

Developing Multicore xTCA Applications with Embedded Linux:

Synchronizing Multiple Cores

SARA BIYABANI

System Architect

Outline

- How to Make xTCA Applications work with Multiple Cores
- What is Multicore?
- The Whole Picture/Problem Space
- Identify Top Issues Involved
 - Why is Synchronization Important?
 - Solutions
- Real-World Example Application

MultiCore

1

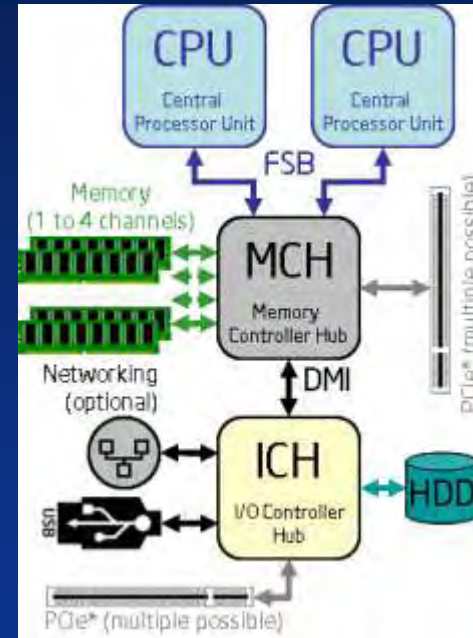
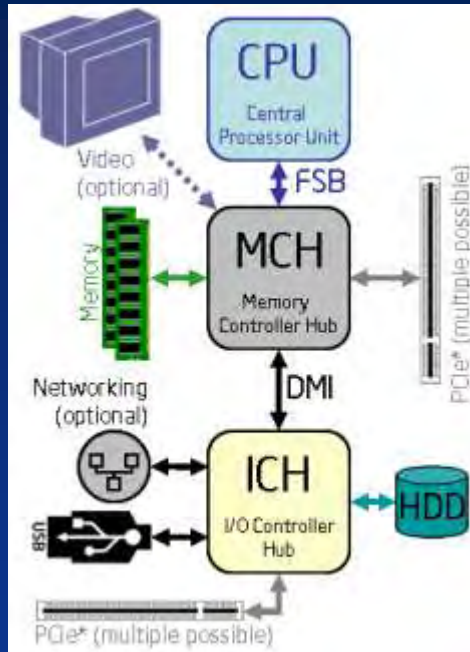
- Multiple Processors on a Single Die
- Almost All Architectures (ISAs) Have It
- Common Theme: Parallelism!

MultiCore

- “From there to here, and here to there, funny things are everywhere”! (Dr. Seuss)
- Example MultiCore Processors & SOCs:

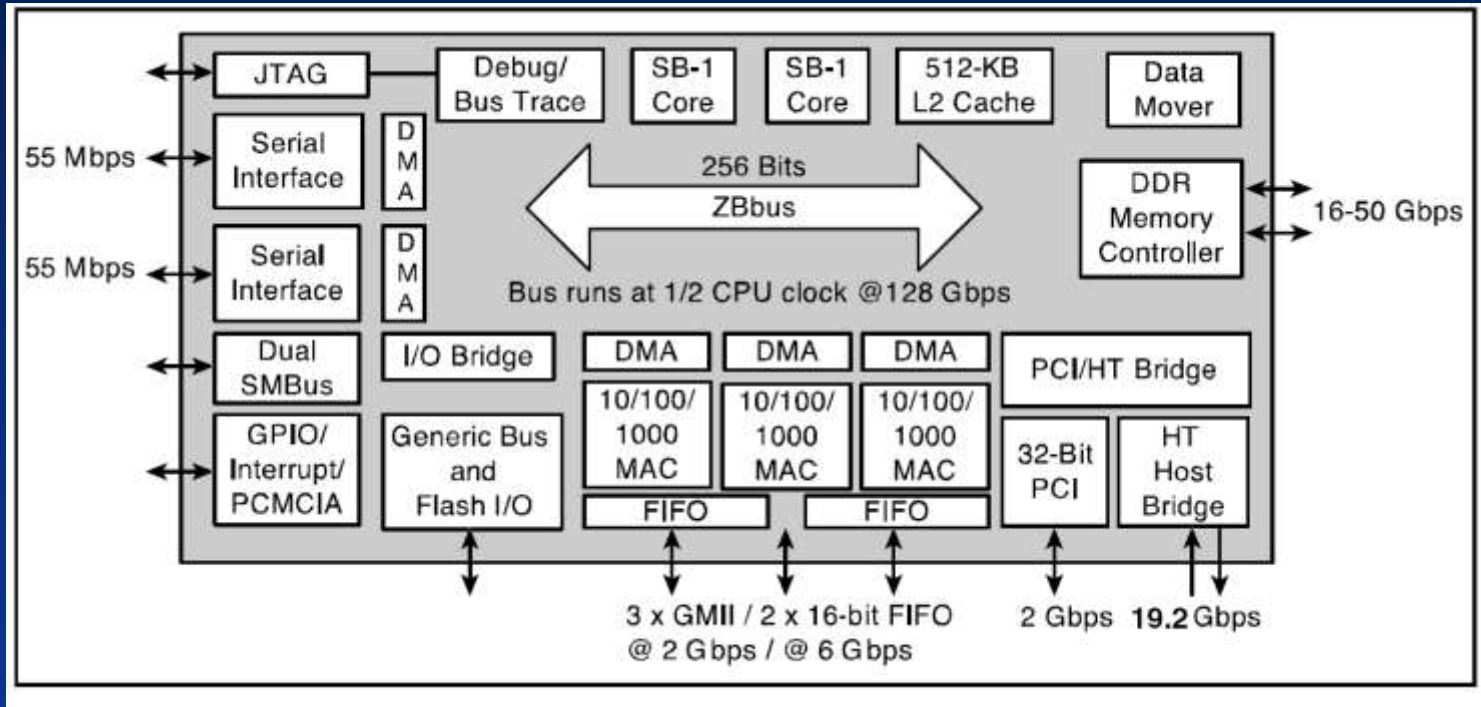
ISA	Example Processor / SOC	Number of Cores/Die
ARM	OMAP Cortex-A9 Series	2, 4
MIPS	BCM1455, 1480	2, 4
PowerPC	Cell Processor, Power7	many, 8
SPARC	Rainbow Falls	16
X86	Athlon, Opteron, Core 2 Duo, Quad	2, 4 2, 4
Other	Itanium, Nvidia Tesla GPU	2, many

