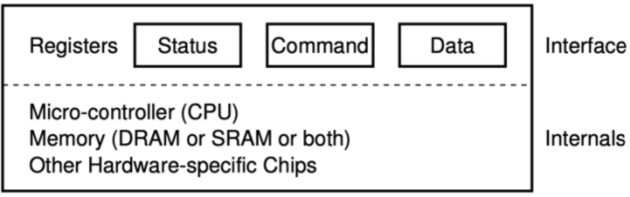
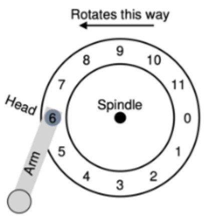


Hardware <u>interface</u>: present to the rest of the system, and system software can control the operations. <u>Internal</u>: implement the abstraction presented. <u>Status</u>: read to see the current status of the device; <u>Command</u>: tell the device to perform a certain task; <u>Data</u>: pass data to the device, or get data from the device. <u>Step</u>: polling for status ready → OS send data→ OS send command→ polling for status finished.

<u>Polling</u>: waste CPU time, rather than switch to another proc. <u>Data movement</u>: CPU intensive. Solution: <u>Interrupt</u> and DMA.
Interrupt: if status = busy: semi_wait(&io); //put this proc to sleep, context switch to another task.
When device finish operation/disk request finished: raise a hardware interrupt → CPU jump into the OS at an interrupt handler (which signals OS) → interrupt handler wakes the process that's waiting for I/O.
<u>DMA</u>: OS telling DMA engine 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. DMA is complete → raise an interrupt.

1 sec=1000ms
10 MB at 50 MB/s = 200 ms
1 MB=1000KB
2^10 = 1024, 2^12 = 4096
2^15 = 32768, 2^16 = 64KB
2^20 = 1 mil, 2^6 = 64

<u>Hard drives</u>: arr of sectors (512 bytes), 0 to n-1 is the addr space. 512 bytes write is atomic.
A platter has 2 sides/surfaces → read and write. A platter consists of tracks, which consist of sectors.
A spindle spins the platter around at a constant fixed rate with unit Rotations Per Minute.



<u>Seek</u>: move disk arm to correct track.
Rotational delay: wait for the desired sector to rotate under the disk head.
e.g. transfer rate = 100MB/sec, time to transfer 512KB block : $\frac{Time\ (ms)}{1\ Request} = \frac{512\ \cancel{KB}}{1\ Request} \cdot \frac{1\ \cancel{MB}}{1024\ \cancel{KB}} \cdot \frac{1\ \cancel{second}}{100\ \cancel{MB}} \cdot \frac{1000\ ms}{1\ \cancel{second}} = \frac{5\ ms}{Request}$
e.g. avg seek = 7ms, avg rotate = 3ms, transfer = 0ms. Could do ~200MB/s for 512 bytes (sequential)
But if random I/O: (512b/10ms)*(1000ms/1sec) = 0.5mb/s → $R_{I/O}$ = Size of tranfer/$T_{I/O}$
<u>Disk scheduling</u>: have an estimate of a disk request's seek+rotation time, so try greedily SJF.
Shortest seek time first (SSTF)/nearest block first: closest track. SCAN: simply moves/sweeps back and forth across the disk servicing requests in order across the tracks. Shortest Positioning Time First (SPTF): both seek and rotate accounted.
Problems: starvation → bounded SATF: service all request in a window of requests before any subsequent requests.
<u>RAID</u>: redundant array of inexpensive drives; designed to detect and recover from certain kinds of disk faults.
<u>Chunk size</u>: big → reduce intra-file parallelism, reduce positioning time; small → many file striped across many disks, positioning time to access blocks across multiple disks increases.

