Bell Labs and CSP Threads

Russ Cox
rsc@swtch.com

Also available in Serbo-Croatian, Swedish

Introduction

This page is a slice of the history of concurrent programming, focusing on one particular lineage of Hoare's language of communicating sequential processes (CSP) [1] [1a]. Concurrent programming in this style is interesting for reasons not of efficiency but of clarity. That is, it is a widespread mistake to think only of concurrent programming as a means to increase performance, *e.g.*, to overlap disk I/O requests, to reduce latency by prefetching results to expected queries, or to take advantage of multiple processors. Such advantages are important but not relevant to this discussion. After all, they can be realized in other styles, such as asynchronous event-driven programming. Instead, we are interested in concurrent programming because it provides a natural abstraction that can make some programs much simpler.

What this is not

Most computer science undergraduates are forced to read Andrew Birrell's "An Introduction to Programming with Threads." The SRC threads model is the one used by most thread packages currently available. The problem with all of these is that they are too low-level. Unlike the communication primitive provided by Hoare, the primitives in the SRC-style threading module must be combined with other techniques, usually shared memory, in order to be used effectively. In general, programmers tend not to build their own higher-level constructs, and are left frustrated by needing to pay attention to such low-level details.

For the moment, push Birrell's tutorial out of your mind. This is a different thread model. If you approach it as a different thread model, you may well find it much easier to understand.

Communicating Sequential Processes

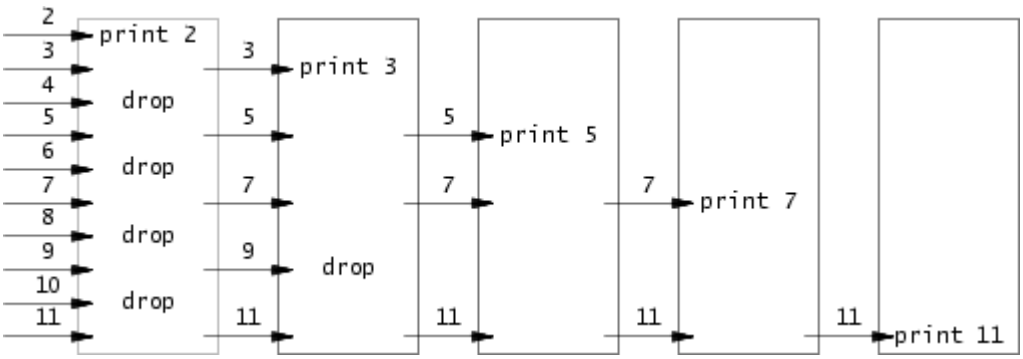
By 1978, there were many proposed methods in use for communication and synchronization in the context of programming multiprocessors. Shared memory was the most common communication mechanism, and semaphores, critical regions, and monitors were among the synchronization mechanisms. C. A. R. Hoare addressed both issues with a single language primitive: synchronous communication. In Hoare's CSP language, processes communicate by sending or receiving values from named unbuffered channels. Since the channels are unbuffered, the send operation blocks until the value has been transferred to a receiver, thus providing a mechanism for synchronization.

One of Hoare's examples is that of reformatting 80-column cards for printing on a 125-column printer. In his solution, one process reads a card at a time, sending the disassembled contents character by character to a second process. This second process assembles groups of 125 characters, sending the groups to the line printer. This sounds trivial, but in the absence of buffered I/O libraries, the necessary bookkeeping involved in a single-process solution is onerous. In fact, buffered I/O libraries are really just encapsulations of these two sorts of processes that export the single-character communication interface.

