Fast Inter-Process Communication

This week:
System calls and IPC

- You will be creating a system call interface.
- In Barrelfish (as in microkernels) this is mostly done by calling a server process
- This requires sending messages between processes
- This needs to be fast…

UNIX-STYLE IPC

Lots of Unix IPC mechanisms

- Pipes
- Signals
- Unix-domain sockets
- POSIX semaphores
- FIFOs (named pipes)
- Shared memory segments
- System V semaphore sets
- POSIX message queues
- System V message queues
- etc.

And many, many more in Windows!

IPC is usually heavyweight

IPC in conventional systems tends to combine:

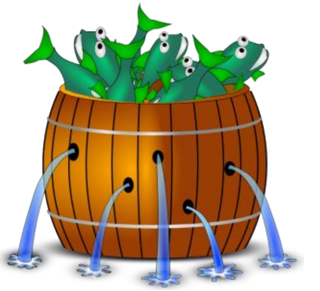
- **Notification**: (telling the destination process that something has happened)
- **Scheduling**: (changing the current runnable status of the destination, or source)
- **Data transfer**: (actually conveying a message payload)

Unix doesn't have a *lightweight* IPC mechanism

IPC in Unix is usually polled

- Blocking `read()/recv()` or `select()/poll()`
- Signals are the nearest thing to upcalls, but...
  - Dedicated (small) stack
  - Limited number of syscalls available (e.g. semaphores)
  - Calling out with `longjmp()` problematic,
- Unix lacks a good upcall / event delivery mechanism

Basic questions



- How to perform cross-domain invocations?
- Does the calling domain/process block?
- Is the scheduler involved?
- Is more than one thread involved?

And later:

- What happens across physical processors?

Remote Procedure Call

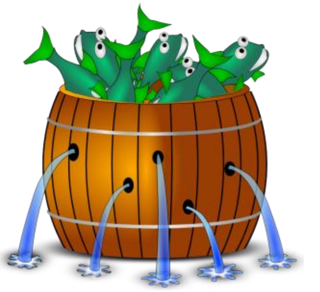
1. Caller:
   – Marshals arguments in to a buffer
   – Sends buffer over the network
2. Receiver:
   – Unmarshals arguments
   – Calls procedure
   – Marshals return value(s)
   – Sends it back
3. Caller:
   – Unmarshals return value(s)
   – Returns

- Basis for most distributed systems.
- Why not apply to intra-OS communication?

Local RPC: High overhead!

- Stubs copy lots of data
  - not an issue for the network
- Message buffers usually copied through the kernel
  - 4 copies!
- Access validation
- Message transfer
  - queueing/dequeuing of messages
- Scheduling
  - programmer sees thread crossing domains
  - system actually rendezvous's two threads in different domains
- Context switch
  - x 2
- Dispatch
  - Find a receiver thread to interpret message, and either dispatch another thread, or leave another one waiting for more messages

Slow!



LIGHTWEIGHT RPC

Lightweight RPC (LRPC): Basic concepts

- Simple control transfer: client's thread executes in server's domain
- Simple data transfer: shared argument stack, plus registers
- Simple stubs: i.e. highly optimized marshalling
- Design for concurrency: Avoids shared data structures

