Linux Plumbers Conference Student Day

Concurrency and Race Conditions

Paul E. McKenney, Ph.D.
IBM Distinguished Engineer & CTO Linux
Linux Technology Center

September 16, 2008

How I Got This Way

- **Born in Pt. Townsend, WA; raised in Silverton, OR**
  - I had no idea that one of my classmates was astronaut-to-be!
- **BS in CS & in ME at Oregon State University, 1981**
  - Worked my way through with a job at the computer center
- **Self-employed contract programmer in early 80s**
  - Building control systems, card-access security systems, acoustic navigation systems. 64K address spaces; 640K seemed unimaginably huge to me at the time!
- **SRI International in late 80s**
  - UNIX systems administration, packet-radio research, Internet protocol research
- **Sequent Computer Systems through 90s**
  - Parallel UNIX kernel: memory allocation, RCU, ...
- **IBM since 2000**
  - A bit of AIX, then Linux: recovering proprietary programmer

Avoiding Concurrency Problems: Avoid Concurrency

- **Do you need more than one CPU's worth of performance**
  - If not, be happy with sequential programming (for non-Java languages)
- **Have you done straightforward algorithmic optimizations?**
  - If not, look into them
- **Can you run multiple independent instances of your app?**
  - Do not be afraid to exploit cheap-shot concurrency!
- **Can you use parallel infrastructure (DBMS, &c)?**
  - If so, leave the concurrency problems to the parallel infrastructure!
- **Otherwise, you need to look concurrency in the face**
- **Of course, if you are doing a hobby project, feel free to go wild with concurrency and much else besides!**

Why Concurrency?

- **Higher performance**
- **Acceptable productivity**
- **Reasonable generality**

- **Or because it is fun!!!**
  - (Though your manager/professor/spouse/whatever might have a different opinion on this point...)

- **Reliability goes without saying, aside from this self-referential bullet point**
  - If it doesn't have to be reliable: “return 0;” is simple and fast

Concurrency Likely to be Somewhat Intuitive...

Concurrency Likely to be Somewhat Intuitive...

TAP
Look, Urg. I can only do one thing at a time. Just wait until I'm done.
Is he done?
Yep! He's done.
POKE

Concurrency Likely to be Somewhat Intuitive...

TAP
Look, Ury. I can only do one thing at a time. Just wait until I'm done.
Is he done?
Yep! He's done.
POKE

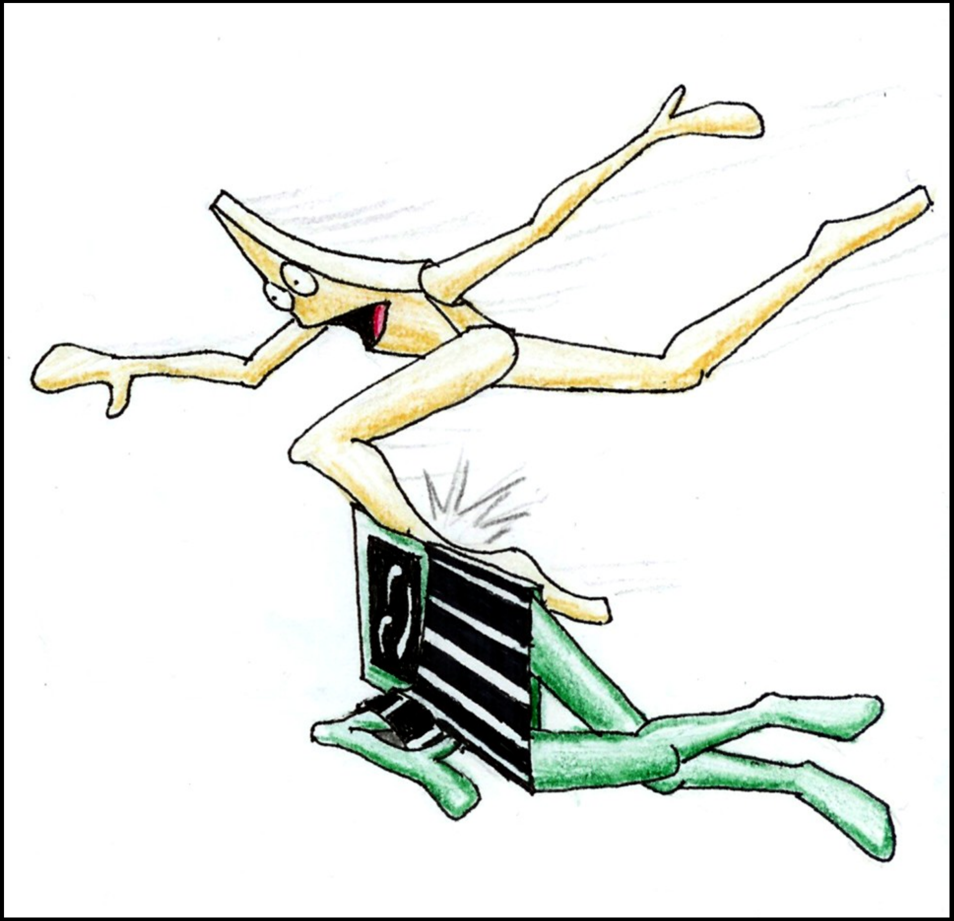
But this in no way implies that concurrent *programming* is intuitive...
Time unfolds sequentially, but sequential programming not universal!

