Embedded Systems CONFERENCE SILICON VALLEY

Debugging Multicore & Shared-Memory Embedded Systems

Classes 249 & 269

2007 edition

Jakob Engblom, PhD
Virtutech
jakob@virtutech.com

Scope & Context of This Talk

- Multiprocessor revolution
- Programming multicore
- (In)determinism
- Error sources
- Debugging techniques

Scope and Context of This Talk

- Some material specific to **shared-memory** symmetric **multiprocessors** and **multicore** designs
  - There are lots of problems particular to this
- But most concepts are general to almost any parallel application
  - The problem is really with parallelism and concurrency rather than a particular design choice

Introduction & Background

Multiprocessing: what, why, and when?

The Multicore Revolution is Here!

- The imminent event of parallel computers with many processors taking over from single processors has been declared before...
- **This time it is for real**. Why?

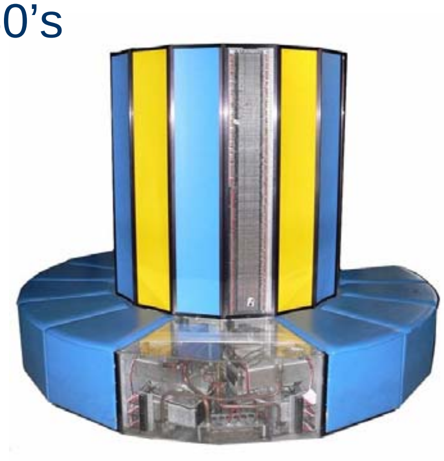
- More instruction-level parallelism hard to find
  - Very complex designs needed for small gain
  - Thread-level parallelism appears live and well
- Clock frequency scaling is slowing drastically
  - Too much power and heat when pushing envelope
- Cannot communicate across chip fast enough
  - Better to design small local units with short paths
- Effective use of billions of transistors
  - Easier to reuse a basic unit many times
- Potential for very easy scaling
  - Just keep adding processors/cores for higher (peak) performance

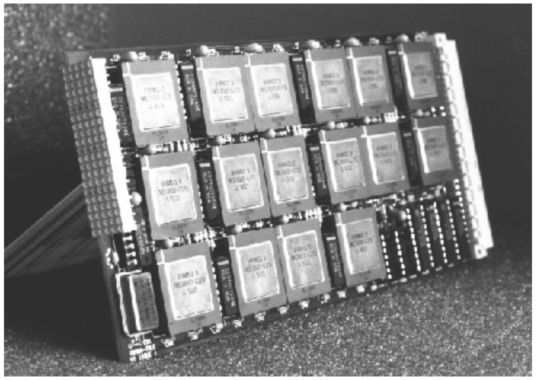
Parallel Processing

- John Hennessy, interviewed in the ACM Queue sees the following eras of computer architecture evolution:
  1. Initial efforts and early designs. 1940. ENIAC, Zuse, Manchester, etc.
  2. Instruction-Set Architecture. Mid-1960s. Starting with the IBM System/360 with multiple machines with the same compatible instruction set
  3. Pipelining and instruction-level parallelism. ~1980. The “RISC revolution”, and the single-core performance work since
  4. Explicitly parallel processing. 2005.
Mainstream going parallel, parallel going mainstream

Multi(core) Processing History

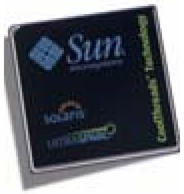
- Multiprocessors have been around since the 1950’s
  - 1959: Burroughs D825,
  - 1960: Univac LARC,
  - 1965: Univac 1108A, IBM 360/65,
  - 1967: CDC6500,
  - 1982: Cray X-MP
  - 1984: Transputer T414





Multi(core) Processing History

- Multicore is more recent
  - 1995: TI C80: video processor: RISC + 4xDSP on a chip
  - 1999: Sun MAJC (2)
  - 2001: IBM Power4 (2): first non-embedded multicore in production
  - 2002: TI OMAP 5470: (ARM + DSP)
  - 2004: ARM11 MPCore (4)
  - 2005: Sun UltraSparc T1 (8x4), AMD Athlon64 (2), IBM XBox 360 CPU (3x2)
  - 2006: Intel Core Duo (2), Freescale MPC8641D (2), 8572(2), IBM Cell (1x2+8)
  - 2007: Intel Core 2 Quad (4)



Multiprocessing is Here

- Multiprocessor and multicore systems are the future
- The only option for maximum performance
- Some current examples:

Vendor	Chip	#Cores	Arch	AMP	SMP
ARM	ARM11 MPCore	4	ARMv6	X	X
Cavium	Octeon CN38	16	MIPS64		X
Freescale	MPC8641D	2	PPC	X	X
IBM	970MP	2	PPC64		X
IBM	Cell	9	PPC64,DSP	X	
Raza	XLR 7-series	8	MIPS64		X
PA Semi	PA6T custom	8	PPC		X
TI	OMAP2	3	ARM,C55,IVA	X	

Manycore

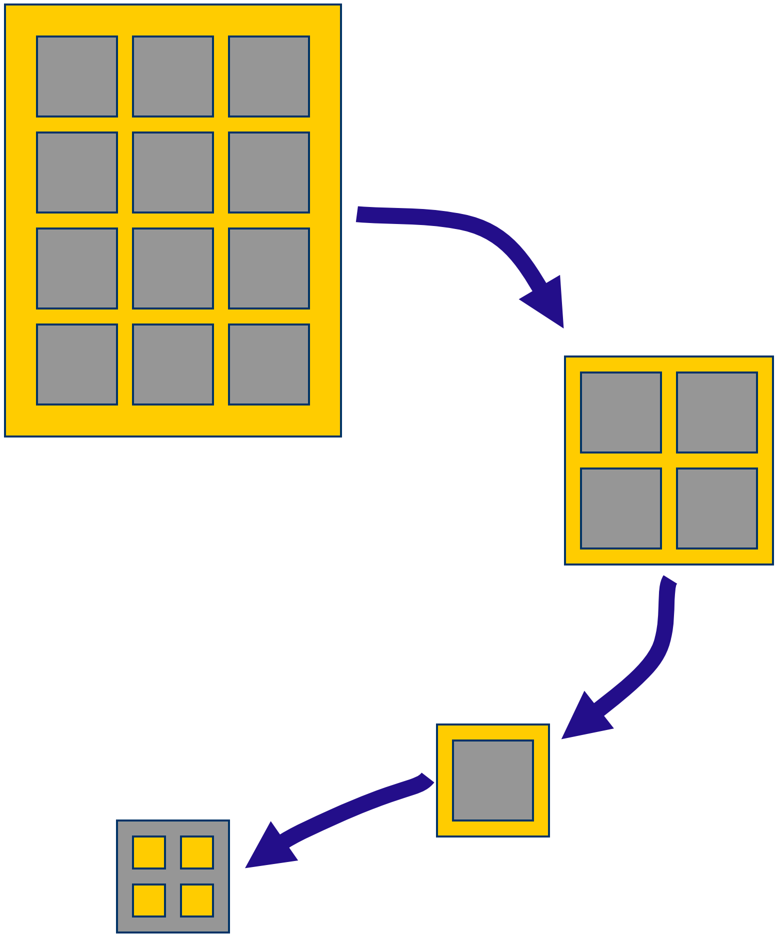
- In specialized naturally parallel domains, there are already many “manycore” chips
  - Manycore is an emerging term for 10+ cores
  - Goal: extreme performance/power, performance/chip

Vendor	Chip	#Cores	Arch	GOps
Cisco	Metro	188	Tensilica	50 @ 35 W
PicoChip	PC102	344	Heterogeneous	41.6 @ 160 MHz
Ambric	AM2045	360	Homogeneous	60 @ 333 MHz
ClearSpeed	CS301	64	Homogeneous	25.6 @ 2.5 W

See Asanovic et al. The Landscape of Parallel Computing Research: A View From Berkeley, Dec 2006

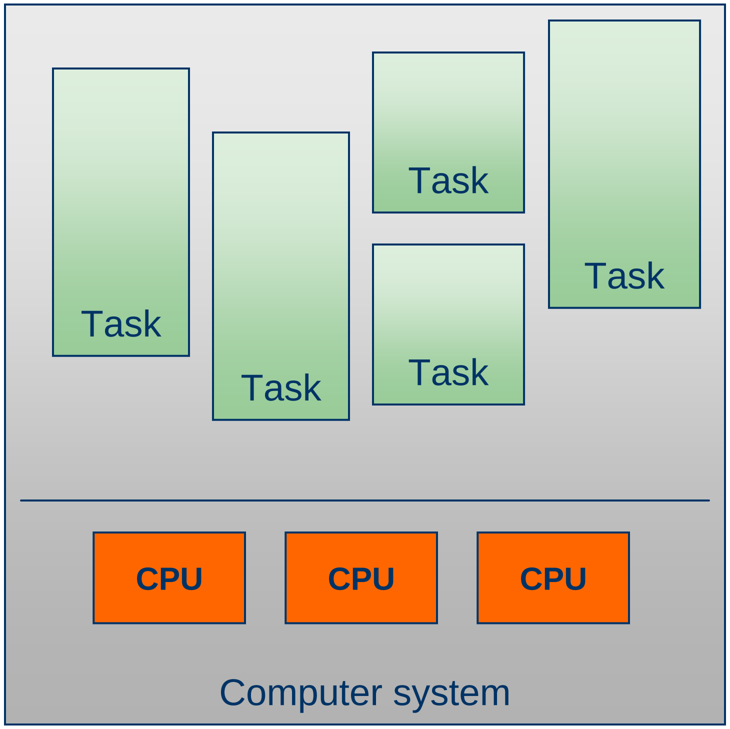
The Road Here

- One processor used to require several chips
- Then one processor could fit on one chip
- Now many processors fit on a single chip
  - This changes everything, since single processors are no longer the default



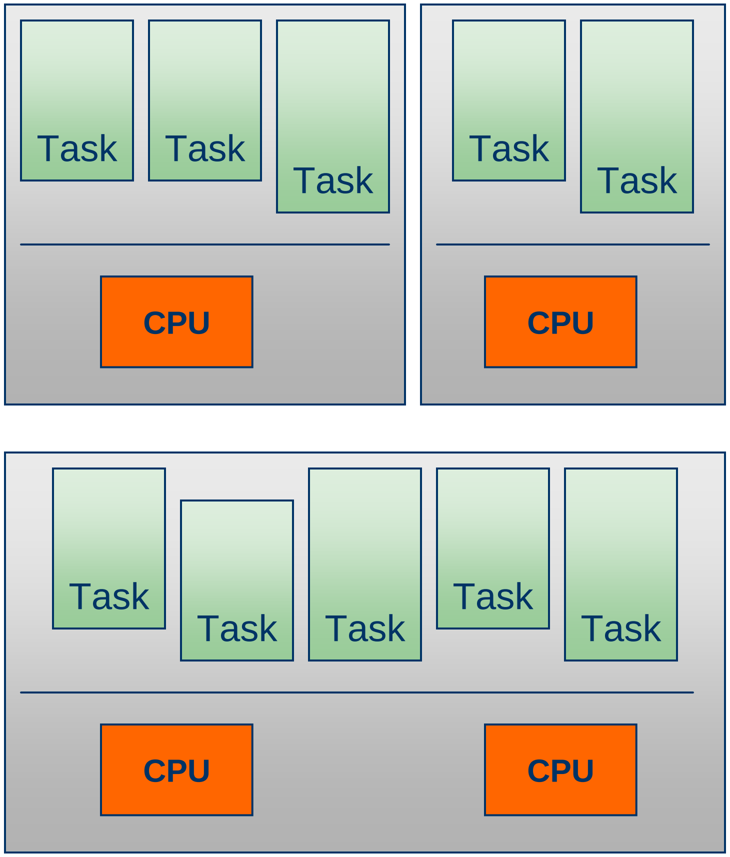
Vocabulary in the Multi Era

- **Multitasking**: multiple tasks running on a single computer
- **Multiprocessor**: multiple processors used to build a single computer system



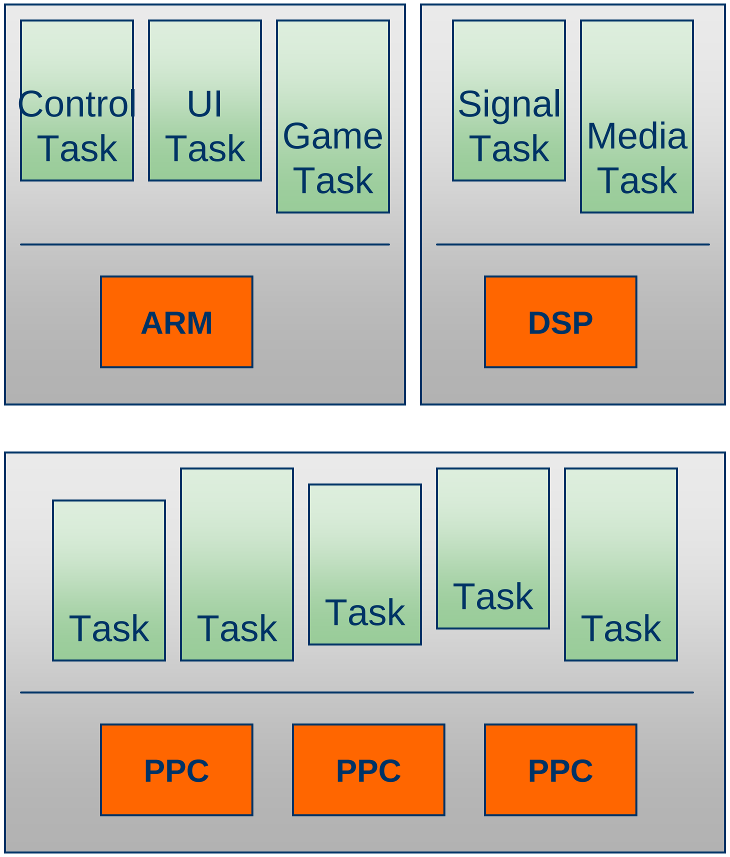
Vocabulary in the Multi Era

- **AMP, Assymetric MP**: Each processor has local memory, tasks statically allocated to one processor
- **SMP**, **Shared-Memory MP**: Processors share memory, tasks dynamically scheduled to any processor



Vocabulary in the Multi Era

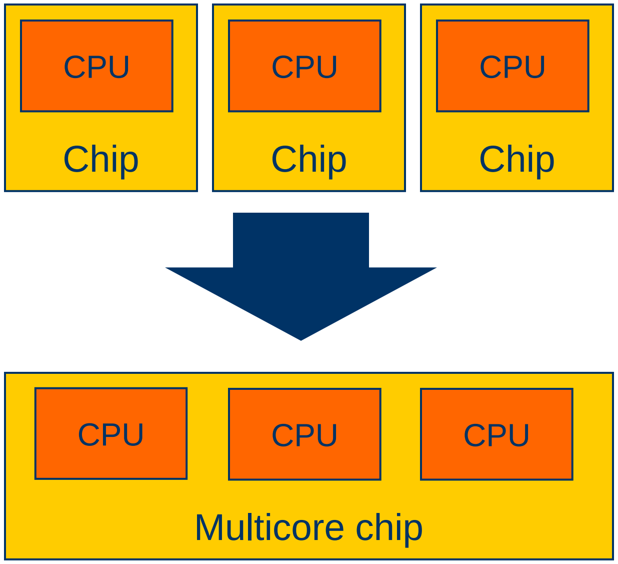
- **Heterogeneous:** Specialization among processors. Often different instruction sets. Usually AMP design.
- **Homogeneous**: all processors have the same instruction set, can run any task, usually SMP design.



