Debugging Multicore & Shared-Memory Embedded Systems

Jakob Engblom, PhD
Virtutech
jakob@virtutech.com

Scope & Context of This Talk

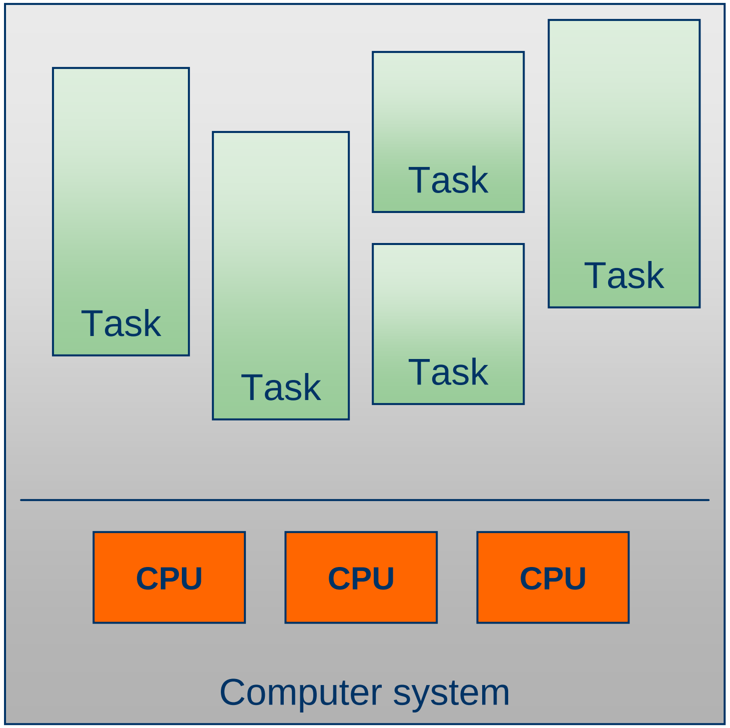
- Multiprocessor revolution
- Error sources
- Debugging techniques

- For **shared-memory symmetric multiprocessors**

Introduction & Background

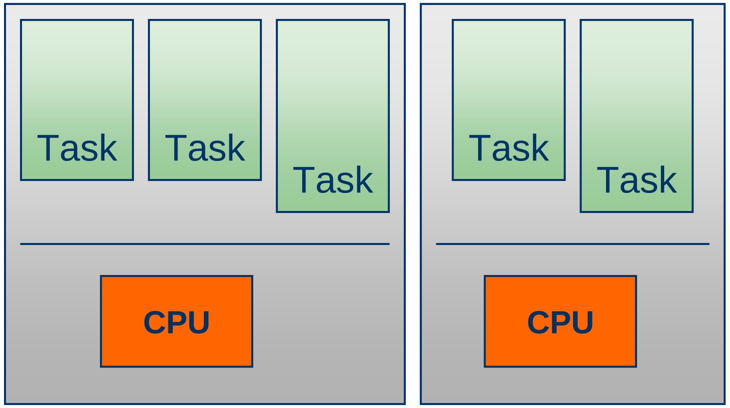
Vocabulary in the Multi Era

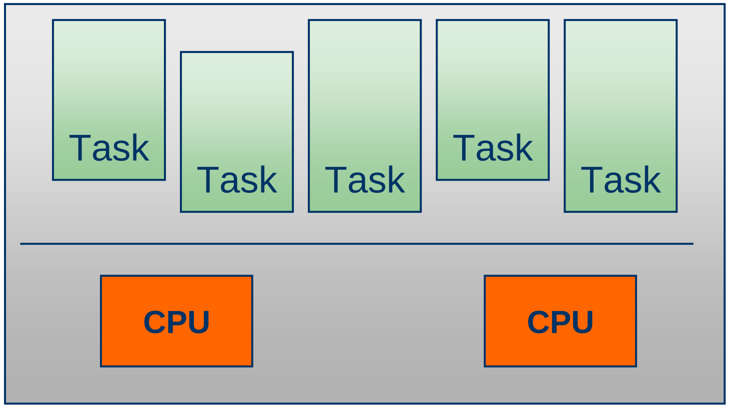
- **Multitasking**: multiple tasks running on a single computer
- **Multiprocessor**: multiple processors used to build a single computer system



Vocabulary in the Multi Era

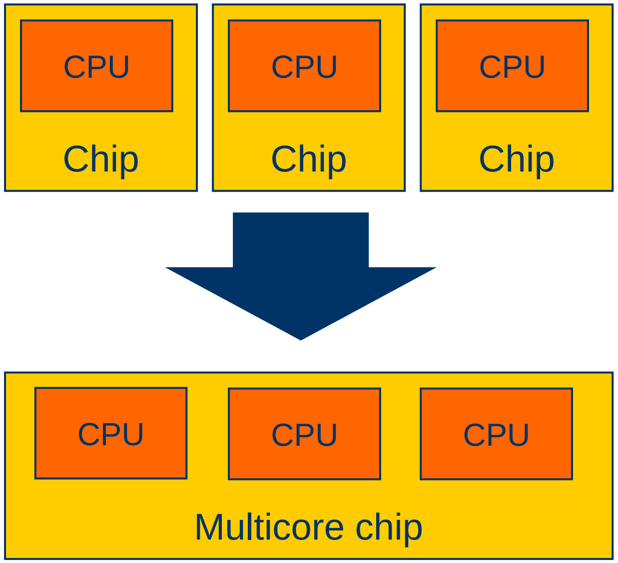
- **AMP, Assymetric MP**: Each processor has local memory, tasks statically allocated to one processor
- **SMP**, **Shared-Memory MP**: Processors share memory, tasks dynamically scheduled to any processor





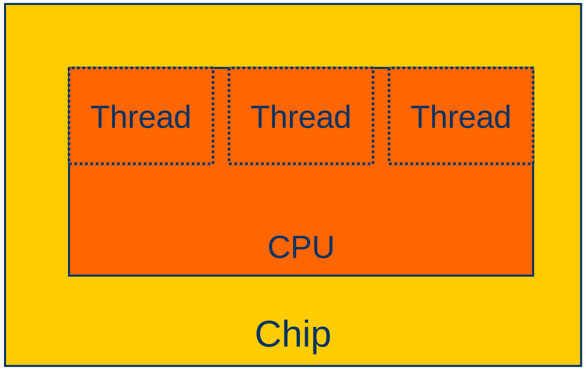
Vocabulary in the Multi Era

- **Multicore**: more than one processor on a single chip
- **CMP, Chip MultiProcessor**: Shared-memory multiprocessor on a single chip