Concurrency Problem #1: Poor Performance

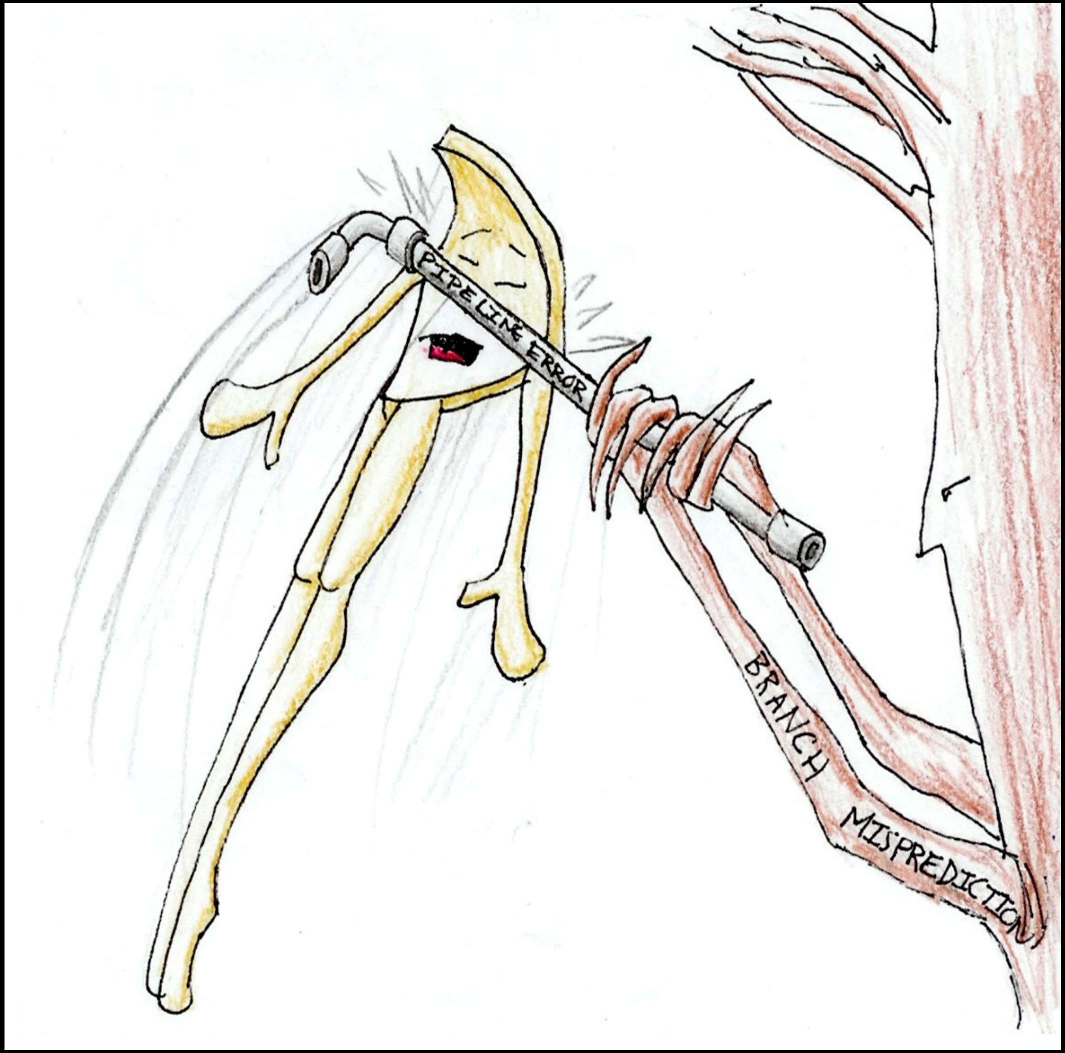
- **This is a severe problem in cases where performance was the only reason to exploit concurrency...**
- **Lots of effort, little (or no) result**
- **Why???**

