Ken Birman

Moore’s Law and CPU speed

Transistor count still rising according to Moore’s Law
Clock speed flattening

2/20/2009 Transactional Memory: Part I — P. Felber

What this tells us?

- It is getting harder to just speed up a chip
  - One issue: the wires on the chip get very thin
  - But more serious: heat becomes a problem

- Heat dissipated by a chip rises roughly as the square of the clock speed!
  - So: if the clock is four times as fast...
  - ...the chip produces sixteen times as much heat!

- Chips are at risk of burning up

Multicores are the answer

- Multicores are the answer to keeping up with increasing CPU performance despite:
  - The memory wall (gap between CPU and memory speeds)
  - The ILP wall (not enough instruction-level parallelism to keep the CPU busy)
  - The power wall (higher clock speeds require more power and create thermal problems)

- Consequence:
  - Single-thread performance doesn't improve...
  - ...but we can put more cores on a chip

2/20/2009 Transactional Memory: Part I — P. Felber

Why?

- Suppose you have four cores and each of them runs at 1/4th the speed of some hypothetical fast chip
  - Your power consumption turns out to be less than four times the power consumption of one slow chip
  - Because they share a lot of hardware
  - And a fast chip would generate easily ten times as much heat as four slow chips, to do the same amount of work

- But there's one “issue”
  - Programs need to use all those processors!

Multicores are everywhere

- Dual-core commonplace in laptops
- Quad-core in desktops
- Dual quad-core in servers
- All major chip manufacturers produce multicore CPUs
  - SUN Niagara (8 cores, 32 concurrent threads)
  - Intel Xeon (4 cores)
  - AMD Opteron (4 cores)
  - ...
Threads
- On Tuesday we discussed processes that run in separate address spaces
  - Two processes can execute the same program, but they will have distinct registers, stacks, data regions. Only the code is shared.
  - A thread (sometimes called a lightweight process) runs within a single process. Each thread
    - Has its own registers and its own stack
    - Threads share data with other threads in same process

Applications that use threads
- Web browsers
  - Each frame (mini-window) might have its own thread
- Web servers
  - Each request could be done with a separate thread.
- Graphics or gaming application
  - Each object could be rendered by its own thread

Threads
- With threads we can write programs that
  - Stay busy when doing input/output (I/O) operations
  - Exploit multicore parallelism
- But exploiting threads isn’t trivial
Case for Parallelism

Consider the following code fragment

```plaintext
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
CreateProcess(fn, 0, n/2);
CreateProcess(fn, n/2, n);
fn(l, m)
    for(k = l; k < m; k++)
        a[k] = b[k] * c[k] + d[k] * e[k];
```

Case for Parallelism

Consider a Web server

create a number of process, and for each process do

- get network message from client
- get URL data from disk
- compose response
- send response

Threads and Processes

- Most operating systems adopt the view that processes are containers in which threads execute
- Initially, a process is created with one thread
  - It runs the main() procedure
- But you can create additional threads
  - They start by running any procedure you like
  - Then they can create additional threads, or terminate

  - A process is expensive to create (fork/exec)
  - A thread is very cheap to create

Thread Creation: Java syntax

Create a new thread, start it. It will be initialized via a call to the constructor and then later, but we can't predict precisely when, run() will be invoked.

```java
public class Main {
    Thread A = new ThreadDemo("my name is A");
    Thread B = new ThreadDemo("my name is B");
    A.start();
    B.start();
}
```

```java
public class ThreadDemo extends Thread {
    private final String name;
    public ThreadDemo(String s) {
        this.name = s;
    }
    public void run() {
        System.out.println("Thread <"+this.name+"> is running");
    }
}
```

Thread Creation in Java

Once "started", the "run" procedure will be invoked

```java
public void run() {
    System.out.println("Thread <"+this.name+"> is running");
}
```
Multithreaded Processes

Cooperative (non-preemptive) threads

Cooperative threads use non-preemptive scheduling
  • Definition: one thread runs at a time, until it passes by “yielding” the CPU
  • Advantages:
    • Simple
    • Scientific apps
  • Disadvantages:
    • For badly written code
    • Scheduler gets invoked only when Yield is called
    • A thread should yield the processor when it blocks for I/O, e.g. to read from a file or the keyboard

Non-Cooperative Threads

User-Level Threads

User-Level Threads

For speed, implement threads at the user level
A user-level thread is managed by the run-time system
Each thread is represented simply by:
  • PC
  • Registers
  • Stack
  • Small control block
All thread operations are at the user-level:
  • Creating a new thread
  • Switching between threads
  • Synchronizing between threads

Non-Cooperative Threads

Cooperative Threads

• Cooperative threads use non-preemptive scheduling
  • Definition: one thread runs at a time, until it passes by “yielding” the CPU

• Advantages:
  • Simple
  • Scientific apps

• Disadvantages:
  • For badly written code
  • Scheduler gets invoked only when Yield is called
  • A thread should yield the processor when it blocks for I/O, e.g. to read from a file or the keyboard

User-Level Threads

User-level threads
  • the thread scheduler is part of a library, outside the kernel
  • thread context switching and scheduling is done by the library
  • Can either use cooperative or pre-emptive threads
    • cooperative threads are implemented by:
      • CreateThread(), DestroyThread(), Yield(), Suspend(), etc.
    • pre-emptive threads are implemented with a timer (signal)
      • where the timer handler decides which thread to run next
Example User Thread Interface

User thread interface:
- `t = thread_fork(initial context)`
  - Create a new thread of control
- `thread_stop()`
  - Stop the calling thread, sometimes called thread_block
- `thread_start(t)`
  - Start the named thread
- `thread_yield()`
  - Voluntarily give up the processor
- `thread_exit()`
  - Terminate the calling thread, sometimes called thread_destroy

