Concurrent Programming: Motivation, Theory, Practice

Emmett Witchel
First Bytes Teacher Conference
July 2008

Uniprocessor Performance Not Scaling

Graph by Dave Patterson
Power and heat lay waste to processor makers

  - 1.3GHz to 3.8GHz, 31 stage pipeline
  - “Prescott” in 02/04 was too hot. Needed 5.2GHz to beat 2.6GHz Athalon
  - Too much power

- Intel Pentium Core, (2006-)
  - 1.06GHz to 3GHz, 14 stage pipeline
  - Based on mobile (Pentium M) micro-architecture
    - Power efficient
  - Designed by small team in Israel

- 2% of electricity in the U.S. feeds computers
  - Doubled in last 5 years

---

What about Moore’s law?

- Number of transistors double every 24 months
  - Not performance!
Architectural trends that favor multicore

- Power is a first class design constraint
  - Performance per watt the important metric
- Leakage power significant with small transistors
  - Chip dissipates power even when idle!
- Small transistors fail more frequently
  - Lower yield, or CPUs that fail?
- Wires are slow
  - Light in vacuum can travel ~1m in 1 cycle at 3GHz
- Quantum effects
- Motivates multicore designs (simpler, lower-power cores)

Multicore are here, and coming fast!

4 cores in 2008  16 cores in 2009  80 cores in 20??

AMD Quad Core   Sun Rock   Intel TeraFLOP

"[AMD] quad-core processors … are just the beginning....”
http://www.amd.com

"Intel has more than 15 multi-core related projects underway”
http://www.intel.com
Houston, We have a problem!

- Running multiple programs only goes so far
- How does one application take advantage of multiple cores?
  - Parallel programming is a hard problem

- Even systems programmers find it challenging
  “Blocking on a mutex is a surprisingly delicate dance”
  — OpenSolaris, mutex.c

- What about Visual Basic programmers?
  “The distant threat has come to pass.....parallel computers are the inexorable next step in the evolution of computers.”
  — James Larus, Microsoft Research
  In Transactional Memory,

What’s hard about parallel programming?

- Answer #1: Little experience
  - Most application programmers have never written or substantially modified a significant parallel program

- Answer #2: Poor programming models
  - Primitive synchronization mechanisms
  - Haven’t changed significantly in 50 years

- Answer #3: People think sequentially
  - Programming models approximate sequential execution
Application performance with more processors

- Not scalable: Most current programs
- Moderate: The hope for the future
- Scalable: Scientific codes, some graphics, server workloads

Processes

Process Management
What is a Process?

- A process is a program during execution.
  - Program = static executable file (image)
  - Process = executing program = program + execution state.

- A process is the basic unit of execution in an operating system

- Different processes may run several instances of the same program

- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

Program to Process

- We write a program in e.g., Java.
- A compiler turns that program into an instruction list.
- The CPU interprets the instruction list (which is more a graph of basic blocks).

```java
void X (int b) {
    if(b == 1) {
      ...
      int main() {
        int a = 2;
        X(a);
    }
```
Process in Memory

- Program to process.
- What you wrote

```c
void X (int b) {
  if(b == 1) {
    ...
      int main() {
        int a = 2;
        X(a);
      }
  }
}
```

- What must the OS track for a process?

What is in memory.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Heap</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>main; a = 2</td>
<td>X; b = 2</td>
<td>void X (int b)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if(b == 1) {</td>
</tr>
</tbody>
</table>
|               |               |       ...
|               |               |     int main() {
|               |               |       int a = 2;
|               |               |       X(a);
|               |               |   }           |

Keeping track of a process

- A process has code.
  - OS must track program counter (code location).
- A process has a stack.
  - OS must track stack pointer.
- OS stores state of processes' computation in a process control block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB.
Anatomy of a Process

Executable File

Process's address space

mapped segments

DLL's

Stack

Heap

Initialized data

Code

Process Control Block

PC

Stack Pointer

Registers

PID

UTD

Scheduling Priority

List of open files...

