University of Glasgow

School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

# Implications of Concurrency

Advanced Operating Systems

Lecture 9

# Memory and Multicore Systems

- Hardware trends: multicore with non-uniform memory access
- Cache coherency is expensive, since the cores communicate by message passing and memory is remote

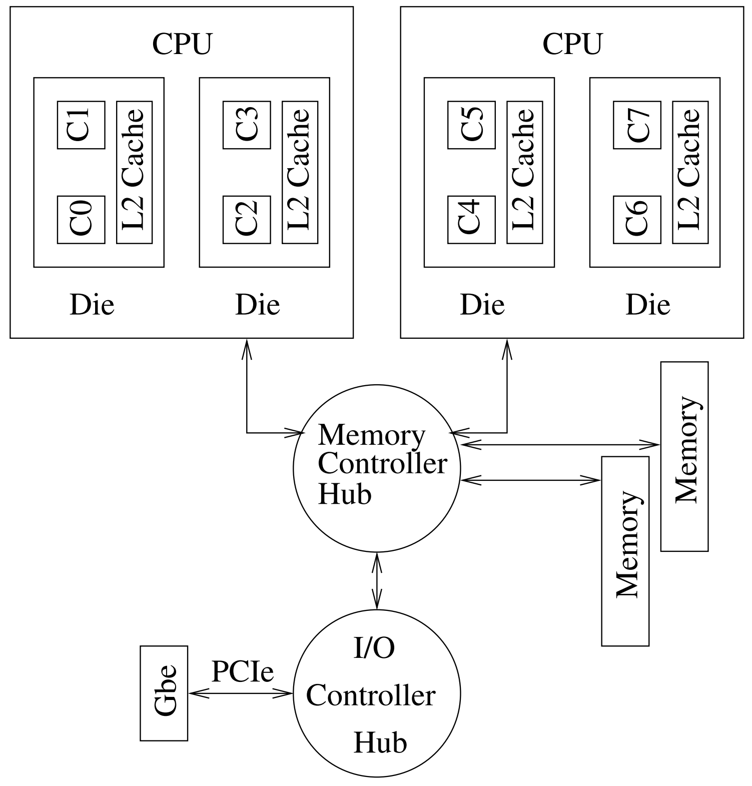


**Figure 1.** Structure of the Intel system

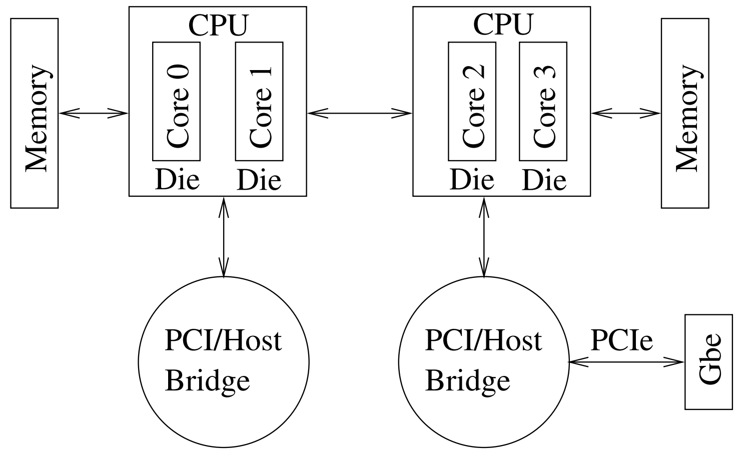


**Figure 2.** Structure of the AMD system

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.

## Multicore Memory Models

- When do writes made by one core become visible to other cores?
  - Prohibitively expensive for all threads on all core 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
  - Varies between processor architectures – differences generally hidden by language runtime, *provided language has a clear memory model*

## Multicore Memory Models

- Memory Model defines space in which language runtime and processor architecture can innovate, without breaking programs
    - Synchronisation between threads occurs only at well-defined instants; memory may appear inconsistent between these times, if that helps the processor and/or runtime system performance
    - Without well-defined memory model, cannot reason about lock-based code
    - Essential for portable code using locks and shared memory

## Example: Java Memory Model

[Somewhat simplified: see Java Language Specification, Chapter 17, for full details http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf]