Most messages are short

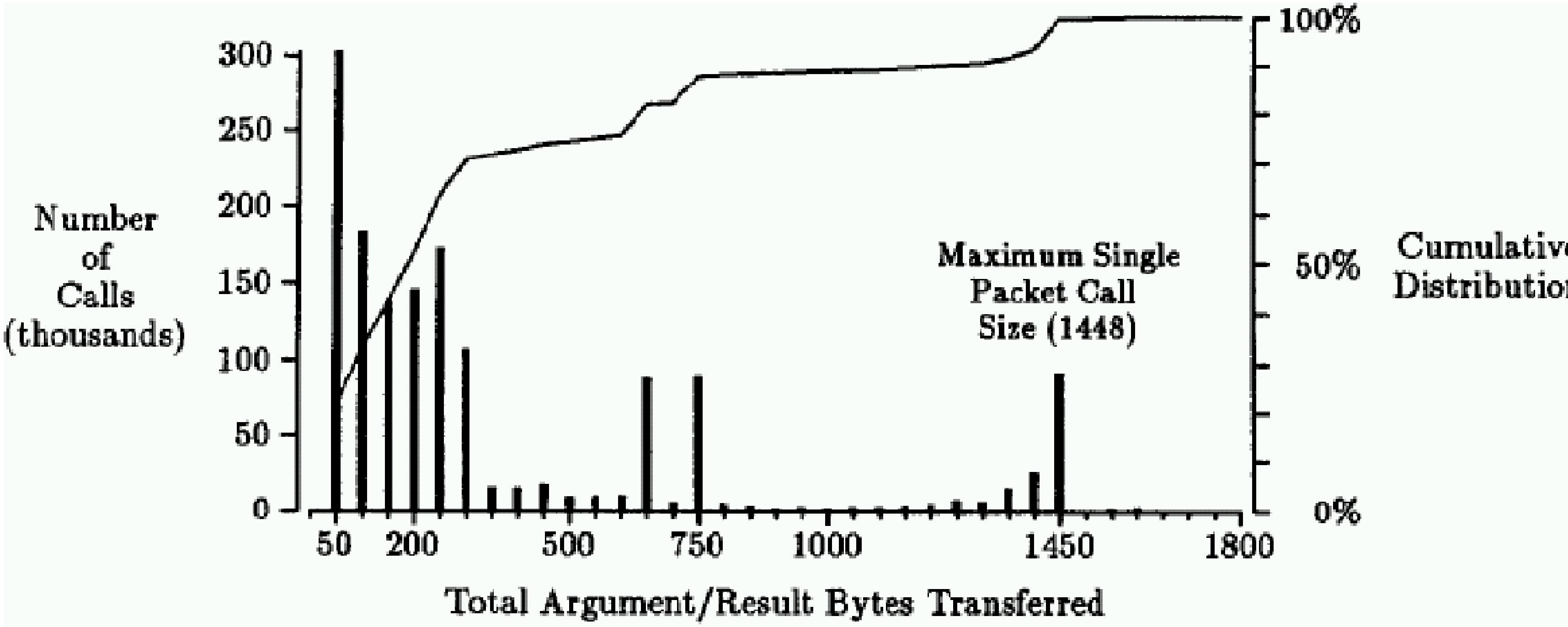


Fig. 1. RPC size distribution.

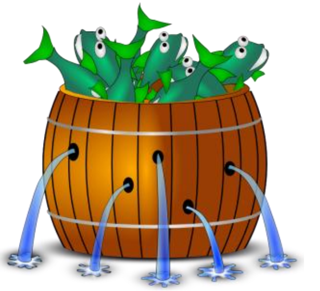
LRPC Binding: connection setup phase

- *Procedure Descriptors* (PDs) registered with kernel for each procedure in the called interface
- For each PD, *argument stacks* (A-stacks) are preallocated and mapped read/write in both domains
- Kernel preallocates linkage records for return from A-stacks
- Returns A-stack list to client as (unforgeable) Binding Object

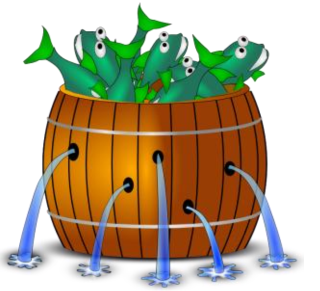
Calling sequence (all on client thread)

1. Verify Binding Object, find correct PD
2. Verify A-Stack, find corresponding linkage
3. Ensure no other thread using that A-stack/linkage pair
4. Put caller’s return addr and stack pointer in linkage
5. Push linkage on to thread control block’s stack (for nested calls)
6. Find an execution stack (E-stack) in server’s domain
7. Update thread’s SP to run off E-stack
8. Perform address space switch to server domain
9. Upcall server’s stub at address given in PD