Vocabulary in the Multi Era

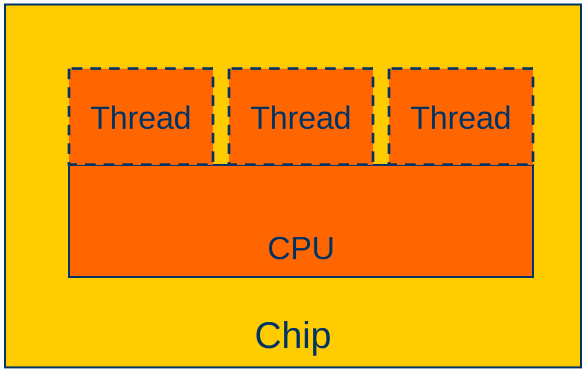
- **Multicore**: more than one processor on a single chip

- **CMP, Chip MultiProcessor**: Shared-memory multiprocessor on a single chip



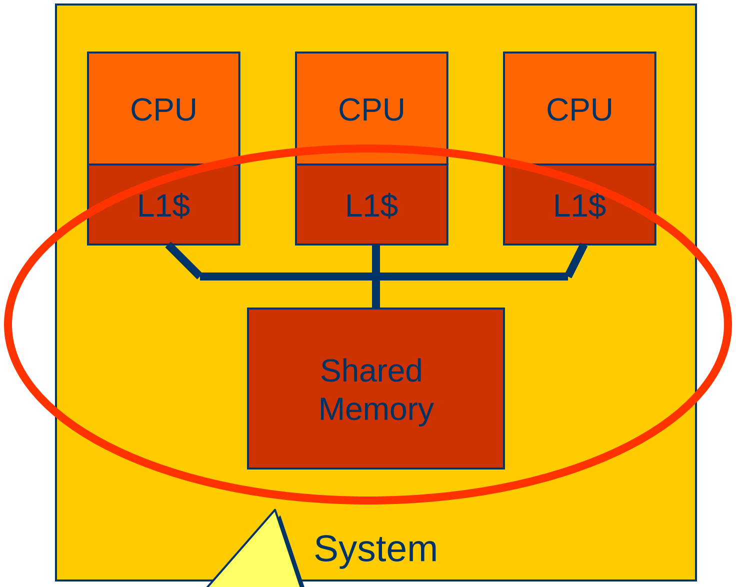
Vocabulary in the Multi Era

- **MT, Multithreading**: one processor appears as multiple thread. The threads share resources, not as powerful as multiple full processors. Very efficient for certain types of workloads



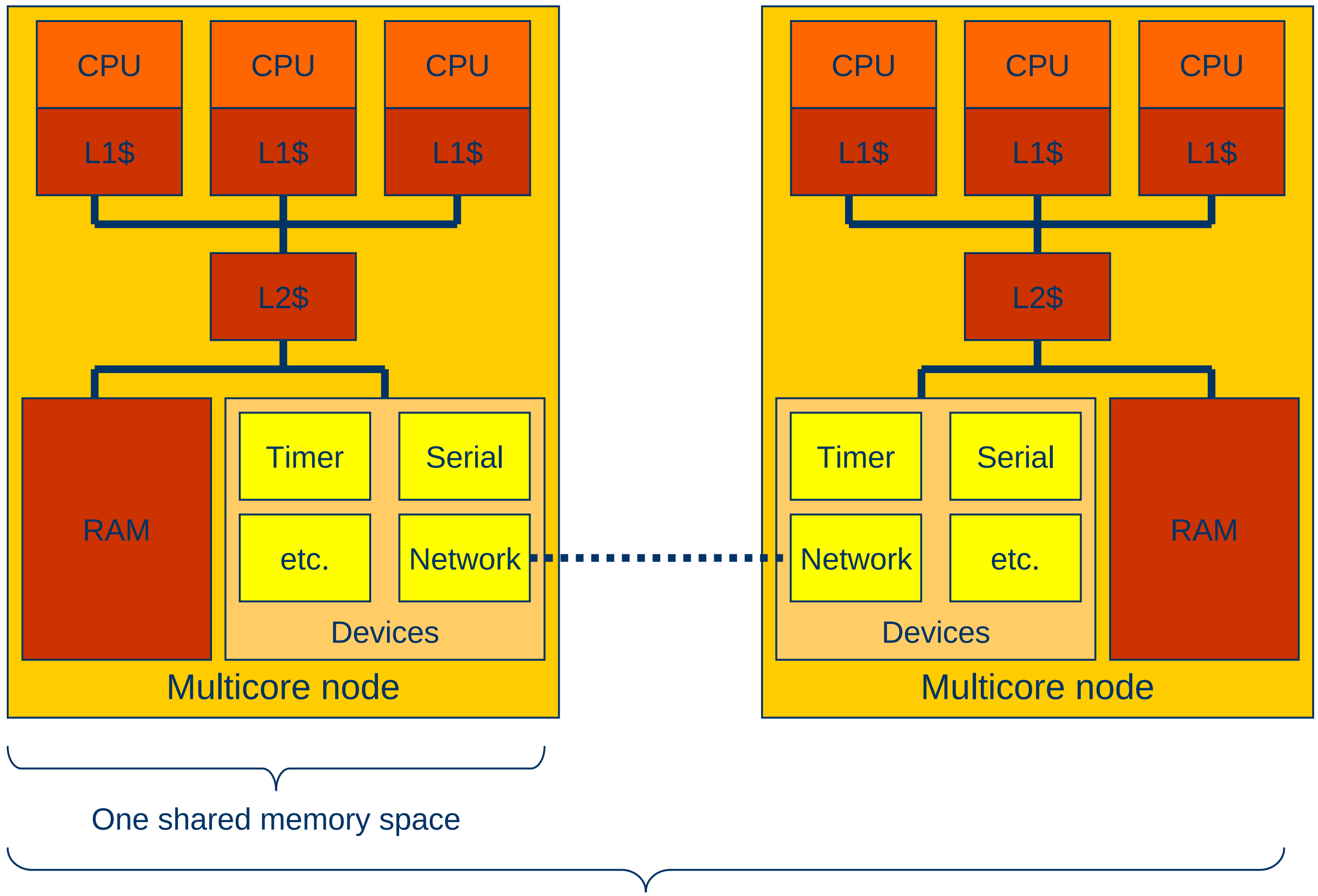
Vocabulary in the Multi Era

- **Cache coherency**:
  - Fundamental technology for shared memory
  - Local caches for each processor in the system
  - Multiple copies of shared data can be present in caches
  - To maintain correct function, caches have to be **coherent**
    - When one processor changes shared data, no other cache will contain old data (eventually)



You want data to be coherent across all caches and the shared memory in a system

Future Embedded Systems



CPU CPU CPU

L1$ L1$ L1$

L2$

RAM

Timer Serial

etc. Network

Devices

Multicore node

CPU CPU CPU

L1$ L1$ L1$

L2$

Timer Serial

Network etc.

Devices

RAM

Multicore node

One shared memory space

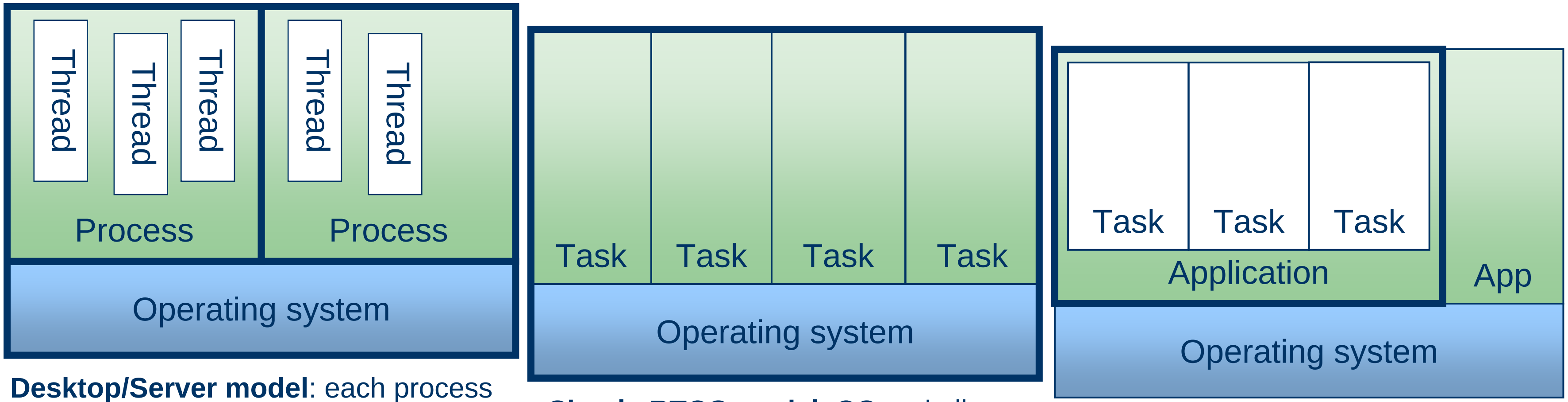
Network with local memory in each node

Programming Models

The Software becomes the Problem

- Parallelism required to gain performance
  - Parallel hardware is “easy” to design
  - Parallel software is (very) hard to write
- Fundamentally hard to grasp true concurrency
  - Especially in complex software environments
- Existing software assumes single-processor
  - Might break in new and interesting ways
  - Multi*tasking* no guarantee to run on multi*processor*

Process, Thread, Task



Thread Thread Thread
Process

Thread Thread
Process

Operating system

Task Task Task Task

Operating system

Task Task Task
Application

App

Operating system

Desktop/Server model: each process in its own memory space, several threads in each process with access to the same memory. Memory protected between processes.

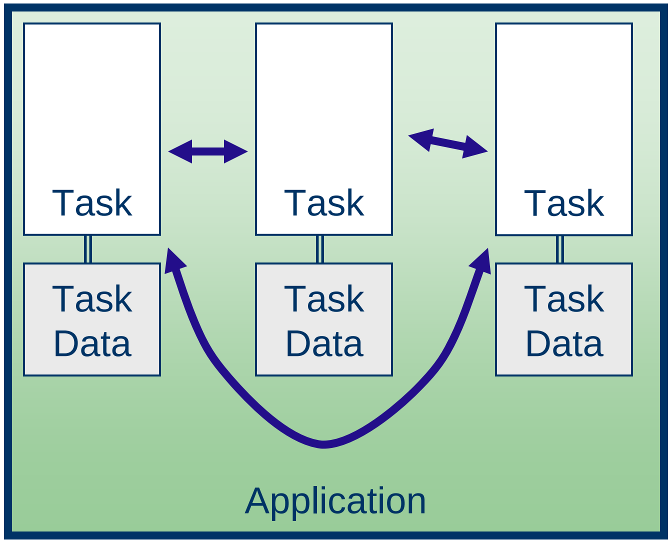
Simple RTOS model: OS and all tasks share the same memory space, all memory accessible to all

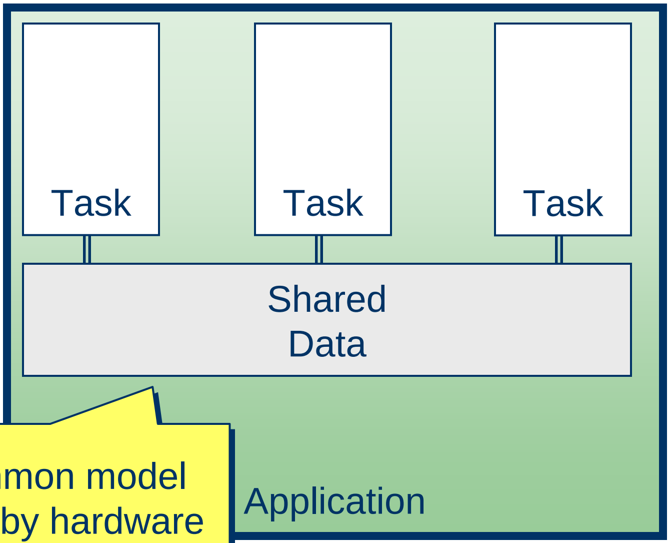
Generic model: a number of tasks share some memory in order to implement an application

- This talk will use “task” for any software thread of control

Message-Passing & Shared-Memory

- Local memory for each task, explicit messages for communication
- All tasks can access the same memory, communication is implicit





Most common model presented by hardware and operating systems

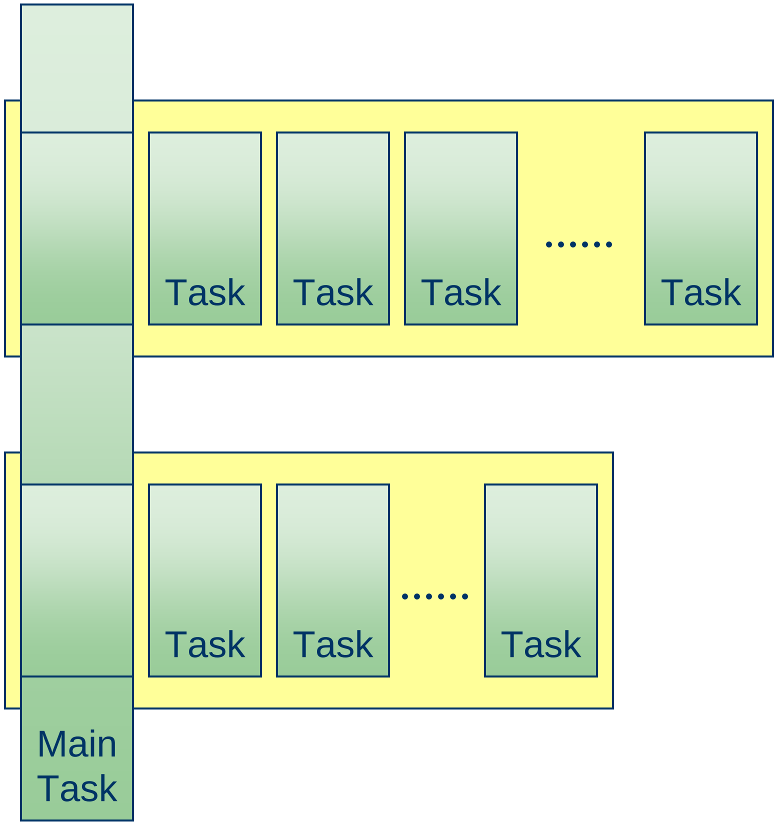
Programming Parallel Machines

- Synchronize & coordinate execution
- Communicate data & status between tasks
- Ensure parallelism-safe access to shared data

- Components of the shared-memory solution:
  - All tasks see the same memory
  - Locks to protect shared data accesses
  - Synchronization primitives to coordinate execution

