John Wineman
CS390C – Concurrency and Parallelism
Notes for 1/19/2012

Announcements:

- No class next Tuesday (1/24/2012)
- Guest Lecture next Thusday (1/26/2012)

Today:

- Threads
- Abstraction, message passing
- Erlang (sweedish telecom uses erlang system built on 3 million lines of code)
- Shared Memory Locks

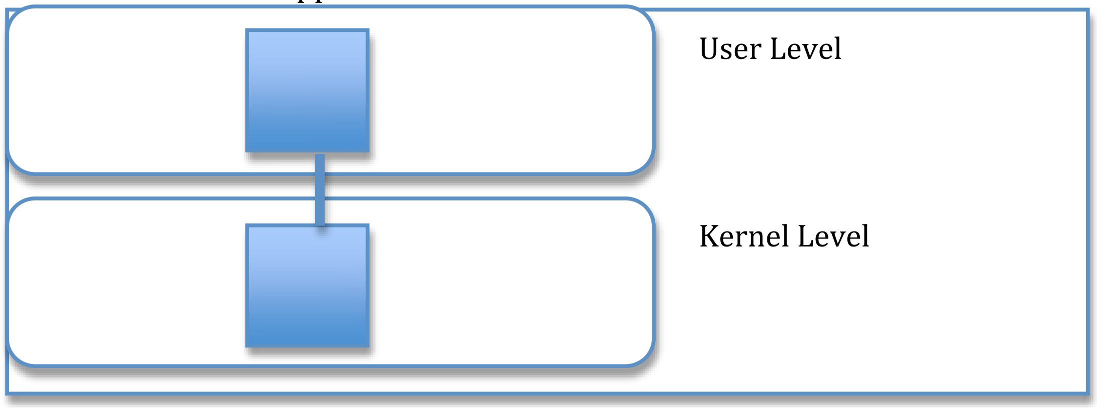
Last time:

- How to express concurrency?
    - Processes with state, independent units of computation
    - Scheduler decides when each process should run
    - The number of processes is irrelevant for the appearance of concurrency
    - OS has always handled concurrency.

What are the issues with this approach? There is no shared data and there is a lot over overhead because a kernel call is needed every time there is a context switch.

Other alternatives:

- User threads are mapped to kernel threads



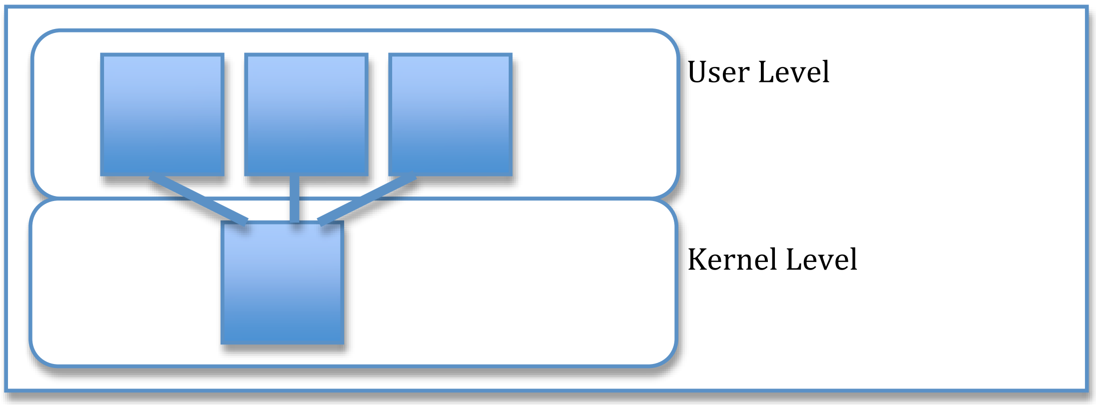
Pros:
- Kernel threads share address space
- Threads can share state
- Threads can act on the same data

Cons:
- Kernel is privileged, threads could be malicious

  - There is still overhead
  - A seg fault could potentially crash the OS
  - Scheduler agnostic to application, can't take into account patterns of use.

Next step:

- Multiple user threads map to one kernel thread



Pros:

- Scheduling is the User's job
- Faster, no kernel call for context switches
- Can use application semantics to schedule better
- Recover from deadlock without corrupting OS
- Seg fault DOESN'T corrupt OS data

Cons:

- Have to write your own scheduler
- How do you map user threads to kernel threads?
- How does the user stack map to the kernel stack?

