REVIEW SESSION

Kai Mast

CS 537

Fall 2022

ANNOUNCEMENTS

- Extra Office Hours
  - Friday 12/16 1-2pm
  - Tuesday 12/20 3-4pm
  - More by request

- Final 12/21 @ 5:05pm

- Alternate final 12/19 @ 10am
  - Sign up by Friday (see Piazza post)

SSD SUMMARY

No mechanical parts unlike HDD

- An SSD contains blocks made of pages
- Best for fast random access

Types of operations

- Read
- Erase – Set all bits to 1
- Program (Write) – Clear some bits

Relies on FTL for fast writes

- Sequential and random writes close in cost
- Garbage collection more expensive with random writes

INTERACTION OF FILE DESCRIPTORS

```
int fd1 = open("file.txt"); // returns 12
int fd2 = open("file.txt"); // returns 13
read(fd1, buf, 16);
int fd3 = dup(fd2);         // returns 14
read(fd2, buf, 16);
lseek(fd1, 100, SEEK_SET);
```

What are the value of offsets in fd1, fd2, fd3 after the above code sequence?

100, 16, 16
(fd2 and fd3 reference the same in-kernel datastructure)

IDENTIFY THE KIND OF JOURNALING

In-place blocks: Need to write data bitmap in block 2, inode in block 4, data in blocks 5,6

Journal can run from blocks 8 to 15

B Data journaling

C Data journaling with transactional checksum

A Ordered journaling (writes data first, then logs metadata)

D Writeback journaling (logs metadata first, then writes data)

A

Write 5,6
Barrier (optional)
Write 8, 9, 10
Barrier
Write 11
Barrier
Write 4, 2

B

Write 8, 9, 10,11,12
Barrier
Write 13
Barrier
Write 2,4,5,6

C

Write 8, 9, 10,11,12, 13
Barrier
Write 2,4,5,6

D

Write 8, 9, 10
Barrier
Write 11
Barrier
Write 5,6
Write 4,2

HDD PERFORMANCE

Given a drive that having the following characteristics:
RPM - 7200
Average Seek - 9 ms
Max transfer - 105 MB/sec
Platters - 4
Capacity 300 GB

How long does an average random 16-KB read take?
(Round answer to closest millisecond)

Full Rotation Time: 1/ 7200 RPM = 1 / (7200 (1/min))
= 60/7200 * s ≈ 8ms

Avg. Read Time: Avg. Seek + Avg. Rotate = 9ms + (8ms/2) = 13ms

HDD SCHEDULING

Assume seek+rotate = 10 ms for random request. Assume the transfer time is negligible (0).
Assume FCFS (First-come-first-serve) scheduling algorithm is being used.

Imagine a workload that access the given sector numbers:

300001, 700001, 300002, 700002, 300003, 700003

How many seeks happen in this way? 6

How long (roughly) does the below workload take? 6*10ms = 60ms

HDD SCHEDULING

Assume seek+rotate = 10 ms for random request. Assume the transfer time is negligible (0). Assume FCFS (First-come-first-serve) scheduling algorithm is being used.

Now, imagine another workload that access the given sector numbers:

300001, 300002, 300003, 700001, 700002, 700003

How many seeks happen in this way? 2

How long (roughly) does the below workload take? 20ms

FILE SYSTEMS

- For this question about basic file systems, assume a simple disk model where each disk **read of a block takes D time units**. Also assume the basic layout is quite like the **very simple file system**.
- Assume that all data and metadata begin on disk. Assume further that all inodes are in separate blocks, and that each directory is only one block in size. How long does it take to **open the file /a/b/c/d.txt**?

Answer:

- 4 parent folders (need to read inode and data block for each)
- Need to also read inode for d.txt
- 9D

(From Fall ‘11 Final)

FILE SYSTEMS

- For this question about basic file systems, assume a simple disk model where each disk **read of a block takes D time units**. Also assume the basic layout is quite like the **very simple file system**.
- Assume **after opening the file**, we read the file in its entirety. It is a big **file, containing 1036 blocks**. The inode itself has room for 12 direct pointers and 1 indirect pointer. Disk addresses are 4 bytes long, and disk blocks are 4KB in size. After opening it, how long does it take to **read the entire file**?

Answer:

- 1024 data pointers per indirect block => Need one indirect block
- “Root” inode is already in memory
- Need to read 1036 data blocks
- Total of **1037D**

CONCURRENCY REVIEW

INTERRUPTS

One technique used to implement a lock was to turn off interrupts, say with a routine TurnOffInterrupt(), and then re-enable interrupts when unlocking TurnOnInterrupts(). Why is it dangerous to expose this functionality to user-level programs?

- Programs might never yield control back to the OS or other processes
- Additionally, disabling interrupts only works in systems with a single CPU

(From Spring ‘02 Final)

SEMAPHORE BASICS

```
sem_t s;
void child(void *arg) {
    // do some stuff, then signal completion
    sem_post(&s)
}
int main() {
    thread_t p1, p2, p3;
    sem_init(XXX);
    thread_create(&p1, child);
    thread_create(&p2, child);
    thread_create(&p3, child);
    // wait for all children to finish
    sem_wait(&s);
}
```

What should XXX be initialized to?

