THE DESIGN AND IMPLEMENTATION OF THE UVM VIRTUAL MEMORY SYSTEM

Charles D. Cranor
chuck@ccrc.wustl.edu

Department of Computer Science
Washington University in St. Louis

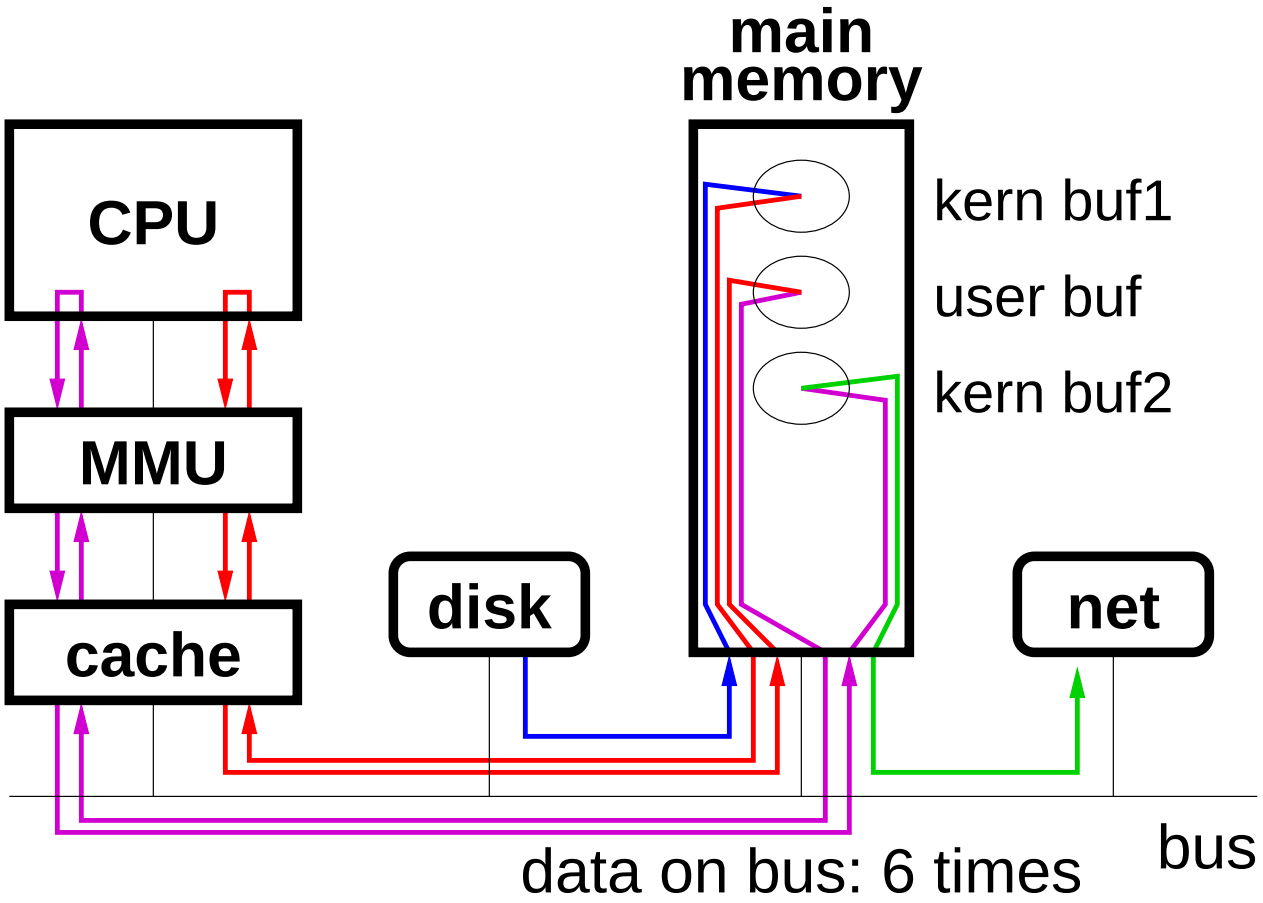
MOTIVATION

- **New applications expect the highest performance from an operating system's I/O and IPC systems**

- **Problem: traditional system requires unnecessary data copies**

- **Why are data copies costly?**
  - limited memory bandwidth
  - CPU busy moving data one word at a time
  - side effect: cache flush

Transmitting a file over the network

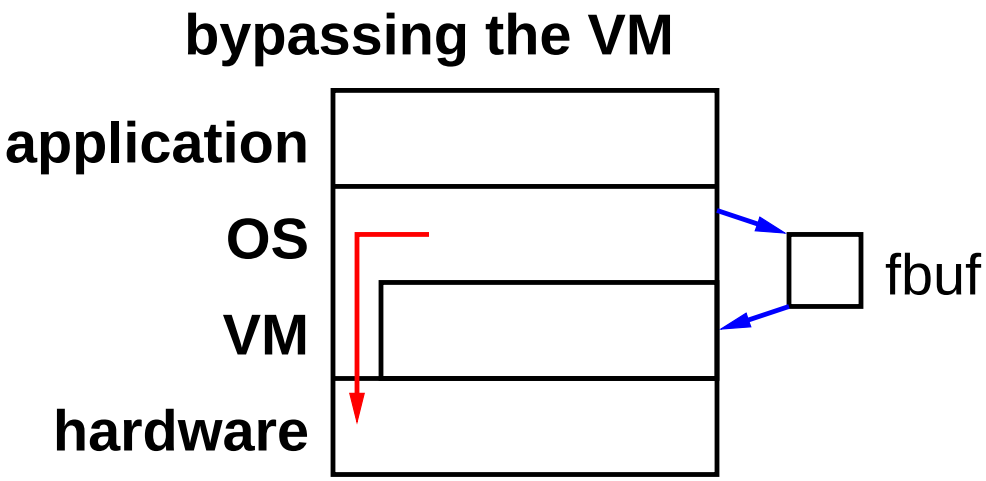


Steps:

1. **DMA data from disk to kernel buffer #1**
2. **copy data from kernel buffer #1 to user buffer**
3. **copy data from user buffer to kernel buffer #2**
4. **DMA buffer from kernel buffer #2 to network**

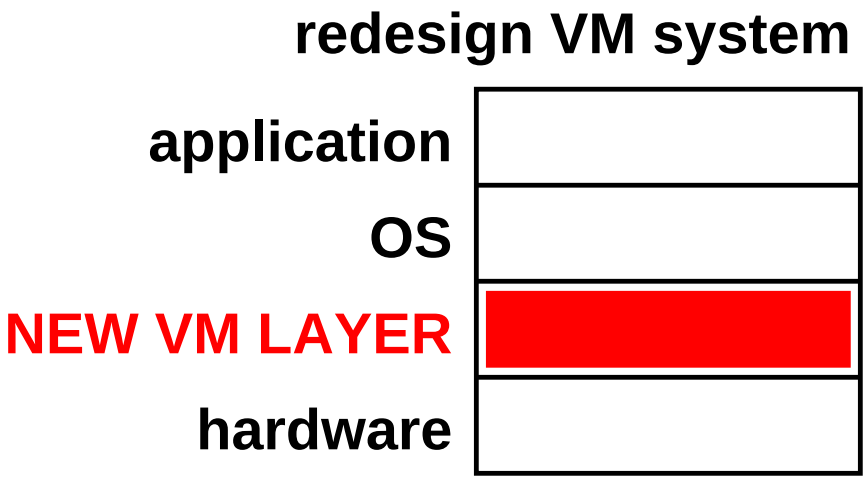
Using virtual memory hardware to reduce data copying

- **Past work**
  - focused on network and IPC systems
  - built on top of existing VM system
  - high-level VM system bypassed
  - loss of flexibility



- **Our work**
  - focused on internal design of VM system

Our work



This is a challenging approach

- VM data shared between all processes
- asynchronous operations
- harder to debug
- not as well documented as networking/IPC systems

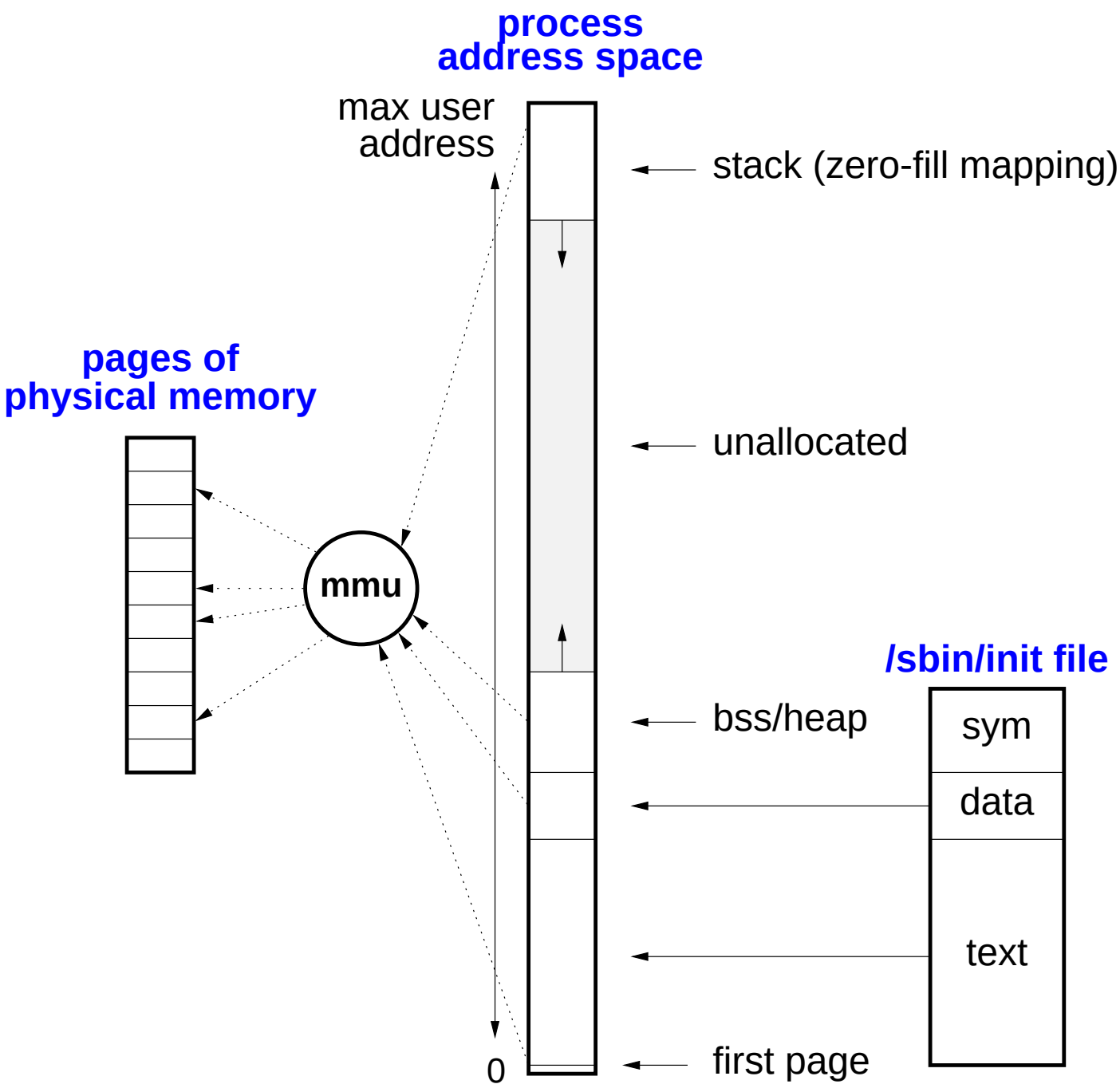
The result: UVM

- **Provides three new mechanisms for efficient data movement**
- **Implemented in and is part of NetBSD**
- **Runs on multiple platforms: alpha, i386, m68k, mips, ns32k, powerpc, sparc, and vax**
- **Source code freely available**