Process Life Cycle

- Processes are always either **executing, waiting to execute** or waiting for an event to occur

- A preemptive scheduler will force a transition from running to ready. A non-preemptive scheduler waits.
From Processes to Threads

Processes, Threads and Processors

- **Hardware can interpret N instruction streams at once**
  - Uniprocessor, N==1
  - Dual-core, N==2
  - Sun’s Niagara 2 (2008) N == 64, but 8 groups of 8
- **An OS can run 1 process on each processor at the same time**
  - Concurrent execution increases throughput
- **An OS can run 1 thread on each processor at the same time**
  - Do multiple threads reduce latency for a given application?
Processes and Threads

- Process abstraction combines two concepts
  - Concurrency
    Each process is a sequential execution stream of instructions
  - Protection
    Each process defines an address space
    Address space identifies all addresses that can be touched by the program

- Threads
  - Key idea: separate the concepts of concurrency from protection
  - A thread is a sequential execution stream of instructions
  - A process defines the address space that may be shared by multiple threads

Introducing Threads

- A thread represents an abstract entity that executes a sequence of instructions
  - It has its own set of CPU registers
  - It has its own stack
  - There is no thread-specific heap or data segment (unlike process)

- Threads are lightweight
  - Creating a thread more efficient than creating a process.
  - Communication between threads easier than btw. processes.
  - Context switching between threads requires fewer CPU cycles and memory references than switching processes.
  - Threads only track a subset of process state (share list of open files, mapped memory segments ...)

- Examples:
  - OS-level: Windows threads, Sun's LWP, POSIX's threads
  - User-level: Some JVMs
  - Language-supported: Modula-3, Java
Context switch time for which entity is greater?

1. Process
2. Thread

Programmer’s View

```c
void fn1(int arg0, int arg1, ...) {...}

main() {
    ...
    tid = CreateThread(fn1, arg0, arg1, ...);
    ...
}
```

At the point `CreateThread` is called, execution continues in parent thread in main function, and execution starts at `fn1` in the child thread, *both in parallel (concurrently)*.
**Threads vs. Processes**

**Threads**
- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed
- Inter-thread communication via memory.

**Processes**
- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die
- Inter-process communication via OS and data copying.

---

**Implementing Threads**

- Processes define an address space; threads share the address space

- Process Control Block (PCB) contains process-specific information
  - Owner, PID, heap pointer, priority, active thread, and pointers to thread information

- Thread Control Block (TCB) contains thread-specific information
  - Stack pointer, PC, thread state (running, ...), register values, a pointer to PCB, ...
Threads have the same scheduling states as processes

1. True
2. False

Threads’ Life Cycle

- Threads (just like processes) go through a sequence of start, ready, running, waiting, and done states
How Can it Help?

- How can this code take advantage of 2 threads?
  
  ```
  for(k = 0; k < n; k++)
  a[k] = b[k] * c[k] + d[k] * e[k];
  ```

- Rewrite this code fragment as:

  ```
  do_mult(l, m) {
      for(k = l; k < m; k++)
      a[k] = b[k] * c[k] + d[k] * e[k];
  }
  main() {
      CreateThread(do_mult, 0, n/2);
      CreateThread(do_mult, n/2, n);
  }
  ```

- What did we gain?

---

How Can it Help?

- Consider a Web server
  Create a number of threads, and for each thread do
  - get network message (URL) from client
  - get URL data from disk
  - send data over network

- Why does creating multiple threads help?
Overlapping Requests (Concurrency)

Request 1
Thread 1
- get network message (URL) from client
- get URL data from disk
  (disk access latency)
- send data over network

Request 2
Thread 2
- get network message (URL) from client
- get URL data from disk
  (disk access latency)
- send data over network

Total time is less than request 1 + request 2

Latency and Throughput

- Latency: time to complete an operation
- Throughput: work completed per unit time
- Multiplying vector example: reduced latency
- Web server example: increased throughput
- Consider plumbing
  - Low latency: turn on faucet and water comes out
  - High bandwidth: lots of water (e.g., to fill a pool)
- What is “High speed Internet?”
  - Low latency: needed to interactive gaming
  - High bandwidth: needed for downloading large files
  - Marketing departments like to conflate latency and bandwidth...
Relationship between Latency and Throughput

- Latency and bandwidth only loosely coupled
  - Henry Ford: assembly lines increase bandwidth without reducing latency
- Latency reduction is difficult
- Often, one can buy bandwidth
  - E.g., more memory chips, more disks, more computers
  - Big server farms (e.g., google) are high bandwidth

Thread or Process Pool

- Creating a thread or process for each unit of work (e.g., user request) is dangerous
  - High overhead to create & delete thread/process
  - Can exhaust CPU & memory resource
- Thread/process pool controls resource use
  - Allows service to be well conditioned.
Thread Synchronization:
Too Much Milk

