

## Concurrency Support

Ken Birman

## Our topic...

- To get high performance, distributed systems need to achieve a high level of concurrency
  - As a practical matter: interleaving tasks, so that while one waits for something, another can run
- Multi-core processors are about to make this a central focus of the OS community after a period of relative inattention
- So: how can an OS help the developer build concurrent applications that perform well?

## Why threads?

- Emerged as an early issue with UNIX!
- Consider challenge of building a program for a single-processor machine that
  - Accepts input from multiple I/O sockets
  - Needs to handle timeouts
  - May launch internal threads
  - May receive interrupts or signals
  - May want to do some blocking I/O
- Using cthreads (no kernel thread support)

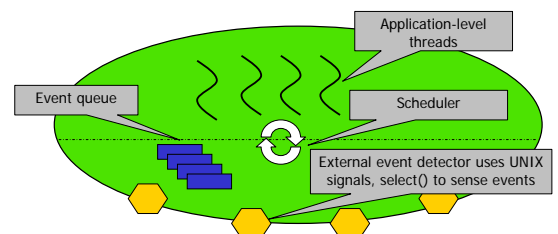
## Why is this a problem?

- In UNIX blocking I/O blocks the whole address space! So... any I/O leaves cthreads blocked!
- Options?
  - Select system call only understands I/O channels and timeout, not signals or other kinds of events
  - UNIX asynchronous I/O is hard to use

## Classic solution?

- Go ahead and build a multithreaded application, but threads never do blocking I/O
- Connect each "external event source" to a hand-crafted "monitoring routine"
  - Often will use signals to detect that I/O is available
  - Then package the event as an *event object* and put this on a queue. Tickle the scheduler if it was asleep
  - Scheduler dequeues events, processes them one by one... forks lightweight threads as needed

## Resulting architecture



## Problems with this?

- Only works if all the events show up as signals
- Depends on UNIX not “losing” signals
- Often must process a signal and also do a select call to receive an event
- Scheduler needs a way to block when no work to do (probably select()) and must be sure that signal handlers can wake it up

## Classic issues

- Threads that get forked off, then block for some reason
  - Address space soon bloats, causing application to crash
- Program is incredibly hard to debug, some problems seen only now and then
- Outright mistakes because C/C++ don't support “monitor style” synchronization
  - (Easier in Java, C#)

## Bottom line?

- Concurrency bugs are incredibly common and very hard to track down and fix
  - We want our cake but not the calories
- Programmers find concurrency unnatural
  - Try to package it better?
  - Identify software engineering paradigms that can ease the task of gaining high performance

## Hauser *et. al.* case study

- Their focus is on how the world's best hackers actually use threads
  - They learned the hard way
  - Maybe we can learn from them

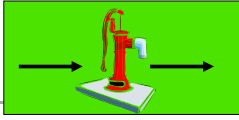
## Paradigms of thread usage

- Defer work
- General pumps
- Slack processes
- Sleepers
- One-shots
- Deadlock avoidance
- Rejuvenation
- Serializers
- Encapsulated fork
- Exploiting parallelism

## Defer work

- A very common scenario for them
- Client sees snappy response... something client requested is fired off to happen in the background
  - Examples: forking off a document print operation, or code that updates window
  - Issue? What if thread hangs for some reason. Client may see confusing behavior on a subsequent request!

## Pumps



- Components of producer-consumer pipelines that take input in, operate on it, then output it “downstream”
- Value is that they can absorb transient rate mismatches
- *Slack process*: a pump used to explicitly add delay, employed when trying to group small operations into batches

## Sleepers, one-shots



- These are threads that wait for some event, then trigger, then wait again
- Examples:
  - Call this procedure every 20ms, or after some timeout
  - Can think of device interrupt handler as a kind of sleeper thread

## Deadlock avoiders



- Thread created to perform some action that might have blocked, launched by a caller who holds a lock and doesn't want to wait
- A fairly dangerous paradigm...
  - Thread may be created with **X** true, but by the time it executes, **X** may be false!
  - Surprising scheduling delays a big risk here

## Task rejuvenation



- A nasty style of thread
  - Application had multiple major subactivities, such as input handler, renderer. Something awful happened.
  - So create a new instance and pray
- Seems to invite “heisenbugs”

## Aside: Bohr-bugs and Heisenbugs



- Bruce Lindsey, refers to models of the atom
- A Bohr nuclear was a nice solid little thing. Same with a Bohr-bug. You can hit it reproducibly and hence can fix it
- A Heisenbug is hard to pin down: if you localize an instance, the bug shifts elsewhere. Results from non-deterministic executions, old corruption in data structures, etc....
- Some thread paradigms invite trouble!