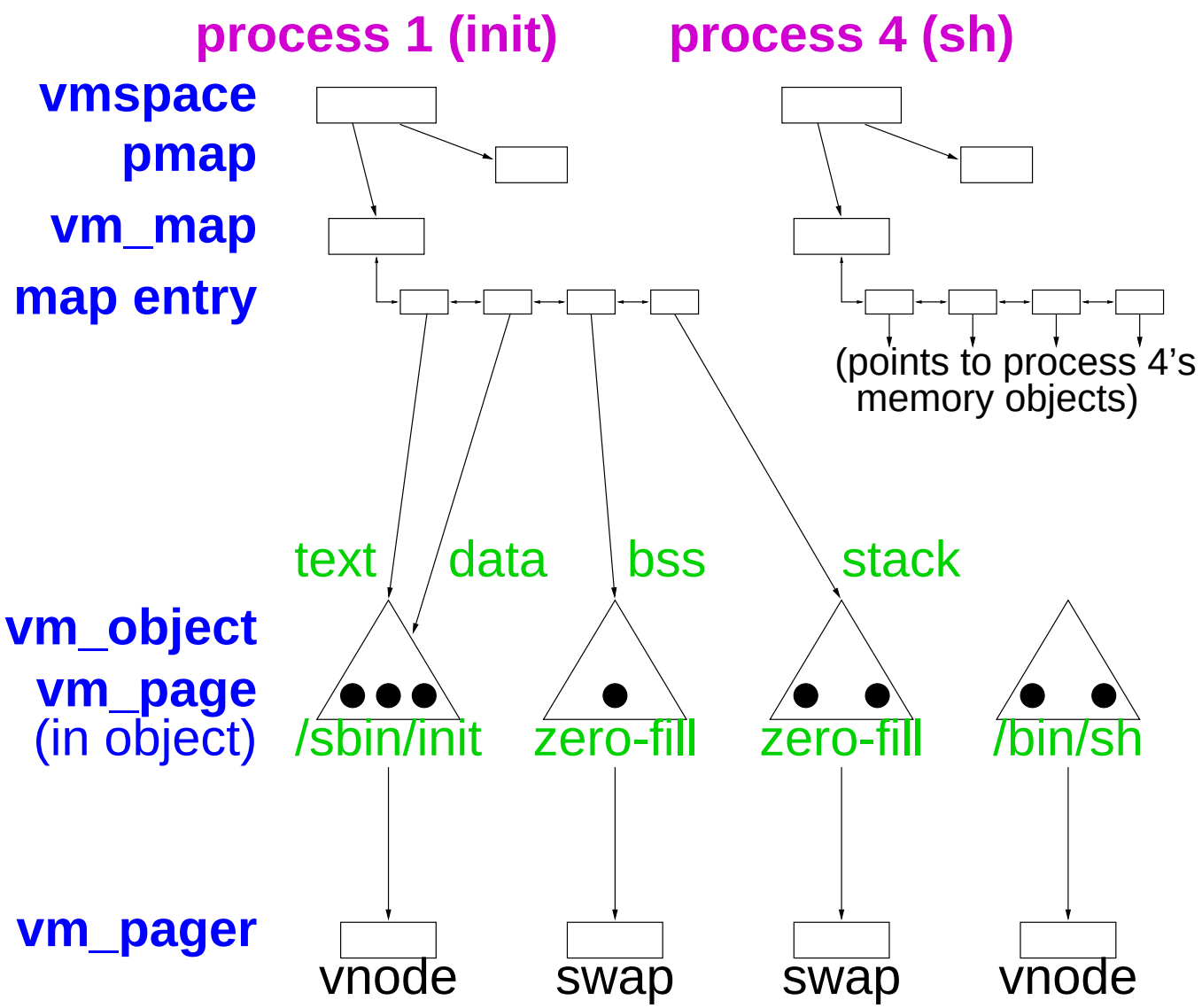
OUTLINE

- **Background: BSD VM**
- **Overview of UVM**
- **Related Work**
- **UVM Data Movement**
- **Data Movement Measurements**
- **Conclusions and Future Research**

BACKGROUND


process
address space
max user
address
stack (zero-fill mapping)
pages of
physical memory
unallocated
mmu
/sbin/init file
bss/heap
sym
data
text
0
first page

Basic structure

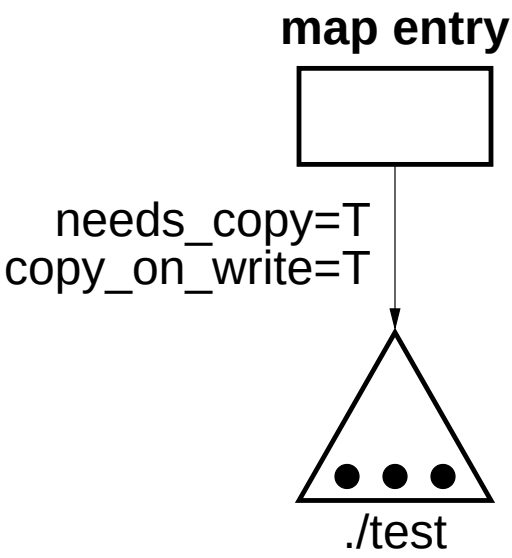


Copy-on-write using object chaining

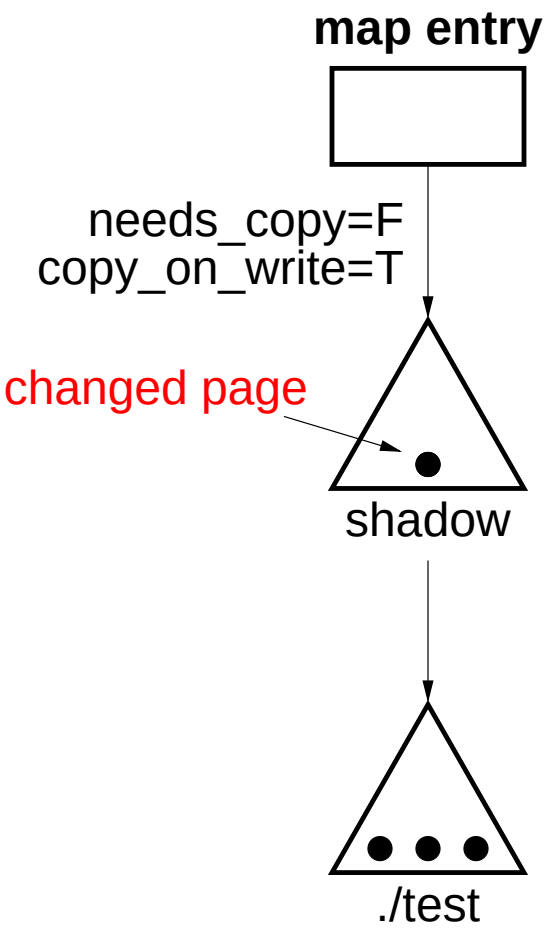
Why talk about copy-on-write?