Success: Classic Supercomputing

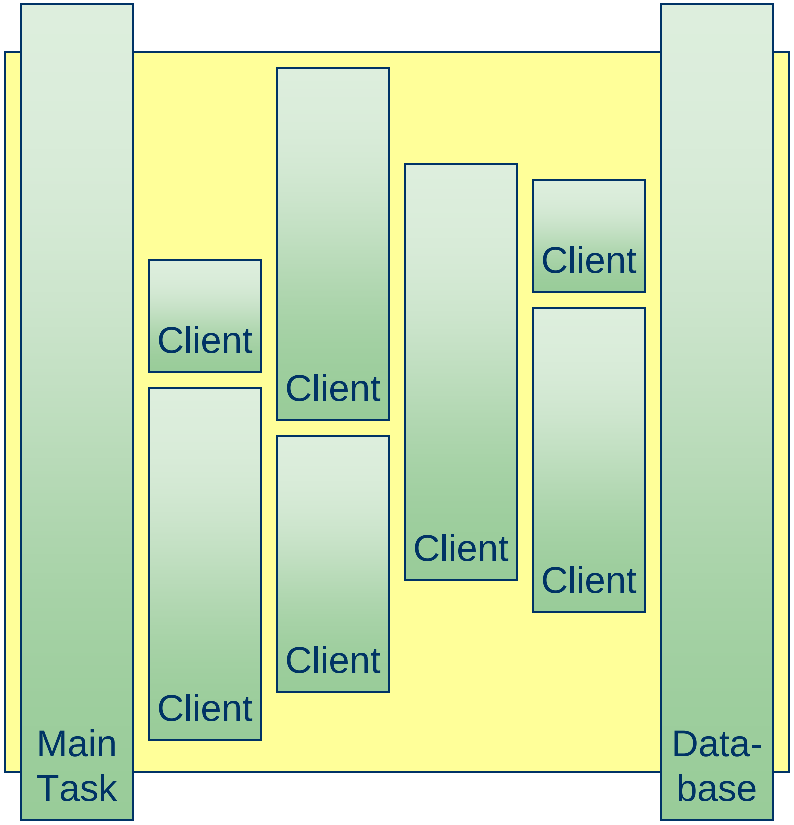
- Regular programs
- Parallelized loops + serial sections
- Data dependencies between tasks
- Very high scalability, 1000s of processors
- FORTRAN, OpenMP, pthreads, MPI



Success: Servers

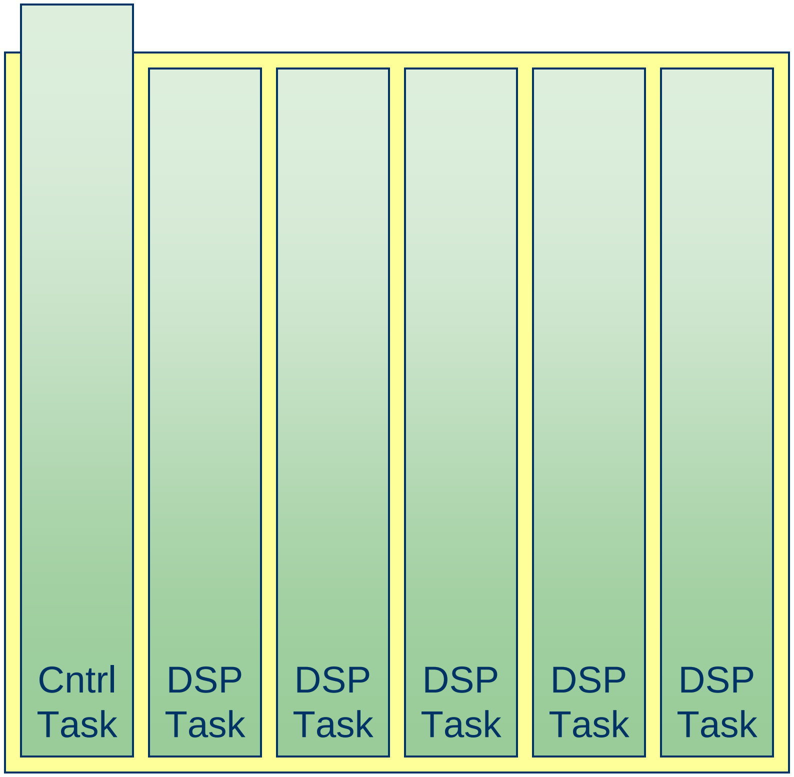
- Natural parallelism
- Irregular length of parallel code, dynamic creation
- Master task coordinates
- Slave tasks for each connection client connection
- Scales very well – using a strong database for the common data

- C, C++, OpenMP, OS API, MPI, pthreads, ...



Success: Signal Processing

- “Embarrassing” natural parallelism
  - No shared data
  - No communication
  - No synchronization
- Single controller CPU
- Parallelizes to 1000s of tasks and processors
- Good fit for AMP



Programming model: Posix Threads

- Standard API
- Explicit operations
- Strong programmer control, arbitrary work in each thread
- Create & manipulate
  - Locks
  - Mutexes
  - Threads
  - etc.

```
main() {
     ...
     pthread_t p_threads[MAX_THREADS];
     pthread_attr_t attr;
     pthread_attr_init (&attr);
     for (i=0; i< num_threads; i++) {
          hits[i] = i;
          pthread_create(&p_threads[i], &attr,
            compute_pi,
               (void *) &hits[i]);
     }
     for (i=0; i< num_threads; i++) {
          pthread_join(p_threads[i], NULL);
          total_hits += hits[i];
     }
     ...
```

Programming model: OpenMP

- Compiler directives
- Special support in the compiler
- Focus on loop-level parallel execution
- Generates calls to threading libraries
- Popular in high-end embedded

```
#pragma omp parallel private(nthreads, tid)
{
 tid = omp_get_thread_num();
 printf("Hello World from thread = %d\n",tid);
 if (tid == 0)
 {
   nthreads = omp_get_num_threads();
   printf("Number of threads: %d\n",nthreads);
 }
}
```

Programming model: MPI

- Message-passing API
  - Explicit messages for communication
  - Explicit distribution of data to each thread for work
  - Shared memory not visible in the programming model
- Best scaling for large systems (1000s of CPUs)
  - Quite hard to program
  - Well-established in HPC

```
main(int argc, char *argv[])
{
    int npes, myrank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("From process %d out of %d,
       Hello World!\n", myrank, npes);
    MPI_Finalize();
}
```

Programming: Thread-oriented

- Language design
  - Threads fundamental unit of program structure
  - Spawn & send & receive
  - Local thread memory
  - Explicit communication
- Designed to scale out and to be distributed
- Control-code oriented, not compute kernels

```
-module(tut18).
-export([start/1,  ping/2, pong/0]).
ping(0, Pong_Node) ->
    {pong, Pong_Node} ! finished,
    io:format("ping finished~n", []);
ping(N, Pong_Node) ->
    {pong, Pong_Node} ! {ping, self()},
    ping(N - 1, Pong_Node).
pong() ->
    receive
        finished ->
            io:format("Pong finished~n", []);
        {ping, Ping_PID} ->
            Ping_PID ! pong,
            pong()
    end.
start(Ping_Node) ->
    register(pong, spawn(tut18, pong, [])),
    spawn(Ping_Node, tut18, ping, [3,
    node()]).
```

source: Erlang tutorial at www.erlang.org

Programming: Performance Libraries

- Vendor-provided function library customized for each machine
  - Optimized code “for free”
  - Tied to particular machines
- Supports computation kernels
  - Arrays of data
  - Function calls to compute
- Supercomputing-style loop-level parallelization
- Limited in available functions

```
int i, large_index;
float a[n], b[n], largest;
large_index = isamax (n, a, l) - 1;
largest = a[large_index];
large_index = isamax (n, b, l) - 1;
if (b[large_index] > largest)
  largest = b[large_index];
```

source: Sun Performance Library documerntation

Programming: Stream Processing

- Idea is simple:
  - Stream data between compute kernels
    - Rather than loading from memory and storing results back
  - Execute all kernels in parallel, keep data flowing
  - Aimed at data-parallelism
- Hip concept currently, especially for massive parallel programming; with many different interpretations

Stream Processing (variant 1)

- Array parallelism
  - Special types and libraries
  - Sequential step-by-step program, each step parallel compute kernel
  - “Better OpenMP”
- Current implementations:
  - Compiles into massively parallel code for DSPs, GPUs, Cell, etc. **Hides details!**
- PeakStream, RapidMind, et al.

```
Arrayf32 SP_lb, SP_hb, SP_frac; {
 Arrayf32 SP_mb; {
  Arrayf32 SP_r; {
   Arrayf32 SP_xf, SP_yf; {
    Arrayf32 SP_xgrid =
      Arrayf32::index(1,nPixels,nPixels) + 1.0f;
    Arrayf32 SP_ygrid =
      Arrayf32::index(0,nPixels,nPixels) + 1.0f;
    SP_xf = (SP_xgrid - xcen) * rxcen;
    SP_yf = (SP_ygrid - ycen) * rycen;
    } // release SP_xgrid, SP_ygrid
   SP_r = SP_xf*cosAng + SP_yf*sinAng;
   } // release SP_xf, SP_yf
   SP_mb = mPoint + SP_r*mPoint;
  } // release SP_r
  SP_lb = floor(SP_mb);
  SP_hb = ceil(SP_mb);
  SP_frac = SP_mb - SP_lb;
  SP_lb = SP_lb - 1;
  SP_hb = SP_hb - 1;
  } // release SP_mb
```

source: PeakStream white papers

Stream Programming (variant 2)

- “Sieve C”
  - More general than array parallelism, can do task parallelism as well
  - Explicit parallel coding
  - “Better OpenMP”
- Smart semantics to simplify programming and debug
  - No side-effects inside block
  - Local memory for each parallel piece
  - Deterministic, serial-equivalent semantics and compute results

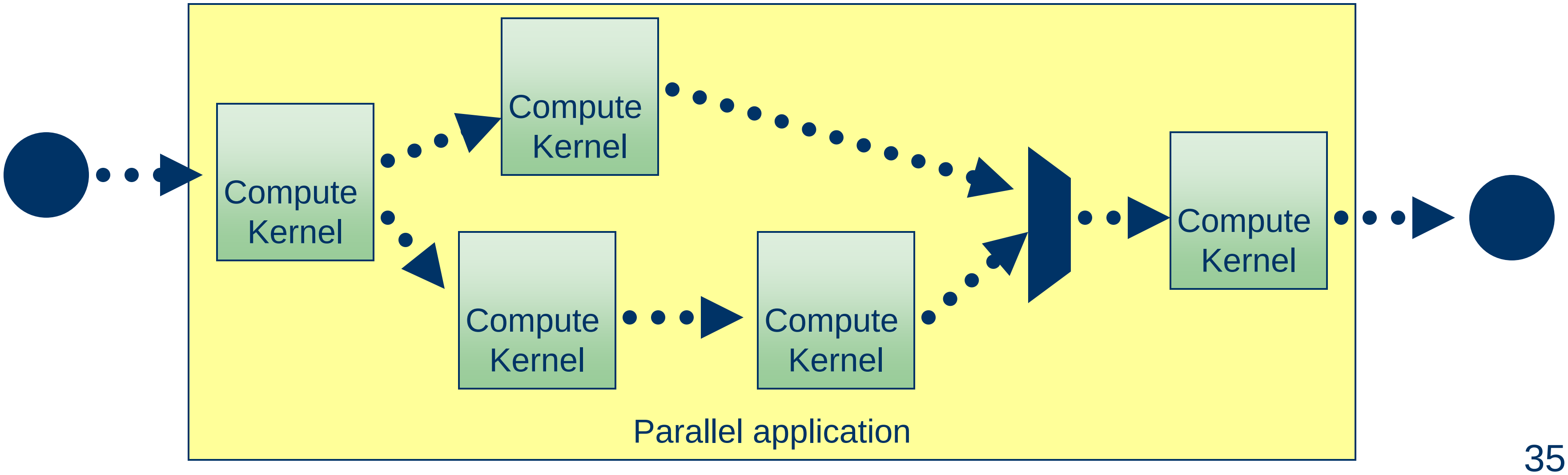
```
sieve {
  for(i = 0; i < MATRIX_SIZE; ++i) {
    for(j = 0; j < MATRIX_SIZE; ++j)  {
      pResult->m[ i ][ j ] =
        sieveCalculateMatrixElement
          ( a, i, b, j );
    }
  }
}    // memory writes are moved to here
```

Main reason that I want to mention this fairly niche product. The design of the parallel language or parallel API can greatly affect the ease of bug finding

source: CodeTalk talk by Andrew Richards, 2006

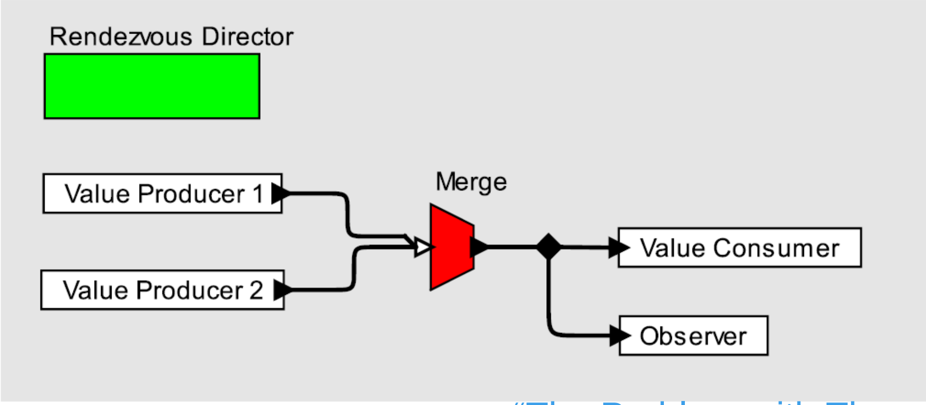
Stream Processing (variant 3)

- Network-style data-flow API
  - Send/receive messages, similar to classic message-passing
  - But with support to scale up to many units, map directly to fast hardware communications channels
  - Example: Multicore Association CAPI



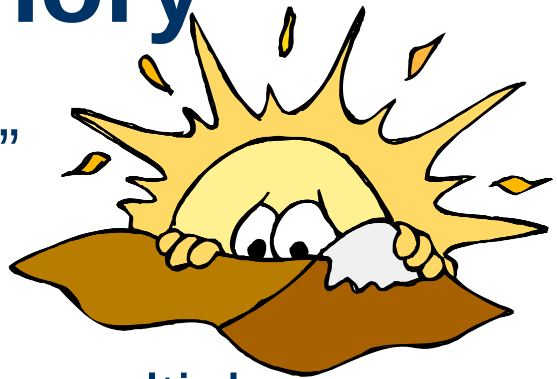
Programming: Coordination Languages

- Separation of concerns: computations vs parallelism
  - Express sequential computations in sequential language like C/C++/Java, familiar to programmers
  - Add concurrency in a separate coordinating layer
- Research approach



source: “The Problem with Threads”, Edward Lee, 2006

Prog: Transactional Memory



- Make the main memory into a "database"
  - With hardware support in the processors
  - Extension of cache coherency systems
- Define atomic transactions encompassing multiple memory accesses
  - Abort or commit as a group
  - Simplifies maintaining a consistent view of state
  - Software has to deal with transaction failures in some way
  - Simplification of shared-memory programming
- Research topic currently, the dust has not settled

Programming Models Summary

- Many different programming languages, tools, methodologies and styles available
- Choice of programming model can have a huge impact on performance, ease of programming, and debuggability
- Current market focus is on this essentially constructive activity: create parallel code
  - ...with less concern for the destructive activity of testing and reconstructive activity of debugging

Determinism

The fundamental issue with parallel programming and debugging

Determinism

- In a perfectly deterministic system, rerunning a program with the same input produces
  - The same output
  - Using the same execution path
  - With the same intermediate states, step-wise computation
- “Input”
  - The state of the system when execution starts
  - Any inputs received during the execution
- The behavior and computation of such a system can be investigated with ease, “classic debugging”