CPU Performance: The Marketing Pitch

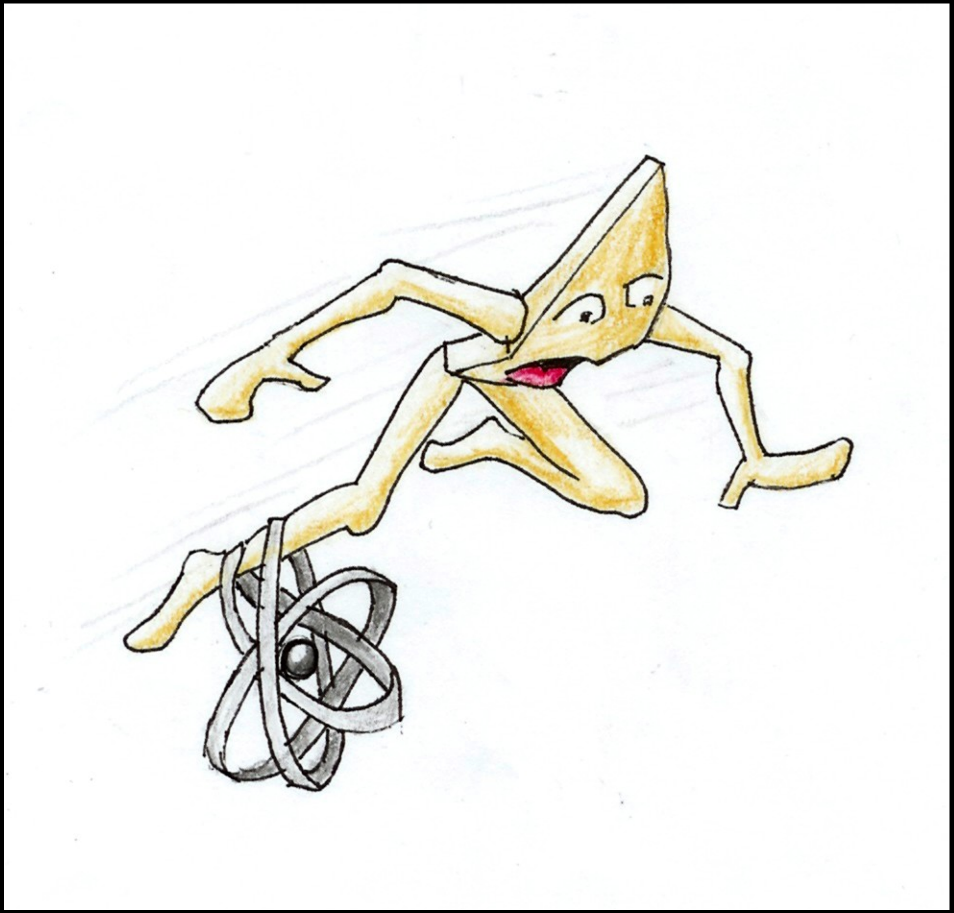
CPU Performance: Memory References



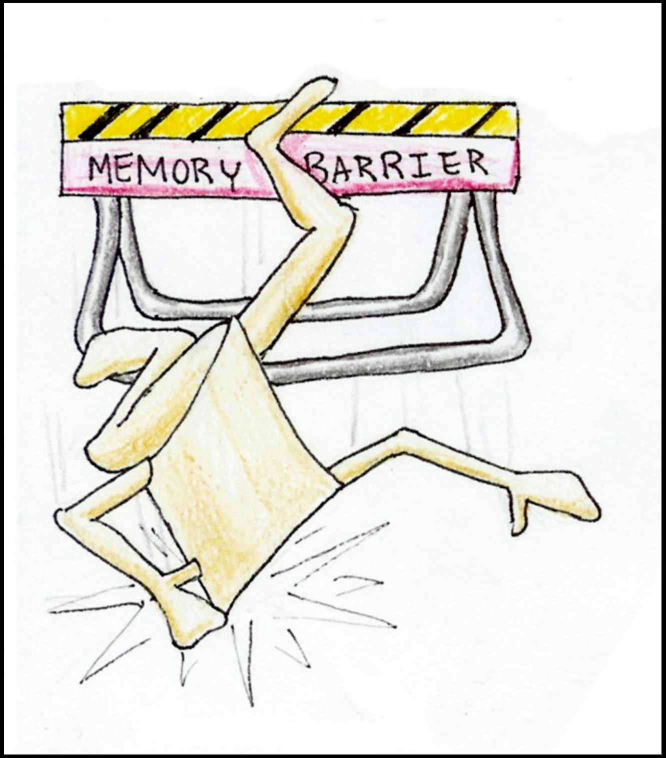
CPU Performance: Pipeline Flushes



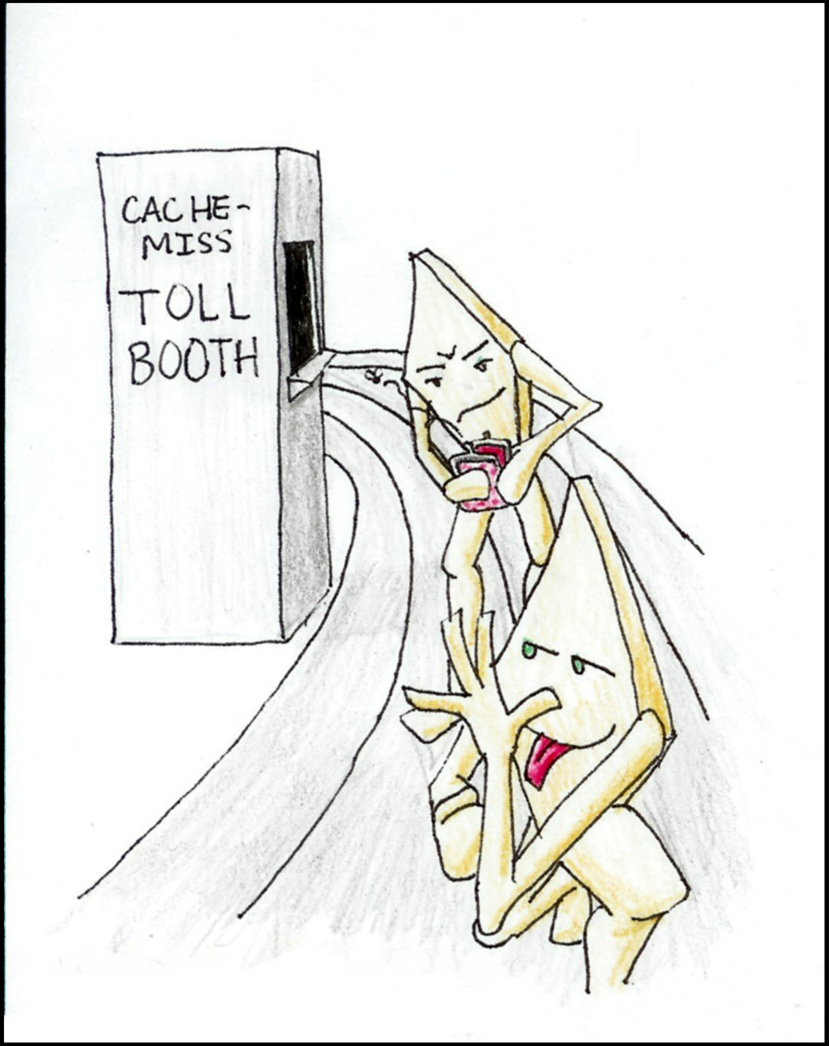
CPU Performance: Atomic Instructions



CPU Performance: Memory Barriers



CPU Performance: Cache Misses



CPU Performance: Intel 2GHz Core Duo

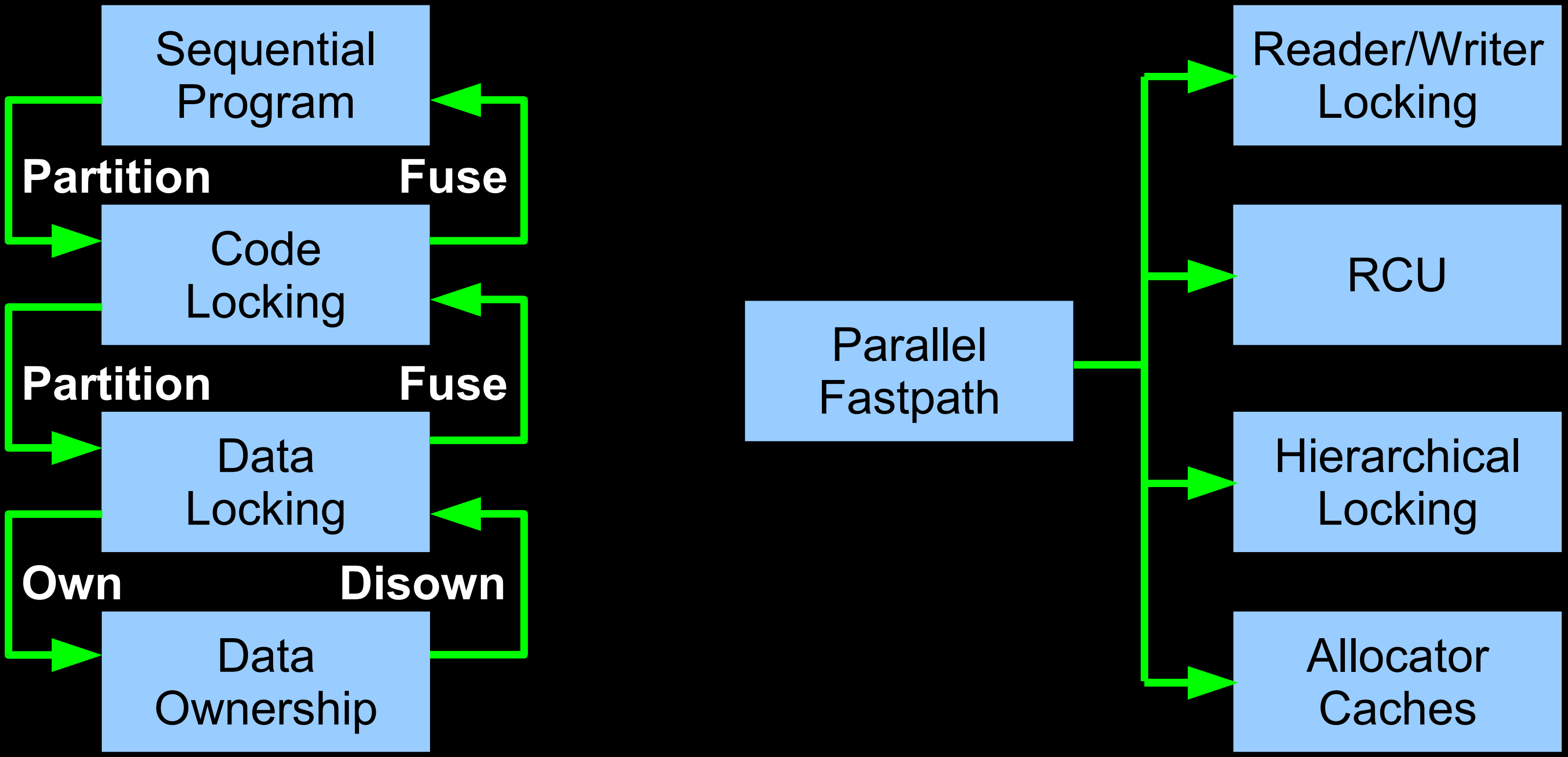
Operation	Min Cost (ns)
Normal Increment (++)	3.25