As another example, which Hoare credits to Doug McIlroy, consider the generation of all primes less than a thousand. The sieve of Eratosthenes can be simulated by a pipeline of processes executing the following pseudocode:

```
p = get a number from left neighbor
print p
loop:
    n = get a number from left neighbor
    if (p does not divide n)
        send n to right neighbor
```

A generating process can feed the numbers 2, 3, 4, ..., 1000 into the left end of the pipeline: the first process in the line eliminates the multiples of 2, the second eliminates the multiples of 3, the third eliminates the multiples of 5, and so on:



The linear pipeline nature of the examples thus far is misrepresentative of the general nature of CSP, but even restricted to linear pipelines, the model is quite powerful. The power has been forcefully demonstrated by the success of the filter-and-pipeline approach for which the Unix operating system is well known [2] Indeed, pipelines predate Hoare's paper. In an internal Bell Labs memo dated October 11, 1964, Doug McIlroy was toying with ideas that would become Unix pipelines: “We should have some ways of coupling programs like garden hose--screw in another segment when it becomes necessary to massage data in another way. This is the way of IO also.” [3]

Hoare's communicating processes are more general than typical Unix shell pipelines, since they can be connected in arbitrary patterns. In fact, Hoare gives as an example a 3x3 matrix of processes somewhat like the prime sieve that can be used to multiply a vector by a 3x3 square matrix.

Of course, the Unix pipe mechanism doesn't require the linear layout; only the shell syntax does. McIlroy reports toying with syntax for a shell with general plumbing early on but not liking the syntax enough to implement it (personal communication, 2011). Later shells did support some restricted forms of non-linear pipelines. Rochkind's 2dsh supports dags; Tom Duff's rc supports trees.

Hoare's language was novel and influential, but lacking in a few key aspects. The main defect is that the unbuffered channels used for communication are not first-class objects: they cannot be stored in variables, passed as arguments to functions, or sent across channels. As a result of this, the communication structure must be fixed while writing the program. Hence we must write a program to print the first 1000 primes rather than the first *n* primes, and to multiply a vector by a 3x3 matrix rather than an *n*x*n* matrix.

Pan and Promela

In 1980, barely two years after Hoare's paper, Gerard Holzmann and Rob Pike created a protocol analyzer called pan that takes a CSP dialect as input. Pan's CSP dialect had concatenation, selection, and looping, but no variables. Even so, Holzmann reports that “Pan found its first error in a Bell Labs data-switch control protocol on 21 November 1980. ” [14]. That dialect may well have been the first CSP language at Bell Labs, and it certainly provided Pike with experience using and implementing a CSP-like language, his first of many.

Holzmann's protocol analyzer developed into the Spin model checker and its Promela language, which features first-class channels in the same way as Newsqueak (q.v.).

Newsqueak

Moving in a different direction, Luca Cardelli and Rob Pike developed the ideas in CSP into the Squeak mini-language [4] for generating user interface code. (This Squeak is distinct from the Squeak Smalltalk implementation.) Pike later expanded Squeak into the fully-fledged programming language Newsqueak [5][6] which begat Plan 9's Alef [7] [8], Inferno's Limbo [9], and Google's Go [13]. The main semantic advantage of Newsqueak over Squeak is that Newsqueak treats communications channels as first-class objects: unlike in CSP and Squeak, channels *can* be stored in variables, passed as arguments to functions, and sent across channels. This in turn enables the programmatic construction of the communication structure, thus allowing the creation of more complex structures than would be reasonable to design by hand. In particular, Doug

McIlroy demonstrated how the communication facilities of Newsqueak can be employed to write elegant programs for manipulating symbolic power series [10]. Similar attempts in traditional languages tend to mire in bookkeeping. In a similar vein, Rob Pike demonstrated how the communication facilities can be employed to break out of the common event-based programming model, writing a concurrent window system [11].

Alef

Alef [7] [8] was a language designed by Phil Winterbottom to apply the Newsqueak ideas to a full-fledged systems programming language. Alef has two types of what we have been calling processes: procs and threads. The program is organized into one or more procs, which are shared-memory operating system processes that can be preemptively scheduled. Each proc contains one or more tasks, which are cooperatively scheduled coroutines. Note that each task is assigned to a particular proc: they do not migrate between procs.

