An Efficient Zero-Copy I/O Framework for UNIX

Moti N. Thadani, et. al.

Technical Report of Sun Microsystems Lab.

1996, 1, 10

Park, Han Kyu

Outline

- Introduction
- Related Works
- Fast Buffers(*Fbufs*)
- UNIX Fast Buffers
- Implementation
- Performance Evaluation
- Conclusions and Future Work

Introduction

- Motivation
  - Advent of high-speed network and new application domain
    $\rightarrow$ to increase demands for efficient I/O data transfer
- Focus
  - To reduce the cost of data-copying between applications and operating system kernel
- Goal
  - To present an efficient zero-copy framework for buffer management and exchange between application programs and the UNIX kernel

Related Works

- Three main approaches
  - Integrated layer processing
  - Careful design of network adaptors
  - Restructuring operating system software

Fbufs - Basic Mechanism (1)

- Assumptions
  - I/O data is stored in buffers called *fbufs*.
  - A protection domain gains access to an fbuf either by allocating the fbuf or by receiving the fbuf.
  - An abstract data type can be layered on top of fbufs to support buffer aggregation.

Fbufs - Basic Mechanism (2)

- Steps to transfer an aggregate object
  1. Allocate an aggregate object (Originator)
     a) Find and allocate a free virtual address range
     b) Allocate physical memory pages and clear contents
     c) Update physical page tables
  2. Send aggregate object (Originator)
     a) Generate a list of fbufs from the aggregate object
     b) Raise protection in originator
     c) Update physical page tables

Fbufs - Basic Mechanism (3)

3. Receive aggregate object (Receiver)
   a) Find and reserve a free virtual address range in the receiver
   b) Update physical page tables
   c) Construct an aggregate object from the list of fbufs
4. Free an aggregate object (Originator, Receiver)
   a) Deallocate virtual address space
   b) Update physical page tables
   c) Free physical memory pages

Fbufs - Optimization (1)

1. Restricted dynamic read sharing

   – Only pages from a limited range of virtual addresses can be remapped
     $\rightarrow$ an fbuf is mapped at the same virtual address in the originator and all receivers
     (remove step 3a)

   – Write accesses to an fbuf while a receiver is holding a reference illegal
     $\rightarrow$ eliminate the need for COW (copy-on-write) mechanism

2. Fbuf caching

   – Write permissions are returned to the originator after the data is consumed by the destination domain
     $\rightarrow$ remove step 1a-c, 3a-b, 4a-c

Fbufs - Optimization (2)

3. Integrated buffer management/transfer

   – To incorporates knowledge about the object into the transfer facility

   → remove step 2a, 3c

4. Volatile fbufs

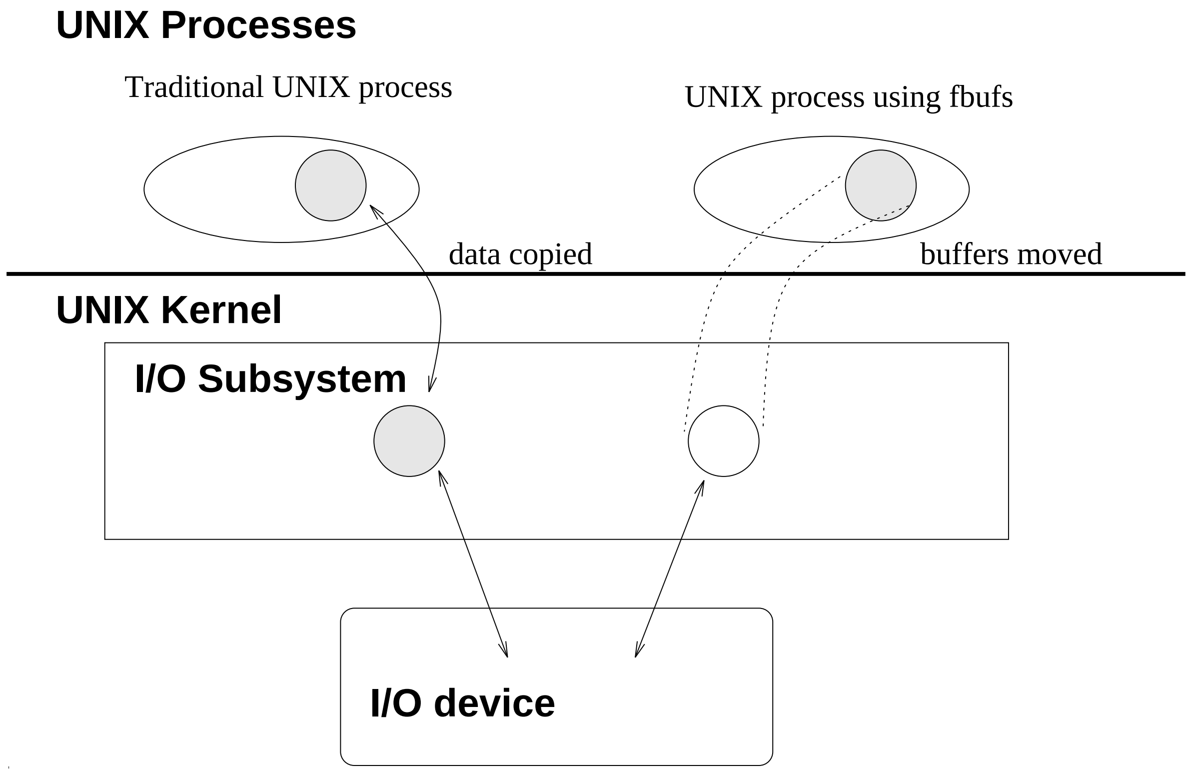
   – Receiver assumes that the contents of a received fbuf may change asynchronously unless it explicitly requests that the fbuf be *sequred*