Atomic Increment	20.91
Compare-And-Exchange Increment	34.70
Memory Barrier	25.23
Lock Round Trip	59.04
CPU-to-CPU Cache Miss	130.56

Larger machines usually incur larger penalties...
(1) Favor low-cost operations
(2) Use coarse-grained parallelism: embarrassingly parallel is good!

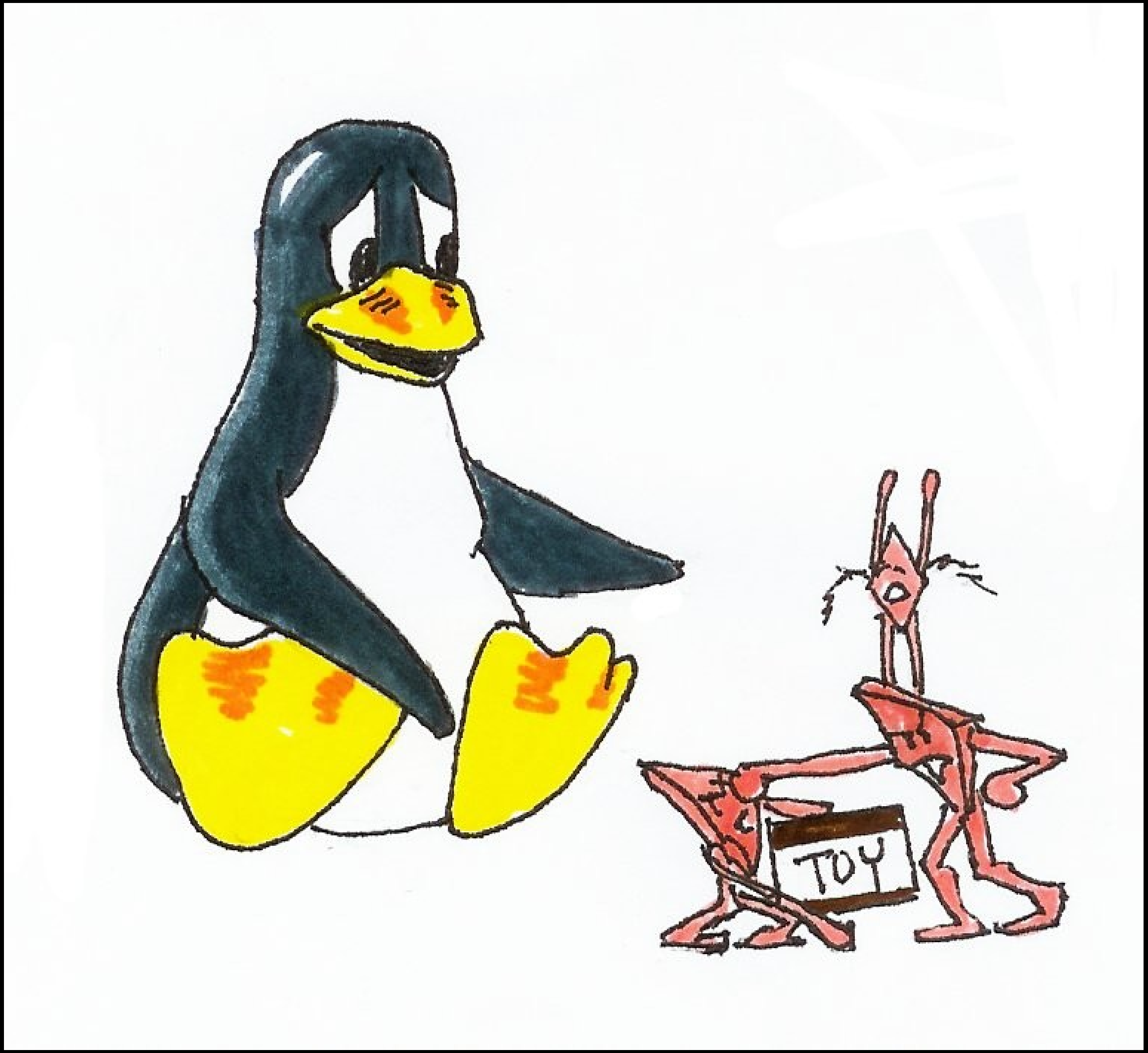
Other Performance Obstacles

- TLB misses
- Context switches
- Interrupts and NMIs
- Page faults
- Thread creation/destruction
- Disk I/O
- Network latencies

