Detecting Concurrency Bugs:
Eraser & TSVD

Prof. Phillip Gibbons

Spring 2020, Lecture 6
Today’s Reminders / Announcements

• Now have a full semester schedule
  – **Midterms**: scheduled for 2/17 and 4/15
  – **Project deadlines**: 2/10 form teams, 2/28 proposals, 4/3 interim report, 5/1 posters, 5/4 & 5/5 presentations, 5/7 final report
  – **Project discussions** with Andrew & me: 2/24, 3/30, 4/20
Today’s Papers

“Eraser: A Dynamic Data Race Detector for Multithreaded Programs”
Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson 1997

“Efficient Scalable Thread-Safety-Violation Detection”
Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, Rohan Padhye 2019
“Eraser: A Dynamic Data Race Detector for Multithreaded Programs”
Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, Thomas Anderson

- Stefan Savage (UCSD, CMU undergrad, ACM Fellow)
- Michael Burrows (Google, BWT in bzip2, FRS Fellow)
- Greg Nelson (HP, d. 2015, Herbrand Award 2013)
- Patrick Sobalvarro (Upward Labs, many start-ups)
- Tom Anderson (U. Washington, 49000+ citations, Usenix Lifetime Achievement Award 2014)
Data Race Detection

Data Race: Two concurrent threads access a shared variable and
- At least one access is a write
- The threads use no explicit mechanism to prevent the accesses from being simultaneous

- Monitors [Hoare 1974] prevent data races at compile time, but only when all shared variables are static globals
- Static Analysis must reason about program semantics
- Happens-before Analysis
  - E.g., using vector clocks

This paper: Based on locking discipline
Vector Clocks for Race Detectors

- $a \rightarrow b$ iff $V(a) < V(b)$

- Using vector clocks
  - Inter-thread arcs are from unlock $L$ to next lock $L$; otherwise, report a data race
  - Check each access for conflicting access unrelated by $\rightarrow$
Drawbacks of Happens-Before

**Difficult to implement efficiently**
- Require per-thread info about concurrent accesses to each shared-memory location

**Effectiveness highly dependent on interleaving that occurred**
- Can miss a data race
Lockset Algorithm (1\textsuperscript{st} version)

- Let \texttt{locks\_held}(t) be the set of locks held by thread t
- For each \texttt{v}, initialize \texttt{C(v)} to the set of all locks
- On each access to \texttt{v} by thread t:
  - Set \texttt{C(v)} := \texttt{C(v)} \cap \texttt{locks\_held}(t)
  - If \texttt{C(v)} is empty, then issue a warning

\begin{verbatim}
Program locks_held  C(v)
lock(mu1);      {mu1,mu2}   {mu1, mu2}
v := v+1;
lock(mu1);      {mu1}       {mu1}
unlock(mu1);
lock(mu2);      {}          {}  \textcolor{red}{!!}
v := v+1;
unlock(mu2);
\end{verbatim}
Handling Initialization & Read Sharing

• State machine tracked for each variable $v$

• Empty lockset $C(v)$ reported only if $v$ is Shared-Modified
Lockset Algorithm in Shared-Modified State

Let $\text{locks}_\text{held}(t)$ be the set of locks held by thread $t$; Let $\text{write}_{-}\text{locks}_\text{held}(t)$ be set of locks held in write mode by $t$

When enter Shared-Modified state:
For each $v$, initialize $C(v)$ to the set of all locks

On each read of $v$ by thread $t$:
- Set $C(v) := C(v) \cap \text{locks}_\text{held}(t)$
- If $C(v)$ is empty, then issue a warning

On each write of $v$ by thread $t$:
- Set $C(v) := C(v) \cap \text{write}_{-}\text{locks}_\text{held}(t)$
- If $C(v)$ is empty, then issue a warning

Correct: Locks held purely in read mode do not protect against a data race between the writer & some other reader thread
Discussion: Summary Question #1

- **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Implementation

• Binary instrumentation

• Instruments lock/unlock calls, thread init/finalize to maintain lock_held(t)

• Instruments each load/store, malloc to maintain \( C(v) \)
  – 32-bit (aligned) words
  – But not stack-based accesses (stack is assumed private)
  – 32-bits in “shadow memory” for each word
    (holds 2-bit state + thread ID or “lockset index”)

