

Parallelism and Concurrency

©Andrey Kruglyak, 2010

Performance

- ◁ How can we measure system performance?
- ◁ Two important metrics:
 - ◁ throughput (how many tasks are completed per time unit)
 - ◁ responsiveness (how fast does the system react to input)
 - ◁ the difference between the two becomes clear if we combine short "important" tasks (e.g. screen updates in response to user input) with longer "less important" ones (e.g. heavy calculations in the background)

Improving performance through parallelism

- ◁ Both throughput and responsiveness can be improved by utilizing parallelism provided by parallel hardware (multiple execution units, out-of-order execution, multiple cores, hyperthreading, multiple processors)
- ◁ Responsiveness can also be improved on a single CPU by switching between tasks after short periods of time (interleaving execution)

Parallelism

- ◁ Task-level parallelism:
 - ◁ tasks are independently defined, implemented and executed
 - ◁ each lives in its own virtual memory space and assumes to have an exclusive access to the underlying hardware
 - ◁ tasks rarely communicate, and such communication or sharing of data requires special setup and support from the OS

Parallelism

- ◀ Task-level parallelism
- ◀ Thread-level parallelism:
 - ◀ a thread of execution is basically a program counter; a multi-threaded program will have multiple program counters, i.e. multiple pointers into the same code
 - ◀ executing multiple threads means maintaining multiple program counters and hence (normally) separate memory stacks, registers, etc. for each thread
 - ◀ as threads are part of the same program, they often need to cooperate and share data (which leads to synchronization problems)

Wednesday, May 12, 2010

5

Parallelism

- ◀ Task-level parallelism
- ◀ Thread-level parallelism
- ◀ Instruction-level parallelism:
 - ◀ offered by hardware: multiple execution and memory units, out-of-order execution

Wednesday, May 12, 2010

6

Parallelism

- ◀ Task-level parallelism is well-understood and is visible for the user but the programmer does not need to care about it – task isolation is performed by the OS
- ◀ Instruction-level parallelism is well-understood and transparent for both the user and the programmer
- ◀ Thread-level parallelism has two major drawbacks:
 - ◀ it is often difficult to grasp the meaning of a multi-threaded program, which make multi-threaded code error-prone
 - ◀ parallel modification of parts of the same state may leave the system in an inconsistent state (race condition)

Wednesday, May 12, 2010

7

Race condition example

Thread 1

```
...  
temp = x  
...  
x = temp+1
```

Thread 2

```
temp = x  
...  
x = temp+1  
...
```

Here we increased x twice, so its value should be 2. However, with the order above the value will be 1.

Wednesday, May 12, 2010

8

Other challenges of using threads

- ◀ Threads may need to be synchronized to deliver a combined result or to deliver the results in a particular order
- ◀ Shared resources – often need to be locked before use, e.g. to make sure that interaction with external hardware complies with a certain protocol
 - ◀ Memory is the single most important shared resource; when memory needs to be allocated or freed, this has to happen on the process-wide basis, often resulting in an implementation bottleneck
 - ◀ Maintaining consistency of large data structures by locking the whole structure is extremely inefficient (transactional memory and lock-free data structures can help here)

Parallelism in languages

- ◀ However, multi-threading is essential if we want to write efficient (i.e. utilizing parallel hardware to speed up computations) and responsive (i.e. handling short important tasks fast) programs!
- ◀ This has led to a search for a good abstraction on top of threads that would be:
 - ◀ easy to understand
 - ◀ safe to use
- ◀ This clearly requires support in the programming language!

Visible and transparent parallelism

- ◀ It is important to distinguish two kinds of parallelism:
 - ◀ visible to the user, when the result depends on the order of computations – this parallelism can be either desirable (improving responsiveness without compromising correctness, e.g. parallel processing of independent inputs resulting in independent outputs) or undesirable (leading to incorrect behavior of the system)
 - ◀ transparent to the user, i.e. the result is the same for all orders of computations (e.g. $(3+4)+(5+6) = ((3+4)+5)+6=...$)

Parallelism and concurrency

- ◀ Concurrency is a more general and often more correct term than parallelism
- ◀ Parallel execution requires separate hardware, whereas concurrent execution is an umbrella term for parallel execution, interleaving execution, and sequential execution where the order of execution is not significant
- ◀ This we should speak about concurrency when we define correct behavior of a program, and about parallelism when we speak about implementation. However, “parallelism” is very often used to mean “concurrency”.
- ◀ We can say that each programming language, as well as some design and modeling frameworks, defines a certain concurrency model

