Systems@ETH Zürich

Bulk Inter-process Transport

Advanced Operating Systems
(263-3800-00)

Timothy Roscoe

Thursday 11 November 2010

We’ve seen fast IPC mechanisms last week

- What’s “fast”?
  - low latency
  - small messages
- But what about:
  - throughput?
  - large messages?

I/O and IPC

- Most of the low-latency, small message work has been done in classic IPC (server to client) situations.
  - Motivation: microkernel performance
  - Extends to I/O by turning interrupts into events
- High-throughput, large message work tended to start with I/O
  - In particular: multimedia networking
  - Extends to IPC via message paths
- Irony: both around the same time

Disclaimer

- Neither L4 nor Barrelfish (yet) have a standard bulk transport mechanism!
  - Barrelfish will look a lot like the techniques here
  - Current implementation is greatly simplified, quick'n'dirty version

Fbufs

[Druschel and Peterson, 1993]

- Not the first work – incorporates many previous ideas...
- Good because:
  - illustrates a lot of issues.
  - nice description of a particular technique
  - fast

Fbufs key ideas

1. Combine virtual page remapping with shared virtual memory
2. Exploit locality in IPC traffic
3. Achieve high throughput (what does this mean?)

Fbufs motivation

- High-speed network interfaces (ATM at the time)
- LRPC/URPC/L4 about lowering latency, for small arguments.
- Need throughput for large messages (e.g. packets)
- Realtime video, digital image retrieval, large scientific data sets
- Moving data across protection domains (microkernels)
- Can’t copy too much as CPU-Memory is bottleneck [Ousterhout, 1989]

Fbufs requirements

Systems@ETH Zürich

- Single, contiguous buffers and non-contiguous aggregates of buffers
- OK to use a special buffer allocator for I/O data
- At time of allocation, I/O data path is known, so transfer facility can employ a path-specific allocator
- I/O system can be designed to only use immutable buffers
- Asynchronous modification of a buffer by originating domain is a problem.
- Two solutions:
  - eager (raise protection on buffer when originator tranfers it)
  - lazy (raise protection when receiver requests it).
- Buffers should be pageable.

Prior art: page remapping

Systems@ETH Zürich

- Used in, e.g., the V system [Cheriton, 1988]
- Move rather than copy semantics
  - "linear typing"
  - cf. modern Singularity
- Sender can no longer access data once sent
- Expensive:
  - VMdata structures must be manipulated
  - pages must be reallocated/deallocated, and zeroed.
  - D&P recognized that data flow for I/O mostly unidirectional.

Prior art: copy-on-write

Systems@ETH Zürich

- Used in, e.g., Accent & Mach [Barrera, 1991]
- Provides copy semantics
- Requires actual copying if either side modify data.

- Page mapping and COWhard to get to go fast, particularly in a portable manner:
  - esp. from a network interface
  - original Mach implementation slower than copying!

Prior art: shared memory

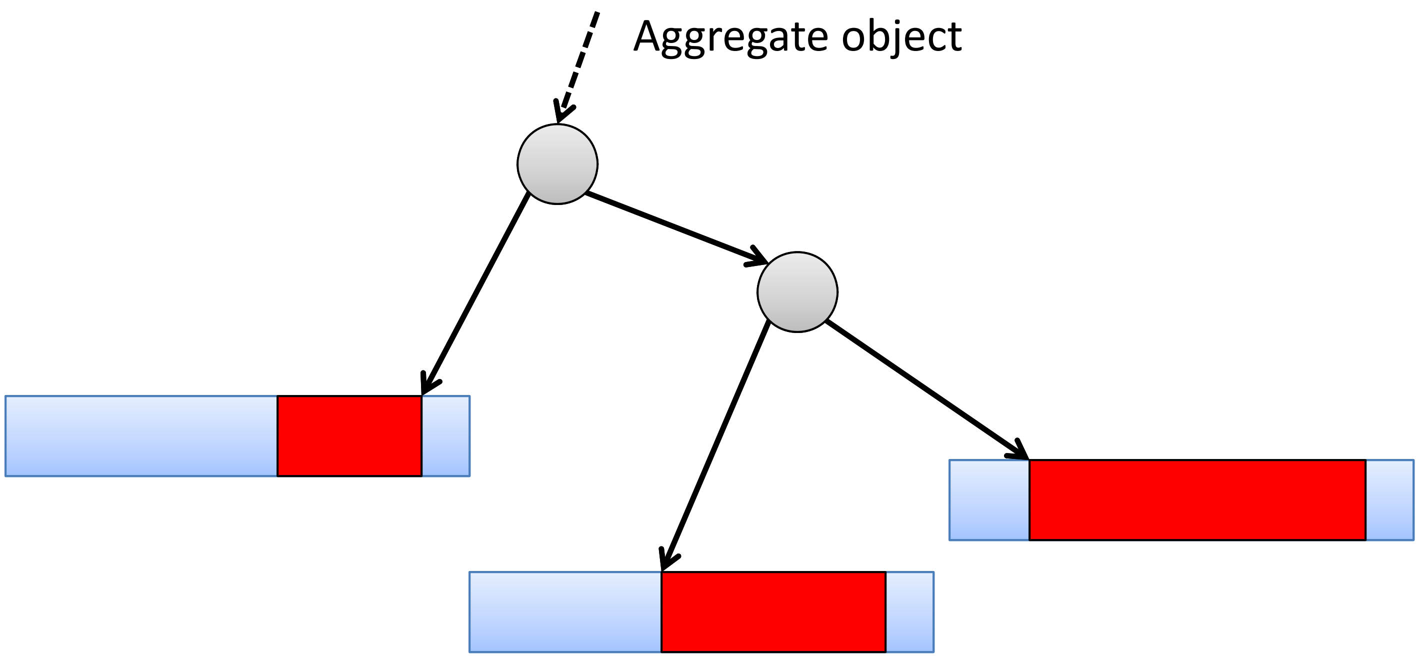
- Used in, e.g. LRPC, Firefly RPC [Schroeder and Burrows, 1990]
- Statically share memory and don't map
- Memory is now dedicated
- Useful for, e.g. kernel-userspace buffers
- Tradeoff: security vs. efficiency
- E.g. Firefly had global shared RPC pool