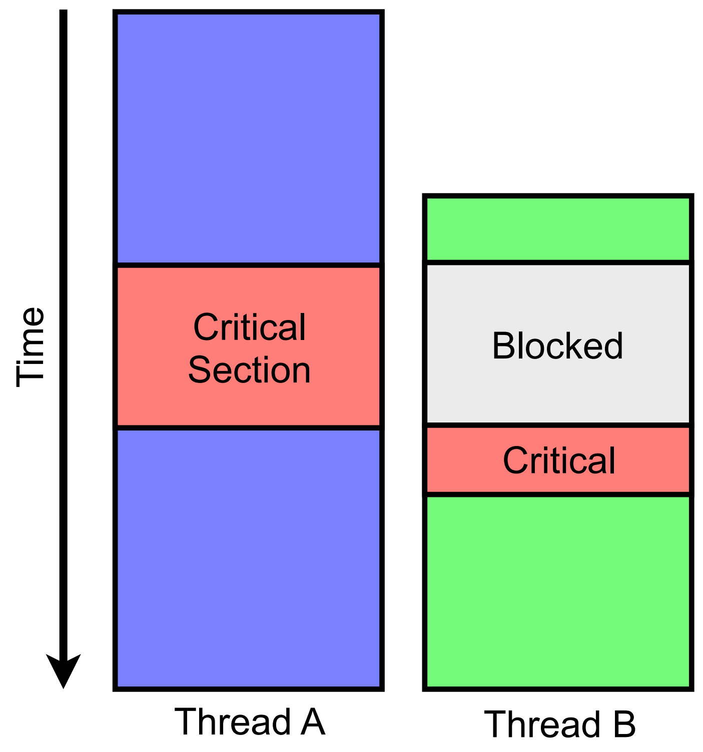
- Java has a formally defined memory model
- Between threads:
    - Changes to a field made by one thread are visible to other threads if:
        - The writing thread has released a synchronisation lock, and that same lock has subsequently been acquired by the reading thread (writes with lock held are atomic to other locked code)
        - If a thread writes to a field declared `volatile`, that write is done atomically, and immediately becomes visible to other threads
        - A newly created thread sees the state of the system as if it had just acquired a synchronisation lock that had just been released by the creating thread
        - When a thread terminates, its writes complete and become visible to other threads
    - Access to fields is atomic
        - i.e., you can never observe a half-way completed write, even if incorrectly synchronised
        - Except for `long` and `double` fields, for which writes are only atomic if field is `volatile`, or if a synchronisation lock is held
- Within a thread: actions are seen in program order

## Multicore Memory Models

- 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, but not widely used

## Concurrency, Threads, and Locks

- Operating systems expose concurrency via *processes* and *threads*
  - Processes are isolated with separate memory areas
  - Threads share access to a common pool of memory
- The processor/language memory models specify how concurrent access to shared memory works
  - Generally enforce synchronisation via explicit locks around critical sections (e.g., Java synchronized methods and statements; pthread mutexes)
  - Very limited guarantees about unlocked concurrent access to shared memory



## 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

## Composition of Lock-based Code

- Correctness of small-scale code using locks can be ensured by careful coding (at least in theory)
- 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 `a1` and move it to `a2`, without exposing an intermediate state where the money is in neither account
  - Can't be done without locking all other access to `a1` *and* `a2` while the transfer is in progress
  - The individual operations are correct, but the combined operation is not

```
a1.debit(v)
a2.credit(v)
```