Selected concurrency models

- ◁ Tasks & interrupt handlers in tinyOS
- ◁ Threads (Java, POSIX)
- ◁ Tasks (Ada)
- ◁ Active objects/actors (QNX, Ptolemy framework)
- ◁ Reactive objects (Timber)
- ◁ Time-triggered languages (Giotto)
- ◁ Synchronous languages (Esterel, SCADE, Lustre, Signal)
- ◁ Functional languages

Tasks and interrupt handlers in tinyOS

- ◁ tinyOS was developed for IO-centric systems
- ◁ Two-tiered scheduling:
 - ◁ tasks are executed one at a time (in FIFO order or according to some schedule)
 - ◁ interrupt handlers interrupt any task
- ◁ There is no blocking or communication between tasks or interrupt handlers
- ◁ Interrupt handlers may schedule tasks, and one task can schedule other tasks

POSIX and Java threads

- ◁ When the program starts, there is only one thread
- ◁ Other threads can be spawned explicitly by the programmer, a thread may also yield or join another thread (wait for it to complete)
- ◁ The (system-wide or JVM) scheduler will most probably switch between ready threads once in a while according to some scheduling policy, taking into account the threads' priorities
- ◁ All memory is accessible by all threads, and locks (mutexes, semaphores, monitors, barriers, etc.) can be used to protect memory consistency and enforce synchronization
- ◁ A thread may block waiting for input or access to a resource

Tasks in Ada

- ◁ Tasks are scheduled and executed one at a time
- ◁ When the program starts, all tasks are ready and the scheduler picks one of them for execution
- ◁ Inside the task, it is possible to delay execution and to block waiting for a message from another task; in both cases, the scheduler determines which task to execute next
- ◁ Tasks can make use of shared memory, and semaphores may be used to protect its consistency

Active and reactive objects

- ◁ Objects encapsulate state and communicate by sending messages
- ◁ A reactive object is idle until activated by an external event, a message, or a timer; each reaction is limited in time and never blocks
- ◁ An active object is always active, and may block waiting for messages or a timer event
 - ◁ in QNX, basic structures are threads that communicate by passing messages
 - ◁ in Ptolemy, basic structures are actors that can activate independent of their environment

Time-triggered languages

- ◁ Different blocks are assumed to accept and produce data at certain "clock ticks"
- ◁ Execution within each block is sequential and only local variables together with inputs are used for computing outputs
- ◁ No additional synchronization is required

Synchronous languages

- ◁ Vary in their basic structures
- ◁ Concurrency expressed in a program is eliminated by the compiler, resulting in a single-threaded execution at run time

Functional languages

- ◁ Are not suitable for expressing visible parallelism
- ◁ Pure computations can easily be parallelized as they are guaranteed not to produce side-effects (e.g. for call-by-value languages, computation of arguments can be done in parallel)
 - ◁ However, the great promise of functional languages delivering massively parallel systems has so far failed to materialize

Timber concurrency model

- ◁ Reactive objects are units of concurrency, i.e. only one method of each object can be active at any given time but methods of different objects can execute concurrently (implemented using POSIX threads in the Timber run-time system for POSIX). As methods may produce side-effects (state updates, output), this gives us visible parallelism.
- ◁ A Timber program should be structured in such a way that parallelism between objects is desirable, and when parallelism would be incorrect, the state and the methods are concentrated in one object – but this is the responsibility of the programmer!
- ◁ Concurrency originates from defining that different objects handle different external events; it can also be introduced explicitly by the programmer, via actions.

Timber concurrency model

- ◁ As functions are pure (no side-effects), it is always safe to parallelize them. This gives us transparent parallelism (not implemented in the compiler today)
- ◁ What Timber offers is
 - ◁ an intuitive way to structure data into objects, with consistency of data within an object automatically guaranteed
 - ◁ a highly concurrent implementation by means of an automatic parallelization of code placed in different objects
 - ◁ synchronization that can easily be expressed by two reactions involving (e.g. ending at) the same object

Transactional memory

- ◁ An attempt to simplify parallel programming by replacing locks with rollbacks
- ◁ A group of load/store operations can be declared atomic, parallel writing to the affected memory region is then detected and the operation is “rolled back” and re-tried
- ◁ Works because in most cases there will be no contention!
- ◁ Implementation:
 - ◁ in software (requires atomic compare-and-swap in hardware)
 - ◁ in hardware (e.g. Rock from Sun, 2007)

Lock-free data structures

- ◁ Reads have no locking at all
- ◁ Updates make a copy of the entire data structure, update the copy, then try to CAS it with the original. While the CAS operation does not succeed, the copy/update/CAS process is tried again in a loop