CS 3214: Computer Systems Lecture 17: Performance of Multi-threaded Programs

Instructor: Huaicheng Li

Oct 20 2022

VIRGINIA TECH™

Announcement

- Midterm on 10/27 (Thur), In-class (Surge 104C), 75min
- Exam includes 3-5 multipart questions
- Format
  - Closed book, closed notes, closed computer
  - One letter size "cheatsheet" (front + back)
- Covered topics
  - Processes: dual-mode, context/mode switching, process states
  - Process APIs, system calls, signals, basic I/Os
  - Linking and loading: static + dynamic linking, scoping
  - Multi-threaded programming: locks, semaphore, condition variable, thread-safety, deadlock
  - Resources:
    - CSAPP3e: Chapters 1, 7, 8, 9, 10, and 12
    - Project 1, 2
    - Exercises 1, 2, 3
- Special accommodations

Performance Considerations

- Ways to evaluate synchronization implementation
  - Correctness
  - Fairness
  - Performance
- Cost of locking
  - Indirect cost
    - resulting in loss of performance due to the use of locking
    - fully concurrently → partially concurrent
  - Direct cost
    - (involved in actions the system had to take to implement it)
    - ak.a, Lock function implementation overhead

Indirect Cost

- A microbenchmark / simulation experiment to measure locking cost
  - 5 CPU-bound processes contending for L locks
  - holding each lock for duration D
  - then running for duration U without lock
  - Thread chooses lock randomly.