The main use of procs is to provide contexts that can block for I/O independently of the main tasks. (Plan 9 has no select call, and even on Unix you need multiple procs if you want to overlap computation with non-network I/O.) The Acme paper [12] has a nice brief discussion of procs and threads, as do the lecture notes about the Plan 9 window system, also mentioned below.

Limbo

The Inferno operating system is a Plan 9 spinoff intended for set-top boxes. Its programming language, Limbo [9], was heavily influenced by Alef. It removed the distinction between procs and tasks, effectively having just procs, though they were of lighter weight than what most people think of as processes. All parallelism is preemptive. It is interesting that despite this, the language provides no real support for locking. Instead, the channel communication typically provides enough synchronization and encourages programmers to arrange that there is always a clear owner for any piece of data. Explicit locking is unnecessary.

Libthread

Back in the Plan 9 world, the Alef compilers turned out to be difficult to maintain as Plan 9 was ported to ever more architectures. Libthread was originally created to port Alef programs to C, so that the Alef compilers could be retired. Alef's procs and tasks are called procs and threads in libthread. The manual page is the definitive reference.

Go

Rob Pike and Ken Thompson moved on to Google and placed CSP at the center of the Go language's concurrency support.

Getting Started

To get a feel for the model, especially how processes and threads interact, it is worth reading the Alef User's Guide [8]. The first thirty slides of this presentation are a good introduction to how Alef constructs are represented in C.

The best examples of the power of the CSP model are McIlroy's and Pike's papers, mentioned above [10] [11].

Rob Pike's home page contains lecture notes from a course on concurrent programming: an introduction, and slides about the two aforementioned papers: squinting and window system. The last of the three provides a good example of how Plan 9 programs typically use procs and tasks.

Rob Pike gave a tech talk at Google that provides a good introduction (57 minute video).

Rob Pike's half of his 2010 Google I/O talk with Russ Cox shows how to use channels and Go's concurrency to implement a load balancing work management system.

Related Resources

John Reppy has applied the same ideas to ML, producing Concurrent ML. He used CML to build, among other things, the eXene multithreaded (non-event-driven) X Window System toolkit.

References

[1] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* 21(8) (August 1978), 666-677.

[1a]C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[2] Michael S. Mahoney, ed., *The Unix Oral History Project, Release 0: The Beginning*

[3] M. Douglas McIlroy, internal Bell Labs memorandum, October 1964.

[4] Luca Cardelli and Rob Pike, "Squeak: a Language for Communicating with Mice," *Computer Graphics*, 19(3) (July 1985: SIGGRAPH '85 Proceedings), 199-204.

[5] Rob Pike, "The Implementation of Newsqueak," *Software--Practice and Experience*, 20(7) (July 1990), 649-659.

[6] Rob Pike, "Newsqueak: a Language for Communicating with Mice," *Computing Science Technical Report 143*, AT&T Bell Laboratories, Murray Hill, 1989.

[7] Phil Winterbottom, "Alef Language Reference Manual," in *Plan 9 Programmer's Manual: Volume Two*, AT&T, Murray Hill, 1995.

[8] Bob Flandrena, "Alef Users' Guide," in *Plan 9 Programmer's Manual: Volume Two*, AT&T, Murray Hill, 1995.

[9] Dennis M. Ritchie, "The Limbo Programming Language," in *Inferno Programmer's Manual, Volume 2*, Vita Nuova Holdings Ltd., York, 2000.

[10] M. Douglas McIlroy, "Squinting at Power Series," *Software--Practice and Experience*, 20(7) (July 1990), 661-683.

[11] Rob Pike, "A Concurrent Window System," *Computing Systems*, 2(2) 133-153.

[12] Rob Pike, "Acme: A User Interface for Programmers," *Proceedings of the Winter 1994 USENIX Conference*, 223-234.

[13] golang.org, "The Go Programming Language".

[14] Gerard Holzmann, "Spin's Roots".

[15] Gerard Holzmann, "Promela Language Reference".