P5: xv6 Memory Allocation

Due Nov 4, 2019 by 5pm **Points** 100

You can work on this project with one other partner.

Objectives

- To learn about the virtual memory system in xv6
- To modify how physical memory can be allocated (compared to a simple linked list)

Background

The page tables and memory subsystem of xv6 are described in Chapters 1 and 2 of the **xv6 book. (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf)**

Main Idea

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 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 project, we are using security attacks as a motivation for our changes in xv6, but our purpose is really to understand how to change the xv6 memory system in controlled ways.

In this project, you'll implement one (not very sophisticated) way to alleviate the impact of a malicious process: allocating pages from different processes so that they are not physically adjacent

You will also implement a routine to allow a user to gather statistics about your memory system and test your implementation.

Page Allocation

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 **freerange()** 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 (i.e., 0xdfff) and begins sorted in reverse order.

To translate an address allocated by kalloc() to a page number, you should convert the address from a virtual address in the kernel's address space to a physical address, and then shift and mask off the page offset from a 32-bit address to obtain the frame number; you can find the format of addresses in **mmu.h** and **memlayout.h**

You will modify these allocation routines to ensure that there is always a free page between pages belonging to different processes. Note that **kalloc()** should fail (i.e., **return NULL**) when it cannot find a suitable page.

There are a few different levels of sophistication you could try for your new allocator. Some number of points will be given based on how efficiently you use physical memory (i.e., how much of physical memory you can correctly allocate before **kalloc()** fails). For simplicity in this project, we limit you to one of two deterministic implementations.

1. Keep a free frame between every allocated frame in the system. How to do easily this? Look at how the original free list is constructed and only add every other frame to this list. This approach satisfies our security requirements, but uses twice as much memory as the original xv6 approach; allocations will fail much more quickly with this approach than they could with a better implementation. A base level of points will be allocated for this approach.

2. In this implementation, you must keep the list of free physical pages **in (reverse) sorted order and allocate the first frame that fits the security requirements** (i.e., that has either free pages or pages belonging to this same process as both neighbors).

   For example, if kalloc() sees 5 frame allocations for process A, then 3 frame allocations for process B, and then 2 for process A again, it will end up allocating the pages of physical memory such that they belong to processes as follows (F represents a free page): AAAAAFBBBFAAFFFFF. How well this works will depend on the pattern of **kalloc()** and **kfree()** that you see for a given workload. For example, the worst case occurs if kalloc() sees an alternating stream of frame allocations for process A and B in which case it will still end up allocating a wasted free frame between each of those allocations; that is, the pages of physical memory will be allocated as: AFBFAFBFA...

   This approach has two challenges. First, you must be able to ***identify*** the process for which **kalloc()** is allocating a frame (or a few UNKNOWN cases). For most calls to **kalloc()** throughout the kernel code, you will be able to determine (without too much difficulty) the process this page is being allocated to (this is sometimes the calling process, but not always). For example, when **fork()** calls **copyuvm()**, you should ensure that the copy is associated with the child process. Or, when **exec()** or **sbrk()** eventually call **allocuvm()**, you should ensure the pages are associated with the calling process. The final case is that you should associate the kernel stack of a process with that process

(see **allocproc()**).

The three exceptions are 1) the allocations performed in **inituvm()** for the initcode, 2) those performed by **walkpgdir()** to allocate page tables, and 3) any allocations performed in **setupkvm()** for the kernel's page tables. You may mark those pages as belonging to an UNKNOWN process. Pages belonging to an UNKNOWN process may be allocated next to other pages from an UNKNOWN process.

The second challenge is that you must ***track*** which frames have been allocated to which processes; you can use whatever data structure you would like to track this information. For simplicity, we recommend statically allocating the data structures for tracking the process associated with each frame. For this project, you can assume that the maximum number of page frames that will ever be simultaneously allocated will be no more than **16384**.