• Warnings report file, line number, active stack frames, thread ID, memory access address & type, PC, SP
  – Option: Log all accesses to \( v \) that modify \( C(v) \)
False Alarms & Annotations

• Memory reused without resetting shadow memory
  – When app uses private memory allocator
  – Annotation: EraserReuse(address, size) – reset to Virgin

• Synchronization outside of instrumented channels
  – E.g., Private lock implementations of MultiRd/SingleWr locks
  – E.g., Spin on flag
  – Annotation: EraserReadLock(lock), EraserReadUnlock(lock),
    EraserWriteLock(lock), EraserWriteUnlock(lock)

• Benign races
  – Annotation: EraserIgnoreOn(), EraserIgnoreOff()

“We found that a handful of these annotations usually suffices to eliminate all false alarms.”
Experience

• Ten iterations of races/false alarms to resolve all reported races

• Worked well on servers
  – Experienced programmers obey the simple locking discipline

• AltaVista Web indexing service: mhttpd & Ni2
  – Some good examples of benign races in production codes
  – 24 annotations reduced false positives from 100+ to 0
  – Test: Reintroduced 2 old bugs & found/corrected in 30 minutes

• Vesta Cache Server
  – Found data race on “valid” bit—serious on weak memory model
  – Benign: Main thread passes RPC request to worker thread;
    Head-of-log lock makes entire log private
  – 10 annotations & 1 bug fix reduced alarms from 100s to 0
Experience

• Petal distributed storage system
  – Implements distributed consensus, failure detector/recovery
  – False alarms: private RW-lock implementation; statistics; stack frame reuse (could not annotate away)
  – Found one real race

• Undergraduate coursework
  – 100 runnable assignments
  – Found data races in 10% of them

• Sensitivity to thread interleavings
  – Reran Ni2 & Vesta on 2 threads instead of 10
  – Same race reports, in different order
• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Performance

“Performance was not a major goal in our implementation”

- Typical app slowdown: 10x-30x
  - Estimate half due to procedure call at every load/store
  - Today: dynamic binary instrumentation (DBI) using inlining for short code segments

“Eraser is fast enough to debug most programs and therefore meets the most essential performance criterion.”
Aside: Performance Comparison from FastTrack paper [Most Influential Paper from PLDI’09]

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (loc)</th>
<th>Thread Count</th>
<th>Base Time (sec)</th>
<th>Instrumented Time (slowdown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colt</td>
<td>111,421</td>
<td>11</td>
<td>16.1</td>
<td>0.9</td>
</tr>
<tr>
<td>Crypt</td>
<td>1,241</td>
<td>7</td>
<td>0.2</td>
<td>7.6</td>
</tr>
<tr>
<td>Lufact</td>
<td>1,627</td>
<td>4</td>
<td>4.5</td>
<td>2.6</td>
</tr>
<tr>
<td>Moldyn</td>
<td>1,402</td>
<td>4</td>
<td>8.5</td>
<td>5.6</td>
</tr>
<tr>
<td>Montecarlo</td>
<td>3,669</td>
<td>4</td>
<td>5.0</td>
<td>4.2</td>
</tr>
<tr>
<td>Mtrt</td>
<td>11,317</td>
<td>5</td>
<td>0.5</td>
<td>5.7</td>
</tr>
<tr>
<td>Raja</td>
<td>12,028</td>
<td>2</td>
<td>0.7</td>
<td>2.8</td>
</tr>
<tr>
<td>Raytracer</td>
<td>1,970</td>
<td>4</td>
<td>6.8</td>
<td>4.6</td>
</tr>
<tr>
<td>Sparse</td>
<td>868</td>
<td>4</td>
<td>8.5</td>
<td>5.4</td>
</tr>
<tr>
<td>Series</td>
<td>967</td>
<td>4</td>
<td>175.1</td>
<td>1.0</td>
</tr>
<tr>
<td>Sor</td>
<td>1,005</td>
<td>4</td>
<td>0.2</td>
<td>4.4</td>
</tr>
<tr>
<td>Tsp</td>
<td>706</td>
<td>5</td>
<td>0.4</td>
<td>4.4</td>
</tr>
<tr>
<td>Elevator*</td>
<td>1,447</td>
<td>5</td>
<td>5.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Philo*</td>
<td>86</td>
<td>6</td>
<td>7.4</td>
<td>1.1</td>
</tr>
<tr>
<td>Hede*</td>
<td>24,937</td>
<td>6</td>
<td>5.9</td>
<td>1.1</td>
</tr>
<tr>
<td>Jbb*</td>
<td>30,491</td>
<td>5</td>
<td>72.9</td>
<td>1.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ERASER</th>
<th>MULTIRACE</th>
<th>GOLDILOCKS</th>
<th>GOLDILOCKS</th>
<th>BASICVC</th>
<th>DIIT+</th>
<th>FASTTRACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colt</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Crypt</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lufact</td>
<td>4</td>
<td>0</td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Moldyn</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Montecarlo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mtrt</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Raja</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Raytracer</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Sparse</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Series</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sor</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tsp</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Elevator*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Philo*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hede*</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Jbb*</td>
<td>3</td>
<td>1</td>
<td>–</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Benchmark Results. Programs marked with ‘*’ are not compute-bound and are excluded when computing average slowdowns.
Protection by Multiple Locks