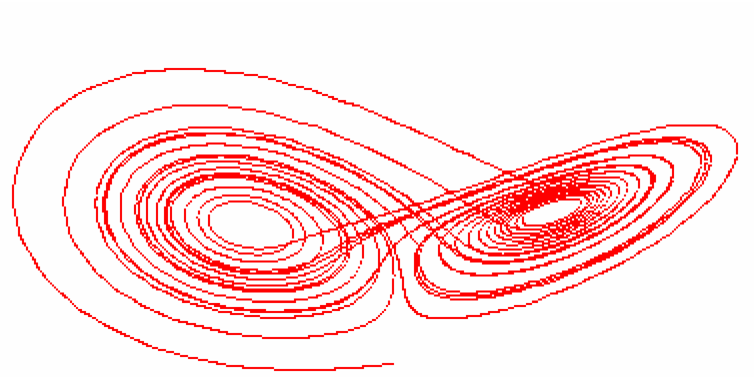
Indeterminism

- An indeterministic program will not behave the same every time it is executed
  - Possibly different output results
  - Different execution path
  - Different intermediate states
  - Much harder to investigate and debug
- Very common phenomenon in practice on multiprocessor computers. **Why**?
  - Chaos theory
  - Emergent behavior

Chaos Theory

- Even the smallest disturbance can cause total divergence of system behavior
  - Mathematically, the system can be deterministic. It is just very sensitive to input value fluctuations
  - Popularized as the “**Butterfly Effect**”

- Lorenz attractor example
  - Jumps between left and right loops seemingly at random
  - Very sensitive to input data



picture from Wikipedia

Emergent Behavior

- Complex behavior arises as many fundamentally simple components are combined
- **Global** behavior of system cannot be predicted or understood from the **local** behavior of its components

Disclaimer: this is my personal intentionally simplifying interpretation of a very complex philosophical theory

- Examples:
  - Weather systems, built up from the atoms of the atmosphere following simple laws of nature
  - Termite mounds resulting from the local activity of thousands of termites
  - Software system instability and unpredictability from layers of abstraction and middleware and drivers and patches

Determinism & Computers

- A computer is a man-made machine
  - There is no intentional indeterminism in the design
  - It consists of a large number of deterministic component designs
    - Processor pipeline, Branch prediction, DRAM access, cache replacement policies, cache coherence protocols, bus arbitration, etc.
  - But in practice, the combined, emergent, behavior is not possible to predict from the pieces. New phenomena arise as we combine components.

Determinism & Multiprocessors

- Each run of a multiprocessor program tends to behave differently
  - Maybe not the end result computed by the program, but certainly the execution path and system intermediate states leading there
- Differences can be caused by very small-scale variations that happen all the time in a multiprocessor:
  - Number of times a spin-lock loop is executed
  - Cache hits or misses for memory accesses
  - Time to get data from main memory for a read (arbitration collisions, DRAM refresh, etc.)

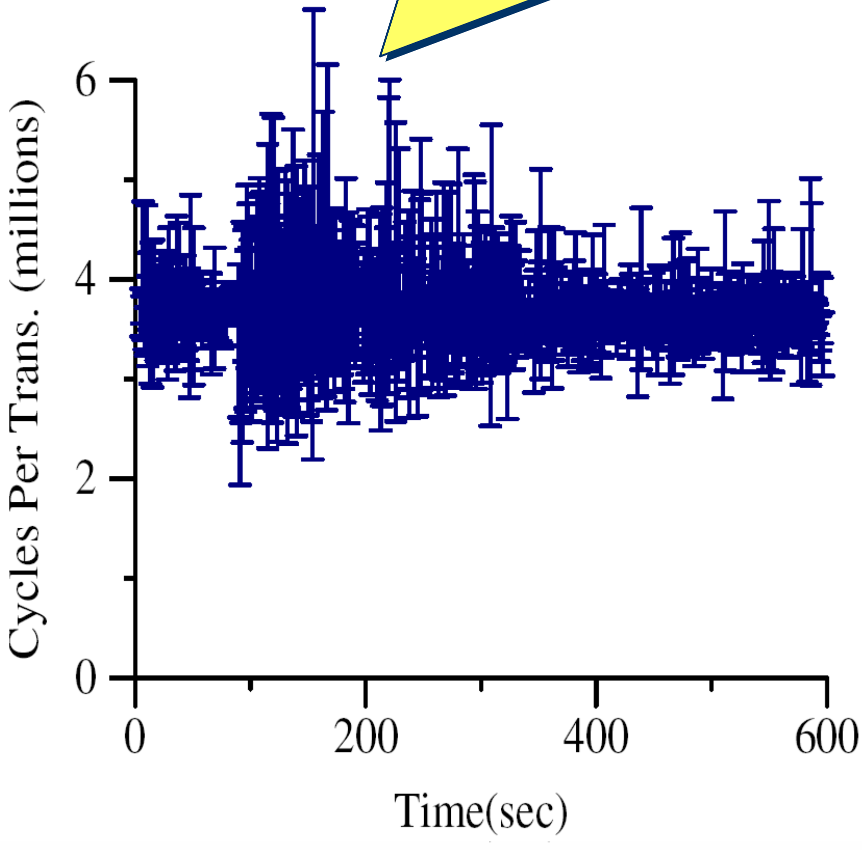
Determinism & Multiprocessors

- Fundamentally, multiprocessor computer systems exhibit **chaotic behavior**
- A concrete example documented in literature:
  - A simple delay of a single instruction by a few clock cycles can cause a task to be interrupted by the OS scheduler at a slightly different point in the code. With different intermediate results stored in variables, leading other tasks to take different paths... and from there it snowballs. It really is the butterfly effect!

Variability Example

- The diagram:
  - Average time per transaction in the OLTP benchmark
  - Measured on a Sun multiprocessor
  - Minimal background load
  - Average over one second, which correspond to more than 350 transactions
  - Source: *Alameldeen and Wood: “Variability in Architectural Simulations of Multi-Threaded Workloads”, HPCA 2003.*

And here is the result when five identical runs are started on a fresh machine. Still huge variation across “identical” runs



Macro-Scale (In)Determinism

- Note that reasons for indeterministic behavior on multiprocessor systems can be found in the macro scale as well
  - (coming up)
- The micro-scale events discussed above just makes macro scale variation more likely, and fundamentally unavoidable in any dynamic system

(In)Determinism in the Macro Scale

- Background noise:
  - With multiple other tasks running, the scheduling of the set of tasks for a particular program is very likely to be different each time it is run
- Asynchronous inputs:
  - The precise timing of inputs from the outside world relative to a particular program will always vary
- Accumulation of state:
  - Over time, a system accumulates state, and this is likely to be different for any two program runs

Macro-Scale Determinism

- We can bring control, determinism, and order back to our programs at the macro scale
  - We have to make programs robust and insensitive to micro-scale variations and buffeting from other parts of the system
- This happens in the real world all the time
  - Bridges remain bridges, even when they sway
  - Running water on a bathroom floor ends up at the lowest point... even if the flow there can vary

Macro-Scale Determinism in Software

- Synchronize
  - Your program dictates the order, not the computer
  - Any important ordering has to be specified
- Discretize
  - Structure computations into “atomic” units
  - Generate output for units of work, not for individual operations
- Impose your own ordering
  - Do not let the system determine your execution order
  - For example, traversal of a set should follow an order given explicitly in your program

Is this Insanity?

- *"Insanity is doing the same thing over and over again and expecting a different result"*
  - *Folk definition of insanity*

- That is exactly what multiprocessor programming is all about: doing the same thing over again *should* give a different result

What Goes Wrong?

The real bugs that bite us

True Concurrency = Problems

- Fundamentally new things happen
  - Some phenomena cannot occur on a single processor running multiple threads
- More stress for multitasking programs
  - Exposes latent problems in code
  - Multitasking != multiprocessor-ready
  - Even supposedly well-tested code can break
  - Bad things happen more often and more likely

(Missing) Reentrancy

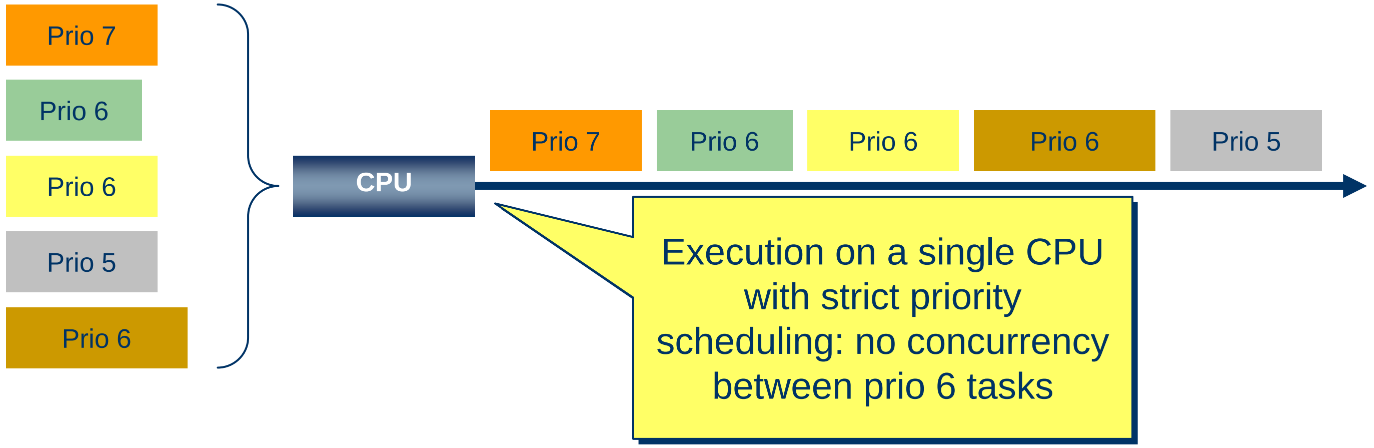
- Code shared between tasks has to be reentrant
  - No global/static variables used for local state
  - Do not assume single thread of control

- True concurrency = much higher chance of parallel execution of code
  - Problem also occurs in multitasking
  - But is much less likely to trigger
    - See example later in this talk on race conditions

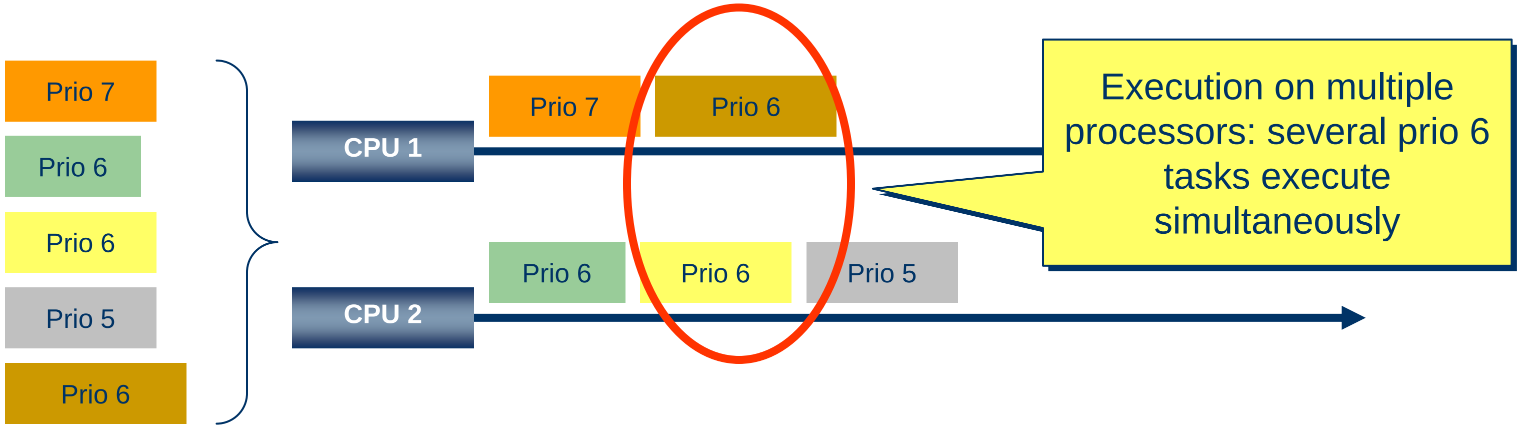
Priorities are not Synchronization

- Strict priority scheduling on single processor
  - Tasks of same priority will be run sequentially
  - No concurrent execution = no locking needed
  - Property of application software design
- Multiple processors
  - Tasks of same priority will run in parallel
  - Locking & synchronization needed in applications

Priorities are not Synchronization



Execution on a single CPU with strict priority scheduling: no concurrency between prio 6 tasks



Execution on multiple processors: several prio 6 tasks execute simultaneously

Disabling Interrupts is not Locking

- Single processor: DI = cannot be interrupted
  - Guaranteed exclusive access to whole machine
  - Cheap mechanism, used in many drivers & kernels
- Multiprocessor: DI = stop interrupts on one core
  - Other cores keep running
  - Shared data can be modified from the outside

Disabling Interrupts is not Locking

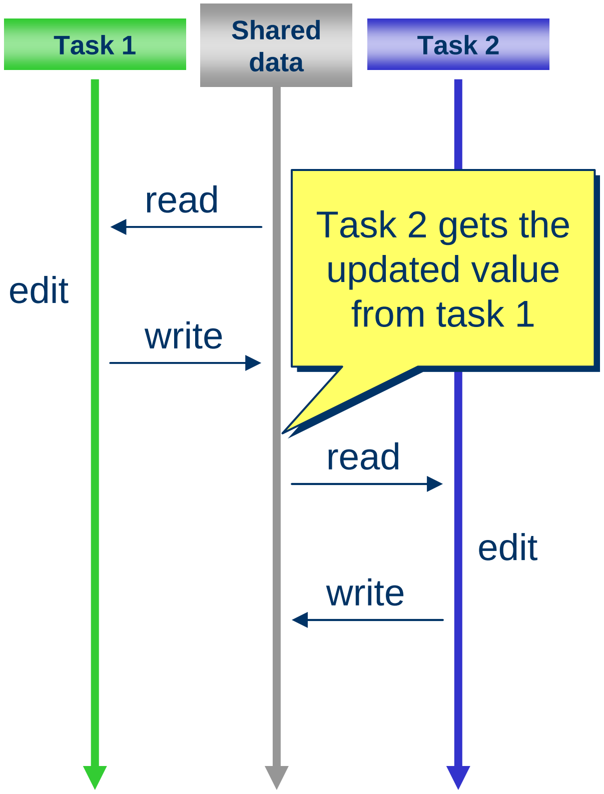
- Big issue for low-level code, drivers, and OS
- Note that interrupts is typically how the different cores in a multiprocessor communicate
  - The interrupt controller lets the OS code locally on each processor communicate with the others
  - Disabling interrupts for a long time might break the operating system
- Need to replace DI/EI with proper locks