Example: Typical Multicore Processor-based System/Platform

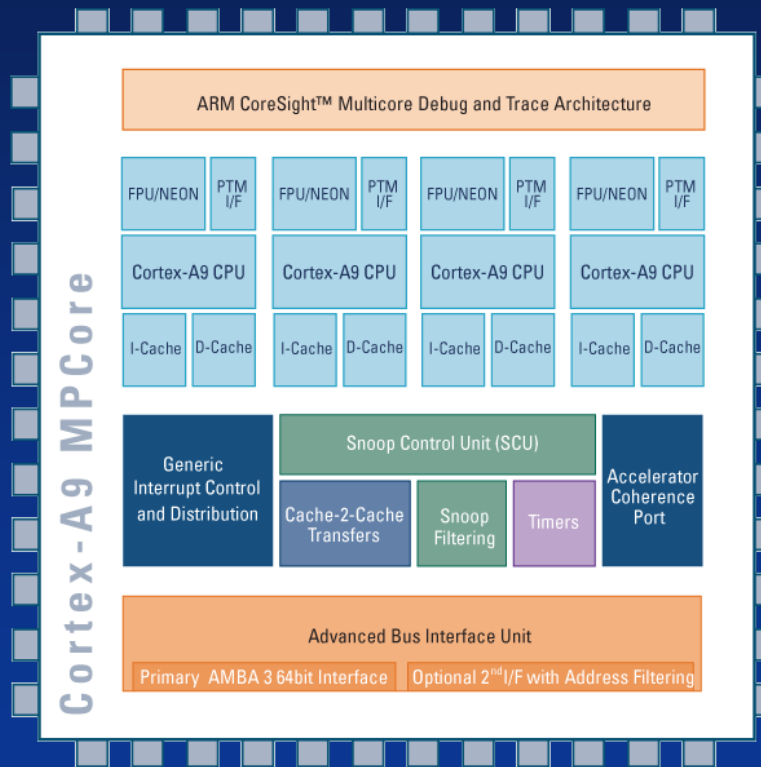


Example: Typical SOC

Dual Core MIPS-based BCM1250



Example: Typical Multicore Quad Core ARM Cortex-A9



How did we get here?

- Increased clock rates getting difficult
 - Limitations imposed by physics
 - Power, heat dissipation problems
- But it's still feasible to double transistor count
- Answers
 - Add parallelism within each core (superscalar)
 - Add more cores