LRPC discussion

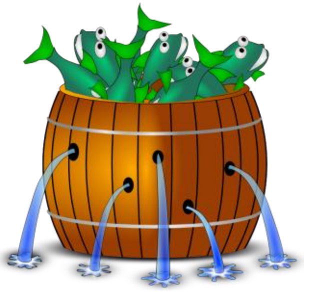


- Main kernel housekeeping task is allocating A-stacks and E-stacks
- Shared A-stacks reduce copying of data while still safe
- Stubs incorporated other optimizations (see paper)
- Address space switch is most of the overhead (no TLB tags)
- For multiprocessors:
  - Check for processor idling on server domain
  - If so, swap calling and idling threads
    - (note: thread migration was very cheap on the Firefly!)
  - Same trick applies on return path



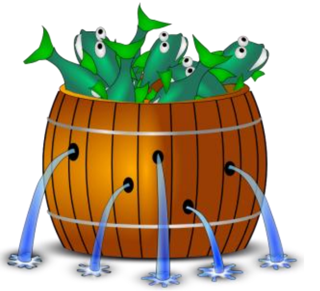
SYNCHRONOUS IPC IN L4

L4 synchronous RPC



- L4 pushed this idea further (for uniprocessor case)
- No kernel-allocated A-stack: server must have waiting thread (no upcalls possible)
- RPC just exchanges register contents with calling thread
- *Synchronous RPC:* calling thread blocks, waits for reply
- Scheduler bypassed completely
  - The infamous “null RPC” microbenchmark
  - Latency of a single call, nothing else happening
- Design couples notification, transfer, scheduling

IPC overview

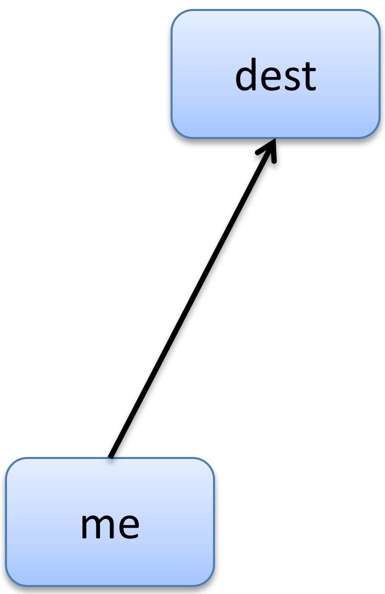


- L4 provides a single system call for all IPC
  - Synchronous and unbuffered *(apart from async notify)*
  - Has a send and a receive component
  - Either send or receive may be omitted
- Receive may specify:
  - A specific thread ID from which to receive (“closed receive”)
  - Willingness to receive from any thread (“open wait”)

Logical IPC operations

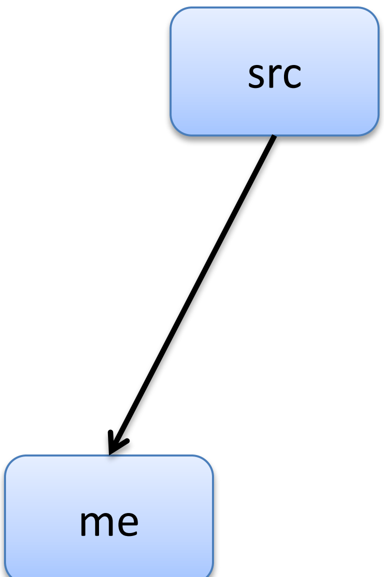
- `Send` sends a message to a specific thread
- `Receive` "closed" receive from a specific sender
- `Wait` "open" receive from any sender
- `Call` send to and wait for reply from specific thread
  - Typical client RPC operation
- `ReplyWait` send to specific thread, "open" receive
  - Typical server operation

IPC Send



to:	dest
FromSpecifier:	*nil*

IPC Receive (closed)

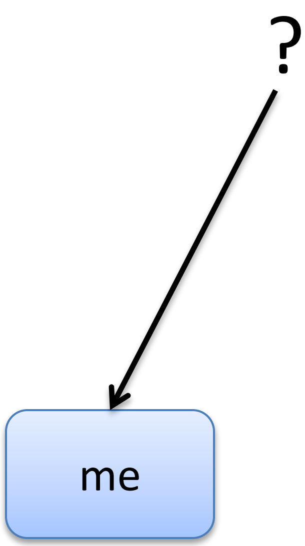


to:	*nil*
FromSpecifier:	src

IPC Wait (open)

