Concurrent Programming

Stephen Cross

Definitions

- **Concurrency:** “when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. E.g. multitasking on a single-core machine.”
- **Parallelism:** “when tasks literally run at the same time, e.g. on a multi-core processor.”

Problem(s)

- Managing a number of separate tasks.
- Controlling access to shared resources.
- Taking advantage of multiple processing units.
- Efficient access to memory.
- Handling I/O.
- Liveness (e.g. preventing GUI freezing).
- and...

COMPLEXITY

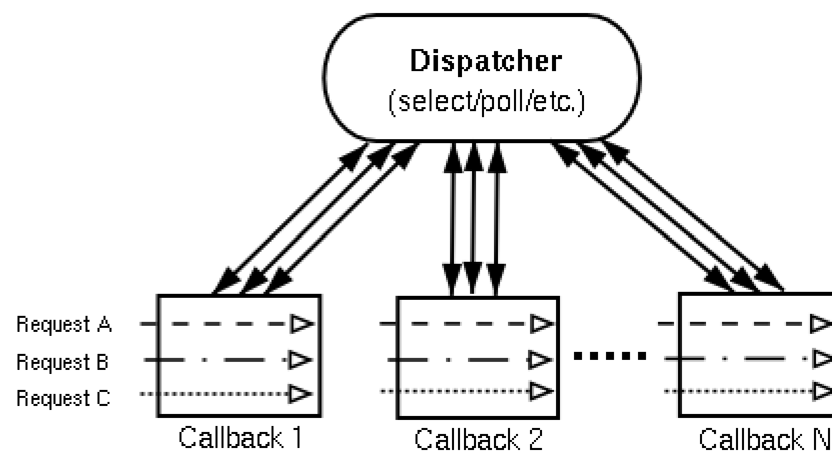
- A HUGE problem:
  - Complexity of maintaining state.
  - Complexity in controlling access to shared resources.
  - Complexity in performing I/O.
  - Complexity in hardware.
  - Complexity...
- Implies greater costs (both in terms of production and the end result).
- MUST avoid 'accidental' complexity.

Solutions

- **Threads** - Create a number of execution sequences, which run concurrently. System calls are blocking. Uses a shared address space.
- **Processes** - Equivalent to threads, except that each process has its own address space. Processes can communicate via IPC (Inter-Process Communication).
- **Events** - Single loop which handles events by running relevant routines. System calls are non-blocking.

Events

- All calls are 'non-blocking': ask the system to do something, but don't wait for it to complete.
- Progress of long-lasting events (e.g. socket connections, timers etc.) is given as events.
- Can be extended to multiple loops to exploit multiple processors.



(Image from State Threads)

Threads vs Events?

- Proven duality (i.e. Threads = Events).
- but...
  - Threads use stack space for managing state (efficient allocation/deallocation). Events require the heap, which implies garbage collection of some form (includes reference counting).
  - Threads make control flow clear. Events require designing and implementing state machines.
  - Exception handling is much easier in a thread-based environment.
  - Threaded code has better spatial locality.
- Event-based systems that solve these problems often end up reinventing threads.

Claim: Callbacks are 'bad'

- Disclaimer: callbacks are likely to be useful at a low level (similar to pointer arithmetic).
- Uses non-blocking calls. Combination is often referred to as 'asynchronous' programming.
- Highly related to event-based programming...
- ...has same problems:
  - State needs to be managed on the heap.
  - Control flow is less clear.
  - etc...

Claim: Callbacks are 'bad'

- Callbacks represent inversion of control: higher level component loses choice of where and when code is executed.
- Lower level component becomes blocked when executing callback.
- This can often lead to deadlock, since the lower level component can become 'blocked', so if higher level executes lower level command inside callback...!
- Destruction problems: need a way to stop callback being run and need to remember to do it.

Alternative to callbacks

- Instead of callbacks, use blocking methods.
- With blocked calls, all destruction problems are gone and higher level component now retains control.
- Event queues are a nice alternative for lower level components to deliver information to higher level components.

“The Problem with Threads”

- Many arguments against threads (this is the title of one).
- Referring to threads scheduled by the kernel.
- Controlling access to shared resources is very difficult.
- Non-determinism: Bugs in threaded code can appear occasionally and unreliably.
- Threads can make performance worse:
  - Controlling access to shared resources adds 'bureaucratic' overhead.
  - Context Switching
  - Cache Coherency

Non-Determinism

- Start two threads, with global variable X:
  - Thread A: *X := -1*
  - Thread B: *X := 1*
- In the end, X = ?

