JASON DUSEK

concurrent patterns: parallels in system and language design

Jason Dusek is a systems architect with a self-funded startup in the Bay Area. His technical interests include distributed computing, functional programming, and serialization.

jason.dusek@gmail.com

CONCURRENCY AND PARALLELISM ARE usually discussed together, and they are both important in reorganizing a program for execution on multiple cores or multiple computers. Parallelism asks, "How can we break a program up into independently executing parts?" It is very much in the domain of compiler writers and library providers, as the necessary program transformations are tedious at best. Concurrency asks, "How can independent programs share a value once in a while?" It bedevils program designers at every level—providers of parallelism toolkits, operating systems designers, application programmers, system administrators providing distributed network services, and even folks just trying to sell a few books.

Sharing

For programs to share a value, they may both possess a copy or they may both possess a handle.

If they both have a handle, that is cheapest and allows them to communicate most efficiently. For example, with file handles, two applications opening the same file are using less memory than two applications with one sending a copy to the other. They also communicate more quickly. However, handles can result in confusion and danger—for example, memmapped files. If two programs memmap the same file, and the "receiver" is not careful to close it and reopen it at the right time, one can very likely crash the receiver by writing into the memmapped file. The less extreme example of two database clients overwriting parts of one another's changes, resulting in a mutually confusing (but readable) record, is a more common pitfall of these handle-based (we'll call them "shared-memory") approaches to concurrency. Down this path lay transactions and locking. There are numerous forms of locking, the most dreadful being spin-lock, deadlock, and live-lock.

When we pass around copies, each process can pretty much go on its merry way from the point of copy transfer. As mentioned earlier, this can result in a great waste of memory. These "message-passing" strategies are inevitable, though, in network services—an LDAP server cannot share memory with its clients; it has to just send them the data. If we structure all our applications this way, without

regard to locality, then it is at least consistent, if not efficient. Garbage collection is very important in message-passing architectures, because you create a lot of garbage by copying things all over the place. However, a program is safe from interruption from other programs as long as it does not get (or request) any messages. The problem of data synchronization can be resolved through frequent polling or publish-subscribe (possible in LDAP, if you are brave)—but at least there is some kind of warning, some kind of formal indicator, that a resource has changed.

At this point you may wonder how the two database clients are sharing memory while the LDAP server and its clients are passing messages? It is really a matter of where I choose to point the camera. Between the servers and clients there is message passing—clients make a request and servers send back a message full of data. Any changes in the server's data do not affect the clients at all as long as they don't ask for updates. Between the two database clients, there is shared state—each application writes to the database and reads from the database to communicate with the other processes. The difference shows up in where we put the locks—a database locks its tables, not its socket.

We Don't Need Concurrency In Programming Languages . . .

It stands to reason (although I cannot prove it) that all models of concurrency are explored in systems first and in languages later. Network services and operating systems offer a kind of parallelism—operating system process, multiple clients, servers running on distinct machines—and thus invite all the problems we usually associate with threading and message passing.

Within the *NIX filesystem, as in other shared-memory systems, we need to protect processes from mutual confusion—we need transactions, locks, and semaphores. The *NIX filesystem models all three. Through file locking, multiple POSIX programs protect one another from contradictory updates. A *shared lock* allows multiple shared locks to coexist with it, whereas an *exclusive lock* can only be taken when there are no other locks. When the locks are advisory—the default for *NIX—the lock is merely a baton, which processes can only possess under certain circumstances. These advisory locks are really semaphores, tokens of resource ownership without enforcement [1, 2]. In contrast, mandatory locks are true locks—processes block if they don't have the right lock. No reads can proceed without some kind of lock, and writes cannot proceed without a write lock. Often, an operating system will provide special signals and subscriptions to allow a process to know that a file is again open for reading, that another process wants to write a file, and other things. We can see how message passing and shared memory complement one another [3].

A transaction is a collection of operations on different resources that is performed in an all-or-nothing manner. To perform a transactions in a *NIX filesystem we must:

- Lock the greatest parent of all the files in the transaction and every subordinate file. (We'll be waiting a long time to get all those locks in an active system.)
- Recursively copy the greatest parent into a new directory.
- Perform our changes.
- Use `mv` to atomically overwrite the old greatest parent with the now updated copy.

Our rather overly generous lock scope has prevented anything from changing in the meantime.

