Project 3 - Memory Management *

Total Points: 100
Out: October 26, 2018
Due: 11:59 pm, Sunday, November 11, 2018

Introduction

The outcome of this project is to implement a series of system calls in the Linux kernel to report memory management statistics. The objective of this project is to get familiar with the following memory management components:

1. Process virtual address space.
2. Page tables.

Project submission

For each project, please create a zipped file containing the following items, and submit it to Blackboard.

1. A report that includes the name of your virtual machine (VM), and the password for the `instructor` account (details see *Accessing VM* below); and (3) a brief description about how you solved the problems and what you learned. The report can be in txt, doc, or pdf format.

2. The Linux kernel source files that your modified or created, and your user-level testing programs.

Accessing VM. Each team should specify the name of your VM that the instructor can login to check the project results. Please create a new user account called `instructor` in your VM, and place your code in the home directory of the instructor account (i.e., `/home/instructor`). Make sure the `instructor` has the appropriate access permission to your code. In your project report, include your password for the `instructor` account.

*

Project description

This project reviews the key memory management concepts we studied in class: virtual memory and page tables. Virtual memory often refers to the process address space assigned to user-space processes. Such virtual addresses need to be translated into physical addresses with the help of page tables. Please read Section 10.4 in our textbook to get an overview of Linux's memory management.

You tasks: This project consists of two parts. You are going to write two system calls (and a user-level test program for each call) to collect memory management statistics.

Part 1 (50 points)

Write a system call to report statistics of a *process's* virtual address space. The system call should take a process ID as input and output the size of the process's virtual address space in use. Additionally, write a user-level program to test your system call.

HINT: First, to get a process ID, you can use command `ps` to get the PID of `bash`.

The Linux kernel uses the *memory descriptor* data structure, `mm_struct`, to represent a process's address space. `mm_struct` is defined in `linux/mm_types.h` and included in a process's `task_struct` as a field called `mm`. If the `task_struct` of a process with `PID` is `task`, then `task->mm` is the process's memory descriptor. Within the memory descriptor, you can also find the work horse for managing program memory: a set of virtual memory areas (VMAs) of type `vm_area_struct`. All the VMAs together form a process's virtual address space.

A process's VMAs are stored in its memory descriptor as a linked list in the `mmap` field (`task->mm->mmap`). You may get a sense of how the `mmap` field is linked via the `vm_next` field, by looking at the second diagram in this blog post:

`https://manybutfinite.com/post/how-the-kernel-manages-your-memory/`

An instance of `vm_area_struct` fully describes a memory area, including its start and end addresses. To calculate the size of a virtual address space, you need to sum the sizes of individual VMAs.

Part 2 (50 points)

Write a system call to report the current status of a specific *address*. The system call takes a virtual address of a process (whose process ID is `pid`) and the `pid`, then outputs whether this address is in memory or on disk. Additionally, write a user-level program to test your system call.

HINT: To get a virtual address of a process, you can take the PID of `bash` in Part One, and pass the PID to command `pmap` to get a listing of the virtual addresses. Then pass one of the virtual addresses from `pmap`, to your user-level test program to see the results from your system call.

The page descriptor `page` defined in `linux/mm_types.h` contains information about the page. You need to figure out how to obtain a reference to the page descriptor given a virtual address and read information from the page descriptor. Note that Linux uses multi-level page tables, so you might need multiple steps to reach the page table entry (PTE) of a given virtual address.

http://www.cs.uccs.edu/~yzhuang/CS4500_5500/fall2018/slides/LEC10_Mem_Intro.pdf, on slide 35 we introduced how to navigate the page directories. Each process has its own pointer to what is called a page global directory (PGD). A page table entry is a PTE. To navigate the page directories, several macros are provided by the kernel to break up a virtual address into its component parts. For example, `pgd_offset()` takes a virtual address and the `mm_struct` for the process (the `mm` field of `task_struct`) and returns the PGD entry that covers the requested address. `pud_offset()` takes a PGD entry and an address and returns the relevant PUD entry. And `pmd_offset()` takes a PUD entry and an address and returns the relevant PMD entry.

Note: The output of `pmap` are hexadecimal strings. To use `pgd_offset(mm, address)`, `pmd_offset(mm,address)`, and `pte_offset(mm,address)`, the argument `address` has to be converted to unsigned long. Some more information can be found on this page: `http://www.chudov.com/tmp/LinuxVM/html/understand/node24.html`

Finally, we need to get the PTE from the PMD entry. This step can be a little tricky. `pte_offset(pmd, address)` could work but sometimes you would get kernel oops (`https://en.wikipedia.org/wiki/Linux_kernel_oops`). The reason is that `pte_offset()` returns an address of a page table entry, but the entry itself could be changed by the process at runtime. Therefore, you should try the following:

```
#include <linux/mm.h>
#include <linux/highmem.h>

spinlock_t *lock;
ptep = pte_offset_map_lock(mm, pmd, address, &lock);
...
pte_unmap_unlock(ptep, lock);
```

To test if an address is in memory, you need to use the macro `pte_present()` to test if the corresponding PTE have the `PRESENT` bit set. Note that `pte_present()`'s argument is a PTE, not the address of a PTE.