Race Condition

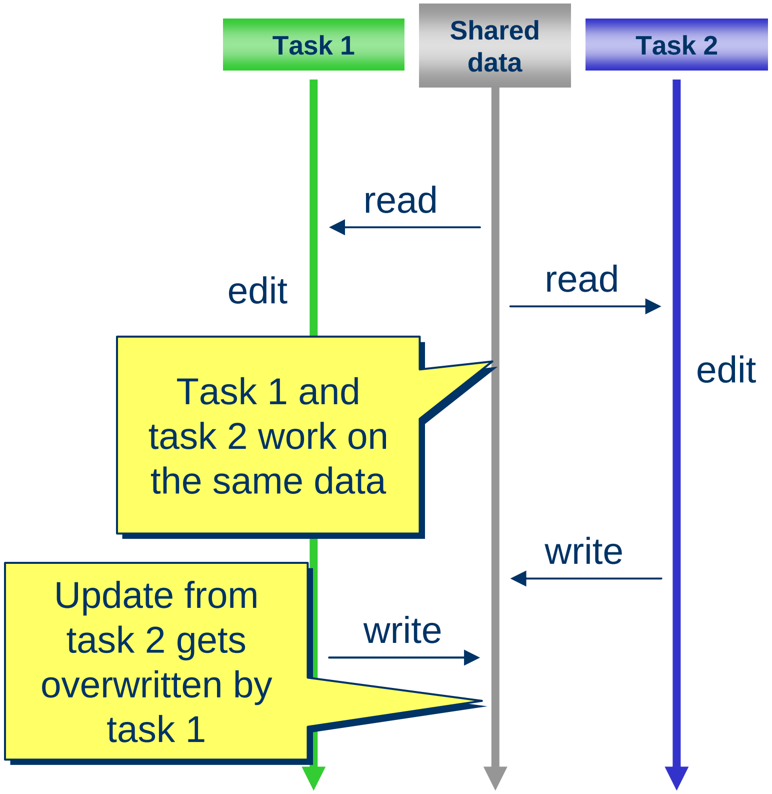
- Tasks “race” to a common point
  - Result depends on who gets there first
  - Occurs due to insufficient synchronization
- Present with regular multitasking, but much more severe in multiprocessing
- Solution: protect all shared data with locks, synchronize to ensure expected order of events

Race Condition: Shared Memory

- Correct behavior

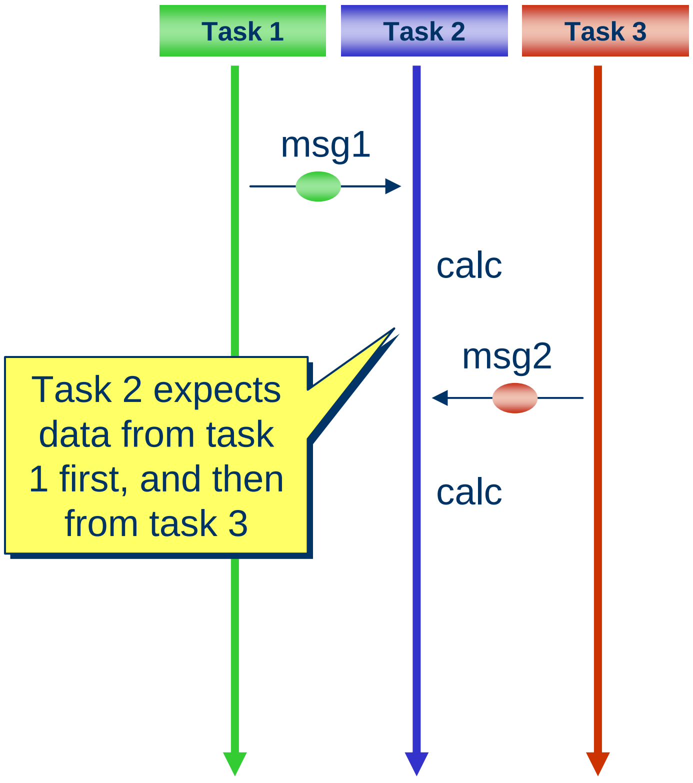


- Incorrect behavior


Task 1
Shared data
Task 2
read
edit
read
Task 1 and task 2 work on the same data
edit
write
Update from task 2 gets overwritten by task 1
write

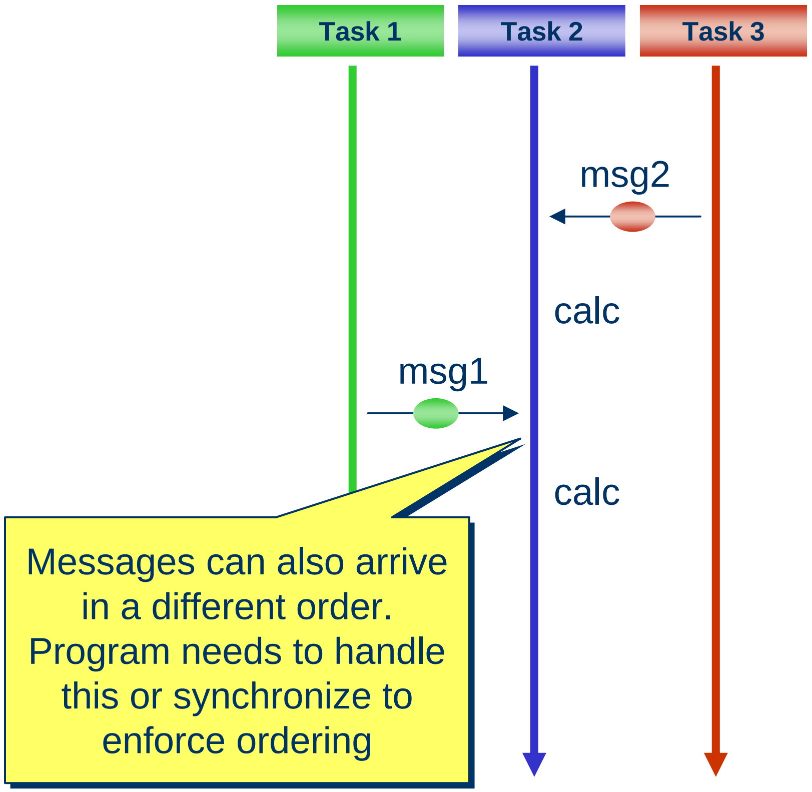
Race Condition: Messages

- Expected sequence



Task 2 expects data from task 1 first, and then from task 3

- Incorrect sequence



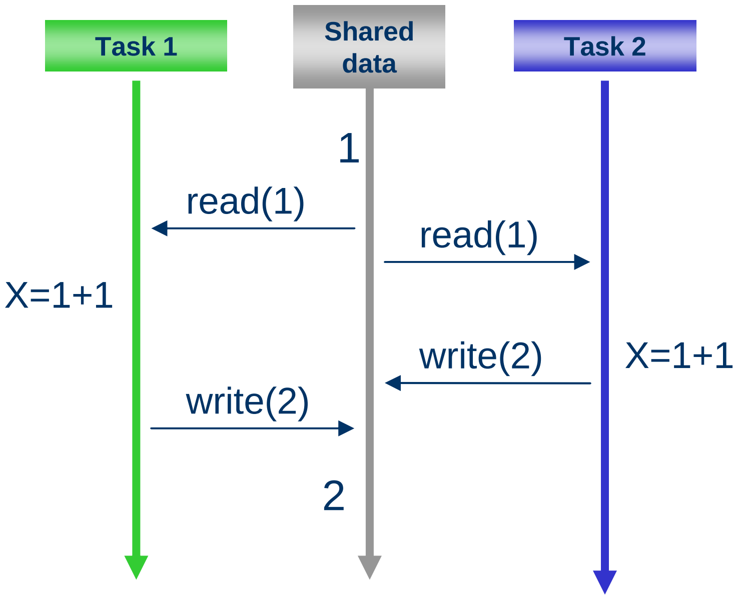
Messages can also arrive in a different order. Program needs to handle this or synchronize to enforce ordering

Race Condition Example

- Test program:
  - Two parallel threads
  - **`Loop 100000 times:`**

```
Read x
Inc x
Write x
Wait...
```

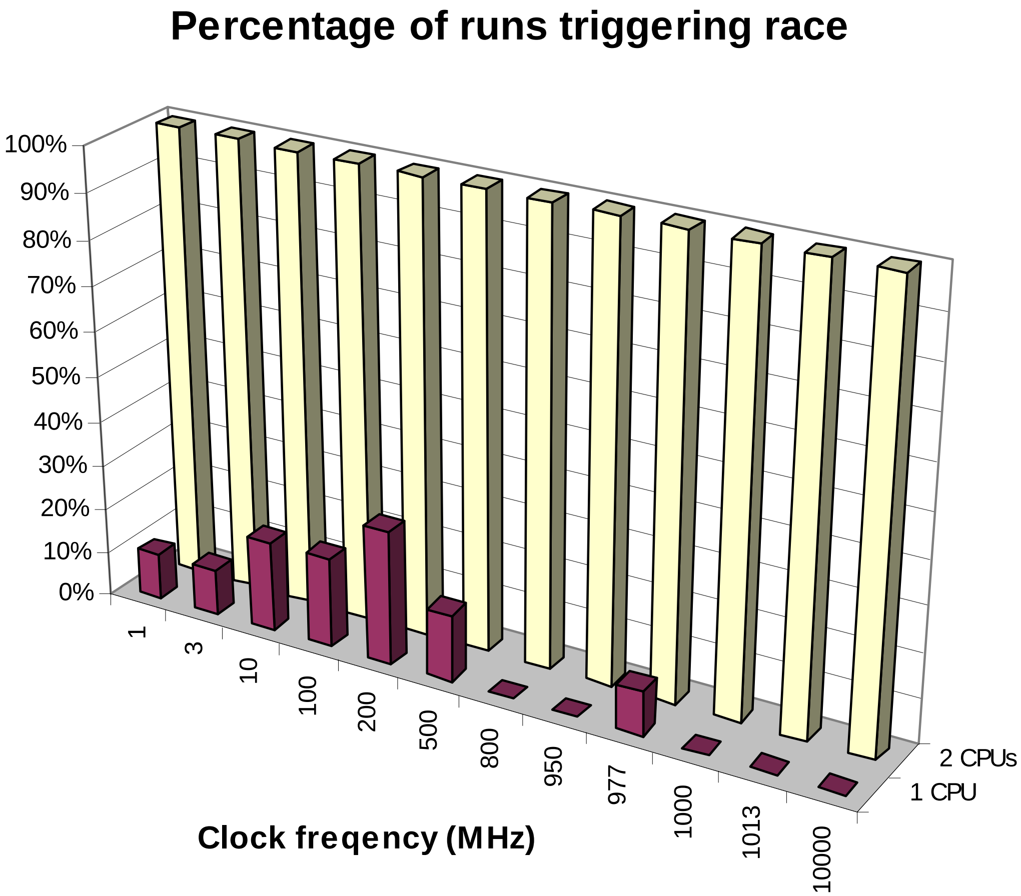
- Intentionally bad: not designed for concurrency, easily hit by race
- Observable error: final value of x less than 200000
- Will trigger **very** easily in a multiprocessor setting
- But less easily with plain multitasking on single pro



Thanks to *Lars Albertsson at SiCS*

Race Condition Example

- Simulated single-CPU and dual-CPU MPC8641
- Different clock frequencies
- Test program run 20 times on each case
- Count percentage of runs triggering the bug
- Results:
  - Bug **always** triggers in dual-CPU mode
  - Triggers around 10% in single-CPU mode
  - Higher clock = lower chance to trigger

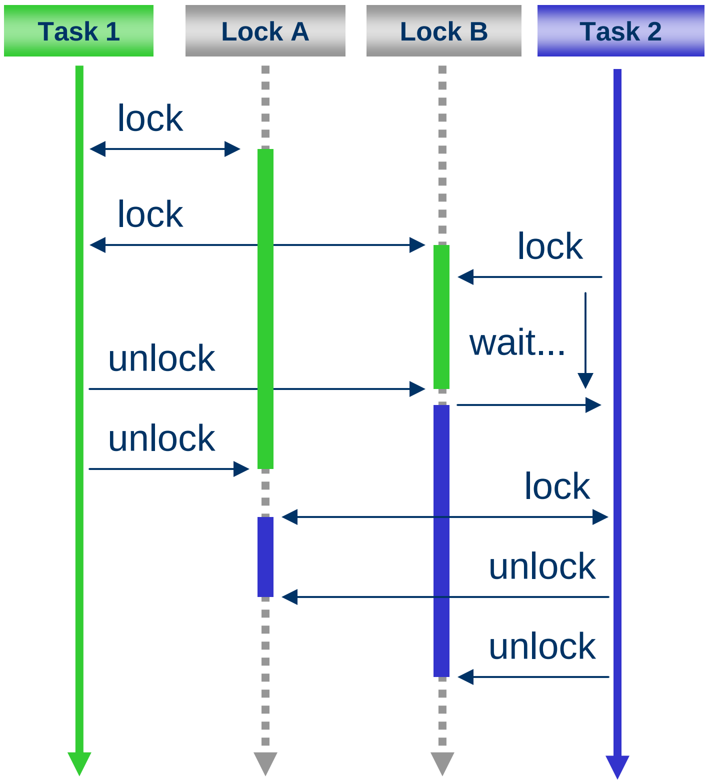


Deadlocks

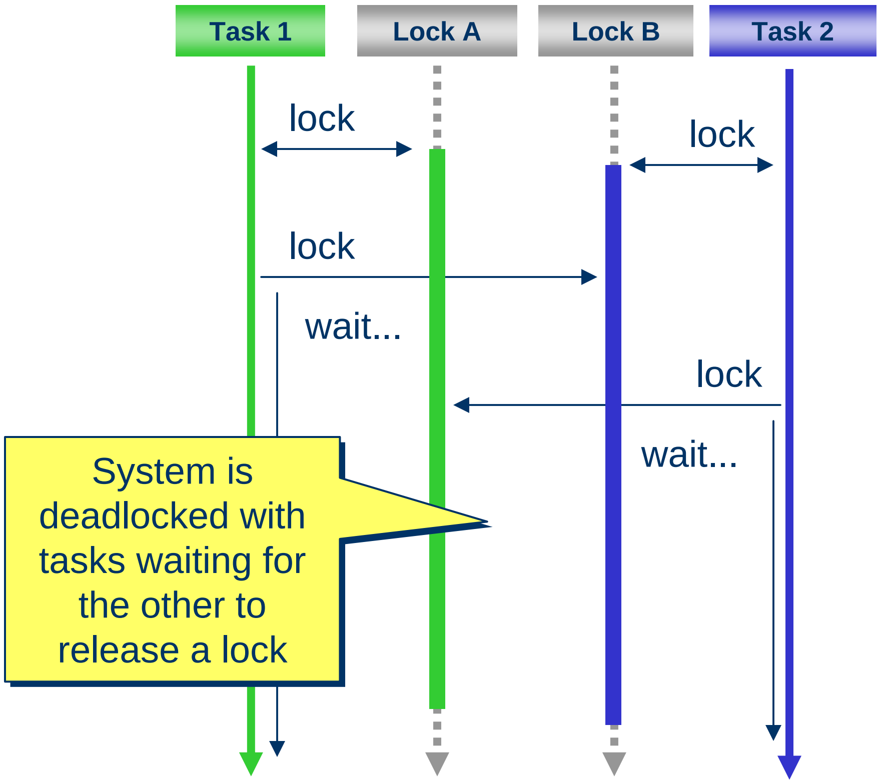
- Locks are intrinsic to shared-memory parallel programming to protect shared data & synch
- Taking multiple locks requires care
  - Deadlock occurs if tasks take locks in different order
  - Impose locking discipline/protocol to avoid
  - Hard to see locks in shared libraries & OS code
  - Locking order often hard to deduce
- Deadlocks also occur in “regular” multitasking
  - But multiprocessors make them much more likely
  - And multiprocessor programs have many more locks

