Old Tricks…

- speed disparity between main memory and disks?
- where else have we seen speed disparity?
Old Tricks…

- speed disparity between main memory and disks?
- where else have we seen speed disparity?
  - CPU ↔ memory
- what did we use to mitigate performance problems arising from the speed disparity?
  - cache
- will this work for disks?
Disk Cacheing

- keep in-memory cache of disk sectors
- on disk read request:
  - copy cached disk block to user process space
  - send ptr to proc, pointing at cached copy

- replacement policy:
  - LRU: widely used, stack implementation
Disk Cacheing

- keep in-memory cache of disk sectors
- on disk read request:
  - copy cached disk block to user process space
  - send ptr to proc, pointing at cached copy
- replacement policy:
  - LRU: widely used, stack implementation
  - FIFO: also popular (easy to build) but has ‘poor-choice’ problem
- modified cached sectors must be written before cache slot is re-used

Disk Cacheing: Reading and Writing

- write request:
  - update cached copy of disk block
- cached copies must be written out (coherency)
- can batch writes
What Happens If…

- suppose we are using disk caching
- at some moment k of the blocks have been modified and are not yet written to disk
- there is a power failure
- on reboot, what do we see on the disk?

Disk Consistency

- cached data not flushed does not appear on disk
  ⇒ loss of data
  - you did two bank transactions on your account
  - one debit, which was successfully written to disk
  - one deposit, which did not get written to disk
  - what’s your account balance?
Disk Consistency

- cached data not flushed does not appear on disk
  ⇒ loss of file system integrity
    • new block of data added to a 2-Kbyte UNIX file
    • block was successfully written
    • inode was not successfully written out to disk ilist
    • where’s the new file block?

Disk Consistency

- two concerns:
  1. actual data integrity
  2. file system integrity
- not just a cache–related issue
  • power may fail as drive is part–way through the write operation
- problem: OS doesn’t know that I/O operation did not complete successfully
- solution?
Regular Updates

- UNIX has sync command that flushes all modified disk-bound traffic to disk
  - sync is not a privileged command; why not?
- regular running of the sync command can minimize amount of ‘lost’ data
  - e.g., every 5 seconds
- 5 seconds is enough time to develop plenty of disk inconsistency
- running sync more often degrades performance
  - defeating utility of cache

Transaction-Based I/O

- use transaction-based I/O operations
  - each I/O request logged as it arrives in special log file (journal file)
    - as I/O ops complete successfully the logged transaction is marked as completed
- in event of failure,
  - last transaction can be retried
  - last transaction can be un-done
  - goal: restore filesystem to the consistent state it was in at time of failure
- used in both UNIX and Windows I/O systems
UNIX filesystem

- UNIX uses “buffer caches”
- DMA between disk and cache, between cache and user space on copies
- block size?
  - too small, and performance suffers (e.g., 512 bytes)
  - too large, and fragmentation rises
  - UNIX systems tend to have lots of small files

UNIX filesystem

- block size?
  - too small, and performance suffers (e.g., 512 bytes)
  - too large, and fragmentation rises
  - UNIX systems tend to have lots of small files
- drop “one size fits all” model
  - introduce larger block size (e.g., 4 Kb)
  - with fragments (e.g., 512 bytes)
- fragments used for small files
  - not allowed to extend across blocks
- size of fragments and blocks can be set per partition
UNIX filesystem

- inodes contain 15 file block pointers
  - 12 direct block pointers
  - 1 for each of single, double, tertiary indirect
- max file size is $2^{31}$ bytes
  - 32-bit file pointers are signed integers
- a file is assumed to have no particular internal structure
- unless it is a directory
  - distinguished by ‘d’ bit set in mode field in inode
  - contains <file_name><inode_number> pairs

- unless it is a directory
  - distinguished by ‘d’ bit set in mode field in inode
  - contains <file_name><inode_number> pairs
  - file_name can be up to 255 characters long
  - every directory has at least two entries:
    - ‘.’ the directory itself
    - ‘..’ its parent directory
UNIX filesystem

- an open() requests:
  - kernel to load inode for file into memory
  - kernel to allocate a structure for managing the open file
- this structure contains, among other things:
  - file pointer: points to byte at “current” position in file
    - is automatically incremented on read(), write()
    - can be moved to arbitrary location within file using lseek()
- why isn’t this pointer in the inode?