Fbufs basic design

- One or more contiguous virtual memory pages
- Application either allocates one or receives one
- Used to build higher-level abstractions
- Initially, kernel only knows about Fbufs...
- But see later...

Aggregate objects

- Aggregate abstractions layered on top:



Gives you the functionality of Unix **`mbufs`** [Leffler et al., 1989].

1. Allocate aggregate object

1. Allocate virtual address range in the sender
2. Allocate physical pages and zero contents
3. Update physical page tables

2. Send or forward aggregate object

1. Generate list of fbufs concerned
2. Raise protection in originator (read-only or no access)
3. Update physical page tables, flush TLB/cache if necessary

3. Receive aggregate object

1. Allocate virtual address range in the receiver
2. Update physical page tables
3. Construct an aggregate object

Note: aggregate objects are always local to an address space

4. Free aggregate object

1. Deallocate virtual address range
2. Update physical page tables, flush TLB/cache if necessary
3. Free physical memory pages if no more references

Overhead of the basic scheme

- This is all very slow!
- Lots of page table updates
- Lots of clearing new physical pages
- But illustrates the basic technique ...

Druschel and Peterson introduced (or borrowed) a bunch of optimizations...

Optimization 1: Restricted dynamic read sharing

- Always map fbufs into the same virtual region in any address space
- Make write accesses illegal after the first send
- Eliminates cost 3.1
- Reminiscent of SASOSes...
- Don’t have to allocate a virtual address range on send
- Might not have to flush a virtually tagged cache
- Don’t need COW

Sources of overhead

1. Allocate an aggregate object
   1.1 Allocate virtual address range in the sender
   1.2 Allocate physical pages and zero contents
   1.3 Update physical page tables
2. Send or forward an aggregate object
   2.1 Generate list of fbufs concerned
   2.2 Raise protection in originator (read-only or no access)
   2.3 Update physical page tables, flush TLB/cache if necessary
3. Receive aggregate object
   ~~3.1 Allocate virtual address range in the receiver~~
   3.2 Update physical page tables
   3.3 Construct an aggregate object
4. Free an aggregate object
   4.1 Deallocate virtual address range
   4.2 Update physical page tables, flush TLB/cache if necessary
   4.3 Free physical memory pages if no more references

Sources of overhead

1. Allocate an aggregate object
   - 1.1 Allocate virtual address range in the sender
   - 1.2 Allocate physical pages and zero contents
   - 1.3 Update physical page tables
2. Send or forward an aggregate object
   - 2.1 Generate list of fbufs concerned
   - 2.2 Raise protection in originator (read-only or no access)
   - 2.3 Update physical page tables, flush TLB/cache if necessary
3. Receive aggregate object
   - ~~3.1 Allocate virtual address range in the receiver~~
   - 3.2 Update physical page tables
   - 3.3 Construct an aggregate object
4. Free an aggregate object
   - 4.1 Deallocate virtual address range
   - 4.2 Update physical page tables, flush TLB/cache if necessary
   - 4.3 Free physical memory pages if no more references

Optimization 2: cache Fbufs

- When Fbuf is deallocated, return to originator
- Per-originator free list of reusable Fbufs
- Elminates 1.1, 1.2, 1.3, 2.1, 2.2, 4.1, 4.2, 4.3 in the common case
- How common is this? Chances are communication path is to be reused
- Need to know where the Fbuf is going before allocation...