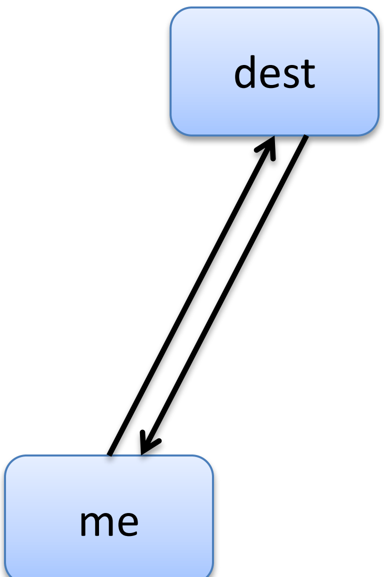
to: *nil*

FromSpecifier: *any*

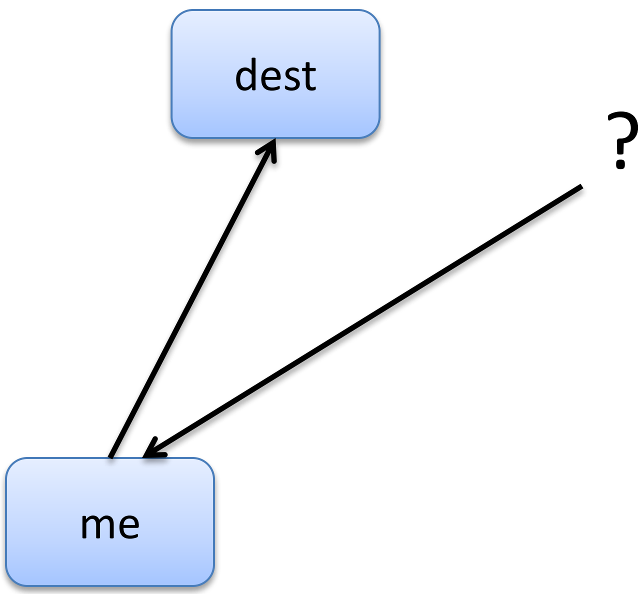


IPC Call

to:	dest
FromSpecifier:	dest

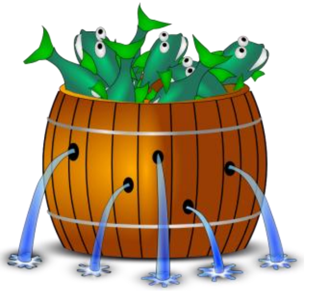


IPC ReplyWait



to:	dest
FromSpecifier:	*any*

Passing device interrupts to a user-space driver

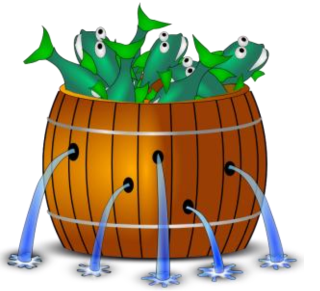


- Basic idea: an interrupt is a message.
  - Driver registers an IPC endpoint with the kernel
  - 1st-level IRQ handler in kernel
    - Masks the interrupt
    - Creates a message
    - Makes the driver runnable
  - Driver is dispatched
    - Handles interrupt
    - Replies to message to acknowledge the interrupt
  - Kernel unmasks the interrupt

IPC message registers (MRs)

- Virtual registers
  - Not necessarily backed by CPU registers
  - Part of thread state
- On ARMv7-A: 6 physical registers, rest in UTCB
- Actual number is a system configuration parameter
  - At least 8, no more than 64
- Contents of MRs form message
  - First MR stores the message tag defining message size etc.
  - Rest are untyped words, not normally interpreted by the kernel
  - Kernel protocols define semantics in some cases
- IPC just copies data from sender's to receiver's MRs

Asynchronous Notification



- Very restricted form of asynchronous IPC
  - Delivered without blocking sender
  - Delivered immediately, directly to receiver's UTCB
  - Message consists of a bit mask ORed to the receiver:
    - receiver.NotifyBits |= sender.MR1
  - No effect if receiver's bits are already set
  - Receiver can prevent asynchronous notification by setting a flag in its UTCB

LIGHTWEIGHT MESSAGE PASSING IN BARRELFISH

RPC on Barrelfish?