This is the approach most modern languages (Java etc) use. Its important to not that the user threads do not have to map to one kernel thread. Most implementations map n user threads to k kernel threads.

Trade-offs of this approach:

- Cost of thread creation
- Blocking I/O – User based would block easier, kernel can context switch until I/O is ready.
- Application Sensitivity – User based scheduler can take into account application semantics, Kernel based cannot.

Problems with threads:

- Deadlock – when every process is waiting for a resource in order to run. Since all processes are waiting nothing runs.
- Live lock – One process "hogs" resources by acquiring and releasing resources so fast that slower resources are unable to run.
- Famines – If you leave your computer idle for long periods of time one process can start hogging memory, paging out the memory of other processes (like processes vital to the OS). When you return to your computer it will run very slowly because the OS is busy paging in all of the memory for the vital OS processes.

What if threads WANT to talk?

Three types of commnication:

- Synchronous
  - Guarantees delivery
  - Can't communicate if handshake doesn't succeed
- Asynchronous
  - Doesn't guarantee delivery
  - Preemptive
  - Uses call backs
- Demand Driven
  - Events (touching your iphone screen)
  - Extension of a call back.

You can think of **synchronous communication like a phone call**. If you want to tell someone something you pick up the phone and call them. If they do not answer your phone call you know the message is undelivered and you can continue waiting while the phone rings or call back another time.

Asynchronous communication is like a blackboard. I can come and write something on the blackboard, and leave without waiting to see if anyone sees what I wrote. You aren't guaranteed delivery because the blackboard might get erased or over written before the other person sees it. The other person might also never look at your blackboard.

Threadss add a lot of complexity to the way we program. In programming languages like Java there is no thread for the mouse. We only care about certain states (clicked, un-clicked etc) and activate methods based on these states. If you want to know the location of the mouse you don't need a thread, you can just query the mouse directly. Event driven programming is largely used in GUIs for this reason.

Issues with event based programming:

- How do you know how long the handler will run? (For scheduling purposes)
- No consistency in handler response (timing wise)
- Handler must run to completion

Performance and Correctness with regards to shared locks:

- Too many locks and you limit performance
- Too few locks and you risk incorrect data

Readings

● Why Threads are a Bad Idea (for most purposes),
Ousterhout, 1996

Why threads are bad:

- Synchronization
    - Must coordinate access to shared data with locks
    - Forget lock? Corrupted Data
- Deadlock
    - Circular dependency among locks
- Hard to debug
    - Data dependencies
    - Timing dependencies
- Callbacks don't work with locks

Events vs Threads:

- Events avoid concurrency as much as possible, threads embrace it.
- Debugging easier with events
- Events faster than threads on single CPU
- Events more portable than threads

Conclusions:

- Concurrency is fundamentally hard, avoid when possible
- Threads more powerful than events, but power is rarely needed
- Threads are harder to program than events
- Use threads only for performance critical kernels.

● Why Events are a Bad Idea (for high-concurrency servers),
von Behren et. al (2003)

Problems with threads

- **Performance** – Many attempts at high concurrency applications using threads have not performed well. – Major source of overhead is from O(n) number of threads. A modified GNU Pth user level thread package optimized to remove most of the O(n) overhead matches the performance of an event based server up to 100,000 threads.
- **Synchronization –** Event based systems claim "free" synchronization. This really only applies to synchronization on uniprocessors and is irrelevant in high concurrency servers because they often run on multiple cores.

- **State Management –** Thread stacks are an ineffective way to manage live state. A mechanism to allow for dynamic stack growth could solve this problem.
- **Scheduling –** Lauer-Needham duality indicates we can apply many of the same scheduling tricks used in event based scheduling to cooperatively scheduled threads.

●Cooperative Task Management without Manual Stack Management,
Adya et. al (2002

This paper argues a compromise between event based programming and thread based programming. The author argues you can have the best of both worlds fixing the "task based" management that event based programming lacks as well as the "stack based" management that threaded programming lacks. The following quote from the article best sums up the idea:

> "In our project there are passionate advocates for each of the two styles of stack management. There is a hybrid approach that enables both styles to coexist in the same code base, using adaptors to connect between them. This hybrid approach also enables a project to be written in one style but incorporate legacy code written in the other. In the Windows operating system, "threads" are scheduled preemptively and "fibers" are scheduled cooperatively. Our implementation achieves cooperative task management by scheduling multiple fibers on a singlethread; at any given time, only one fiber is active."