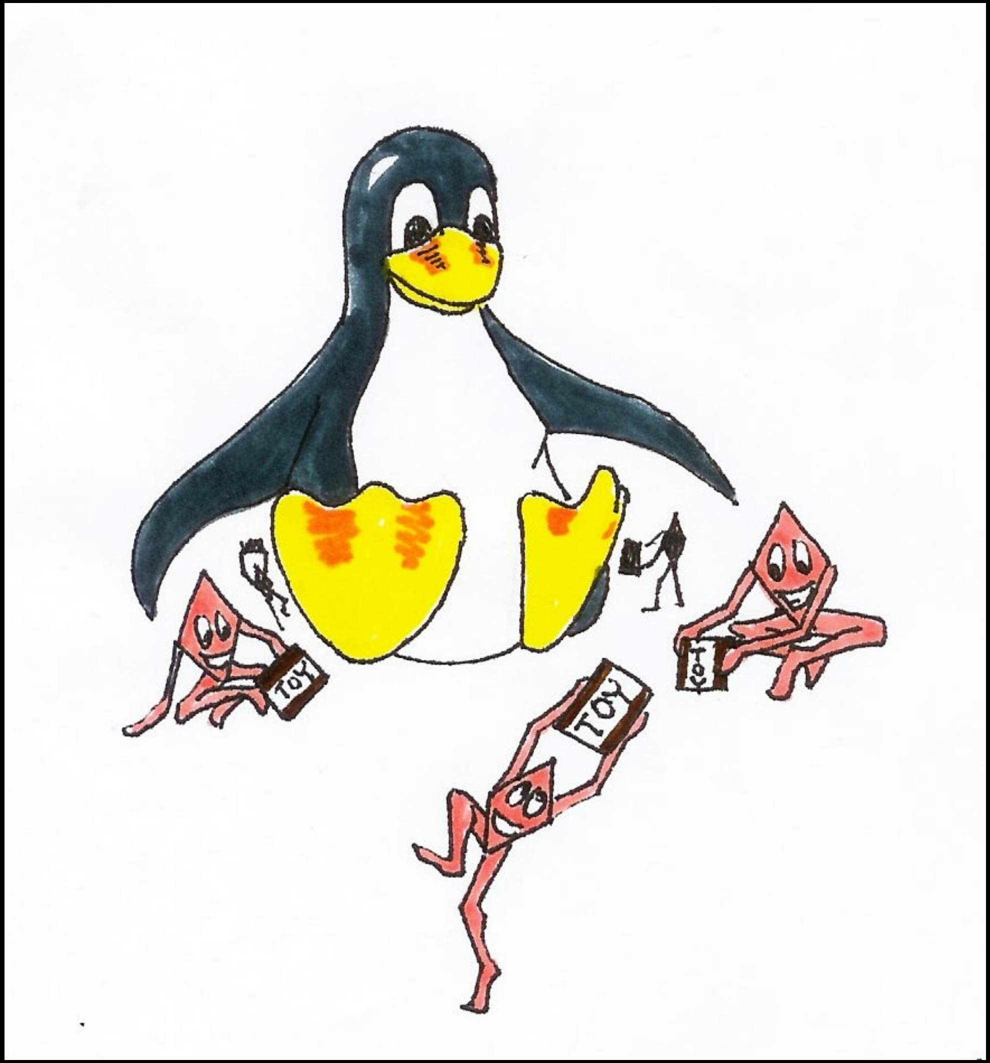
Parallel Design Patterns



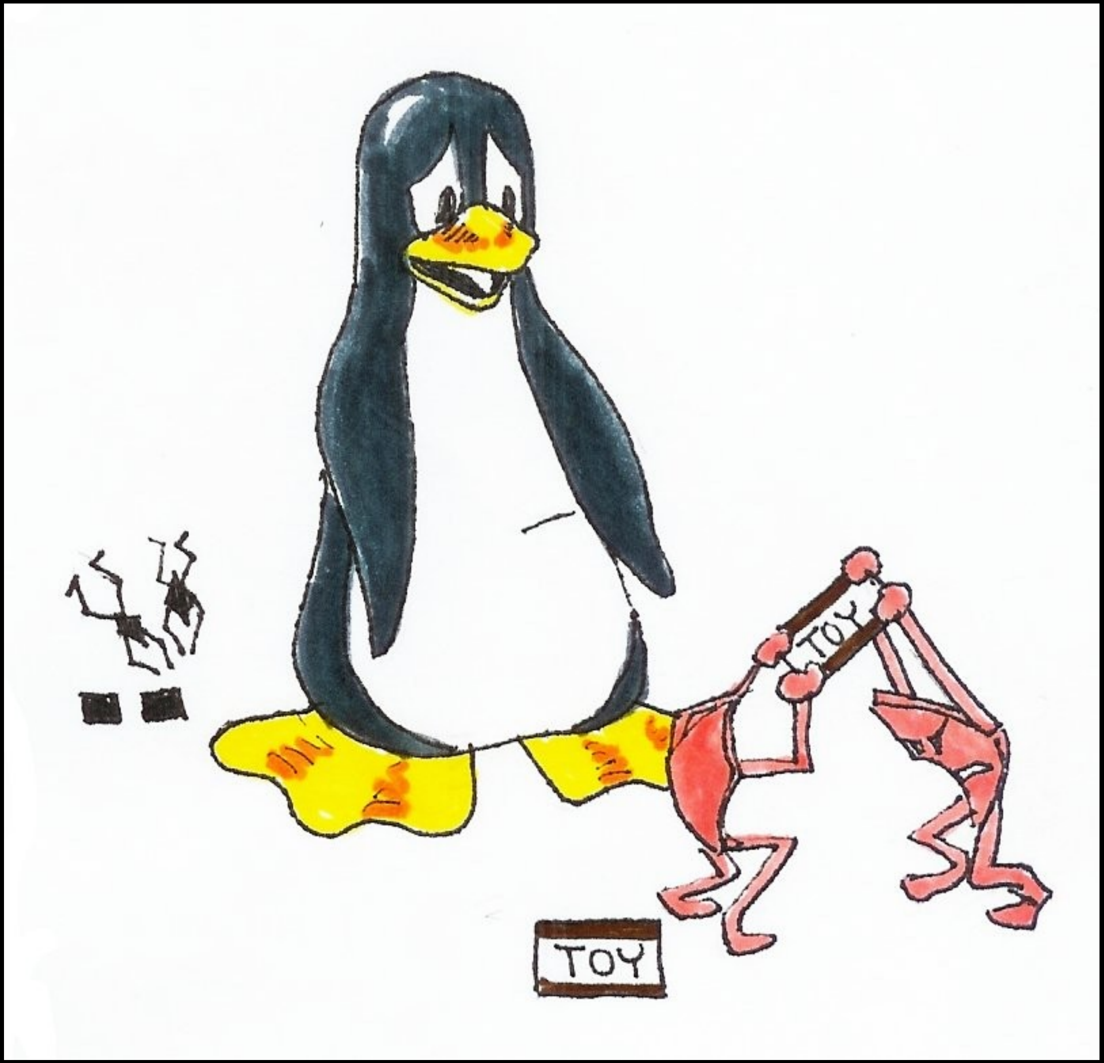
Problem With Code Locking