	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	$2T$	$2T$

Note: level 0 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

Data0	Data1	Data2	Data3	Parity
0	1	1	0	0 [strip 0]
1	0	0	0	1 [strip 1]

"full stripe write": do all writes in parallel, including the parity block. "random write": Additive → read the rest of the stripe in parallel and write the new data + parity in parallel. Subtractive → compare the new data with old data = 2 reads + 2 writes ☹
Note: we can not parallel reading/writing to parity disk, so all writes will be seriliazed regardless of data disks.
Memory: contents lost after power loss → <u>Persistant storage/hard drive</u>: file system
<u>File</u>: linear arr of bytes, its name is low-level = inode number; <u>Directory</u>: list of files &/or directories, map "main.c" → 1000
Absolute path name: starts at root "/" and includes the entire path to file or directory

<u>Lseek</u>:

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

<u>Offset</u>:

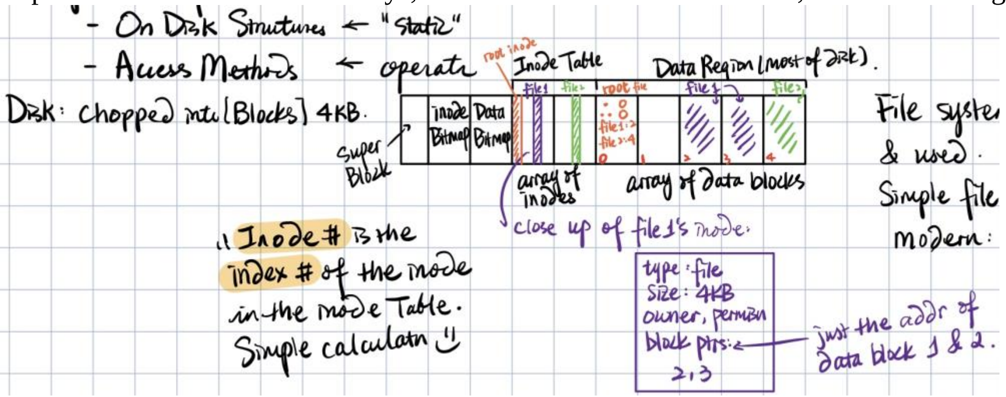
System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

The lseek() call first sets the current offset to 200. The subsequent read()reads the next 50 bytes, updates the current offset accordingly.

<u>Read and write</u>: naturally update the current offset.
<u>Fork</u>: a parent creates a child and waits for it to complete. The child adjusts the current offset with lseek() and exits. The parent, after waiting for the child, checks the current offset and prints out its value. Answer: child offset = parent offset = 10.
Fork is an example of using shared open file table entry, when an entry is shared, its reference count ++.
<u>Dup</u>: create a new file descriptor that refers to the same underlying open file as an existing descriptor.
<u>Hard links</u>: "link" a new file name to an old one, create another way to refer to the same file → creates another name in the di- rectory you are creating the link to, and refers it to the same inode number (i.e., low-level name) of the original file.
-when create a file: 1. Make a struct/inode that viturally track all info; 2. Linking a readable name to file and put link in dir.
ln file file2 → rm file → cat file2 → still works, print out "hello" ☺
Unlink hard links: check the ref count (# of file names link to this inode) in inode number, remove "file" link, decrement ref count, if ref count = 0: free inode and related data blocks → truly "delete" the file.
<u>Soft links</u>: is actually a file itself, of a different type → is formed is by holding the pathname of the linked-to file as the data of the link file. Unlike hard links, removing the original file named file → the link to point to a pathname that no longer exists.
lh -s file file2 → rm file → cat file2 → cat: file2: No such file or directory

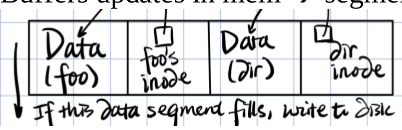
<u>File System Implementation</u>

1. Max # of files = max # of inodes = # of inode blocks * (size of 1 block/size of 1 inode); 2. Inode table size = # of inode blocks * size of 1 block; 3. Blk= (inumber * size of 1 inode) / block size; 4. Sector # for fetching inode block = (blk * block size + inodeStartAddr)[aka offset into inode region] / sector size = (inumber * size of 1 inode + inodeStartAddr) / sector size
Bitmaps: track whether inodes or data are allocated, each bit indicate whether the corresponding block is free (0/1)
Superblock: info about this file sys, like # of inodes and data blocks, where inode begins, etc.