- a mechanism to avoid data copies
- not general purpose
- inefficient in BSD VM

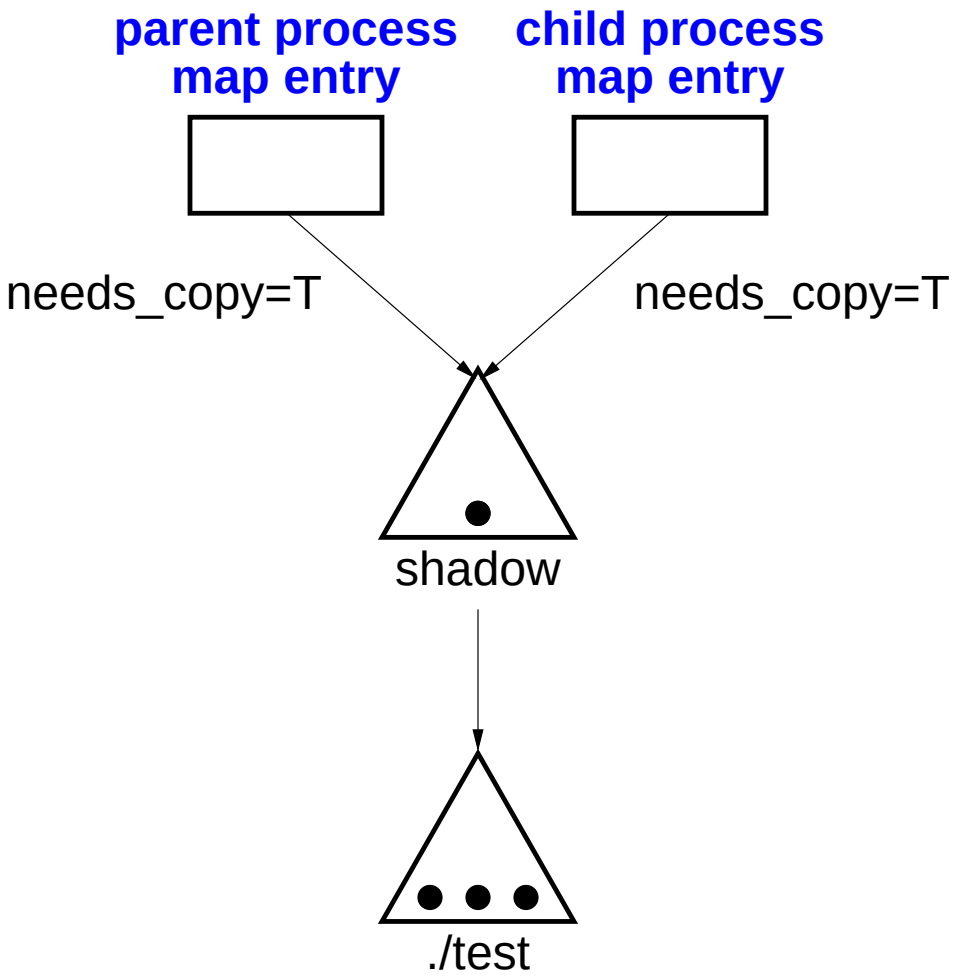
Before write fault



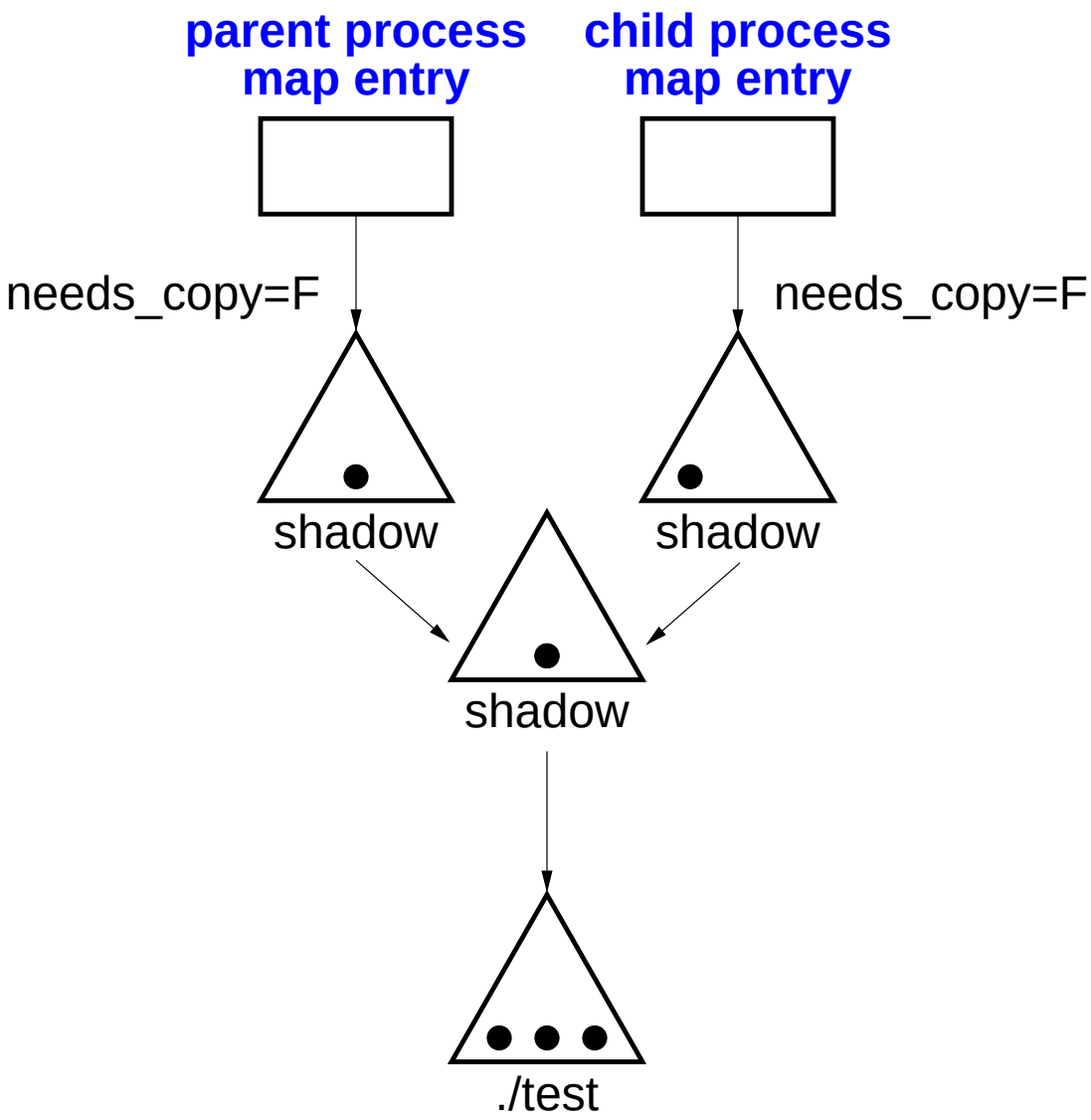
After write fault



Copy-on-write after fork



copy-on-write after write fault



- complex search/fault
- "swap memory leak"
- collapse problems

"A source of headaches"

Structure: two-layer mapping

- **top:** anonymous layer (copy-on-write, anon pages)
- **bottom:** object layer (files)

Advantages

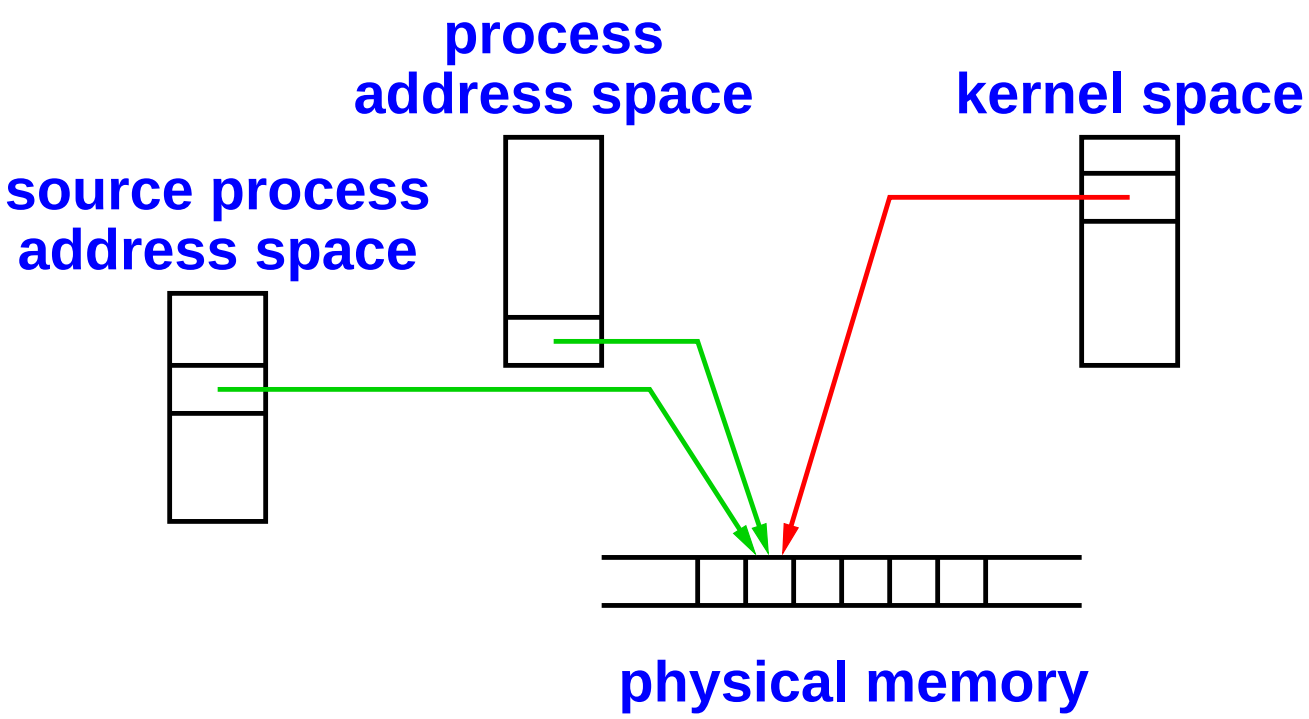
- reduces the complexity of BSD VM
- allows for a pageable page of anonymous memory to be a basic unit of page granular virtual memory operations

New data movement features

- page loanout
- page transfer
- map entry passing

Page Loanout

Allow a process to safely let a shared copy-on-write copy of its memory be used either by other processes, the I/O system, or the IPC system

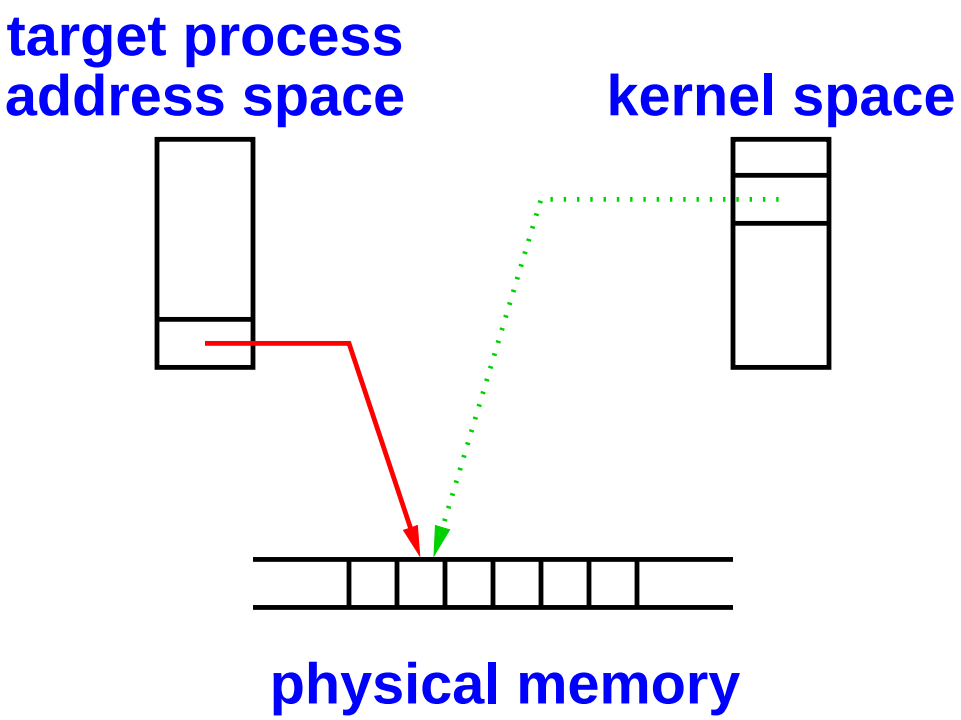


Example: process loans page to kernel

1. identify page
2. write protect all current mappings
3. return loaned page to kernel
4. kernel adds read-only mapping to loaned page

Page Transfer

Allow pages of memory from the I/O system, IPC system, or other processes to be inserted easily into a process' address space