Non-Determinism

- Start two threads, with global variable X:
  - Thread A: *X := -1*
  - Thread B: *X := 1*
- In the end, X = ?
- Clearly X could be -1 or 1.
- But maybe assignment compiles to multiple instructions, so X could be something ***other than*** -1 or 1!
- For example, handling values larger than a word is likely to take two instructions.

Non-Determinism

- For non-trivial programs, there are a large number of possible interactions between threads.
- Very few possibilities are exposed during testing.
- Mutexes/locks/semaphores are available to restrict concurrent flow.
- But, it's difficult to verify that they are correct.
- In practice, it seems people find it very hard.
- Locks can be added automatically through techniques such as monitors, but this involves additional overheads.

Solution: Use Processes?

- Access to shared state by IPC: one or more processes manage the state and expose manipulating methods to other processes.
- Synchronization is much easier and processes are more independent than threads.
- But...
  - Communication overhead is larger.
  - Context switching overhead is larger.
  - Heavy-weight and likely to hit kernel limits.

Solution: Use Fibers?

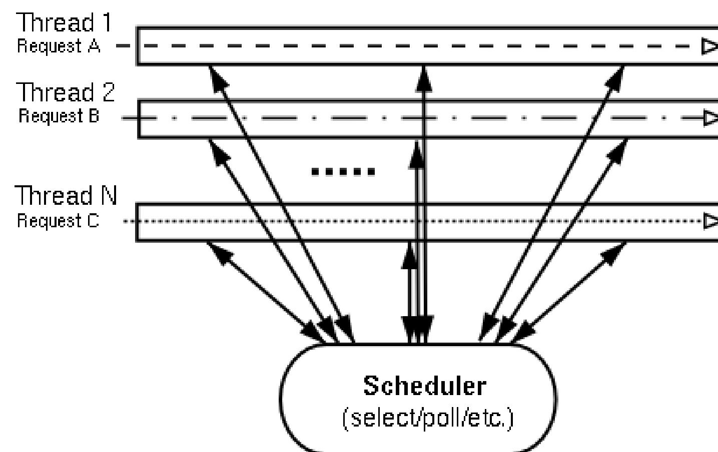
- Very light-weight threads.
- Cooperative Multitasking: fibers must explicitly yield control. Can do this at blocking function calls.
- Extra constraint (I added): Only run one fiber at any one time in a process.
- Managing access to shared resources is now much easier and no need for expensive synchronization...
- ...for example, containers such as linked lists don't need any synchronization.
- Context switching overhead is much smaller than kernel threads.
- But:
  - No parallelism.
  - Synchronization still required; there is still some non-determinism.

Solution: Processes + Fibers?

- Processes for safe parallelism.
- Fibers for non-parallel concurrency inside processes.
- Has the advantages of both:
  - Processes add parallelism that fibers lack -> good for scaling performance over multiple processors.
  - Fibers eliminate parallelism problems of threads inside processes...
  - ...but still allow processes to react to multiple external events concurrently.
- Example implementation: State Threads.

State Threads

- Light-weight C library that provides fibers (called threads in this context).
- Based on an event model (remember: event-based systems that solve their problems often end up reinventing threads).
- Calling a blocking function will cause control to be passed to another fiber.
- Targeted towards performance and scalability of internet applications.



(Image from State Threads)

