CS 2510: COMPUTER OPERATING SYSTEMS FALL 2017

Project 3 – Virtual Memory

Introduction

Managing memory is perhaps the single most important activity undertaken by an operating system. In this project we will explore some of the features that an OS memory management layer must provide to the rest of the operating system. As a result of this project you will implement a memory manager in the Linux kernel that operates in parallel with the built in management layer to provide virtual memory to a user space process.

The project will focus on implementing code in a Linux kernel module that will handle memory allocation requests made by a user space process. These allocation requests will be implemented by compiling linking a test program with `user/harness.c`. Your kernel module will be responsible for allocating a region of virtual address space for the process that matches the size of the allocation request. Once the virtual address space has been allocated your module will need to back the virtual memory to physical memory using the hardware page tables. The mapping will be done on demand based on page faults that occur as the process tries to access the allocated memory.

Code description

For this project you will be provided with a framework that implements a skeleton kernel module that you will fill in. Various helper functions have been provided for you to use. The kernel module consists of the following:"

- `Makefile` The makefile that handles compilation of the kernel module. You must edit this file to point it to the headers of the kernel running on your system. These should be available in a kernel-headers package that is part of your Linux distribution.
- `buddy.[ch]` A buddy system memory allocator. This will provide you a means of allocating memory pages from a memory pool made available to your module.
- `main.c` A file implementing the module and user space interfaces that the module will use. This initializes the module, and handles all communication from user space.
- `on_demand.[ch]` These files implement the on demand paging implementation that you will write. They currently consist of stub functions that you will need to flesh out.
- `petmem.h` A set of global definitions that will be needed by your code.
- `pgtables.h` A set of page table structures and macros that provide the hardware layout definitions for the page table structures. You will want to become familiar with this file's contents.

There is also a directory called `user` that contains the user space code needed for this project. The contents of this directory include:

- `Makefile` The makefile for the user space tools
- `harness.[ch]` The harness code that implements the interface to the kernel module. The usage of these functions is demonstrated in the test program.
- `petmem.c` This is a utility that claims a given amount of memory from the running linux kernel, and gives the kernel module direct control over that. For this to work, you will need a kernel configured with HOT REMOVABLE MEMORY enabled.
- `test.c` This is a test program that will exercise the virtual memory implementation. It currently includes a very simple test case, but you should augment it with more complicated cases.

Getting Started

For this project you will definitely need to have a test Linux environment available. Inserting kernel modules requires root access, so you will need either sudo permissions or the root password. By default an Ubuntu 16 environment should be configured correctly.

The first step is to make sure that the kernel module compiles and loads against the running kernel version. This is done by simply running `make` in the petmem directory. The kernel module should compile and generate the file `petmem.ko`. This is a kernel module that you can insert into a running kernel using the command `insmod petmem.ko`. Note that you will need to be root for this to work. To verify that it is working check the kernel message log by running `dmesg` to look for any errors. You should also check for the presence of the device file `/dev/petmem`. If this file is present then everything should be good to go.

Once the kernel module is inserted you will need to assign system memory to it using the `petmem` program. To do this you will need to compile the user space tools, by switching to the `user` directory and running `make`. This will create a program called `petmem` that you will run as root. `petmem` takes a single argument which is the amount of memory you want to assign to your kernel module. For this project you should probably stick with a small value, but don't worry if it rounds up the size up. Memory can only be offlined in blocks of 128MB, so any requested size will be rounded up to an appropriate multiple.

Once you have done these two steps, you are ready to run the test program. Since you haven't implemented anything yet, it should just segfault. We will discuss the user level API before getting into the kernel code.

User Level Interface

The interface to the kernel module is given in `harness.h`. To activate the memory manager you must first call `init_petmem()`. This creates a connection to the kernel module that will be used for communication. After initialization, you may allocate and free memory from the module using the functions `pet_malloc()` and `pet_free`. These functions work exactly like the standard `malloc()` and `free()` functions, but redirect to the kernel module. Note that while the default memory allocator incorporates a user space library that manages a virtually contiguous heap we are instead implementing it completely in the kernel. Therefore the allocation and free operations are passed directly to the kernel module, instead of being implemented in a user library.

The test program contains an example of how to use these functions. For this project it is recommended that you simply modify the existing `test.c`.

Kernel Module

The actual project implementation will take place in the context of the kernel module. The module itself leverages current the sparsity of the 64 bit address space in order to take over management for an unused region of the virtual address space. The bounds of this unused region are given in `petmem.h`. When a process starts it is allocated its own virtual address space with various regions used for the stack, heap, program code, and libraries. This leaves a number of unused regions in the address space that we can use without overwriting anything important. Because the process has a virtual address space from the main memory manager it already has a set of page tables set up for it. For this project we will add page table entries to the existing page table hierarchy. This is safe, because the page tables are blank for unused regions. Therefore we can modify these entries without screwing up any other state.

