P6: xv6 Memory Encryption, Kernel Version

Due Monday by 11:59pm **Points** 100

You can work on this project with one other partner. P6 is due Monday, April 19.

Updates:

- An early version of this spec talked about manually setting a reference bit in software. That has now been removed in favor of the hardware-managed PTE_A bit
- dump_rawphymem CANNOT simply use copyout anymore. Explained below.
- Add more detailed implement hints to each step in the Hint section.

Objectives

- To learn about the virtual memory system in xv6
- To understand page table entries in detail
- To modify page table entries to be able to detect the current state of a page
- To modify the trap handler to be able to handle the page fault
- To implement the clock algorithm to keep track of the most referenced page

Background

The page tables, traps, and interrupts of xv6 are described in Chapters 2 and 3 of the **xv6 book. (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)**

We're going to assume that you've completed P5 and know everything from that writeup. Note that many parts of P5 can be copied over to P6. You might want to start with your P5 implementation or you might want to start with a clean version of xv6.

You can find a copy of unmodified xv6 source code in `~cs537-1/projects/xv6.tar.gz`. Copy it into your private directory and untar it, if you'd like.

```
prompt> cp ~cs537-1/projects/xv6.tar.gz /path/to/your/private/dir
prompt> tar -xvzf xv6.tar.gz
```

In your Makefile, make sure you set CPUS = 1 and **change the compilation flag from O2 to O0 (not Og).**

Main Idea: Kernel Memory Encryption Pager

We will explore the idea of letting the kernel manage page encryption and decryption. Encrypting pages with the P5 mencrypt interface required modifying each application and required each application to know which pages were worth encrypting. In this version, the system will keep a fixed number of

recently-accessed pages for each process stored in cleartext (decrypted). The idea is to minimize the number of page decryptions (and re-encryptions) by keeping each process's working set in cleartext.

Let the constant N be the number of recently accessed pages that should be tracked for each process (and kept decrypted). Add the following line to `param.h` which defines N:

```
#define CLOCKSIZE 8   // CLOCKSIZE represents N above
```

When an encrypted page is accessed by the user, a page fault should be triggered (same as P5). If the number of currently decrypted pages of the calling process is smaller than N, then this page should be decrypted and pushed to the tail of a queue (see details in the next paragraph). If the calling process already has N decrypted pages, we need to find a victim page to replace.

In order to decide the victim page, you will implement a clock (also called FIFO with second-chance) algorithm. To implement this algorithm, you need to (statically) allocate a clock queue for each process. The clock queue is a queue-like structure storing all the virtual pages that are currently decrypted. The other essential part of a clock algorithm is a reference bit that gets set to 1 every time a page is accessed. Luckily for us, x86 hardware sets the **sixth** bit (or fifth if you start from 0. 0x020, to be precise) of the corresponding page table entry to 1 every time a page is accessed. Let's call this bit **PTE_A**. See Figure 2-1 (Page 30) in the xv6 reference **book** (https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf) for more details. The hardware-managed access bit should be cleared by the kernel (in software) at the appropriate time and have the hardware automatically set the bit when that page is accessed.

To select a victim, examine the page at the head of the queue. If the head page has been accessed since it was last enqueued (PTE_A is one), then clear the reference bit and move the node to the tail of the queue; the victim selection should proceed to the next page in the queue. Repeat this procedure until you find a head page that has not been accessed since it was last enqueued (reference bit is zero); this page should be evicted. When a page is "evicted", it should be encrypted and the appropriate bits in the PTE should be reset. When a virtual page is decrypted, it should be placed at the tail of the clock queue. The hardware will subsequently set PTE_A to 1, but there's no harm in manually setting it.

Make sure that pages that are in the working set are in cleartext (PTE_E is 0) and do not generate page faults (PTE_P is 1).

Note that

- In this part of the project, new user pages (including the program text, data, and stack pages) start as encrypted and are ONLY decrypted when accessed.
- Same as P5, when a child process is created, its initial memory state (including whether or not a page is encrypted) and clock queue, should match that of its parent.
- If a decrypted page is deallocated by the user, it should be removed from the clock queue.
- If a process calls exec(), it starts with fresh memory, and thus the working set should be emptied. All user-level pages should be encrypted.