Sources of overhead

1. Allocate an aggregate object
   - ~~1.1 Allocate virtual address range in the sender~~
   - ~~1.2 Allocate physical pages and zero contents~~
   - ~~1.3 Update physical page tables~~
2. Send or forward an aggregate object
   - ~~2.1 Generate list of fbufs concerned~~
   - ~~2.2 Raise protection in originator (read-only or no access)~~
   - 2.3 Update physical page tables, flush TLB/cache if necessary
3. Receive aggregate object
   - ~~3.1 Allocate virtual address range in the receiver~~
   - 3.2 Update physical page tables
   - 3.3 Construct an aggregate object
4. Free an aggregate object
   - ~~4.1 Deallocate virtual address range~~
   - ~~4.2 Update physical page tables, flush TLB/cache if necessary~~
   - ~~4.3 Free physical memory pages if no more references~~

Optimization 3: Integrated Buffer Management/Transfer

- Embed aggregate information inside the Fbufs
- Sender passes kernel reference to “root” Fbuf
- Kernel walks data structure,
  - transfers all relevant Fbufs
  - passes root reference to receiver
- Eliminates 2.1, 3.3.
- Works well with single-virtual-address technique

Sources of overhead

Systems@ETH Zürich

1. Allocate an aggregate object
   - ~~1.1 Allocate virtual address range in the sender~~
   - ~~1.2 Allocate physical pages and zero contents~~
   - ~~1.3 Update physical page tables~~
2. Send or forward an aggregate object
   - ~~2.1 Generate list of fbufs concerned~~
   - ~~2.2 Raise protection in originator (read-only or no access)~~
   - 2.3 Update physical page tables, flush TLB/cache if necessary
3. Receive aggregate object
   - ~~3.1 Allocate virtual address range in the receiver~~
   - 3.2 Update physical page tables
   - ~~3.3 Construct an aggregate object~~
4. Free an aggregate object
   - ~~4.1 Deallocate virtual address range~~
   - ~~4.2 Update physical page tables, flush TLB/cache if necessary~~
   - ~~4.3 Free physical memory pages if no more references~~

