EEE3535-02 **Fall 2024**

Assignment 5: Paging

Due: Sunday, Dec. 1, 2024, 11:59PM

1 Introduction

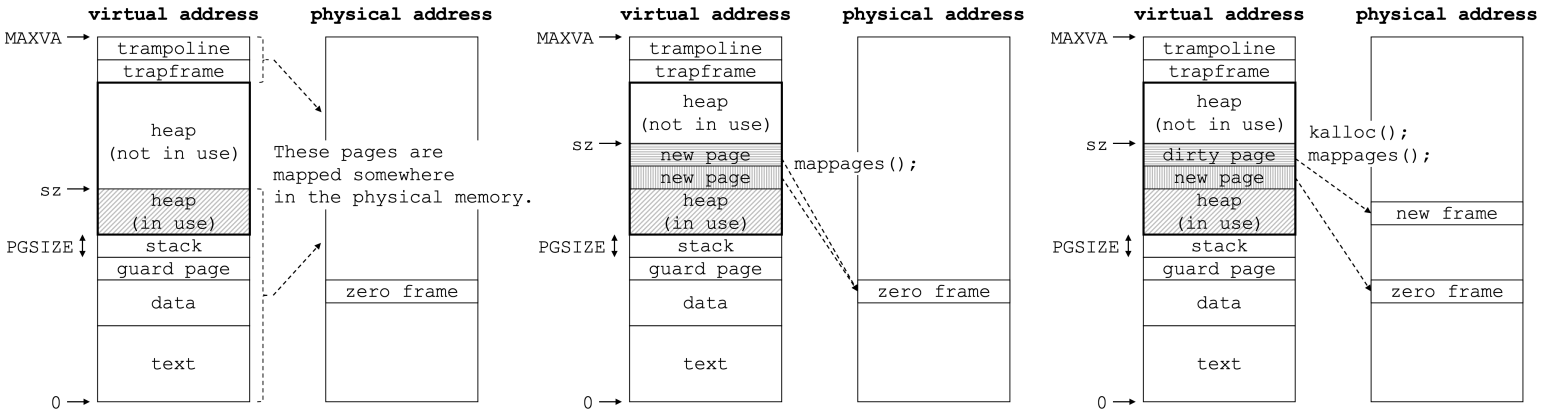
- The objective of this assignment is to modify the paging scheme of xv6-riscv to allocate physical frames lazily upon their first write access, known as *copy-on-write*.
- This assignment will implement the copy-on-write policy for the heap segment pages of user processes. Pages in other memory segments are allocated as usual.
- Under the copy-on-write scheme, physical frames are not allocated when `malloc()` is called. Instead, they are allocated when the memory is accessed for the first time.
- The copy-on-write policy can help save physical memory space, as it avoids allocating pages that are never actually used. Pages that are not accessed will not be stored in the physical memory at all.

1.1 `malloc()` and `sbrk()`

- The `malloc()` function searches the free list of a process, starting from `freep`, to find a free block that is greater than or equal to the size of a requested memory allocation.
- If it finds one in the free list, it does not explicitly allocate new memory. Instead, one of the existing free blocks is used (or split) to place the memory chunk.
- If it does not find an appropriate free block in the free list, it calls `sbrk()` via `morecore()` to add a free block to the free list. This is when new pages are allocated.
- The `sbrk()` syscall executes `sys_sbrk()` in `kernel/sysproc.c`. It retrieves the function argument (i.e., memory allocation size) using `argint()` and passes it to `growproc()`.
- The syscall returns the previous value of the process's virtual memory size (i.e., `myproc()->sz`), which corresponds to the starting address of the new memory space.
- Refer to Figure 1a for the visual representation of what `sz` means. It points to the top of the heap segment of a process, equivalent to the current size of the virtual memory space in use, excluding `trampoline` and `trapframe` at the top of the virtual address space (i.e., `MAXVA`).
- Noticeably, xv6-riscv places the user stack of a process below the heap, which is limited to 4KB (i.e., `PGSIZE`). The rest of the virtual address space is used for the heap.
- If the input argument of `growproc()` in `kernel/proc.c` is a positive number, it expands the heap segment via `uvmalloc()`. In the opposite case, the heap shrinks via `uvmdealloc()`.