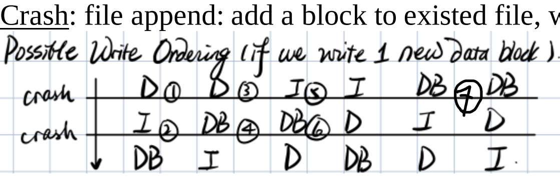
Directory: just a file.
Root inode: store in a fixed loc in inode arr.
Process of accessing file2 data given directory: [/file2] → /root inode → root data → file 2 inode → file 2 data
Inode refer to data block: direct pointers (disk addr).
Large files: indirect pointers (to a block that contains more direct pointers) in inode b.c. inode size are fixed but # of blocks are not.

Cache/Buffer: write() → fast buffers in mem and returns, if crash lost data. Not lose data: write() then fsync (slow).
FFS: block group is continuous part of disk, allocate related item into a group. Mkdir spread across disk (like pick a group with a small # of dir in it). Creat puts files in same group as prent directory. ☹: internal fragmentation
Directory: each directory has two extra entries, . "dot" and .. "dot-dot"; the dot directory is just the current directory (in this example, dir), whereas dot-dot is the parent directory (in this case, the root).
Read: final step of open a file /foo/bar: to read bar's inode into memory; Then read from bar file: read bar inode → read bar's first data block → update bar inode with file offset → repeat, but will read bar's second data block if exist. Close: no disk I/O.
Write: Create(/foo/bar): … → read foo data → read inode bitmap → write inode bitmap → write foo data → read bar inode → write bar inode → write foo inode. Write(): read bar inode → read data bitmap → write data bitmap → write bar data[0] → write bar inode → repeat all over starting from write, until write all data, but next time write to bar data[1]…
Solution: caching reads and write buffering; Solution 2: Static structure → Dynamic Log Structured
Buffers updates in mem → segment (~1MB); when segment fills, write to end of log (never overwrite existing data)



Read(fd, buffer, 4KB): read root inode → read root data(look for inode # of foo) → read inode of foo. How to find inode: 1. Scan disk (slow); 2. Inode map(arr), inode # is index, content is latest version of inode.
Beginning of disk: checkpt region; 具体的看下方的截图， write and update to new locations.

Segments: large chunk of data LFS writes at one time, LFS updates in memory segment and writes it at once to disk.
Use the checkpoint region to read in all inode maps and cache them for later read.
Garbage: what's live/dead: look @ data block and its inode, check if the inode pt to it. LFS cleans segment by segment.
Crash: file append: add a block to existed file, which changes **data bitmap** + inode (add a ptr to the new data block) + the data.



1. lose data, file sys fine; 2. Inode: data block 100 valid and belong to this file. DB: data block 100 free, other can overwrite (inconsistency btwn inode and old version of Db); 4. Mem / space leek, no idea which file the data belongs, inconsist; 5. Read garbage, inconsist; 6. Consistent (both pt to garbage); 7. Space leek (data may never be used by file system).

Solution: file sys checker (lazy) Solution 2: write ahead logging (WAL, journalising, eager)
Fsck: run checker after reboot/crash, scan entire file sys, find inconsistencies btwn bitmaps and inode, fix mismatches (Slow)
WAL: before update, record some info → do update → Use info to recover if crash
Jounral block is in btwn super block and bitmaps.