Deadlocks

- Lucky Execution



- Deadlock Execution



System is deadlocked with tasks waiting for the other to release a lock

Deadlocks and Libraries

Task T1	Task T2

main():
lock(L1)
// work on V1
lock(L2)
// work on V2
unlock(L2)
// work on v1
unlock(L1)
	main():
	lock(L2)
	// work on V2
	foo()
	// work on V2
	unlock(L2)
	
	foo():
	lock(L1)
	// work on V1
	unlock(L1)

- Easy way to deadlock:
  - Calling functions that access shared data and their locks
  - Order of locks become opaque
- Need to consider the complete call chain

Partial Crashes

- A single task in a parallel program crashes
  - Partial failure of program, leaves other tasks waiting
  - For a single-task program, not a problem
- Detect & recover/restart/gracefully quit
  - Parallel programs require more error handling
- More common in multiprocessor environments as more parallel programs are being used

Parallel Task Start Fails

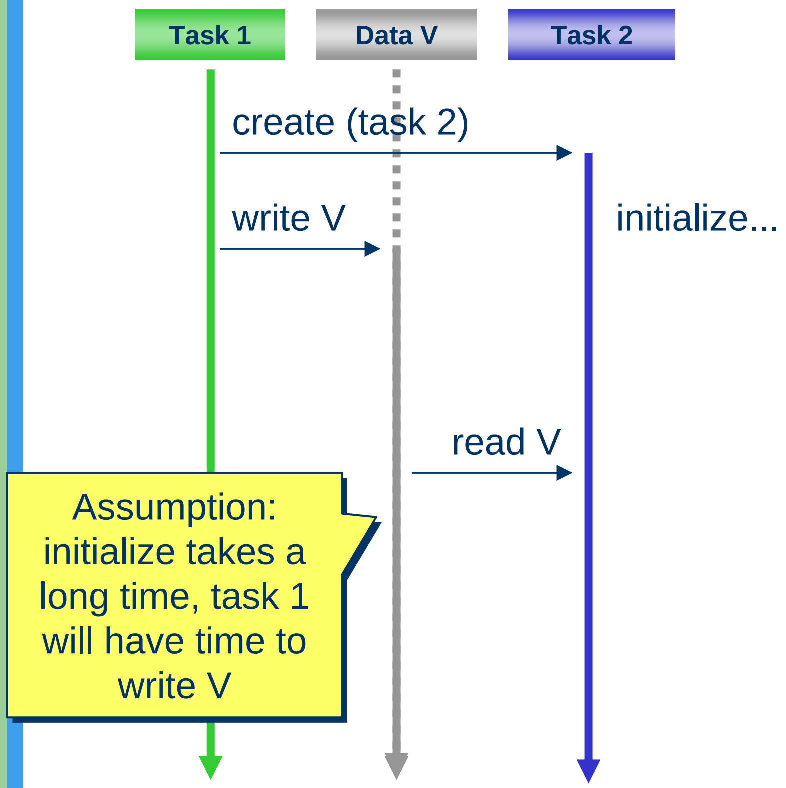
- Programs need to check if parallel execution did indeed start as requested
  - Check return codes from threading calls
- For directive-based programming like OpenMP, there is no error checking available in the API
  - Be careful!

Invalid Timing Assumptions

- We cannot assume that any code will run within a certain time-bound relative to other code
  - Any causal relation has to be enforced
  - Locks, synchronization, explicit checks for ordering
- Easy to make assumptions by mistake
  - Will work most of the time
  - Manifest under heavy load or rare scheduling

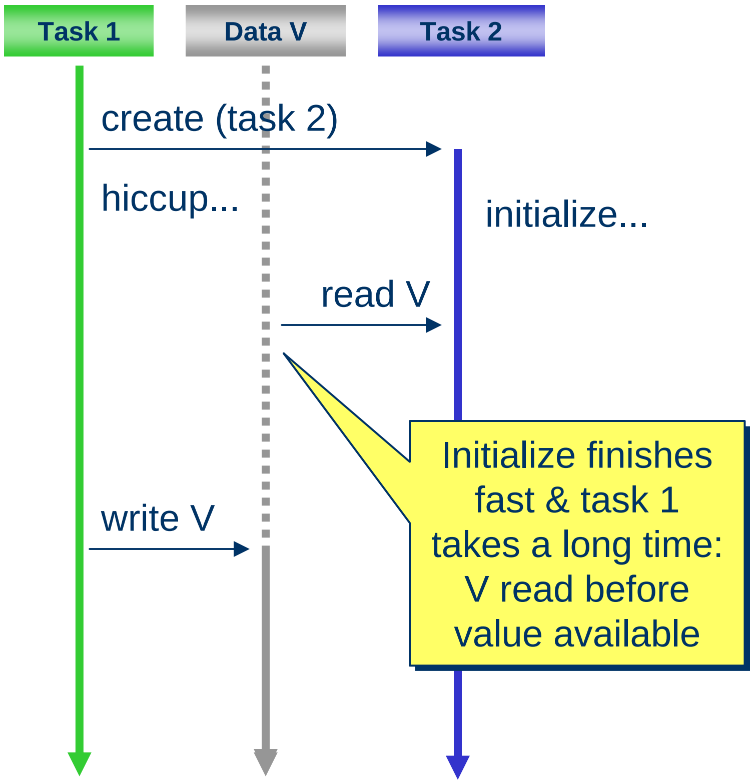
Invalid Timing Assumptions

- Assumed Timing



Assumption: initialize takes a long time, task 1 will have time to write V

- Erroneous Execution



Initialize finishes fast & task 1 takes a long time: V read before value available

Relaxed Memory Ordering

- Single processor: all memory operations will occur in program order*
  - * as observed by the program running
  - A read will always get the latest value written
  - Fundamental assumption used in writing code
- Multiprocessor: not necessarily the case
  - Processors can see operations in *different orders*
  - “**Weak consistency**” or “**relaxed memory ordering**”

Relaxed Memory Ordering: Why?

- **Performance**, nothing else
  - It complicates implementation of the hardware
  - It complicates validation of the hardware
  - It complicates programming software
  - It is very difficult to really understand
- Imposing a single global order of all memory events would force synchronization among all processors all the time, and kill performance

Relaxed Memory Ordering: What?

- It specifically allows the system to be less strict in synchronizing the state of processors
  - More slack = more opportunity to optimize
  - More slack = allow more reordering of writes & reads
  - More slack = greater ability to buffer data locally
  - More slack = more opportunity for weird bugs
- Exploited by *programming languages*, *compilers*, *processors*, and the *memory system* to reduce stall time

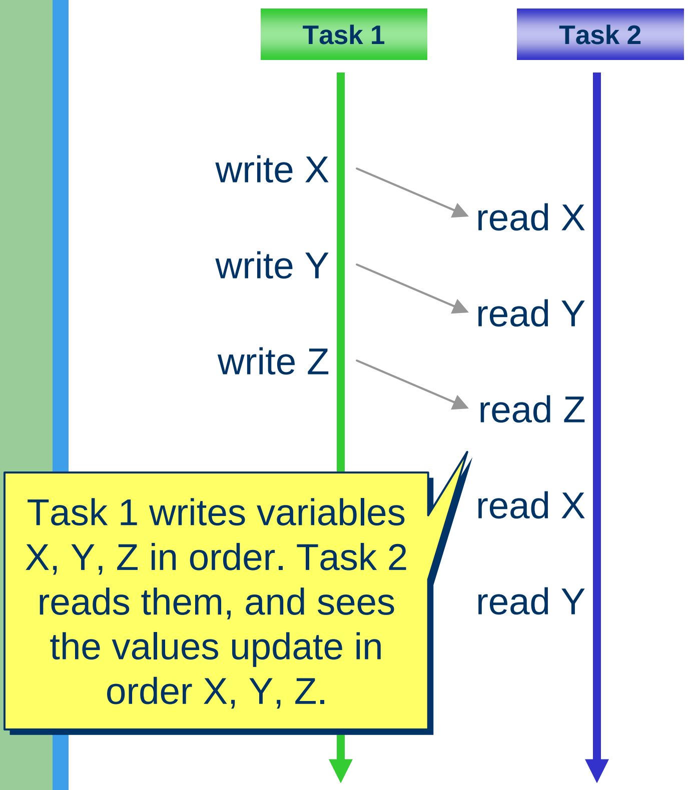
Relaxed Memory Ordering: Types

- Strong, non-relaxed memory order:
  - *SC, Sequential Consistency*, means that all memory operations from all processors are executed in order, and interleaved to form a single global order.
- Some examples of relaxed orders:
  - *TSO, Total Store Order*, reorder reads but not writes. Common model, not totally unintuitive (Sparc)
  - *WO, Weak Ordering*, reorder reads and writes and bring order explicitly using synchronization primitives (PowerPC, partially ARM)

For more information, see Hennessy and Patterson, *Computer Architecture, a Quantitative Approach* or other text books

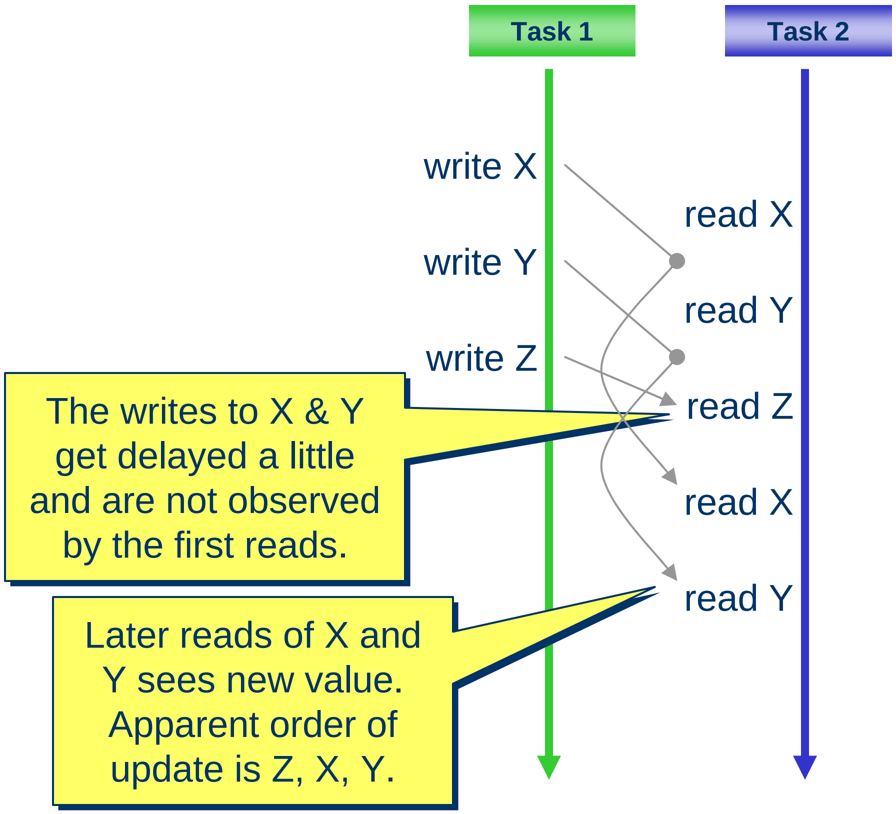
Relaxed Memory Ordering: Example

- Expected obvious case



Task 1 writes variables X, Y, Z in order. Task 2 reads them, and sees the values update in order X, Y, Z.

- Legal less obvious case



The writes to X & Y get delayed a little and are not observed by the first reads.

Later reads of X and Y sees new value. Apparent order of update is Z, X, Y.

Disclaimer: This example is really very very simplified. But it is just an example to show the core of the issue.

Relaxed Memory Ordering: Issues

- Synchronization code from single-processor environments might break on a multiprocessor
- Programs have to use synchronization to ensure that data has arrived before using it
- Subtle bugs that appear only in extreme circumstances (high load, odd memory setups)
- Latent bugs that only appear on certain machines (typically more aggressive designs)

Multipro-Unsafe Synchronization

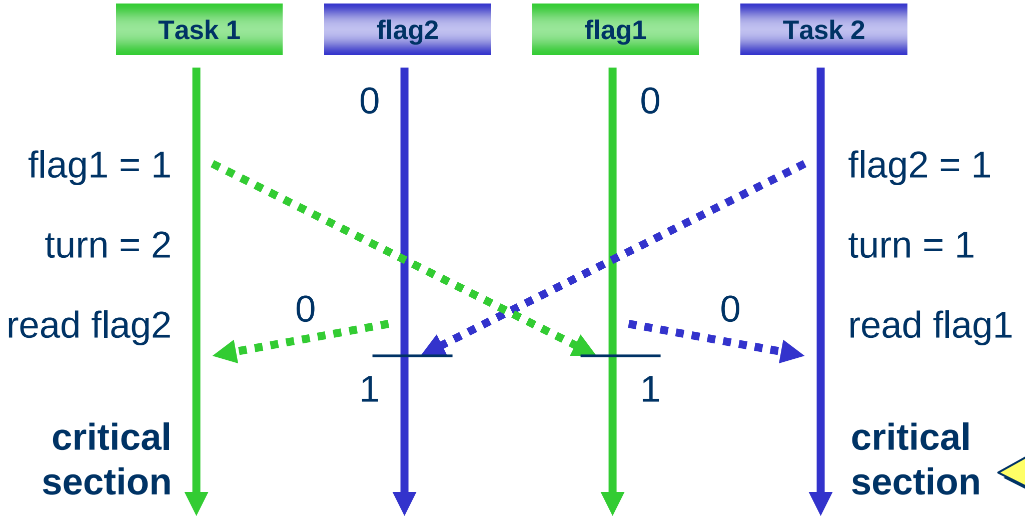
- Example algorithm
  - Simplified Dekker's
  - Textbook example
  - Any interleaving of writes allow a single task to enter the critical section
  - Works fine on single processor with multitasking
  - Works fine on sequential consistency machines

Task 1	Task 2

flag1 = 1
turn = 2
while(flag2 == 1 && turn == 2) wait;
//critical section
flag1 = 0
	flag2 = 1
	turn = 1
	while(flag1 == 1 && turn == 1) wait;
	//critical section
	flag2 = 0

Multipro-Unsafe Synchronization

- Example with relaxed memory ordering:
  - Both tasks do their writes in parallel, and then read the flag variables
  - Quite possible to read “old” value of flag variables since nothing guarantees that a write to one variable has completed before another one is read



Both tasks in critical section a the same time, not good

Memory Ordering & Programming

- Data synchronization operations:
  - *fences/barriers* to prevent execution from continuing until all writes and/or reads have completed. All memory operations are ordered relative to a fence.
  - *flush* to commit local changes to global memory
- Note that a weak memory order might cause worse performance if programmers are conservative and use too much synchronization
  - TSO is often considered “optimal” in this respect

Memory Ordering & Programming

- C/C++ has no concept of memory order or data handling between multiple tasks
  - *volatile* means nothing between processors
  - Use APIs to access concurrency operations
- Java has its own memory model, which is fairly weak and can expose the underlying machine
  - Not even “high level” programs in “safe” programming languages avoid relaxed memory ordering problems
- OpenMP defines a weak model that is used even when the hardware itself has a stronger model (compiler)

Relaxed Memory Ordering: Fixing

- Use SMP-aware synchronization
- Use data synchronization operations
- Read the documentation about the particular memory consistency of your target platform
  - ... and note that it is sometimes not implemented to its full freedom on current hardware generations...
- Use proven synchronization mechanisms
  - Do not implement synchronization yourself if you can avoid it, let the experts do it for you

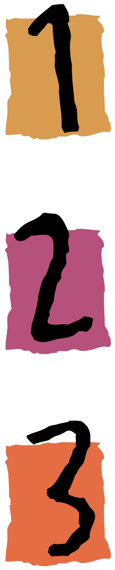
Missing Flush Operations

- Data can get “stuck” on a certain processor
  - In the cache or in the store buffer of a processor
  - Aggressive buffering and a relaxed memory ordering avoids sending updated data to other processors
  - Real example: a program worked fine on a Sparc multiprocessor, but it made no progress on a PowerPC machine. Flushes had to be added.
- Solution: explicit ”flush” operations to force data to be written back to shared memory

How Can We Debug It?

