Introduction to Concurrent Programming

Rob Pike
Computing Sciences Research Center
Bell Labs
Lucent Technologies
rob@plan9.bell-labs.com

February 2, 2000

Overview

The world runs in parallel, but our usual model of software does not. Programming languages are sequential.

This mismatch makes it hard to write systems software that provides the interface between a computer (or user) and the world.

Solutions: processes, threads, concurrency, semaphores, spin locks, message-passing. But how do we use these things?

Real problem: need an approach to writing concurrent software that guides our design and implementation.

We will present our model for designing concurrent software. It's been used in several languages for over a decade, producing everything from symbolic algebra packages to window systems.

This course is not about parallel algorithms or using multiprocessors to run programs faster. It is about using the power of processes and communication to design elegant, responsive, reliable systems.

History (Biased towards Systems)

Dijkstra: guarded commands, 1976.

Hoare: Communicating Sequential Processes (CSP), (paper) 1978. Run multiple communicating guarded command sets in parallel.

Hoare: CSP Book, 1985. Addition of channels to the model, rather than directly talking to processes.

Cardelli and Pike: Squeak, 1983. Application of CSP model to user interfaces.

Pike: Concurrent Window System, (paper) 1988. Application of Squeak approach to systems software.

Pike: Newsqueak, 1989. Interpreted language; used to write toy window system.

Winterbottom: Alef, 1994. True compiled concurrent language, used to write production systems software.

Mullender: Thread library, 1999. Retrofit to C for general usability.

Other models exist

Our approach is not the only way. There are many others; here is a sampling:

Languages:

- Occam
- CCS
- Esterel
- Concurrent ML
- Algol 68
- Ada
- Java

Techniques:

- Fork/join
- Semaphores
- Asynchronous message passing
- Condition Variables
- RPC

Other models exist (2)

Packages:

- POSIX and other Unix threads
- RPC

Operating Systems:

- Amoeba (Mullender & Tanenbaum)
- V (Cheriton)
- Topaz (Redell et al.)
- Saguaro (Andrews)
- SOS (Shapiro)

Notation

In this lecture, I will use a simplified notation (close to Newsqueak) to concentrate on the ideas.

In the next lecture, Sape Mullender will introduce the notation we will use to write the programs (in particular your exercises) in this course.

Function to Process

Consider a function call

```
result = func(arg)
```

The computation that needs `result` must wait until the function completes before it can proceed.

What if it doesn't need the answer yet? Want to be able to do something like:

```
func(arg)
```

do something else while `func` runs
when needed, use result; wait if `func` not done yet

This idea can be broken into three parts:

1. Starting the execution of a function while proceeding one's own calculation. (Process)
2. Retrieving values from functions/processes executing independently of us. (Communication)
3. Waiting for things to complete. (Synchronization)

Of course, once we've separated the function's execution, we can run it remotely - RPC.

Process

Famously hard to define. Lampson: "A locus of points of control." We’ll assume you know what one is, and worry about details, not the big picture.

In our world, a process is a function, and the functions it calls, executing with a private stack but global memory. (However, our programming model will avoid the global memory because it is unsynchronized.)

Function to Process, Part 2.

Four things to do with a function:

1. Call it. (Function call.)
2. Call it and let it run independently of us. (Process creation; `fork()` in Unix terminology.)
3. Replace the current calculation by the function, like tail recursion except that the function need not be the same:

```
double sin(double x)
{
    return cos(x - PI/2.);
}
```

(Called `exec` in Unix, `become` in Newsqueak.) [Generalized return statement.]

4. A function `A` can call `B`, and then `B` can return to where `A` left off; then `A` can return to where `B` left off, etc. Not really a call, but a combined call/resume. (Coroutines; Unix pipelines and also operating system schedulers.) [Stack switching.]

Our attention will concentrate on number 2.

Process management

To create a process, annotate a function call with a `begin` keyword:

```
   begin func(arg)
```

There is no return value from the function, and the arguments are the best way to deliver information to the new process. The next statement of the creating process will execute in parallel with the first statement of the new process.

To terminate a process, two possibilities:

1. Return from the top-level function in the process.
2. Call an explicit implementation-defined process-terminating function.

Communication channels

To communicate between processes, we use `channels`, declared like this:

```
c: chan of int;
```

Now `c` is a channel that can pass integer values (only).

A channel may be created for any type:

```
c1: chan of array of int;
c2: chan of chan of int;
```

etc.

