An Analysis of Linux Scalability to Many Cores

What are we going to talk about?

- Scalability analysis of 7 system applications running on Linux on a 48-core computer
  - Exim, memcached, Apache, PostgreSQL, gmake, Psearchy and MapReduce
- How can we improve the traditional Linux for better scalability

Introduction

- Popular belief that traditional kernel designs won’t scale well on multicore processors
- Can traditional kernel designs be used and implemented in a way that allows applications to scale?

Why Linux? Why these applications?

- Linux has a traditional kernel design and the Linux community has made a great progress in making it scalable
- The chosen applications are designed for parallel execution and stress many major Linux kernel components

How can we decide if Linux is scalable?

- Measure scalability of the applications on a recent Linux kernel
    - 2.6.35-rc5 (July 12,2010)
- Understand and fix scalability problems
- Kernel design is scalable if the changes are modest

Kind of problems

- Linux kernel implementation
- Applications' user-level design
- Applications' use of Linux kernel services

The Applications

- 2 Types of applications
  - Applications that previous work has shown not to scale well on Linux
    - Memcached, Apache and Metis (MapReduce library)
  - Applications that are designed for parallel execution
    - gmake, PosgtreSQL, Exim and Psearchy
- Use synthetic user workloads to cause them to use the kernel intensively
  - Stress the network stack, file name cache, page cache, memory manager, process manager and scheduler

Exim

- Exim is a mail server
- Single master process listens for incoming SMTP connections via TCP
- The master forks a new process for each connection
- Has a good deal of parallelism
- Spends 69% of its time in the kernel on a single core
- Stresses process creation and small file creation and deletion

memcached – Object cache

- In-memory key-value store used to improve web application performance
- Has key-value hash table protected by internal lock
- Stresses the network stack, spending 80% of its time processing packets in the kernel at one core

Apache – Web server

- Popular web server
- Single instance listening on port 80.
- One process per core – each process has a thread pool to service connections
- On a single core, a process spends 60% of the time in the kernel
- Stresses network stack and the file system

PostgreSQL

- Popular open source SQL database
- Makes extensive internal use of shared data structures and synchronization
- Stores database tables as regular files accessed concurrently by all processes
- For read-only workload, it spends 1.5% of the time in the kernel with one core, and 82% with 48 cores

gmake

- Implementation of the standard make utility that supports executing independent build rules concurrently
  - Unofficial default benchmark in the Linux community
- Creates more processes than cores, and reads and writes many files
- Spends 7.6% of the time in the kernel with one core

Psearchy – File indexer

- Parallel version of searchy, a program to index and query web pages
- Version in the article runs searchy indexer on each core, sharing a work queue of input files

Metis - MapReduce

- MapReduce library for single multicore servers
- Allocates large amount of memory to hold temporary tables, stressing the kernel memory allocator
- Spends 3% of the time in the kernel with one core, 16% of the time with 48 cores

Quick intro to Linux file system

- Superblock - The superblock is essentially file system metadata and defines the file system type, size, status, and information about other metadata structures (metadata of metadata)

- Inode - An inode exists in a file system and represents metadata about a file.

- Dentry - A dentry is the glue that holds inodes and files together by relating inode numbers to file names. Dentries also play a role in directory caching which, ideally, keeps the most frequently used files on-hand for faster access. File system traversal is another aspect of the dentry as it maintains a relationship between directories and their files.
- Taken from: http://unix.stackexchange.com/questions/4402/what-is-a-superblock-inode-dentry-and-a-file

Common problems

- The tasks may lock shared data structures, so that increasing the number of cores increases the lock wait time

- The tasks may write a shared memory location, so that increasing the number of cores increases the time spent waiting for the cache coherence protocol

Common problems - cont

- The tasks may compete for space in a limited size shared hardware cache, so that increasing the number of cores increases the cache miss rate

- The tasks may compete for other shared hardware resources such as DRAM interface

- There may be too few tasks to keep all cores busy

Cache related problems

- Many scaling problems are delays caused by cache misses when a core uses data that other core have written
- Sometimes cache coherence related operation take about the same time as loading data from off-chip RAM
- The cache coherence protocol serializes modifications to the same cache line

Multicore packet processing

- The Linux network stack connects different stages of packet processing with queues
  - A received packet typically passes through multiple queues before arriving at per-socket queue
- The performance would be better if each packet, queue and connection be handled by just one core
  - Avoid cache misses and queue locking
- Linux kernels take advantage of network cards with multiple hardware queues

Multicore packet processing (2)

- Transmitting – place outgoing packets on the hardware queue associated with the current core
- Receiving – configure the hardware to enqueue incoming packets matching a particular criteria (source ip and port) on a specific queue
  - Sample outgoing packets and update hardware's flow directing tables to deliver incoming packets from that connection directly to the core

Sloppy counters – The problem

- Linux uses shared counters for reference counting and to manage various resources

- Lock-free atomic inc and dec do not help because of cache coherence

Sloppy counter – The solution

- Each core holds a few spare references to an object
  - It gives ownership of these references to threads running on that core when needed, without having to modify the global reference count

Sloppy counter - cont

- Core increments the sloppy counter by V:
  1. If $local\ count \geq V$
     I. Get V references and decrement $local\ count$ by V and finish
  2. Acquire $U \geq V$ references from the central counter and increment the central counter by U
- Core decrements the sloppy counter by V:
  1. Release V references for local use and decrement the local counter by V
  2. If $local\ count \geq threshold$ release spare references by decrementing local count and central count

Sloppy counter - cont

- Invariant:
  - $\sum local\ counters$+number of used resources = shared counter