Problems with Blocking Calls

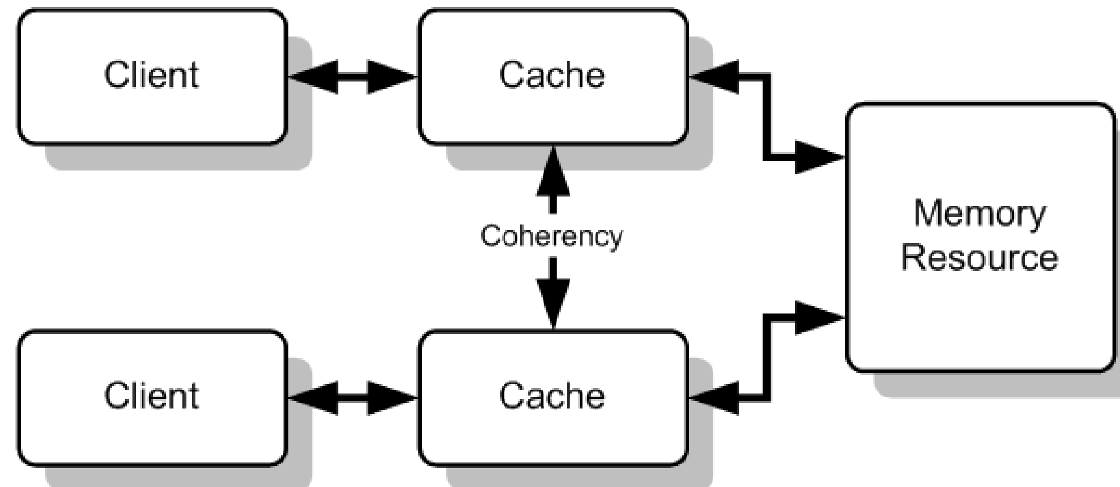
- Threads become blocked...
  - ...should be solved by creating another thread.
- Blocking calls cannot be cancelled...
  - This is actually a significant flaw in the design of some blocking call APIs.
  - Blocking call APIs ***MUST*** be designed with a good mechanism for cancelling calls at any time.
  - Also, timeouts are useful for some functions.

Cancelling Blocking Calls

- The best way to cancel blocking call is on the thread/fiber level.
- Cancel thread -> function call inside thread returns immediately with error code, or an exception can be thrown (there are some even better solutions).
- When threads are to be destroyed, the active thread should signal them to cancel and then wait for them to finish.
- State Threads provides `st_thread_interrupt()`.