Concurrency Problems, Real Life Example

- Imagine multiple chefs in the same kitchen
  - Each chef follows a different recipe
- Chef 1
  - Grab butter, grab salt, do other stuff
- Chef 2
  - Grab salt, grab butter, do other stuff
- What if Chef 1 grabs the butter and Chef 2 grabs the salt?
  - Yell at each other (not a computer science solution)
  - Chef 1 grabs salt from Chef 2 (preempt resource)
  - Chefs all grab ingredients in the same order
    - Current best solution, but difficult as recipes get complex
    - Ingredient like cheese might be sans refrigeration for a while
The Need For Mutual Exclusion

- Running multiple processes/threads in parallel increases performance
- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can’t print two documents at once
- Mutual exclusion is the term to indicate that some resource can only be used by one thread at a time
  - Active thread excludes its peers
- For shared memory architectures, data structures are often mutually exclusive
  - Two threads adding to a linked list can corrupt the list

Sharing among threads increases performance...

```c
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the value of `a` & `b` at the end of execution?
... But it can lead to problems!!

```c
int a = 1, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = 0;
}
What are the values of a & b at the end of execution?
```

Some More Examples

- What are the possible values of $x$ in these cases?

  - Thread1: $x = 1$; Thread2: $x = 2$;

    Initially $y = 10$;
    Thread1: $x = y + 1$; Thread2: $y = y * 2$;

    Initially $x = 0$;
    Thread1: $x = x + 1$; Thread2: $x = x + 2$;
Concurrency Problem

- **Order of thread execution is non-deterministic**
  - **Multiprocessing**
    - A system may contain multiple processors ➜ cooperating threads/processes can execute simultaneously
  - **Multi-programming**
    - Thread/process execution can be interleaved because of time-slicing
- **Operations are often not “atomic”**
  - **Example:** $x = x + 1$ is not atomic!
    - read $x$ from memory into a register
    - increment register
    - store register back to memory
- **Goal:**
  - Ensure that your concurrent program works under ALL possible interleaving

The Fundamental Issue

- In all these cases, what we thought to be an *atomic* operation is not done atomically by the machine
  - An atomic operation is all or nothing:
    - Either it executes to completion, or
    - it did not execute at all, and
    - partial progress is not visible to the rest of the system
Are these operations usually atomic?

- Writing an 8-bit byte to memory
  - True (is atomic)
  - False

- Creating a file
  - True
  - False

- Writing a disk 512-byte disk sector
  - True
  - False

Critical Sections

- A critical section is an abstraction that
  - consists of a number of consecutive program instructions
  - all code within the section executes atomically

- Critical sections are used frequently in an OS to protect
  data structures (e.g., queues, shared variables, lists, ...)

- A critical section implementation must be:
  - **Correct**: for a given k, only k threads can execute in the
    critical section at any given time (usually, k = 1)
  - **Efficient**: getting into and out of critical section must be
    fast. Critical sections should be as short as possible.
  - **Concurrency control**: a good implementation allows
    maximum concurrency while preserving correctness
  - **Flexible**: a good implementation must have as few
    restrictions as practically possible
Safety and Liveness

- **Safety property**: “nothing bad happens”
  - holds in every finite execution prefix
    - Windows™ never crashes
    - a program never produces a wrong answer

- **Liveness property**: “something good eventually happens”
  - no partial execution is irremediable
    - Windows™ always reboots
    - a program eventually terminates

- Every property is a combination of a safety property and a liveness property - (Alpern and Schneider)

Safety and liveness for critical sections

- At most k threads are concurrently in the critical section
  - A. Safety
  - B. Liveness
  - C. Both

- A thread that wants to enter the critical section, will eventually succeed
  - A. Safety
  - B. Liveness
  - C. Both

- Bounded waiting: If a thread /is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread /is request is granted
  - A. Safety
  - B. Liveness
  - C. Both
Critical Section: Implementation

- **Basic idea:**
  - Restrict programming model
  - Permit access to shared variables only within a critical section

- **General program structure**
  - **Entry section**
    - "Lock" before entering critical section
    - Wait if already locked
    - Key point: synchronization may involve wait
  - **Critical section code**
  - **Exit section**
    - "Unlock" when leaving the critical section

- **Object-oriented programming style**
  - Associate a lock with each shared object
  - Methods that access shared object are critical sections
  - Acquire/release locks when entering/ exiting a method that defines a critical section

---

Thread Coordination

Too much milk!

**Jack**
- Look in the fridge; out of milk
- Leave for store
- Arrive at store
- Buy milk
- Arrive home; put milk away

**Jill**
- Look in fridge; out of milk
- Leave for store
- Arrive at store
- Buy milk
- Arrive home; put milk away
- Oh, no!

Fridge and milk are shared data structures
Formalizing “Too Much Milk”

- **Shared variables**
  - “Look in the fridge for milk” - check a variable
  - “Put milk away” - update a variable

- **Safety property**
  - At most one person buys milk

- **Liveness**
  - Someone buys milk when needed

- **How can we solve this problem?**

Introducing Locks

- **Locks** - an API with two methods
  - Lock::Acquire() - wait until lock is free, then grab it
  - Lock::Release() - release the lock, waking up a waiter, if any

- With locks, too much milk problem is very easy!