Sloppy counter - use

- These counters are used for counting references to:
  - *dentrys*
  - *vfsmounts*
  - *dst_entrys*
  - *track amount of memory allocated by each network protocol (such as TCP and UDP)*

Lock-free comparison

- There are situations where there are bottlenecks because of low scalability of name lookups in the dentry cache
  - The dentry cache speed ups lookup by mapping a directory and a file name to a dentry identifying the matching inode
  - When a potential dentry is located, the lookup code acquires a per-dentry spin lock to atomically compare fields of the dentry with the arguments

Lock-free comparison - cont

- The search can be made lock-free
  - Use generation counter which is incremented after every modification. During modification temporarily set the generation counter to 0.
  - Comparison algorithm:
    - If the generation counter is 0, fall back to the locking protocol. Otherwise remember the value of the generation counter.
    - Copy the fields of the `dentry` to local variables. If the generation afterwards differs from the remembered value, fall back to the locking protocol.
    - Compare the copied fields to the arguments. If there is a match, increment the reference count unless it is 0, and return the `dentry`. If the reference count is 0, fall back to the locking protocol.

Per core data structures

- Kernel data structures that caused scaling bottlenecks:
  - Per super-block list of open files
  - Table of mount points
  - Pool of free packet buffers

False sharing

- Some applications caused false sharing in the kernel

A variable the kernel updated often was located on the same cache

line as a variable it read often

Evaluation

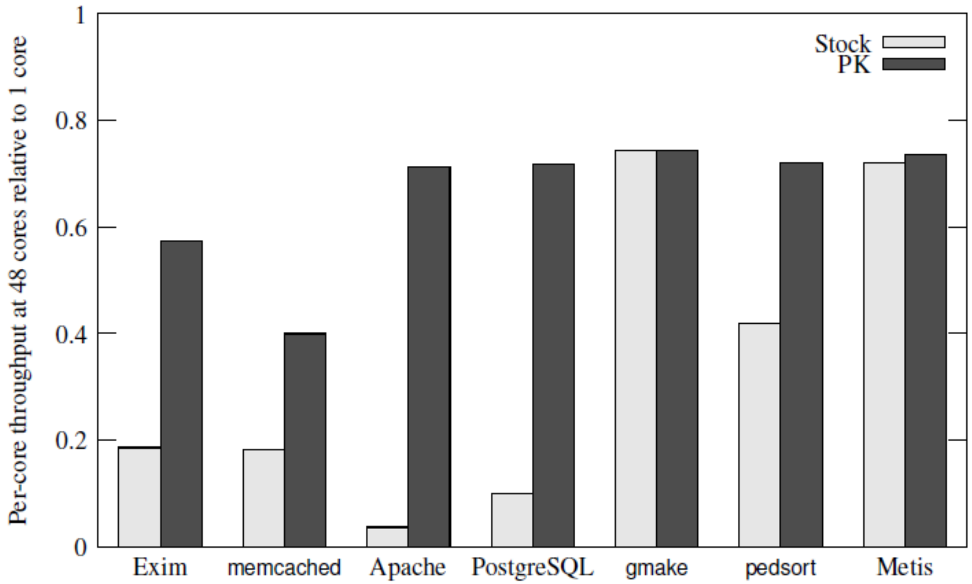


Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.

Technical details

- The experiments were made on a 48 core machine
  - Tyan Thunder S4985 board
  - 8*(2.4 GHz 6-core AMD Opteron 8431 chips)
  - Each core has 64Kb L1 cache and 512Kb L2 cache
  - The cores on each chip share 6Mb L3 cache
  - Each chip has 8Gb of local off-chip DRAM

Exim

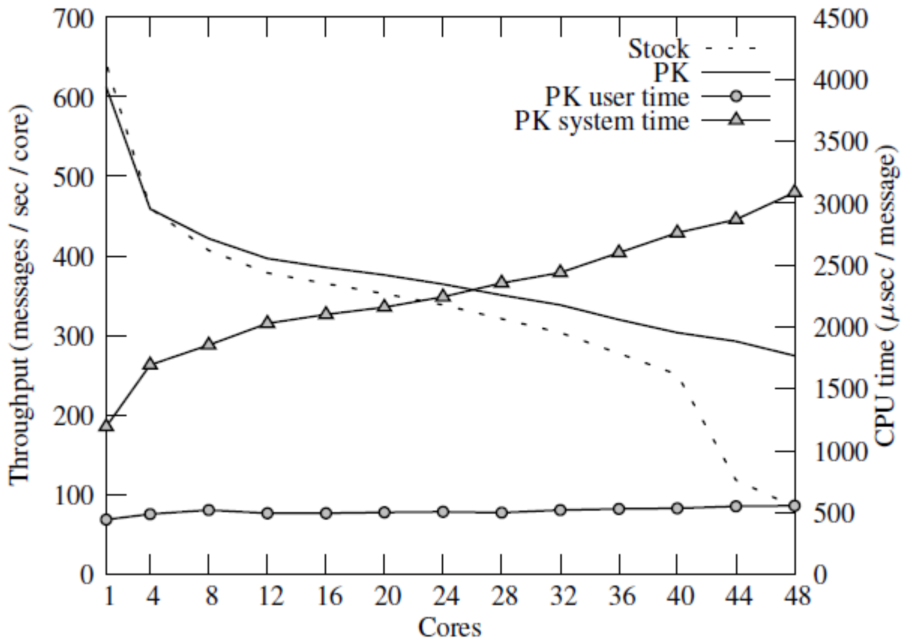


Figure 4: Exim throughput and runtime breakdown.