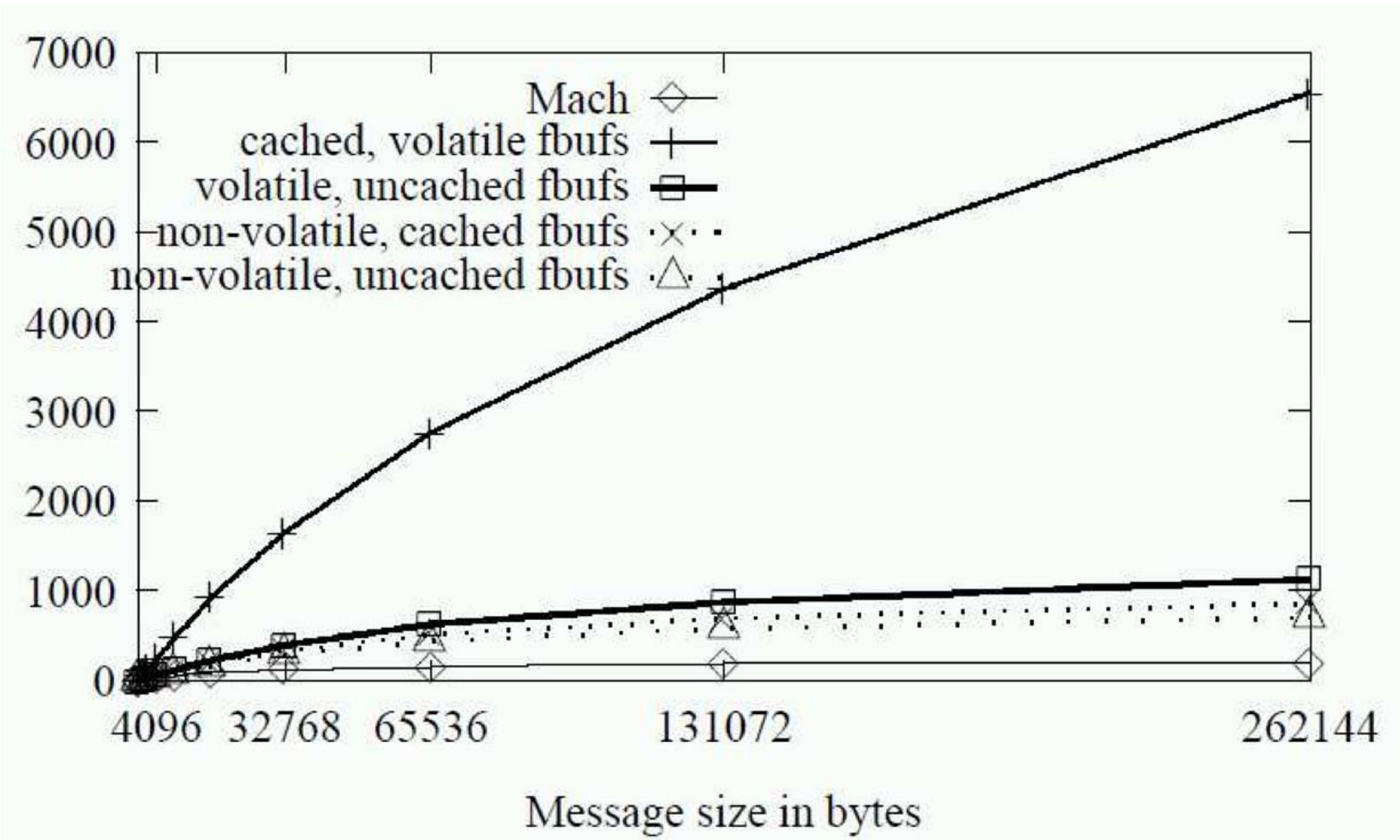
Optimization 4: Volatile fbufs

- Allow sender to modify fbufs after sending, unless receiver forbids it
- Eliminates cost of removing page permissions from sender
- Receiver must now be careful:
  - All DAG pointers must point into fbuf region
  - DAG really must be acyclic
  - Kernel turns receiver read violations into "no data" (zeroed page)

Sources of overhead

1. Allocate an aggregate object
   - ~~1.1 Allocate virtual address range in the sender~~
   - ~~1.2 Allocate physical pages and zero contents~~
   - ~~1.3 Update physical page tables~~
2. Send or forward an aggregate object
   - ~~2.1 Generate list of fbufs concerned~~
   - ~~2.2 Raise protection in originator (read-only or no access)~~
   - ~~2.3 Update physical page tables, flush TLB/cache if necessary~~
3. Receive aggregate object
   - ~~3.1 Allocate virtual address range in the receiver~~
   - ~~3.2 Update physical page tables~~
   - ~~3.3 Construct an aggregate object~~
4. Free an aggregate object
   - ~~4.1 Deallocate virtual address range~~
   - ~~4.2 Update physical page tables, flush TLB/cache if necessary~~
   - ~~4.3 Free physical memory pages if no more references~~

Performance



Single domain crossing

Discussion

- What happens to the rest of the fbuf?
  - Aggregates can't share fbufs.
  - High memory overhead (at least 1 page per message)
- Fbufs are pageable!
  - Claim: this is bad for I/O devices & DMA
  - But is it?
- Fbufs are still allocated when they are needed.
  - Latency becomes a problem with bursts?
- Lots of kernel voodoo:
  - Lots of data manipulation required in the kernel
  - Claim in paper that fbufs are appropriate for URPC, but are they?

RBufs

[Leslie et al., 1996]

- Richard Black’s PhD, adopted for Nemesis
- Not the only one, but representative of the time.

- Key idea: split up:
  – Data area (shared memory)
  – Offset/length aggregation
  – Memory allocation and freeing

RBuf concepts

- Features:
  - Use of Event Channels
    - Combined notification w/ sending a 64-bit integer
  - Explicit connection setup
    $\Rightarrow$ pre-allocation of all resources (including memory).
- Terminology:

  Originator: producer of the data to be sent

  Receiver: consumer of the data

  Master: "owner" of the channel
    - can be either originator or receiver
    - decides where the data goes