   → the need for removing write permissions from the originator can be eliminated

UNIX Fast Buffers - UNIX I/O API (1)

- Traditional UNIX I/O interfaces
  - Data is *copied* between the kernel and user-defined buffers.
- UNIX fbufs scheme
  - On output, by passing an fbuf to the kernel, guarantee a saving in data-copying overheads.
  - On input, by receiving data in an fbuf, guarantee a saving in data-copying overheads.

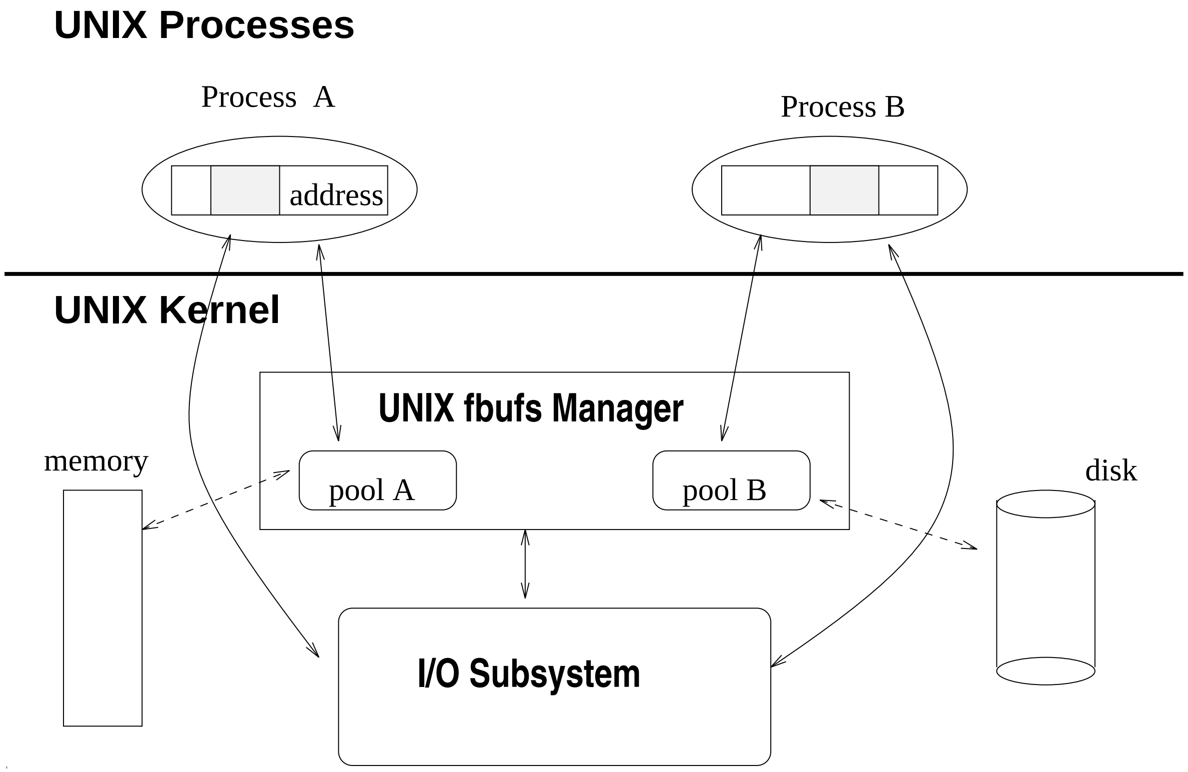
UNIX Fast Buffers - UNIX I/O API (2)



UNIX Fast Buffers - Buffer allocation (1)

- Properties of fbufs
  - Each fbuf is either owned by an application program or the kernel, or is in an fbufs buffer pool.
  - Each fbufs' buffer pool is associated with an application program.
  - No remapping of virtual addresses is required.
  - The I/O subsystem can allocate fbufs and place incoming data directly in them.