Why must we make such a gratuitous copy? Because the filesystem does not offer a means to atomically mv several files. What if we got halfway through the mvs and the user sent SIGKILL? Then we'd have left the system in an inconsistent state, with no means of recovery. So we have to find a way to mv all the changes at once, and that means we have to do the copy. If we had local version control we could be more flexible, simply reverting changes if there were a failure, while keeping everything exclusively locked so no other processes could read the corrupt data until the "rollback" is complete. (Can we be sure reverts never fail?) Although rollbacks are not required for transactions, we see how they can protect us from over-locking.

As mentioned earlier, *NIX provides "signals," a form of message passing. Sockets, a more general and flexible mechanism, allow processes to communicate with one another by establishing a connection and sending messages back and forth, which corresponds to the *channels* of Hoare's CSP [4, 5]. A one-way channel is a *pipe*, both in Hoare's terminology and in *NIX. Unfortunately, *NIX does not allow us to open multiple pipes to a process, so this is a point where *NIX and Hoare diverge. A named pipe—a FIFO—is like a mailbox, a well-known place to leave a package for another process. Unfortunately, FIFOs allow IO interleaving and thus cannot really be used as mailboxes. But I think you get the idea. So what are signals? Signals are mailboxes—the kernel knows where to put messages for every process. We look up its PID to send it a message. Channels, pipes, and mailboxes can fake one another:

- To get a mailbox from channels, the receiver should simply aggregate all messages from every sender regardless of the channel they came in on. To get channels from mailboxes, we must always send a "return address" as part of the message. The receiver, when reading the messages, uses the return address to know which connection it is handling and where to return results.
- To get a pipe from a channel, only send messages in one direction. To get a channel from pipes, we need two pipes, one in either direction. The sender and the receiver must have the sense to keep the pipes together.
- To get pipes from mailboxes, we can *multiplex* the mailbox. Once again, we use the "return address," but this time, we only use the return address to tell us where the message came from, not how to send it back.

Insofar as they work, *NIX concurrency features support a strategy of composing concurrent systems from operating system processes. This is nice in so many ways—the file system handles garbage collection and caching, the scheduler distributes processes over the hardware, and programs in multiple languages can share resources and concurrency model. However, *NIX file locking is brittle, and the only out-of-the-box message-passing model is channels, by way of sockets. *NIX turns out not to be the best platform for an application that is looking for operating-system-level concurrency—but a search for an alternative leads us far afield.

Bell Labs' Plan 9, an evolution of UNIX with some of the original UNIX team on board, offers pipes in the sense of Hoare's CSP. Through resource multiplexing, the same name in the filesystem refers to different resources for different processes, converting a FIFO to a pipe to avert the IO interleaving that bedevils *NIX FIFOs [6]. We could probably emulate this system on any other *NIX, using bind mounts, union mounts, and pipe polling, but it would not be pretty.

At just about the time of Plan 9's emergence, Tandem's NonStop platform was in decline. NonStop SQL ran on top of the Guardian cluster operating

system. In Guardian, every resource—every file, even—was a "process" that could receive messages [7]. Pervasive message passing is the door to easy clustering. NonStop SQL was able to run transactions across the cluster, which is a nontrivial task; and Guardian was able to fail over a process from one machine to another if the need arose [8].

There are numerous "cluster operating systems," but what they address is more a matter of resource usage than concurrent design primitives [9–11]. Plan 9 and Guardian are special because they make message-passing tools available to the application programmer and provide an environment where those message-passing tools are widely used.

Networks services model a few concurrency patterns not mentioned above.

Multi-view concurrency, which we might call transactional copy-on-write, is used in some SQL databases and is a long-tested approach to ACIDity. To read, read. To write, copy everything you wish to read or write, prepare the changeset, and the database will apply it if (only if) nothing that was read has been written since, and nothing that was written has been read or written since [12–14].

IRC is an example of a publish-subscribe message-passing service. We subscribe to the channel and receive change sets to the channel—messages—as they arrive. There is no way to pull "the" channel, though—and so we sometimes miss messages, to the amusement of all [15]. In contrast, OpenLDAP offers publish-subscribe messaging as an optimization on the shared-memory model. A client subscribes to an LDAP subtree and, as changes are made, they are forwarded [16]. However, there is one true tree—the tree on the LDAP server—and we can synchronize with it to ensure our copy is correct [17].

. . . But We Like It

Even when operating-system-level concurrency works, there's some mnemonic load in handling it. Network-service-level concurrency is in some sense simpler, but it also handles less—process management is delegated to the operating system. Concurrent programming languages specify an entire concurrent system: resources and a model for their use, as well as processes and a means of creating them.