Preemption exposes intermediate state

- 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

## Alternative Concurrency Models

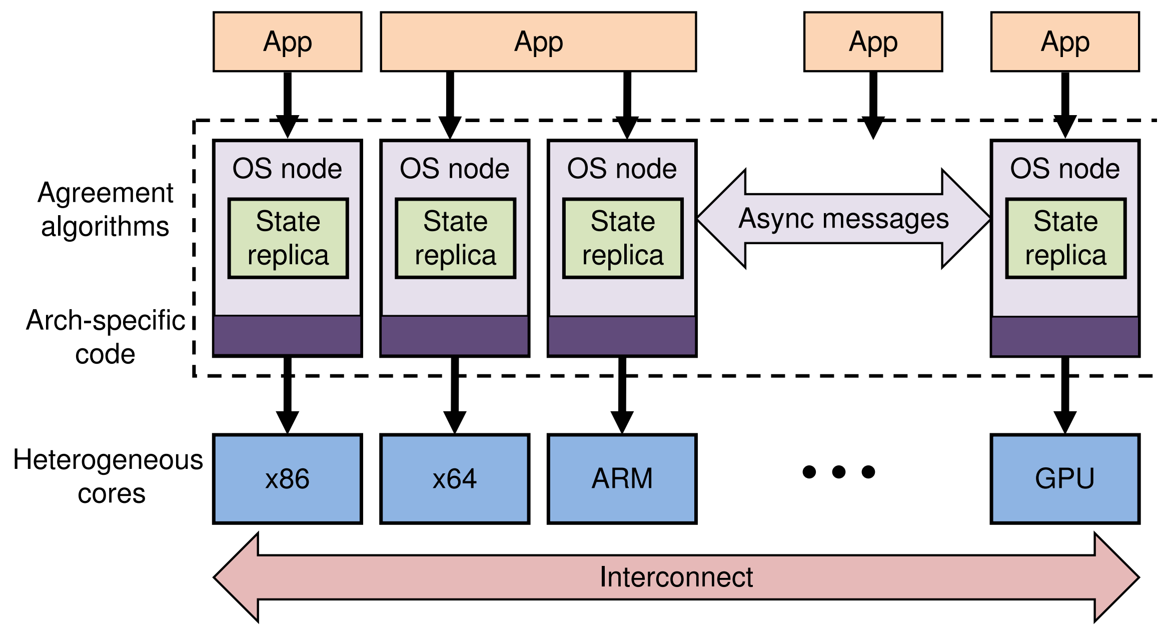
- Concurrency increasingly important
  - Multicore systems now ubiquitous
  - Asynchronous interactions between software and hardware devices
- Threads and synchronisation primitives problematic
- Are there alternatives that avoid these issues?
  - Message passing systems and actor-based languages
  - Transactional memory coupled with functional languages (e.g., Haskell) for automatic rollback and retry of transactions

## Implications for Operating System Design

- A single kernel instance may not be appropriate
  - Memory isn’t shared – don’t pretend it is!
  - There may be no single “central” processor to initialise the kernel
  - How to coordinate the kernel between peer processors?
- Multicore processors are increasing distributed systems at heart – can we embrace this?

## The Multi-kernel Model

- Build a distributed system that can use shared memory where possible as an optimisation, rather than a system that relies on shared memory
- The model is no longer that of a *single* operating system; rather a collection of cooperating kernels



Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

- Three design principles for a multi-kernel operating system
    - Make all inter-core communication explicit
    - Make OS structure hardware neutral
    - View state as replicated instead of shared

## Principle 1: Explicit Communication

- Multi-kernel model relies on message passing
  - The only shared memory used by the kernels is that used to implement message passing (user-space programs can request shared memory in the usual way, if desired)
    - Strict isolation of kernel instances can be enforced by hardware
    - Share immutable data – message passing, not shared state
  - Latency of message passing is explicitly visible
    - Leads to asynchronous designs, since it becomes obvious where the system will block waiting for a synchronous reply
    - Differs from conventional kernels which are primarily synchronous, since latencies are invisible
  - Kernels become simpler to verify – explicit communication can be validated using formals methods developed for network protocols

## Principle 2: Hardware Neutral Kernels

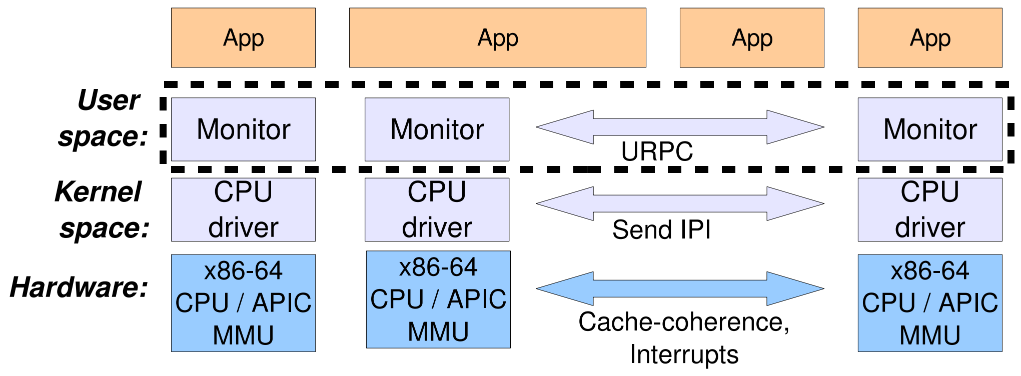
- Write clean, portable, code wherever possible
  - Low-level hardware access is necessarily processor/system specific
  - Message passing is performance critical: should use of system-specific optimisations where necessary
  - Device drivers and much other kernel code can be generic and portable – better suited for heterogeneity
  - Highly-optimised code is difficult to port
    - Optimisations tend to tie it to the details of a particular platform
    - The more variety of hardware platforms a multi-kernel must operate on, the better it is to have acceptable performance everywhere, than high-performance on one platform, poor elsewhere
- Hardware is changing faster than system software