Kernel threads

- User threads can run with even a single CPU
- We simply use context switching when one blocks and another starts, just as the kernel does to run a process
- But modern machines often have multiple cores
- This leads to the idea of a kernel thread
  - Basically, a thread with its own core (CPU)
- User sees the same thread API, but each kernel thread can switch among some set of user threads

Key Data Structures

- Your process address space:
  - Your programs:
    ```
    for i (1, 10, I++)
    thread_fork();
    ```
  - Your data (shared by all your threads):
    ```
    queue of thread control blocks
    ```
  - User-level thread code:
    ```
    proc thread_fork();...
    proc thread_block();...
    proc thread_exit();...
    ```
  - Per-thread stacks

User-Level vs. Kernel Threads

User-Level
- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

Kernel-Level
- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

Key issue: Kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will block even if the program logic allows them to proceed

Common tradeoffs

- Threads do have costs
  - Especially, the stack space required
  - With many threads, this becomes a serious burden
- Modern machines have multiple cores... but rarely more than 2. At most 8 or 16.
  - So we can't just have one kernel thread per user thread
  - Context switching costs can become a big overhead

Thread Hazards: Much like our race condition from last week!

```c
int a = 1, b = 2, w = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 4);
    while(w) ;
} fn() {
    int v = a + b;
    w--;
}
```
Concurrency Problems
A statement like w-- in C (or C++) is implemented by several machine instructions:

```
mov w,r4
sub $1,r4
mov r4,w
```

Now, imagine the following sequence, what is the value of w?

```
mov w,r4
mov r4,-(sp)
mov w,r4
sub $-1,w
mov r4,w
mov (sp)+,r4
```

Threads share global memory
* When a process contains multiple threads, they have
  * Private registers and stack memory (the context switching mechanism needs to save and restore registers when switching from thread to thread)
  * Shared access to the remainder of the process “state”
* This can result in race conditions

Two threads, one counter
Popular web server
* Uses multiple threads to speed things up.
* Simple shared state error:
  * each thread increments a shared counter to track number of hits
  ```
  hits = hits + 1;
  ... 
  ```
* What happens when two threads execute concurrently?

Shared counters
* Possible result: lost update!
  ```
  hits = 0
  time
  read hits (0)
  T1
  hits = 0 + 1
  read hits (0)
  T2
  hits = 1
  ```
* One other possible result: everything works.
  ⇒ Difficult to debug
* Called a “race condition”

Race conditions: Improved defn
* A timing dependent error involving shared state
  * Whether it happens depends on how threads scheduled
  * In effect, once thread A starts doing something, it needs to “race” to finish it because if thread B looks at the shared memory region before A is done, it may see something inconsistent
* Hard to detect:
  * All possible schedules have to be safe
  * Number of possible schedule permutations is huge
  * Some bad schedules? Some that will work sometimes?
  * they are intermittent
  * Timing dependent = small changes can hide bug

Scheduler assumptions
* Process a:
  ```
  while(i < 10)
  i = i + 1;
  print "A won!";
  ```
* Process b:
  ```
  while(i > -10)
  i = i - 1;
  print "B won!";
  ```
* If i is shared, and initialized to 0
  * Who wins?
  * Is it guaranteed that someone wins?
  * What if both threads run on identical speed CPU
  * executing in parallel
Scheduler Assumptions
- Normally we assume that
  - A scheduler always gives every executable thread opportunities to run
  - In effect, each thread makes finite progress
  - But schedulers aren't always fair
  - Some threads may get more chances than others
- To reason about worst case behavior we sometimes think of the scheduler as an adversary trying to "mess up" the algorithm

Critical Section Goals
- Threads do some stuff but eventually might try to access shared data

Critical Section Goals
- Perhaps they loop (perhaps not!)

Critical Section Goals
- We would like
  - Safety: No more than one thread can be in a critical section at any time.
  - Liveness: A thread that is seeking to enter the critical section will eventually succeed
  - Fairness: If two threads are both trying to enter a critical section, they have equal chances of success
- ... in practice, fairness is rarely guaranteed

Solving the problem
- A first idea:
  - Have a boolean flag, inside. Initially false.
  ```
  CSEnter()
  { 
    while(inside) continue; 
    inside = true; 
  }
  ```
  - Now ask:
    - Is this Safe? Live? Fair?

Solving the problem: Take 2
- A different idea (assumes just two threads):
  - Have a boolean flag, inside[i]. Initially false.
  ```
  CSEnter(int i)
  { 
    inside[i] = true; 
    while(inside[i]) continue; 
    inside[i] = false; 
  }
  ```
  - Now ask:
    - Is this Safe? Live? Fair?
Solving the problem: Take 3

- Another broken solution, for two threads
- Have a turn variable, turn, initially

```c
CSEnter(int i)
{
    while (turn != i) continue;
    turn = 1 ^ i;
}
```

- Now ask:
  - Is this Safe? Live? Fair?

A solution that works

- Deker’s Algorithm (book: Section 7.2.1.3)

```c
CSEnter(int i)       CSEnter(int i)       CSEnter(int i)
{                  {                  }
    int J = i^1;
    inside[i] = true;
    turn = J;
    while (inside[J] && turn == J) continue;
    Inside[i] = false;
}
```

Why does it work?

- Safety: Suppose thread o is in the CS.
  - Then inside[o] is true.
  - If thread i was simultaneously trying to enter, then turn must equal o and thread i waits
  - If thread i tries to enter "now", it sets turn to o and waits
- Liveness: Suppose thread 1 wants to enter and can’t (stuck in while loop)
  - Thread o will eventually exit the CS
  - When inside[o] becomes false, thread i can enter
  - If thread o tries to reenter immediately, it sets turn=1 and hence will wait politely for thread i to go first!