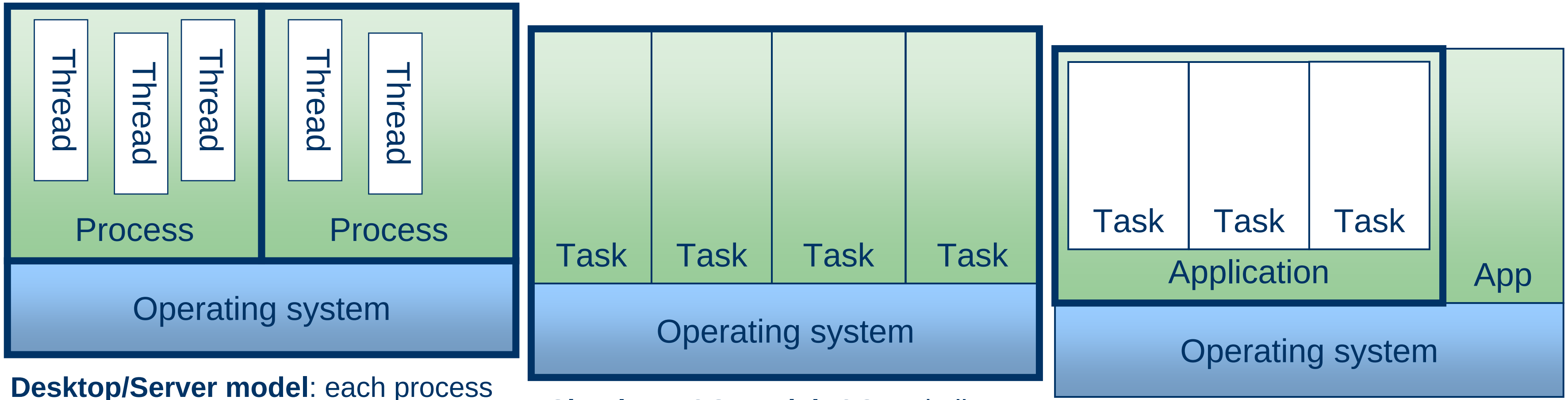


Vocabulary in the Multi Era

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



Process, Thread, Task



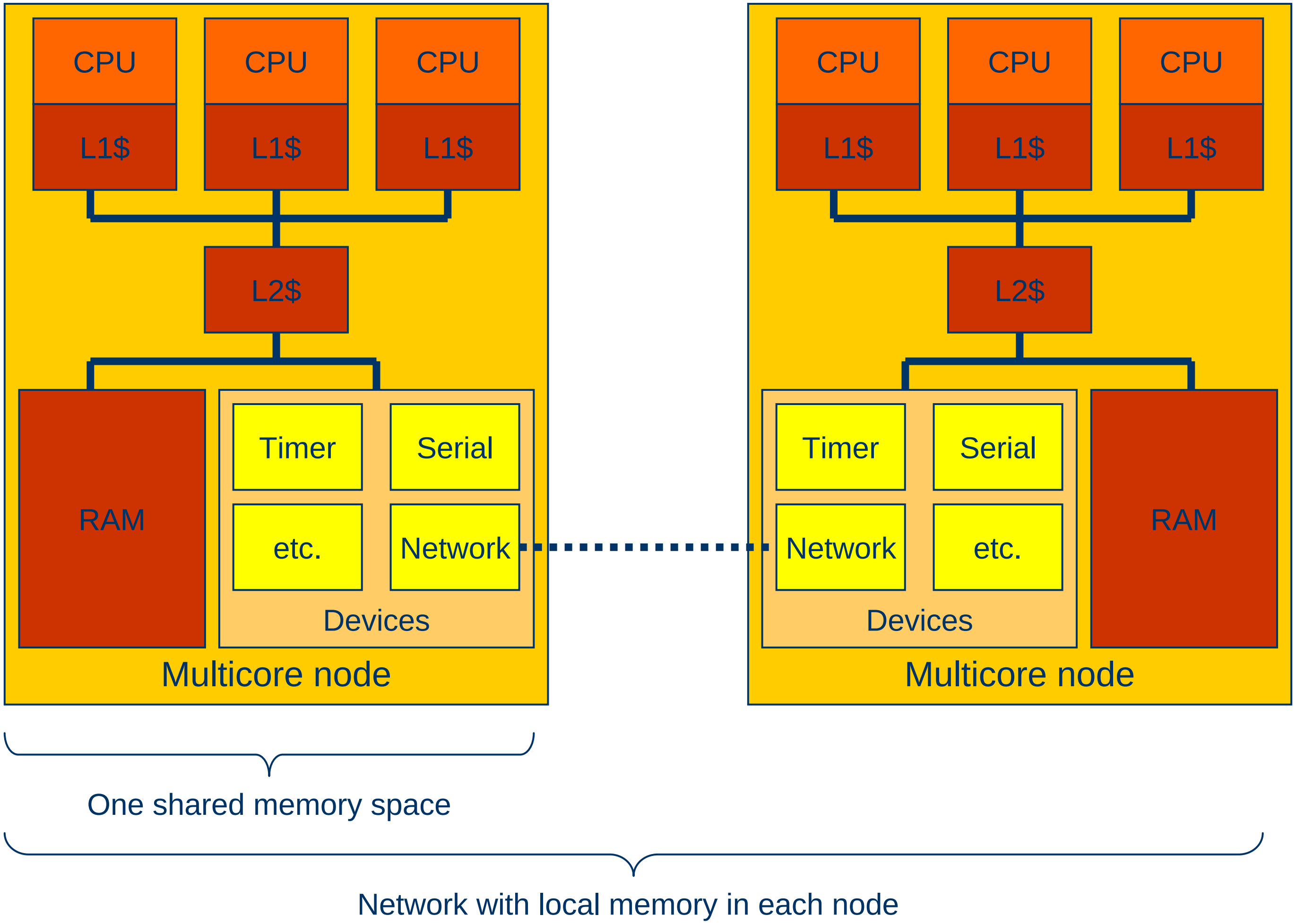
Desktop/Server model: each process in its own memory space, several threads in each process with access to the same memory. Memory protected between processes.

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

Future Embedded Systems



Why Now?

- More instruction-level parallelism hard to find
  - Very complex designs needed for small gain
- Clock frequency scaling is slowing drastically
  - Too much power & 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 performance

The Software is the Problem

- Parallelism required to gain performance
  - Parallel hardware is “easy” to design
  - Parallel software is 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*

