mb = 1024 kb

Address = Size -1

Paging

Log $_2$(1 page size) = offset bits

|VPN| = |PFN|

|VAS (B)|/ |1 page (B)| = **# of VPN**

(# of VPN) = **(# of PTE) for 1 process**

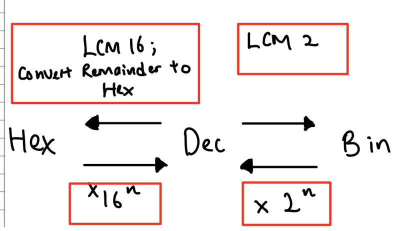
$\sum(\# \, of \; PTE) = (\# \; \boldsymbol{of} \; \boldsymbol{PFN})$

(# of PFN) * (# of addr in 1 page) = **# of PA**

Address structure:

- Linear page table
  - **VAS**: [offset]
- Segmentation
  - **VAS**: [segment (2 bits = 3 seg) | offset]
- Paging
  - **VAS**: [VPN | offset]
  - **PTE**: [valid |…| PFN]
  - **PAS**: [PFN | offset]
  - **TLB**: [valid | VPN | PFN | PID]
- Hybrid
  - **VAS**: [Segment (2 bits = 3 base-bounds) | VPN | offset]
- MLPT
  - **VAS**: [[offset in Page Directory] [offset in page of PT] | offset]
  - **PDE**: [valid | PFN]
- Physical A. S. exceeds capacity
  - **VAS**: "
  - **PTE**: [valid | protect | present | PFN]
    - Protection: read/ read-write
    - Present: is page in physical memory. If 1, fetch data from physical address; if 0, #1raise excep #2 search free page, #3 Update PTE if found/ if no free: Evict an existing page to Swap Space.

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F



Thread creation

- **pthread_create**(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start routine), (void*), void* restrict arg)
- **pthread_join**(pthread_t thread, void** retval)

Locks

- **pthread_mutex_lock**(pthread_mutex_t *mutex)
- **pthread_mutex_unlock**(pthread_mutex_t *mutex)

Condition variables

- **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t * mutex)
  - Releases lock associated with its mutex lock arg.
  - Puts the calling thread to sleep.
- **pthread_cond_signal**(pthread_cond_t* cond)

Semaphores

- **sem_init**(sem_t *sem, int pshared, uint value)
- **sem_wait**(sem_t* sem)
  - Decrement the value of semaphore by 1
  - WAIT if the value of the semaphore is <0.
- **sem_post**(sem_t* sem)
  - Increment the value of the semaphore by 1
  - If there are one or more threads waiting, wake one.

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

Think in **assembly** during code trace!

Concurrency bugs

- **Atomicity violation**: "A situation where the desired serializability among multiple memory accesses is violated". (Identified: If the bug caused by two accesses will work if the code that does so simply runs till completion.) Solution: Atomize using locks.
- **Order violation**: "A situation where the desired order between two (groups of) memory accesses is flipped". (Identified: If the bug caused by the two accesses will work with a simple swap in order.)
- **Deadlock bugs**: Scenario: **>1 thread** runs **concurrently**, **>1 shared** locked **resource**. A situation where >1 process is blocked. E.g., Process 'X' (P(X)) holds resource A. Process 'Y' (P(Y)) wins the race w P(X) and holds resource B. P(Y) wants A. But A is locked by P(X), AND P(X) is waiting for P(Y)'s resource B.

Persistence

- 1 **block**/ **sector** = 512 bytes
- I/O time ($T_{I/O}$) = seek time + rotational delay time + transfer time
- **Avr seek time**: (Given by disk manufacturer, in exam question)
- **rotational delay time**: (RPM **->** RPS **->** time for 1 rot in ms)
- **Avr rotational delay time**: rotational delay time/ 2
- **transfer time**: size of transfer/ peak transfer rate
- I/O rate ($R_{I/O}$) = (Size of transfer/ $T_{I/O}$) <where s.o.transf, adjusts for rand(4kb), seq (>4kb)>
- Average seek distance for disk = $\frac{1}{3}$(N), N – full distance seeks

***Dimensional analysis practice**

Track skew, track buffer, write-back caching (reports disk write success upon memory cache), write-through caching (reports disk write success when it is written to disk),

Performance drives, Capacity drives

- Random reads, Sequential reads (zero seek time)

Scheduling

SSTF/SSF – Shortest Seek Time First: (*Determines seek by shortest relative sector number.) picks requests on the nearest track to complete first.

NBF – Nearest Block First: picks requests with the nearest block address next.

Elevator/ SCAN: (avoid starvation) "sweeps"/ arm moves back and forth across the disk to service request cross tracks in a pendulum fashion.

SPTF – Shortest Positioning Time First = **SATF** (``Access``) – chooses SSTF if seek time > rotational delay. (e.g., favor further track that has short seek t.)

RAID: Trait – Reliability, Capacity, Performance

- RAID 0: Stripping
- RAID 01: Stripping + Mirroring (mirror stripes)
- RAID 10: Mirroring + Stripping (stripes of mirrors)
- RAID 4: Stripping + Parity
- RAID 5: Stripping + Distributed Parity

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**

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**

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**

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**

Metrics
N- # of disks; **B**- # of blocks; **S**- sequential I/O bandwidth; **R** – random I/O bandwidth.
How to calculate Throughput?
1) consider the worst case. E.g., n rows of simultaneous write/ read. 2) Scale it across all numbers.