## Others

- Serializers: a queue, and a thread that removes work from it and processes that work item by item
  - Used heavily in SEDA
- Concurrency exploiters: for multiple CPUs
- Encapsulated forks: threads packaged with other paradigms

## Threads considered marvelous

- Threads are wonderful when some action may block for a while
  - Like a slow I/O operation, RPC, etc
- Your code remains clean and "linear"
- Moreover, aggregated performance is often far higher than without threading

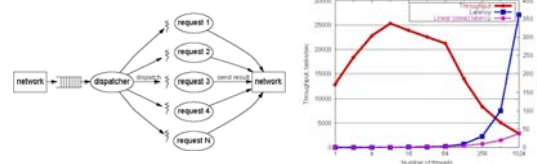
## Threads considered harmful

- They are fundamentally non-deterministic, hence invite Heisenbugs
- Reentrant code is really hard to write
- Surprising scheduling can be a huge headache
- When something "major" changes the state of a system, cleaning up threads running based on the old state is a pain

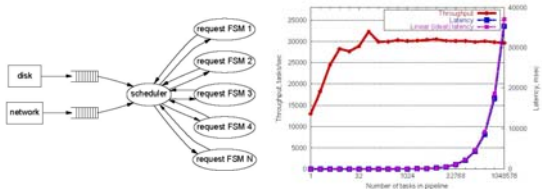
## SEDA paradigm

- Tries to replace most threads with:
  - Small pools of threads
  - Event queues
- Idea is to build a pipelined architecture that doesn't fork threads dynamically

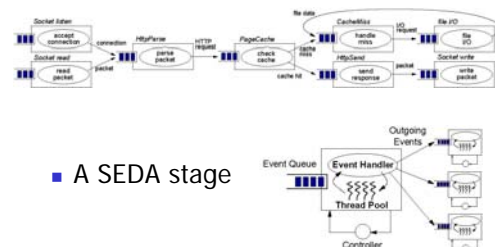
## Classic threading paradigm and a performance issue



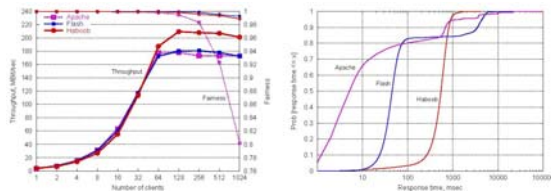
## Event-oriented paradigm



## Staged, Event Driven Approach (SEDA)



## SEDA performance



## Criticisms of SEDA

- It can still bloat, by having event queues get very large
- Demands a delicate match of processing power (basically, stages to the right need to be faster than stages to the left)
- Some claim that Haboob didn't work as asserted in this paper...

## Background

- SEDA was developed by Matt Welsh
  - Now at Harvard
  - Matt ran afoul of some of the über-hackers of the systems world
  - Huge fight ensued. Matt ultimately shifted to work on sensor networks (safer crowd)
- Comments that follow came from one of these angry über-hackers

## "Why SEDA sucks" (from an anonymous über-hacker)


- First, if you read between the lines, the paper's own numbers show how much SEDA sucks.
- For example, it shows comparable throughput to Flash where flash has half the number of file descriptors, which means he's at least twice as slow as flash.
  - (Since usually performance of these systems scales relatively linearly with file descriptors.)

## "Why SEDA sucks" (from an anonymous über-hacker)

- The paper shows graphs like CDFs of response time for clients serviced, when SEDA was dumping most requests (and hence they weren't showing up in the graph), while the other servers were serving way more clients.
- So yeah, SEDA's latency is lower than other servers if SEDA serves fewer clients and thus has lower throughput. And SEDA's throughput is comparable to other servers if you cut the other servers' concurrency.

## "Why SEDA sucks" (from an anonymous über-hacker)

- But on top of that the comparison is still dishonest, because the other servers are production servers with logging and everything.
- SEDA lacks a bunch of necessary facilities that would presumably decrease its performance if implemented. The authors annoyed some members of the systems community by giving talks saying, "You might ask if the use of Java would put SEDA at a performance disadvantage. It doesn't because fortunately I'm a very, very good programmer."
- Not a good move.



## "Why SEDA sucks" (from an anonymous über-hacker)

- "Almost as embarrassing as the performance and the complete failure to solve the stated problem (namely cnn.com's overload on September 11, 2001, which he never demonstrated SEDA could fix) is the fact that the authors completely ignored the related work from Mogul and Ramakrishnan on eliminating receive livelock."
- That work appeared in Journal form in 1997, and earlier in conference form).
  - They made a strong case for dropping events as early as possible, eliminating most queues, and processing events all the way through whenever you can.
  - Viewed in this light, SEDA should not have been published (years later) without even attempting to rebut their argument.



## Threads: What next?

- Language community is exploring threads with integrated *transactional rollback...*
  - Idea: thread is created for parallelism
  - It tracks data that was changed (undo list)
  - At commit point, check for possible concurrent execution by some other thread accessing same data. If so, roll back, retry
- Concerns? Many. Recalls Argus system (MIT) which used threads, transactions on abstract data types.... Life gets messy!



## Bottom line?

- Threads?
- Events?
- Transactions? (With top-level actions, orphan termination, nested commit???)
- A mixture?