Week 3: Mutual Exclusion Principles, Parallel Performance, Spin Locks and Contention

MPCS 52060: Parallel Programming

University of Chicago

Principles of Concurrent Programs

Two types of formal properties in asynchronous computation:

- Safety Properties:
  - Nothing bad happens ever
  - For example - a traffic light never displays green in all directions, even if power fails.
- Liveness Properties:
  - States that a particular "good" thing will happen.
  - For example - a red light will eventually turn green.

[1]

[1] Art of Multiprocessor Programming, Copyright 2006 Elsevier Inc

Formalizing Critical Sections

Synchronization primitives (e.g., locks) need to adhere to the following properties and principles about critical sections in order to be correct:

- Mutual Exclusion Property:
  - Critical sections of different threads do not overlap. Only one thread is executing a critical section at a time
  - Guarantees that a computation's results are correct.
  - This is a safety property.

- Deadlock-freedom property:
  - If multiple threads simultaneously request to enter a critical section, then it must allow one to proceed.
  - Threads outside the critical section have no say in which thread can proceed into the critical section, only those currently waiting have influence.
  - It implies the system never "freezes".
  - This is a liveness property.

Formalizing Critical Sections (cont.)

- Starvation-freedom property:
  - Every thread that attempts to acquire the lock eventually succeeds.
  - Many mutual exclusive algorithms in practice are not starvation free because its less likely starvation will occur in those algorithms.
  - There is no guarantee on how long thread will wait to acquire the lock.
  - Also known as lockout freedom or bounded-waiting
  - This is a liveness property.
- Fairness Principle:
  - A thread who just left the critical section cannot immediately re-enter the critical section if other threads have already requested to enter the critical section.
  - Some algorithms place bounds on how long a thread can wait.

Theoretical ways of implementing locks:

- See Companion Slides: 2-44

Parallel Performance

Amdahl's Law

To calculate how much a computation can be sped up by running part of it in parallel, can be done using Amdahl's Law:

- Potential program speedup is defined by the fraction of code (P) that can be parallelized

$$Speedup = \frac{1}{(1-p)} \tag{1}$$

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory). If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- See Companion slides: 34-53 for more examples.

With adding in the number of processors, the speed up can be modeled as such:

$$Speedup = \frac{1}{\frac{P}{N} + S} \tag{2}$$

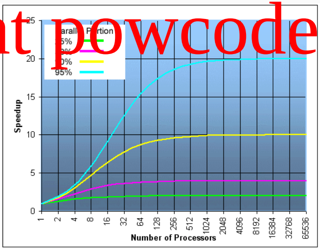
P = parallel fraction, N = number of processors, and S = serial fraction

It becomes obvious that there are limits to the scalability of parallelism:

speedup

N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

[2]



[3]

Practical Implementation of Locks

Go has a package called *sync* that provides basic synchronization primitives such as mutual exclusion locks.

The *sync* package Go's provides mutual exclusion with *sync.Mutex* and its two methods:

- *m.Lock()*: locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.
- *m.Unlock()*: Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.

Busy-Waiting (Spinning)

A lock requires making threads wait until its their turn to enter the critical section. One implementation of a lock uses the notion of spinning to make a thread wait:

- When spinning, a thread repeatedly tests a condition but, effectively, does no useful work until the condition has the appropriate value.
- Can be very wasteful of CPU cycles.
- Can also be unreliable if compiler optimization is turned on.
- Can be dangerous if not done in an deterministic way.

With caches, spinning becomes practical

- First time - Load flag bit into cache
- As long as it doesn't change

  - Hit in cache (no interconnect used)
- When it changes
  - One-time cost (a cache-miss)

  - Observes data has changed and must go grab it from main memory, which stops it from spinning.

Review: Low-level Hardware Synchronization Primitives

Many of the low-level synchronization primitives (e.g., locks, monitors, etc.) are built off of specialized hardware primitives/instructions (also known as atomic operations):

- On a shared memory system, an operation is consider atomic if it completes in a single step relative to other threads.
- No other thread can observe the modification to that shared variable half-way through its operation.

Review: Overview of Synchronization Primitives

Hardware provides simple low-level atomic operations:

- x86 load and store of words
- Special instructions:
  - compare-and-swap (CAS) (AMD, Intel, Sun)
  - pair of test-and-set instructions: loaded-linked and store-conditional (LL/SC) (ARM, IBM PowerPC, etc.)
- Provided in Go: `import "sync/atomic"`

We use those simple low-level atomic operations to build higher-level synchronization primitives:

- Lock (Today)
- Monitor
- Semaphore
- Conditional Variable
- Barrier

Problems with Atomic Operations

Best practices is to use atomic operations sparingly because:
Problem with atomic operations:

- Most atomic operations are implemented using CAS or (LL/SC) which take significant more cycles to complete than a simple load or store instruction.

- Causes a memory fence which forces the write-back buffer to be sent to main memory. This process can then stall other processors from reading/writing to main memory

- Prevents out-of-order execution and various compiler optimizations.
- Cost to performance varies depending on architectures, program design, etc.
- Adds more hardware complexity.

Practical Implementation of Locks

Implementations

- TAS Lock
- TTAS Lock
- Exponential Backoff Lock
- Anderson Lock (ALock)
- CLH Lock
- MCS Lock
- Queue Locks
- Time-out Locks
- See Companion Slides: 54-242