

Concurrency

Advanced Systems Programming (H/M)

Lecture 7

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Implications of Multicore Systems
  - Memory models and their implications for concurrency
  - Multi-threading, locking, and the limitations of lock-based concurrency
- Transactions
- Message passing
- Race Conditions

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Models and Multicore Systems

- Hardware trends: multicore with non-uniform memory access
  - Increasing numbers of cores, for performance
  - Cache coherency increasingly expensive → cores communicate by message passing, memory is remote
- When do writes made by one core become visible to other cores?
  - What is the **memory model** for the language?
  - Prohibitively expensive for all threads on all cores to have the exact same view of memory (“sequential consistency”)
  - For performance, allow cores inconsistent views of memory, except at synchronisation points; introduce synchronisation primitives with well-defined semantics
  - Hardware guarantees vary between processors
  - Differences hidden by language runtime, provided language has a clear memory model

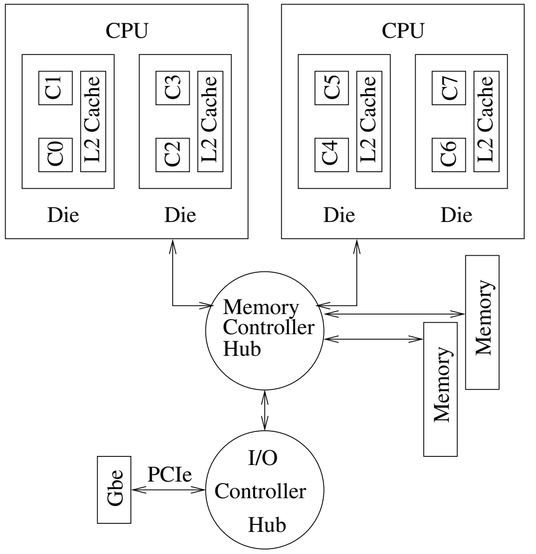


Figure 1. Structure of the Intel system

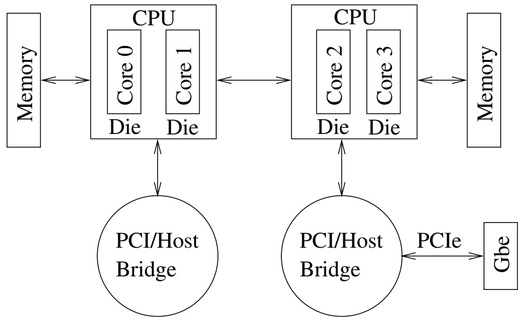


Figure 2. Structure of the AMD system

A. Schüpbach, *et al.*, Embracing diversity in the Barrelfish manycore operating system. Proc. Workshop on Managed Many-Core Systems, Boston, MA, USA, June 2008. ACM.

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Memory Models: Java

- Java has a formally defined memory model

Java Language Specification, Chapter 17
http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

  - Changes to a field are seen in program order *within a thread*
  - Changes to a field made by one thread are visible to other threads as follows:
    - If a **`volatile`** field is changed, that change is done atomically and immediately becomes visible to other threads
    - If a non-**`volatile`** field is changed while holding a lock, and that lock is then released by the writing thread and acquired by the reading thread, then the change becomes visible to the reading thread
    - If a new thread is created, it sees the state of the system as if it had just acquired a lock that had just been released by the creating thread
    - If a thread terminates, changes it made become visible to other threads
    - Access to all 32-bit fields is atomic
      - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
      - This is not true for for **`long`** and **`double`** fields, which are 64-bits in size, where writes are only atomic if field is **`volatile`** or if a lock is held

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

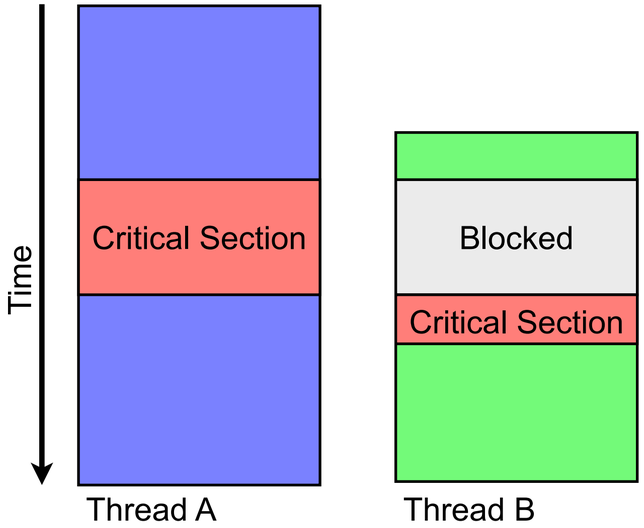
Memory Models: Others

- Java is unusual in having such a clearly-specified memory model
  - Other languages are less well specified, running the risk that new processor designs can subtly break previously working programs
  - C and C++ have historically had *very* poorly specified memory models
    - Latest versions of standards address this, with memory models heavily influenced by the Java memory model
    - Not yet widely implemented
  - Rust does not (yet) have a fully specified memory model
    - Recognised as a limitation – research efforts underway to fix this
    - Complicated by multiplicity of reference types and **`unsafe`** code

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Concurrency, Threads, and Locks

- Operating system exposes concurrency as **processes** and **threads**
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
    - Most operating systems started with processes and message passing, and added threads later due to programmer demand
- The memory model specifies how concurrent access to shared memory works
  - Synchronisation by explicit locks around critical sections
    - **`synchronized`** methods and statements in Java
    - **`pthread_mutex_lock()`**/**`pthread_mutex_unlock()`** in C
  - Synchronisation by **`volatile`** fields
  - Limited guarantees about unlocked concurrent access to shared memory



Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

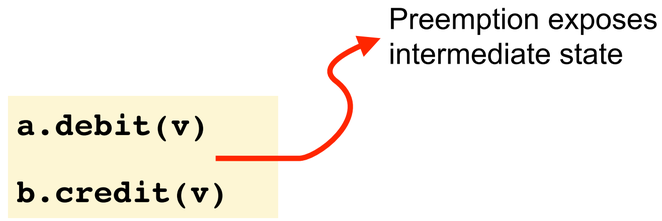
Limitations of Lock-based Concurrency

- Major problems with lock-based concurrency:
  - Difficult to define a memory model that enables good performance, while allowing programmers to reason about the code
  - Difficult to ensure correctness when composing code
    - Difficult to enforce correct locking
    - Difficult to guarantee freedom from deadlocks
  - Failures are silent – errors tend to manifest only under heavy load
  - Balancing performance and correctness difficult – easy to over- or under-lock systems

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Composition of Lock-based Code

- Correctness of small-scale code using locks can, in theory, be ensured by careful coding
- A more fundamental issue: lock-based code does not compose to larger scale
  - Assume a correctly locked bank account class, with methods to credit and debit money from an account
  - Want to take money from **`a`** and move it to **`b`**, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to **`a`** and **`b`** while the transfer is in progress
  - The individual operations are correct, but the combined operation is not



- This is lack of abstraction a limitation of the lock-based concurrency model, and cannot be fixed by careful coding
- Locking requirements form part of the API of an object

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Alternative Concurrency Models

- Concurrency increasingly important
  - Multicore systems now ubiquitous
  - Asynchronous interactions between software and hardware devices
- Threads and synchronisation primitives problematic

- Are there alternatives that avoid these issues?
  - Transactions
  - Message passing

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Implications of Multicore

- Memory Models
- Concurrency, threads, and locks

BY ND