Exim - modifications

- Berkeley DB reads /proc/stat to find number of cores
  - Modification: Cache this information aggressively
- Split incoming queues messages across 62 spool directories, hashing by per connection pid

memcached

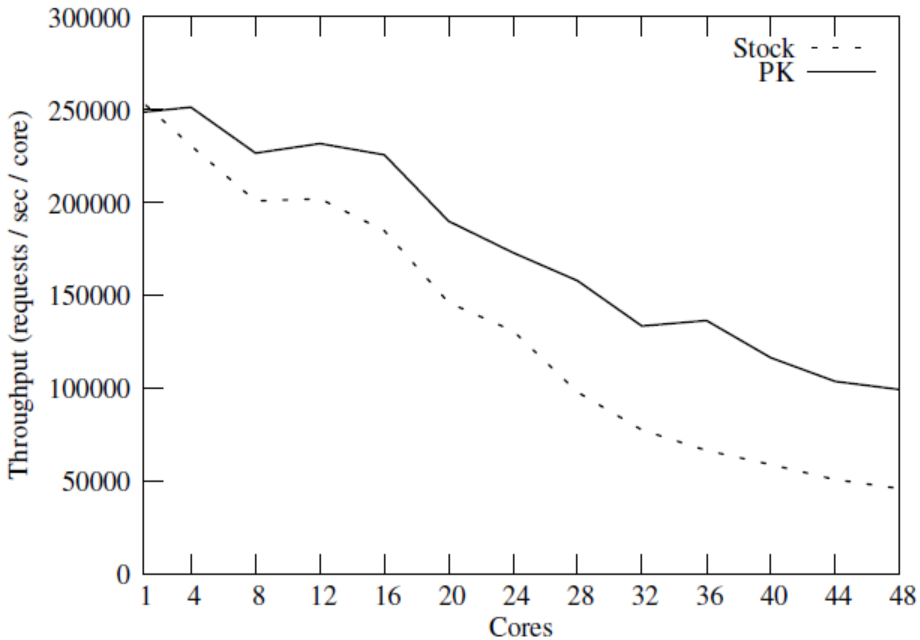


Figure 5: memcached throughput.

memcached - modifications

- False read/write sharing of IXGBE device driver data in the net_device and device structures
  - Modification: rearrange structures to isolate critical read-only members to their own cache lines
- Contention on dst_entry structure's reference count in the network stack's destination cache
  - Modification: use sloppy counter

Apache

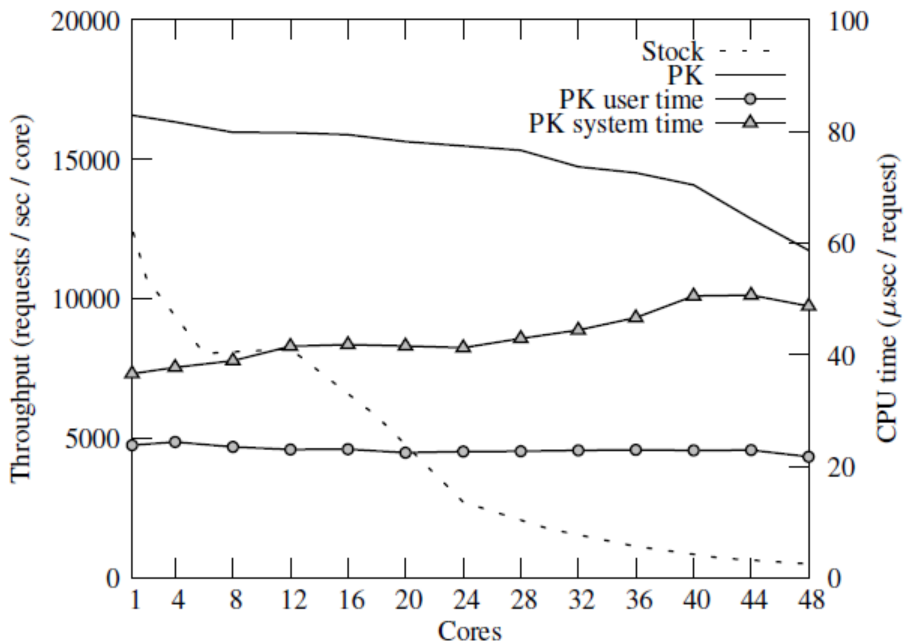


Figure 6: Apache throughput and runtime breakdown.