Remember to correctly update your data structures and your free list (keeping it sorted) when pages are freed. Remember that kalloc() must return the FIRST page (in reverse sorted order) that meets the security requirements. And, remember, that UNKNOWN pages can be placed next to other UNKNOWN pages.

If you implement this approach correctly, you will receive full points for this portion of the project.

Details: You only need to apply this approach to the memory initialized by **kinit2()** (i.e., not **kinit1()**) and gather statistics only after the **kinit2()** has been called.

Statistics

Of course, to help you debug your kernel and for us to test your code, you will need to add the following system call:

- **int dump_physmem(int *frames, int *pids, int numframes):** This system call is used to find which process owns each frame of physical memory.
  - **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
  - **pids**: a pointer to an allocated array of integers that will be filled in by the kernel with the pid that owns the corresponding frame number in the frames list; that is, if **frames[i]** holds frame number 23 and frame number 23 has been allocated to pid 57, then **pids[i**] holds 57.
  - **numframes:** the number of elements in the frames and pids arrays; numframes is guaranteed to be less than 16384. (Note the number of elements is different than the size of the arrays!) Unused elements in the frames and pids lists below numframes should both be set to -1.
  - Notes: If you do not know the pid that owns an allocated frame, you should set the corresponding **pids** element to -2. The corresponding process may not be known for two reasons. First, you could be implementing strategy 1 and you never tracked any pid information (you just know that a

frame has been allocated); in this case, you must set the corresponding pid to -2. Second, the owning process might be UNKNOWN because, as stated above, it was allocated as part of **inituvm()**, **walkpgdir(), or setupkvm().**

- Notes: Unallocated frames should not show up in your frames array at all. In your implementation, be sure to clear the process ownership information when a frame is freed!
- Example: if frames 18, 17, and 16 have been allocated to pid 55, frame 14 is UNKNOWN, and frames 12 and 11 are for pid 99, and no other frames have been allocated, the two lists would be set as follows:
  frames[0] = 18; pids[0] = 55
  frames[1] = 17; pids[1] = 55
  frames[2] = 16; pids[2] = 55
  frames[3] = 14; pids[3] = -2 (UNKNOWN)
  frames[4] = 12; pids[4] = 99
  frames[5] = 11; pids[5] = 99

  All remaining elements of frames and pids up to numframes-1 must be set to -1.

- Return -1 on error (e.g., something wrong with input parameters) and 0 on success.

Hints

You can break this project up into three steps.

v1a: We recommend that you first implement approach 1 to make sure that you have something that meets the requirements. Track statistics so you know which frames are allocated, but not to which processes (all pids in your statistics will be set to UNKNOWN, or -2).

v1b: Then, figure out which process is associated with each frame. This will require modifying a few function call signatures to pass PIDs. Track this information in your statistics. Some of the processes will still be marked as UNKNOWN, according to the code paths described above.

v2: Finally, implement approach 2 in which you allocate contiguous pages when possible, according to the requirements above.

Each of these three steps will be worth additional points. If you have implemented v2 and it works correctly, you will be also awarded the complete points for the v1a and v1b versions. Similarly, if you implement v1b correctly, you will also be awarded the points for the v1a version.

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/p5 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/p5 directory.

Within your p5 directory, make the following directories and place your xv6 code in them as follows:

~cs537-1/handin/<login>/p5/src/v1a/<xv6 files>

~cs537-1/handin/<login>/p5/src/v1b/<xv6 files>

~cs537-1/handin/<login>/p5/src/v2/<xv6 files>

Depending on what you have implemented, you can submit code into just 1, 2, or all 3 of those directories.

If you have implemented v2 and it works correctly, you will be also awarded the complete points for the v1a and v1b versions. Similarly, if you implement v1b correctly, you will also be awarded the points for the v1a version. You don't have to submit code for earlier versions (and you don't have to make those empty directories).

However, if v2 or v1b is only partially working, we recommend submitting the earlier versions to ensure you receive credit for whatever you have working.

You must use this directory structure so we know exactly what version(s) you have implemented.