


- I/O Disk**
- platter** - circular hard surface on which data stored 盘片, 磁轨, ...
  - track** - data encoded on each surface in concentric circle/sector → each circle's name
  - multi-zoned disk drive** - outer track tend to have more sector than inner track
  - disk I/O policy (best/fast to worst/slow): **SATF/SPTF > SSTF > FIFO**
    - I/O/SPTF** - consider seek & rotate; **SSTF** - only consider seek; **FIFO** - most primitive hence slowest; **SCAN** scheduling avoid starvation (only consider seek)
  - 3 basic components of disk I/O time - **seek, rotate, transfer**
  - log-structured read performance similar to reg HDD seq/rand read workload; log-structure write performance only similar to reg HDD rand read workload
  - disk scheduling of write req **sometimes help** improve performance

- RAID**
- RAID 1: Rad time ≈ Write time; slow when read large file two multi blocks
  - RAID 1: 吞吐速度 = 1/2 RAID 0; 2 I/O for single write (same as RAID 0);
  - RAID 4/5 Read time ≠ Write Time (need time update odd-even disk); similar sequential Read speed to RAID 0; **4 I/O for write operation**; have **small write problem**
  - RAID 4: always need to check odd-even disk for write: can only perform 1 block a time (2 \* RW ≈ RW \* 2)
  - RAID benefit: better **performance**, larger **capacity**, improve **reliability**

- File Sys API**
- O\_Trunc() force to give up everything originally in file and set file size = 0 when read a file
  - lseek() change current file's offset
  - typical file **inode** contain: owner, size, block allocated, last access time, **NO** file name!
  - "read0" call is to the "cat" program as the "read0r()" call is to the "ls" program
  - UNIX file sys support **soft links**; a file is not easily deleted
  - Indirect block can point to "sizeof(block\_t) / sizeof(disk\_addr)" of blocks, each size 4kb; plus direct ptr 12 \* 4 = 48kb; so total file size = 48 + 4096 = 4144kb
  - in VFSs a directory contains a list of (file name, inode num) pairs
  - write buffering perf benefit: **batching, scheduling, avoid writes altogether**
  - what **unlink()** do: update data bitmap; inode bitmap; data block; inode metadata
  - VFSs have **NO Fast Crash Consistency**

- Flash-based SSD**
- cell → pages → blocks → flash bank(SSD); typical size of page = 4KB
  - once flash page programmed, it can't re-program until entire block erased
  - biggest SSD problem: wear out ("损耗"); garbage collection ensure dead page reuse
  - T\_entire write of pages = T\_erase block + T\_program 1 page \* # pages
  - T\_write = # pages \* T\_program 1 page
  - SSD write: if block write to already filled previously, write() will not change content store in previous block **firstly!** right after right still old data inside referenced block which will be deleted later through garbage collection
  - repeated writes are not counted when calculating live SSD pages

- Journalising**
- add a data blk to exist file will write:** data bitmap blk, inode blk, data blk to journal
  - read a block from file:** update metadata in inode blk → write to journal
  - create 0-byte file:** update inode blk, dir inode blk, dir data blk, inode bitmap,
  - delete 1-byte file:** update file inode , dir inode blk, dir data blk, inode bitmap, data bitmap
  - ordered mode journaling (only metadata): **user data** not written to journal
    - add data blk to exist file:** data bitmap blk, inode blk, data blk
    - reads a block from file:** only write inode blk to journal (change "last access time" )
    - create 0-byte file:** inode blk, dir inode blk, dir data blk, inode bitmap,
    - delete 1-byte file:** inode blk, dir inode blk, data bitmap blk, inode bitmap, dir data blk
    - data must be written to disk before **transaction commit block** ensure no garbage ptr
  - write journal: only update inode case: data may seemly read after recovery, but is garbage
  - ideally to do file sys move from one consistent state to another **atomically**
  - fsck() biggest problem: slow; *basic journaling protocol: journal write, journal commit, checkpoint*
  - os reboot & fs attempt write blk in transaction to final disk location & crashed: recovery start after reboot & will work correctly ;
  - data journaling** reduce performance **by factor of 2** during seq writes compare to **meta-data journaling**