PostgreSQL

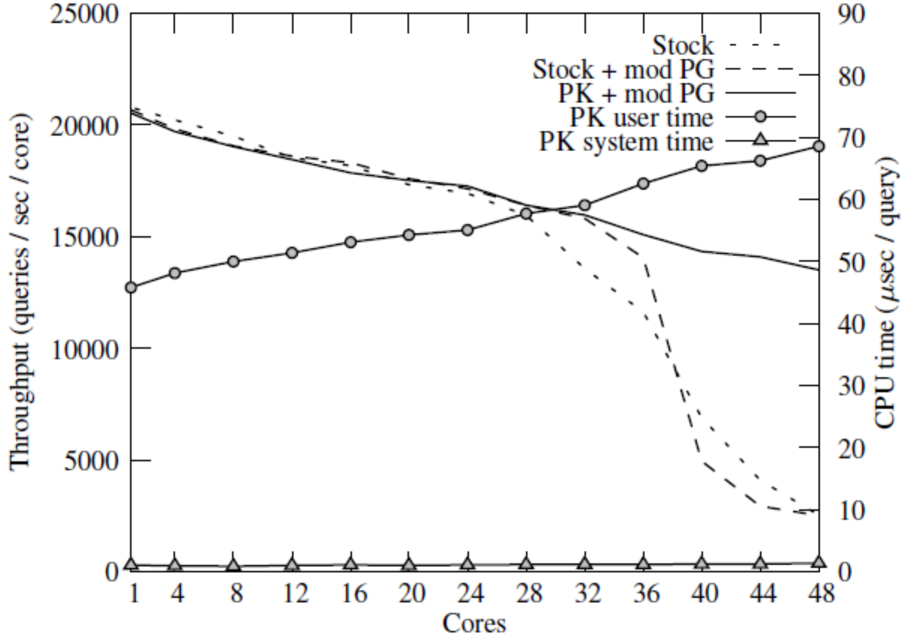


Figure 8: PostgreSQL read/write workload throughput and runtime breakdown.

PostgreSQL - cont

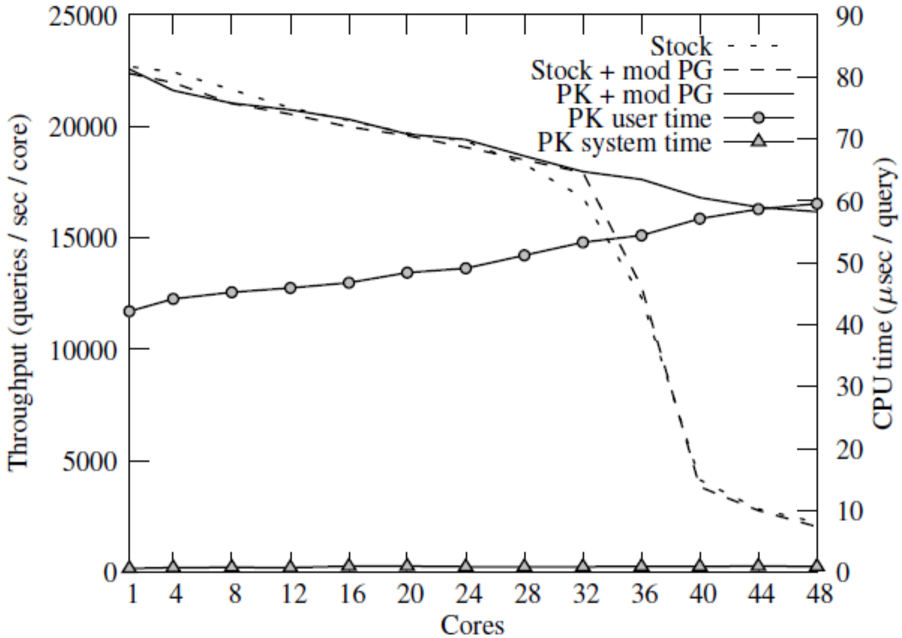


Figure 7: PostgreSQL read-only workload throughput and runtime breakdown.

gmake

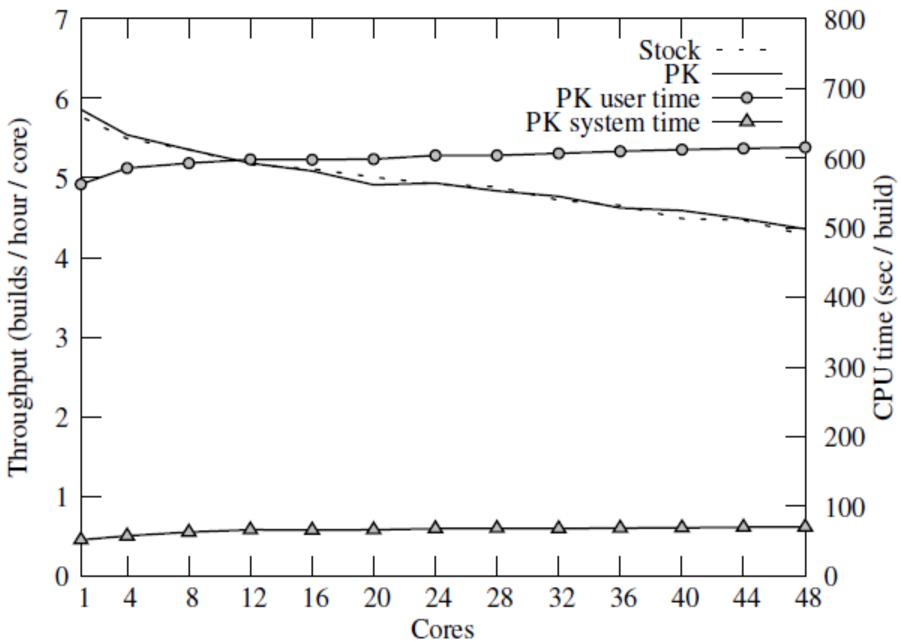


Figure 9: gmake throughput and runtime breakdown.