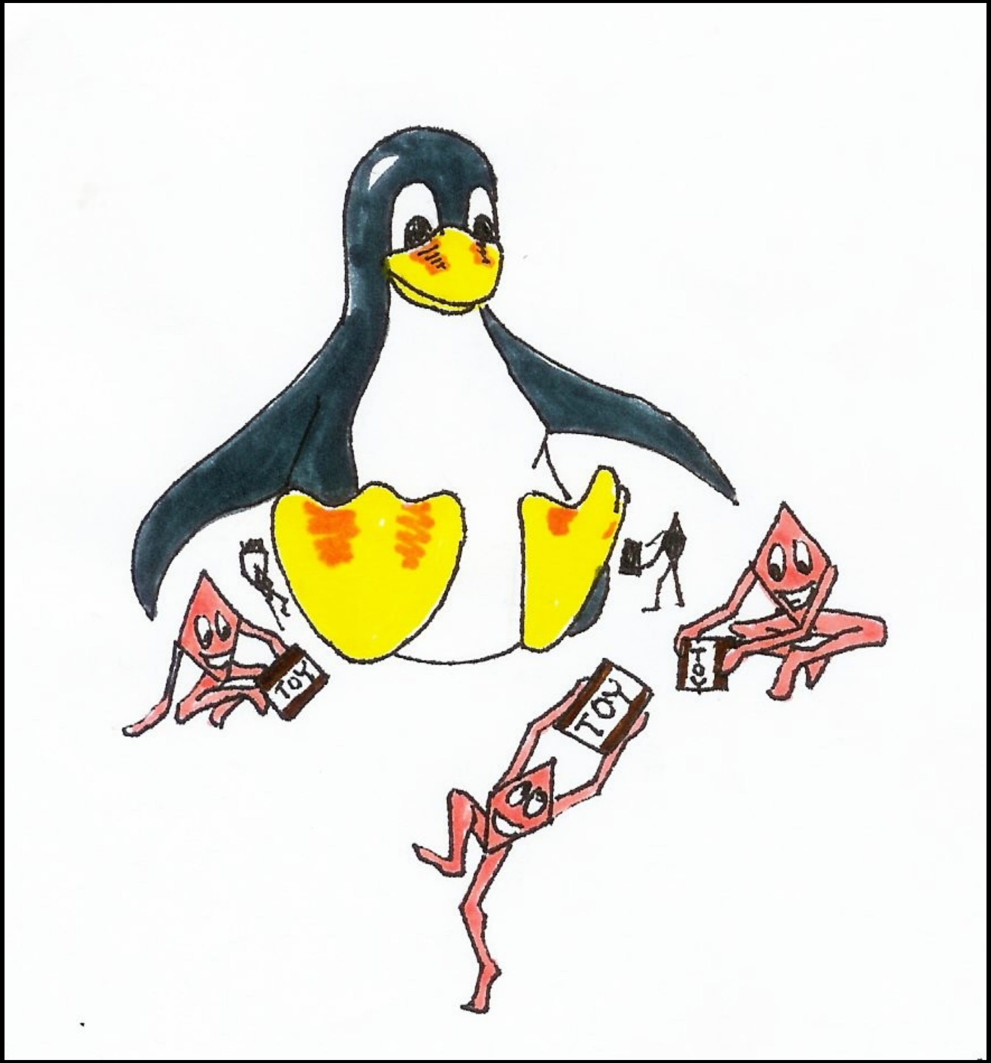
The Promise of Data Locking (Usually Fulfilled)



Potential Data Locking Problem: Data Skew



Data Ownership (Assuming Even Data Usage)