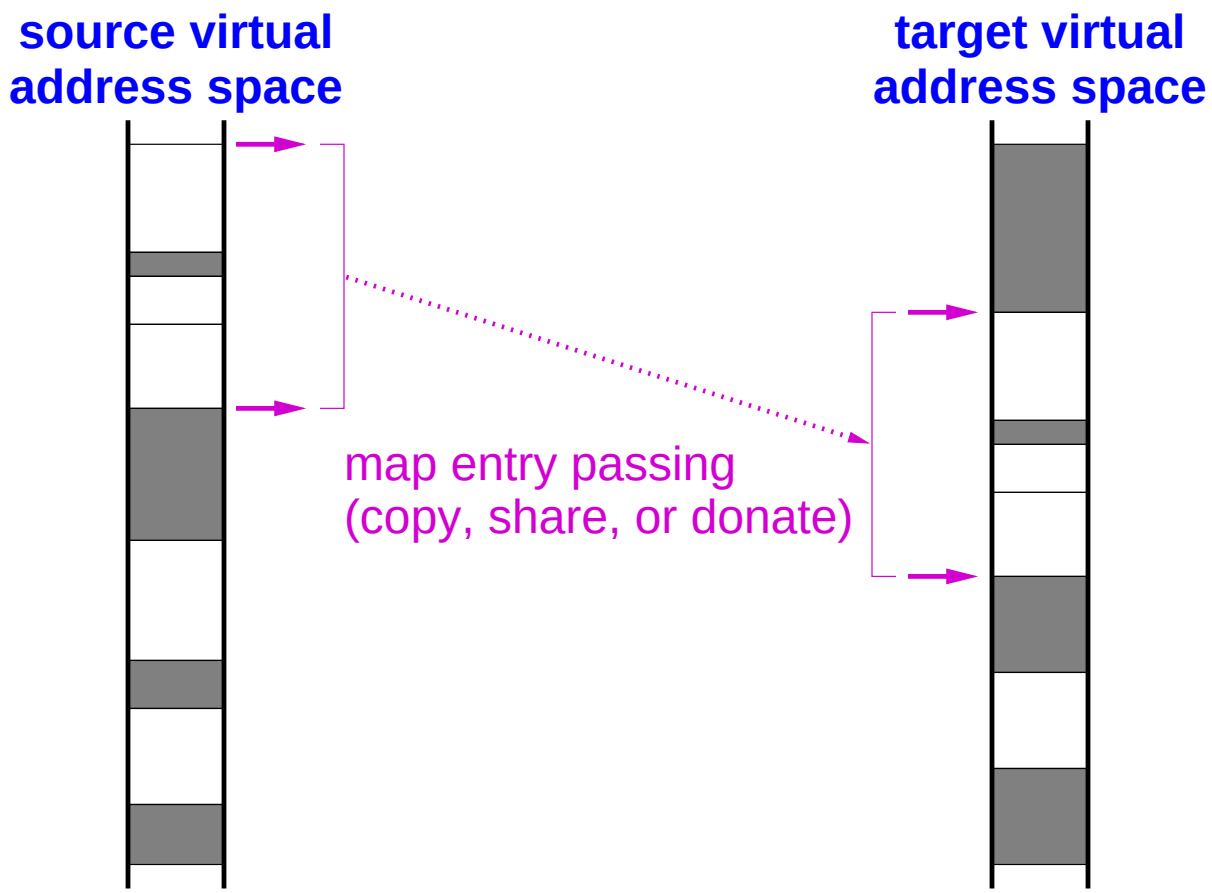


Example: kernel donates page to a process

1. identify page
2. remove page and its mapping from kernel
3. insert page in destination process

Map entry passing

Allow processes and the kernel to dynamically exchange chunks of their virtual address space



- minimal per-page costs
- mapping level granularity

Other Goals

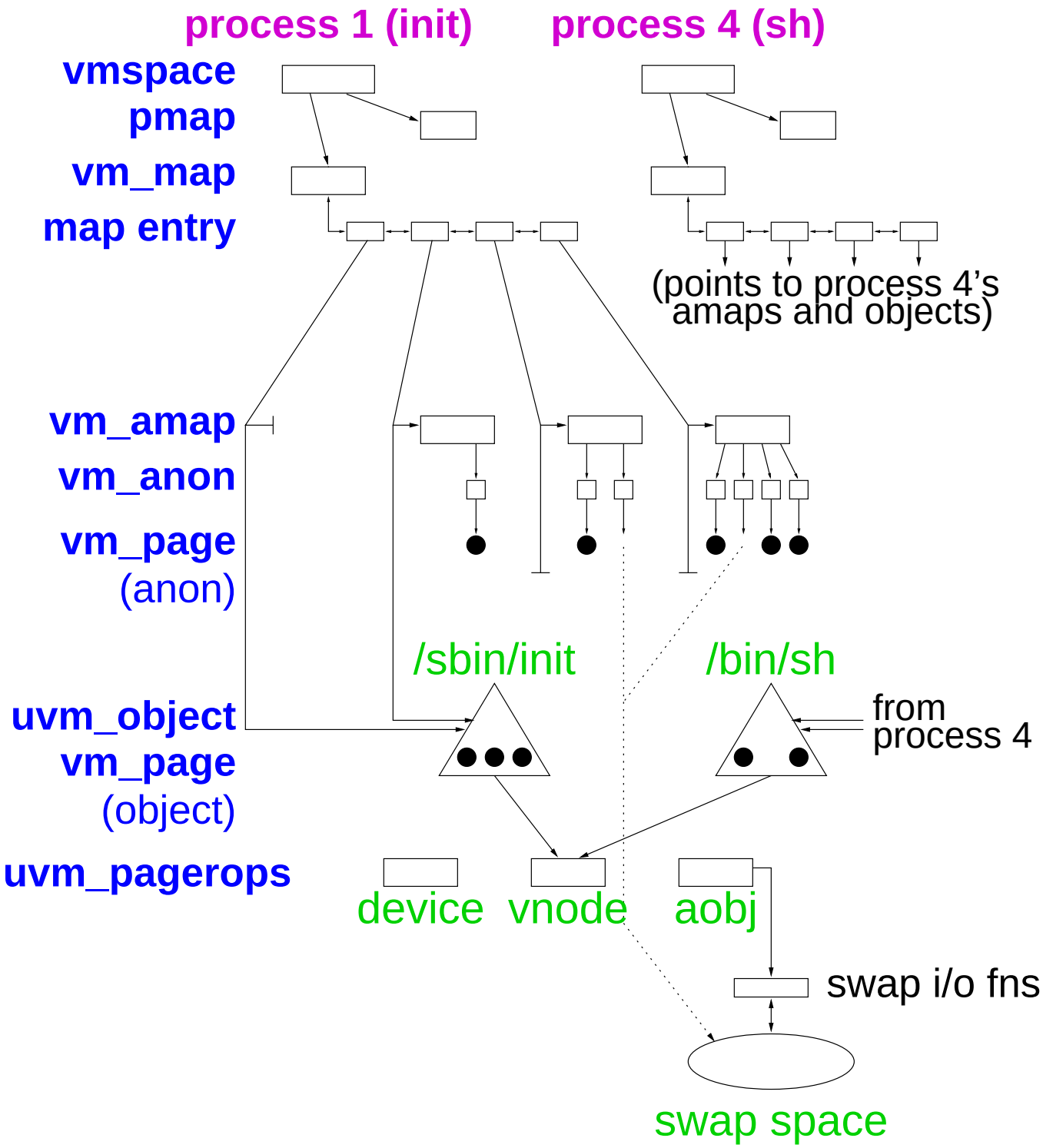
Optimize the parts of the VM system that effect performance and complexity of the new VM operations we introduce

- take advantage of lessons learned from observations of BSD VM and other VM systems

Improve secondary features

- simplified copy-on-write
- aggressive clustered anonymous pageout
- improved handling of page wiring
- new i386 machine-dependent layer (pmap)
- removal of object cache layer

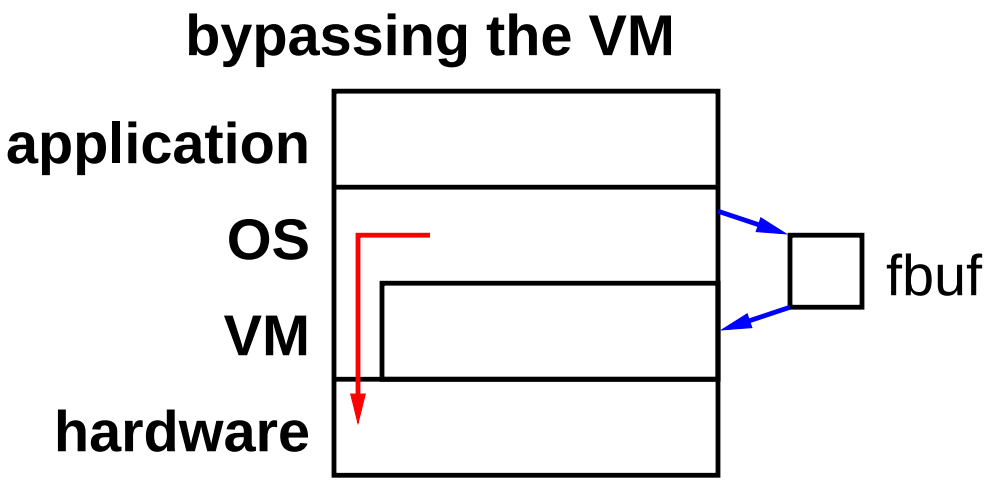
UVM DETAILS


process 1 (init)
process 4 (sh)
vmspace
pmap
vm_map
map entry
(points to process 4's amaps and objects)
vm_amap
vm_anon
vm_page
(anon)
/sbin/init
/bin/sh
uvm_object
vm_page
(object)
from process 4
uvm_pagerops
device
vnode
aobj
swap i/o fns
swap space

Design benefits

- **two-level mapping vs. multi-level chaining**
- **pageable page of anonymous memory as a building block**

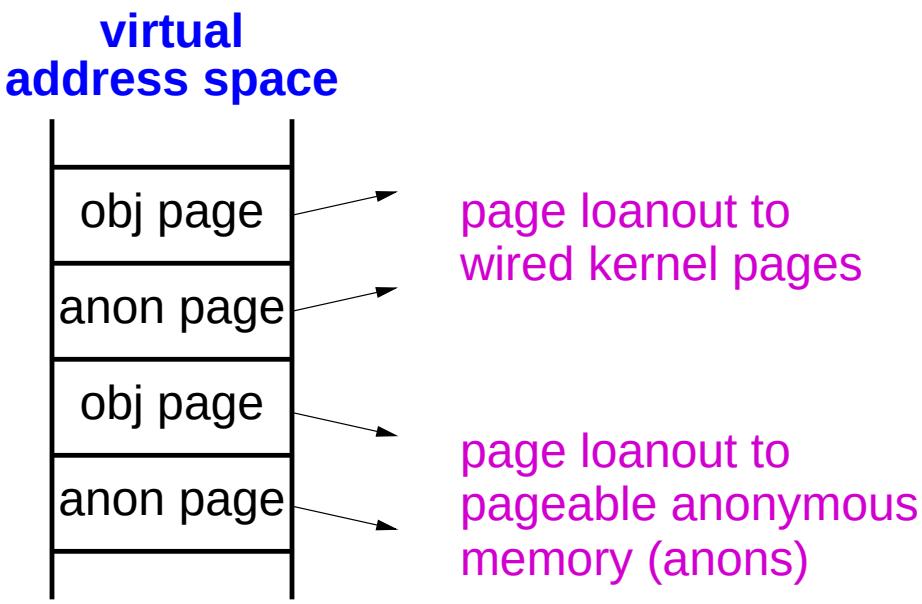
RELATED WORK



Two classes of related work:

- **Virtual memory systems**
  - Mach, FreeBSD, SunOS/Solaris, Linux, NT
- **I/O and IPC Systems**
  - Fbufs, Genie, Container shipping, Splice

Loan types



Four types of loans

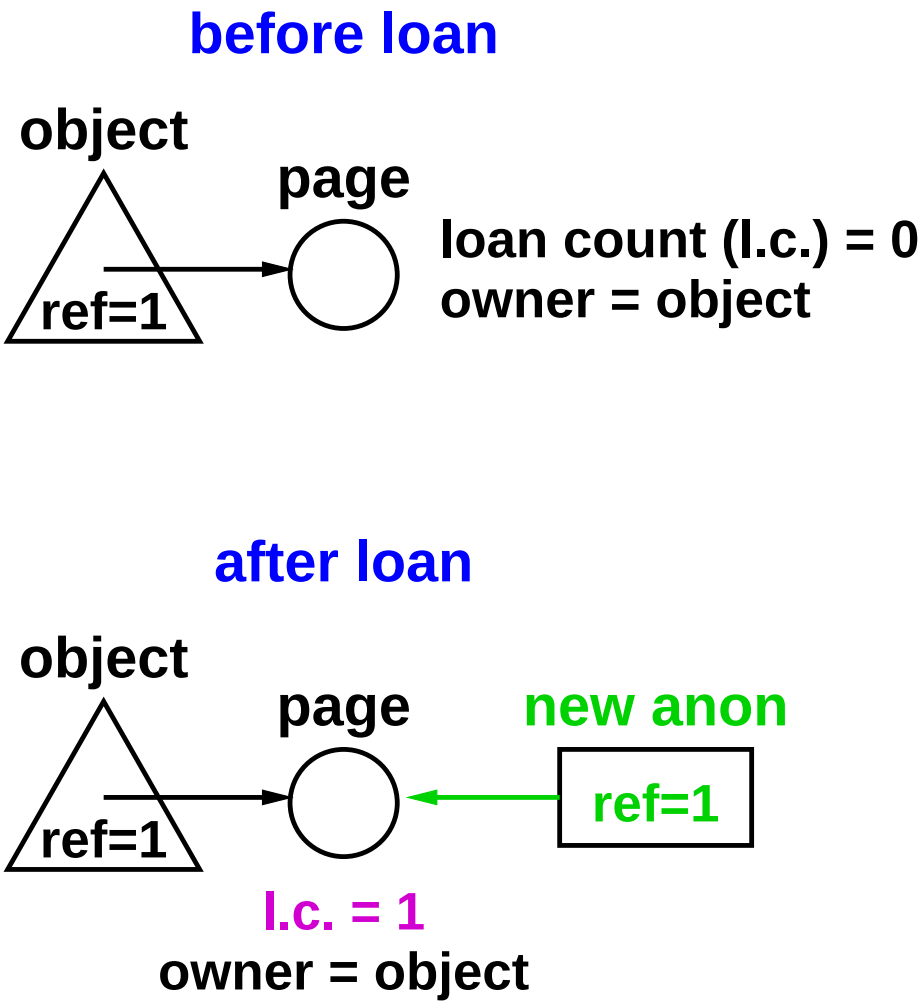
- object to wired kernel page
- object to anonymous page
- anonymous page to wired kernel page
- anonymous page to anonymous page

Loan-related data structure fields

- **per-page loan count**
  - non-zero loan count: page is immutable (read-only)
  - UVM write-protects loaned pages
  - write or free of a loaned page: breaks the loan

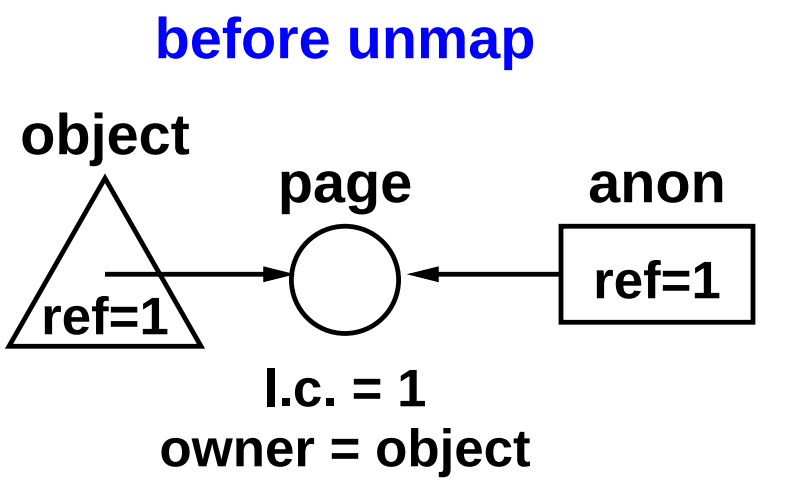
- **referencing pages**
  - object owned page can be referenced by an anon
  - owner of a loaned page can free it, thus causing it to become ownerless

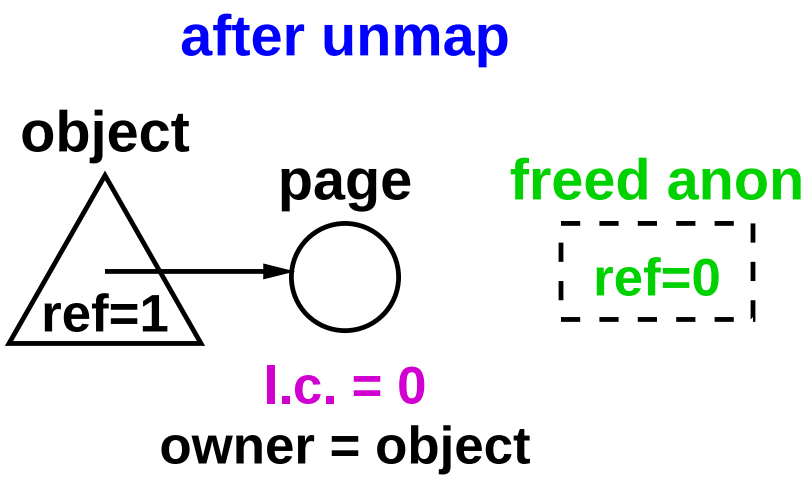
Example: loan from object to new anon



1. locate page
2. increment page's loan count
3. write protect all mappings of page
4. allocate new anon and point it at page
5. new anon can now be used by a process

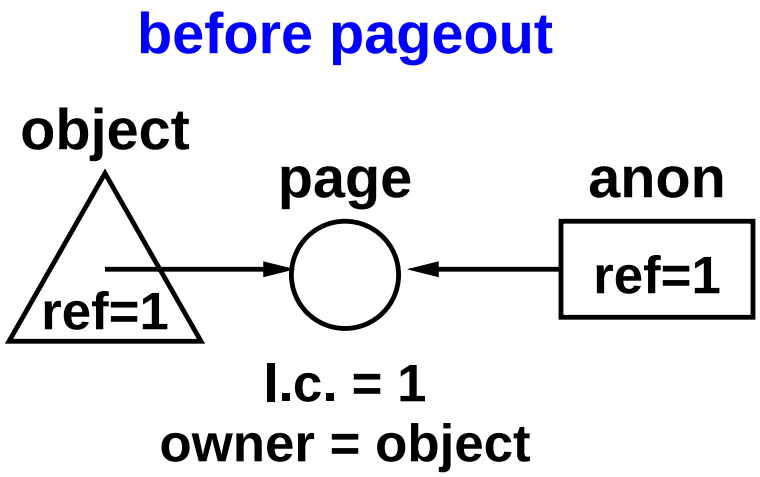
Example: process unmaps loaned page

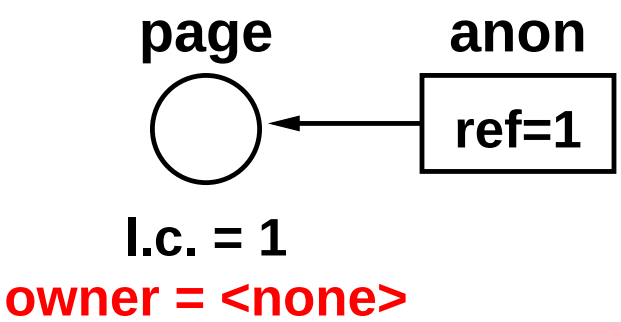




1. drop reference to anon
2. dropping final reference to anon with loaned page causes us to drop the loan count
3. free anon