Indirect Cost: Single Lock

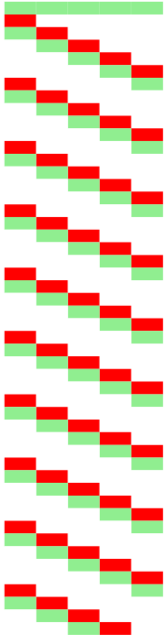

Figure 1: Fixed: U=2/D=2/L=1/40.8%

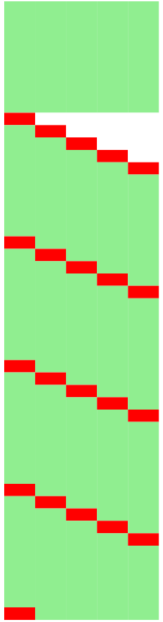

Figure 2: Fixed: U=18/D=2/L=1/96%

Figure 3: Poisson: U=2/D=2/L=1/41.6%

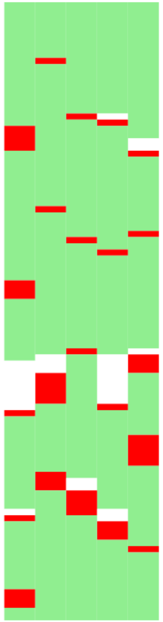

Figure 4: Poisson U=18/D=2/L=1/94.4%

Indirect Cost: Two Locks

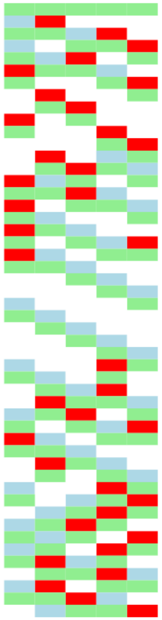

Figure 5: Fixed: U=2/D=2/L=2/65.2%

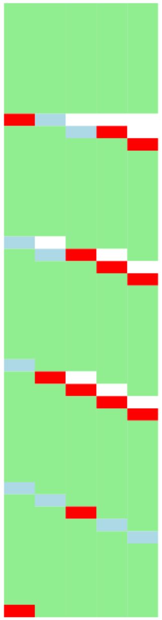

Figure 6: Fixed: U=18/D=2/L=2/96%

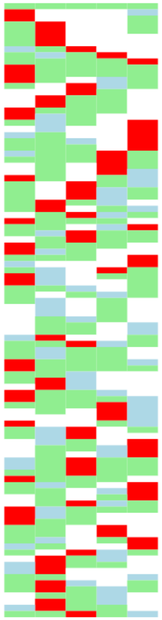

Figure 7: Poisson: U=2/D=2/L=2/72.2%

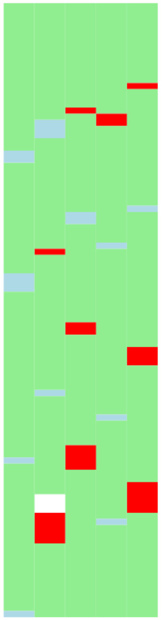

Figure 8: Poisson U=18/D=2/L=2/99.4%

Indirect Cost: Four Locks

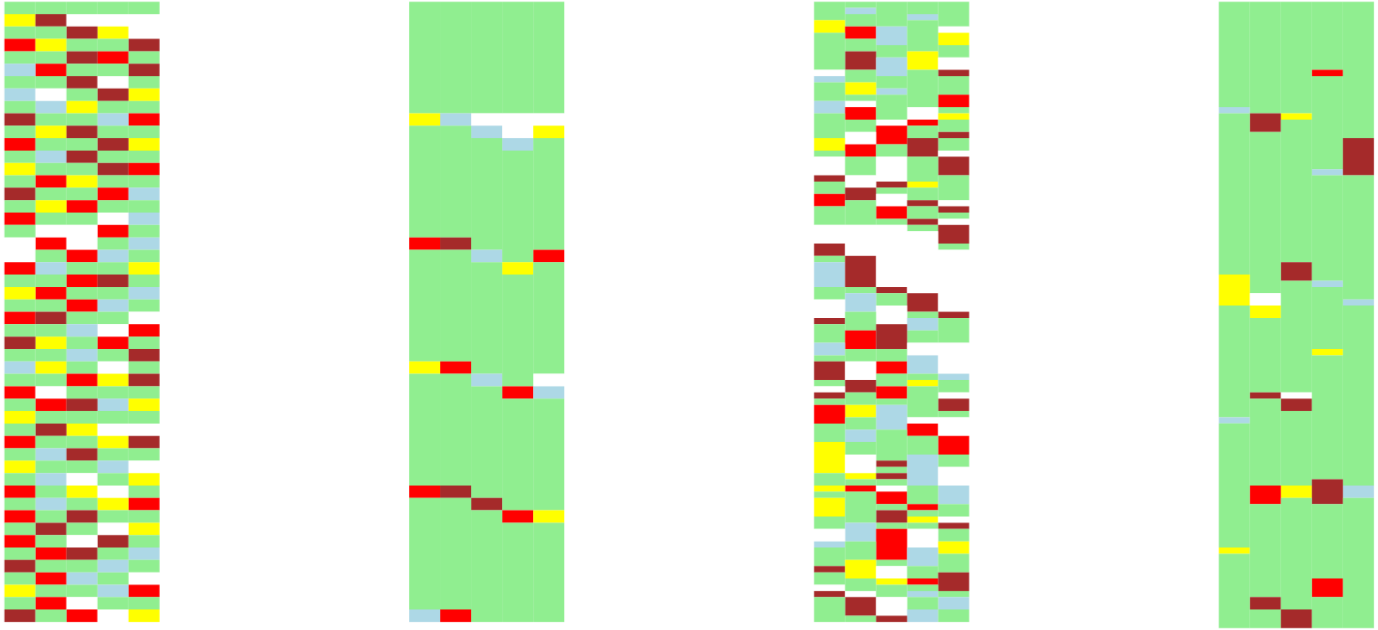

Figure 9: Fixed: U=2/D=2/L=4/89.6%

Figure 10: Fixed: U=18/D=2/L=4/98%

Figure 11: Poisson: U=2/D=2/L=4/79.2%

Figure 12: Poisson U=18/D=2/L=4/99.4%

VIRGINIA TECH

Indirect Cost: Loss of Parallelism

- Serialization due to locks diminishes CPU utilization and increases an individual task's latency
  - For parallel, mostly CPU-bound applications, it translates directly into loss of speedup
  - Particularly if locks are contended, no other tasks to run when threads are blocked
- This serialization effect would be exacerbated if blocked threads held locks (e.g., I/O, sleep, sem wait, pthread join?)
- Rule: Critical sections should not call any functions that may block, or else the critical section may become inaccessible

```
pthread_mutex_lock(&shutdownLock);
pthread_mutex_lock(&infoLock);
while (!moreInformation)
    pthread_cond_wait(&moreInfo, &infoLock);
pthread_mutex_unlock(&infoLock);
pthread_mutex_unlock(&shutdownLock);
```

```
pthread_mutex_lock(&lock);
read(fd, buf, sizeof(buf));
pthread_mutex_unlock(&lock);
```

Solution: Breaking Up Locks

- Cautionary side note: several large software systems were either never parallelized or started with a "big lock" approach: the Linux kernel, Python's GIL, gtk GUI lock
- Idea: instead of having lock L protect data (A, B, C) introduce locks LA, LB , LC to protect A, B, and C , respectively.
- Thus, updates to A will not prevent simultaneous updates to B
- This introduces 3 risks
  - Higher risk of atomicity violations: if A and B must be updated in tandem (atomically) - say update to B is dependent on A having a value, both locks must be held. Always holding both locks negates purpose of having 2 locks; not holding them both where needed leads to atomicity violations
  - Higher risk of deadlocks: if there are situations where both locks must be held, a locking order must be established to avoid deadlocks
  - More frequent calls to lock/unlock translates to increased direct cost (locking overhead)

Direct Cost of Locking

- What happens under the hood in a call to pthread mutex lock()?
  - **Fast path:** an atomic instruction tries to acquire the lock (if available) without causing a mode switch (e.g. cmpxchg %rax, (%rbx)) - in memory flag that indicates if lock is available
  - For fast path numbers, see Jeff Dean/Peter Norvig/Colin Scott Numbers Every Programmer Should Know
    - 17× L1 reference, 4× L2 reference, 16 × main memory reference (**17ns as of 2010's**)
  - **Slow path:** if atomic instruction indicates that lock is already held, make system call (*futex_wait*) and inform kernel that thread should block. Then, context switch to other ready thread (if any)
- *pthread mutex unlock()?*
  - Fast path: just place lock into unlocked state
  - Slow path (someone is waiting for the lock): make system call (*futex_wakeup*) and inform kernel to wake up any waiting thread(s). These threads are unblocked (made ready), placed into ready queue, and eventually scheduled - another context switch
- Both mode and context switches can be costly
  - Pipeline stalls
  - Cache pollution