Moore's Law

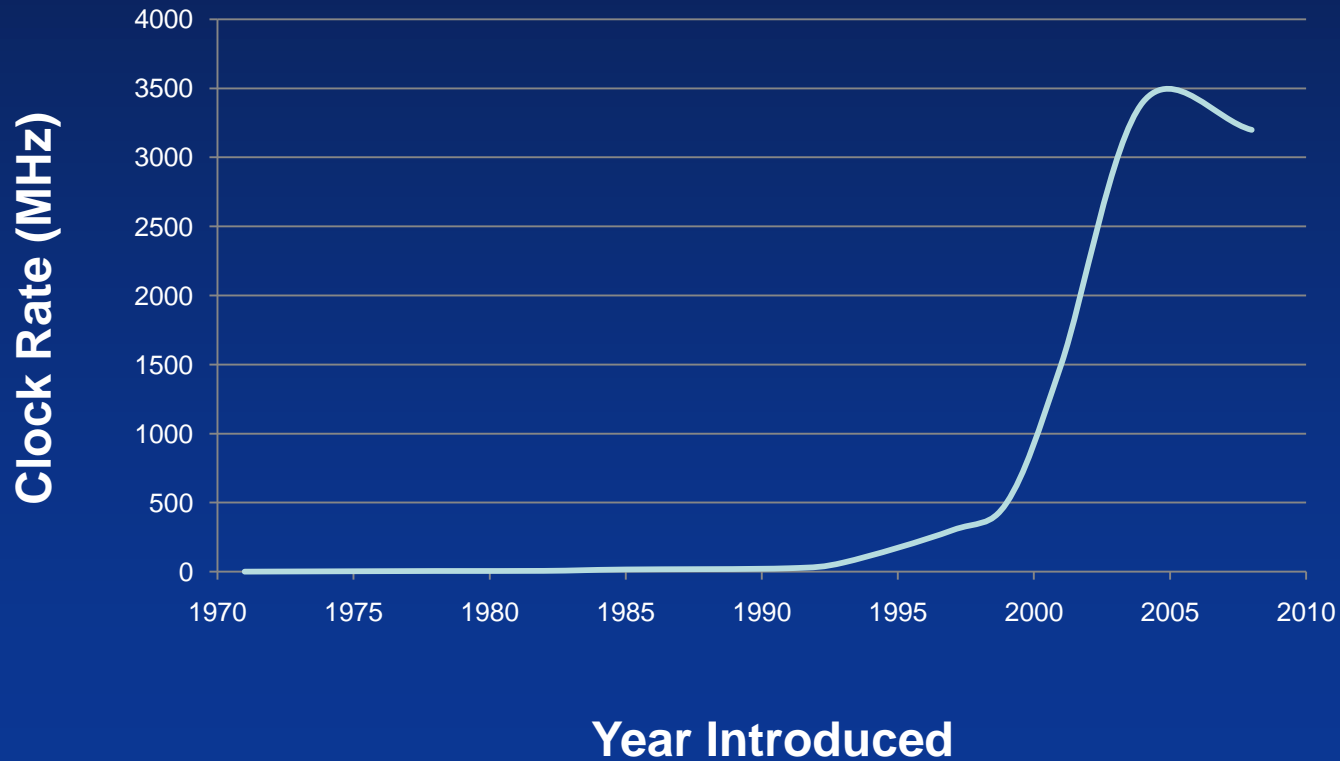
“Transistor density of Integrated Circuits Doubles every
18-24 months”

Caveats:

- * Empirical
- * Applies to Process technology
- * No mention of CPU Frequency
- * Does not mean off-chip memory or storage systems scale similarly