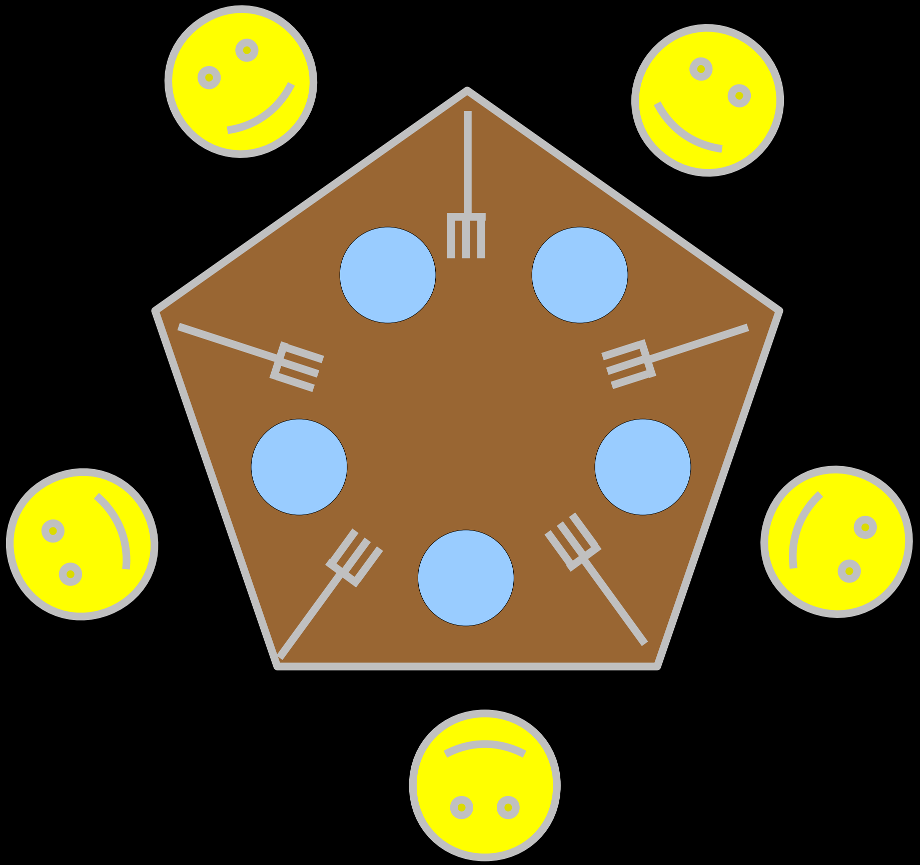


Concurrency Problem #2: Human Error

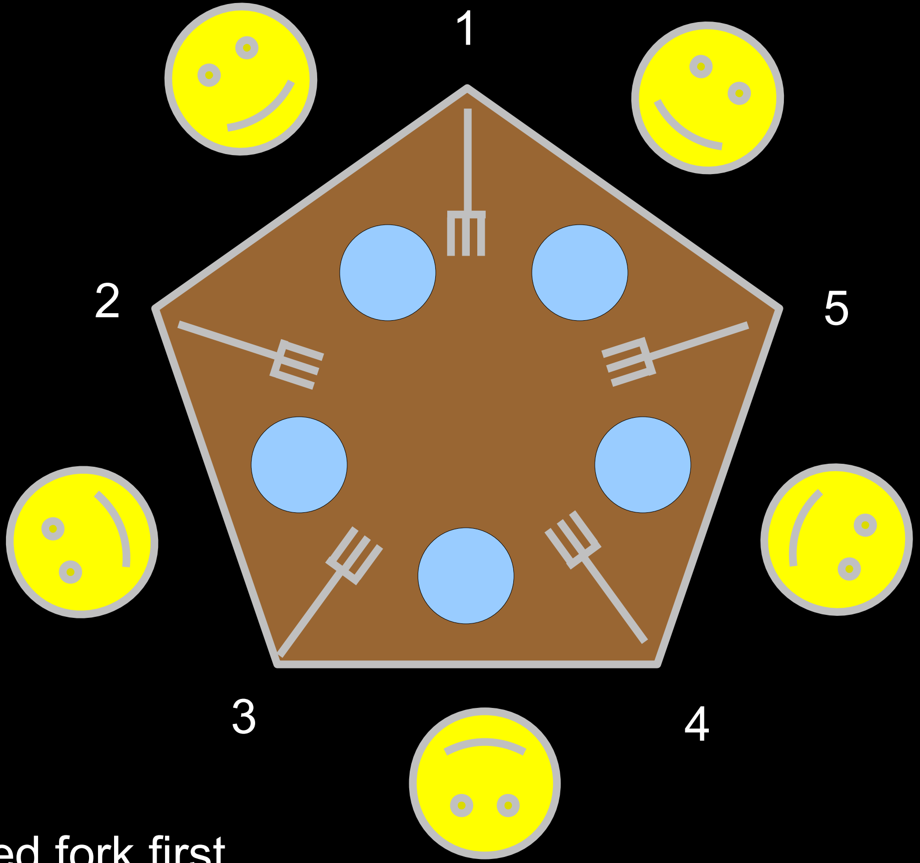
- **Deadlock:**
  - T1: spin_lock(&a); spin_lock(&b);
  - T2: spin_lock(&b); spin_lock(&a);
- **Data races with hand-coded locking primitives:**
  - while (l != -1)
    - continue;
  - l = me;
  - /* critical section
  - l = -1;
- **Memory barriers require either weak memory ordering or more than two x86 CPUs**
  - And beyond the scope of this talk in any case...

Deadlock: Dining Philosophers Problem

Each philosopher requires two forks to eat.
Need to avoid starvation.