Programming Models

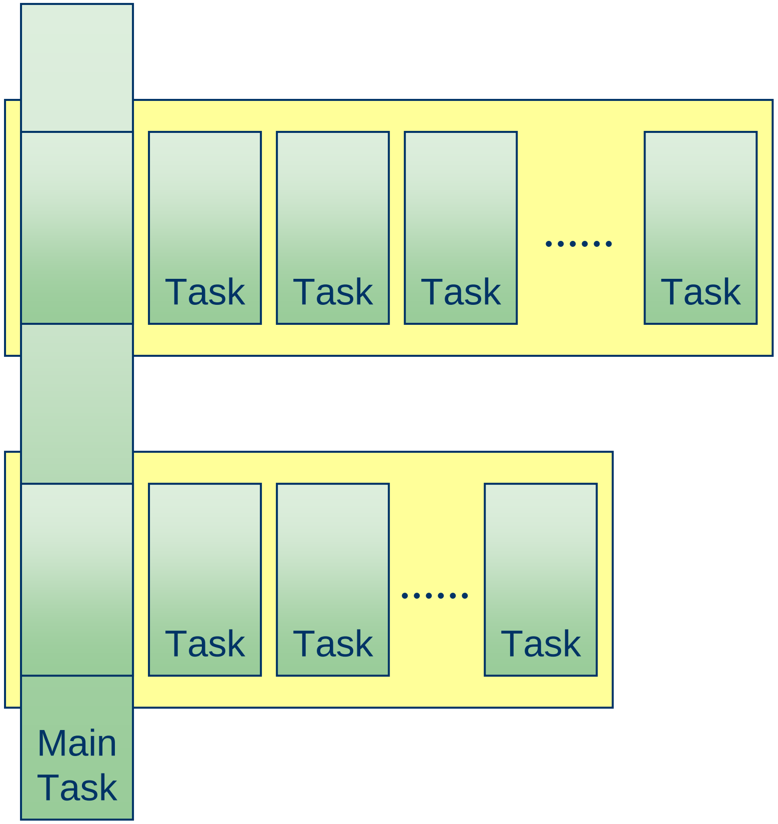
Programming Shared-Memory

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

- Components of the basic solution:
  - Shared memory
  - Locks to protect shared data
  - Synchronization primitives to coordinate execution

Success: Classic Supercomputing

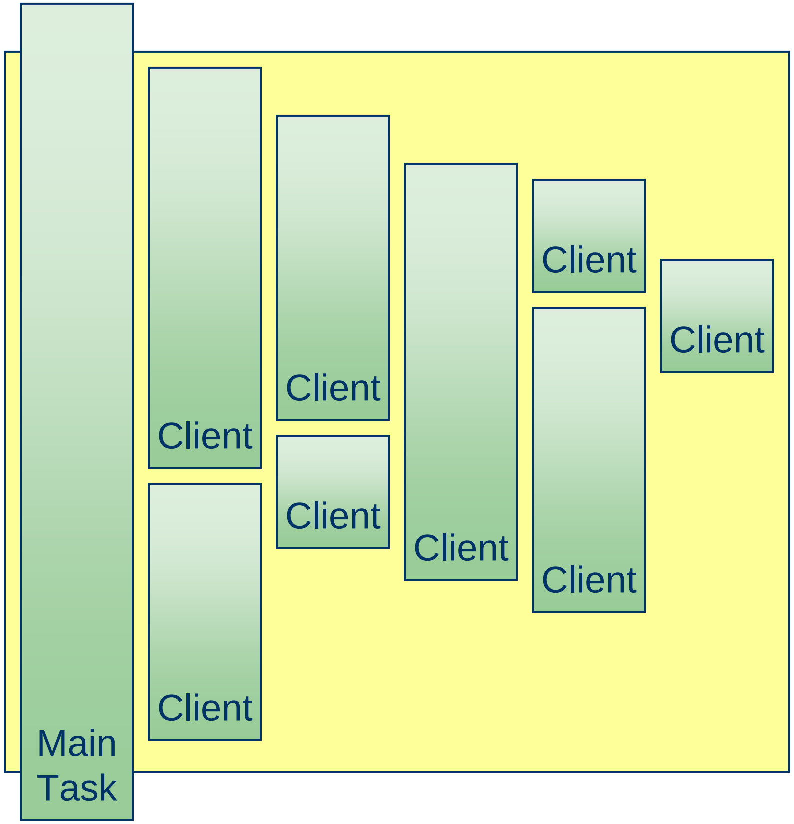
- Regular programs
- Parallelized loops + serial sections
- Data dependencies between tasks
- Very high scalability, 1000s of processors
- OpenMP, pthreads, MPI



Success: Servers

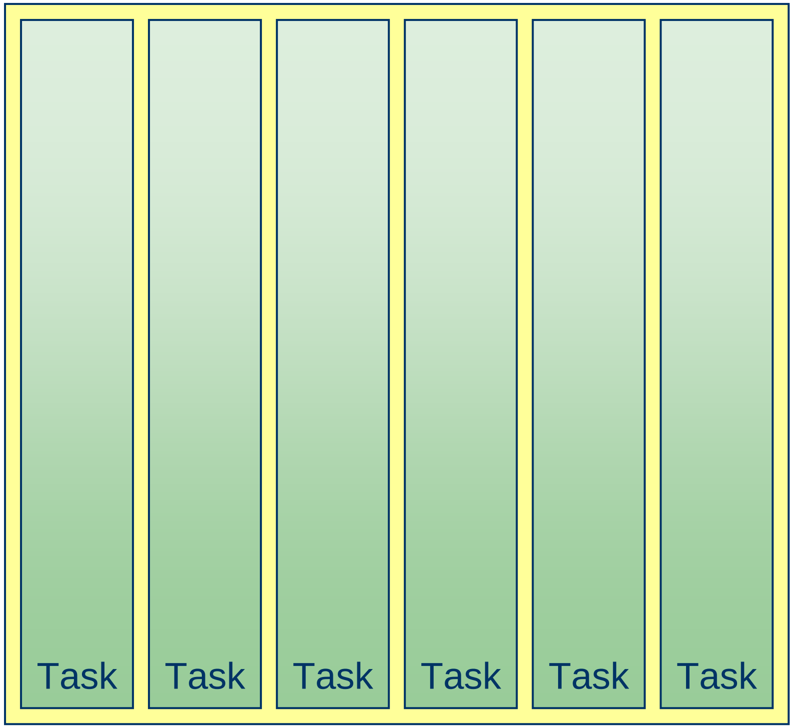
- Natural parallelism
- Irregular length of parallel code, dynamic creation
- Master task
- Slave tasks for each connection
- Scales very well

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



Success: Signal Processing

- "Embarrassing" natural parallelism
  - No shared data
  - No communication
  - No synchronization
- Parallelizes to 1000s of tasks and processors
- Good fit asymmetric MP



Programming model: Posix Threads

- Standard API
- Explicit operations
- Strong programmer control
- 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

- Standard API
- Message-passing
  - Local memory for each thread
  - Explicit messages for communication
  - Shared memory hidden

```
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();
}
```

What Goes Wrong?