UNIX Fast Buffers - Buffer allocation (2)



UNIX Fast Buffers - API

- ssize_t uf_read(int fd, void **buffp, size_t nbytes)
- ssize_t uf_get(int fd, void **buffp)
- ssize_t uf_write(int fd, const void **buffp, size_t nbytes)
- void *uf_allocate(size_t size)
- void *uf_deallocate(void *ptr, size_t size)
- caddr_t uf_mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t off)

Implementation

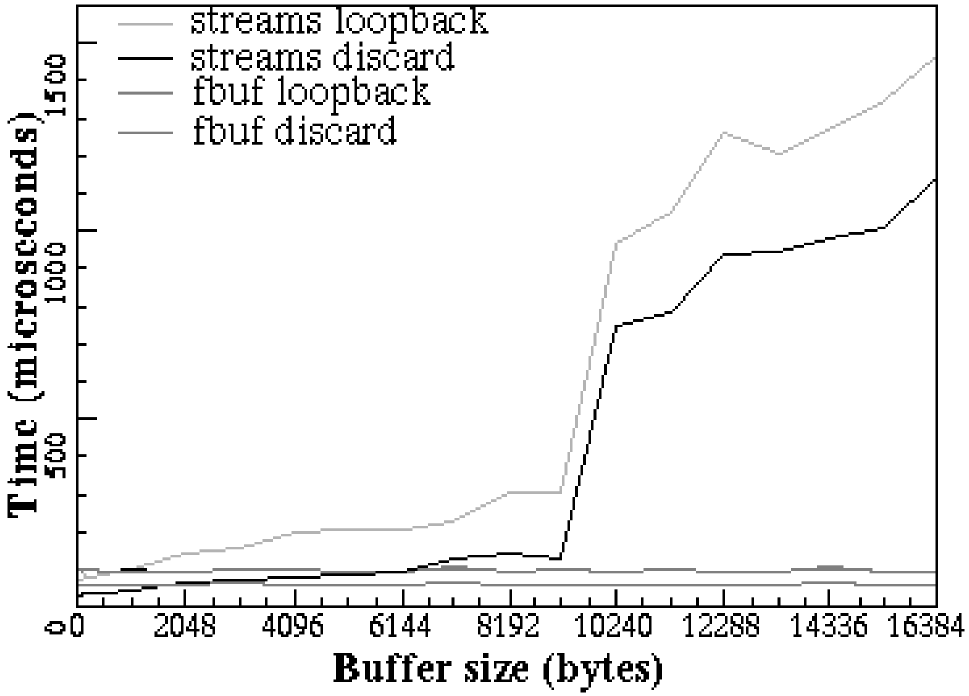
- Components
  - Library(libfbufs)
    - provides the fbuf interfaces to UNIX applications
    - uf_allocate, uf_deallocate, uf_read, uf_get, etc.
  - Buffer pool manager(pool mgr)
    - responsible for allocation of memory, managing mappings between user and kernel addresses
  - System call(fbufsys)
- Impact on the I/O subsystem
  - Device driver must be modified to request buffers from the specific buffer pool that corresponds to the process for which the I/O is initiated.

Performance

- Environment
  - Two uniprocessor 50MHz SPARCstation 10 workstations
  - ATM adapters capable of 140Mb/s
  - Solaris 2.4

Performance - Micro benchmarks

- To measure the latency of I/O operations

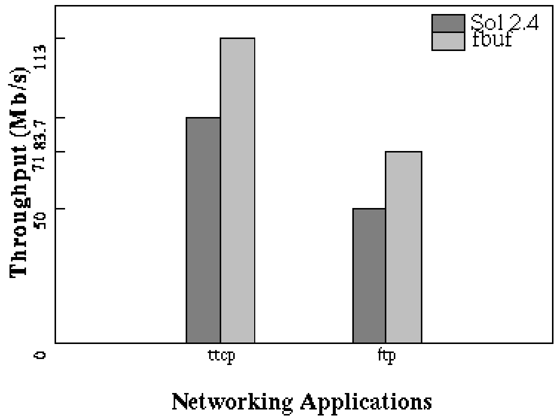


Performance - Macro benchmarks

- To show the improvement in throughput and CPU utilization
- Benchmarks
  - ttcp : modified by replacing the read and write calls by uf_read and uf_write calls to measure TCP throughput in fbufs based system
  - ftp : modified to use new interfaces

Performance - Macro benchmarks

- Throughput benchmarks



- CPU Utilization for ttcp

Node	Solaris 2.4	fbufs
receiver	99%	89%
sender	85%	62%

Conclusions and Future Work

- To present an efficient I/O mechanism based on fbufs mechanism
  - New application programming interface has been defined.
  - New utility routines for device drivers and operating systems components has been designed.
- Extentions
  - Device-to-device data-copying
    - If an fbuf is output to a device that can do DMA directly from the input device, a direct device-to-device data transfer can be performed without the data ever being copied to main memory.
    - Application candidates : Video servers