Example 1:

Suppose we have one running process A, N = 2, and the state of the clock queue (leftmost is the head) is as follows:

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 1

When A accesses virtual page 0x5, a victim page should be selected. In the first iteration, the reference bit of both virtual pages is cleared and the order is not changed. On the second iteration, virtual page 0x3 will be selected as the victim. The resulting state of the clock queue will be the following:

Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x5 Reference bit: 1

Example 2:

Suppose we have one running process A, N = 4, and the state of the clock queue (leftmost is the head) is as follows:

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x5 Reference bit: 0	Virtual page number: 0x6 Reference bit: 1

When A accesses virtual page 0x4, the reference bit of virtual page 0x4 should be set to 1 again,

Virtual page number: 0x3 Reference bit: 1	Virtual page number: 0x4 Reference bit: 1	Virtual page number: 0x5 Reference bit: 0	Virtual page number: 0x6 Reference bit: 1

Then A access an encrypted virtual page 0x7. Virtual page 0x5 will be chosen as the victim and the resulting state would be

Virtual page number: 0x6 Reference bit: 1	Virtual page number: 0x3 Reference bit: 0	Virtual page number: 0x4 Reference bit: 0	Virtual page number: 0x7 Reference bit: 1

Statistics

In order to gather statistics about your memory system and test your implementation, you will implement two syscalls. ~~Only **getpgtable** is different from P5.~~ **BOTH** functions are slightly different from before.

int getpgtable(struct pt_entry* entries, int num, int wsetOnly)

will allow the user to gather information about the state of the page table. The parameter **entries** is an array of pt_entries with **num** elements that should be filled up by the kernel. If **wsetOnly** is 1, you should filter the results and only output the page table entries for the pages in your working set. If **wsetOnly** is 0, then this function does not filter by the working set. Return an error if **wsetOnly** is any other value.

```
struct pt_entry {
    uint pdx : 10; // page directory index of the virtual page
    uint ptx : 10; // page table index of the virtual page
    uint ppage : 20; // physical page number
    uint present : 1; // 1 if page is present
    uint writable : 1; // 1 if page is writable
    uint user : 1;  // 1 if page belongs to user
    uint encrypted : 1; // 1 if page is currently encrypted
    uint ref : 1; // 1 if reference bit is set
};
```

The kernel should fill up the entries array using the information from the page table of the currently running process. Only valid virtual pages should be considered. In addition, filling up the array starts from the valid virtual page with the highest page numbers. For instance, if one process has allocated 10 virtual pages with page numbers ranging from 0x0 - 0x9, then page 0x9 - 0x7 should be used to fill up the array when **num** is 3. Assume all these three pages are in the clock queue, then the array should look as follows (ppage might be different):

```
0: pdx: 0x0, ptx: 0x9, ppage: 0xC3, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
1: pdx: 0x0, ptx: 0x8, ppage: 0xC2, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
2: pdx: 0x0, ptx: 0x7, ppage: 0xC1, present: 1, writable: 1, user: 1, encrypted: 0, ref: 1
```

The file should be copied from `~cs537-1/projects/memory-kernel/ptentry.h`.

Do not edit `ptentry.h`. **Note that P6's ptentry.h is different from ptentry.h of P5.**

int dump_rawphymem(uint physical_addr, char * buffer)

will allow the user to dump the raw content of one physical page where **physical_addr** resides (This is very dangerous! We're implementing this syscall only for testing purposes.). The kernel should fill up the buffer with the content of the page where **physical_addr** resides. **buffer** will be allocated by the user and have the size of **PGSIZE**. You are not required to do any error handling here.

UPDATE: Simply using copyout will not be sufficient. The buffer might be encrypted, in which case you should decrypt that page. Otherwise, when the user program tries to read buffer, it'll read flipped values. It's as simple as either using the buffer's uva for memmove (which copyout does not do) or touching the buffer using `*buffer = *buffer` before copyout.

Hints

We would suggest you break down this project into the following steps:

1. Implement the decryption mechanism when an encrypted page is accessed (already done in P5) and init all the user pages as the encrypted state. ~~You should do this whenever a process grows or shrinks (check out growproc()) or when a new program is executed (check out exec() in exec.c).~~

[UPDATE]

All the user pages are allocated through the function `allocuvm()` in `vm.c`, but directly encrypting all the pages in `allocuvm()` might not work out as you might expect. The reason is that another system call `exec` will call `allocuvm` to init those text, data, and stack pages and copy program content (e.g. program text) into it. If you do the encryption inside the `allocuvm()`, then you probably need to modify other functions like `loaduvm()` and `copyout()` to make sure that those pages are decrypted before the content is copied into it. Considering the difficulty you will encounter, we encourage you to do the initial memory encryption in two parts:

1. Encrypt all the newly-allocated heap pages in `growproc()`. These pages are allocated by the user through syscall `sbrk()` which will call `growproc()`.
2. Encrypt all those pages set up by the exec function at the end of the exec function. These pages include program text, data, and stack pages. These pages are not allocated through `growproc()` and thus not handle by the first case.