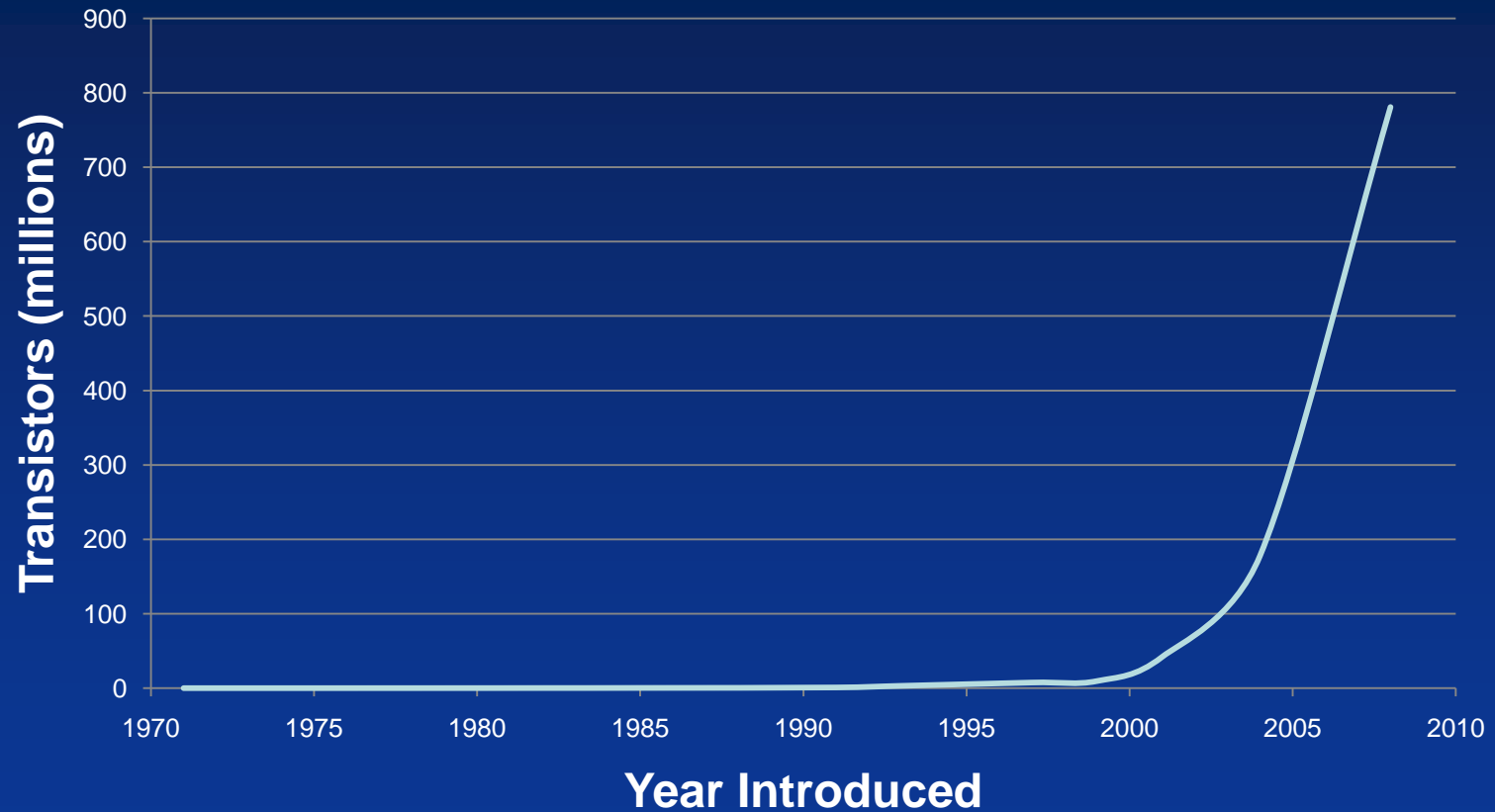
CPU Clock Rates are Leveling Off

Historical Intel® CPU Clock Rates



... but Moore's Law is still in effect

Transistors in Intel[®] Processors



Amdahl's Law

1

- Speedup is related to both Serial and Parallel portions of a program, as well as number of processors

$$\text{SPEEDUP} = \frac{1}{(1 - P) + (P / N)}$$

P = % Parallel
N = # of processors

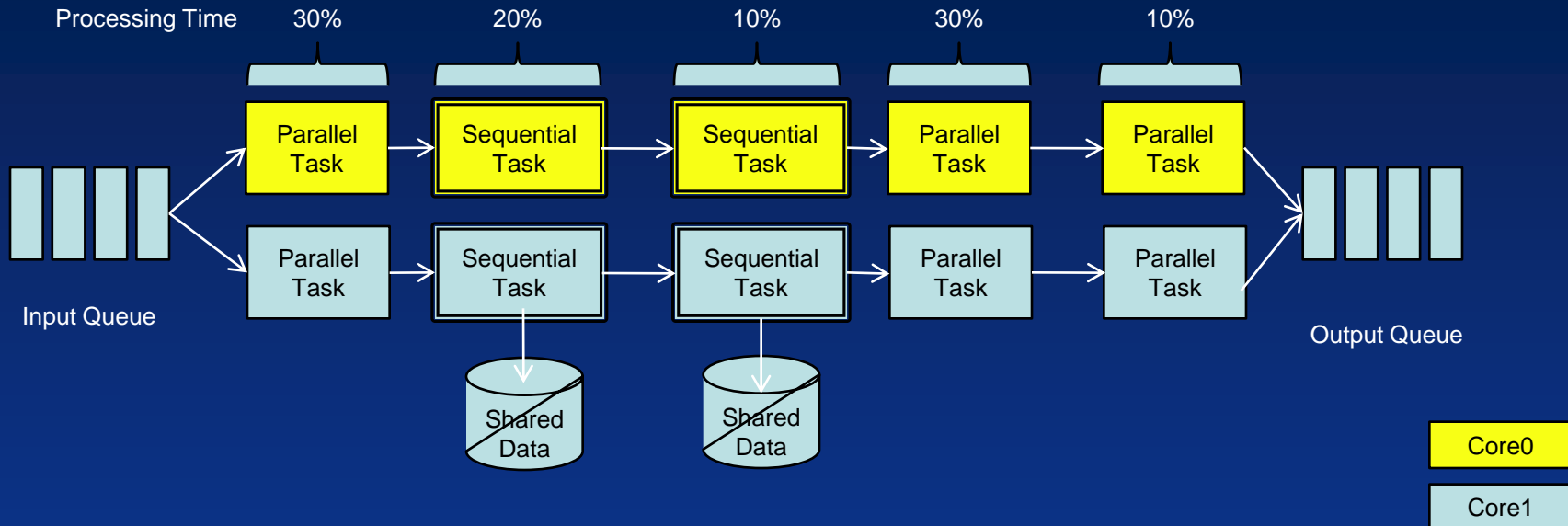
- IDEAL Speedup = N (= number of processors)

Amdahl's Law

2

- Doubling CPUs/cores does not double performance
- Serial portions of program limit speed of parallel portions
- Goal: Reduce serial portions of code
 - Usually not possible to remove all of it

Amdahl's Law limits speedup



- Applying Amdahl's Law:
 - $P = 70\% (0.7)$, $N = 2$ CPUs.
 - $\text{Speedup} = (1/((1-0.7)+(0.7/2))) = 1.54$

MultiCore

The Whole Picture

- On-Die Cores
- System/Platform
- Operating System
- xTCA Application

Issues: The Whole Picture

- **On-Die Cores:** coherence protocols, cache utilization, power management, memory & i/o configuration
- **System/Platform:** cpu & i/o (storage & network) load balancing, power management
- **Operating System:** synchronization/concurrency control, manageability, security, power management
- **xTCA Application:** concurrency control
- “Putting it all together is still more Art than Science”