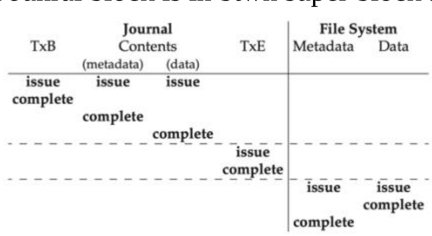


Figure 42.1: **Data Journaling Timeline**

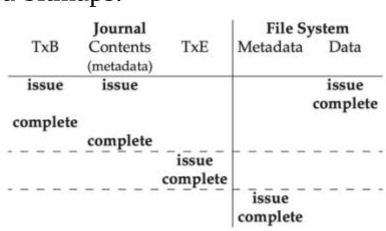
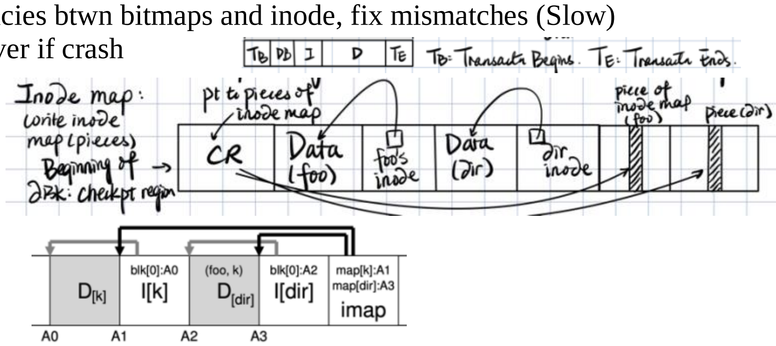
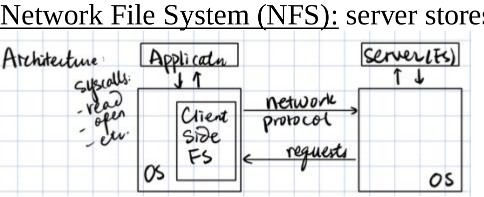


Figure 42.2: **Metadata Journaling Timeline**



Network File System (NFS): server stores data on its disks, client request data. Data naturally shared across dif./ machines.



Make server crash easier: file handle <volume #, inode #, generation #> = <which FS, which file, unique identifier for a file with the right version>
Read(file handle, offset, size) = server don't keep track where to read, client says with offset.
Stateless: don't track which client requred which file descriptor.
Potential problems: requeset lost, reply lost (more severe), server down.
Uniform crash handling approach: client time out, client retry → idepotency (repeatable, doing sth./ n times is the same as doing it once)
Caching: client. Flush on close: when a file is written to and subsequently closed by a client application, the client flushes all updates (i.e., dirty pages in the cache) to the server. Solve staleness: check w/ server if data has changed before using cached version.
SSD: flash consists of blocks, which consists of pages. Operations: read a page (2kb/4kb), write = erase block (256kb) + program (page).

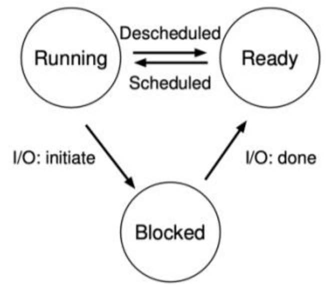
Performance: read = 10s of microsec, erase = a few milisec, program = 100s of usecs

Direct map logical page to physical page: Update a block = read block + erase block + program it again → too many I/O for a single write/update, ☹ if crash after erase, metadata re-written too many times (wear out that block)

Logging: treat falsh blocks like a log. FTL maps page in mem to location on flash. Size of FTL: 1-TB SSD, for example, a single 4-byte entry per 4-KB page →1 GB of memory for mapping = (1TB/4KB) * 4byte = (10^12/4000) * 4 byte = 10^9 bytes = 1GB.

Virtualization: every process has illusion of its own CPU and own memory.

Mechanism: low level machinery, methods or protocols that implement a needed piece of functionality. E.g., context switch (os stops running one program and starts another), time sharing. How multiple processes share the CPU? Limited direct execution.




```
int main(int argc, char *argv[]) {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n",
               rc, (int) getpid());
    }
    return 0;
```

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

"The parent process calls wait() to delay its execution until the child finishes executing. When the child is done, wait() returns to the parent."

