CS 537 Spring 2020, Project 3: xv6 Memory sub-system

Teaming up!

For this project, you have the option to work with a partner. Read more details in Submitting Your Implementation and Collaboration.

Administrivia

- **Due Date** by Mar 5, 2020 at 10:00 PM
- **You have two submissions for this assignment**.
- Questions: We will be using Piazza for all questions.
- This project is to be done on the lab machines (https://csl.cs.wisc.edu/services/instructional-facilities), so you can learn more about programming in C on a typical UNIX-based platform (Linux).
- Please take the **quiz on Canvas** that will check if you have read the spec closely. NOTE: The quiz does not work in groups and will have to be taken individually on Canvas.
- Your group (i.e., you and your partner) will have *2* slip days for the rest of the semester. If you are changing partners please talk to the instructor about it.

Objectives

- To understand how to add the memory protection for pages in xv6 system.
- To understand how the process of page allocation works and add two basic page allocator implementations.

Overview

In the first part of project, you'll be adding read only access to some pages in xv6, to protect the process from accidentally modifying it.

You will be implementing 2 system calls:

- `int mprotect(void *addr, int len)` to add read-only protection to pages.
- `int munprotect(void *addr, int len)` to give read-write permission to pages back.

In the second part, you will implement two basic page allocation schemes. The first scheme allocates frames alternatively to prevent malicious writes. The second scheme allocates frames randomly. You also need to implement a random number generator for second scheme. You will implement a system call `int dump_allocated(int *frames, int numframes)` which dumps information about the allocated frames.

Reading:

- Chapter 2 (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf) in xv6 book.

Part 1: **Read-only** Code

In most operating systems, code is marked read-only instead of read-write. However, in xv6 this is not the case, so a buggy program could accidentally overwrite its own text. Try it and see!

In this portion of the xv6 project, you'll change the protection bits of parts of the page table to be read-only, thus preventing such over-writes, and also be able to change them back.

To do this, you'll be adding two system calls: `int mprotect(void *addr, int len)` and `int munprotect(void *addr, int len)`.

Calling `mprotect()` should change the protection bits of the page range starting at `addr` and of `len` pages to be read only. Thus, the program could still read the pages in this range after `mprotect()` finishes, but a write to this region should cause a trap (and thus kill the process). The `munprotect()` call does the opposite: sets the region back to both readable and writeable.

Also required: the page protections should be inherited on fork(). Thus, if a process has mprotected some of its pages, when the process calls fork, the OS should copy those protections to the child process.

There are some failure cases to consider: if `addr` is not page aligned, or `addr` points to a region that is not currently a part of the address space, or `len` is less than or equal to zero, return -1 and do not change anything. Otherwise, return 0 upon success.

Hint: after changing a page-table entry, you need to make sure the hardware knows of the change. On 32-bit x86, this is readily accomplished by updating the `CR3` register (what we generically call the *page-table base register* in class). When the hardware sees that you overwrote `CR3` (even with the same value), it guarantees that your PTE updates will be used upon subsequent accesses. The `lcr3()` function will help you in this pursuit.

Page-allocation

As you know, the abstraction of an address space per process is a great way to provide isolation (i.e., protection) across processes. However, even with this abstraction, attackers can still try to modify or access portions of memory that they do not have permission for.

For example, in a Rowhammer (https://en.wikipedia.org/wiki/Row_hammer) attack, a malicious process can repeatedly write to certain addresses in order to cause bit flips in DRAM in nearby (physical) addresses that the process does not have permission to write directly. In this part, you'll implement one (not very sophisticated) ways to alleviate the impact of a malicious process: allocating pages from different processes so that they are not physically adjacent.

One of the advantages of page-based systems is that a virtual page can be allocated in any free physical page (or frame); all physical pages are supposed to be equivalent. However, not paying attention to frame allocation could allow a malicious process to repeatedly write to addresses at the edge of one of its pages, which due to the low-level properties of modern DRAM, could then modify the values at the edge of the next physical page which happens to belong to a different process.

- The current physical memory allocator in xv6 uses a simple free list. You can find the routines for allocating physical memory and managing this simple linked list in `kalloc.c`. By looking at `kinit()` you can see how the free list is initially created to include all physical pages across a range and you can see that kalloc() simply returns the first page on that free list when a frame is needed.
- Note that the free list starts with the highest numbered free frame and begins sorted in reverse order.
- You will modify these allocation routines to implement your new allocation scheme.
- Note that kalloc() should fail (i.e., return NULL) when it cannot find a suitable page.

There are a different levels of sophistication for page allocation to reduce the physical memory wastage. For this project however, we ask you to write two very basic implementations. In both these implementations, you will be modifying the `kalloc()` method.

Part 2: Alternate allocation

In this implementation, you keep a free frame between every allocated frame in the system. For example, if kalloc() sees 3 frame allocations for process a, then 2 frame allocations for process b, and then 2 for process a again, it will end up allocating the pages of physical memory to processes as follows (F represents a free page): aFaFaFbFbFaFa. This approach satisfies our security requirements, but uses twice as much memory as the original xv6 approach.

The changes for alternate allocation as well as Part 1 should be submitted in p3a subdirectory. We will run test cases for these two parts in your p3a subdirectory.