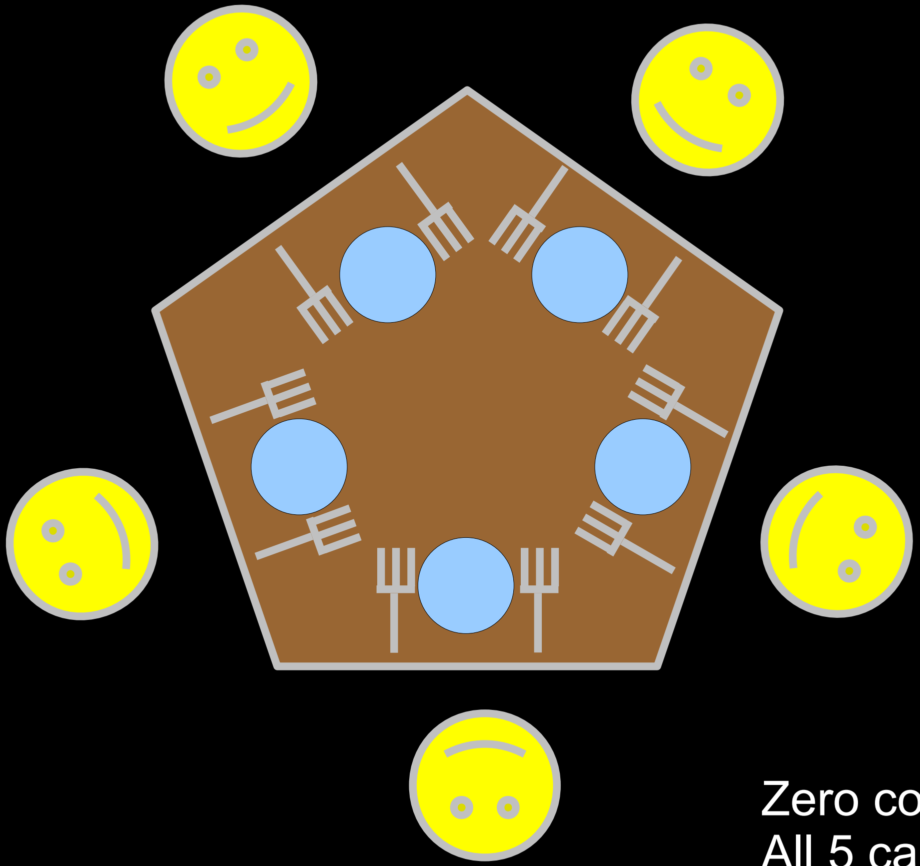


Deadlock: Dining Philosophers Solution: Traditional



Locking hierarchy.
Pick up low-numbered fork first,
preventing deadlock.

Deadlock: Dining Philosophers Solution: Paul



Zero contention.
All 5 can eat concurrently.
Excellent disease control.

Deadlock: Dining Philosophers Solution

- **Objections to Paul's solution:**
  - You can't just change the rules like that!!!
    - There was no rule stating that forks could not be added...
  - Dining Philosophers Problem valuable lock-hierarchy teaching tool – you just destroyed it!!!
    - Lock hierarchy is indeed very valuable and widely used, so the restriction “there can only be five forks” does indeed have its place, even if it didn't appear in this instance of the Dining Philosophers Problem.
    - But the lesson of transforming the problem into perfectly partitionable form is also very valuable, and given the wide availability of cheap multiprocessors, most desperately needed.
  - But what if each fork cost a million dollars?
    - Then we would permit only two forks! Or make the philosophers eat with their fingers... ☺

Analysis of Hand-Coded Locking Primitive

Tools like Promela/spin automate this process.

		while (l != 1)	continue;		l = me;	/* critical section */	l = -1
	🟩						
while (l != 1)	🟩						
continue;							
	🟩	🟩		🟩			
l = me;				🟩	🟩	🟩	
/* critical section */						🟥	
l = -1							

Analysis of Hand-Coded Locking Primitive

- **Moral: Use the system-provided locking primitives**
- **If the performance of the system-provided locking primitives is insufficient:**
  - First revisit your design, striving for coarser-grained parallelism
  - Only if that doesn't work consider getting fancy
- **Getting fancy gives you lots of gray hair**
  - At least if you take the trouble to actually make it work reliably
- **What does it take to create reliable synchronization primitives?**

The Geneva Convention Inapplicable To Software

- **From the "Geneva Convention relative to the Treatment of Prisoners of War (http://www.unhchr.ch/html/menu3/b/91.htm):**
  - Persons taking no active part in the hostilities, including members of armed forces who have laid down their arms and those placed hors de combat by sickness, wounds, detention, or any other cause, shall in all circumstances be treated humanely, without any adverse distinction founded on race, colour, religion or faith, sex, birth or wealth, or any other similar criteria.
  - To this end the following acts are and shall remain prohibited at any time and in any place whatsoever with respect to the above-mentioned persons:
    - (a) Violence to life and person, in particular murder of all kinds, mutilation, cruel treatment and ***torture***;
- **Some debate as to exactly who this covers, but for the moment, inapplicable to software. So torture your software before it tortures you!!!**

Two Axioms and One Theorem

- **The only bug-free programs are trivial programs.**
- **A reliable program has no known bugs**

- **Therefore, any reliable non-trivial program will have at least one bug that you do not know about.**

Exercise (1/2)

- **git clone git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git**
- **Initialize your CodeSamples directory:**
  - cd perfbook/CodeSamples
  - make pthreads-x86
  - cd intro
  - make

Exercise (2/2)

- **Find and fix the bug in CodeSamples/intro/initrace.c**
  - Search for “---”, look at the following six lines
  - “make” and “./initrace 2” to test
- **Write a program to evaluate the performance of your fixed program vs. unconditionally acquiring the lock**
- **What is the purpose of the “while” loop immediately preceding the first “---”?**
  - What happens if you delete it from the original program?
  - What happens if you delete it after fixing the original program?
- **Design a lockless queue that permits a single thread enqueuing concurrently with a single thread dequeuing**
  - Relax the enqueue restriction, so that your design permits concurrent enqueues, but only a single dequeue
  - Relax the dequeue restriction

Summary

- **Avoiding concurrency is fair game**
  - As long as a single CPU provides enough power for you
  - And as long as you are not using Java...
- **Using simple-minded concurrency is fair game**
  - Embarrassing parallelism is an embarrassment of riches
- **If simple-minded concurrency is unworkable, data locking is your friend**
- **If data locking is unworkable, care is required**
  - Data ownership, RCU, reference counting, ...
  - Remember that any thread can pause at any point for any duration!!!

To Probe Deeper

- **Parallel Design Patterns**
  - http://www.rdrop.com/users/paulmck/scalability/paper/mutexdesignpat.pdf
- **Other Parallel Algorithms and Tools**
  - http://www.rdrop.com/users/paulmck/scalability/
- **What is RCU?**
  - Fundamentally: http://lwn.net/Articles/262464/
  - Usage: http://lwn.net/Articles/263130/
  - API: http://lwn.net/Articles/264090/
  - Linux-kernel usage: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html
  - Other RCU stuff: http://www.rdrop.com/users/paulmck/RCU/
- **Parallel Performance Programming (very raw draft)**
  - git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git

Legal Statement

- This work represents the views of the authors and does not necessarily represent the view of IBM.
- Linux is a copyright of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.