- Every writer must hold all locks
  Every reader must hold at least 1 lock
  - Used to avoid deadlock in program that contains upcalls

- Causes false alarms
  - Not worth cost of handling this
Deadlock

“If the data race is Scylla, the deadlock is Charybdis.”

(Sea monsters in Homer’s Odyssey)

• Discipline: Acquire locks in ascending order

• Found cycle of locks in formsedit application

• Would be useful addition to Eraser…
  – Aside: In today’s tools such as Valgrind
Race Detection in OS Kernel

- **OS often raises the processor interrupt level to provide mutual exclusion**
  - Particular interrupt level inclusively protects all data protected by lower interrupt levels
  - Solution: Have a virtual lock for each level; when raise level to \( n \), treat this as first \( n \) per-level locks acquired

- **OS makes greater use of POST/WAIT style synch, e.g., semaphores to signal when a device op is done**
  - Problem: Hard to infer which data a semaphore is protecting
Aside: Race Detection in Kernels

**DataCollider [OSDI’10]**
- Randomly delays a kernel thread to see if racy access occurs while stalled (but can’t use for time-critical interrupts)
- “Active Delay Injection”

**Guardrail [ASPLOS’14] for kernel-mode drivers addresses these challenges:**
- Single thread can race itself (!)
- Synchronization invariants based on context of device state
- Synchronization based on deferred execution using softirqs or timers
- Mutual exclusion via HW test-and-set or disabling interrupts & preemption
Representing $C(v)$s

- Represent by small integer “lockset index” into table
  - Never observed $> 10K$ distinct lock sets

- Append-only table

- Lock vectors sorted

- Cache results of set intersections

- Shadow word: 30-bit index, 2-bit state

Issue: Shadow memory doubles size of memory!
(Aside: Can fix with 2-level shadow memory)
Aside: Data Race Detection in 2000s

- **Valgrind tools: Helgrind, DRD, ThreadSanitizer**
  - Use Happens-before
  - Only ThreadSanitizer also uses Lockset
  - Early versions of Helgrind used Lockset

- **Intel ThreadChecker**
  - Uses Happens-before

- **Cilk: Nondeterminator, Cilkscreen**
  - Relies on fork-join structure of Cilk programs to determine whether two conflicting accesses are ordered
  - Reports race or that no race can occur with the given input

Hundreds of papers & prototype systems
“Efficient Scalable Thread-Safety-Violation Detection”

Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, Rohan Padhye 2019

- Guangpu Li (U. Chicago student, MSR intern)
- Shan Lu (U. Chicago, Li’s advisor, SigOps Chair, OSDI’20 PC Chair)
- Madan Musuvathi (MSR, DataCollider author)
- Suman Nath (MSR, CMU PhD 2005)
  - Who is Suman’s most frequent co-author?
- Rohan Padhye (UC Berkeley student, MSR intern)
Efficient and Scalable Thread-Safety-Violation Detection

Finding thousands of concurrency bugs during testing

Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, Rohan Padhye

Slides are from Li’s presentation at SOSP’19, with a few modifications
Thread-safety violation (TSV)

class MyClass{
    public:
        m1();
        m2();
        ...
}

Thread-safety contract

m1 \not\rightarrow m1
m1 \not\rightarrow m2

Thread-safety violation

Thread 1
myObj.m1()