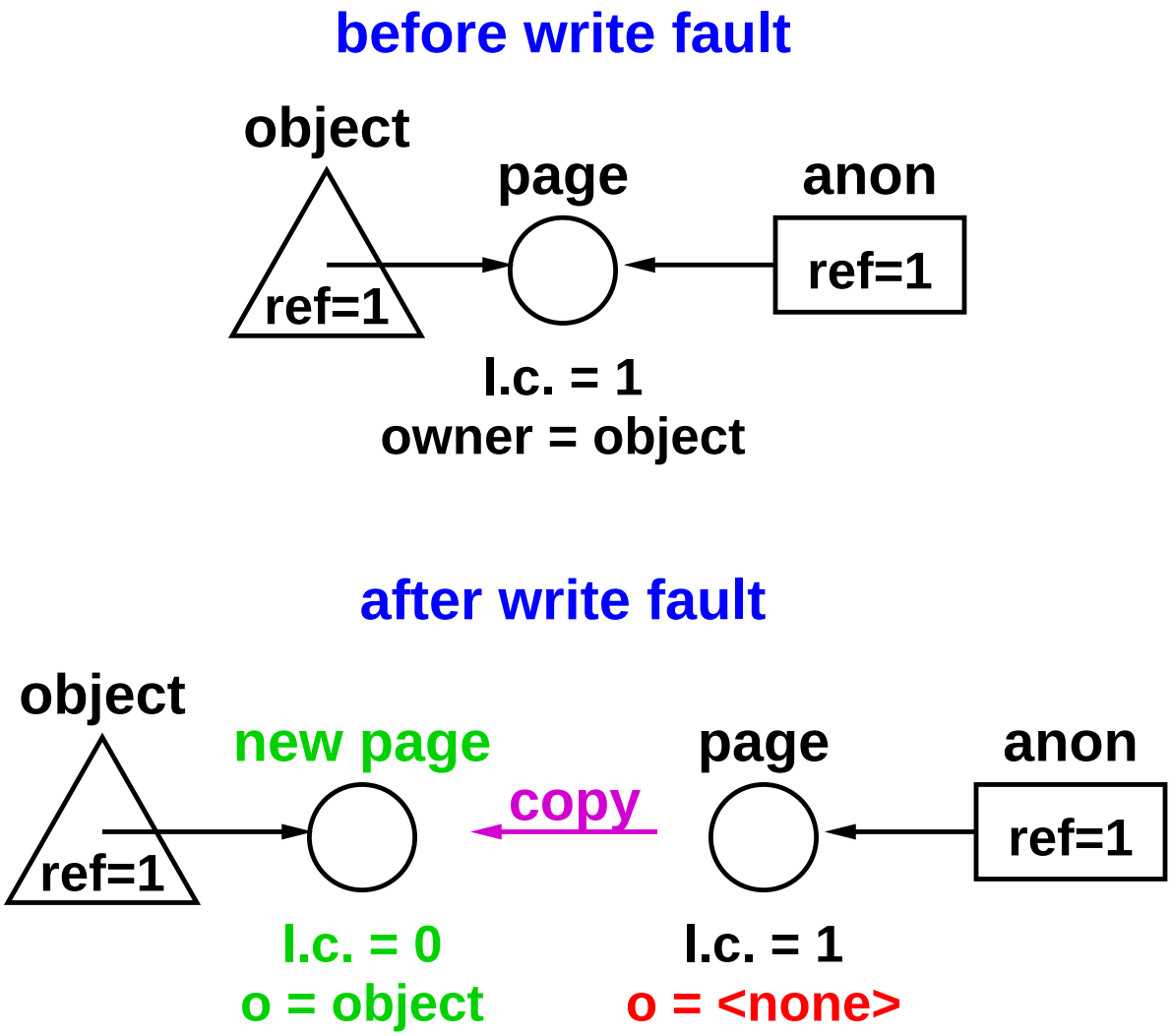
Example: pageout of loaned page





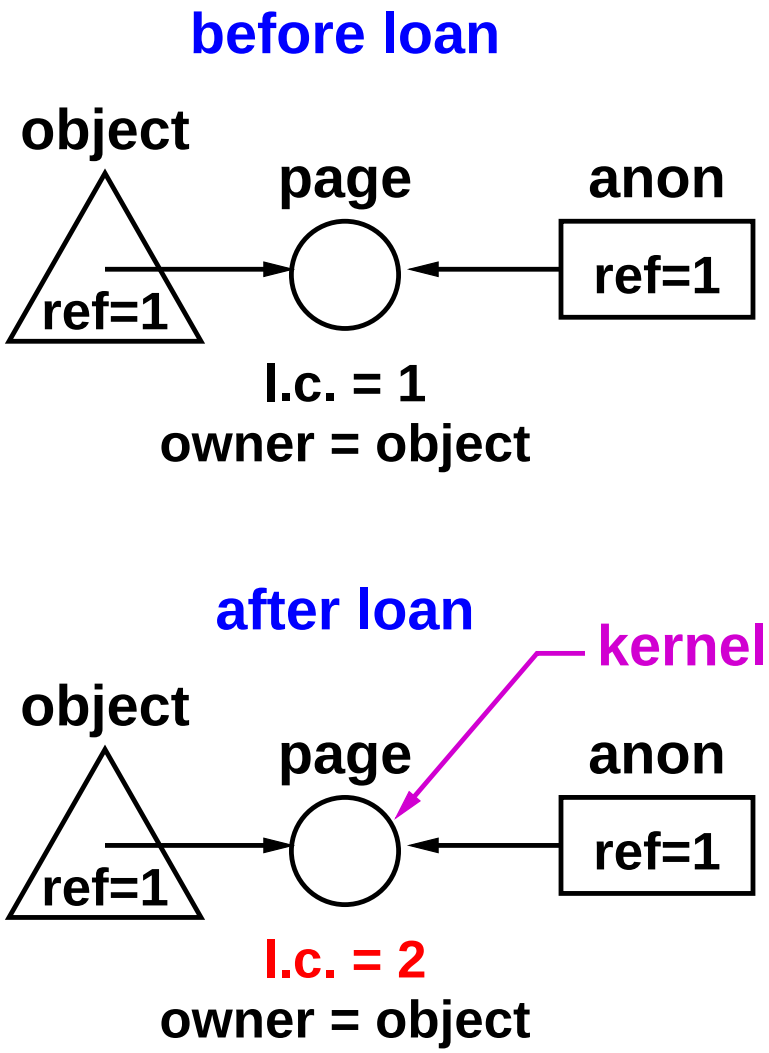
1. clean page (if dirty)
2. object drops ownership of the page
3. done: object isn't related to page anymore

Example: write fault on object page



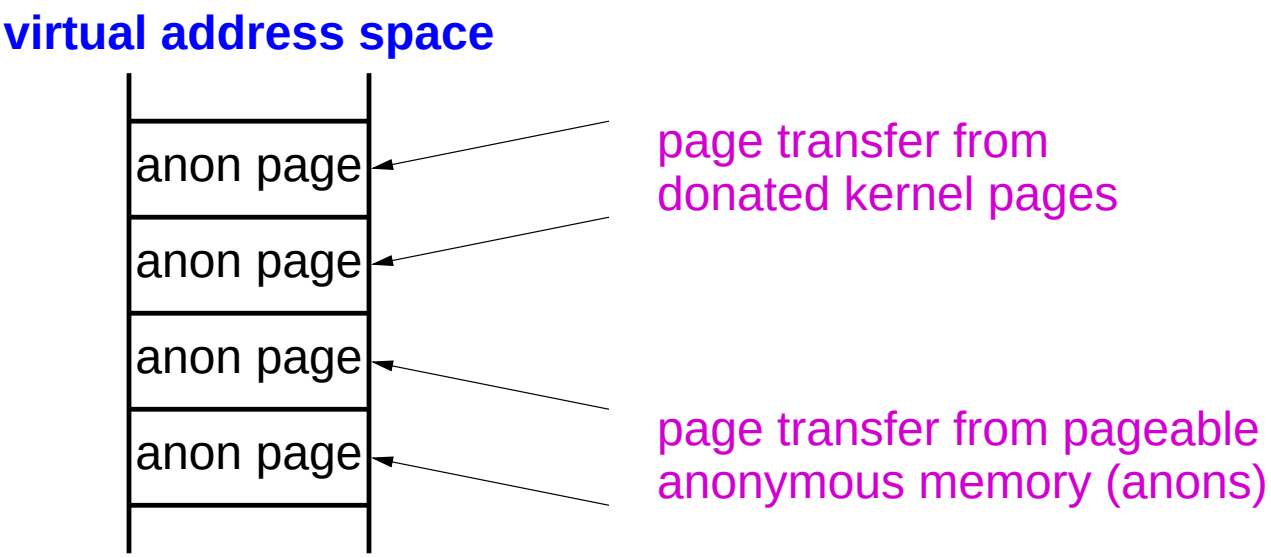
1. allocate a new page
2. copy data from the old page to the new page
3. object drops ownership of old page
4. new page inserted into object, fault can be resolved

Example: loaning to kernel



1. locking
2. increment loan count
3. wire page
4. return reference to page to kernel

UVM DETAILS: PAGE TRANSFER

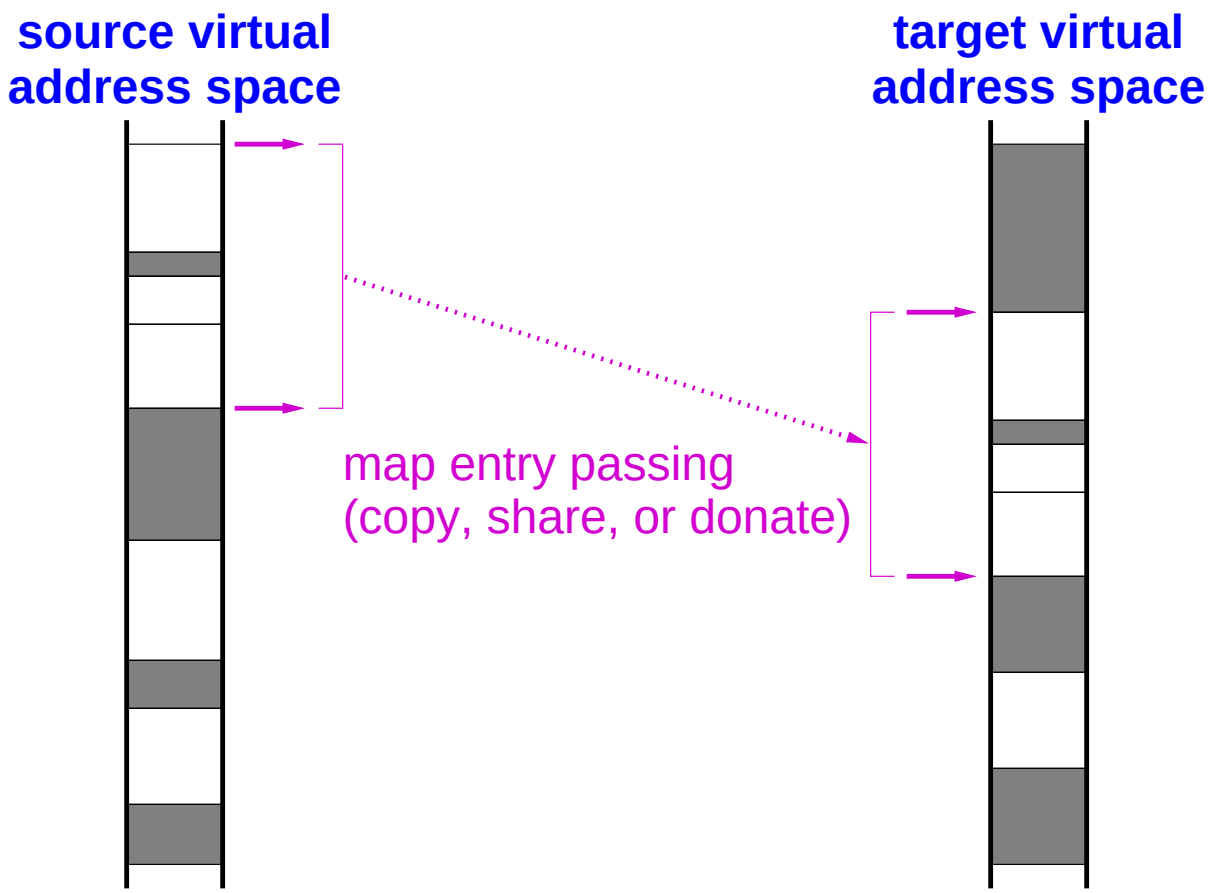


Transfer life-cycle

1. allocate anon for donated pages
2. find space in map for transfer
3. place anons in amap of target map
4. use data