Three Steps of Debugging

1. Provoking errors
   - Forcing the system to a state where things break
2. Reproducing errors
   - Recreating a provoked error reliably
3. Locating the source of errors
   - Investigating the program flow & data
   - Depends on success in reproduction



Parallel Debugging is Hard

- Reproducing errors and behavior is hard
  - Parallel errors tend to depend on subtle timing, interactions between tasks, precise order of micro-scale events
  - Determinism is fundamentally not given
- **Heisenbugs**
  - Observing a bug makes it go away
  - The intrusion of debugging changes system behavior
- **Bohr bugs**
  - Traditional bugs, depend on the controllable values of input data, easy to reproduce

Breakpoints & Classic Debuggers

- Still useful
- Several caveats:
  - Stopping one task in a collaborating group might break the system
  - A stopped task can be swamped with traffic
  - A stopped task can trigger timeouts and watch dogs
  - Might be hard to target the right task

3

Tracing

- Very powerful tool in general
- Can provide powerful insight into execution
  - Especially when trace is “smart”

- Weaknesses:
  - Intrusiveness, changes timing
  - Only traces certain aspects
  - No data between trace points

Tracing Methods...

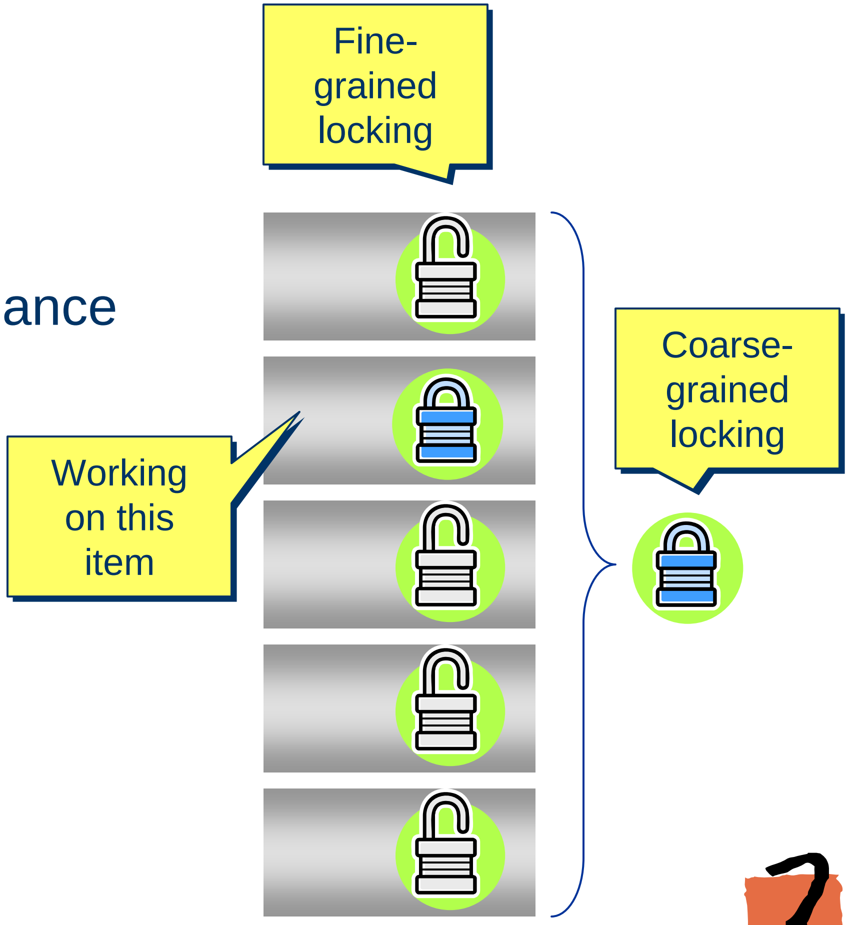
- Printf
  - Added by user to program
- Monitor task
  - Special task snooping on application, added by user
- Instrumentation
  - Source or binary level, added by tool
- Bus trace
  - Less meaningful in a heavily cached system

...Tracing Methods

- Hardware trace
  - Using trace support in hardware + trace buffer
  - Mostly non-intrusive
  - Hard to create a consistent trace due to local timing
- Simulation
  - Can trace any aspect of system
  - Differences in timing, requires a simulation model
- More information later about HW trace & sim

Bigger Locks

- Fine-grained locking:
  - Individual data items
  - Less blocking, higher performance
  - More errors
- Coarse locking:
  - Entire data structures
  - Entire sections of code
  - Lower performance
  - Less chance of errors, limits parallelism
- Make locks coarser until program works



Fine-grained locking

Working on this item

Coarse-grained locking

Apply Heavy Load

- Heavy load
  - More interference in the system
  - Higher chance of long latencies for communication
  - Higher chance of unexpected blocking and delays
  - Higher chance of concurrent access to data
- Powerful method to break a parallel system
  - Often reproduces errors with high likelihood
- Requires good test cases & automation

Use Different Machine

- Provokes errors by challenging assumptions
  - Different number of processors
  - Different speed of processors
  - Different communications latency & cache sizes
  - Different memory consistency model
- It is easy to inadvertently tie code to the machine the code is developed and initially tested on

1

Use Different Compiler

- Different compilers have different checks
- Different compilers implement multiprocessing features in different ways
- Thus, compiling & testing using different compilers will reveal errors both during compilation and at run-time

Note this techniques works for many categories of errors. Making sure a program compiles cleanly in several different environments makes it much more robust in general.

1

Multicore Debuggers

- Many debuggers are starting to provide support for multiple processors and cores
- Basics features:
  - Handling several tasks within a single debug session, at the same time
  - Understanding of multiple tasks and processors
  - Ability to connect to multiple targets at the same time

3

Multicore Debuggers

- Advanced features:
  - Visualizing tasks
  - Visualizing data flow and data values
  - Grouping processes and processors
  - Profiling that is aware of multiple processors
  - Pausing sets of tasks
  - Understanding the multiprocessor programming system used (synch operations, task start/stop)
- For multicore, you need new thinking in debug

Multicore Debugger Implementation

- Impact on potential probe effects and observation power
- Implementation choices
  - Instrument the code
  - Instrument the parallelization library (OpenMP, MPI, CAPI-aware debuggers)
  - Use an OS-level debug agent
  - Use hardware debug access
- In all cases, the debugger has to understand what is running where, and this makes OS-awareness almost mandatory

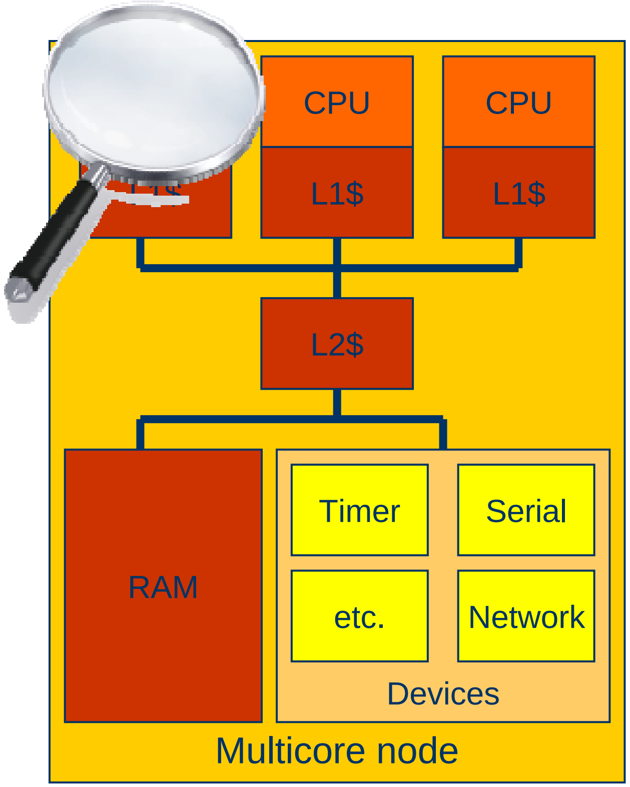
Multipro Hardware Debug Support

- Requires a multicore-aware debugger to be really useful, obviously
- Hardware should supply the ability to:
  - Access data and code on all processors
  - Stop execution on any processor
  - Trace memory and instructions
  - Access high volumes of debug and trace data
  - Synchronize stops across multiple processors

3

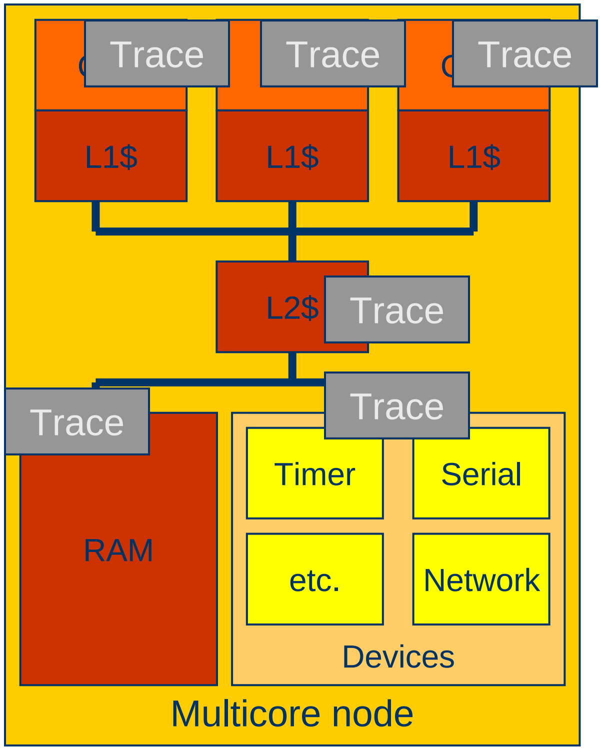
Multipro Hardware Debug Support

- Trace
  - Trace behavior of one or more processors (or other parts)
  - Without stopping system or affecting timing
  - Can be local to a core
  - Present in many designs today (e.g., ARM ETM)
  - Good and necessary start



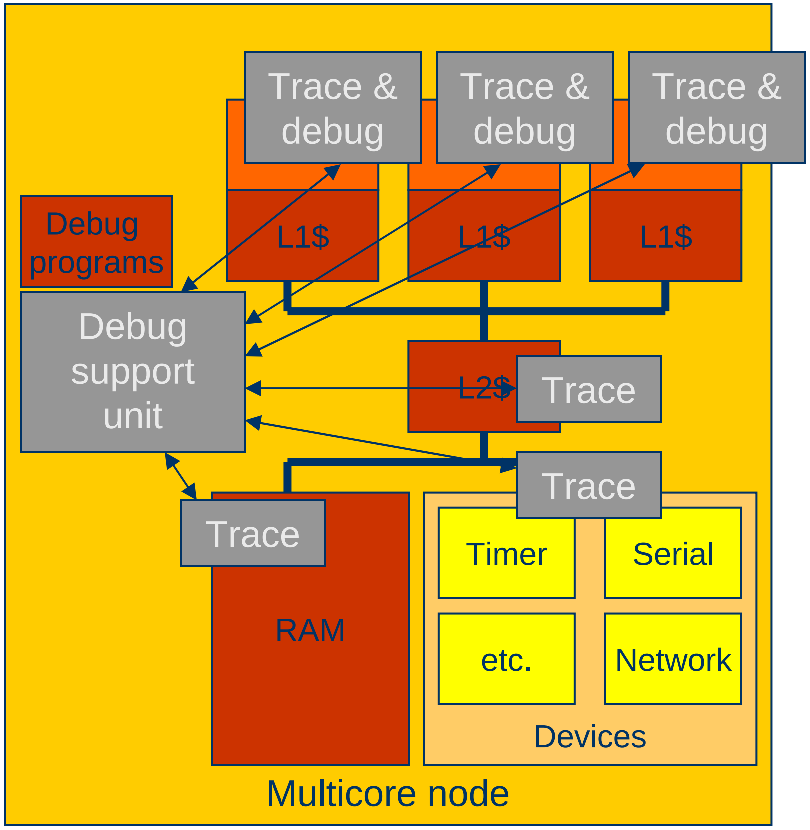
Multipro Hardware Debug Support

- Trace
  - For full effect, want trace units at all interesting places in a system, not just at processors
  - Costs some chip area, might not be present in “shipping” versions of a multicore SoC
  - Note that debug interface **bandwidth limitations** can put a limit on effectiveness



Multipro Hardware Debug Support

- Cross-triggering
  - Hardware units listen to events on all cores
    - Breakpoints, raw memory trace, watchpoints, interrupts...
  - Cause action in one core based on events occurring in other cores or elsewhere in the system
    - Stop execution, start tracing, stop tracing, interrupt, ...
    - Requires logic on the multicore chip
    - Basically, it is programmable



Multipro Hardware Debug Support

- Currently quite sketchy
  - New implementations and ideas are coming out
  - Standards are arriving, like ARM CoreSight
  - Better debug access ports are being added to HW
- The acceptable overhead cost in hardware seems fixed at about 10%

Replay Execution

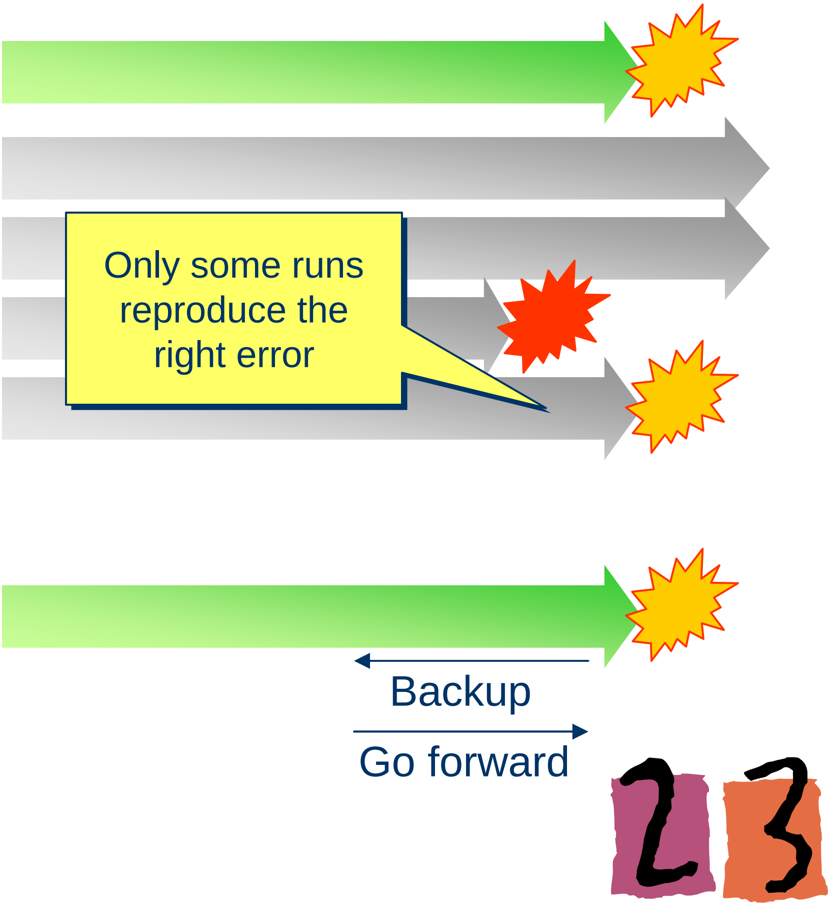
- Record a system execution, replay it
  - Solves reproduction problem, if an error is recorded
  - Controlled replay minimizes the probe effect
  - Apply debuggers during replay
- Record asynchronous events & inputs
  - Interactions between tasks
  - Isolates the system from the outside world
- Requires specialized tool support

