Concurrent Programming with Threads: Why you should care deeply

Don Porter

Portions courtesy Emmett Witchel
Uniprocessor Performance Not Scaling

Performance (vs. VAX-11/780)

- 25% / year
- 52% / year
- 20% / year

Graph by Dave Patterson
Power and Heat Lay Waste to CPU Makers

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

- Intel Pentium Core, (2006-)
  - 1.06GHz to 3GHz, 14 stage pipeline
  - Based on mobile (Pentium M) micro-architecture
    - Power efficient

- 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!
Transistor Budget

• We have an increasing glut of transistors
  – (at least for a few more years)

• But we can’t use them to make things faster
  – Techniques that worked in the 90s blew up heat faster than we can dissipate it

• What to do?
  – Use the increasing transistor budget to make more cores!
Multi-Core is Here: Plain and Simple

• Raise your hand if your laptop is single core?
• Your phone?

• That’s what I thought
Multi-Core Programming == Essential Skill

- Hardware manufacturers betting big on multicore
- Software developers are needed
- Writing concurrent programs is not easy
- You will learn how to do it in this class

Still treated like a bonus: Don’t graduate without it!
Threads: OS Abstraction for Concurrency

• 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
  – Threads can execute on different cores on a multicore CPU (parallelism for performance) and can communicate with other threads by updating memory
Practical Difference

• With processes, you coordinate through nice abstractions (relatively speaking – e.g., lab 1)
  – Pipes, signals, etc.

• With threads, you communicate through data structures in your process virtual address space
  – Just read/write variables and pointers
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)
Implementing Threads: Example Redux

Virtual Address Space

- 2 threads requires 2 stacks in the process
- No problem!
- Kernel can schedule each thread separately
  - Possibly on 2 CPUs
  - Requires some extra bookkeeping
How can it help?

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

- Rewrite this code fragment as:
  ```c
  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 Threads Help?

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

• What did we gain?
Overlapping I/O and Computation

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
Why threads? (summary)

• Computation that can be divided into concurrent chunks
  – Execute on multiple cores: reduce wall-clock exec. time
  – Harder to identify parallelism in more complex cases

• Overlapping blocking I/O with computation
  – If my web server blocks on I/O for one client, why not work on another client’s request in a separate thread?
  – Other abstractions we won’t cover (e.g., events)
## Threads vs. Processes

<table>
<thead>
<tr>
<th><strong>Threads</strong></th>
<th><strong>Processes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>A thread has no data segment or heap</td>
<td>A process has code/data/heap &amp; other segments</td>
</tr>
<tr>
<td>A thread cannot live on its own, it must live within a process</td>
<td>There must be at least one thread in a process</td>
</tr>
<tr>
<td>There can be more than one thread in a process, the first thread calls main &amp; has the process’s stack</td>
<td>Threads within a process share code/data/heap, share I/O, but each has its own stack &amp; registers</td>
</tr>
<tr>
<td>If a thread dies, its stack is reclaimed</td>
<td>If a process dies, its resources are reclaimed &amp; all threads die</td>
</tr>
<tr>
<td>Inter-thread communication via memory.</td>
<td>Inter-process communication via OS and data copying.</td>
</tr>
<tr>
<td>Each thread can run on a different physical processor</td>
<td>Each process can run on a different physical processor</td>
</tr>
<tr>
<td>Inexpensive creation and context switch</td>
<td>Expensive creation and context switch</td>
</tr>
</tbody>
</table>
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, ...
Thread Life Cycle

- Threads (just like processes) go through a sequence of *start*, *ready*, *running*, *waiting*, and *done* states
Threads have their own...?

1. CPU
2. Address space
3. PCB
4. Stack
5. Registers

Smiley face emojis next to 4 and 5 denote a positive or neutral sentiment.
Threads have the same scheduling states as processes

1. True😊
2. False

- In fact, OSes generally schedule *threads* to CPUs, not processes

Yes, yes, another white lie in this course
Lecture Outline

• What are threads?
• Small digression: Performance Analysis
  – There will be a few more of these in upcoming lectures
• Why are threads hard?
Performance: Latency vs. 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…
Latency and Throughput

• Latency and bandwidth only loosely coupled
  – Henry Ford: assembly lines increase bandwidth without reducing latency

• My factory takes 1 day to make a Model-T ford.
  – But I can start building a new car every 10 minutes
  – At 24 hrs/day, I can make $24 \times 6 = 144$ cars per day
  – A special order for 1 green car, still takes 1 day
  – Throughput is increased, but latency is not.

• 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
Latency, Throughput, and Threads

- Can threads improve throughput?
  - Yes, as long as there are parallel tasks and CPUs available

- Can threads improve latency?
  - Yes, especially when one task might block on another task’s IO

- Can threads harm throughput?
  - Yes, each thread gets a time slice.
  - If # threads >> # CPUs, the % of CPU time each thread gets approaches 0

- Can threads harm latency?
  - Yes, especially when requests are short and there is little I/O

Threads can help or hurt: Understand when they help!
So Why are Threads Hard?

• 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 often consist of multiple, visible steps
  – Example: \( x = x + 1 \) is not a single operation
    • read \( x \) from memory into a register
    • increment register
    • store register back to memory

• Goal:
  – Ensure that your concurrent program works under ALL possible interleavings
Questions

• Do the following either completely succeed or completely fail?
  • Writing an 8-bit byte to memory
    – A. Yes B. No
  • Creating a file
    – A. Yes B. No
  • Writing a 512-byte disk sector
    – A. Yes B. No
Sharing Amongst Threads Increases Performance