- NFS**
- NFS largest benefit is **sharing**; server may crash due to **power outage, bug, network partition**
  - NFS protocol requests  stateless; **NFS file handle** consists of: **volume id, inode id, generation num**
  - NFS **client** handle packet loss & server crash using **timeout/retry** approach
  - NFS client **caches** subproblem (of consistency): **correctness; staleness**
  - "flush on close" behavior: visibility problem; client buffer write in mem for some time before flush to server; if crash then all pack flush to server

RAID 容量、可靠性和性能				
	RAID-0	RAID-1	RAID-4	RAID-5
容量	N	N/2	N-1	N-1
可靠性	0	1 (肯定)		
		N/2 (如果走运)		
吞吐量				
顺序读	N · S	(N/2) · S	(N-1) · S	(N-1) · S
顺序写	N · S	(N/2) · S	(N-1) · S	(N-1) · S
随机读	N · R	N · R	(N-1) · R	N · R
随机写	N · R	(N/2) · R	1/2 · R	N/4 · R <sub>256</sub>
	RAID-0	RAID-1	RAID-4	RAID-5
延迟				
读	T	T	T	T
写	T	T	2T	2T

- Disk = A % number of disks**; Offset = A / number\_of\_disks
- throughput = # disks \* Sequential bandwidth
- Any disk failure will cause data loss in RAID 0
- best performance & capacity; worst reliability
- capacity = N / level of mirroring
- R1: worst capacity, best reliability, worse performance than RAID 0
- Cat -> open read/write; read; When first open & read the file file type will be 3: a section to read and display; section to write; section to display fault data
- NOTE:** use lseek() will not cause disk I/O

## Spring 18 midterm

- Q1 - 5: When two **processes** run in parallel, their outputs can be arbitrarily interleaved. therefore, output from two processes can be **in any sequence**
- Q6 - 10: if program run on **separate processes**, their result will NOT impact each other.
- Q11 - 15: do not give assumption that the program will run indefinitely → NOT MENTIONED! it is checking results after one-time execution
  - because there is no locks implemented, there will be two cases:
    - two threads perform decrement one after another: minus 1 two times
    - decrement take place concurrently: only minus 1
  - therefore, the result of counter could only be 999 or 998
- Q16 - 20: note that in xv6, process state switch can only be: **Embryo → Ready; Ready → Running; Running → Blocked/Ready/Zombie** (I/O; or waiting var/semaphore) (decscheduled) (exited, but parent hasn't call wait() on it)
- child processes start running code after the parent's fork() call:**
  - Q21 - 25: parent process must execute before child process (what above fork() will always execute earlier than what below fork()); fork() creates a new process which is copy of existing process; therefore here: parent print 'a', then parent print b, then child print b; so, b must print out twice (impossible to print once given fork()never fail in this case)
  - Q26 - 30: if fork() possibly fail & return err code, the main() method will still execute. Therefore, b must be printing out once by parent process.
  - Q31 - 35: if fork() returns 0, the code chunk in if loop will be in child process; specifically in this question, if rc == 0, this means child process created successfully and we are in child process, so we will enter program /bin/true and this program will not print anything, just return 0; therefore, 1 will not print in any cases; so 2 will print after parent wait child process complete. So, the program will only possible to print 2.
  - Q36 - 40: same code chunk as above. NOTE in this case fork() or execv() or wait() could fail;
    - if fork() work, execv()fail, wait()work then we fail to enter /bin/true and hence able to print 1; therefore will print 1,2;
    - if fork() work, execv()fail, wait()fail, it would print 2, 1 (2 print first from parent as it does not need to wait anymore)
    - if fork() fails, regardless of other two it will directly print 3
    - Impossible to print 2,3 as this need fork() succeed & fail same time
- Q41 ~ 45: \$T\_{turnaround} = T\_{end} - T\_{arrive}\$
- 46~50, 51~55, 56~60, ... ~95 Review of Before Midterm
- Q96 ~ 100: **Lock:** because lock in this case is broken, all threads may acquire this lock at the same time (this is the very first implementation of lock according to chapter 28 textbook): the reason is if a scheduler has round robin policy and interrupt at inappropriate position it may enable all threads to acquire a usable lock and mark the flag as 1
- Q101 - 105: **Ticket Lock:** 1. ticket is incremented before turn. Therefore, we will never see a combination of a larger turn value with smaller ticket value; also, the difference between ticket and turn cannot be bigger than the total number of threads; 2. when a thread calls ticket lock, the ticket number increments
- Q106 - 110: **List Insertion:** no lock case: thread can utilize the List\_insert() one by one, then list length is of length 3; or 2 threads take place simultaneously and hence list length is 2; or all threads fetch the method simultaneously, and list length is 1. impossible to be 0 as there will be no errors and the method is fetched at least once
- Q111 ~ 115: the code update race result & access resulting value in main thread
- Q116 - 120: remind that the threads may run in any order without implementation of locks; but we indeed create 5 threads, and each thread will print out some value; **MOST IMPORTANT:** we allocate different mem location for variable char, so threads will access the printout variable individually (and thus secured)
- Q121 - 125: the only difference now is all created process is visiting same memory location that stores the printout variable. Thus, the location matters: the only impossible case is to access 5 'a' s in this question, as every time a new process is created, the printout value get updated
  - Mv -> use rename() to rename the file
  - Read file statistic (metadata): use stat() or fstat()
  - 64 block VFSs contains:
    - super blk \* 1 + inode bitmap \* 1 + data bm \* 1 + inode table \* 5 + data region \* 56
  - Each block size = 4kb
    - Super block: record how many data block, how many inode
    - Inode/data bitmap: series of 0/1 record each block is used or not
  - Disk can't find address by bit (consist of series of block region)
    - blk = (inumber \* sizeof(inode\_t)) / blockSize;
    - sector = ((blk \* blockSize) + inodeStartAddr) / sectorSize;
  - NOTE: read() no need to fetch bitmap