Psearchy/pedsort

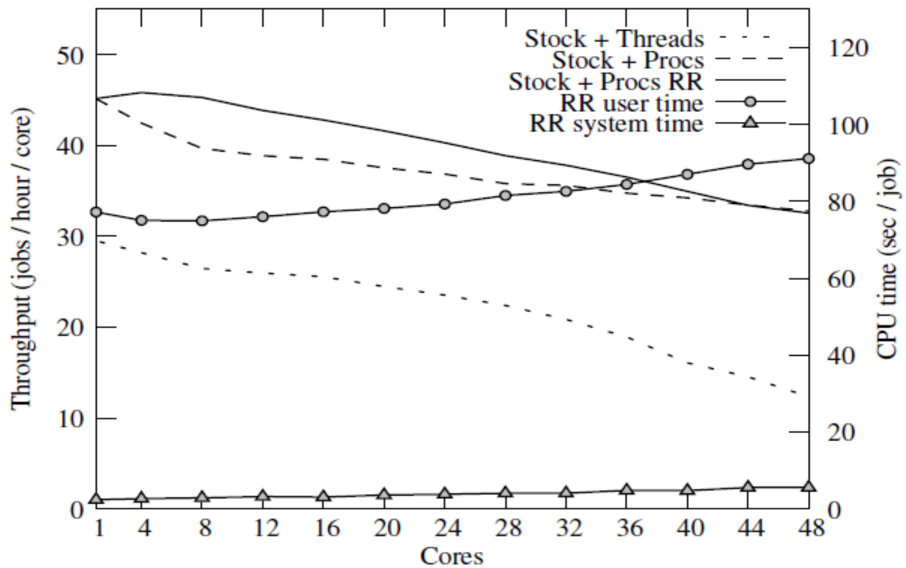


Figure 10: pedsort throughput and runtime breakdown.

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

Metis

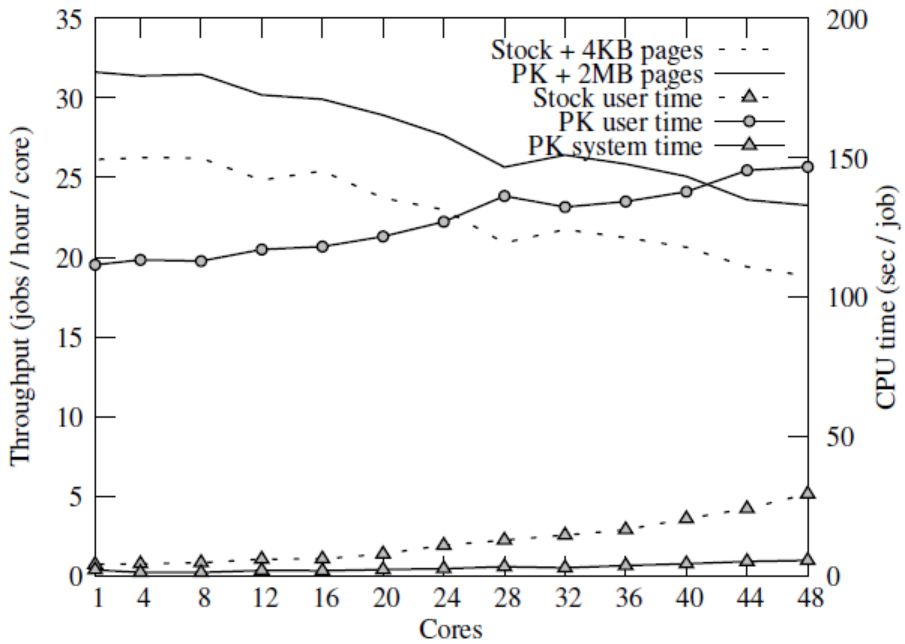


Figure 11: Metis throughput and runtime breakdown.

Summary of Linux scalability problems

Parallel `accept`			Apache
Concurrent `accept` system calls contend on shared `socket` fields.	⇒	User per-core backlog queues for listening sockets.	
`dentry` reference counting			Apache, Exim
File name resolution contends on directory entry reference counts.	⇒	Use sloppy counters to reference count directory entry objects.	
Mount point (`vfsmount`) reference counting			Apache, Exim
Walking file name paths contends on mount point reference counts.	⇒	Use sloppy counters for mount point objects.	
IP packet destination (`dst_entry`) reference counting			memcached, Apache
IP packet transmission contends on routing table entries.	⇒	Use sloppy counters for IP routing table entries.	

Summary of Linux scalability problems - cont

Protocol memory usage tracking		memcached, Apache
Cores contend on counters for tracking protocol memory consumption.	⇒	Use sloppy counters for protocol usage counting.
Acquiring directory entry (`dentry`) spin locks		Apache, Exim
Walking file name paths contends on per-directory entry spin locks.	⇒	Use a lock-free protocol in `dlookup` for checking filename matches.
Mount point table spin lock		Apache, Exim
Resolving path names to mount points contends on a global spin lock.	⇒	Use per-core mount table caches.
Adding files to the open list		Apache, Exim
Cores contend on a per-super block list that tracks open files.	⇒	Use per-core open file lists for each super block that has open files.

Summary of Linux scalability problems - cont

False sharing in `net_device` and `device`		memcached, Apache, PostgreSQL
False sharing causes contention for read-only structure fields.	⇒	Place read-only fields on their own cache lines.
False sharing in `page`		Exim
False sharing causes contention for read-mostly structure fields.	⇒	Place read-only fields on their own cache lines.
`inode` lists		memcached, Apache
Cores contend on global locks protecting lists used to track `inode`s.	⇒	Avoid acquiring the locks when not necessary.
Dcache lists		memcached, Apache
Cores contend on global locks protecting lists used to track `dentry`s.	⇒	Avoid acquiring the locks when not necessary.

Summary of Linux scalability problems - cont

Per-`inode` mutex	**PostgreSQL**
Cores contend on a per-`inode` mutex in `lseek`.	⇒ Use atomic reads to eliminate the need to acquire the mutex.
Super-page fine grained locking	**Metis**
Super-page soft page faults contend on a per-process mutex.	⇒ Protect each super-page memory mapping with its own mutex.