Replay Execution: Inputs

- Replay problem can also be attacked at higher level by capturing and replaying input streams
  - Does not capture precise execution pattern inside the system being debugged
  - A very useful tool for rare events if they are somewhat deterministic from the input
  - Reasonable approach to reproduce problems found in the field, if the field has capture equipment

Reverse Debugging

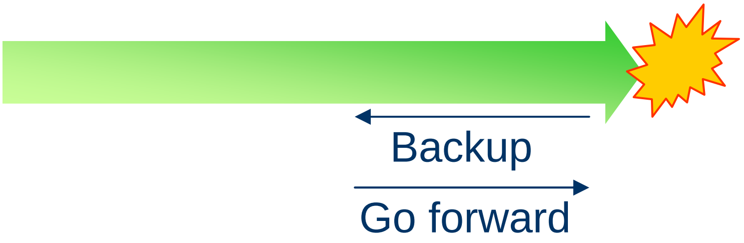
- Stop & go back in time
  - Instead of rerunning program from start
  - No need to rerun and hope for bug to reoccur
  - Investigate exactly what happened this time
  - Breakpoints & watchpoints backwards in time
  - Very powerful for parallel programs



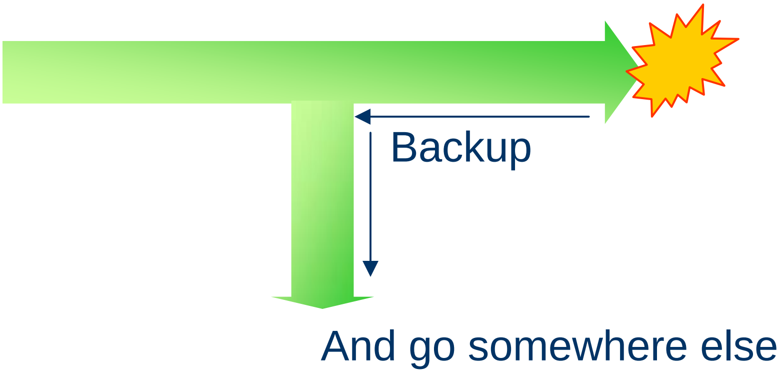
Only some runs reproduce the right error

Reverse Debugging: Techniques

- Trace-based
  - Record system execution
  - Special hardware or simulator support
  - Use as “tape recorder”, fixed execution observed



- Simulation-based
  - Record in simulator
  - Replay in same simulator
  - Can change state and continue execution

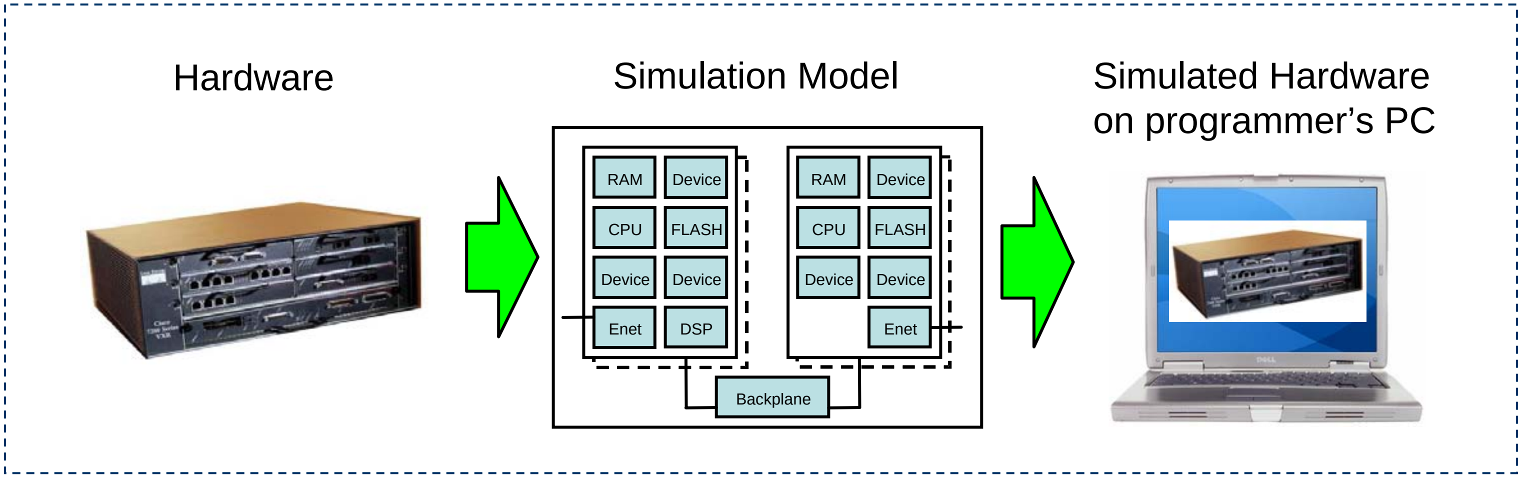


Reverse Debugging Tools

- For new techniques like this, I like to point out some vendors that you can find on the show floor today:
  - Green Hills
  - Virtutech
  - IAR Systems (?)
  - Lauterbach (?)

Simulate the System

- Use simulation model instead of the hardware



Hardware

Simulation Model

Simulated Hardware on programmer's PC

- Offers more control and insight than physical hardware = better debug

Simulate the System

- Simulation uses:
  - Vary parameters to provoke errors
  - Inject variations in execution to provoke errors, steer system towards known corner cases
  - Reliable reproduction of problems
  - Powerful inspection abilities
  - No probe effect from tracing and breakpoints
  - Can support record & replay, and reverse debugging

Simulate the System

- Need to run the same binaries as the physical hardware target
  - Including the operating system with synchronization primitives and scheduling algorithms
  - Requires modeling not only processors but also devices, interrupt controllers, shared memory, etc.
  - Simple classic instruction-set simulators do not run operating systems or handle multiple processors
  - Performance has to be sufficient for SW load
    - Varies widely between types of target machines

Simulate the System

- In practice:
  - To be fast enough, simulation cannot use RTL implementation. Has to be at a higher level.
  - Modeling is an additional task in development, you cannot use simulation without a simulation model.
  - Most cases, you can keep using existing debuggers and other programming tools. Simulation replaces hardware, not your software tools.
  - You still need the hardware, but possibly less of it and not quite as often.

Simulate the System

- Fidelity of simulation:
  - Simulation is never quite like the real thing, but it is close enough to be useful
  - Any bugs found in simulation are valid bugs
    - Simulation does explore a range of behaviors of the real system. If desired, simulation can force certain behavior.
  - Precise timing simulation is not really possible
    - “Cycle Accuracy” is really a “Cycle Approximate”

See Ekblom and Engblom, “Simics: a commercially proven full-system simulation framework”, *SESP 2006*, for a deeper discussion

Example Bug found in Simulation

- Changed clock frequency of virtual MPC8641D
  - From 800 to 833 Mhz
  - OS froze on startup – quite unexpectedly
- Investigation:
  - Only happened at 832.9 to 833.3 MHz
  - Determinism: 100% reproduction of error trivial
  - Time control: single-step code feasible
  - Insight: look at complete system state, log interrupts, check the call stack at the point of the freeze, check lock state
- What we found:
  - ISR takes a lock on entry, and then expect a second external interrupt to occur to unlock the data structure. But this interrupt arrives before interrupts are reenabled, and thus we are stuck in deadlock. Took a few hours to find.

Simulation Tool Vendors

- Some vendors:
  - ARM
  - CoWare
  - Synopsys (Virtio)
  - Vast
  - Virtutech
  - (Qemu, Bochs, and other open-source tools)

Static Code Analysis

- Analyze the source code to determine all possible program behaviors
  - Possible variable values, possible (and impossible) execution paths, etc.
  - Without running the program
  - Think of “lint” on steroids
  - Current tools have some support for parallel programs

Static Code Analysis

- In practice:
  - Analysis times can be very long (hours, days)
  - Code should be written to support analysis
    - Best solution: well-designed subsets like SPARK Ada
    - Existing code often hard to analyze properly

Static Code Analysis & Parallelism

- Parallel analysis exponentially more difficult
- Parallel support limited to certain APIs/primitives
  - Synchronization and locking operations known
  - Semantics of operations known
- Limited to certain classes of errors
  - Check locking order for deadlocks, for example
  - Cannot find unprotected access to global data
  - Cannot prove correctness of fundamental synchronization operations
  - Cannot see effects of subtle timing shifts or memory consistency

Static Code Analysis Vendors

- Vendors I know of:
  - Coverity
  - Polyspace
  - Green Hills
  - + a range of MISRA-compliance checkers

Dynamic Analysis

- Run a program and trace its behavior
  - Unlike static analysis, no attempt to analyze source code offline
- Generalize from behavior in a concrete run to a larger set of potential program execution paths
  - “What happens if these operations are interleaved differently?”
  - “What other orders are allowed by synchronization?”
- Analyze the set of potential paths to find possible errors
  - Locking order
  - Use of uninitialized variables
  - Efficiently find such hard-to-find bugs

Dynamic Analysis

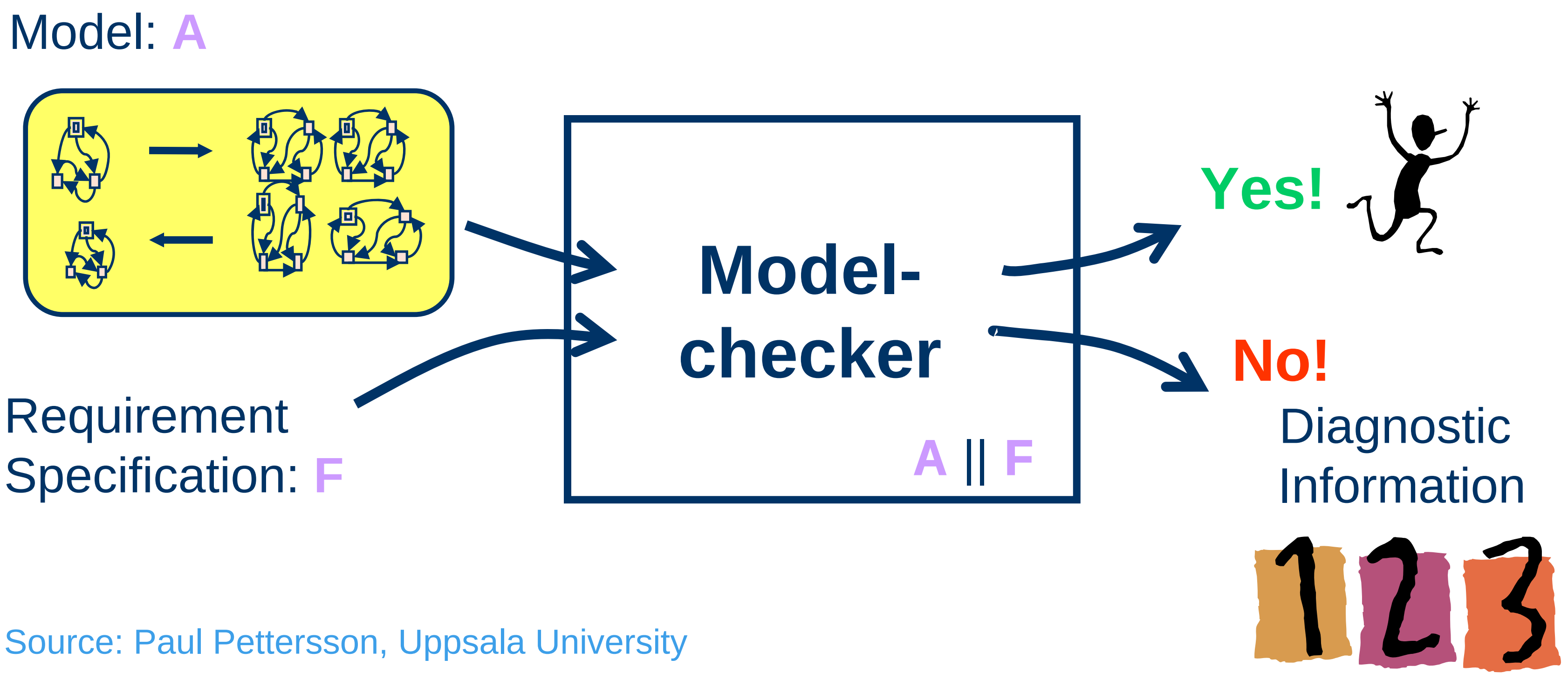
- In practice
  - Cannot find all possible paths, only those similar to the ones found in the concrete runs
    - Code that is never executed in concrete runs will never be examined, for example
  - Specific algorithms needed for each class of errors
  - Programs have to be instrumented
    - Often run-time slowdown of factor 10 or more
- Highly practical approach for some errors

Dynamic Analysis Tools

- Tools on the market:
  - Intel ThreadChecker, works with locking discipline
  - Open-source program Valgrind

Model Checking

- Static verification technique for parallel systems



Model: A

Requirement Specification: F

Model-checker

A || F

Yes!

No!

Diagnostic Information

Source: Paul Pettersson, Uppsala University

Model Checking

- Explores the entire state space of a system
  - Verifies that certain properties hold
- Requires a model of the **behavior** of a system
  - Typically expressed as an extended state machine
  - Not the same as a simulation model of the hardware
- Requires a specification of **properties** to check
  - Invariants – bad states are never entered
  - Liveness – something good will eventually happen

Model Checking

- In practice:
  - You will verify a model, not the actual system
  - Building a model and formulating properties to check can be hard work
  - Compared to many other formal methods, the output is much better since it is a concrete trace of events
- Applicability:
  - Very successful for protocols (leader elections, transmission protocols, synchronization protocols)
  - Best used at the specification stage of a project

Summary & Outlook

Summary

- We are at the beginning of the era of widespread parallel computation

- Hardware is leading the move
  - Parallelism is a major paradigm shift for software
  - Software and software tools are racing to catch up
  - Education and training needs to be updated
  - Programmers need to relearn programming

Outlook

- To manage the software, we need:
  - New programming paradigms
    - Expressed in new or modified programming languages
  - New debug and analysis techniques
    - Supported by good tools
  - Hardware support!
    - For programming paradigms
    - For debug and analysis
- I think we are going to see many interesting hardware-software combinations

Questions?

Thank You!

Please remember to fill in the course evaluation forms completely!