- **to dispose of transfered page data:**
  - normal unmap
  - "anflush" operation
- **works with loaned pages**

MAP ENTRY PASSING



- **implemented with map extraction function**
  - copy, share, move/donate
  - quick reference supported
  - zero-fill on unmap
- **used by procfs and ptrace(2)**
- **sample API: dynamic export/import of memory**

Tests of the three data movement mechanisms

Test platform

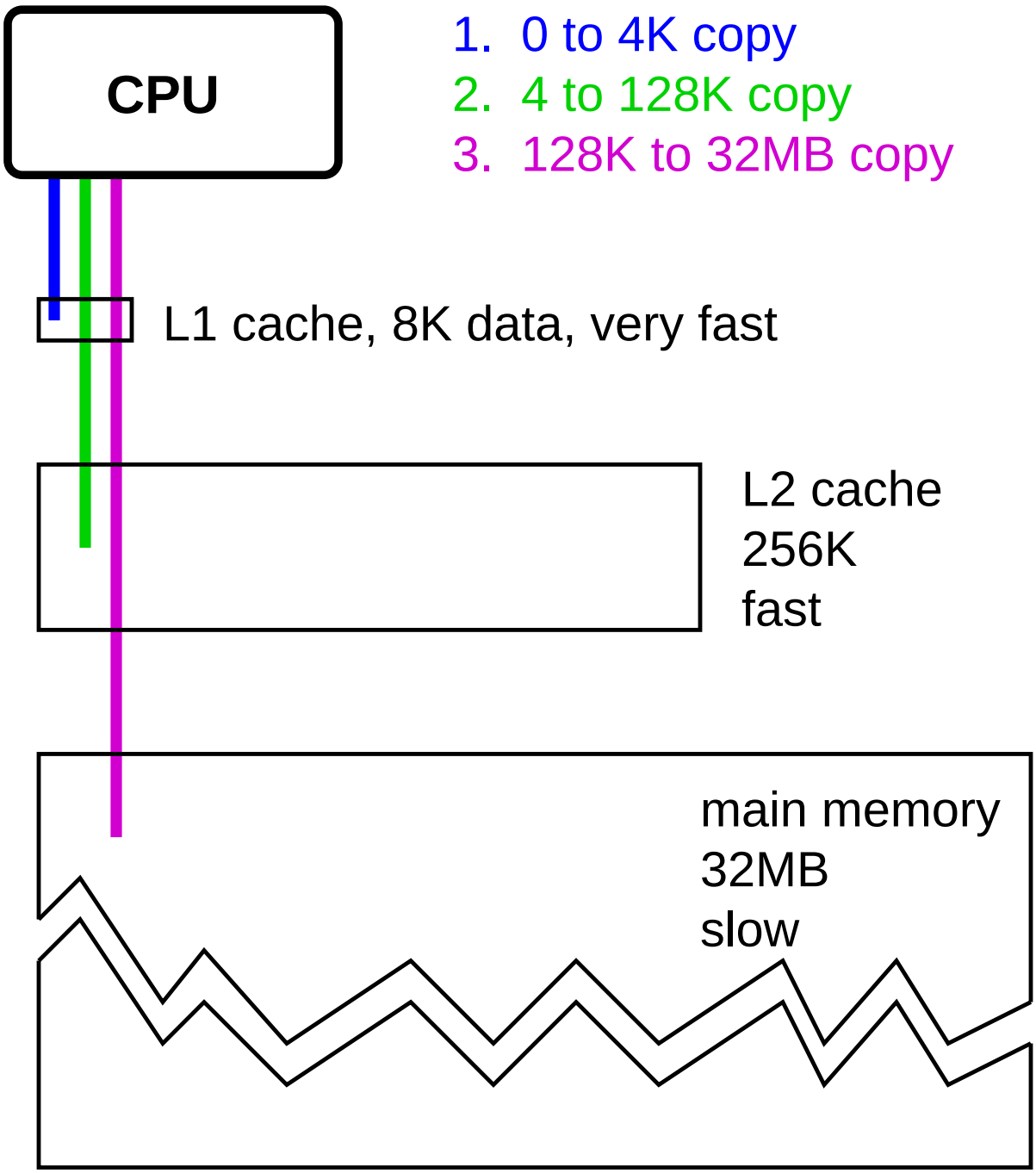
- Pentium PRO, 200MHz
- 32MB RAM
- Cache: L1=16K (8K instruction, 8K data), L2=256K

Main memory bandwidth*

- read = 225 MB/sec
- write = 82 MB/sec
- copy = 50 MB/sec

* from lmbench bw_mem_{rd,wr,cp}, rev 1.2

Caching effects



Loanout vs. data copy

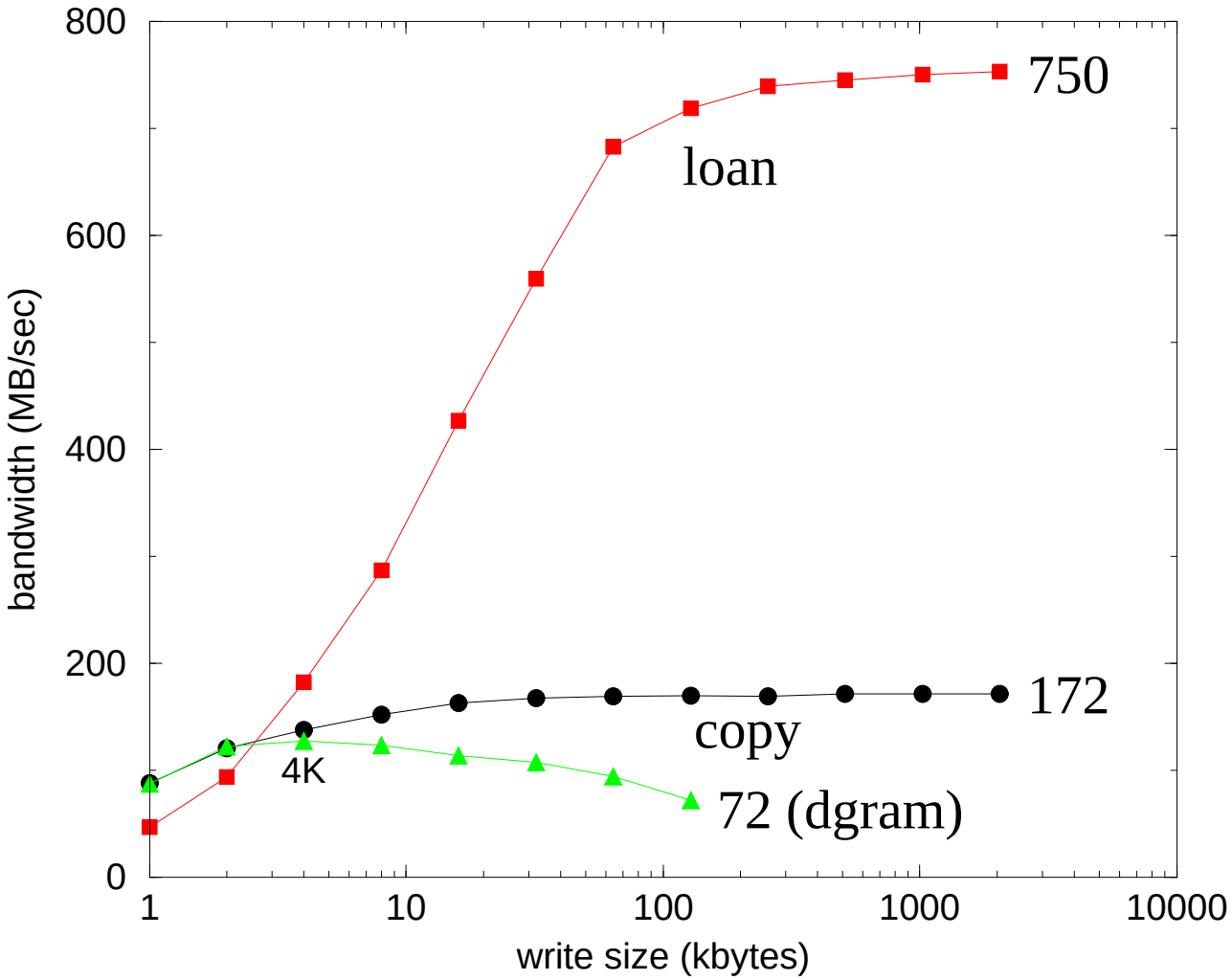
- **P6-200MHz can easily swamp available networks**
  - fast ethernet, 155 Mbps ATM
  - Instead add "null" protocol to kernel

- **Socket level modifications**
  - new write system call that uses a modified sosend() that does page loanout

- **Test program**
  - allocate a buffer
  - write buffer in fixed-sized chunks to null socket
  - repeat previous step the specified number of times
  - calculate bandwidth

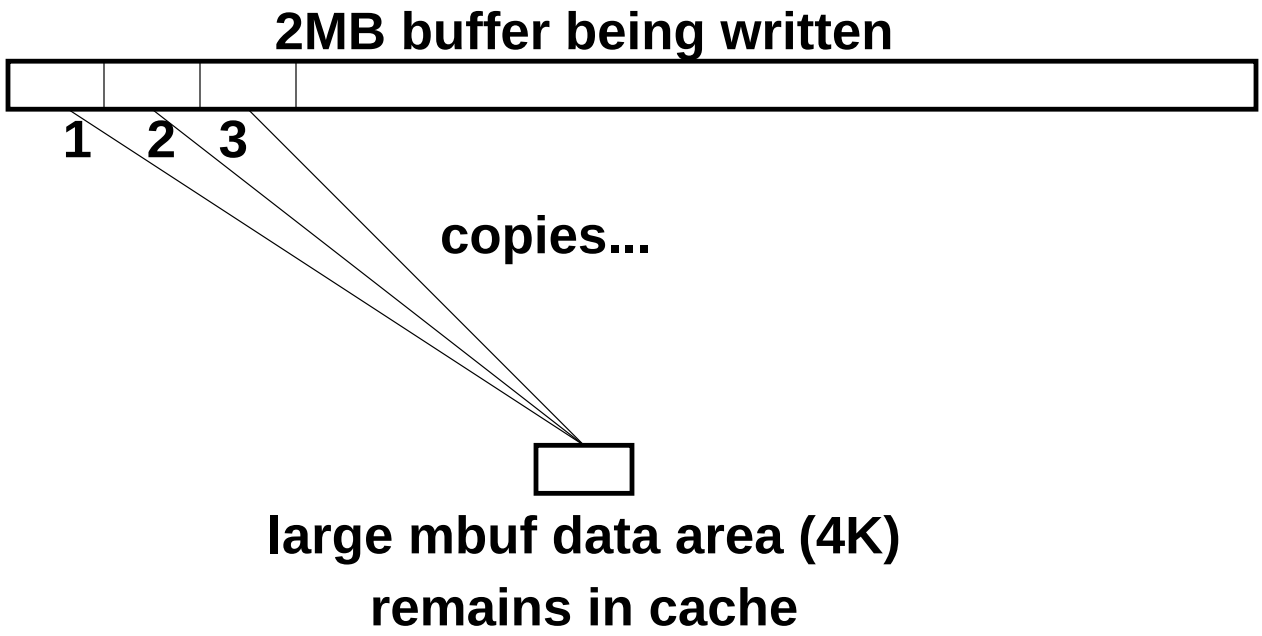
  we transfer 2GB of memory in our tests, varying the write size