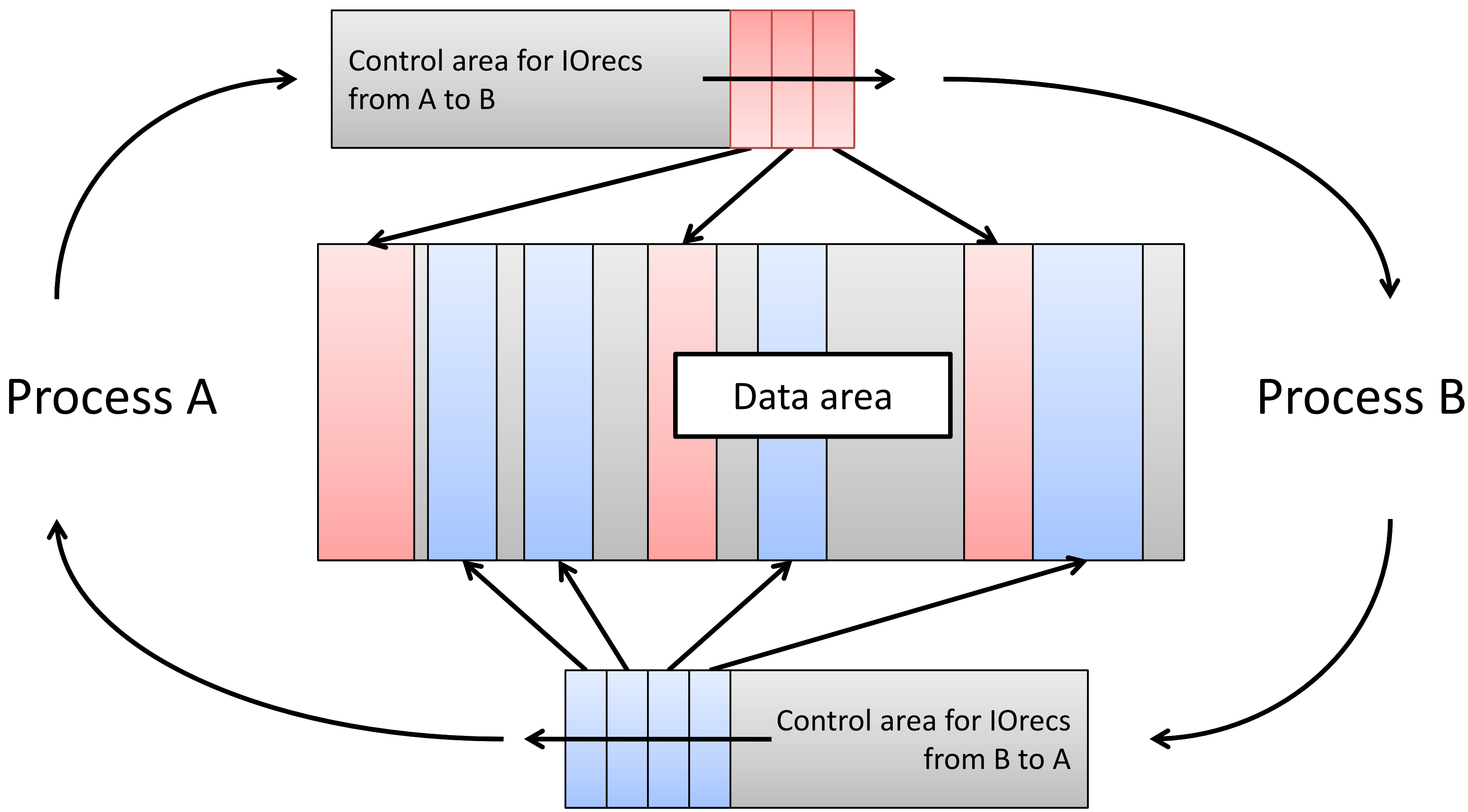
RBuf IDC channel

Systems@ETH Zürich

Basically a bidirectional channel for sending memory regions between two protection domains, one of which can write the regions.

- *data area* (shared memory region)
- 2 *control areas* (in different directions), each:
  - a shared memory region
  - 2 *event channels* (in different directions)

Complete Rbuf channel

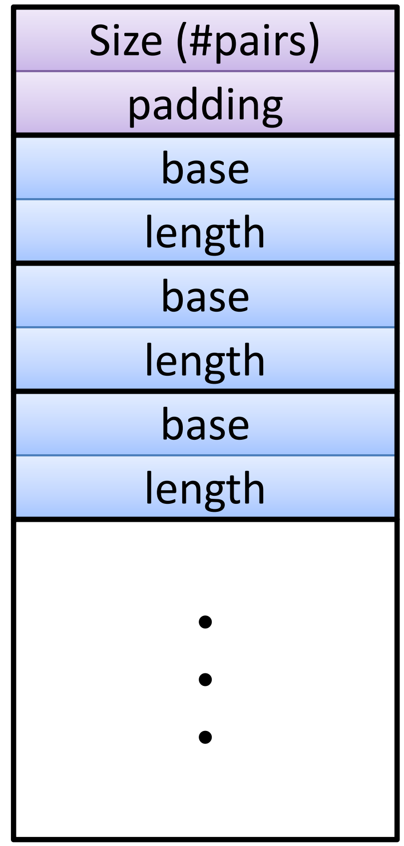


RBufs data area

- One per channel
- Small number of large areas of virtual address space
- Allocated by the system, always backed (i.e. can't page)
  - Motivation: drivers can DMA directly into here
  - System maps virtual->physical for drivers
- At least writable by originator
  - Always volatile.
- At least read-only by receiving domain
- Managed by the Master.

RBufs aggregate objects

- Descriptors for data areas (full or empty):
  - “IOrecs” (like a Unix **`iovec`**)
  - Header followed by base/length pairs
  - Header padded to length of pair
  - Variable size, but aligned