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 well-tested code can break

Embedded Systems CONFERENCE SILICON VALLEY

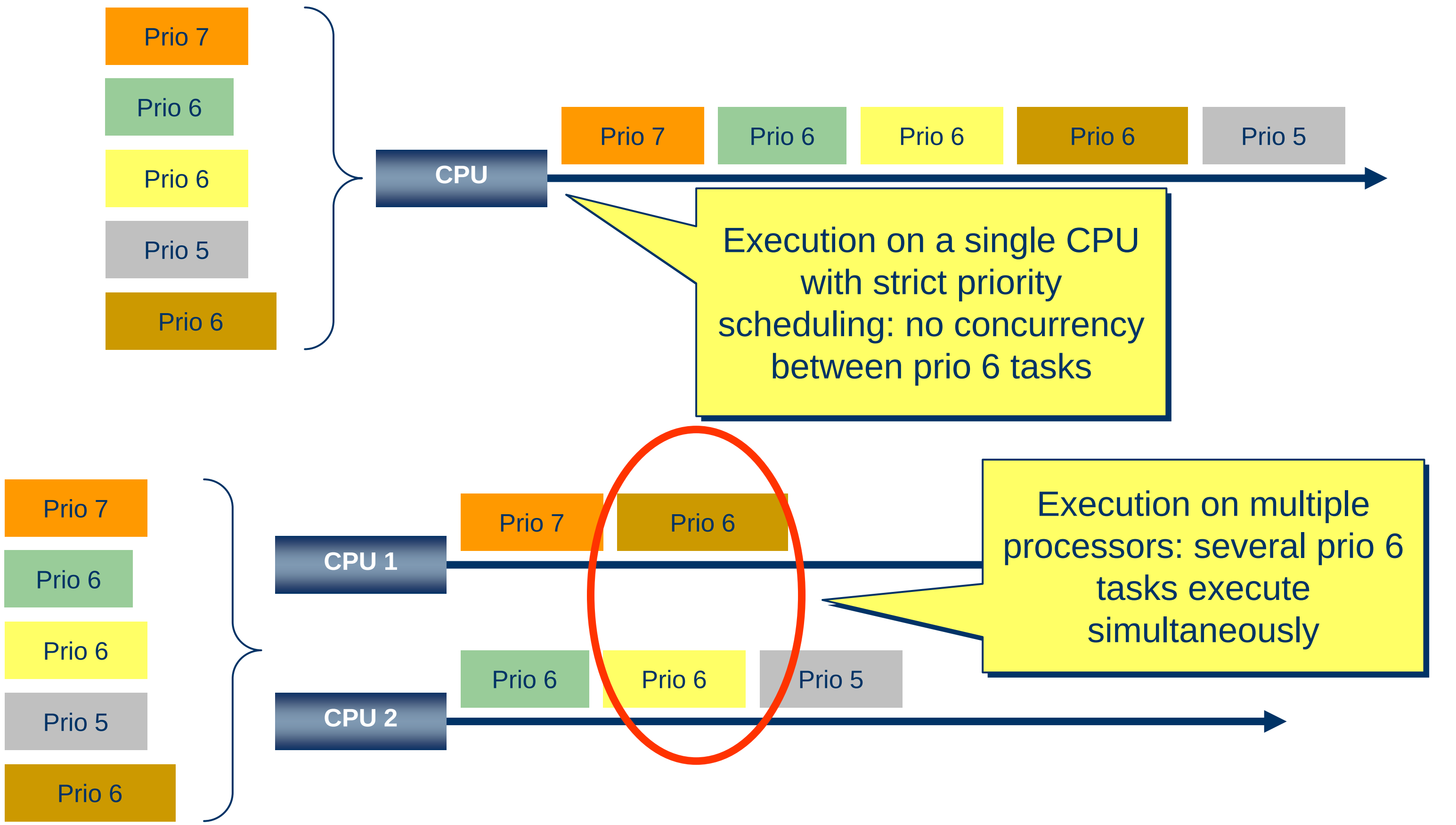
(Missing) Reentrancy

- Code shared between tasks has to be reentrant
  - No global variables
  - No assumption of single thread of control
- True concurrency = much higher chance of parallel execution of code
  - Problem also occurs in multitasking

Priorities are not Synchronization

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

Priorities are not Synchronization



Disabling Interrupts is not Locking

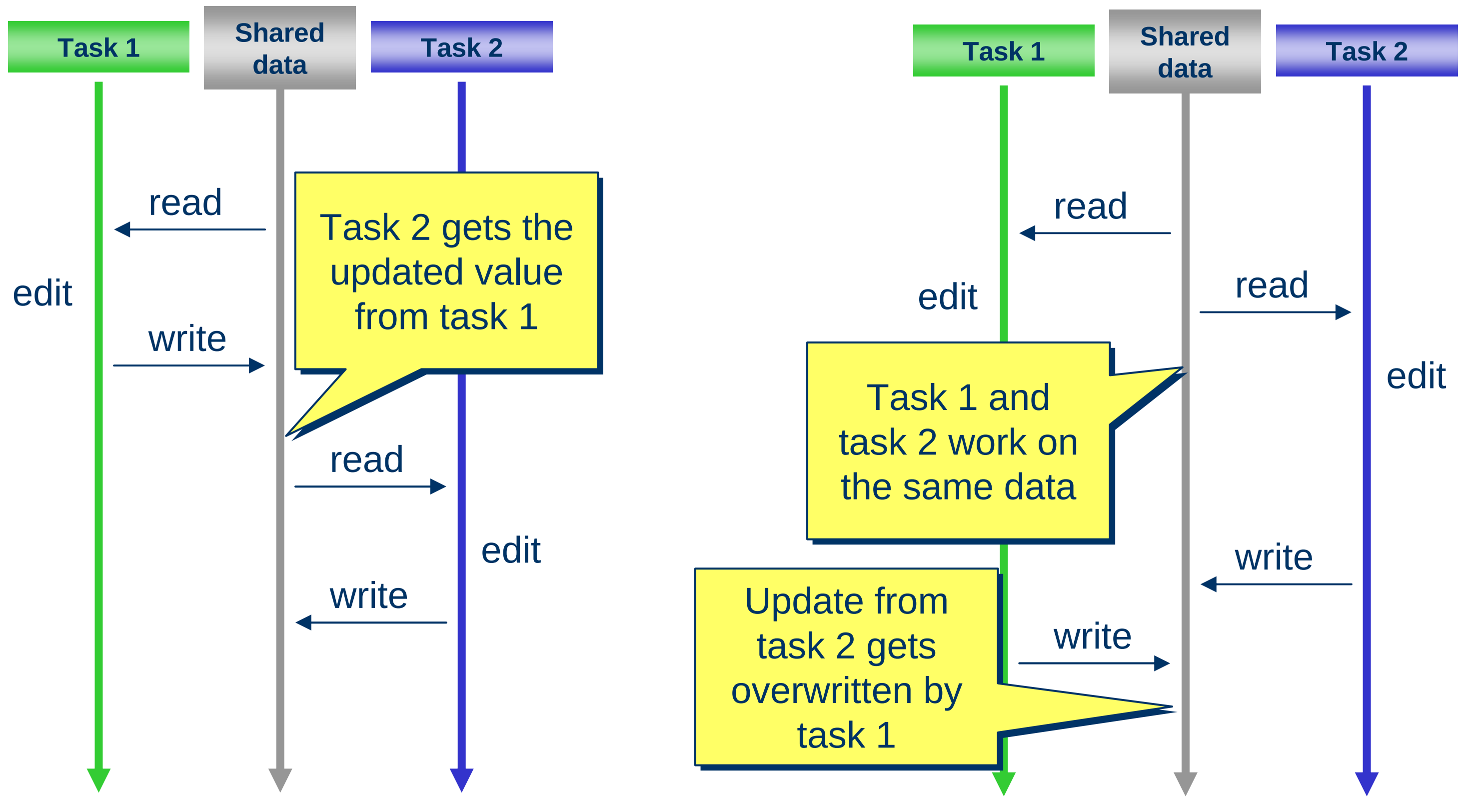
- Single processor: DI = cannot be interrupted
  - Guaranteed exclusive access to whole machine
  - Cheap mechanism, used in many drivers & kernels
- Multiprocessor: DI = stop interrupts on one core
  - Other cores keep running
  - Shared data can be modified from the outside
- Big issue for kernel porting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

