Concurrent Programming in the Bell Labs CSP style

Russ Cox

rsc@mit.edu

PDOS Group Meeting
February 5, 2002

http://plan9.bell-labs.com/˜rsc/thread.html

Overview

World runs in parallel; our typical models of programming do not.

Lots of solutions: processes, threads, semaphores, spin locks, condition variables, message-passing, monitors, ...

Present Bell Labs model for concurrent programs.

Credo

The most important readers of a program are people.

‘‘We observe simply that a program usually has to be read several times in the process of getting it debugged. The harder it is for *people* to grasp the intent of any given section, the longer it will be before the program becomes operational.’’
— Kernighan and Plauger, 1974.

‘‘Write programs for people first, computers second.’’
— McConnell, 1993.

‘‘Programs are read not only by computers but also by programmers. A well-written program is easier to understand and to modify than a poorly-written one. ... Code should be clear and simple—straightforward logic, natural expression, conventional language use, meaningful names, neat formatting, helpful comments—and it should avoid clever tricks and unusual constructions.’’
— Kernighan and Pike, 1999.

In rare cases, performance or other concerns keep code from being as readable as possible.

Why should you care?

From a programming language and style point of view, the interface presented by libasync is very detail-oriented with no real benefit.

> Like writing a word processor in assembly: too much detail, language doesn't do enough for the programmer.

Libasync is really doing the jobs of (at least) two libraries:

> concurrency support - multiple loci of execution
>
> asynchrony support - overlapping I/O with computation

Let's try again with a helpful concurrency model.

Layer asynchrony on top.

Roadmap from here

Introduction

Bell Labs threads

Example: Chord network mapper

Thread library model

Layering asynchrony on top of threads

Summary

What this is not

Andrew Birrell, ‘‘An Introduction to Programming with Threads,’’ DEC SRC Technical Report #35, January 6, 1989.

You must unlearn what you have learned.

- too much detail
- language doesn’t do enough for the programmer

What this is

A different style of programming.

A different way to think about programming.

Not exclusive to systems.

- really slick power series calculator
- don’t have time to discuss, see web page

Hoare, 1978

‘‘Communicating Sequential Processes,’’ CACM 21(8), August 1978.

Lots of ways to program multiprocessors:

communication: shared memory, mainly

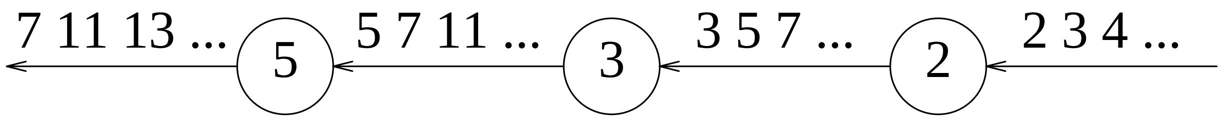
synchronization: semaphores, critical regions, monitors, etc.

Hoare: one primitive, synchronous communication over unbuffered channels.

Example for Flavor

Parallel version of sieve of Eratosthenes:

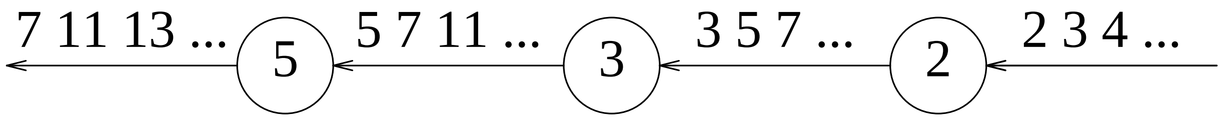
One process for each prime found so far; filters out multiples of that prime from a stream carrying the natural numbers.



```
# generate 2, 3, ...
counter(out: chan of int)
{
    for(i=2;; i++)
        out <-= i;
}

# copy in to out, removing mutiples of p
filter(in, out: chan of int, p: int)
{
    for(;;){
        n = <-in;
        if(n%p)
            out <-= n;
    }
}
```

Example, cont.



```
# return new channel behaving like c
# but without the multiples of p
applyfilter(c: chan of int, p: int)
{
    nc = new chan of int;
    spawn filter(c, nc, p);
    return nc;
}

# print primes by building sieve chain
sieve()
{
    c = new chan of int;
    spawn counter(c);
    for(;;){
        p = <-c;
        print p;
        c = applyfilter(c, p);
    }
}
```

Language Development

CSP: Hoare’s 1978 paper, based on Dijkstra.

Squeak: CSP applied to scripting GUIs (generate C).

Newsqueak: a full interpreted programming language.

- syntax basically as in the example
- channels are first-class objects

Alef: compiled language, for systems.

Limbo: portable, bytecode-based, for embedded systems

- language for Inferno
- equivalent in goal to original Java
- syntax almost identical to example

Libthread: translate Alef programs to C.

