CS 3204
Operating Systems

Lecture 30

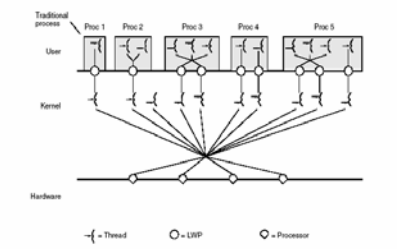
Godmar Back

Announcements

- Project 3 due April 13
- Th Apr 6, 7pm, 655 McBryde: attend town-hall meeting regarding planned restructuring of 6th floor undergrad space

Kernel-level vs User-level Threads

M:N Model



- Solaris Lightweight Processes

M:N Model (cont'd)

- Invented for use in Solaris OS early 90s
- Championed for a while
  - Idea was to get the best of both worlds
  - Fast context switches if between user-level threads
  - Yet enough concurrency to exploit multiple CPUs
- Since abandoned in favor of kernel-level threads only approach
  - Too complex – what's the "right" number of LWP?
  - 2-level scheduling/resource management was hard: both user/kernel operated with half-blind

Multi-Threading in Linux

- Went through different revisions
- Today (Linux 2.6): NPTL – Next-Generation POSIX Thread Library
- 1:1 model
- optimizes synchronization via "futexes"
  - avoids mode switch for common case of uncontended locks by performing atomic operation
  - constant-time scheduling operation allow for scaling in number of threads

Summary

- Memory Management
- Address Spaces vs Protection Domains
- Kernel vs User-Level Threads

Disks & Filesystems

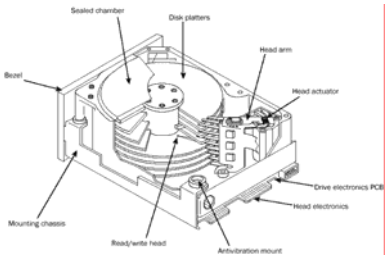
What Disks Look Like

Hitachi Deskstar T7K500 SATA

How Disks Work

- Flash Animation
- See http://cis.poly.edu/cs2214rvs/disk.swf

Disk Schematics



Source: Micro House PC Hardware Library Volume I: Hard Drives

Typical Disk Parameters

- 2-30 heads (2 per platter)
- Diameter: 2.5” – 14”
- Capacity: 20MB-500GB
- Sector size: 64 bytes to 8K bytes
  - Most PC disks: 512 byte sectors
- 700-20480 tracks per surface
- 16-1600 sectors per track

What's important about disks from OS perspective

- Disks are big & slow - compared to RAM
- Access to disk requires
  - Seek (move arm to track) – to cross all tracks anywhere from 20-50ms, on average takes 1/3.
  - Rotational delay (wait for sector to appear under track) 7,200rpm is 8.3ms per rotation, on average takes ½: 4.15ms rot delay
  - Transfer time (fast: 512 bytes at 998 Mbit/s is about 3.91us)
- Seek+Rot Delay dominates
- Random Access is expensive
  - and unlikely to get better
- Consequence:
  - avoid seeks
  - seek to short distances
  - amortize seeks by doing bulk transfers

Disk Scheduling

- Can use priority scheme
- Can reduce avg access time by sending requests to disk controller in certain order
  - Or, more commonly, have disk itself reorder requests
- SSTF: shortest seek time first
  - Like SJF in CPU scheduling, guarantees minimum avg seek time, but can lead to starvation
- SCAN: "elevator algorithm"
  - Process requests with increasing track numbers until highest reached, then decreasing etc. – repeat
- Variations:
  - LOOK – don't go all the way to the top without passengers
  - C-SPAN: - only take passengers in one direction

Accessing Disks

- Sector is the unit of atomic access
- Writes to sectors should always complete, even if power fails
- Consequence:
  - Writing a single byte requires read-modify-write

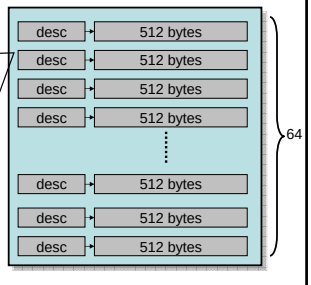
```
void set_byte(off_t off, char b) {
    char buffer[512];
    disk_read(disk, off/DISK_SECTOR_SIZE, buffer);
    buffer[off % DISK_SECTOR_SIZE] = b;
    disk_write(disk, off/DISK_SECTOR_SIZE, buffer);
}
```


Disk Caching – Buffer Cache

- How much memory should be dedicated for it?
  - In older systems (& Pintos), set aside a portion of physical memory
  - In newer systems, integrated into virtual memory system: *e.g.*, page cache in Linux
- How should prefetching be done?
- How should concurrent access be mediated (multiple processes may be attempting to write/read to same sector)?
  - How is consistency guaranteed? (All must go through buffer cache!)
- What write-back strategy should be used?

Buffer Cache in Pintos

Cache Block Descriptor
- disk_sector_id, if in use
- dirty bit
- valid bit
- # of readers
- # of writers
- # of pending read/write requests
- lock to protect above variables
- signaling variables to signal availability changes
- usage information for eviction policy
- data (pointer or embedded)



A Buffer Cache Interface

```
// cache.h
struct cache_block;                    // opaque type
// reserve a block in buffer cache dedicated to hold this sector
// possibly evicting some other unused buffer
// either grant exclusive or shared access
struct cache_block * cache_get_block (disk_sector_t sector, bool exclusive);
// release access to cache block
void cache_put_block(struct cache_block *b);
// read cache block from disk, returns pointer to data
void *cache_read_block(struct cache_block *b);
// fill cache block with zeros, returns pointer to data
void *cache_zero_block(struct cache_block *b);
// mark cache block dirty (must be written back)
void cache_mark_block_dirty(struct cache_block *b);
// not shown: initialization, readahead, shutdown
```