PAGE LOANOUT MEASUREMENTS



- **shows the cost of touching the data**
- **for copy: mbuf area remains in cache**
- **flattening: loan size, kernel cache, TLB flush, socket buffer size**

Why copy bandwidth is > 50 MB/sec

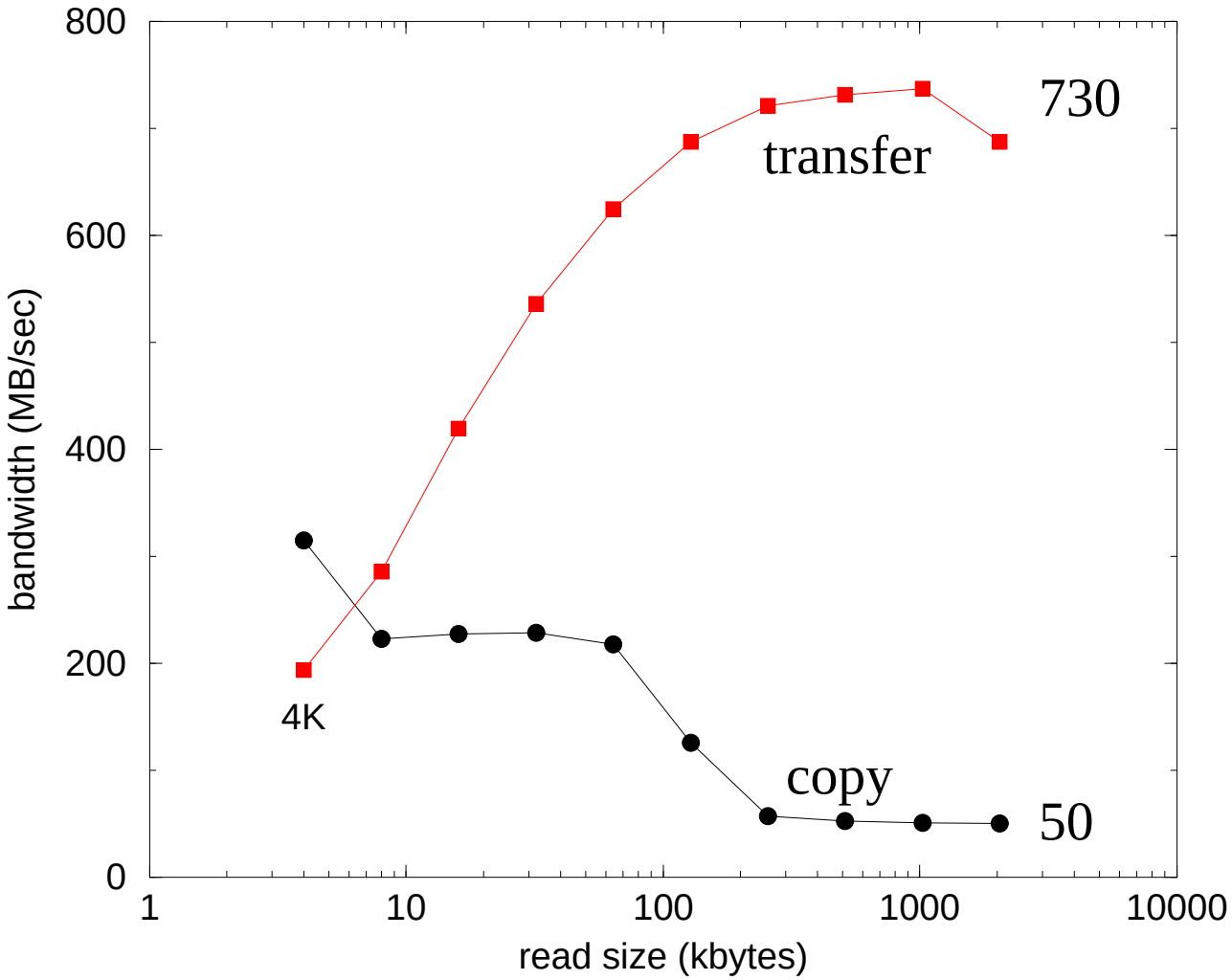


Page transfer vs. data copying

- **Socket modifications**
  - null read allocates an mbuf chain and passes it upward
  - use page sized large mbufs so we can use them for page transfer
  - new read system call that uses a modified soreceive()

- **Test program**
  - read "x" MB of data in "y" sized chunks
  - we read 1GB of data in our test

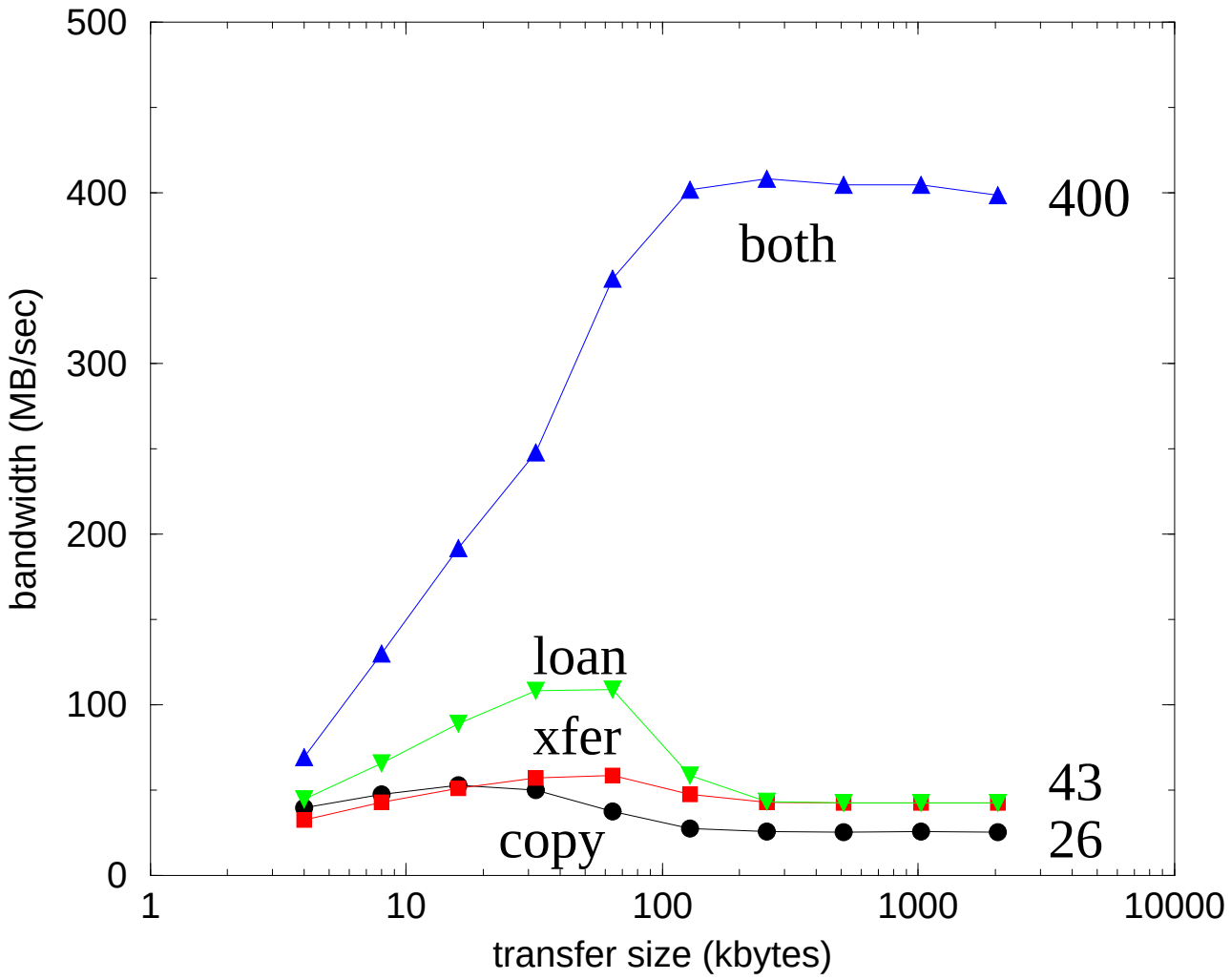
PAGE TRANSFER MEASUREMENTS



- **copy bandwidth drops at cache sizes**
- **networks: need big MTU (e.g. midway ATM)**
- **drop off: as amap footprint gets close to 8K**

Transfer data over a socket pair (pipe)

- **socket mods: detect mbufs with loaned out data**



- **touching the data reduces bandwidth**
- **flexibility of UVM: loanout/transfer at same time**

Two processes exchanging a buffer

bandwidth (MB/sec)	transfer size		
	4K	64K	4MB
Copy bandwidth	36	27	18
M.E.P. bandwidth	36	546	18500

- data not touched

UVM virtual memory system

- **Page loanout allows process to safely share a copy-on-write copy of its memory with other processes, the I/O system, or the IPC system**
- **Page transfer allows pages of memory from the I/O system, the IPC system, and other processes to be inserted easily in a process' address space**
- **Map entry passing allows processes and the kernel to dynamically exchange chunks of their virtual address space**
- **VM operations have been optimized to improve performance and reduce VM complexity**
- **Secondary aspects of the VM system improved**

FUTURE RESEARCH

- **Modify I/O and IPC systems to dynamically choose their data movement techniques based on available information**
- **Evaluate the effect of virtually addressed caches (VACs) on the effectiveness of UVM's data movement mechanisms**
- **Devise system call level APIs that can easily be wrapped within a higher-level middleware library**
- **Evaluate the effect of fine grain locking (multiprocessors) on UVM**
- **Use UVM to support APIC features**