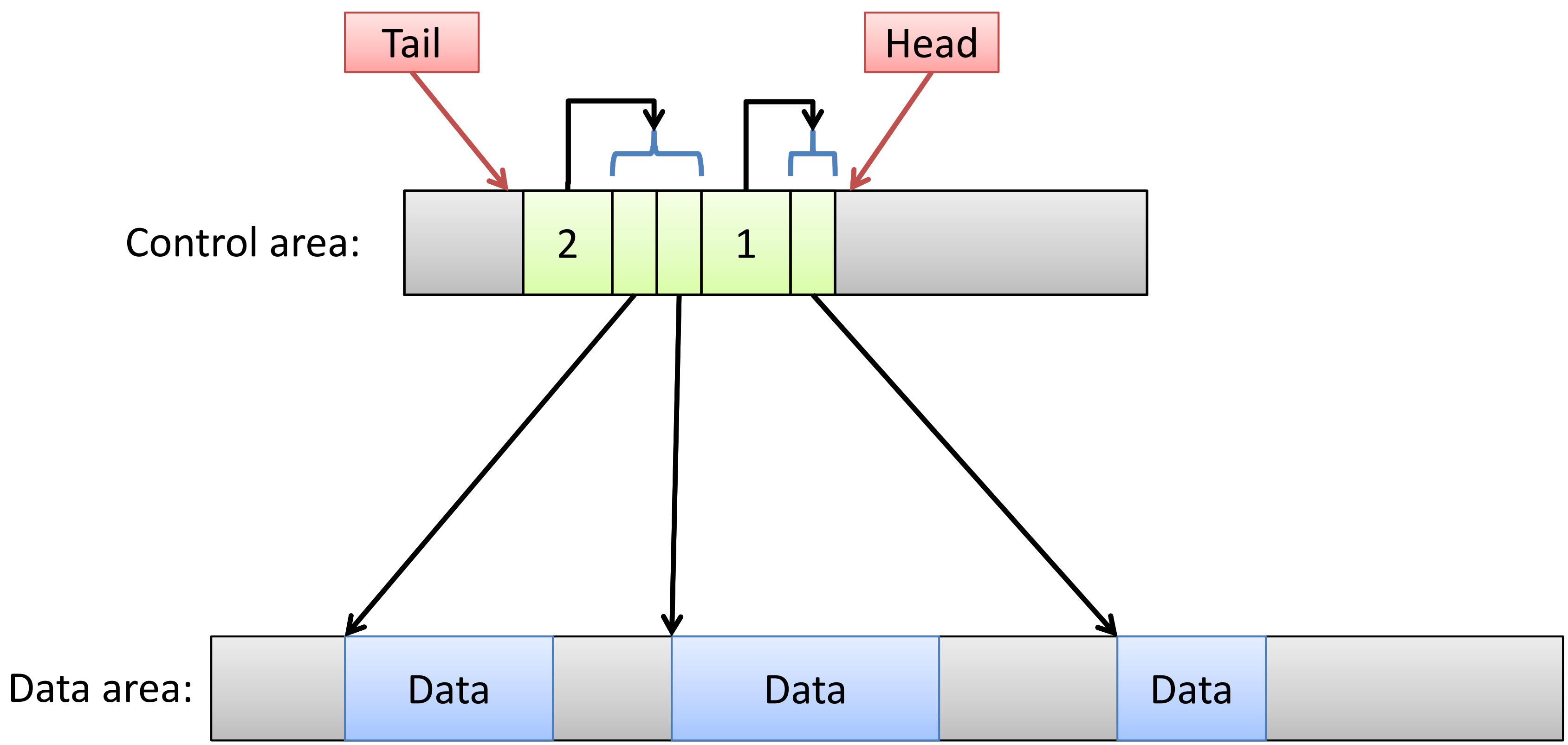


Control area

Circular buffer: producer-consumer queue of IOrecs

- 2 Event channels, one each way.
  - Sender → Receiver: head position
  - Receiver → Sender: tail position
- Shared memory area holding Iorecs
  - writeable by the one end
  - readable by the other side.
  - Note: different from permissions on the data area.

Control area



Operating modes

Transmit Master mode:

- Master = originator
- Originator chooses data addresses
- Writes data into data area
- Writes IOrecs into Originator control area
- Sends an event updating the head pointer (“sending” an IOrec)
- Receiver reads IOrecs from control area
- Sends an event updating the tail pointer (acking)
- Later frees data area by sending IOrecs via Receiver control area

Operating modes

Receive Master mode:

- Master = receiver
- Receiver chooses data addresses
- Writes IOrecs into Receiver control area
- Sends an event updating the head pointer ("sending" an IOrec)
- Originator reads IOrecs from control area
- Sends an event updating the tail pointer
- Writes data into data area
- "Sends" data by sending IOrecs via Originator control area

Operating modes: summary

Systems@ETH Zürich

	Transmit master	Receive master
Chooses data area	Originator	Receiver
Write access to data	Originator	Originator
Read access to data	Receiver	Receiver

Why these two modes?

- RMM suitable for network receive path (asynchronous)
- TMM for transmit path (synchronous)

Multidomain

- What about IDC paths crossing multiple domains?
  - Story is weak: share data areas between multiple pair-wise channels

Multicast

- What about multiple receivers of the same data?
  - One data area, multiple control areas (one for each receiver)
  - Originator puts IOrecs in every receivers control area
  - Waits for all to be acknowledged
  - Reference counts each part of the data area
  - Bounded control queues limit ability for a receiver to hog data
- But still ...