Summary of Bottlenecks

Application	Bottleneck
Exim	App: Contention on spool directories
memcached	HW: Transmit queues on NIC
Apache	HW: Receive queues on NIC
PostgreSQL	App: Application-level spin lock
gmake	App: Serial stages and stragglers
pedsort	HW: Cache capacity
Metis	HW: DRAM throughput

Figure 12: Summary of the current bottlenecks in MOSBENCH, attributed either to hardware (HW) or application structure (App).

Summary

- Most applications can scale well to many cores with modest modifications to the applications and to the kernel
- More bottlenecks are expected to be revealed when running on more cores

Filesystems

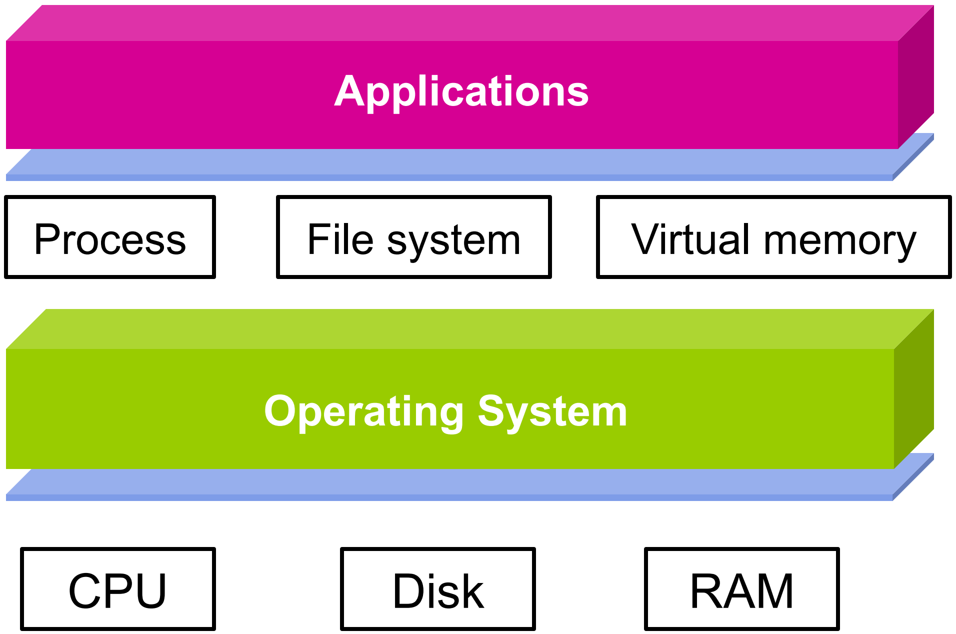
Credit: some slides by John Kubiatowicz and Anthony D. Joseph

Today and some of next class

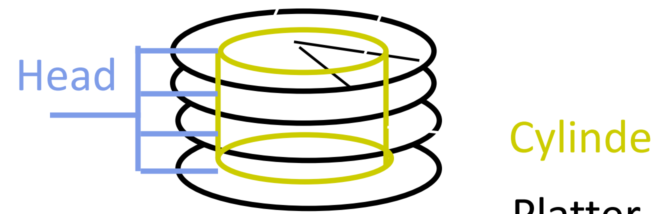
- Overview of file systems
- Papers on basic file systems
  - A Fast File System for UNIX
    Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry. Appears in *ACM Transactions on Computer Systems* (TOCS), Vol. 2, No. 3, August 1984, pp 181-197
  - Log Structured File Systems (LFS), Ousterhout and Rosenblum

- System design paper and system analysis paper

OS Abstractions



Review: Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queuing Time + Controller time + Seek Time + Rotation Time + Xfer Time

- Highest Bandwidth:
  - Transfer large group of blocks sequentially from one track

