Are Concurrent Programs That Are Easier to Write Also Easier to Check?

Kedar S. Namjoshi
Bell Labs

A most important, but also a most elusive, aspect of any tool is its influence on the habits of those to train themselves in its use. If the tool is a programming language, this influence is—whether we like it or not—an influence on our thinking habits.
Edsger W. Dijkstra, "A Discipline of Programming" (1976)

This quote expresses the importance of choosing the right vocabulary to describe and reason about a concept. There is a long and rich history of language proposals for concurrency: semaphores, locks, wait-and-notify, monitors, conditional critical regions, rendezvous, message passing, and transactional memory. Which of these, if any, forms a good vocabulary? What criteria should one apply?

My touchstone is the expression of common concurrency patterns. An important criterion for a vocabulary, in my view, is that it is possible to express common patterns in an "obviously correct" manner. The expectation is that simpler solutions are likely to have fewer or more easily detectable errors. An alternative view of this test is that common concurrency patterns should have simple and short proofs of correctness.

To this end, consider three common concurrency patterns: a pipeline, a barrier, and a thread pool. The pipeline is pervasive in network processing. Typically, incoming packets are passed through a number of processing stages. This pattern is trivial to describe in terms of message exchanges (Figure 1). The particular style of message exchange is from Hoare's work on *Communicating Sequential Processes* (CSP) [2]: `in?msg` represents a read from channel `in` to a variable `msg`, `out!msg` represents a write of the value of `msg` to channel `out`. Each channel has a single reader and writer. Communication is *unbuffered* and *synchronous*, in that both reads and writes block until the complementary operation is enabled, at which point the communication takes place.

Next, consider a barrier for `N` processes. All processes have to reach the barrier before any can proceed. In the code in Figure 2, the barrier process counts arrivals; once all processes have arrived, it issues `Proceed` messages. The `alt` construct (also from CSP) is used to synchronize with one of `toBarrier[1]?Arrived, ..., toBarrier[N]?Arrived`.

For the final, everyday example, consider a taxi rank at an airport (Figure 3). A taxi driver registers with a dispatcher, she is assigned a passenger, and she

```
void do_task(int index, message msg);
process P(int i, channel in, channel out){
  message msg;
  while(true){
      in ? msg;
      do_task(i, msg);
      out ! msg;
  }
}
process pipeline(channel in, channel out){
  channel c1 = new channel();
  process p1 = new P(0,in,c1); process p2 = new P(1,c1,out);
}
```

Figure 1: A simple pipeline.

re-registers after the passenger has been dropped off. This pattern is precisely that seen in a thread pool, a common pattern in systems programming.

Now consider programming these patterns with locks or semaphores—or Java's synchronized methods. While the barrier is easily implemented, the others require much more work. Moreover, the delicate interplay between locks, synchronized methods, and wait and notify operations tends to obscure the simplicity of the communication pattern, making the program far from being obviously correct. This is not an original observation—Hoare has similar remarks in his seminal paper and book [3] on CSP—but it gets to the heart of the question considered here.

Further evidence for the advantages of CSP-style programming comes from the experience with the Plan 9 operating system [5], developed at Bell Labs. Hoare's work on CSP was a direct influence on the user-level threads library, `libthread`, provided in Plan 9 [1]. This has been used for several systems programs: examples include the Plan 9 window manager, and an ongoing wireless protocol implementation. Describing an early version of the window manager, Pike remarks [4]: "*The structure of the system – a set of concurrent processes communicating on synchronous channels – makes the control of multiple complex inputs decomposable into small, easily understood components whose design is chosen by the programmer, not the interface.*".

The Plan 9 library allows for the dynamic creation of processes and channels. It also introduces a co-routine structure for threads within a process. Processes are scheduled preemptively, while threads are scheduled cooperatively. The effect is as if a thread, once scheduled, holds a lock on the scheduler. This simplifies programming: thread code can use standard library operations, which are not required to be "thread-safe" (i.e., safe under preemption), before explicitly yielding to another thread. Moreover, there are fewer context switches between threads, all of which occur by design, at points where shared data structures are known to be in a consistent state.

```
process P(channel toBarrier, channel fromBarrier){
    ... code before barrier ...
    toBarrier ! Arrived;
    fromBarrier ? Proceed;
    ... code after barrier ...
}
process barrier(int N, channel toBarrier[], channel fromBarrier[]){
  int n,i; // n is the number of arrived processes

  while(true){
    n=0;
    while(n!=N){ // wait for arrival from some process k.
      (alt k: toBarrier[k] ? Arrived -> n++)
    }
    for(i=0;i<N;i++){fromBarrier[i] ! Proceed;}
  }
}
```

Figure 2: A barrier.

An empirical observation about this style of programming—which makes it particularly attractive for model checking—is that the synchronization pattern of a program is often largely data-independent. (Each message can be viewed as having a tag and associated data. E.g., in Figure 3, the `Passenger(p)` message has tag `Passenger` and data `p`.) Data independence makes it easier to extract a finite-state "synchronization skeleton" from a C program, which can analyzed for synchronization errors using standard model checking tools.

The original CSP model does not permit shared data; thus, a data race is an impossibility. However, disallowing sharing can be inefficient in several circumstances. For instance, in a packet pipeline with no data sharing, the input packet must be copied at each transfer. The Plan 9 library, therefore, allows pointers to shared data to be sent through channels. This, of course, raises the possibility of a data race: two processes or threads might both hold a pointer to the same data item.

However, as all synchronization is explicit, programmers tend to follow a discipline where each data item has a single owner at any point in time; ownership is transferred when a pointer to the data is sent to another thread. As a consequence of this disciplined approach, data races are rare in Plan 9 programs.

In summary, the CSP/Plan 9 approach often results in concurrent code which directly and clearly expresses the desired orchestration between components. It also happens to result in code which is easier to analyze than that based on locks or monitors, which I believe is not an accident.

Acknowledgements: Thanks go to my colleagues at Bell Labs, especially to Al Aho, Glenn Bruns, Dennis Dams, Sape Mullender, and Peter Bosch, for many illuminating discussions about concurrency and the Plan 9 model. This work was supported, in part, by NSF grant CCR-0341658.

```
process Taxi(int index, channel fromTaxi, channel fromDispatcher){
  fromTaxi ! Free;
  while(true){
     fromDispatcher? Passenger(p);
     drive(p);
     fromTaxi ! Free;
  }
}
// this process places free taxi's in a queue
process Queue(channel fromTaxi[], channel taxiQueue){
  while(true){
        (alt t: fromTaxi[t] ? Free -> taxiQueue ! Free(t))
  }
}

process Dispatcher(channel passengers, channel taxiQueue,
                   channel fromDispatcher[])
{
  while(true){
    passengers ? Passenger(p);
    taxiQueue ? Free(t); // wait for a free taxi
    fromDispatcher[t] ! Passenger(p);
  }
}
```

Figure 3: A taxi rank.

References

[1] Russ Cox. Bell Labs and CSP Threads. `http://swtch.com/~rsc/thread/`.

[2] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[3] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. available at http://www.usingcsp.com.

[4] R. Pike. A concurrent window system. *Computing Systems, 2(2) 133-153.*, 2(2):133–153, 1989.

[5] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(2):221–254, 1995. Manuals and source at http://plan9.bell-labs.com/plan9/.