Homework 2 - Extent-based Page Table (100 pts + Extra Credit)

Due: November 20th, 2023, 11:55pm

In the second homework, inspired by the new virtual memory design, we will build a software-based page table that supports multiple page size granularity. Remember that larger page sizes reduce the number of TLB misses and the pagetable size. The homework will have two parts (and an optional third extra-credit part).

PART 1

The first step would be to write a simple benchmark to measure the cost of the system call; you will add a new dummy system call (hello_kernel) to your OS in the Linux kernel's mm/mmap.c file. A couple of references below show how to add a new system call; there are thousands of other references online.

Reference 1 Reference 2

Once you have added a system call, measure the cost of invoking a system call from userspace by invoking the system call few hundred times and measuring the average latency.

NOTE: Make sure that your system call is getting called by adding a **printk()** within your dummy system call. You should remove the print statement after confirming that the new system is getting invoked inside the kernel.

PART 2 (40 pts)

In the Linux page fault function, you will create a new structure called *Extent* that groups a set of *physically* contiguous pages into a standard structure. An *extent* represents some range of contiguous addresses. So, your extent structure could contain a linked list (or radix tree or Xarrays) of physically contiguous pages, the number of pages in the list, the starting physical address and the ending address, and an extent ID (an incrementing number). First, to create an extent structure, you must allocate the extent in the kernel (using kmalloc()). Each element of the linked list (in the extent structure) has a physical address of a page and the next pointer. To learn about the kernel linked list in the Linux kernel, use the below reference.

Kernel Linked List

Next, each process can have several extents (regions of physically contiguous regions). To maintain all these extents (and easily lookup), you will add each extent into a software page table. We will maintain the page table as a red-black tree indexed by the starting physical address of the extent.

Each time there is a page fault, a page gets created. You will get the physical address of the page, find the corresponding extent in the red-black tree (if one exists or create a new extent), add the page to a linked list of the extent, increment and update the necessary extent fields and continue.

See the following link to learn about the Linux red-black tree. Kernel Red-black tree

Required output: Before the process exits, you will print the number of extents for a user-level process. Use your homework 1's page-fault benchmark that you developed for calculating the number of extents. Verify your implementation with any other application that you have used in the past. To evaluate the correctness of your approach, I will be using a custom benchmark to measure the number of extents.

Your code must also be thread-safe. Use a multi-threaded program to check the thread safety of your code.

PART 2 (20 pts)

Required output:

(a) In part 2 of the homework, first, you will measure the additional software overhead for maintaining the page table and extent. You will use the page fault benchmark (homework 1) to compare your extent implementation against page fault handler without extent.

Required output:

(b) Your current x86 platforms store one TLB entry for each page. If we had the capability to store the extent address (that maps to a group of physically contiguous pages) in the TLB, instead of one 4K pages, how many TLB entries can Do we save? Provide a simple calculation by measuring the number of page faults suffered by a process (your page-fault benchmark) and compare it with the number of extents.

References

To remind you, the page faults for anonymous memory are handled in the following function inside mm/memory.c source file.

```
int do_anonymous_page(struct vm_fault *vmf)
```

This function first checks whether the page fault is a read or write fault (due to read or write access), then (1) allocates a page, (2) creates a new page table entry (PTE), and (3) adds the PTE to the page table.

Reference 1 Reference 2

PART 3 (Extra Credit 10 pts)

Parts 1 and 2 mainly focus on anonymous pages (i.e., heap pages). But note that a significant number of memory pages are also allocated for I/O buffer cache pages, mapped to the page table and cached in TLBs.

In the third extra-credit part, you will generate the extents for the I/O cache pages. The mechanism for I/O cache pages is the same as anonymous pages, but you must handle faults for I/O pages (find the function similar to do_anonymous_page() that handles faults for I/O pages.

Resources

Similar to HW 2, you will use the QEMU-based virtual machine to test the modified OS. QEMU VMs run either on a bare-metal OS or even inside another virtual machine. More details on QEMU can be obtained [here] (https://www.qemu.org/).

#For students new to QEMU or hacking kernel, we have created a set of #instructions about how to compile a custom kernel (OS) and how to run the #OS using QEMU. Detailed step-by-step instructions can be found here.

`https://github.com/SudarsunKannan/CS519`

Computing Resource

If you need access to a development environment and cannot use your laptop, please send me an email (sudarsun.kannan@cs.rutgers.edu).

Starting Early

This a significant-but-essential homework for understanding the basics of OS virtual memory. Please start working on this homework early. If you have questions, make sure to ask them during office hours.