Issue: On-Die “Coherence” Issues

- **On-Die Cores:**
- Memory Models
(Ordering of operations to a single location)
- Cache Coherence Protocols
- Instructions for Serializing
(ISA Support for implementing lock primitives)
- “Putting it all together is still more Art than Science”

“Embarrassingly Parallel” Applications

- Rendering of computer graphics:
 - In ray tracing, each pixel may be rendered independently.
 - In computer animation, each frame may be rendered independently.
- Brute force searches in cryptography.
- Searches (e.g. Google search)
- Mandelbrot set and other fractal calculations
- BLAST searches in bioinformatics.
- Large scale face recognition
- Climate models comparing independent scenarios.
- Genetic algorithms
- Evolutionary computation meta-heuristics.
- Ensemble calculations of numerical weather prediction.
- Event simulation and reconstruction in particle physics.

Making Applications Work with MultiCore

1) Rewrite (ideal)

Issues:

- Effort
- Algorithmic Changes for Parallelizing

2) Port existing apps

Issues:

- Locating Synchronization Points
- Identifying Shared Data Structures
- Selecting Correct Locks & Strategies

OS/Linux Features Needed for MultiCore

- User space
 - Threads Library
 - Proprietary Packages
 - PThreads
 - **POSIX Standard**
 - <https://computing.llnl.gov/tutorials/pthreads/>
 - <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>
 - Compilers, Debuggers & Profiling Tools
- Kernel space
 - Synchronization/Locking APIs

What are Synchronization APIs?

- Provide Mutual Exclusion
- Protect *Critical Sections* where Shared Resources are accessed (read or written)
Examples of shared resources:
HW registers, re-entrant code, SW data structures, etc.

Similar to dealing with kernel preemption & interrupt handling on a uni-processor

Synchronization in Linux: Mechanisms Available

1

- Synchronization / Locks
 - Semaphores
 - Mutex
 - Spinlocks
 - Completions

Synchronization in Linux: Mechanisms Available

2

- LOCK FREE Alternatives
 - Circular Buffer
 - Atomic Variables
 - Bit Operations
 - Seqlock
 - Read-copy-update, RCU

Synchronization in Linux: Scenarios (when to use what)

1

- Synchronization / Locks
 - Semaphores
 - Keeps a count, up to 'n' processes allowed through
 - Lock can be held for a LONG time
 - Mutex
 - Exactly one process allowed at a time
 - Lock can be held for a LONG time
 - Spinlocks
 - Exactly one process allowed at a time
 - Lock should be held for a SHORT time
 - Available only in the kernel

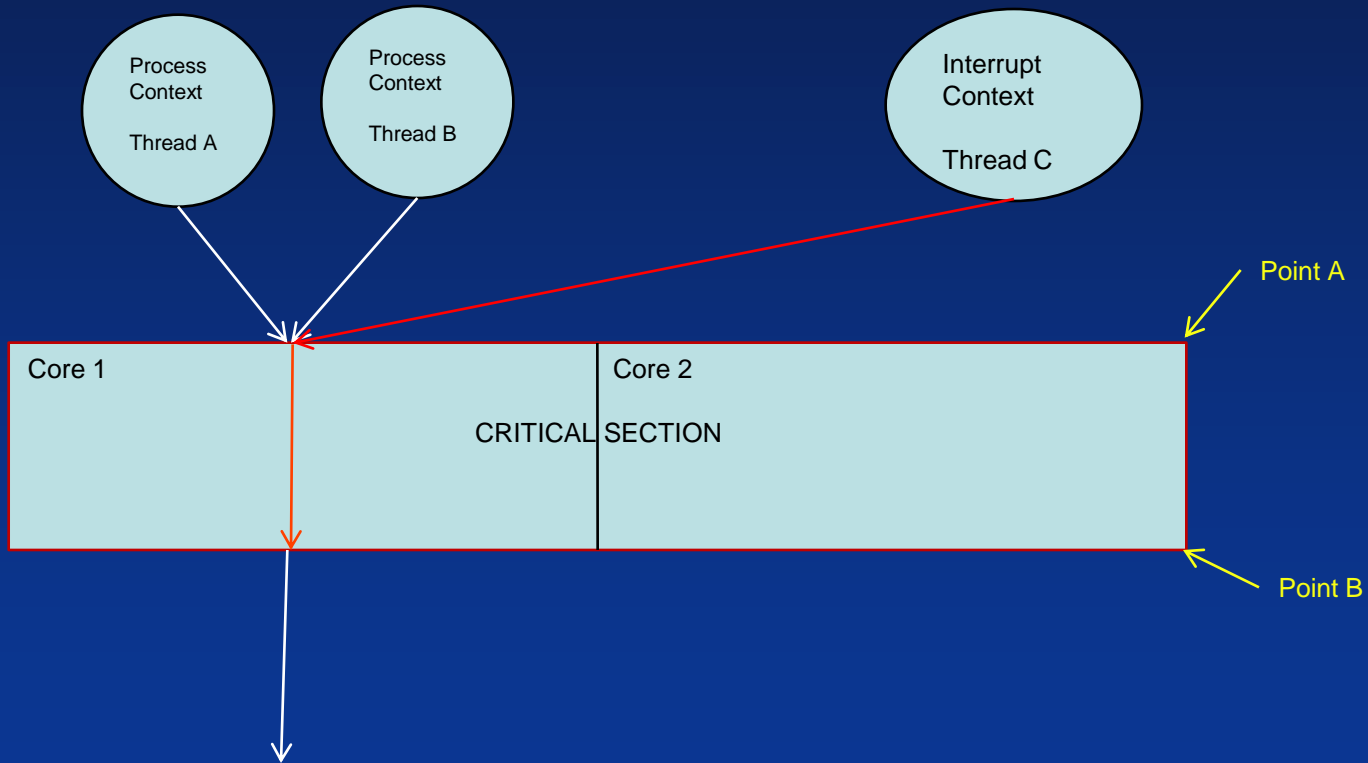
Synchronization in Linux: Scenarios (when to use what)

