Contents

Process Synchronisation

- Need to be able to coordinate processes working on a common task
- Lock variables (mutexes) are used to coordinate or synchronise processes
- Need an architecture-supported arbitration mechanism to decide which processor gets access to the lock variable
  - Single bus provides arbitration mechanism, since the bus is the only path to memory - the process that gets the bus wins
- Need an architecture-supported operation that locks the variable
  - Locking can be done via an *atomic compare and swap operation* (processor can both read a location and set it to the locked state in the same bus operation)

Atomic Compare and Swap

- Machine instruction used to synchronise threatds
  - Threads may be running on different cores
- Compare and swap
  - Acts like a simple C function
  - Completes atomically
  - No possibility of the memory location being modified
  - Can be used to implement locks
- Most locks use compare and swap to attempt to acquire the lock
- Major different between different locking algorithms is what to do if you fail to acquire the lock
- Two main choices
  - Go into a loop and repeatedly try to acquire the lock

    * Known as a spin lock
    * Keeps spinning until another thread releases the lock
    * Usually faster than putting a thread to sleep
        · Acquire the lock as soon as it released
        · No need to interact with operating system
        · Works well when lock is seldom contended
    * But wasteful of resources and energy when the lock is contended
        · Thread busy spins doing busy work
        · Large number of memory operations on bus
        · Disaster scenario is spinning while waiting for a thread that is not running
  - Put the current thread to sleep and add it to a queue of threads waiting for the lock to be released
    * Usually slower
        · Need operating system call to put thread to sleep
        · Need to manage queue of waiting threads
        · Every time the lock is released, you must check the list of waiters
    * Sleeping locks make better use of total resources
        · Core is released while thread waits
        · Works well when lock is often contended and there are more threads than cores
        · Especially important in virtualised systems
  - Many lock algorithms are hybrids
    * Spin for a little while to see if lock is released quickly
    * Then go to sleep if it is not
    * Used by Linux futex (fast mutex) library

Memory Fences

- Modern superscalar processors typically execute instructions out of order
  - In a different order to the original program
- Loads and stores can be executed out of order
  - Processor may speculate that the load takes data from a different address to the store
  - Processor may actually have load and store addresses, and thus be able to tell which conflict and which are independent
- Each core typically has a load/store queue
  - Stores are written to the queue and will be flushed to memory eventually

  - A load may take data directly from the load/store queue if there has been a recent store to the location
  - So a load instruction may not ever reach memory
  - If there is already a store to a memory location in the load/store queue, an additional store may overwrite the existing store

Atomic Increment

- Alternative to atomic compare and swap
  - Atomically increments a value in memory
  - Can be used for multithreaded counting without locks
- Atomic increment locks
  - Can build locks from atomic increment
  - Known as ticket lock
  - Based on ticket system in passport office, GNIB, or Argos
- Queueing spin lock
  - Multiple threads can be waiting for a lock
  - With a regular spin lock, all waiting threads compete for the lock when it becomes free
  - With a queueing spin lock, the waiting threads for an orderly FIFO queue
  - Guarantees fairness