-2 (1 - #threads)

RACE CONDITIONS

```
cond_t c;
void child(void *arg) {
    printf("child\n");
    cond_signal(&c);
}

int main() {
    thread_t p1;
    thread_create(&p1, child, NULL);
    cond_wait(&c);
    printf("done\n");
    return 0;
}
```

What could this print?

`child` and `done`
or
just `child` and then gets stuck

SPIN LOCK

```
int m = 0; // global

void lock_acquire() {
    while (xchg(&m, 1) == 0)
        { /* spin */ }
}
void lock_release() {
    xchg(&m, 0);
}
```

What is wrong here?

We should wait until xchg returns 0, so change to
`xchg(&m, 1) == 1`

Help

```
int xchg(int *val, int new) {
    int old = *val;
    *val = new;
    return val;
}
```

CONCURRENT DATA STRUCTURES

```
typedef struct node {
     int key;
     node_t *next;
} node_t;
typedef struct {
     node_t *head;
} list_t;

void list_insert(list_t *l, int key) {
     node_t *new = malloc(sizeof(node_t)); // L1
     new->key = key; // L2
     new->next = l->head; // L3
     l->head = new; // L4
     mutex_unlock();
}
```

Where is the **best** place to put the `mutex_lock`?

Between L2 and L3

DEAD LOCKS

```
void function1() {
    mutex_lock(&outer);
    mutex_lock(&inner);
    function2();
    mutex_unlock(&inner);
    mutex_unlock(&outer);
}

void function2() {
    mutex_unlock(&outer);
    // do some stuff
    mutex_lock(&outer);
}
```

Multiple clients call `function1` at the same time.

Can this dead lock? If so, why?

Yes

1. Mutual exclusion
2. Threads may wait while holding a lock
3. No pre-emption (threads will wait holding the lock forever)
4. Circular dependency
    outer -> inner (in function 1)
    inner -> outer (in function 2)

VIRTUALIZATION REVIEW

SCHEDULING

What are the major strengths of the Round Robin approach?

- Most important: improves response time of jobs, as they get scheduled immediately
- very simple, and fair across jobs (no starvation for example)

What are its major weaknesses?

- Really bad for turnaround time, as it strings each job out as long as possible. P
- Possibly context-switch overhead too.

(From Fall ‘09 Midterm)

GAMING THE SCHEDULER

In this question, we explore the process of "gaming" the scheduler. In gaming, the idea is to **exploit aspects of the scheduler design** or the behavior of other processes so as of the CPU for your process.

a) Assume a priority-based scheduler that works as follows. A newly created process enters at the highest priority. If the process uses up its entire time quantum when it runs, it moves to the next lowest priority. If a process initiates an I/O request, it is moved back to the top priority, and its time quantum begins anew. How would you "game" this scheduler?

Answer:

Issue an I/O request at the end of every time slice.

(From Spring '02 Final)

GAMING THE SCHEDULER

In this question, we explore the process of "gaming" the scheduler. In gaming, the idea is to **exploit aspects of the scheduler design** or the behavior of other processes so as of the CPU for your process.

b) Assume there is a single lock in the system that most processes periodically grab, in order to examine some shared state. The lock is not a spin lock.
What could you do here in order to gain an unfair share of the CPU?

Answer:

Hold the lock the entire time, so all other processes eventually get moved to blocked.

GAMING THE SCHEDULER

In this question, we explore the process of "gaming" the scheduler. In gaming, the idea is to **exploit aspects of the scheduler design** or the behavior of other processes so as of the CPU for your process.

c) Assume a shortest-job first scheduler, which somehow "knows" the run-time of each process, and schedules the shortest ones first. How could you (re)write your programs so as to take advantage of this scheduler?

Answer:

- Split your program into multiple smaller processes

VIRTUAL MEMORY

- In this question, we'll see how long a simple **load instruction takes to execute**, given some different assumptions about the virtual memory system.
- Assume we have a linear page table per process, but the system we are using has **no TLB** and **no swapping to disk**. Assume each memory access takes M units of time. How long does it take to execute a single load instruction (e.g., mov VirtualAddress, register), in terms of memory accesses?

Answer:

- 2 fetches from memory (one for "mov" and one for "VirtualAddress")
- Each needs one page table load and one regular page load
- **4 M total**

(From Fall '11 Final)

VIRTUAL MEMORY

- In this question, we'll see how long a simple **load instruction takes to execute**, given some different assumptions about the virtual memory system.
- Assume now we have a two-level page table with a page directory. Assume each reference to the TLB is a miss, but that all referenced pages are found in memory (no swapping again). What is the slowest (i.e., worst case) time for the load instruction ?

Answer:

- 4 total fetches for page table (2 each)
- 2 total fetches for regular pages
- **6M total**

THAT'S ALL

Thank you so much for taking the class!