Because we are using existing page tables you will need to find the location of the root of the page table tree. This address is stored in the CR3 register. `petmem.h` and `pgtables.h` contains helper functions and macros that can extract the pointer for you. Once you have this pointer you will be able to add new page table entries (for allocation requests) and delete entries as well (for free requests). You will need to manage page table pages as well as data pages. These pages will be added to the page table hierarchy at the different levels below the root. **Important Note:** You will be responsible for freeing these pages as well as adding them to the hierarchy. Any page that you allocate must be freed before the program exits. Hint: A page table with no valid entries can be expressed as an invalid page table entry at the next level up in the hierarcy.

Implementing this virtual memory manager will require two separate components: An allocator of virtual address space to satisfy memory allocation requests, and an on demand mapping component that assigns allocated virtual memory to physical memory as it is accessed by the user space process.

Memory Allocation

The memory allocation functions will respond directly to user space allocation and free requests. These requests will operate on virutal memory that the process thinks it has. Allocations will consist of the number of pages that the process wants to map into its virtual address space. You will be responsible for finding an available range in the unused address region defined in `petmem.h`. Once you have allocated space in that region, you will need to let the process know where in that region the request was allocated. This will be the virtual address where the newly allocated range begins, and it is this address that the process will use directly. Remember that allocations and frees can happen multiple times and in any order. Therefore you must ensure that the ranges you allocate can be later freed and reallocated as many times as necessary. You are free to choose any of the memory management algorithms that we have covered, but First Fit is probably the easiest.

Accomplishing this task will require implementing a memory map data structure that tracks the unused memory region. This memory map is what you will use to locate free regions to satisfy an allocation and also to ensure proper deallocation as a result of a free() request. Once you have located a free region in the guests virtual address space (inside the unused region) the allocation is done; You do not need to locate physical memory yet. However when a free() request is made you will need to deallocate the physical memory backing the freed pages, and unmap it from the process address space. **NOTE:** Remember that you must notify the hardware with invlpg whenever you modify a page entry that might have been cached in the TLB.

Memory Assignment

Once a block of virtual memory has been given to the process as a result of a memory allocation, the process will probably start to access it using memory operations. Because the allocation only results in reserving the virtual address space, these operations will fail as there is no physical memory backing those addresses. These failures will generate page faults that vector into the Linux kernel. When the Linux kernel receives a page fault it will inspect its internal memory map to determine how to handle it. Because we are using our own memory map, Linux will believe that the address is invalid and most likely a pointer bug of some kind. The kernel's response to these events is to deliver a segfault to the process that generated the fault. Inside the harness code we have added a signal handler that intercepts these segfaults and converts them to page fault events that we then deliver to the kernel module. These events are then forwarded to your code via the `petmem_handle_pagefault` function. Your responsibility is to implement this function such that a page fault that occurs for an address inside an address range previously allocated by your module is mapped to an available physical memory page. This mapping is accomplished by updating the process' page tables to map the faulted virutal address to a given physical page.

The physical memory you will be using to back the virtual address space will be assigned from the offlined memory that was provided to your module via the `petmem` utility. This memory is managed by the module's buddy allocator, and provides an interface to both allocate and free memory pages. The interface for these two operations is included in `petmem.h` and consists of these two functions:

```
uintptr_t petmem_alloc_pages(u64 num_pages);
void petmem_free_pages(uintptr_t page_addr, u64 num_pages);
```

As you create the mapping for the faulting addresses you will need to allocate page table pages as well as pages to hold the processes data. For this project, all page operations should use the petmem allocation functions.

For this project we are not worrying about overcommitting of memory. Therefore, if you are unable to allocate a page from the buddy allocator it is acceptable to return an error. In this case the harness code will not catch the resulting segfault, and the process will terminate. However you should still allow the over allocation of virtual memory, since it will not need physical memory until it is actually accessed.

This project will most likely be a significant challenge. I was able to implement this project in 500 lines of code, if you find yourself writing considerably more than that then it is probably time to take a step back and re-evaluate your approach.

Submission

To submit your code, please send a tarball containing your modified files to jacklange@cs.pitt.edu and YUZ69@pitt.edu. Along with the code please include a readme file that explains the state of your code (What is working and what is not). Your grade will be based on whether your code passes a set of test scenarios (a subset of which are provided in your test.c file), and a manual inspection of the code itself. In order to make sure that you code works you will need to implement additional test scenarios in test.c. These test cases should be submitted with the kernel module code.