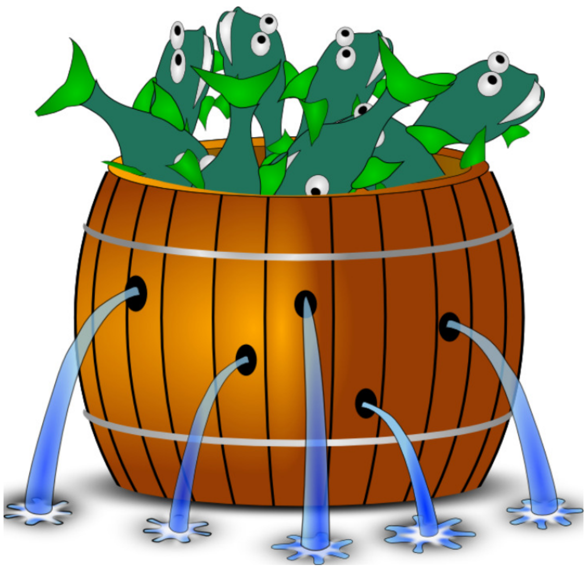
Barrelfish Project
ETH Zurich



Virtual Memory in Barrelfish

Barrelfish Technical Note 004

Akhilesh Singhania, Simon Gerber

02.06.2017

Systems Group
Department of Computer Science
ETH Zurich
CAB F.79, Universitätstrasse 6, Zurich 8092, Switzerland
`http://www.barrelfish.org/`

Revision History

Revision	Date	Author(s)	Description
1.0	08.02.2010	AS	Initial version
2.0	09.12.2013	SG	Updated with new kernel memory interface
3.0	02.06.2017	SG	Updated to match recent implementation changes

Chapter 1

Introduction

This document describes the virtual memory system of Barrelfish. It tries to describe the setup and the design decisions made in the construction of the system.

Chapter 2

Description

The virtual memory system is made up of four user space components listed below and utilizes the capability system to create hardware page tables.

- Vspace
- Vregion
- Memory Object
- Pmap

Vspace, vregion, and memory object are designed to be architecture independent. All architecture specific implementation for manipulating page tables is contained in pmaps.

The pmap code will invoke certain capability operations to create hardware page tables that match the user space vspace layout.

Each are described in more detail below.

2.1 Vspace

A pagetable is represented by a vspace object. The vspace object is associated with exactly one pmap object and a list of vregions. By maintaining a list of vregions, vspace also maintains a list of all virtual address regions that are currently mapped in.

To construct a new pagetable, a new vspace is created and it is associated with an appropriate pmap.

Pagefaults in a dispatcher should be directed to it. The vspace object looks up the appropriate vregion for the faulting address and passes the fault to it.

2.2 Vregion

A vregion represents a contiguous block of virtual address space. There is only one vregion for a block of address space in a page table. Therefore, it is associated with exactly one vspace and exactly one memory object.

The object also maintains a set of architecture independent mapping flags.

2.3 Memory Object

A memory object manages a block of memory of various types. Multiple vregions can be mapped into the same memory object.

Currently, the two prevalent memory objects are anonymous type (maintains a list of frames) and as an optimization a single frame memory object.

Further, a memory object of type pinned is implemented. Its functionality is near identical to the anonymous type except that it does not track the frames. This object is used to back memory required for metadata in the library. It can only be mapped into a single vregion.

Another memory object is of type single frame which maps portions of the frame lazily. This supports users that have a large frame that need not be mapped in its entirety.

2.4 Pmap

Pmap performs actual page table mappings. This is the only architecture specific portion of the virtual memory system.

The size of a virtual address space is architecture dependent and established at compile time. It is a property of the pmap.

Pmap is also responsible for deciding where a given memory object can be mapped. Since the vspace maintains the list of currently occupied regions, pmap may have to consult it.

For MMUs that do not offer address translation, pmap can simply inspect the memory object and return the address of the physical object itself.

2.5 Capability invocations

The pmap calls into the kernel using invocations on the page table and mapping capability types (namely `VNode_Map`, `VNode_Unmap`, and `Mapping_ModifyFlags`). These invocations are implemented in an architecture specific manner and are used to install, remove and modify mappings respectively.

In order for this system to make it possible to remove mappings when the underlying Frame capability is deleted, the `VNode_Map` invocation also creates a *Mapping* capability which records both the Frame capability and the page table entry at which the mapping was established. This mapping capability can then be used to refer to the mapping, e.g. when wanting to modify the mapping from read-write to read-only.

The map invocation can create multi-page mappings in one system call, as long as the mapping does not cross a page table boundary. In the case of mappings that cross page table boundaries, we need a map invocation per page table that the mapping touches.

Additionally, the map invocation can be used to create superpages (e.g. 2MB and 1GB pages on x86-64) by calling it on a x86-64 page directory or PDPT capability with a large enough Frame capability as the source capability.

Chapter 3

Known issues

Detail some of the known issues with the system.

3.1 Mapping the same memory object in different pagetables

Currently, a shared frame between two or more pagetables is represented by different memory objects with no association. The shared frame should be represented by the same object so that changes in one are reflected on the other. We envision that this maybe in the form of a single object with multiple instances which communicate via messages/sharing. Changes in one will be communicated to the others and operations will fail if not consistent.

3.2 Shared page table between domains

Actual mappings are visible between domains but the userlevel structures are not updated to reflect the mapping.

3.3 Memory object of type anonymous

Anonymous type memory objects can only be used if the MMU supports address translation. The reason is because the object reserves virtual address space when initialized and during initialization not all frames have been allocated yet. Currently, the memory object is used in the heap, slot allocator, spawning domains, pci domain, and vmkit domain.

3.4 Portability

Libbarrelfish initialization path will require knowledge of the architecture type so as to initialize the right type of pmap object.

A domain can only construct pagetables for an MMU of the same type as the one it is using.

Architecture of one type cannot compile pmaps for another architecture.

3.5 Page faults

Currently, we do not handle page faults in user space. Although the API is generally designed to allow for demand paging, this is not yet possible. Therefore, the user of the library explicitly uses the pagefault API to create mappings. This is hidden behind wrapper functions and most users of the library are shielded from this.