**NOTE:** write() don't force OS to write onto disk immediately. Fsync(fd) will force File system to do that

Time end = 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 <sup>5</sup> = 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 = 268,435,456
16 = 65,536	29 = 536,870,912
17 = 131,072	30 = 1,073,741,824

**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<sup>18</sup> entries. Each VPN has one entry, so 2<sup>18</sup> VPNs.

**Thread:** like a separate process, except they SHARE the same address space and thus

Two threads running on a single processor, T1 → T2: **context switch**.

**Critical Section:** piece of code that accesses a shared variable and must not be

1 thread.

**Atomically:** instruction executed not be interrupted in the middle → either instr not run at all or run to completion

**Lock:** mutually exclusiveness between 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 be both threads set the flag to 1
- spin-waiting wastes time waiting for another thread to release a lock = # of thread \* len(time slice)

**Lock:** Test-And-Set/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 }
```

- If lock free, flag = 0, T1 calls lock(): T1 calls TestAndSet(flag, 1), return the old value

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 repeatedly 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 preemptive schedule that will interrupt a thread via a timer.

**Lock:** Compare-And-Swap

// test whether the value at the addr in ptr is equal to expected, if so, update the memory location 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.
```

**Lock:** Fetch-And-Add (Fairness ☺)

// atomically increments a value while return the old value at a particular address

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

- tickets: T1 acquire 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.

**Yield:** 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 process schedules 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

**CV:** signaling between threads, have a lock associated with the condition, needs lock to be held calling wait and signal

- cond\_wait(&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 between lock acquire at the beginning of the wait seq and the lock release at the end, it will hold the lock.

- WHILE loop to check the wait condition (s.t. thread rechecks condition after being woken by signal from wait)

- cond\_signal(&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 between 2 threads

- **CV is a queue of waiting threads because some state of execution is not desired**

- 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

**Bounded Buffer:** block put when queue is full, consumer wakes producers but not other consumers, 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. Producer: while count == 1: cond\_wait(&empty, &mutex) then put when count == 0.

Consumer: 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;

**Semaphores:** lock and CV in one. Sem\_init(&lock, 0, 1): 0: semaphore shared between threads, 1: semaphore value

- Many threads to acquire at "same time": let 1 acquire and put others to wait

- Parent/Child aka fork/join: initialize semaphore w/ 0; always think about the 2 cases here.

- Read/Write: 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.

- Bounded buffer: sem\_init(&empty, 0, MAX), sem\_init(&fill, 0, 0), consumer wait for full and post empty

- sem\_wait(sem\_t \*s): value --; while value < 0: caller sleeps;

- sem\_post(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;

- Circular wait: set of processes 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

## 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)
  - o Releases lock associated with its mutex lock arg.
  - o 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)

- o Decrement the value of semaphore by 1

- o WAIT if the value of the semaphore is < 0.

- sem\_post(sem\_t \*sem)

- o Increment the value of the semaphore by 1

- o If there are one or more threads waiting, wake one.

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] 0x80000000 <- 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

concurrently executed by >

## 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.

**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.