Race Condition

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

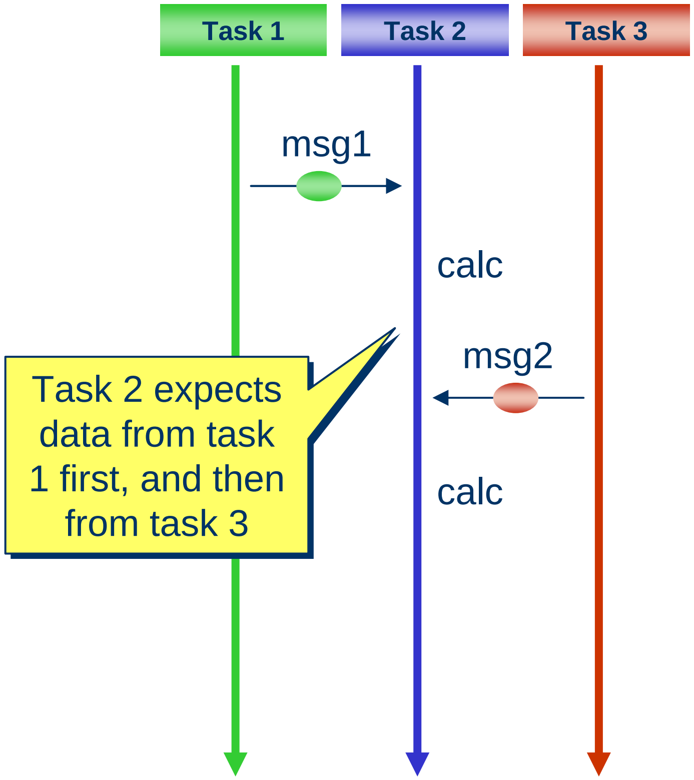
Race Condition

- Correct behavior
- Incorrect behavior

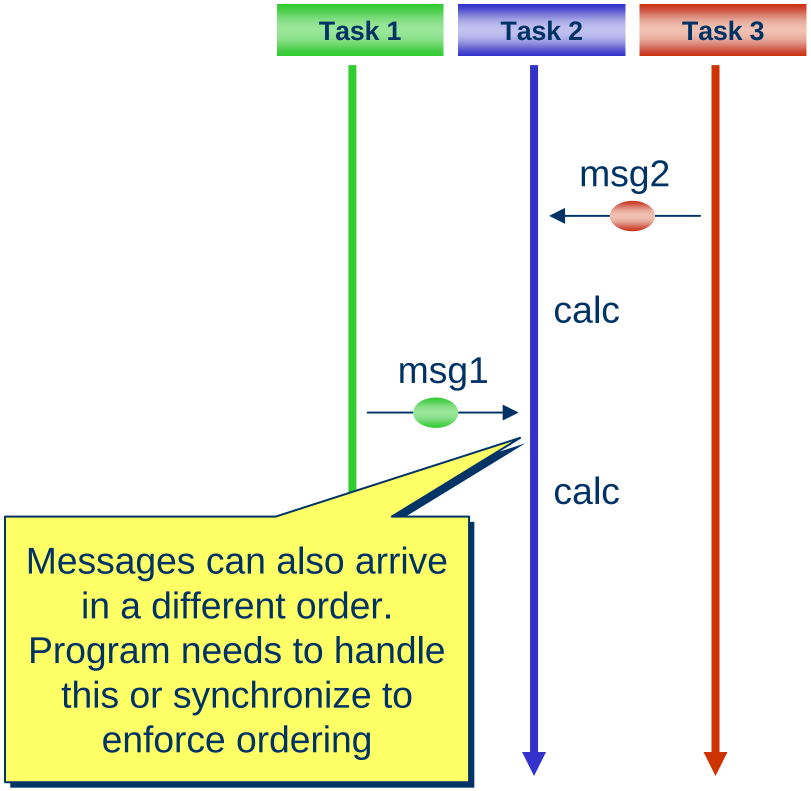


Race Condition: Messages

- Expected sequence



- Incorrect sequence

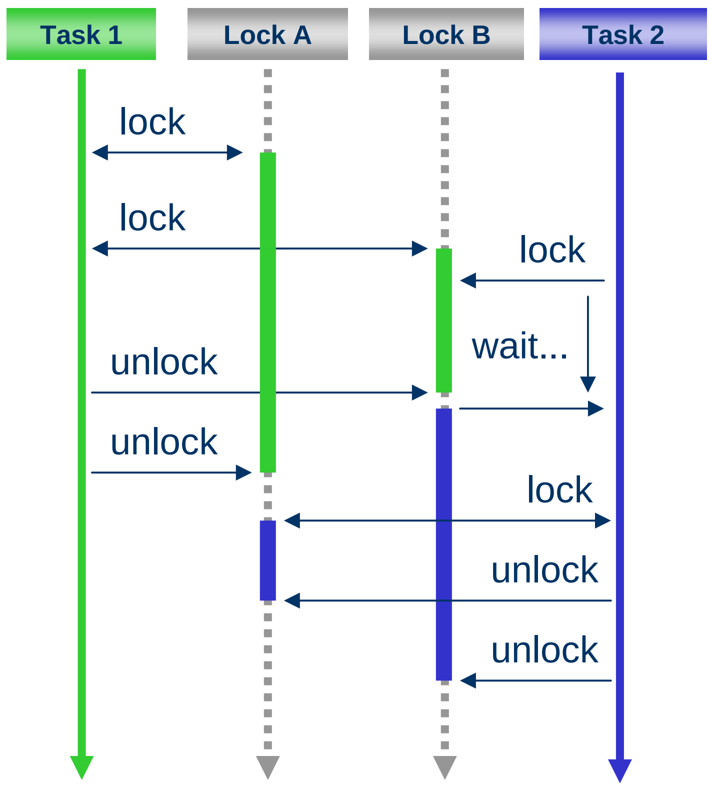


Deadlocks

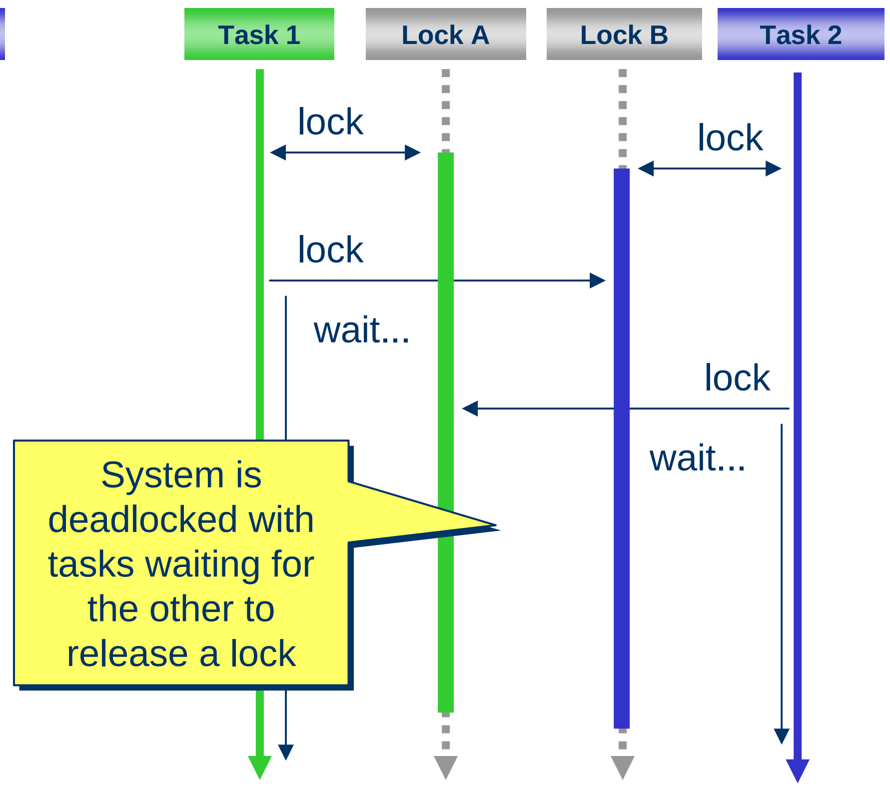
- Locks are intrinsic to parallel programming
  - Necessary to protect shared data, for example
- Taking multiple locks requires care
  - Deadlock occurs if tasks take locks in different order
  - Impose locking discipline/protocol to avoid
  - Hard to see locks in shared libraries & OS code
  - Locking order often hard to deduce
- Deadlocks also occur in “regular” multitasking