Policy: on top of mechanisms, intelligence in the OS, algorithms for making decision within the OS. Which program should OS run?

Exec: does not create a new process; rather, it transforms the currently running program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

Note: Response time = T first runs – T arrives; Turnaround time = T job completes – T arrives

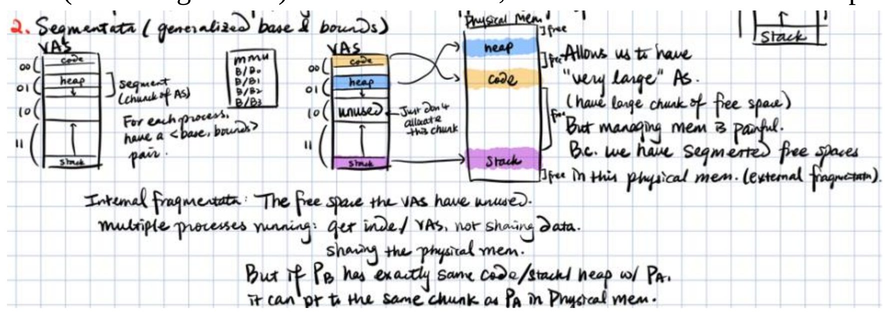
MMU: Base(where is 1st byte of AS in physical mem) and bound (valid range in PA): PA = base + VA; MMU hardware structure kept on the chip, one pair per CPU.

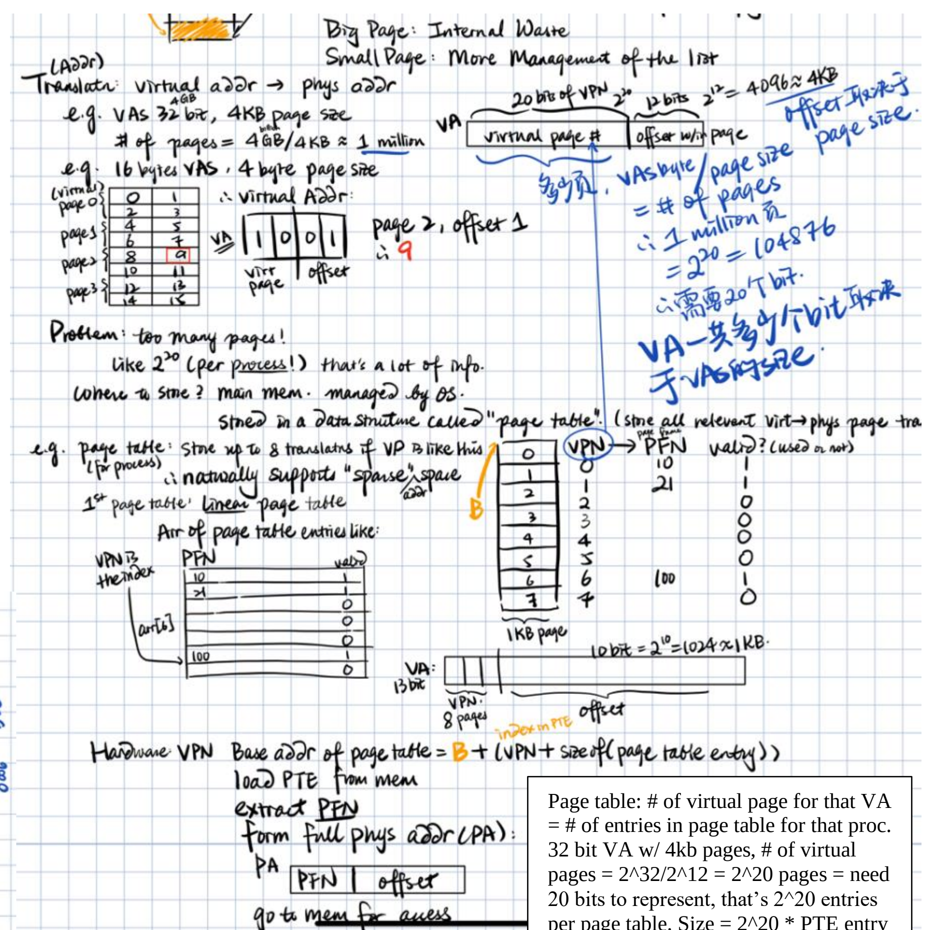
OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of... system call handler, timer handler, illegal mem-access handler, illegal instruction handler
start interrupt timer	
	start timer; interrupt after X ms
initialize process table **initialize free list**	