Thread 2
myObj.m2()
Thread-safety violation example

```cpp
class MyClass {
    public:
        MyClass();
        MyClass(MyClass &other);
        MyClass(const MyClass &other);
    private:
        MyClass(MyClass &&other);
        MyClass(MyClass &&other) = default;
};

MyClass myObj;
```

A Major Bug In Bitcoin Software Could Have Crashed the Currency

Thread-safety contract
Reasoning about TSVs is difficult (I)

too many possibilities of concurrency

Multithreading

Event-driven

Message passing
Reasoning about TSVs is difficult (II)

Too many forms of *synchronizations*

lock.Lock()
semaphore.Require()
Obj.signal()
synchronized{...}

RPC_wait()
Obj.wait()  pthread_mutex_lock()
Obj.notify()
While(flag != 1){}

Thread.join() Event.WaitFor()
monitor.exit() monitor.tryenter()
RPC_wait()
In small scale, it is fine.

Program source code
Integration → Static data-race analysis → Pruning
~1 out of 5

Test Inputs
Program binary
Integration → Dynamic data-race analysis → Pruning
~10x[1] → ~1 out of 2

In large scale, it is NOT fine.

CloudBuild: 100M tests from 4K teams, up to 10K machines /day
Three challenges

Integration  Overhead  False positives
TSVD

A scalable dynamic analysis tool for TSVs:

Test inputs

Program binary

Thread-safety contracts

Integration

Dynamic analysis

Testing run

TSVD

Pruning

*1134 >> others

Push-button

30% overhead

0 false positive
TSVD

• Motivation
• TSVD design
• Evaluation
• Conclusion
How to achieve zero false positive?

Report after violation

+ inject delays to trigger violations

thread 1

thread 2

mylist.Add(x)

mylist.Add(x)

mylist.Add(x)

mylist.Add(y)
What are the potential unsafe calls?

- Every call: [DataCollider, ...]
- Every racing call: [RaceFuzzer, ...]

Cost of identification vs. Cost of delay

Happen-Before analysis:
- lock.lock()
- Obj.wait()
- Obj.signal()
- monitor.exit()
What are the potential unsafe calls?
likely Racing Calls

- Two conflict methods
- Called from different threads
- Accessing the same object
- Having concurrent logical timestamps close-by physical
Identify likely race calls

likely racing list

<table>
<thead>
<tr>
<th>Operation1</th>
<th>Operation2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mylist.add(x)</td>
<td>mylist.add(y)</td>
</tr>
</tbody>
</table>

thread 1

thread 2
Remove **unlikely** race calls

likely racing list

<table>
<thead>
<tr>
<th>Operation1</th>
<th>Operation2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mylist.add(x)</td>
<td>mylist.add(y)</td>
</tr>
</tbody>
</table>

**Loop**

thread 1

thread 2
Insights for synchronization analysis inference

Many ways to implement synchronization:

- lock.Lock()
- Obj.wait()
- RPC_wait()
- Thread.join()
- While(flag != 1){}

One common effect of all synchronizations:

*If m1 synchronized before m2 and m1—m2 are nearby*

\[ \text{delay to m1 will cause delay to m2} \]
Synchronization inference: transitive delay

```
[Operation1] mylist.add(x)
[Operation2] mylist.add(y)
```

```
[Operation1] mylist.Add(x)
[Operation2] mylist.Add(y)
```

```
thread 1
Unlock()
```

```
thread 2
Unlock()
Lock()
mylist.Add(y)
```

```
likely racing list

<table>
<thead>
<tr>
<th>Operation1</th>
<th>Operation2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mylist.add(x)</td>
<td>mylist.add(y)</td>
</tr>
</tbody>
</table>
```

Caused by any synchronization

“Happens-before interference” technique
Each entry has a prob of inserting a delay. Decays if TSV not found.
Discussion: Summary Question #1

• **State the 3 most important things the paper says.** These could be some combination of their motivations, observations, interesting parts of the design, or clever parts of their implementation.
Overall Results

Thread safety contract:

14 system classes (e.g., List, Dictionary)

Test in Microsoft:

1.6K projects, run 1 or 2 times, 1134 TSVs

Validation of 80 bugs by 4 product teams:

96% unknown, 47% causing severe customer facing issues
Comparison with other techniques

1K software components, 3K+ unit tests

- **#TSVs found in 1 run**: (higher is better)
  - Random (5%): 6
  - DataCollider: 22
  - HB-Tracking: 25
  - TSVD: 42

- **Time overhead (%)**: (lower is better)
  - Random (5%): 178
  - DataCollider: 378
  - HB-Tracking: 310
  - TSVD: 33
Comparison with other techniques

All tools combined

TSVD

HB-analysis

DataCollider

Random delay injection

# TSVs found vs. Runs
Conclusion

TSVD is a push-button TSV detection tool.