1.2 Page Allocation by `uvmalloc()`

- The first argument of `uvmalloc()` in `kernel/vm.c` is a pointer to the root page table (or page directory) of a process. The second and third arguments represent the old and new values of `sz`, respectively.
- The `uvmalloc()` function rounds up the `sz` value to the nearest multiple of `PGSIZE` for page-granular allocation. It calls `kalloc()` in a `for` loop to grab as many pages as needed.
- The physical address of a new frame is mapped in the page table using `mappages()` in `kernel/vm.c` to record the virtual-to-physical address translation.
- The new page is set with `PTE_W`, `PTE_X`, `PTE_R`, and `PTE_U` flags, meaning that the page is writeable, executable, readable, and accessible in user mode, respectively.



(a) Initial state (b) Two new pages mapped to the zero frame (c) Dirty page remapped to a new frame

Figure 1: (a) In the initial state, assume that all pages in use are mapped to some frames in the physical memory. The *zero frame* is a zero-stripped, read-only dummy frame that will be universally shared across all pages and all processes. (b) Suppose that calling `malloc()` allocates two new pages. `sz` is accordingly shifted up. These new pages are not allocated in the physical memory but mapped to the zero frame. (c) If one of the unallocated pages becomes dirty (i.e., written), it can no longer be mapped to the read-only zero frame. The copy-on-write scheme belatedly allocates a new physical frame and remaps the dirty page to the new frame.

2 Implementation

- To start the assignment, go to the `xv6-riscv/` directory, download `paging.sh`, and execute the script. This assignment has no new user programs or syscalls.

```
$ cd xv6-riscv/
$ wget https://casl.yonsei.ac.kr/teaching/eee3535/paging.sh
$ chmod +x paging.sh
$ ./paging.sh
```

2.1 Skipping Page Allocation

- To skip page allocation at `malloc()`, one of the functions in the call chain (i.e., `sys_sbrk()`, `growproc()`, or `uvmalloc()`) must be modified.
- `uvmalloc()` is called at various places in the xv6-riscv kernel, so you may not want to touch this function.
- Modifying `sys_sbrk()` requires differentiating the case of expanding or reducing the heap segment. It is fine to change this function, but there is a better implementation for copy-on-write.
- The `growproc()` function already contains `if-else` statements for the growing (`n > 0`) and shrinking (`n < 0`) cases. For the growing case, it needs a new function instead of `uvmalloc()`, which increases `sz` but does not allocate new physical frames to virtual pages.

2.2 The Zero Frame

- If pages are not allocated at `sbrk()`, access to unallocated pages causes page faults.
- In normal situations, page faults indicate that the corresponding page table entries (PTEs) are invalid or have no right `PTE_R/X/W` permissions.
- However, the copy-on-write policy makes page faults ambiguous. They may be due to skipped page allocation or illegal memory access.
- To avoid such ambiguity, we will reserve one physical frame, called the *zero frame*.

Table 1: Encoding of the `scause` register in RISC-V

`scause` value	Description	`scause` value	Description
0	Instruction address misaligned	7	Store access fault
1	Instruction access fault	8	System call
2	Illegal instruction	12	Instruction page fault
3	Break point	13	Load page fault
5	Load access fault	**15**	**Store page fault**
6	AMO address misaligned	Others	Reserved

- The zero frame is created only once and for all, not per process. It is stripped with all zeros (i.e., all bits set to zero) and globally shared across all pages and all processes.
- A good place to create the zero frame is when the xv6-riscv kernel is initialized after paging is enabled, such as `procinit()` in `kernel/proc.c`. Once created, the zero frame should never be deallocated, such as in `uvmunmap()` in `kernel/vm.c`.
- When `sbrk()` skips page allocation, make all unallocated pages should map to the zero frame. The page table of a process must be accordingly updated to have the corresponding PTEs point to the zero frame, as shown in Figure 1b.
- The PTEs mapped to the zero frame must be set only with the `PTE_R` and `PTE_U` flags, meaning that they are accessible in user mode and read-only.
- On a page fault, checking the physical address (or physical frame number) mapped to the PTE can determine whether the page fault is due to skipped page allocation or illegal memory access.
- If the faulty physical address is the same as the zero frame, it is a valid memory access but faulty because of copy-on-write. Otherwise, illegal access should kill the process.