2

- LOCK FREE Alternatives
 - Circular Buffer
 - when there is One Writer and One Reader
 - Atomic Variables
 - read/modify/write of single variables
 - Bit Operations
 - modify/test individual bits atomically
 - Seqlock:
 - when write operations outnumber reads;
e.g. jiffies_64 variable
 - Read-copy-update, RCU
 - when Readers outnumber Writers

Synchronization: Example of Concurrency Control

1



Synchronization: Example of Concurrency Control

2

Threads A & B

@ Point A:

- 1) Save Interrupt state & Preemption on local CPU
- 2) Disable Interrupts on Local CPU
- 3) Get Lock:
`spin_lock_irqsave()`

<Critical Section>

@ Point B:

- 1) Restore Interrupt state & Preemption on local CPU
- 2) Release Lock:
`spin_unlock_irqrestore();`

Kernel Thread C

@ Point A:

- 1) Get Lock:
`spin_lock();`

<Critical Section>

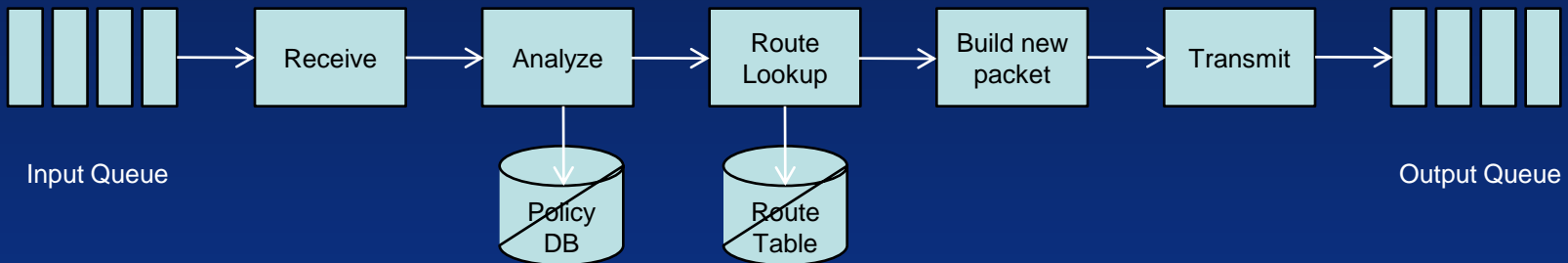
@ Point B:

- 1) Release Lock:
`spin_unlock();`

Real-World Example: Packet Processing for Wireless Base Station

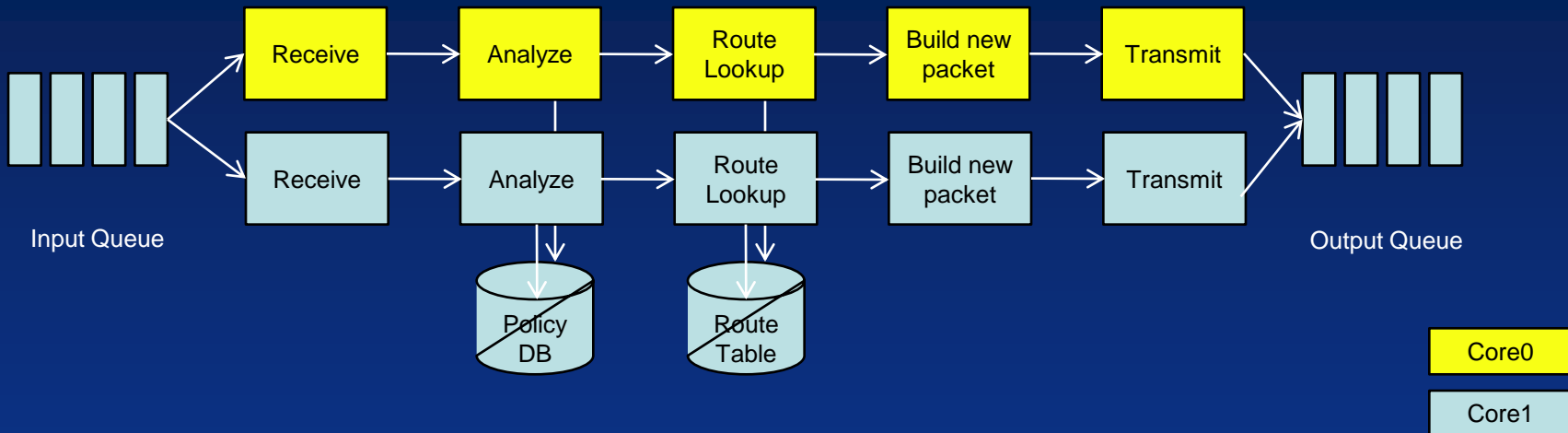
- Application Description
 - “Wireless Base Station”
 - Receives packets
 - Performs some CPU-intensive analysis
 - Policy, Provisioning decisions made
 - Looks up destination routes
 - Reformats and retransmits packets
- The problem
 - Your current packet processing application cannot keep up with traffic
 - You have the opportunity to move it to a multi-core CPU