- How to integrate with upcalls (activations)?
  - N.B. LRPC also used scheduler activations…
  - Threads are now user-space things
- Authorization check
  - RPCs are sent to capabilities
- End-points are capabilities
  - RPC is connection oriented
  - not connectionless, as in L4

LMP: Barrelfish local RPC

- On a single core:
  - IPC is asynchronous: one-way messaging only
  - RPC implemented at higher level in stubs
- Message is queued at destination, may cause an upcall
  - L4-style fast path: thread can optionally wait for a message
- Unlike L4, can decouple notification & transfer
  - Scheduler is always involved (but . . .)
- Between cores: later in this course…

Local Message Passing

For Later Milestones

- Programs communicate by passing messages
- Messaging on one core is done via an **Endpoint capability**
- Usually wrapped by **Flounder** (another DSL!)
  - LMP channels (as opposed to UMP channels)
- LMP channels can transmit capabilities
- Messages are signalled on **waitsets**
  - Each domain has a default waitset

LMP Interface

`lmp_sendX()`	Send an *X* word payload
`lmp_chan_recv()`	Receive a message (that's already there)
`lmp_chan_register_recv()`	Register a callback, to be notified when a message arrives on a channel
`get_default_waitset()`	Get the default waitset for a domain

LMP Example: Send

N.B. Error checking omitted!

```
errval_t send_message(struct lmp_chan *c, uintptr_t *msgbuf,
                      size_t msg_words, size_t *words_sent)
{
    uintptr_t buf[LMP_MSG_LENGTH];
    if (msg_words > LMP_MSG_LENGTH) {
        msg_words = LMP_MSG_LENGTH;
    }
    memcpy(buf, msgbuf, msg_words*sizeof(uintptr_t));
    memset(buf+msg_words, 0, (LMP_MSG_LENGTH-msg_words));
    if (words_sent) { *words_sent = msg_words; }
    return lmp_chan_send9(c, LMP_SEND_FLAGS_DEFAULT, NULL_CAP,
                   buf[0], buf[1], buf[2], buf[3], buf[4],
                   buf[5], buf[6], buf[7], buf[8]);
}
```

LMP Example: Receive

- Error checking omitted
- Handler is deregistered when called, reregister after each message

```
errval_t recv_handler(void *arg)
{
    struct lmp_chan *lc = arg;
    struct lmp_recv_msg msg = LMP_RECV_MSG_INIT;
    struct capref cap;
    err = lmp_chan_recv(lc, &msg, &cap);
    if (err_is_fail(err) && lmp_err_is_transient(err)) {
        // reregister
        lmp_chan_register(lc, get_default_waitset(),
            MKCLOSURE(recv_handler, arg));
    }
    debug_printf("msg buflen %zu\n", msg.buf.msglen);
    debug_printf("msg->words[0] = 0x%lx\n", msg.words[0]);
    lmp_chan_register(lc, get_default_waitset(),
        MKCLOSURE(recv_handler, arg));
}

void some_func(void) {
     // assumption: we have channel here
     lmp_chan_register_recv(chan, get_default_waitset(),
         MKCLOSURE(recv_handler, chan));
}
```

Extract message from registers

Handle temporary receive failure

Process received message

LMP Example: Dispatch

Something like this is in most Barrelfish applications:

```
int main_loop(struct waitset *ws)
{
    // go into messaging main loop
    while (true) {
        err = event_dispatch(ws);
        if (err_is_fail(err)) {
            DEBUG_ERR(err, "in main event_dispatch loop");
            return EXIT_FAILURE;
        }
    }
    return EXIT_SUCCESS;
}

int main(void)
{
    // do initialization
    // ...
    // run messaging loop on default waitset
    return main_loop(get_default_waitset());
}
```