TSVD *infers* synchronizations and uses them in the same run.
- Easy integration: oblivious to synchronization patterns
- Lightweight: 30% overhead
- Accuracy: 0 false positives
- Coverage: better than HB-based detection tools

https://github.com/microsoft/TSVD
• **Describe the paper's single most glaring deficiency.** Every paper has some fault. Perhaps an experiment was poorly designed or the main idea had a narrow scope or applicability.
Limitations

• Finds TSVs but not other data races or timing bugs
  – Good news: Need not monitor every shared-memory access

• Assumes for each data structure:
  – Methods can be grouped into a read set and a write set
  – Two concurrent methods are TSVs iff at least one in write set

• Its parallel delay injection can muddy the waters

• Two TSV stack-trace pairs may correspond to the same bug

• Implemented only for in-memory data structures and .NET applications (e.g., C# and F#)
TSVD Parameter Sensitivity Analysis

Figure 9. Sensitivity analysis of various parameters of TSVD.
## Technique Sensitivity & Open Source Results

<table>
<thead>
<tr>
<th></th>
<th># bug</th>
<th>Total</th>
<th>Run1</th>
<th>Run2</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSVD</td>
<td>53</td>
<td>42</td>
<td>11</td>
<td></td>
<td>33%</td>
</tr>
<tr>
<td>No HB-inference</td>
<td>45</td>
<td>36</td>
<td>9</td>
<td></td>
<td>84%</td>
</tr>
<tr>
<td>No windowing in near-miss</td>
<td>46</td>
<td>35</td>
<td>11</td>
<td></td>
<td>143%</td>
</tr>
<tr>
<td>No concurrent phase detection</td>
<td>54</td>
<td>42</td>
<td>12</td>
<td></td>
<td>61%</td>
</tr>
</tbody>
</table>

Table 3. Removing one technique at a time from TSVD

<table>
<thead>
<tr>
<th>Project</th>
<th>LoC</th>
<th># tests</th>
<th># run</th>
<th># TSV</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationInsights [3]</td>
<td>67.5K</td>
<td>934</td>
<td>2</td>
<td>1</td>
<td>15.31%</td>
</tr>
<tr>
<td>DateTimeExtention [12]</td>
<td>3.2K</td>
<td>169</td>
<td>1</td>
<td>3</td>
<td>18.51%</td>
</tr>
<tr>
<td>FluentAssertion [20]</td>
<td>78.3K</td>
<td>3076</td>
<td>1</td>
<td>2</td>
<td>8.89%</td>
</tr>
<tr>
<td>K8s-client [33]</td>
<td>332.3K</td>
<td>76</td>
<td>2</td>
<td>1</td>
<td>11.79%</td>
</tr>
<tr>
<td>Radical [50]</td>
<td>96.9K</td>
<td>965</td>
<td>1</td>
<td>3</td>
<td>1552.13%</td>
</tr>
<tr>
<td>Sequolocity [59]</td>
<td>6.6K</td>
<td>209</td>
<td>1</td>
<td>3</td>
<td>2.97%</td>
</tr>
<tr>
<td>Stastd [62]</td>
<td>2.5K</td>
<td>34</td>
<td>2</td>
<td>1</td>
<td>9.72%</td>
</tr>
<tr>
<td>System.Linq.Dynamic [63]</td>
<td>1.2K</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>41.39%</td>
</tr>
<tr>
<td>Thunderstruck [64]</td>
<td>1.1K</td>
<td>52</td>
<td>1</td>
<td>2</td>
<td>3.33%</td>
</tr>
</tbody>
</table>

Table 4. TSVD results on open source projects.
Discussion: Summary Question #3

For Both Papers

• Describe what conclusion you draw from the paper as to how to build systems in the future. Most of the assigned papers are significant to the systems community and have had some lasting impact on the area.
Wednesday’s Paper

Starting a new topic:
File Systems and Disks

“A Fast File System for UNIX”
Marshall K. McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry 1984