Packet Processing Example: Single Core, Serial Design



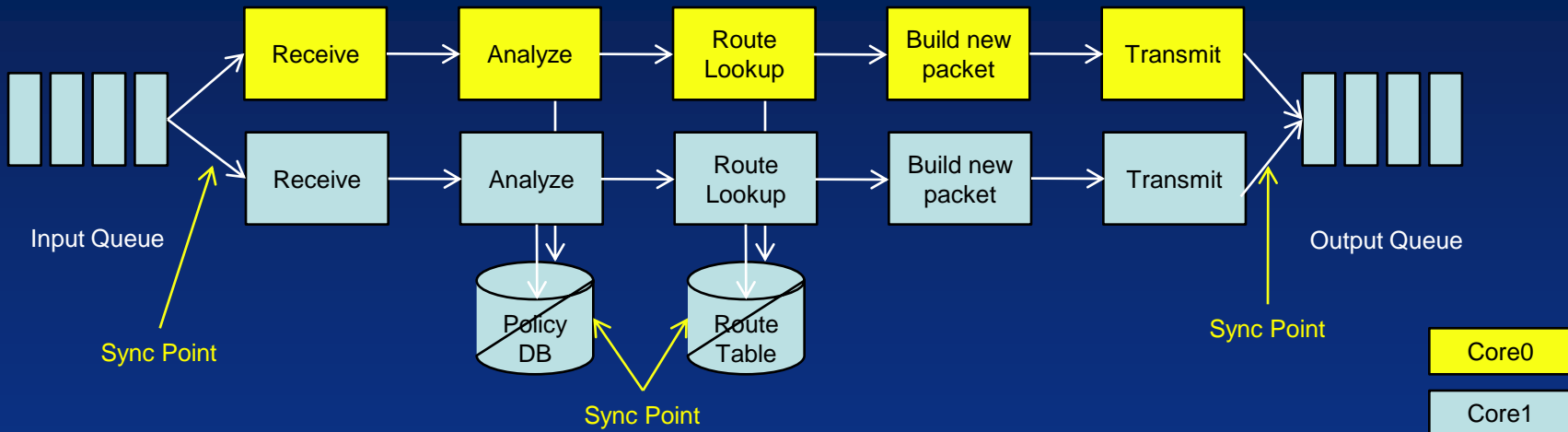
- Application as it stands before multiple cores
- Packet forwarding & analysis, with inspection
 - Policy database guides packet acceptance (security, provisioning)
 - Routing table determines packet destination
 - Assume analysis and route lookup are *not* trivial operations

Packet Processing Example: Dual Core, Symmetric



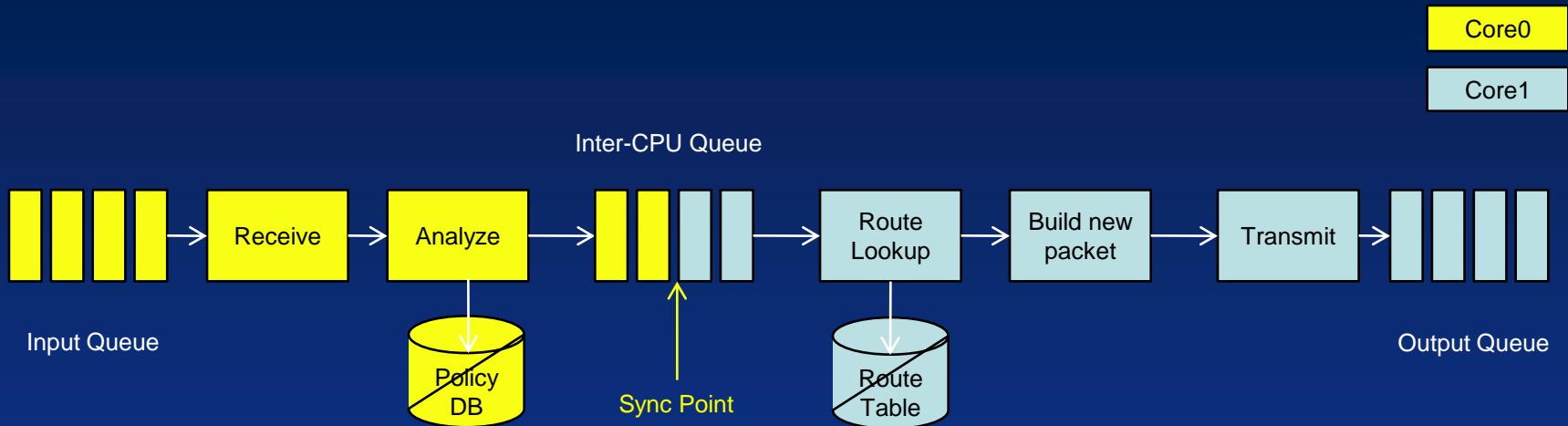
- Simply divide packets among cores
 - Not as easy as it looks!