- But parallel programming *requires* multiple tasks

Deadlocks

- Lucky Execution



- Deadlock Execution



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

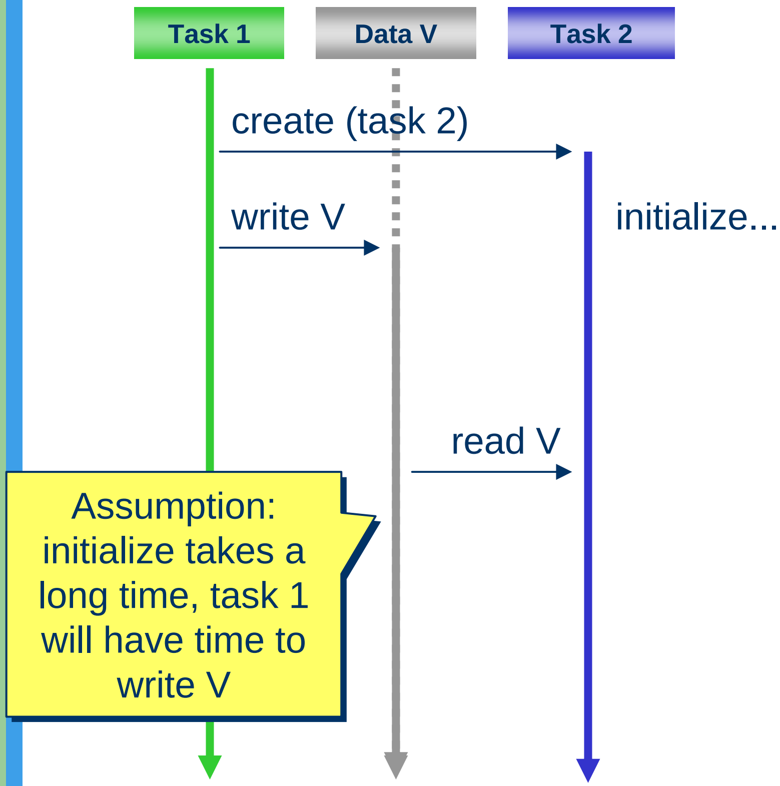
- Programs need to check if parallel execution did indeed start as requested
  - Check return codes from threading calls
- For directive-based programming like OpenMP, there is no error checking available
- Be careful!

Invalid Timing Assumptions

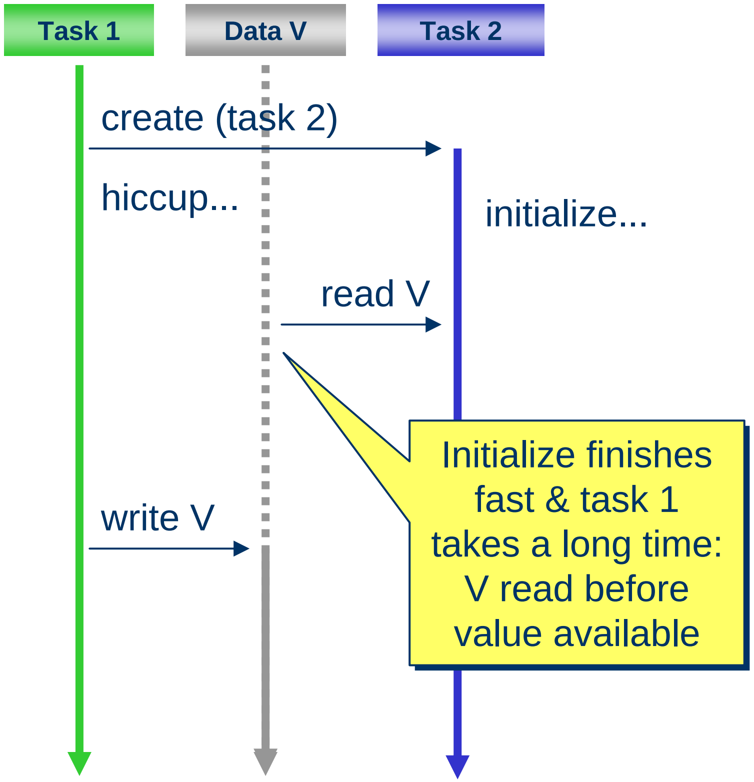
- We cannot assume that any code will run within a certain time-bound relative to other code
  - Unless there is explicit synchronization & checks
- Easy to make assumptions by mistake
  - Will work most of the time
  - Manifest under heavy load