Language-level concurrency has tended toward message passing. Early versions of Smalltalk were distinctively message-passing, and Carl Hewitt, the founder of the "Actors model," was inspired by Smalltalk [18]. More recent message-passing languages include Stackless Python, used to implement the massively multiplayer game EVE [19]; Limbo, a project by the team that worked on Plan 9 [20]; and Erlang [20]. The former two are channel languages, whereas Erlang is a mailbox language. MPI, a message-passing library and runtime for C, Java, C++, O'Caml, and Fortran, brings message-passing to languages that do not have it natively [21, 22]. Shared memory is an unusual paradigm at the language level.

Erlang, a Message-Passing Language

Erlang, superficially similar to Prolog, makes RPC a language primitive. Processes (function calls) can send messages, listen for messages, and access their present PID. They perform message sending and nonconcurrent things (e.g., math and string handling and ASN.1 parsing) until they hit a receive statement, which is rather like a case statement, only with no argument. The argument is implicit—the next message in the mailbox. After a message is

handled, the executing function may call itself or another function recursively, or it may do nothing, which ends the process.

Erlang processes run within instances of the Erlang virtual machine, called nodes. Nodes are clustered to form applications that run on several machines at once. Although Erlang is often called a functional programming language, this is missing the point. Erlang offers easy IPC, a notion of process hierarchy and identity, rapid process spawning, and excellent libraries for process control and monitoring. Erlang is what the shell could have been.

Processes register with a cross-node name server so that they can offer named services to other processes. Processes are "linked" to other processes in the sense that a parent is made aware of the linkee's exit status. The combination of global name registry and linking allows us to implement reliable services in the face of "fail fast" behavior. Processes are arranged with a monitor—an error handler—linked to a worker. The worker runs a function that registers itself for the global name and then calls the main function that handles requests and recursively calls itself over and over. The monitor hangs out. If the server process dies, the monitor receives a signal to that effect and recalls the function.

The special thing about Erlang is not that we can write programs this way, but that we write *every* program this way. Programs are collections of communicating services in Erlang, not a main thread of execution with subroutines (at least, not on the surface.) The innumerable executing threads are easy to parallelize, because they share no state with one another and thus can be interrupted and resumed at any time. When we see Erlang trumpeted as the multi-core solution, that is why.

Erlang's bias toward concurrent design is perhaps too great. Although the standard libararies are rich in protocol handlers and design patterns for concurrent applications, the language is weak at string handling and arithmetic.

Haskell's Approach to Shared Memory

Programming languages that offer safe access to shared memory are rare. Haskell, a purely functional language, offers "Software Transactional Memory," which is much like multi-view concurrency in databases.

Haskell's type system allows it to make very strong guarantees about shared-memory operations. A pure function is a function that yields the same answer for the same arguments [e.g., sin(x)] whereas an impure function—hereafter a procedure—returns different things depending on the context [e.g., `gettime()`].

How do we perform input and output and get the time of day in a purely functional language? It turns out that there is a "pure view" of these impure operations. The principle is not hard to understand—values from impure functions are wrapped in a special container—so `gettime` does not return an `Int`; it returns an `IO Int`, an `Int` within the `IO` container.

A pure function, even when it shares state with another pure function, is not allowed to mutate it. It is immaterial in which order we evaluate the pure functions, so long as we evaluate calls that a function depends on before evaluating the function itself. This no-mutation property makes parallelization easy, and the Haskell compiler takes advantage of that.

So we have all these functions, and they are likely to execute at any time. They do not have process identifiers, and there is no global registry of names. How do these functions communicate? One means, an early one, was to in fact brand every concurrent function with `IO` and force it to operate on

references. This brings us all the problems of shared memory and none of the solutions; it also forces the compiler to be as paranoid about code accessing references as it is about code accessing the filesystem or network. The Glasgow Haskell Compiler project later introduced a new container, STM, which is specifically for operations that work on a large global store of values [24]. A procedure that executes within STM creates a log of its reads and writes. When the procedure ends, the log is checked to ensure that none of the values have changed since the procedure read them. As long as the reads are okay, the writes in the log are committed. If they are not okay, the procedure retries until it works (including forever).

STM offers true fine-grained transactions on a runtime system that can run millions of threads, but there are no provisions for clustering or process hierarchy. This is in some sense inevitable; shared memory systems don't dovetail nicely with the network's natural separation of state across servers.

Concurrency and You

Splitting a program into concurrently running parts can simplify design and always parallelizes the program. Whether this results in a net performance benefit depends on the overhead of communication.