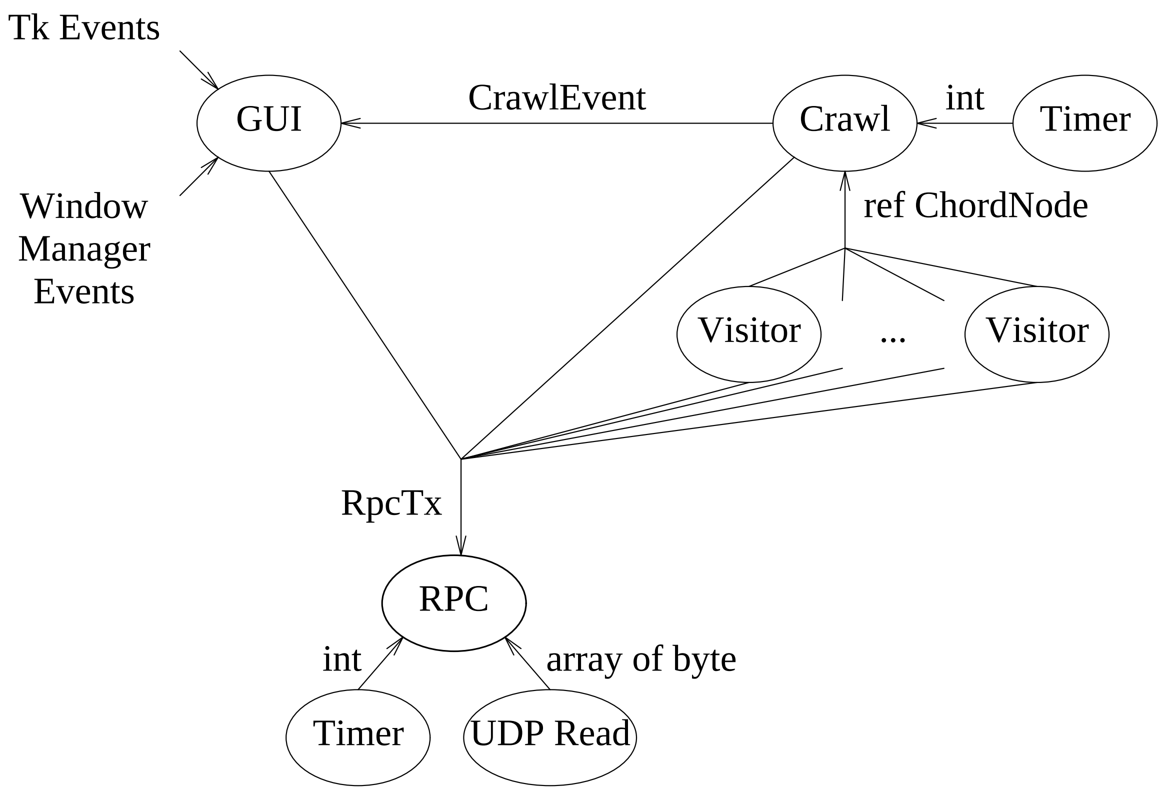
- Lose syntactic sugar.

Others: concurrent ML, Moby, ...

A Chord Network Crawler and Mapper

Written in Fall 2001, from scratch, in Limbo.

Inferno is threading heaven: threads are preemptively scheduled yet very cheap.



(Channels are labeled with type of data sent over them.)

Overall program does a complex job, but the individual pieces are all very simple.

RPC Supporting Pieces

Input generators.

```
# generate clock ticks
timer(c: chan of int)
{
    for(;;){
        sys->sleep(100);
        c <-= sys->millisec();
    }
}

# watch a udp port
udpreader(fd: ref FD, c: chan of array of byte)
{
    m = array[UDPmax] of byte;
    while((n = read(fd, m, len m)) >= UDPhdrsize)
        c <-= copy(m[0:n])
}
```

RPC

The `rpcchan` is the exported interface to the RPC module. Clients send (request, address, return-channel, timeout) and then wait for a response along the return channel.