UNIX filesystem

- a disk contains ≥ 1 partitions
- each partition = 1 filesystem
- advantages of multi-partitions per drive:
  - different filesystem types for different needs
    - e.g., data files, swap
  - reliability: software bugs can only affect one partition
  - maintenance: backups and restores are faster
Supported File System Types (linux)

1  FAT12 1e  Hidden Win95 FA 70  DiskSecure Mult c1  DRDOS/sec (FAT-
2  XENIX root 24  NEC DOS 75  PC/IX c4  DRDOS/sec (FAT-
3  XENIX usr 39  Plan 9 80  Old Minix c6  DRDOS/sec (FAT-
4  FAT16 <32M 3c PartitionMagic 81  Minix / old Lin c7  Syrinx
5  Extended 40  Venix 80286 82  Linux swap da  Non-FS data
6  FAT16 41  PPC PreP Boot 83  Linux db  CP/M / CTOS / .
7  HPFS/NTFS 42 SFS 84  OS/2 hidden C: de Dell utility
8  AIX 4d  QNX4.x 85  Linux extended df BootIt
9  AIX bootable 4e  QNX4.x 2nd part 86  NTFS volume set e1 DOS access
a  OS/2 Boot Manager 4f  QNX4.x 3rd part 87  NTFS volume set e3 DOS R/O
b  win95 FAT32 50 OnTrack DM 8e  Linux LVM e4 SpeedStor
c  win95 FAT32 (LB 51 OnTrack DM6 Aux 93 Ameoba eb BeOS fs
d  win95 FAT16 (LB 52 CP/M 94 Ameoba BBT ee EFI GPT
e  Win95 Ext’d (LB 53 OnTrack DM6 Aux 9f BSD/OS ef EFI (FAT-12/16/
f  OPUS 54 OnTrackDM6 a0 IBM Thinkpad hI f0 Linux/PA-RISC b
g  Hidden FAT12 55 EZ-Drive a5 FreeBSD f1 SpeedStor
h  Compaq diagnostic 56 Golden Bow a6 OpenBSD f4 SpeedStor
i  Hidden FAT16 c3 Priam Edisk a7 NEXTSTEP f2 DOS secondary
j  Hidden HPFS/NTFS 63 GNU Hurd or Sys b7 BSDI fs fe LANstep
k  AST SmartSleep 64 Novell Netware b8 BSDI swap ff BBT
l  b  Hidden Win95 FA

CS571 Operating Systems 29 Oct 2003 © C. M. Snow 2003 09.22

UNIX filesystem

- partition structure:
  - boot block: bootstrap info
  - superblock: contains free block list
  - ilist: the inodes in this partition
  - data blocks: for use in files
- data blocks for a file may be taken from anywhere
  - consecutive file blocks may be far from each other
  - gets worse as files come and go on disk
UNIX filesystem

- to mitigate such scattering of file blocks, introduce concept of cylinder group: small number of physically contiguous cylinders
- has its own:
  - superblock
  - cylinder block: dynamic parms for that CG
  - array of inodes
  - data blocks
- inodes for files in a directory are chosen from same CG as the directory
  - new directories chosen from different CGs

NTFS

- uses many of newest features we've seen
- particularly design emphasis on recoverability
  - uses transaction-based I/O
  - uses redundant info for recovery of filesystem integrity in event of sector loss
NTFS

- storage organized in levels
- sector:
  - smallest possible unit of allocation
  - always power of 2 in size, usually 512 bytes
  - largely ignored by NTFS as too small to be useful
- cluster:
  - $\geq 1$ physically contiguous sectors
  - identified by LCN (logical cluster number)
  - default cluster size depends on size of volume

NTFS

- volume:
  - a logical disk partition
  - contains $\geq 1$ cluster
  - has:
    - filesystem information
    - collection of files
    - free space (allocatable to files)
  - if using RAID5, volume is stripes spanning multiple physical disks
  - max volume size is $2^{64}$ bytes
  - files defined as set of attribute:value pairs
NTFS

- volume layout (partial):
  - MFT
  - MFT mirror
  - log file
  - volume info
  - attributes definition table
  - root directory
  - cluster map
  - boot record
  - bad cluster list
  - file blocks

  Master File Table: one entry per file in volume

  "system files"

NTFS' MFT

- one entry per file, organized like rows in rel db
- includes an entry for MFT itself
- rows are variable length
- some fields in MFT records:
  1. standard information:
     - access attributes, e.g., read, write
     - times of read, access, create and modify
     - version number, max version number
     - sequence number
  2. file name: UNICODE, variable length
NTFS’ MFT

3. security descriptor:
   • user and group identifiers: who owns this object
   • audit and permissions access control lists

4. data
   • contents of the file itself (start)
   • if data too large to fit in MFT record, it continues elsewhere (other available clusters in volume)
   and MFT record has pointers to those clusters

NTFS

**volume layout (partial):**

- MFT
- MFT mirror
- log file
- volume info
- attributes definition table
- root directory
- cluster map
- boot record
- bad cluster list
- file blocks

copy of first 4 rows of MFT (to provide redundant access mechanism to MFT in case of single-sector failure)
### NTFS

- **volume layout (partial):**

<table>
<thead>
<tr>
<th>MFT</th>
<th>MFT mirror</th>
<th>log file</th>
<th>volume info</th>
<th>attributes definition table</th>
<th>root directory</th>
<th>cluster map</th>
<th>boot record</th>
<th>bad cluster list</th>
<th>file blocks</th>
</tr>
</thead>
</table>

- Log file contains transactions posted on the volume.