## Principle 3: Replicated State

- A multi-kernel does not share state between cores
  - *All* data structures are local to each core
  - Anything needing global coordination must be managed using a distributed protocol
  - This includes things like the scheduler run-queues, network sockets, etc.
    - e.g., there is no way to list all running processes, without sending each core a message asking for its list, then combining the results
  - A distributed system of cooperating kernels, not a single multiprocessor kernel

# Multi-kernel Example: Barrelfish

- Implementation of multi-kernel model for x86 NUMA systems
- CPU drivers
  - Enforces memory protection, authorisation, and the security model
  - Schedules user-space processes for its core
  - Mediates access to the core and associated hardware (MMU, APIC, etc.)
  - Provides inter-process communication for applications on the core
  - Implementation is completely event-driven, single-threaded, and non-preemptable
  - ~7500 lines of code (C + assembler)
- Monitors
  - Coordinate system-wide state across cores
- Applications written to a subset of the POSIX APIs

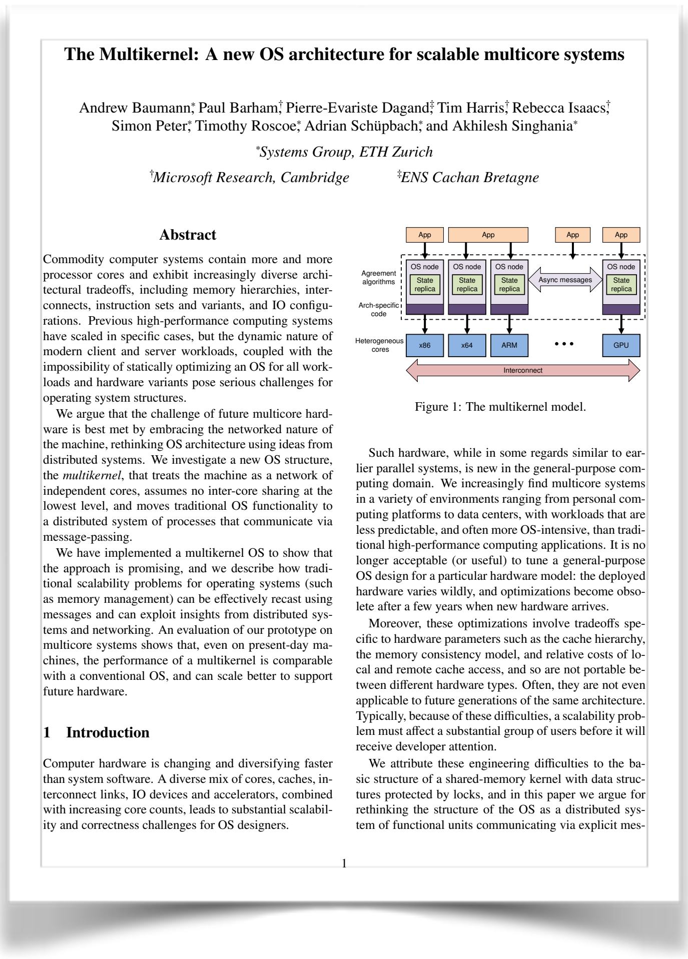


- Microkernel: network stack, memory allocation via capability system, etc., all run in user space
- Message passing tuned to details of AMD HyperTransport links and x86 cache-coherency protocols – highly system specific

# Further Reading and Discussion

- A. Baumann *et al.*, “The Multikernel: A new OS architecture for scalable multicore systems”, Proc. ACM SOSP 2009. DOI:10.1145/1629575.1629579

- Barrelfish is clearly an extreme: a shared-nothing system implemented on a hardware platform that permits some efficient sharing
  - Is it better to start with a shared-nothing model, and implement sharing as an optimisation, or start with a shared-state system, and introduce message passing?
- Where is the boundary for a Barrelfish-like system?
  - Distinction between a distributed multi-kernel and a distributed system of networked computers?


**The Multikernel: A new OS architecture for scalable multicore systems**

Andrew Baumann[*], Paul Barham[†], Pierre-Evariste Dagand[‡], Tim Harris[†], Rebecca Isaacs[†], Simon Peter[*], Timothy Roscoe[*], Adrian Schüpbach[*], and Akhilesh Singhania[*]

[*]*Systems Group, ETH Zurich*

[†]*Microsoft Research, Cambridge* [‡]*ENS Cachan Bretagne*

**Abstract**

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

**1 Introduction**

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Figure 1: The multikernel model.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit mes-

1