```c
int a = 0, b = 2;
main() {
    CreateThread(fn1, 4);
    CreateThread(fn2, 5);
}
fn1(int arg1) {
    if(a) b++;
}
fn2(int arg1) {
    a = arg1;
}
```

What are the values of a & b at the end of execution?

But can lead to problems...
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: $y = y \times 2$;

Initially $x = 0$;
Thread1: $x = x + 1$; Thread2: $x = x + 2$;
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
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
Critical Sections

• Key abstraction: A group of instructions that cannot be interleaved

• Generally, critical sections execute under mutual exclusion
  – E.g., a critical section is the part of the recipe involving butter and salt – you know, the important part

• One critical section may wait for another
  – Key to good multi-core performance is minimizing the time in critical sections
    • While still rendering correct code!
The Need to Wait

- Very often, synchronization consists of one thread waiting for another to make a condition true
  - Master tells worker a request has arrived
  - Cleaning thread waits until all lanes are colored
- Until condition is true, thread can sleep
  - Ties synchronization to scheduling
- Mutual exclusion for data structure
  - Code can wait (wait)
  - Another thread signals (notify)
Example 2: Traverse a singly-linked list

• Suppose we want to find an element in a singly linked list, and move it to the head

• Visual intuition:

```
  lhead

  lprev  lptr
```

![Diagram of singly-linked list with arrows pointing to lhead and lptr connections](image-url)
Example 2: Traverse a singly-linked list

• Suppose we want to find an element in a singly linked list, and move it to the head

• Visual intuition:

```
  lhead
  └── lprev ─── lptr ─── ...
```

Even more real life, linked lists

lprev = NULL;
for(lptr = lhead; lptr; lptr = lptr->next) {
    if(lptr->val == target) {
        // Already head?, break
        if(lprev == NULL) break;
        // Move cell to head
        lprev->next = lptr->next;
        lptr->next = lhead;
        lhead = lptr;
        break;
    }
    lprev = lptr;
}

• Where is the critical section?
Even more real life, linked lists

Thread 1

// Move cell to head
lpnext->next = lptr->next;
lptr->next = lhead
lhead = lptr;

A critical section often needs to be larger than it first appears

– The 3 key lines are not enough of a critical section
Even more real life, linked lists

Thread 1

if (lptr->val == target) {
    elt = lptr;
    // Already head?, break
    if (lprev == NULL) break;
    // Move cell to head
    lprev->next = lptr->next;
    // lptr no longer in list
}

Thread 2

for (lptr = lhead; lptr;
    lptr = lptr->next) {
    if (lptr->val == target) {

• Putting entire search in a critical section reduces concurrency, but it is safe.
Safety and Liveness

- **Safety property**: “nothing bad happens”
  - holds in every finite execution prefix
    - Windows™ never crashes
    - a program never terminates with 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 \( i \) is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section (only 1 thread is allowed in at a time) before thread \( i \)'s request is granted.
  - A. Safety
  - B. Liveness
  - C. Both
Lecture Summary

• Understand the distinction between process & thread
• Understand motivation for threads
• Concepts of Throughput vs. Latency
• Intuition of why coordinating threads is hard
• Idea of mutual exclusion and critical sections
  – Much more on last two points to come