OS @ run (kernel mode)	Hardware	Program (user mode)
To start process A: allocate entry in process table, allocate memory for process, set base/bounds registers, **return-from-trap** (into A)		
	restore registers of A, move to **user mode**, jump to A's (initial) PC	
		Process A runs Fetch instruction
	Translate virtual address and perform fetch	
		Execute instruction
	If explicit load/store: Ensure address is in-bounds; Translate virtual address and perform load/store	
		...
	Timer interrupt move to **kernel mode** Jump to interrupt handler	
Handle the trap Call switch() routine: save regs(A) to proc-struct(A) (including base/bounds), restore regs(B) from proc-struct(B) (including base/bounds), **return-from-trap** (into B)		
	restore registers of B, move to **user mode**, jump to B's PC	
		Process B runs Execute bad load
	Load is out-of-bounds; move to **kernel mode**, jump to trap handler	
Handle the trap Decide to terminate process B, de-allocate B's memory, free B's entry in process table		

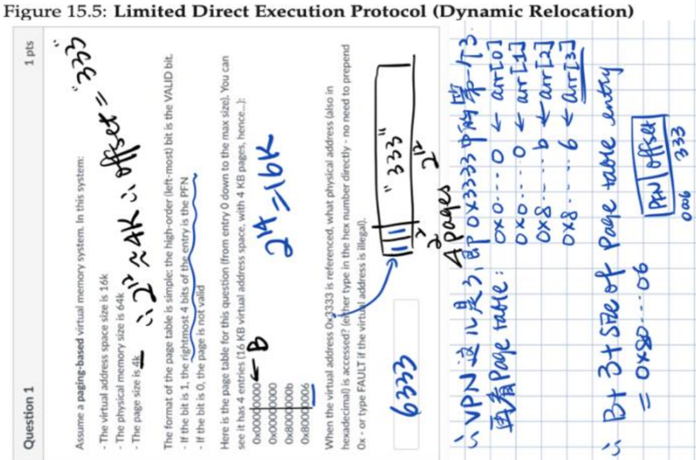
Trap handler: code in OS handle service requests from app. App issue a system call → os issue a trap.

Figure 15.5: Limited Direct Execution Protocol (Dynamic Relocation)





Page table: # of virtual page for that VA = # of entries in page table for that proc. 32 bit VA w/ 4kb pages, # of virtual pages = 2^32/2^12 = 2^20 pages = need 20 bits to represent, that's 2^20 entries per page table. Size = 2^20 * PTE entry size (say it's 4b) = 4MB for this one proc.



Question 76: You are now given some new information about a particular system. Specifically, this system has **1 MB linear page table size** (per process), and has a **1KB page size**. Assuming page table entry size is **4 bytes**, how many bits are in the virtual page number (**VPN**) on this system?

a) 28
b) 18
c) 8
d) 32

1 MB linear page table size, 4 byte per page table entry. So 1MB / 4 byte = 2^18 entries. Each VPN has one entry, so 2^18 VPNs.