### NTFS

- **volume layout (partial):**

<table>
<thead>
<tr>
<th>MFT</th>
<th>MFT mirror</th>
<th>log file</th>
<th>volume info</th>
<th>attributes definition table</th>
<th>root directory</th>
<th>cluster map</th>
<th>boot record</th>
<th>bad cluster list</th>
<th>file blocks</th>
</tr>
</thead>
</table>

- Volume name (in UNICODE), flag info (like needs consistency check);
NTFS

- **volume layout (partial):**
  - MFT
  - MFT mirror
  - log file
  - volume info
  - attributes definition table
  - root directory
  - cluster map
  - boot record
  - bad cluster list
  - file blocks

  **attributes defined on this volume**

NTFS

- **volume layout (partial):**
  - MFT
  - MFT mirror
  - log file
  - volume info
  - attributes definition table
  - root directory
  - cluster map
  - boot record
  - bad cluster list
  - file blocks

  **root directory of volume**
NTFS

- Volume layout (partial):
  - MFT
  - MFT mirror
  - Log file
  - Volume info
  - Attributes definition table
  - Root directory
  - Cluster map
  - Boot record
  - Bad cluster list
  - File blocks

Cluster map: 1 bit per LCN marking it as used/unused

-points at boot data; cannot be relocated
**NTFS**

- volume layout (partial):
  - MFT
  - MFT mirror
  - log file
  - volume info
  - attributes definition table
  - root directory
  - cluster map
  - boot record
  - bad cluster list
  - file blocks

  *in essence, a file containing the bad clusters in the volume*

---

**Non-FS Disk Usage**

- not all disk usage is file-system related
- swap-file, paging-space: disk usage pattern is different
- view paging swap partition as sequence of pages
  - holds page images swapped out
  - first page is bit-map of swap space showing which page areas are available, which are not
    - because in use, or bad blocks, or this index page
- virtual mem mgr keeps track of which memory page in which disk page

<table>
<thead>
<tr>
<th>swap dev #</th>
<th>blk #</th>
<th>storage type</th>
</tr>
</thead>
</table>

© Charles M Snow 19
Process Scheduling

- when do we give resources to a process?
- we’ve seen some elements of this already with:
  - handling deadlock
  - process placement in fixed-allocation-with-different-sizes memory
  - basic “life of process” model

Scheduling Levels

- newly created
- ready
- running
- exit
- release
- suspended
- blocked

- long term
- medium term
- short term
Scheduling Levels

1. long term:
   • how to add processes to set of processes to be processed
2. medium term:
   • how to add processes to set of processes now partially or completely in main memory
3. short term:
   • how to select a runnable process for execution
4. I/O:
   • how to choose which process' I/O request should be processed by available I/O device

Long Term Scheduler

- controls degree of multiprogramming
- two-step decision:
  • can system take on more work?
    - more processes ⇒ less % CPU time to procs ⇒ more real time to run
    - check as processes exit: can more come in?
    - if CPU idle time is above some threshold, bring in more
  • which process from those queued waiting to enter should be selected
    - try to pick for current load conditions?
Medium Term Scheduler

- includes swapping function
  - when to block a process waiting for an event
  - how and which to pick when not using virtual memory

Short Term Scheduler

- a.k.a. “dispatcher”
- used whenever an event occurs which might
  - interrupt current process
  - allow pre-emption of current proc for another
- two perspectives to assessing...
STS: User Perspective

- things users care about:
  - response time: time between submitting request and start of some response
    - e.g., IBM study shows for timesharing system, should be $\leq 2$ seconds
  - turnaround time: time between submitting request and completion of servicing request
    - includes actual processing time + all waiting times
**STS: User Perspective**

- things users care about:
  - **response time**: time between submitting request and start of some response
  - **turnaround time**: time between submitting request and completion of servicing request
  - **predictability**: same job should run in about same time, at same cost, regardless of system load
    - resource use time & cost
  - **deadlines**: if provided, STS takes all measures to maximize percentage of procs complete within deadline
STS: System Perspective

- things system cares about:
  - **throughput**: rate at which processes are complete
    - depends on avg. length of process, on scheduler
  - **CPU utilization**: % time CPU not idle
    - more important to super-computer owner than desktop computer owner
STS: System Perspective

- things system cares about:
  - throughput: rate at which processes are complete
  - CPU utilization: % time CPU not idle
  - fairness: in absence of specific direction, all procs should be treated equally, no starvation
  - priority: when priorities used, always honour high priority requests first
STS: System Perspective