Invalid Timing Assumptions

- Assumed Timing



- Erroneous Execution



Relaxed Memory Ordering

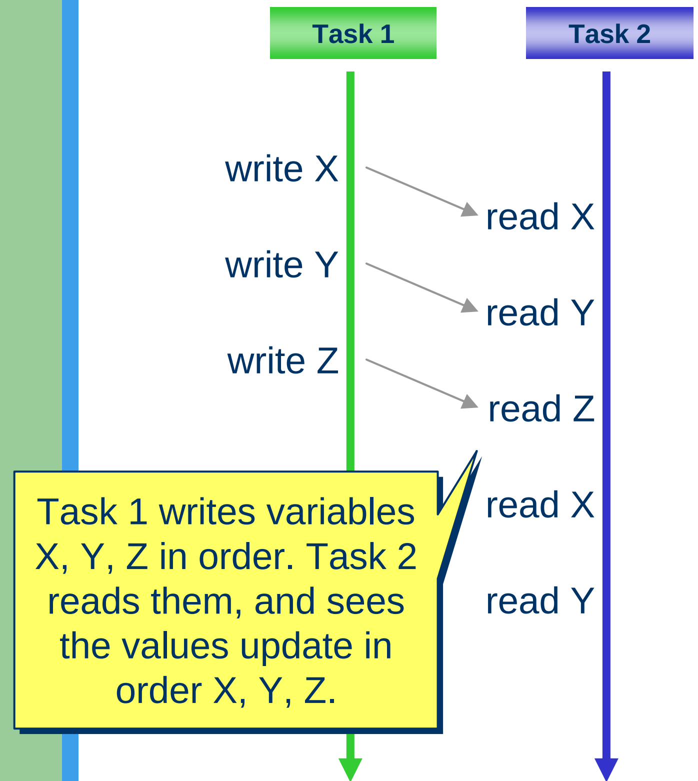
- Single processor: all memory operations will occur in program order*
  - * as observed by the program running
  - A read will always get the latest value written
  - Fundamental assumption used in writing code
- Multiprocessor: not so easy
  - Processors can see operations in different order
  - “**Weak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

Relaxed Memory Ordering: Why?

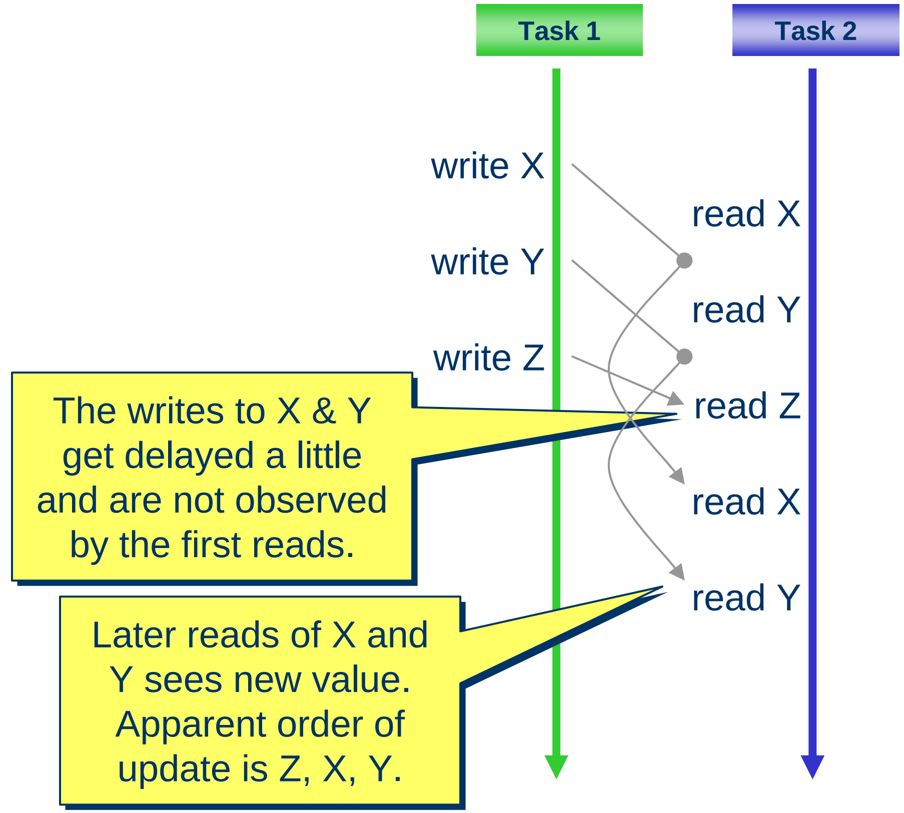
- Global unique ordering = needs synchronization
  - For almost every instruction executed
  - Would kill the performance of parallel computers
- Solution: specify some slack for the system
  - More slack = more opportunity to optimize
  - More slack = allow more reordering of writes & reads
  - More slack = more opportunity for weird bugs
- Exploited by *compilers*, *processors*, and the *memory system* to reduce stall time

Relaxed Memory Ordering: Example

- Expected obvious case



- Legal less obvious case



Disclaimer: This example is really brutally simplified. But it shows the gist of the problem.

Relaxed Memory Ordering: Problem

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

Relaxed Memory Ordering: Fixing

- Use SMP-aware synchronization
- Explicit data synchronization necessary

- Read up on the particular memory consistency of your target platform
  - ... and note that it is sometimes not implemented to its full freedom on current hardware ...

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 is hard
  - Parallel errors depend on subtle timing, interactions between tasks, precise order of events
- **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, but with several caveats:
  - Stopping one task in a collaborating group might break the system
  - A stopped task can be swamped with traffic
- Desired tool support for multiprocessors:
  - Synchronized stop of multiple processors
  - Understanding of multiple tasks
  - Inspection of multiple tasks

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...

