CS 4375 Fall 2025
Homework 4: Lazy Allocation for xv6
75 points

MD Armanuzzaman

Dude Date: November 10, 2025 (11.59 PM))

xv6 currently allocates and maps physical memory frames for all allocated virtual memory pages. Most operating systems wait to allocate a physical memory frame until a virtual memory page that is part of the heap space is accessed. This lazy allocation saves both time and space is not all of the allocated virtual memory is actually used. For this assignment, we will implement lazy allocation of the heap for xv6 and also allow overcommitment of the physical memory.

Task 1. `freepmem()` system call (15 pts)

You are given the code for an xv6 free command in the file `free.c`. You should place this code in the user directory and add the command to `UPROG` in Makefile. The code calls the `freepmem()` system call which does not currently exit. The user prototype for `freepmem()` is

```
uint64 freepmem(void);
```

You should place this declaration in `user/user.h` and add an entry for freepmem in `usys.pl`. You will also need to modify `kernel/syscall.h`, `kernel/syscall.c`, `kernel/sysproc.c`, and `kernel/kalloc.c` to implement the `freepmem()` system call. After you implement the `freepmem()` system call, test it using the provided free command.

You are provided with another test program in `memory-user.c`. This program calls `sleep()` so that you can run it in the background and run the free command between allocations. Show and explain the results when you do this. How much memory can you request before `malloc()` fails?

Note you can run the `memory-user` program in the background and get back the user control of the shell by use "&" as below:

```
memory-user 1 100 10 &
```

Sample out put for task 1:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ memory-user 1 100 10 &
$ allocating 0x0000000000000001 mebibytes
malloc returned 0x0000000000003010
free
Free memory: 132300800 bytes
$ freefreeing 0x0000000000000001 mebibytes

Free memory: 132300800 bytes
$ freeallocating 0x000000000000000B mebibytes
malloc returned 0x0000000000103020

Free memory: 120741888 bytes
$ free
Free memory: 120741888 bytes
$ freeing 0x000000000000000B mebibytes
free -m
Free memory: 115 MB
$ allocating 0x0000000000000015 mebibytes
malloc returned 0x0000000000C03030
free -mfreeing 0x0000000000000015 mebibytes

Free memory: 94 MB
$ free -m
Free memory: 94 MB
$ allocating 0x000000000000001F mebibytes
malloc returned 0x0000000000203030
free -m
Free memory: 94 MB
$ freeing 0x000000000000001F mebibytes
free -mallocating 0x0000000000000029 mebibytes
malloc returned 0x0000000002103040

Free memory: 53 MB
$ free -mfreeing 0x0000000000000029 mebibytes

Free memory: 53 MB
$ free -m allocating 0x0000000000000033 mebibytes
malloc returned 0x0000000001703040
```

```
Free memory: 53 MB
$ free -mfreeing 0x0000000000000033 mebibytes

Free memory: 53 MB
$ free -m allocating 0x000000000000003D mebibytes
malloc returned 0x0000000000D03040

Free memory: 53 MB
$ free -m freeing 0x000000000000003D mebibytes

Free memory: 53 MB
$ free -mallocating 0x0000000000000047 mebibytes
malloc returned 0x0000000000303040

Free memory: 53 MB
$ free -mfreeing 0x0000000000000047 mebibytes

Free memory: 53 MB
$ free -m
Free memory: 53 MB
$ allocating 0x0000000000000051 mebibytes
malloc returned 0x0000000000000000
malloc failed //failed, available memroy 53MB, requested 81MB
```

Task 2. Change `sbrk()` so that it does not allocate physical memory (15 pts)

The `sbrk()` system call grows the heap space if needed to satisfy a `malloc()` request. Note that the heap space is in the process's virtual address space. The end of the heap space is marked by the `sz` field in `struct proc`. Currently `sbrk()` calls `growproc()` which calls `uvmalloc()` to allocate physical memory for the newly allocated virtual pages. Modify `sbrk()` so that it changes to `sz` field in `struct proc` – i.e., so that it allocates virtual memory space – but remove the call to `growproc()`. After this modification, xv6 will generate a `usertrap()` (page fault) when a process accesses its newly allocated heap memory, but since there is no page-fault handler implemented yet, the process will be killed and the system may crash.
Sample output of this task (your output may not be exactly the same):