You are given a system with 64 bytes of physical memory, 4 byte pages, and 16-byte virtual address spaces. Your forensics tools dig up the following page table structure (high bit: Valid/NOT, rest is the PFN):

```
[0] 0x00000000
[1] 0x800000?? <- miss
[2] 0x00000000
[3] 0x00000000
```

page size = 4b → 2 bits in the VA are for the page offset. given VA - 0000 0111, trim the last 2 bits to 0000 01. the table has a valid entry in [1] (0x8 -> 1000, MSB set)

Question 71: A trace you have accesses virtual address 0x7, which translates to 0x33. What two hex digits are missing from page table entry 1 above?

a) 0x0a
b) 0x0b
c) 0x0c
d) 0x0d

given Physical Address 0011 0011 (33), of which the last 2 came from the VA page offset. so left with 0011 00. this needs to be the answer. converting it to hex [00]00 1100 -> 0x0c

<u>Thread</u>: like a separate prcess, except they SHARE the same addr sapce and thus
Two threads running on a single processor, T1 → T2: **<u>context switch</u>**.
<u>Critical Section</u>: piece of code that accesses a shared var/res and must not be concurrently executed by > 1 thread.
<u>Atomically</u>: instruction executed not be interrupted in the middle → either instr not run at all or run to completion
<u>Lock</u>: mutually exclusiveness btwn threads; ONLY 1 thread is running within critical section at a time.
- use a simple flag: init mutex->flag = 0 and spin-wait in the while loop if flag is 1, lock flag = 1, unlock flag = 0
  - can't work bc both threads set the flag to 1
  - spin-waiting wastes time waiting for another thread to release a lock = # of thread * len(time slice)

☺ **<u>Lock</u>**: <u>Test-And-Set</u>/atomic exchange (correctness ☺, fairness ☹, performance ☹ with 1 CPU and ☺ on >1 CPUs)

```
// test the old val and simultaneously
// set the mem loc a new val
  int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value }
```

```
init(lock_t *lock) { lock->flag = 0; //lock available}
void lock(lock_t *lock) { while (TestAndSet(&lock->flag, 1) == 1); //spin wait-do nothing}
void unlock(lock_t *lock) { lock->flag = 0; }
```

- If lock free, flag = 0, T1 calls lock(): T1 calls TestAndSet(flag, 1), return the ol atomically set the value to 1, indicating the lock is now held.
- If lock held, flag = 1, T2 calls lock(): T1 calls TestAndSet(flag, 1), return the old value of flag, which is 1; will simultaneously setting it to 1 again. As long as the lock is held by T1, TestAndSet() will repeadtely return 1. When T1 releases lock and flag = 0, T2 calls TestAndSet(flag, 1) again and acquire the lock and enter its critical section.
- Single CPU/single processor needs a <u>preemptive schedule that will interrupt a thread via a timer</u>.

☺ **<u>Lock</u>**: <u>Compare-And-Swap</u>
// test whether the value at the addr in ptr is equal to expected, if so, update the memory locatoin pointed by ptr with the new value, if not, do nothing. Return the actual value at that mem loc.

```
int CompareAndSwap(int *ptr, int expected, int new) {
  int actual = *ptr; if (actual == expected): *ptr = new; return actual; }
void lock(lock_t *lock) { while (CompareAndSwap(&lock->flag, 0, 1) == 1); //spin }
```

- checks if flag is 0: atomically swaps in a 1 thus acquiring the lock. Other threads will stuck spinning.

☺ **<u>Lock</u>**: <u>Fetch-And-Add</u> (Fairness ☺)
// atomically increments a value while return the old value at a particular addr

```
int FetchAndAdd(int *ptr){ int old = *ptr; *ptr = old + 1; return old; }
```

- tickets: T1 acqure a lock, atomic fetch-and-add on the ticket value, and that value is T1's thread's turn. The globally shared lock->turn is used to determine which thread's turn it is. When myturn == turn for a given thread, that thread enters the critical section. Unlock: turn ++ s.t. the next waiting thread can enter the critical section.