Very object oriented (in the original Alan Kay sense).

```
#
rpcmux()
{
    spawn timer(timerchan);
    spawn udpreader(netfd, udpchan);
    for(;;)
        alt{
        (req, addr, c, timeout) = <-rpcchan =>
            send req to addr
            add to list of pending requests
        t = <-timerchan =>
            look through list,
            send timeout notifications
        msg = <-udpchan =>
            unpack, find call on list,
            send notification
        }
}
```

Chord RPC Transaction

```
tx(rpcchan: chan of RpcTx, addr: string, req: ref ChordReq):
  (string, ref ChordRep)
{
    msg = pack(req);
    c = new chan of (string, ref RpcMsg.Reply);

    # hand off to RPC
    rpcchan <-= (msg, addr, c, timeout);

    # wait for RPC response
    (err, msg) = <-c;

    if(err != nil)
        return (err, nil);
    return (nil, unpack(msg));
}
```

One Chord RPC Call

```
succ(c: chan of RpcTx, v: ref ChordNode):
  (string, Status, ref ChordNode)
{
    req = ChordReq.Succ(v.id);
    (err, rep) = tx(c, v.addr, req);
    if(err != nil)
        return (err, RpcFailure, nil);
    return (nil, rep.status, rep.node);
}
```

Chord Find Successor

```
findsucc(mux: chan of RpcTx, v: ref ChordNode,
  id: ref ChordID): ref ChordNode
{
    # walk around ring looking for id
    n = v;
    do{
        lastn = n;
        (e, s, n) = closestpred(mux, n, id);
    }while(lastn.id != n.id);

    # have predecessor of id; take successor
    (e, s, n) = succ(mux, n);
    return n;
}
```

Why do we care?

Libasync is also a concurrency library. Each sequence of callbacks is effectively a thread of control.

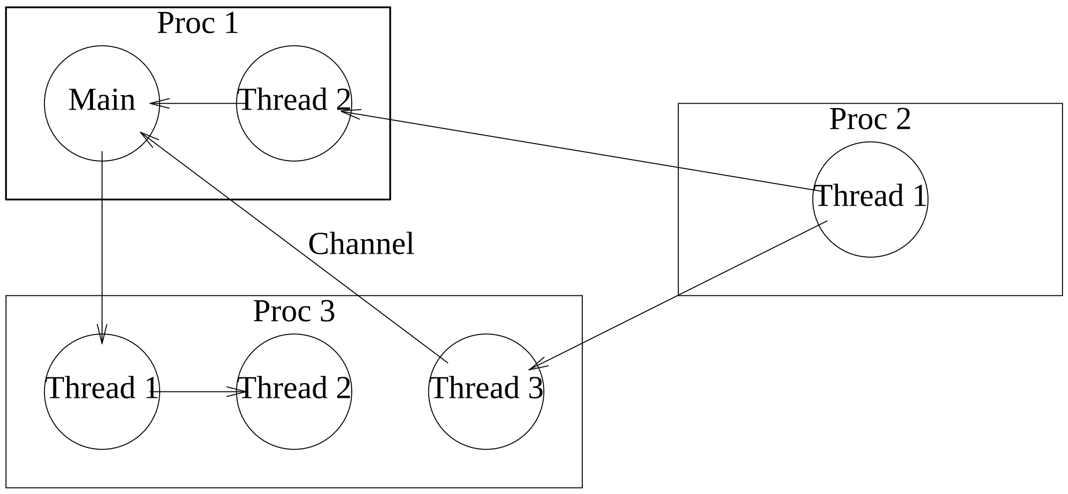
Libasync pushes the burden of managing threads onto the user, in the form of `wrap` continuations.

Between the first and second editions of their *Elements of Programming Style*, Kernighan and Plauger added the tip ‘‘Use do and do...while to emphasize the presence of loops.’’

24 years later, it is *pathetic* that libasync effectively prohibits the use of standard looping constructs.

Instead, we can layer asynchrony on top of decent threading support.

Libthread System Model



Procs: preemptively scheduled by operating system, shared address space.

Threads: cooperatively scheduled coroutines within a proc.

Channels: communication channels as described before, may be buffered or not.

Libthread system model, II

No built-in asynchrony: if one thread blocks in a system call, that whole proc blocks.

Usually no need for locks, since threads yield cooperatively; arrange so shared data is only modified inside one proc.

Yield and schedule via `setjmp` and `longjmp` equivalents.

Convention defines what yields: channel operations, `yield` function, maybe others (`t`-routines in future example).