```
usertrap(): unexpected scause 0x000000000000000f pid=3
sepc=0x00000000000012c0 stval=0x0000000000004008
panic: uvmunmap: not mapped
```

Task 3. Handle the load and store faults that result from Task 2 (15 pts)

Note the value of scause from Task 2. Accessing unallocated memory will result in a load or a store fault. See the RISC-V Privileged ISA document for the relevant exception codes. To handle the load and store faults that result from Task 2, you will need to check for these exception codes in `usertrap()` in `kernel/trap.c`. If the exception is a load or store fault, then you should also check if the faulting address (which will be in the stval register) is within the process's allocated virtual memory. If it is, then it is a valid virtual address and the fault occurred because physical memory has not been allocated and the page table mapping has not been installed. In this case, you should handle the fault by allocating a physical memory frame (hint: use `kalloc()`) and installing the page table mapping for the virtual page that contains the faulting address (hint: `use mappages()`).

Task 4. Fix kernel panic and other errors. (15 pts)

After you complete Task 3 and attempt to run xv6 commands, you will find that you get some errors, such as a kernel panic from `uvmunmap()` since that function expects all virtual memory pages to have been mapped. You may see other errors as well. Find and fix the errors. Try to figure out what is causing each error, rather than trying random things to try to fix it. You may use `memory-user` process to find the panics/errors with and without background process ("&").

Similar to task 1, after fixing the panic or other errors you may still run out of memory. In your implementation you should write appropriate message to indicate when it happens (hint: kalloc will return 0).

Task 5.Test your lazy memory allocation. (15 pts)

Test your lazy memory allocation thoroughly and show the results. Test cases should include: 1) allocating and freeing memory without touching it, 2) allocating and touching memory, 3) allocating memory and randomly touching just some pages.
You can modify `memory-user.c` for all three cases, put your output in the report and discuss your findings:

1. the given test code satisfies this case. In the terminal (follow the steps in task 1), if you run `free` in between mallocs, you will see that the free memory does not change as we have implemented lazy allocation. The expected partial result is shown below:

```
$ memory-user 1 100 10 &
...
$ allocating 0x0000000000000015 mebibytes
malloc returned 0x0000000000C03030
free -m
```

```
Free memory: 127 MB
$allocating 0x000000000000001F mebibytes
malloc returned 0x0000000000203030
free -m
Free memory: 127 MB
```

In the above output, even if we are allocating 0x15 and 0x1F MB of memory the free return value shows lazy allocation is properly implemented, as the amount of free physical memory is constant when the memory is untouched.

2. Uncomment the piece of code after "CASE 2: Touch every page (uncomment to test)". It will write on each 4KB of pages and your lazy allocation implementation should allocate all the requested memory. Partial sample output below:

```
...
$ allocating 0x000000000000001F mebibytes
malloc returned 0x0000000000203030
Touched all pages in 31 MiB
$ free -m
Free memory: 94 MB
$ freeing 0x000000000000001F mebibytes
free -m
Free memory: 94 MB
$ allocating 0x0000000000000029 mebibytes
malloc returned 0x0000000002103040
Touched all pages in 41 MiB
freeing 0x0000000000000029 mebibytes
free -m
Free memory: 53 MB
...
```

3. Uncomment the piece of code after "CASE 3: Touch some pages only (uncomment to test)". This will touch a in every 16 pages. In the output you will see amount of free memory reduces a lot slowly compared to test case 2. Sample partial output is below:

```
...
$ allocating 0x0000000000000047 mebibytes
malloc returned 0x0000000000303040
Touched ~1/16 of pages in 71 MiB
$ free -m
Free memory: 122 MB
$ freeing 0x0000000000000047 mebibytes
$ allocating 0x0000000000000051 mebibytes
malloc returned 0x0000000004A03050
Touched ~1/16 of pages in 81 MiB
$ free -m
```

```
Free memory: 117 MB
$ freeing 0x0000000000000051 mebibytes
free -m
Free memory: 117 MB
...
```

In the above listing, you can see even if we malloc larger amount of memory with lazy allocation and touching only few pages, it consumed only 5 MB of memory.

Extra Credit Task 6: Enable use of the entire virtual address space. (15 pts) The current xv6 user library code seems to assume that heap size and the size of malloc requests will not be larger than a 32-bit value. Modify the code as needed to allow the heap to grow until the virtual memory space is exhausted and to allow the size of a malloc request to be as large as the remaining contiguous available virtual memory space.

Extra Credit Task 7: Allow a process to turn memory overcommitment on and off. (15 pts) Instead of running everything with lazy allocation and possible memory overcommitment, make the default be not to overcommit and add a system call that will let a process turn memory overcommitment on and off. For example, a program could turn overcommit on to allocate a large array that is expected to be used sparsely, and then turn overcommit back off.

Turn-in procedure and Grading

Please use the accompanying template for your report. Convert your report to PDF format and push your hw4 branch with your code and report to your xv6 GitHub repository by the due date. Also, turn in the assignment on Teams with the URL of your GitHub repo by the due date. Give access to your repo to the TA.

This assignment is worth 75 points. The breakdown of points is given in the task descriptions above. The points for each task will be evaluated based on the correctness of your code, proper coding style and adequate comments, and the section of your report for that task.

You may discuss the assignment with other students, but do not share your code. Your code and lab report must be your own original work. Any resources you use should be credited in your report. If we suspect that code has been copied from an online website or GitHub repo, from a book, or from another student, or generated using AI, we will turn the matter over to the Office of Student Conduct for adjudication.