Memory overhead

- Problem:
  - Data area requires at least 1 page per pair of domains
  - Control areas require 2 pages per pair of domains
- (Partial) solution: “the GateKeeper”
  - Aggregates domain pairs to amortize memory costs

Xen

[Fraser et al., 2004]

Systems@ETH Zürich

- **Device channels** provide fast bulk IPC in Xen
  - Aside: by now you should realize that I/O and IPC are closely related
- Use
  - shared memory
  - buffer descriptor rings
- But more serious about protection than RBufs...

Xen use of shared memory

- Shared memory mappings:
    1. Asymmetric
    2. Transitory
- Only one domain owns each page of memory at a time
- Owner may reclaim page mapped elsewhere at any time
- Domain maps foreign pages by presenting a valid "grant request" to Xen
    – index into ...

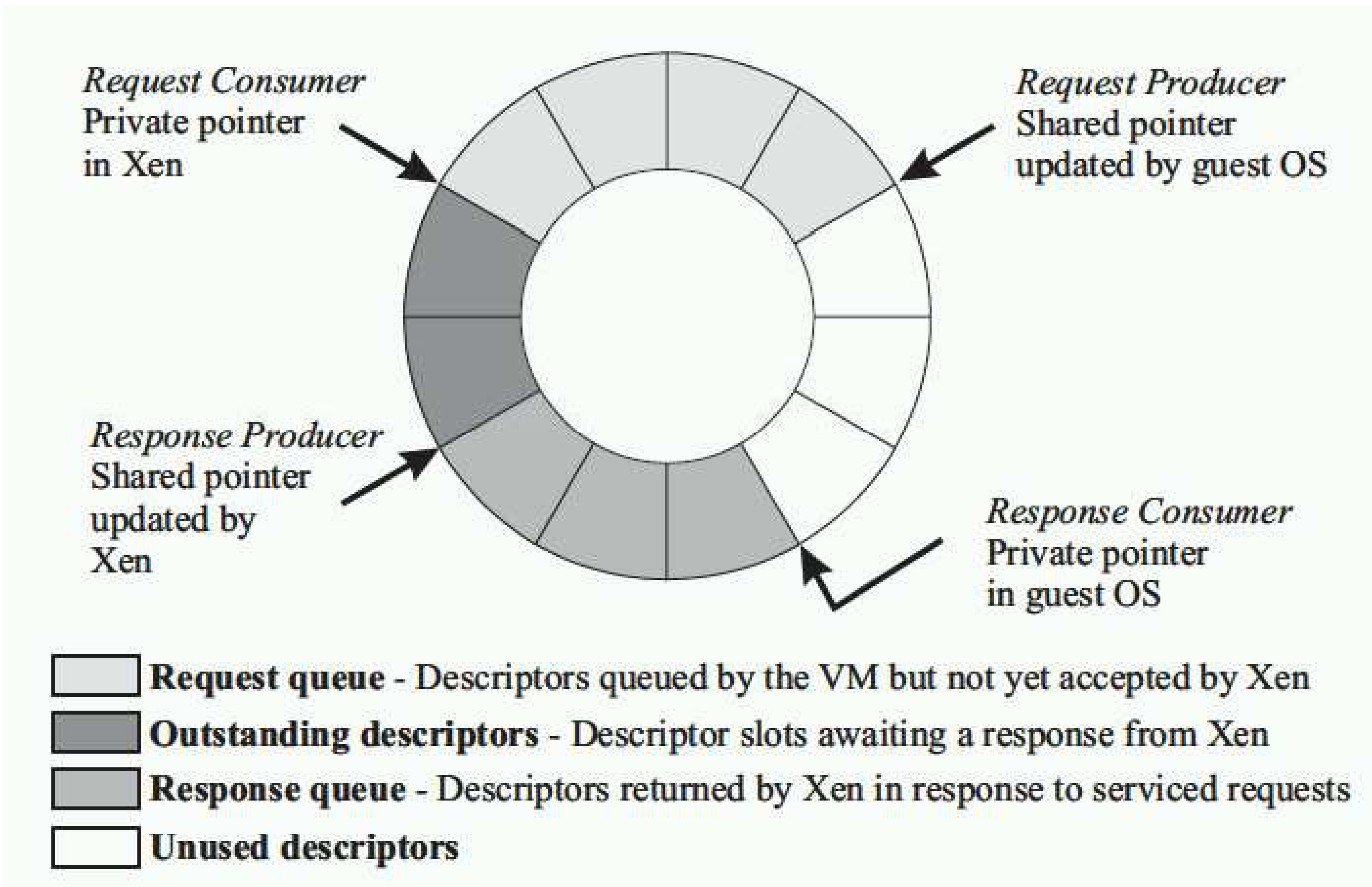
Grant tables in Xen

Per-domain table of tuples (grant, D, P, R, U):

- **D**: domain being granted permission
- **P**: page frame number
- **R**: read-only mappings only
- **U**: *in use*: whether D currently maps P.