To send a value, we say:

```
c1 <-= val;
```

To receive the value again, we say:

```
x = <-c1;
```

`<-` is the communication operator: postfix for send, prefix for receive.

Note that the same variable (`c1`) is used to send and to receive.

Interlude: What is memory?

In our idealized model, memory is shared between processes. A value may be passed between two processes using a channel. However, there is no protection against multiple processes changing a value simultaneously.

In our idealized model, references to memory may be shared between processes, but again there is no protection against multiple processes changing the referenced values simultaneously.

Values on the stack are initially private to a process, but may be passed to another process over a channel. References to the stack are also private, but passing stack references between processes has undefined semantics.

In the next lecture, we will see how these rules change in a real implementation.

Synchronization

When a value is transmitted on a channel, the sending and receiving processes *synchronize*: both most be at the right point (the send or the receive statement) for the transfer to take place.

If sending and no receiver, sender blocks until receiver executes receive.
If receiving and no sender, receiver blocks until sender executes send.

Note: process cannot send to itself because one process cannot simultaneously execute a send and a receive.

This is called synchronous communication. We will see some asynchronous communication later in the course.

Note that sending a value addresses both *communication* and *synchronization*.

Other models of synchronization

Like recursion and iteration, various synchronization methods are equivalent − any one can be built from any other − but some are better for some problems, others for others. We won't dwell on them all here, but you should know some names:

Semaphore

Spin lock

Queued lock

Condition variables

Multi-way sending: the select or alt statement

What if I wish to receive from either of two things simultaneously? Use a `select` (called `alt` in Alef and the thread library):

```
select{
case i = <-c1:
    print("got ", i, " from c1\n");
case i = <-c2:
    print("got ", i, " from c2\n");
}
```

Sends can also be used in `selects`, and sends and receives can be mixed.

The definition of `select` is much like the definition of Dijkstra’s guarded commands. The entire `select` statement blocks until one of the communications can proceed (has a sender and a receiver). That communication then occurs, and the associated statement executes.

If multiple cases are simultaneously executable, one is chosen *at random*.

A simple example

```
run := prog(data: chan of array of char,
    exit: chan of unit)
{
    str: array of char;
    for(;;)
        select{
        case str = <-data:
            print("received", str, "\n");
        case <-exit:
            become unit; # return
        }
};

data := mk(chan of array of char);
exit := mk(chan of unit);

begin run(data, exit);

data <-= "hello";
data <-= "world";
exit <-= unit;
```

Channels as capabilities

Channels are a little like file descriptors or capabilities. Given a channel, we don’t know who’s at the other end (what file is open) but we do know that we can read it to get some useful information.

If that channel points to some service, it provides an interface to that service, somewhat like a capability. We can hand the channel to another process, and thereby give away access to the service.

Example: prime sieve

Part of today’s exercise involves exploring this idea. Here, in words, is the idea of the program, which exploits the concept of using channels to represent sources of data.

Our goal is a program that prints primes. We observe that the integers 2, 3, 4, ... could be the successive values read from a channel. The first prime is the first integer in the series, 2.

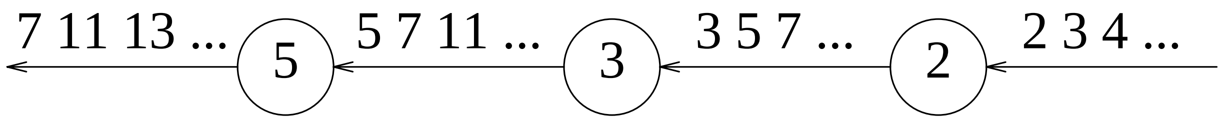
Create a process to print that value, then pass along, on a different channel, the integers that are non-zero modulo 2. That is, the process copies the input stream to the output stream, removing the numbers that are zero modulo 2.

The next prime is the first integer that emerges from the output stream of the first process, 3. Create a process that prints that number and passes the rest along, removing numbers that are zero modulo 3.

The next prime is the first integer that emerges from the output stream of the second process, 5....

Example continued

In other words, the program is a dynamic chain of processes, each of which filters out values divisible by its prime.



This is the concurrent programming version of the Sieve of Eratosthenes, and your assignment is to implement it.

Notice the abstraction provided by channels as interfaces to processes. This idea is central to concurrent programming as we practice it.

Next: Concurrent programming in C