```cpp
Lock->Acquire();
if (noMilk) {
    buy milk;
}
Lock->Release();
```

How can we implement locks?
Atomic Read-Modify-Write (ARMW)

- For uni- and multi-processor architectures: implement locks using atomic read-modify-write instructions
  - Atomically
    1. read a memory location into a register, and
    2. write a new value to the location
  - Implementing ARMW is tricky in multi-processors
    Requires cache coherence hardware. Caches snoop the memory bus.

- Examples:
  - Test&set instructions (most architectures)
    Reads a value from memory
    Write "1" back to memory location
  - Compare & swap (68000), exchange (x86), ...
    Test the value against some constant
    If the test returns true, set value in memory to different value
    Report the result of the test in a flag
    if [addr] == r1 then [addr] = r2;

Using Locks Correctly

- Make sure to release your locks along every possible execution path.
  
  ```c
  unsigned long flags;
  local_irq_save( flags );  // Disable & save
  ...
  if(somethingBad) {
    local_irq_restore( flags );
    return ERROR_BAD_THING;
  }
  ...
  local_irq_restore( flags );  // Reenable
  return 0;
  ```
Using Locks Correctly

- Java provides convenient mechanism.
  
  ```java
  import java.util.concurrent.locks.ReentrantLock;
  
  aLock.lock();
  try {
      ...
  } finally {
      aLock.unlock();
  }
  return 0;
  ```

Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks

- Lock implementation generally requires some level of hardware support
  - Atomic read-modify-write instructions
    Uni- and multi-processor architectures

- Locks are good for mutual exclusion but weak for coordination, e.g., producer/consumer patterns.
### Why Locks are Hard

- **Coarse-grain locks**
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

- **Fine-grain locks**
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
  - Not composable
  - Potential data races
  - Which lock to lock?

```c
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
    LOCK(s);
    LOCK(d);
    tmp = s.remove(key);
    d.insert(key, tmp);
    UNLOCK(d);
    UNLOCK(s);
}
```

**DEADLOCK!**

- **Thread 0**
  - move(a, b, key1);

- **Thread 1**
  - move(b, a, key2);

---

### Monitors & Condition Variables

- **Three operations**
  - **Wait()**
    - Release lock
    - Go to sleep
    - Reacquire lock upon return
  - **Notify()** (historically called Signal())
    - Wake up a waiter, if any
  - **NotifyAll()** (historically called Broadcast())
    - Wake up all the waiters

- **Implementation**
  - Requires a per-condition variable queue to be maintained
  - Threads waiting for the condition wait for a notify()

- **Butler Lampson and David Redell, “Experience with Processes and Monitors in Mesa.”**
Summary

- Non-deterministic order of thread execution ➔ concurrency problems
  - Multiprocessing
    A system may contain multiple processors ➔ cooperating threads/processes can execute simultaneously
  - Multi-programming
    Thread/process execution can be interleaved because of time-slicing
- Goal: Ensure that your concurrent program works under ALL possible interleaving
- Define synchronization constructs and programming style for developing concurrent programs
  Locks ➔ provide mutual exclusion
  Condition variables ➔ provide conditional synchronization

More Resources

- Sun’s Java documentation
  ➢ http://java.sun.com/javase/6/docs/api/
  ➢ http://java.sun.com/docs/books/tutorial/essential/concurrency/
- Concurrent Programming in Java: Design Principles and Patterns by Doug Lea (ISBN-10: 0201310090)