Packet Processing Example: Dual Core, Symmetric



- Synchronization points needed any time you have two cores competing for one resource.
- Cores must compete to manipulate queues or access shared tables
- Sync points reduce overall performance (Amdahl's Law)
- If implemented in the kernel, *spinlocks* are the best choice

Packet Processing Example: Dual Core, Pipelined



- Pipelining can help (fewer sync points in this case)
- Increases latency
- Reduces performance if processing not evenly matched
 - Slower process limits entire system performance
- This type of application is more common in user space

Synchronization in Linux: Cross-cutting Issues

Major Implications for Scaling & Performance

- Granularity of Locks
 - Course-grained
 - Protect entire subsystems with one lock
 - Easy to implement, greatly reduces parallelism
 - Fine-grained
 - Protect individual structures with locks
 - Much better parallelism, harder to implement and debug

- Lock Ordering Rules (for multiple locks)
 - Acquire, Release in the same order to avoid Deadlocks

Synchronization: Why Do We Care?

- Correctness of Computation
- Avoidance of Race Conditions
- Implications for Real Time Response
- Performance Metrics:
 - Response Time
 - Throughput

Recap: The Whole Picture

- **On-Die Cores:**
 - coherence,
 - cache utilization,
 - power management,
 - memory & i/o configuration
- **System/Platform:**
 - power management,
 - cpu & i/o (storage & network) load balancing
- **Operating System:**
 - synchronization/concurrency control,
 - manageability, security, power management
- **xTCA Application: concurrency control**

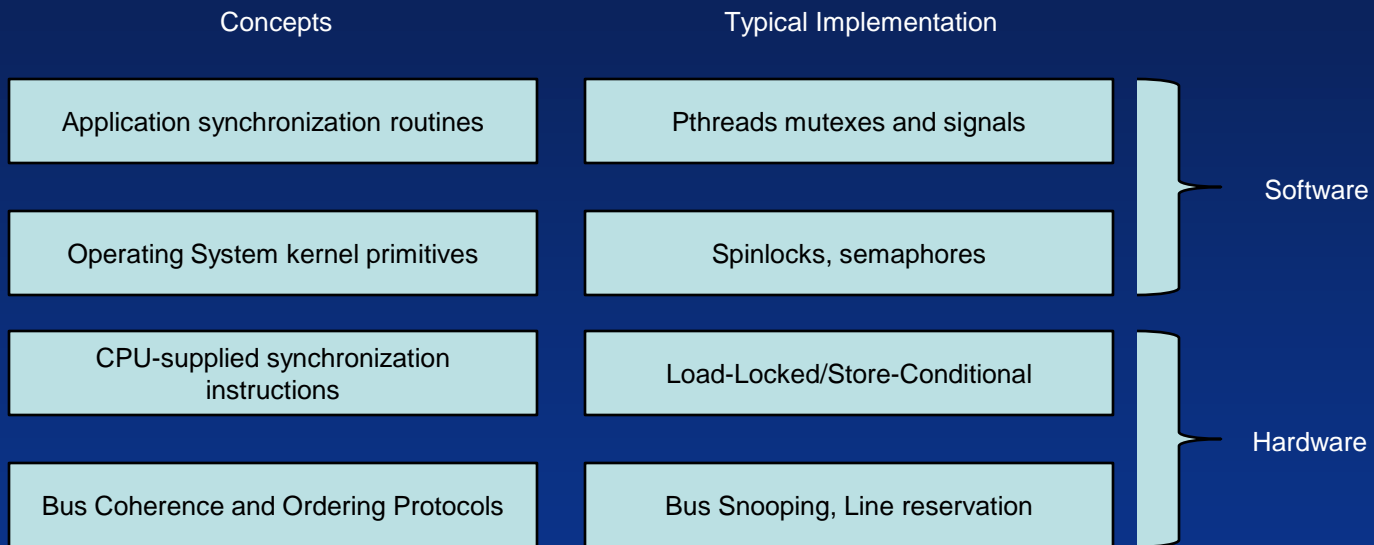
- “Putting it all together is still more Art than Science”

Questions & Answers

- Further Discussions
- Sara Biyabani
sbiyabani@gmail.com

Backup Slides

Synchronization Concepts



- Synchronization concepts are *layered*.
- All eventually resolve to CPU and bus design features