- things system cares about:
  - throughput: rate at which processes are complete
  - CPU utilization: % time CPU not idle
  - fairness: in absence of specific direction, all procs should be treated equally, no starvation
  - priority: when priorities used, always honour high priority requests first
  - balancing resources: aim to keep everything busy enough, but not too busy
    - involves medium and long term schedulers

Doing STS

- selection of a process:
  - priority
  - resource requirements
  - execution characteristics
Doing STS

- decision mode: when to apply selection method
  1. **pre-emptive**: a running process may be interrupted and moved from running to ready
     - e.g., timeslice interrupt
  2. **non pre-emptive**: once a process is running, it continues until done, or blocked (for I/O or OS service)

STS Policies: FCFS (FIFO)

- simplest, easiest to build
- fair?
  - great if you have long, CPU-bound process
  - not great if you are short job behind long, CPU-bound process
- non-preemptive
  - short jobs ‘penalized’
  - I/O intensive jobs ‘penalized’
- by itself, not a great policy
  - enhanced when used with multiple-priority queues
STS Policies: Round Robin

- simple improvement to FCFS: use preemption with clock
- run process until timeslice interrupt, return proc to end of queue, pick next head-of-queue item
- good choice for time-sharing and transaction processing systems

STS Policies: Round Robin

- central design question: how long should timeslice (a.k.a. quantum) be?
  - short: short processes move through system quickly vs. overhead of context switch: expensive if done frequently (quantum too short)
  - long: too long and degenerates into FIFO; as gets longer, tends towards problems of FIFO
STS Policies: Round Robin

- fair?
  - not so good for I/O intensive processes in mixed setting (both CPU intensive and I/O intensive present)
- overcome with variant: VRR: virtual round robin
  - introduce auxiliary queue holding “ready after I/O” processes: these have priority over regular queue
  - process from new queue gets (quantum – already used) timeslice to work with

STS Policies: Virtual Round Robin

![Diagram of virtual round robin scheduling with auxiliary queue]
STS Policies: Feedback

- uses multiple queues
- all processes enter same queue
- a process runs FCFS from this queue until blocked or preempted; then, it is moved to next lower-priority queue
- next process run is taken from entry queue; if none there take from second queue
- when proc taken from second queue, runs until blocked or preempted, then retired to third queue, etc.
- last queue cycles: is RR

arriving processes enter highest priority queue (is FCFS)
STS Policies: Feedback

when blocked or preempted, process enters next lower priority queue: will only be serviced if no higher priority processes to run

suppose no new entries, and highest priority queue is now empty:

when blocked or preempted, process enters next lower priority queue: will only be serviced if no higher priority processes to run
STS Policies: Feedback

- favours short jobs: these run through quickly
- long jobs can starve
- variations are possible:
  - do preemption at regular intervals, timeslices
  - preemption interval depends on queue level, typically $2^i$ where $i$ is queue level
    - so top queue runs 1 time unit, second queue (when it gets service) runs 2 time units, etc.
    - aims to mitigate starvation of older, longer-running procs
  - can move processes up 1 queue if have waited too long without service
STS Policies: SJF

- shortest job first
  - a.k.a. shortest process next (SPN)
- non-preemptive
  - who is favoured?
- depends on knowing cpu requirements of procs
- for some processes times may be known
  - or measured from previous runs
- what to do when can’t know remaining processing time?

what to do when can’t know remaining processing time?

- guess
  - e.g., interactive process: estimate is burst-based
    - guess how much cpu time process next needs based on running average of time so far
    - estimate of cpu time needed next:

\[ \tau_i \text{ estimates } \tau_{n+1} = \frac{1}{n} \sum_{i=0}^{n} T_i \text{ observations} \]
STS Policies: SJF

- reformulate to avoid doing summation:

\[
\tau_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} \tau_n
\]

- gives equal weighting to current and past bursts

STS Policies: SJF

- use exponential weighting to allow adjustment for bias in favour of recent or past behaviour:

\[
\tau_{n+1} = \alpha T_n + (1-\alpha) \tau_n, \quad 0 < \alpha < 1
\]

- e.g., with \( \alpha = 0.8 \), nearly all weighting is 4 more recent observations
  - hence higher \( \alpha \) \( \Rightarrow \) more rapid response to requirements of newly arrived processes
STS Policies: SJF

- can starve long running processes if many short jobs available
- not suited to:
  - OLTP
  - interactive

STS Policies: SRT

- SPN with preemption
- always pick process with shortest expected remaining time
  - might be newest arrival
  - like SPN, may starve long running processes
  - better turnaround time than SPN
Scheduling Summary

- long term: control admission of jobs
  - e.g., WS, “L=S criterion”
- medium term: between memory and secondary storage
- short term: between running and ready
  - FIFO
  - RR, VRR
  - Feedback
  - SJF (SPN)
  - SRT