2. Add the clock queue mechanism. Make sure you encrypt pages when they get kicked from the queue and add pages to the queue when you decrypt them.

[UPDATE]

In this step, you will mainly extend your memory decryption implementation. Before you do that, you will need to determine what's kind of data structure you wish to use as a queue. This data structure should be able to append elements to the tail, check the head element, and remove elements in the middle of the queue. One of the choices is to use the linked list implementation similar to the one in P4. Another choice is to use a ring buffer as a queue. A ring buffer can be implemented using an array that representing the queue and an index pointer pointing to the head of the queue. Each method has its pros and cons that you need to deal with. So choose whatever way make you feel comfortable to implement. See TA's discussion material for more information.

Remembering that we should remove the pages from the queue if it's deallocated. `deallocuvm` might be one of the candidate places to remove the pages from the queue. But you should be aware that you can't directly call `myproc()` to get the clock queue in `deallocuvm` since this function could be called by the parent process to deallocate pages for the child process.

3. Modify the corresponding code to handle the `fork()` behavior and deallocation of pages. This is mainly just queue management.

[UPDATE]

As we mentioned above, the child process should inherit the page table entry flags and clock queue from the parent process. You probably want to check out the `fork()` function in `proc.c` and `copyuvm()` in `vm.c`. The modification mainly involves copying extra flags and clock queue. If you are using a linked list as the queue, you should do a deep copy instead of just copying the pointer.

Make sure you fully test your code after each step.

Code Delivery

Handing in Your Code

EACH project partner should turn in their joint code to each of their handin directories.

So that we can tell who worked together on this project, each person should place a file named **partners.txt** in their handin/p6 directory. The format of **partners.txt** should be exactly as follows:

```
cslogin1 wisclogin1 Lastname1 Firstname1
cslogin2 wisclogin2 Lastname2 Firstname2
```

It does not matter who is 1 and who is 2. If you worked alone, your **partners.txt** file should have only one line. There should be no spaces within your first or last name; just use spaces to separate fields.

To repeat, both project partners should turn in their code and both should have this **partners.txt** file in their handin/p6 directory.

Within your p6 directory, make the following directories and place your xv6 code in them as follows:

```
~cs537-1/handin/<login>/p6/ontime/src/<xv6 files>
```

If you wish to use slip days in this project, then you should submit your code to the corresponding slip directory: **slip1**, **slip2**, or **slip3. slip1** indicates that you wish to use one slip day in this project. We will use the latest submission for grading. This is saying that if you submit both at slip3 and slip1, then we will use the version submitted at slip3 to grade.

Testing

We strongly recommend you first write a few small user programs to test various aspects of this project. A simple user application could look as following

```
char *ptr = sbrk(PGSIZE); // Allocate one page
struct pt_entry pt_entry;
// Get the page table information for newly allocated page
// and the page should be encrypted at this point
getpgtable(&pt_entry, 1, 0);
ptr[0] = 0x0;
```

```
// The page should now be decrypted and put into the clock queue
getpgtable(&pt_entry, 1, 1);
```

Ensure correct behavior from these tests before moving on to our tester. The tester is at `~cs537-1/tests/p6/run-tests.sh` If you want to run just test n, you can run `~cs537-1/tests/p6/run-tests.sh -t n` On any CSL machine, use `cat ~cs537-1/tests/p6/README.md` to read more details about how to list the tests and how to run the tests in batch. Note that there will be a small number of hidden test cases (25%).

NOTE: In your Makefile, make sure you set CPUS = 1 and change the compilation flag from O2 to O0 (not Og). Otherwise, the test suite wouldn't work due to the reason that the compiler would optimize out some of the unnecessary access used in the tests.

Slip Day Policy

A maximum of 3 slip days can be used for this project no matter you are working with a partner or not. Additional 2 slip days for each one have been added as described in this **post (https://piazza.com/class/kjn4sz4kq7t2d2?cid=596)** .

If you are working with a partner, then

- Each of you will only need to contribute 1/2 of any slip days you use; for example, if you use 1 slip day, each of you is charged 1/2 of a day.
- If only one of you runs out of slip days, the needed slip days will be taken from the partner who still has them.
- A 1/2 slip day can't be used (unless you are combining with a 1/2 slip day from your partner). We will assume you are aware of your partner's slip days and the implications.