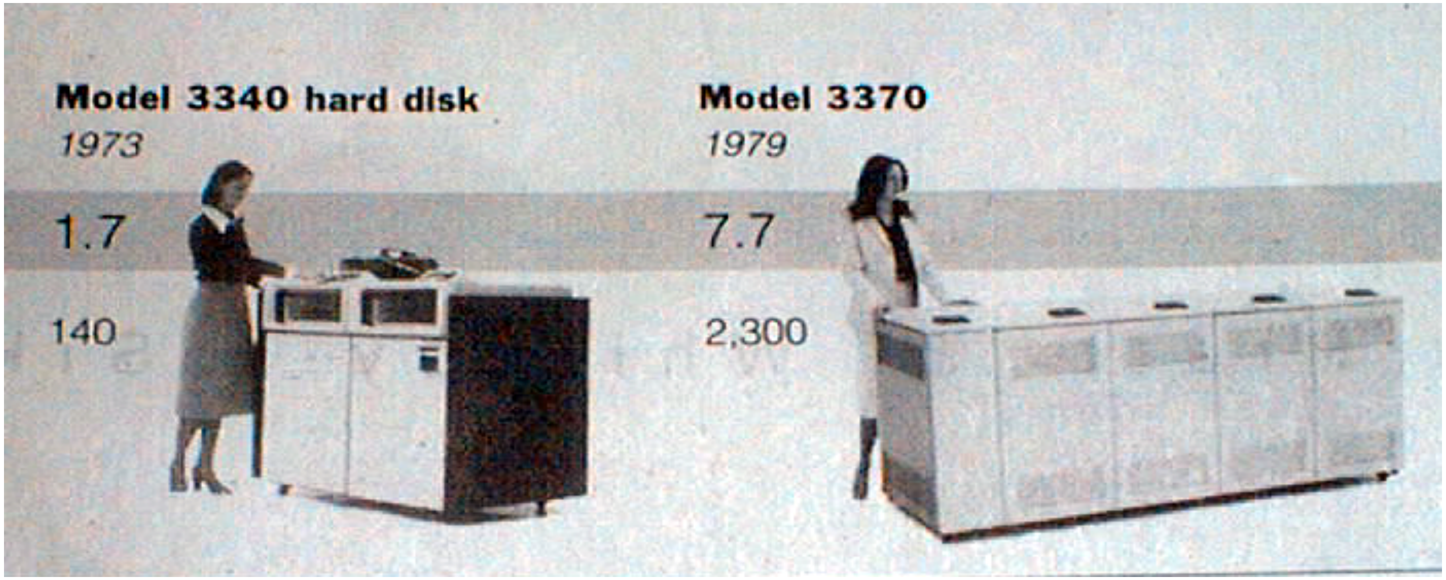
Historical Perspective

- 1956 IBM Ramac — early 1970s Winchester
  - Developed for mainframe computers, proprietary interfaces
  - Steady shrink in form factor: 27 in. to 14 in.
- Form factor and capacity drives market more than performance
- 1970s developments
  - 5.25 inch floppy disk formfactor (microcode into mainframe)
  - Emergence of industry standard disk interfaces
- Early 1980s: PCs and first generation workstations
- Mid 1980s: Client/server computing
  - Centralized storage on file server
    - accelerates disk downsizing: 8 inch to 5.25
  - Mass market disk drives become a reality
    - industry standards: SCSI, IPI, IDE
    - 5.25 inch to 3.5 inch drives for PCs, End of proprietary interfaces
- 1900s: Laptops => 2.5 inch drives
- 2000s: Shift to perpendicular recording
  - 2007: Seagate introduces 1TB drive
  - 2009: Seagate/WD introduces 2TB drive
- 2014: Seagate announces 8TB drives

Disk History

Data density Mbit/sq. in.

Capacity of Unit Shown Megabytes



1973:
1. 7 Mbit/sq. in
140 MBytes

1979:
7. 7 Mbit/sq. in
2,300 MBytes

source: New York Times, 2/23/98, page C3,
“Makers of disk drives crowd even mroe data into even smaller spaces”

Disk History

1989:
63 Mbit/sq. in
60,000 MBytes

1997:
1450 Mbit/sq. in
2300 MBytes

1997:
3090 Mbit/sq. in
8100 MBytes

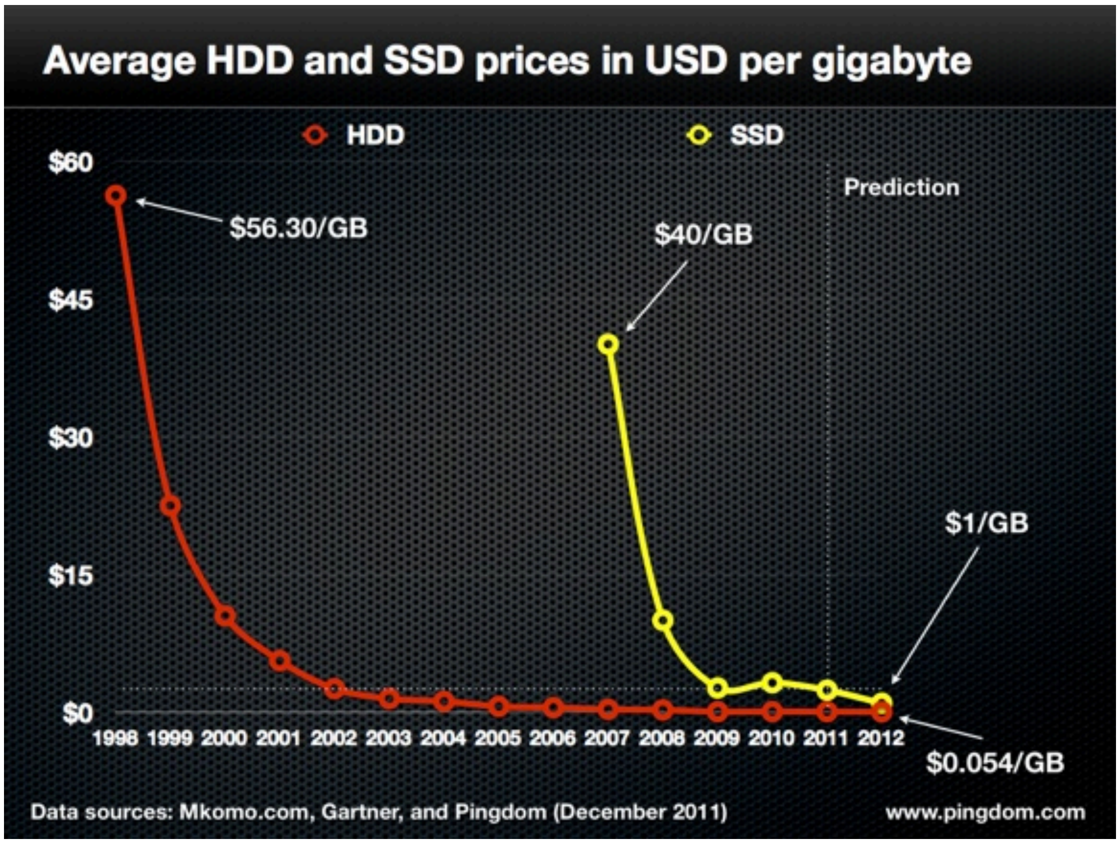
source: New York Times, 2/23/98, page C3,
“Makers of disk drives crowd even mroe data into even smaller spaces”

Recent: Seagate Enterprise (2015)

- 10TB! 800 Gbits/inch^2
- 7 (3.5”) platters, 2 heads each
- 7200 RPM, 8ms seek latency
- 249/225 MB/sec read/write transfer rates
- 2.5million hours MTBF
- 256MB cache
- $650

Contrarian View

› FFS doesn't matter in 2012!