2.3 Copy-on-Write Page Allocation

- The zero frame is readable. If a process has only read access without writes to the zero frame, the read-only access does not cause page faults.
- A page fault occurs on write access to the zero frame. The store page fault means that the faulty page needs a new frame in the physical memory and can no longer be associated with the read-only zero frame.
- The dirty page causing the page fault must be remapped to a new physical frame. It requires calling `kalloc()` to grab a free frame and `mappages()` to update the virtual-to-physical address translation for the faulty page, as shown in Figure 1c.
- You will need to modify `mappages()` to bypass the `panic("remap")` error when a virtual page tied to the zero frame is remapped. Other remapping cases are illegal and should not be allowed.

2.4 Page Faults

- When a process is interrupted for any reason (e.g., timer interrupt, syscall, exception), the `trampoline` routine is called by default in xv6-riscv.
- It saves the context of the interrupted process and jumps to `usertrap()` in `kernel/trap.c`.
- The `usertrap()` function reads the `scause` register to determine the cause of the trap. Table 1 shows how `scause` is encoded in RISC-V. This register can be read via `r_scause()` in xv6-riscv.
- The only change you need in `usertrap()` is to add the case for a store page fault, which should allocate a new physical frame and remap the faulty page to the new frame.

- On the store page fault, reading the `stval` register indicates the faulty virtual memory address, which should be used for the page table walk. This register can be read using `r_stval()` in xv6-riscv.

3 Validation

- Your assignment will be graded based on the following test results.

1. **Skipping page allocation:**
   - The `sbrkfail` test in the `usertests` program attempts to allocate a large number of pages until it runs out of physical frames.
   - This test aims to make the `sbrk()` syscall fail by exhausting all the physical frames, but the copy-on-write policy will make the syscall succeed by mapping all heap pages to the zero frame.
   - If this test succeeds, it is regarded as a failure for `sbrkfail`.

```
$ usertests sbrkfail nofault
usertests starting
test sbrkfail: sbrkfail: allocate a lot of memory succeeded 0
FAILED
SOME TESTS FAILED
```

2. **Copy-on-write:**
   - Rerun the `sbrkfail` test without the `nofault` option.
   - This test generates write access to unallocated pages so that they are remapped to new physical frames.
   - Eventually, the test runs out of free frames, failing the `sbrk()` syscall. This is considered a pass for `sbrkfail`.

```
$ usertests sbrkfail
usertests starting
test sbrkfail: OK
ALL TESTS PASSED
```

3. **`Usertests`:**
   - If you passed the previous two tests, your copy-on-write implementation is likely to be working.
   - As a final check, run the `usertests` program to verify that it ends with `ALL TESTS PASSED`.

```
$ usertests
usertests starting
test copyin: OK
test copyout: OK

...

test outofinodes: ialloc: no inodes
OK
ALL TESTS PASSED
```

4 Submission

- In the `xv6-riscv/` directory, run the `tar.sh` script to create a tar file named after your student ID (e.g., `2024143535`).

```
$ ./tar.sh
$ ls
2024143535.tar  kernel  LICENSE  Makefile  mkfs  README  tar.sh  user
```

- Upload this tar file (e.g., `2024143535.tar`) on LearnUs. Do not rename the file.

5 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add extra rules without notice for the fair evaluation of students' efforts.

  -5 points: The tar file includes extra tags such as a student name, `hw5`, etc.

  -5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

  -5 points: Do not print unasked debugging messages.

  -10 points each: The validation section has three test cases. Each failed test will lose 10 points. Test #2 and #3 can earn points only if the preceding tests pass.

  -30 points: No or late submission.

  Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get "F" for the final grade.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If no agreement is reached between you and the TA, escalate the case to the instructor to review your assignment. Refer to the course website for the contact information of the TA and the instructor: `https://casl.yonsei.ac.kr/eee3535`

- Arguing for partial credits for no valid reasons will be considered as a cheating attempt and lose the assignment score.