Part 3: Random Allocation

In this implementation, the allocator assigns random entries in the free list. For this purpose, you will need to implement a pseudo random number generator, which gives a random number each time. We have provided a header file `rand.h` below.

```
#ifndef _RAND_H
#define _RAND_H
#define XV6_RAND_MAX = 2147483647

/*
    Return a random integer between 0 and XV6_RAND_MAX inclusive.
    NOTE: If xv6_rand is called before any calls to xv6_srand have been made, the same
    sequence shall be generated as when xv6_srand is first called with a seed value of 1.
*/
int xv6_rand (void);

/*
    The xv6_srand function uses the argument as a seed for a new sequence of pseudo-random nu
mbers to be returned by subsequent calls to rand.
    If xv6_srand is then called with the same seed value, the sequence of pseudo-random numbe
rs shall be repeated.
    If xv6_rand is called before any calls to xv6_srand have been made, the same sequence sha
ll be generated as when xv6_srand is first called with a seed value of 1.
*/
void xv6_srand (unsigned int seed);
#endif // _RAND_H
```

You have to implement the two functions `xv6_rand()` and `xv6_srand()` in `rand.c`. The `xv6_rand()` method generates a random number on each call and `xv6_srand()` method is used to set a seed. We recommend using the Xorshift family (https://en.wikipedia.org/wiki/Xorshift) of random number generators, although you are welcome to use other approaches.

Each time in your kalloc implementation, the allocator should call the xv6_rand() method to generate a random number `x`. Then, it takes its remainder with the current free list size to get `y`, and returns the `yth` frame from start in the free list. For example: if the remainder is 0, the first frame in free list is allocated, when remainder is 1 second frame is allocated .. and so on. Also, don't forget to remove the allocated frame from the free list!

Details

- You have to keep both `rand.h` and `rand.c` in the kernel subdirectory.
- You should not change the initialisation of free list portion in the `kinit()` method. Our test cases assume standard free list initilisation. You can however use variables to maintain size of free list and history of allocated frames.
- The `xv6_rand()` method generates a random number based on some seed. When the seed is not specifically set using `xv6_srand()` method, it assumes seed of 1. However, when a seed is set, the next call to `xv6_rand()` should give the first random number corresponding to that seed.
- For ex:, suppose for some `xv6_rand()` method implementation, the following code gives output as below:

```
xv6_srand(2);
printf(" %d\n", xv6_rand()); //outputs 13438
xv6_srand(5);
printf(" %d\n", xv6_rand()); //outputs 64829
```

Then for the following code also, output should be independent of previous seed.

```
xv6_rand();
xv6_srand(5);
printf(" %d\n", xv6_rand()); //should output 64829
```

The changes for random allocation should be submitted in p3b subdirectory. We will run test cases for random allocation on submissions in p3b subdirectory.

Dump allocated pages

You also have to implement a system call to dump the pages which have been allocated. This will help you debug your kernel and us to test your code.

```
int dump_allocated(int *frames, int numframes)
```

- **frames**: a pointer to an allocated array of integers that will be filled in by the kernel with a list of all the frame numbers that are currently allocated
- **numframes**: The previous *numframes* allocated frames whose information we are asking for.
- **Note**: You only have to return the frame numbers of the frames which have been allocated, not the ones left empty. For example, your freelist has frames with address as follows: `(23, 19, 16, 15, 13, 8, 5, 3)` . It then allocates four frames with address `8, 19, 5, 15` in order. The freelist should now become `(23, 16, 13, 3)` . And a call to `dump_allocted(frames, 3)` , should have `frames` pointing to array `(15, 5, 19)` .
- Return -1 on error (e.g., something wrong with input parameters, or numframes is greater than number of allocated pages till now) and 0 on success.
- **This method should work correctly for both the allocation schemes above and be present in both submissions.**

Submitting Your Implementation

1. Please create two subdirectories ~cs537-1/handin/LOGIN/p3a/src and ~cs537-1/handin/LOGIN/p3b/src.

```
mkdir -p ~cs537-1/handin/LOGIN/p3a/src
mkdir -p ~cs537-1/handin/LOGIN/p3b/src
```

To submit your solution, copy all of the xv6 files and directories with your changes for alternate allocation and Part 1 into `~cs537-1/handin/<cs-login>/p3a/src` , and changes for random allocation into `~cs537-1/handin/<cs-login>/p3b/src` . One way to do this is to navigate to your solution's working directory and execute the following commands:

```
cd <your p3a working directory>
cp -r . ~cs537-1/handin/<cs-login>/p3a/src/
cd <your p3b working directory>
cp -r . ~cs537-1/handin/<cs-login>/p3b/src/
```

If you choose to work in pairs, only one of you needs to submit the source code. But, both of you need to submit an additional partner.login file, which contains one line that has the CS login of your partner (and nothing else). If there is no such file, we are assuming you are working alone. If both of you turn in your code, we will just randomly choose one to grade. Consider the following when you submit your project:

- If you use any slip days you have to create a `SLIP_DAYS` file in the `/<cs-login>/p3/` directory otherwise we use the submission on the due date.
- Do not remove the `README` file in the main directory
- Do not change the permissions
- Do not remove the other files that you are not modifying
- Do not modify or remove the Makefile
- Your files should be directly copied to `~cs537-1/handin/<cs-login>/p3a/src` directory. Having subdirectories in `<cs-login>/p3a/src` like `<cs-login>/p3a/src/xv6-sp20` or … is **not acceptable**.

Collaboration

This project is to be done in groups of size one or two (not three or more). Now, within a group, you can share as much as you like. However, copying code across groups is considered cheating. If you are planning to use git or other version control systems (which are highly recommended for this project), just be careful not to put your code in a public repository.