Stack Ripping

- Common in event-driven programming
  - The core logic is split across a chain of callbacks
- Quickly becomes unreadable
- Program state is no longer implicit in the stack
  - We end up **ripping** the relevant parts into continuations and callbacks

Lookup Example
Stack Ripped

```
void Lookup(NSChannel_t *c, char *name) {
     OnRecvLookupResponse(c, &ResponseHandler);
     // Store state needed by send handler
     c->st = name;
     OnSend(c, &SendHandler);
}

void ResponseHandler(NSChannel_t *c, int addr) {
     printf("Got response %d\n", addr);
}

void SendHandler(NSChannel_t *c) {
     if (OnSendLookupRequest(c, (char *)(c->st)) ==
         BUSY) {
         OnSend(c, &SendHandler);
     }
}
```

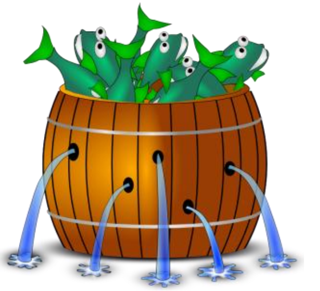
Executed when we get a response

Executed when the channel can send the message

Re-set SendHandler if we couldn't send after all

We can do better!

Composable Asynchronous IO for Native Languages



- T. Harris, M. Abadi, R. Isaacs, R. McIllroy; OOPSLA 2011
- AC —– Asynchronous C
- Make stack-ripped code look sequential
- New primitives:
  `async, do {} finish, cancel`
- Goal: make message-passing code scalable
- No multi-threading, the extensions just identify opportunities where multiple messages can be issued concurrently

Lookup Example Asynchronous

```
// Caution: functions ending in AC may block
void LookupAC(NSChannel_t *c, char *name) {
    int addr;
    SendLookupRequestAC(c, name);
    RecvLookupResponseAC(c, &addr);
    printf("Got response %d\n", addr);
}

void TwinLookupAC(NSChannel_t *c1, NSChannel_t *c2, char *name) {
    do {
        async LookupAC(c1, name); // S1
        async LookupAC(c2, name); // S2
    } finish;
    printf("Got both responses\n"); // S3
}
```

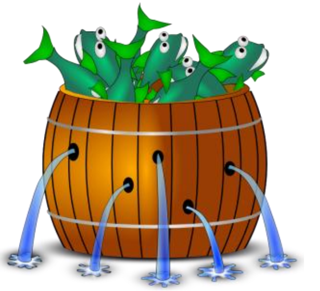
Sending and Receiving may block

Can run both lookups simultaneously

S3 won't be executed before both lookups finish

FURTHER READING

Further reading



- Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. ACM Trans. Comput. Syst. 2, 1 (February 1984), 39-59. http://dx.doi.org/10.1145/2080.357392
- M. Schroeder and M. Burrows. 1989. Performance of Firefly RPC. In Proceedings of the twelfth ACM symposium on Operating systems principles (SOSP '89). ACM, New York, NY, USA, 83-90. http://dx.doi.org/10.1145/74850.74859
- Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1990. Lightweight remote procedure call. ACM Trans. Comput. Syst. 8, 1 (February 1990), 37-55. http://dx.doi.org/10.1145/77648.77650
- Jochen Liedtke. 1993. Improving IPC by kernel design. In Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93). ACM, New York, NY, USA, 175-188. http://dx.doi.org/10.1145/168619.168633
- J. Liedtke. 1995. On micro-kernel construction. In Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP '95), Michael B. Jones (Ed.). ACM, New York, NY, USA, 237-250. http://dx.doi.org/10.1145/224056.224075
- Tim Harris, Martin Abadi, Rebecca Isaacs, and Ross McIlroy. 2011. AC: composable asynchronous IO for native languages. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11). ACM, New York, NY, USA, 903-920. http://dx.doi.org/10.1145/2048066.2048134