	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	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S^1$	$(N-1) \cdot S$	$(N-1) \cdot S$
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	$2T$	$2T$

(the lower the number, the worst its throughput)

Parity block XOR

XOR Truth Table

Input 1	Input 2	Output
0	1	1
0	0	0
1	1	0
1	0	1

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

- If $C_{old} == C_{new}$, $P_{new} = P_{old}$
- If $C_{old} != C_{new}$, $P_{new} = \sim P_{old}$

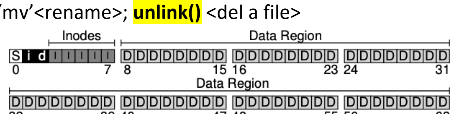
File System Implementation

- 1 inode = 256 bytes
- 1 block = 4096 bytes

- **Inode block** = (inum * size_of_an_inode)/ block_size
- **Sector # within block/ inode #** = ((block_num* block_size) + inodeTableStartAddress)/ sectorSize

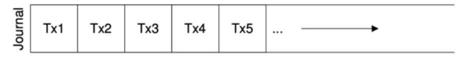
Open(), create(), read(), write() routine in file system implementation.
- **Open()** - #1. Read (inode, data) pairs from the root -> child's inode, disregarding data.
- **Read()** - #1. Access (inode, data) pairs. #2. Update child inode, disregarding data.
- **Create()** - #1. Read (inode, data) pairs from the root -> parent's (inode, data) pair. #2. Read + write inode bitmap. #3. Write parent's data, #4. Read + write child inode. #5. Write parent inode.
- **Write()** - #1. Read child inode. #2. Read + write data bitmap. #3. Write child data. #4. Write child inode.

Sys calls
readdir() = 'ls'; **stat()** = 'stat'<info abt a file>; **rmdir()** <del directory>; **link()** = 'mv'<rename>; **unlink()** <del a file>



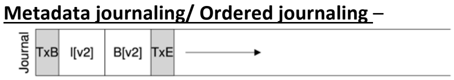
Consistent data: **Inode table, data bitmap** is updated.

Crash recovery routines – by reinforcing Atomicity.
- File system **repair** (fsck) – linear scan + repair (expensive)
- *Write-ahead* **logging** (journaling)
  - this is done before writing data to disk. If disk crash, items in log is lost, there for none/ all is completed (atomized).
  - **Data journaling** –



- 1 logged transaction, within a journal block
- **Create() updates**: 1) directory inode, 2) directory data block, 3) new inode, 4) inode bitmap, 5) data bitmap (only if file s > 0), 6) new data block (only if file s > 0)
- **Read() updates**: 1) new inode
- **Delete() updates**: if cause dir to be empty: None
- Process: Journaling (TxB, I, B, Db) -> committing (TxE) -> checkpointing (write logs into real disk) -> Free "checkpointed"

- **Metadata journaling/ Ordered journaling** –



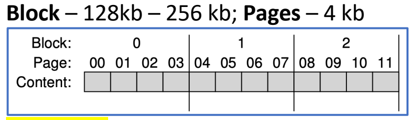
eliminates Db

- Process: Data write (Db) -> Journaling (TxB, I, B)-> **commiting** <strictly>(TxE) -> checkpointing (write logs to real disk) -> Free

DFS – Distributed File System
- Prob: Partial failures (shared states), Network drops (request lost, server down, reply lost otw to server)

NFS – Network File System
- Sol: protocols (a) Stateless **File handle**(s) to identify a file or dir -> **1.** Volume identifier (denotes the file system used), **2.** Inode number (the file in it), **3.** Generation number (distinct it from new/ reused 1, 2).
- **File descriptor** within a file table (client's data struct to store "file handles" received) != **File handles** (denotes a file at server side)
- **Timeout/ Retry approach** – client resends request after a time period of not receiving a reply, assuming operations of request are implemented in an "**idempotent**" fashion.
- **Client caches** – client has **buffers for** "**request**": write buffering (specifically writes()) & "**replies**": caching (received server data)

Flashed-based SSD
- Constraints: "**write**" 1 page (Writes to the next free page. Erase 1 block, sets the pages to programmable, writes 1 page). > "**program**" 1 prog-able page (set page to valid). > "**reads**" 1 valid page. *(> denotes cost lvl.)
- **Block** – 128kb – 256 kb; **Pages** – 4 kb



- Page state: **Invalid (i)** (start stage), **Erased (E) (**always done first before ANY updates**)**, **Valid (V)** (programmed), **Error**(**error**) (if program a programmed page)
- **Flash Translation Layer (FTL)** – converts requests into read, erase, program commands & tracks the right most ERASED page in the block.
- **Log structured FTL** -ALWAYS logs/ appends the write to the next free spot in the currently-being-written-to block.
  - **In-memory mapping table** – ***page mapped*** [virt addr: SSD phy page, …] | ***block mapped*** [virt addr: SSD phy block, …]
- **Garbage collection** – finds dead pages (not pointed in memory mapping table), read live data, write live data, ERASE dead block with dead pages.
- **Wear leveling** – spread writes across blocks of flash evenly