To take advantage of concurrent design and implicit parallelism, one must adopt a new way of thinking about program structure, data structures, and efficiency. In-place update is efficient and natural in sequential computing, but in concurrent systems it is fraught with peril and obstructs parallelism. Bolting concurrency onto a sequential language—an imperative language—leads to inconsistency at best, and so it is understandable that much work in concurrency has taken place under the declarative tent.

Concurrent Perl would find many users if it existed, and there have been numerous attempts to bring concurrent programming to C and C++. Concurrency-friendly languages are unusual languages, bringing more or less of the functional paradigm with them. Will a new language gain a foothold, or will we find a way to bring message passing or transactional memory to C?

Perhaps, like object-oriented design, concurrent programming will find wide use only after it has been integrated with C. Toolkits for parallelism in C, C++, and Java are certainly catching up, although they are used mostly by game programmers and authors of scientific visualization software. Languages used mostly in Web programming and system administration have not received that kind of attention, and consequently we see the growth of Erlang in the area of high-availability network services. Niche programming languages will always have their niche, and it's likely that mainstream programming languages will always have the mainstream.

REFERENCES

[1] fcntl man page: fcntl - manipulate file descriptor: http://linux.die.net/man/2/fcntl.

[2]Semaphores: http://www.ecst.csuchico.edu/~beej/guide/ipc/semaphores.html.

[3] Andy Walker, "Mandatory File Locking for the Linux Operating System," April 15, 1996: http://www.cs.albany.edu/~sdc/Linux/linux/Documentation/mandatory.txt.

[4] C.A.R. Hoare, "Communicating Sequential Processes: 7.3.2 Multiple Buffered Channels," June 21, 2004: http://www.usingcsp.com/.

[5] Sean Dorward, Rob Pike, David Leo Presotto, Dennis M. Ritchie, Howard Trickey, Phil Winterbottom, "The Inferno Operating System": http://doc.cat-v.org/inferno/4th_edition/inferno_OS.

[6] Rob Pike, "Rio: Design of a Concurrent Window System," February 4, 2000: http://herpolhode.com/rob/lec5.pdf.

[7] The Tandem Database Group, "NonStop SQL, a Distributed, High-Performance, High-Availability Implementation of SQL," April 1987: http://www.hpl.hp.com/techreports/tandem/TR-87.4.html.

[8] Wikipedia, Tandem Computers: http://en.wikipedia.org/wiki/Tandem_Computers#History.

[9]: C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, I. Scherson, "Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems," September 2004: http://www.ics.uci.edu/~schark/cluster04.ps.

[10] Wikipedia, QNX: http://en.wikipedia.org/wiki/QNX.

[11]: Nancy P. Kronenberg, Henry M. Levy, William D. Strecker, "VAXclusters: A Closely-Coupled Distributed System," May 1986: http://lazowska.cs.washington.edu/p130-kronenberg.pdf.

[12] Wikipedia, MultiView concurrency control: http://en.wikipedia.org/wiki/Multiversion_concurrency_control.

[13] David Patrick Reed, "Naming and Synchronization in a Decentralized Computer System," September 1978: http://publications.csail.mit.edu/lcs/specpub.php?id=773.

[14] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems," June 1981: http://www.sai.msu.su/ ~megera/postgres/gist/papers/concurrency/p185-bernstein.pdf.

[15] RFC 1459: IRC Concepts: http://www.irchelp.org/irchelp/rfc/chapter3.html.

[16] LDAP for Rocket Scientists: Replication refreshAndPersist (Provider Push): http://www.zytrax.com/books/ldap/ch7/#ol-syncrepl-rap.

[17] LDAP for Rocket Scientists: Replication refreshOnly (Consumer Pull): http://www.zytrax.com/books/ldap/ch7/#ol-syncrepl-ro.

[18] Alan Kay, "The Early History of Smalltalk: 1972-76—The First Real Smalltalk: http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html.

[19] About Stackless: http://www.stackless.com/.

[20] Limbo: http://www.vitanuova.com/inferno/limbo.html.

[21] Concurrent programming: http://www.erlang.org/doc/getting_started/conc_prog.html.

[22] Wikipedia, MPI: http://en.wikipedia.org/wiki/Message_Passing_Interface.

[23] MPI asics: http://www-unix.mcs.anl.gov/dbpp/text/node96.html.

[24] Simon Peyton Jones, "Beautiful Concurrency," December 22, 2006: http://research.microsoft.com/~simonpj/Papers/stm/beautiful.pdf.