› What about Journaling? Is it still relevant?

60 TB SSD ($20,000)

Storage Performance & Price

	Bandwidth (sequential R/W)	Cost/GB	Size
HHD	50-100 MB/s	$0.05-0.1/GB	2-4 TB
SSD[1]	200-500 MB/s (SATA) 6 GB/s (PCI)	$1.5-5/GB	200GB-1TB
DRAM	10-16 GB/s	$5-10/GB	64GB-256GB

[1] http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/

BW: SSD up to x10 than HDD, DRAM > x10 than SSD
Price: HDD x30 less than SSD, SSD x4 less than DRAM

File system abstractions

- How do users/user programs interact with the file system?
  - Files
  - Directories
  - Links
  - Protection/sharing model
- Accessed and manipulated by a virtual file system set of system calls
- File system implementation:
  - How to map these abstractions to the storage devices
  - Alternatively, how to implement those system calls

File system basics

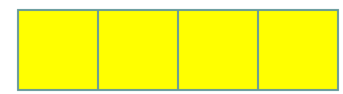
- Virtual file system abstracts away concrete file system implementation
  - Isolates applications from details of the file system

- Linux vfs interface includes:
  - creat(name)
  - open(name, how)
  - read(fd, buf, len)
  - write(fd, buf, len)
  - sync(fd)
  - seek(fd, pos)
  - close(fd)
  - unlink(name)

Disk Layout Strategies

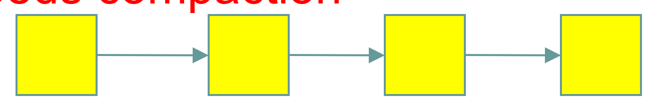
- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation



   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction

2. Linked structure

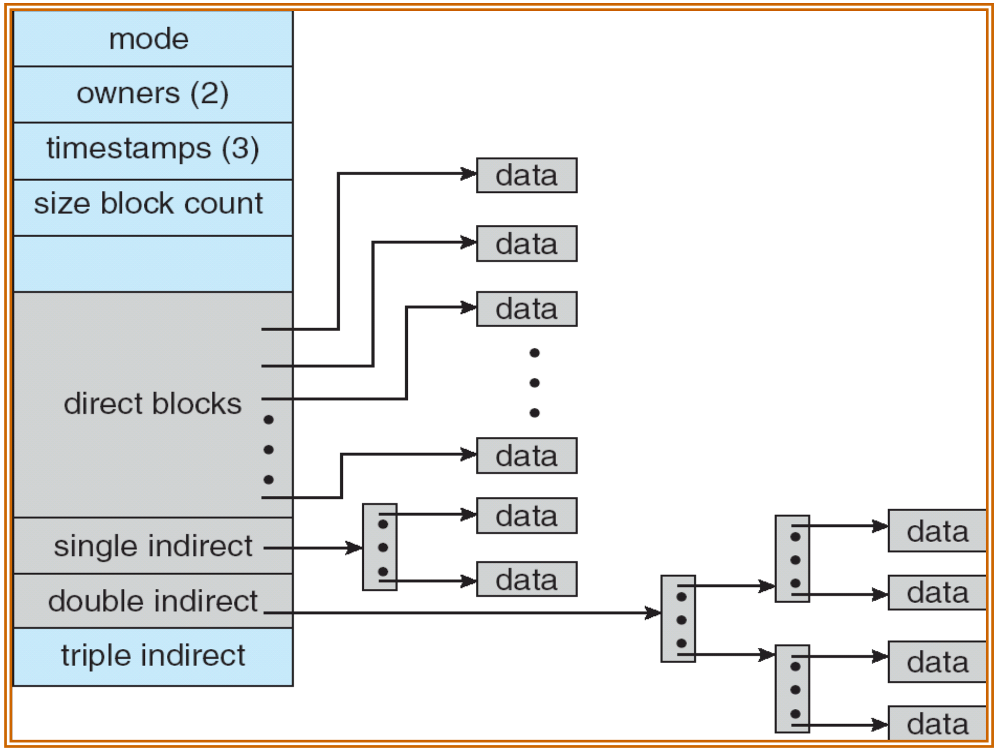


   - Each block points to the next, directory points to the first
   - Bad for random access patterns

3. Indexed structure (indirection, hierarchy)
   - An “index block” contains pointers to many other blocks
   - Handles random better, still good for sequential
   - May need multiple index blocks (linked together)

Zooming in on i-node

- i-node: structure for per-file metadata (unique per file)
  - contains: ownership, permissions, timestamps, about 10 data-block pointers
  - i-nodes form an array, indexed by "i-number" – so each i-node has a unique i-number
  - Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)



- Indirect blocks:
  - i-node only holds a small number of data block pointers (direct pointers)
  - For larger files, i-node points to an indirect block containing 1024 4-byte entries in a 4K block
  - Each indirect block entry points to a data block
  - Can have multiple levels of indirect blocks for even larger files

Unix Inodes and Path Search

- Unix Inodes are not directories
- Inodes describe where on disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open "/one", use Master Block to find inode for "/" on disk
  - Open "/", look for entry for "one"
  - This entry gives the disk block number for the inode for "one"
  - Read the inode for "one" into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
- This is why we have *open* in addition to *read* and *write*

A naïve implementation