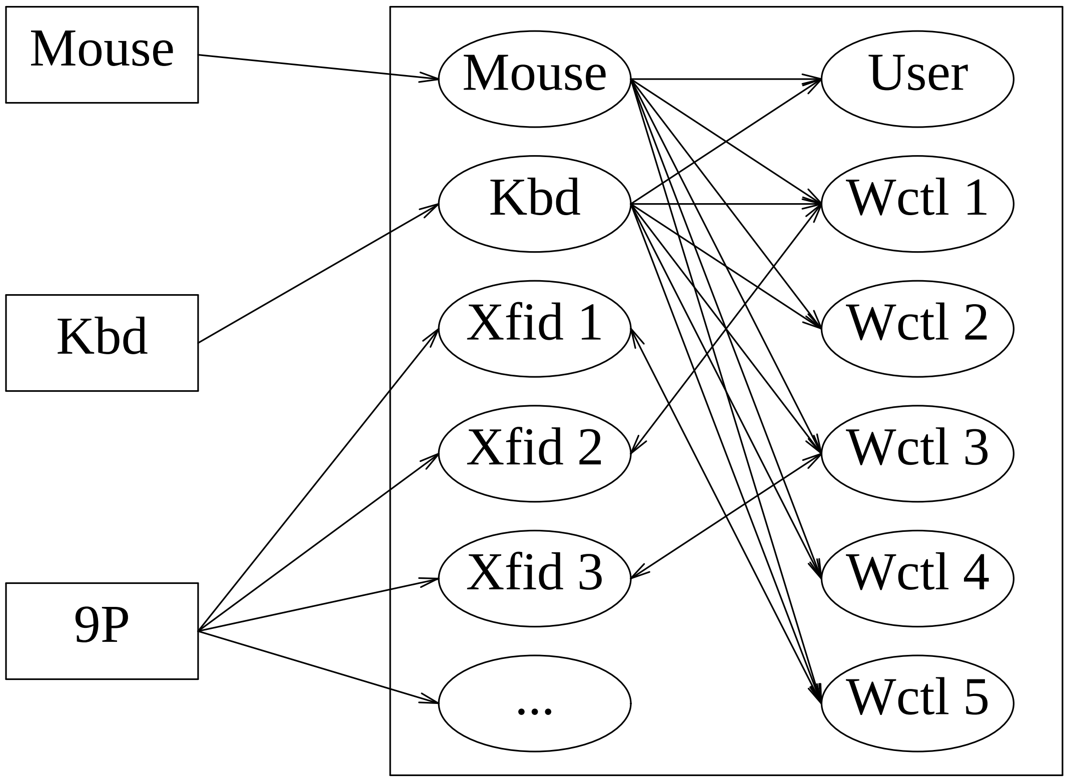
Non-growing stacks. Stack overflow is occasionally a problem, hard to be extremely memory efficient. Compiler help would be nice.

Using Libthread on Plan 9

No `select` or `poll` system calls.

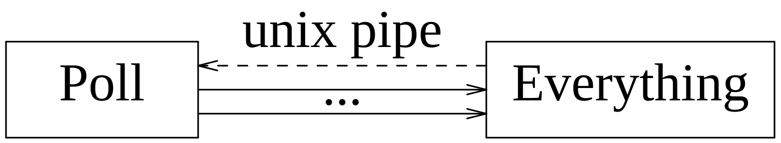
One master process containing most threads.

One slave process for each blocking I/O source, feeds a thread in the master process.



Asynchrony using Libthread

Same idea, but can coalesce the slave processes into a single process.



Send new fds to poll proc over Unix pipe.

Async Poller

```
typedef struct Fdreq Fdreq;
struct Fdreq
{
    int fd;     /* fd to watch */
    Channel *c; /* who to notify */
    int event;  /* e.g., POLLRDNORM */
};

void
pollproc(void)
{
    for(;;){
        switch(i = poll(p, np, INFTIM)){
        case -1:
            ...

        case 0:  /* pollpipe */
            read(p[0].fd, &fr, sizeof fr);
            add fr.fd to poll list
            break;

        default: /* watched fd is ready */
            remove p[i] from poll list
            sendp(c[i], 1);
            break;
        }
    }
}
```

Asynchronous Read

```
typedef struct Fd Fd;
struct Fd
{
    int fd;     /* has O_NONBLOCK flag */
    Channel *c; /* for comm with poller */
};

long
tread(Fd *f, void *a, long n)
{
    while((m = read(f->fd, a, n))==-1
    && errno==EAGAIN){
        fr = mkfdreq(f->fd, f->c, POLLRDNORM);
        write(pollpipe, &fr, sizeof fr);
        recvul(f->c);
    }
    return m;
}
```

Asynchronous Threaded Web Server

```
void
threadmain(int argc, char **argv)
{
    Fd *s, *fd;

    ...
    s = tcplisten(port);
    for(;;){
        fd = taccept(s);
        threadcreate(webserve, fd, Stacksize);
    }
}

void
webserve(Fd *fd)
{
    for(n=0; n<sizeof buf; n+=m){
        if(haveget(buf, n))
            break;
        m = tread(s, buf+n, sizeof buf-n);
    }

    r = topen(req(buf, n), O_RDONLY));
    while((n = tread(r, buf, sizeof buf)) > 0)
        twrite(fd, buf, n);

    tclose(r);
    tclose(fd);
    threadexits(0);
}
```

Summary

Libasync is really (at least) two libraries intertwined:

Concurrency (`wrap`, call backs, etc.)

Asynchronous I/O

Good concurrency support is powerful by itself.

Can layer asynchronous I/O above concurrency.

Libthread approach lets you keep vital language features like loops and function calls.