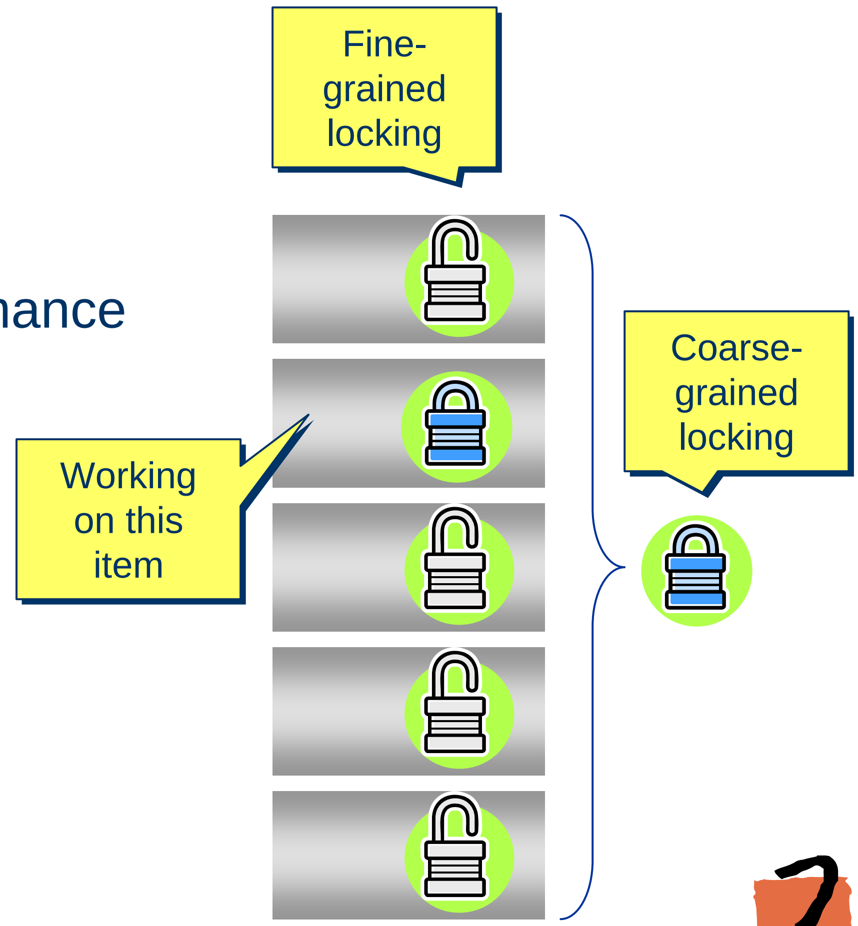
- Printf
  – Added by user to program
- Monitor task
  – Special task snooping on application, added by user
- Instrumentation
  – Source or binary level, added by tool
- Bus trace
  – Less meaningful in a heavily cached system

...Tracing Methods

- Hardware trace
  - Using trace support in hardware + trace buffer
  - Mostly non-intrusive
- Simulation
  - Can trace any aspect of system
  - Differences in timing, requires a simulation model

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



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

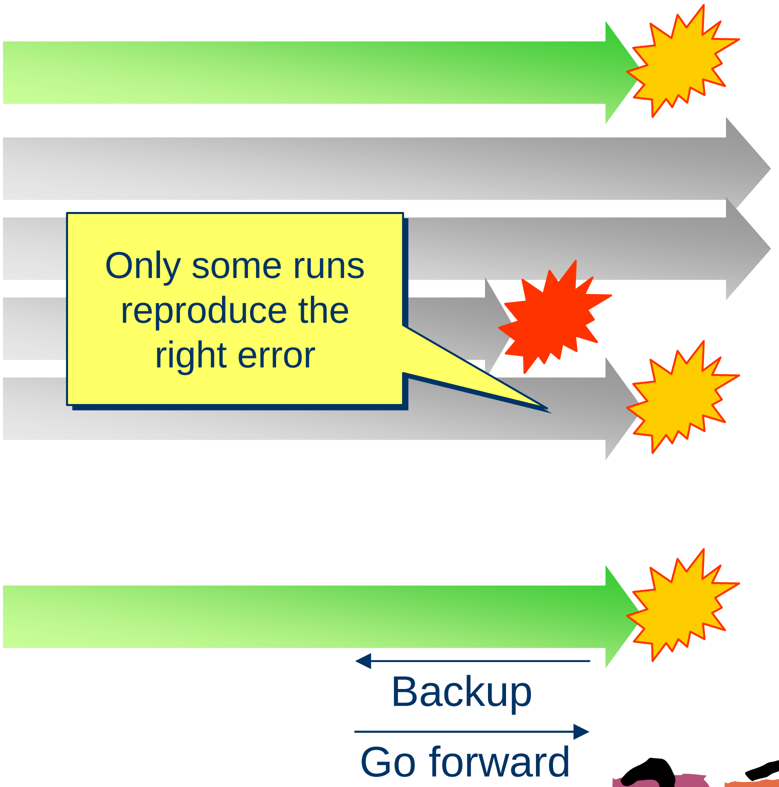
- Provokes errors by challenging assumptions
  - Different number of processors
  - Different speed of processors
  - Different communications latency & cache sizes
- It is easy to accidentally tie code to the machine the code is developed on

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

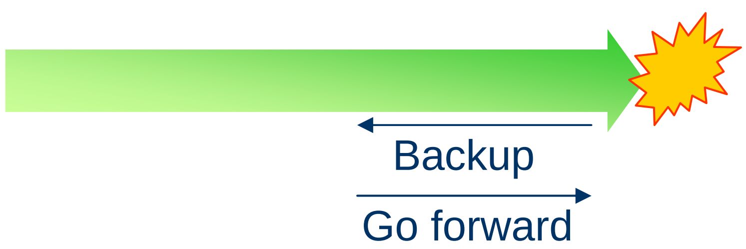
Reverse Debugging

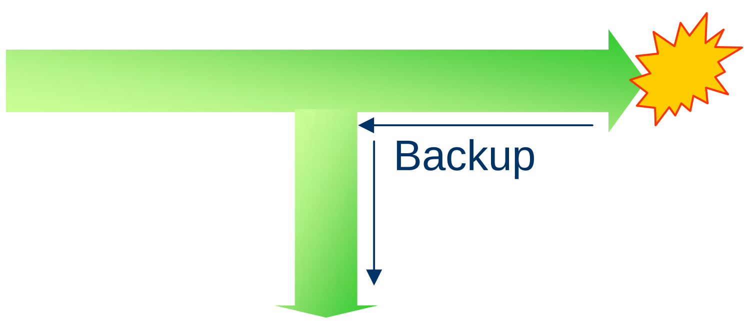
- Stop & go back in time
  - Instead of rerunning program from start
  - No need to rerun and hope for bug to reoccur
  - Investigate exactly what happened this time
  - Breakpoints & watchpoints backwards in time
  - Very powerful for parallel programs



Reverse Debugging: Techniques

- Trace-based
  - Record system execution
  - Special hardware support
  - Use as "tape recorder", fixed execution observed
- Simulation-based
  - Record in simulator
  - Replay in same simulator
  - Can change state and continue execution





Simulate the System

- Simulation offers control over a system
  - Vary parameters to provoke errors
  - Inject variations in execution to provoke errors
  - Reliable reproduction of problems
  - Powerful inspection abilities
  - No probe effect from tracing and breakpoints
  - Can support record & replay, and reverse debugging

Simulate the System: Modeling

- Simulation requires a model of a system
  - Need to run the same binaries as the real target
  - Processors + memories + timers + devices + IO
  - Several commercial tools available
- Simulation is never quite like the real thing
  - But close enough
  - Any bugs found in simulation are valid bugs
  - Precise timing simulation is not really possible

Formal Methods

- Static analysis tools
  - Analyze source code to determine properties
  - “Lint” for parallelism
- Dynamic analysis tools
  - Run a program, collect information, analyze
  - Check that a program follows certain rules
    - Locking discipline, for example
- Some tools exist

Questions?

Thank You!

Please remember to fill in the course evaluation forms!