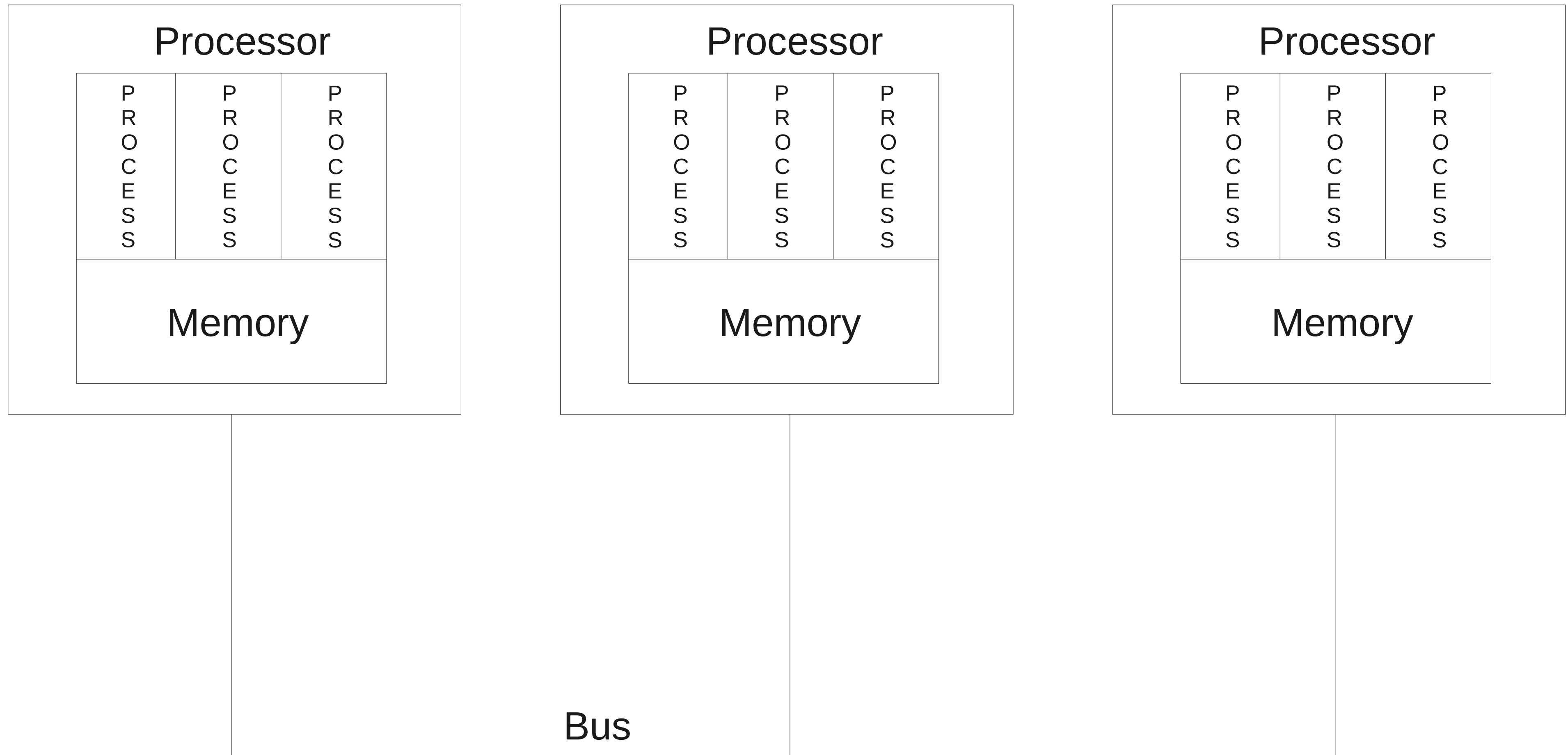
Cache Coherency

- The problem of keeping two caches 'in sync'.
- Client A changes memory -> Client B's cache could now be invalid.
- The system must do work to maintain the conceptual memory model in the presence of caches.
- Consider large numbers (100+) of clients...



(Image from Wikipedia)

Concurrency Model in Hardware



Concurrency Model in Hardware

- Very large number of processors.
- Each processor has its own memory, with caches.
- Assign processes to processors.
- One or more buses for communication between processors, facilitating IPC.
- Cache coherency is no longer a problem.
- Processors can be added without slowing the system (assuming a proportional addition of buses).
- ...

Concurrency Model in Hardware

- Software manages creation and management of fibers (via libraries such as State Threads).
- (Hardware and kernel have no knowledge of fibers).
- However:
  - Procedure to assign processes to processors must be good...
  - ...moving processes between processors would be very expensive...
  - ...software designers should be responsible for balancing work between processes.

Processes as Distributed Objects

- With the previous hardware model, processes become distributed objects, so need to consider:
  - Independent failure modes: Processes can fail independently of each other.
  - Execute in parallel: Processes can be (and are likely to be) on different processors.
  - Communication delay: Communication between processors will have a significant delay.
  - No global time: Only due to communication delay.
- However, programming with these considerations means applications are scalable to 'real' distributed systems (e.g. over the internet).

Example: Internet Server

- Should use a number of processes:
  - Provides some parallelism.
  - Independent failure modes.
- However, it should also use a number of fibers inside processes:
  - Avoids hitting kernel limits.
  - Context switching is faster between fibers.
- Good solution is to create as many processes as there are processors (two for single-core systems to add failure independence).
- Generally, try to group related fibers into processes.
- This is very similar to the design of Apache HTTP Server.

Adoption

- These ideas did not originate with me.
- Hardware clearly isn't required immediately; it provides a future performance boost over shared memory.
- However, there are software problems to be overcome:
  - Programmers need to use blocking calls and design APIs with them. (Fortunately this is a very natural way of programming.)
  - Threading should be switched to use fibers rather than kernel threads.
  - Blocking call APIs need to be improved to adding cancelling capabilities.
- The model is very similar to existing systems and changes mostly affect libraries, so adoption chances are good.

Concluding Quotes

- “For instance, many event systems 'call' a method in another module by sending an event and expect a 'return' from that method via a similar event mechanism. In order to understand the application, the programmer must mentally match these call/return pairs, even when they are in different parts of the code. ... Syntactically, thread systems group calls with returns, making it much easier to understand cause/effect relationships, and ensuring a one-to-one relationship.” - Why Events Are A Bad Idea
- “Unfortunately, the EDSM [Event-Driven State Machine] architecture is monolithic rather than based on the concept of threads, so new applications generally need to be implemented from the ground up. In effect, the EDSM architecture simulates threads and their stacks the hard way.” - State Threads
- “Blocking model was consistently 25-35% faster than using NIO selectors. Lot of techniques suggested by EmberIO folks were employed - using multiple selectors, doing multiple reads if the first read returned EAGAIN equivalent in Java. Yet we couldn't beat the plain thread per connection model with Linux NPTL.” - Java.IO vs Java.NIO