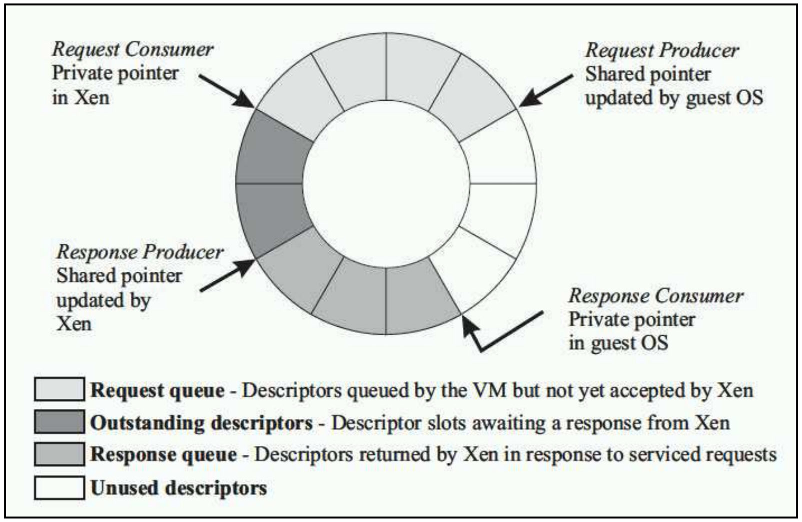
- Also keeps reference counts of active mappings in the per-domain **Active Grant Table**

Descriptor rings



Descriptor rings

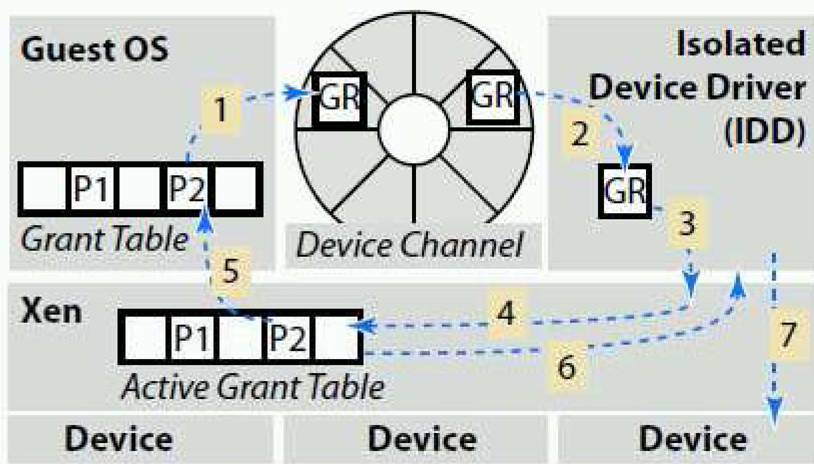
- Notice unlike Rbufs:
  - only one buffer
  - two head/tail pointers (one for each end)
  - both sides can write the buffer ring
- Is there a trust issue here?



Complete path

Guest Requests DMA:

1. Grant Reference for Page P2 placed on device channel
2. IDD removes GR
3. Sends pin request to Xen



4. Xen looks up GR in active grant table
5. GR validated against Guest (if necessary)
6. Pinning is acknowledged to IDD
7. IDD sends DMA request to device

References

- Cheriton, D. R. (1988), **The V distributed system**. Communications of the ACM, 31(3):314–333.
- Druschel, P. and Peterson, L. L. (1993), **Fbufs: a high-bandwidth cross-domain transfer facility**. In SOSP ’93: Proceedings of the fourteenth ACM symposium on Operating systems principles, pages 189–202, New York, NY, USA. ACM.
- Fraser, K., Hand, S., Neugebauer, R., Pratt, I.,Warfield, A., and Williams, M. (2004). **Safe hardware access with the Xen virtual machine monitor.** In Proceedings of the 1stWorkshop on Operating System and Architectural Support for the on demand ITInfraStructure (OASIS), Boston,MA, USA.
- Barrera, J. S. III (1991). **A fast Mach network IPC implementation**. In Proceedings of the Usenix Mach Symposium.

References

Systems@ETH Zürich

- Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. (1989). **The Design and Implementation of the 4.3BSD Unix Operating System**. Addison-Wesley Publishing Company.
- Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P., Evers, D., Fairbairns, R., and Hyden, E. (1996). **The Design and Implementation of an Operating System to Support Distributed Multimedia Applications**. IEEE Journal on Selected Areas in Communications, 14(7):1280–129.
- Ousterhout, J. (1989). **Why aren't operating systems getting faster as fast as hardware?** WRL Technical Note TN-11, Digital Western Research Laboratory.
- Schroeder, M. D. and Burrows, M. (1990). **Performance of the Firefly RPC**. ACM Trans. Comput. Syst., 8(1):1–17.