☺ **<u>Yield:</u>** Solve entire time slice doing nothing but checking lock's value that's not gonna change

```
void lock(lock_t *lock) { while (TestAndSet(&lock->flag, 1) == 1): yield(); // give up the CPU and let other thread run }
```

- moves from RUNNING to READY, yielding proc deschedules itself
- works well with 1 CPU w/ few threads, but not w/ many threads (T1 acquires the lock and preempted before release it, T2-T100 will each execute run-and-yield pattern before T1 gets to run again), but still better than just spinning

<u>CV</u>: signaling between threads, <u>have a lock associated with the condition</u>, needs lock be held calling wait and signal
- <u>cond_wait</u>(&cond, &lock): puts the calling thread to sleep, and thus waits for some other thread to signal it
- wait's second parameter &lock: assumes the lock is held when wait is called, wait releases the lock when putting caller to sleep (atomically), letting the other thread acquire the lock
- before returning after being woken, cond_wait re-acquires the lock, anytime waiting thread running btwn lock acquire at the beginning of the wait seq and the lock release at the end, it will hold the lock.
- <u>WHILE</u> loop to check the wait condition (<u>s.t. thread rechecks condition after being woken by signal from wait</u>)
- <u>cond_signal</u>(&cond): always make sure to have the lock held, wake a single waiting thread if ≥1 thread is waiting, return without doing anything if no thread is waiting. SLEEP->READY/RUNNABLE
- always use CV and associated lock rather than a simple flag to signal btwn 2 threads
- **<u>CV is a queue of waiting threads because some state of execution is not desired</u>**
- a CV is usually paired with some kind of state variable like int state, indicating the state of system we're interested in, like if the child is done or not

<u>Bounded Buffer:</u> block put when queue is full, consumer wakes producers but not other conusmers, vice versa. A good way is to use 2 CV and while loops. Producer thread wait on the condition empty and signals fill. Consumer thread wait on fill and signal empty. <u>Producer</u>: while count == 1: cond_wait(&empty, &mutex) then put when count == 0. <u>Consumer</u>: while count == 0: cond_wait(&fill, &mutex) then get when count == 1.
- Add more slot to the queue: int buffer[MAX]; fill_ptr = 0; use_ptr = 0;

<u>Semaphores:</u> lock and CV in one. Sem_init(&lock, 0, 1): 0: semaphore shared btwn threads, 1: semaphore value
- Many thread to acquire at "same time": let 1 acquire and put others to wait
- <u>Parent/Child</u> aka fork/join: initialize semaphore w/ 0; always think about the 2 cases here.
- <u>Read/Write</u>: either many reads using the file or 1 write editing the file (exclusive or); acquire/release writelock
- Init both write and read lock value to 1. First reader grabs lock and write has to wait until all readers finished.
- Last one exit the critical section calls sem_post on writelock and the waiting writer can acquire the lock.
- <u>Bounded buffer:</u> sem_init(&empty, 0, MAX) and sem_init(&fill, 0, 0), consumer wait for full and post empty
- <u>sem_wait</u>(sem_t *s): value --; while value < 0: caller sleeps;
- <u>sem_post</u>(sem_t *s): value ++ //release lock; if exists a thread waiting to be woken, wakes one of them up
- use semaphores as a lock, initialize value =1;

<u>Deadlock:</u> When all entities (threads, processes) are waiting for a resource held by some other entity in a group. None will release what they hold until they get what they are waiting for.
- Mutual exclusion: ≥1 res held non-shareable; requests delayed until release
- Hold and Wait: Exists a process that is holding >=1 resource, waiting for another that is held by some other process
- Solve above: surround lock acquisition w/ a global lock, guarantees all procs grab resources at ones
- No preemption: resources only release voluntarily → Solve: force others to release the lock
- Circular wait: set of procs s.t. P0->P1->…->Pn->P0 → Solve: impose total order on locks: Acquire in M1…Mn order only, which is done by if v_dst<v_src, then acquire &v_dst->lock first
- Other way to solve deadlock: try to acquire lock, it can't return with error code immediately, go back to "top" of code and check again; if can, acquire the lock and return
- ☹ about try: result in livelock → might get stuck always "trying" to get locks