Machine Problem 2 - Demand Paging and Swapping

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	March 28
Due Date:	April 10, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Tue. 13:00-14:00 before due date, CSIE Building B04 (Please knock the door)

Contents

1 Summary

The *virtual memory* is an isolated and abstracted memory space for each process. Only a portion of virtual memory pages are mapped to physical memory through per-process *page table*. With proper memory management, the operating system can maintain processes with large virtual memory spaces but with small physical memory in use.

In order to serve processes with distinct memory access patterns and distinct *working sets*, the *demand paging* technique comes into play. Each process can claim a large amount of virtual memory by `sbrk()` syscall without allocating actual physical pages. The physical memory is allocated on demand only when the virtual pages are accessed. It works by trapping *page fault* events. A page fault occurs when the accessed virtual address does not have a corresponding physical page.

Swapping is a technique to store memory pages on a disk. With this technique, the virtual memory pages are not only mapped to physical memory pages, but they can be mapped to blocks on a disk. The operating system can either *swap out* "cold" memory pages to disk blocks, or *swap in* disk blocks to physical memory when needed.

This homework will add demand paging and swapping to existing page table on `xv6`. This first step is to add the `vmprint()` syscall to show the details of page table. Then, the default behavior of `sbrk()` syscall will be changed to claim virtual memory without physical memory allocation. The page fault handler will be added to `usertrap()` to allocate physical pages on demand. The last step is to implement `madvise()` syscall to allow the calling process to swap in or swap out certain virtual memory address, and change the page table data structure to support swapping.

2 Launching Docker

It follows the same procedure in MP0. If you're using Windows, it strongly suggested to install WSL2 and run Docker in WSL2.

3 Launching xv6

3.1 Launch the Docker Image for MP2

1. Download the `MP2.zip` from NTUCOOL, unzip it, and enter it.

```
$ unzip MP2.zip
$ cd mp2
```

2. Pull Docker image from Docker Hub

```
$ docker pull ntuos/mp2
```

3. In the `mp2` directory, run `docker run` to enter to the shell in the container.

```
$ docker run -it -v $PWD:/root ntuos/mp2
```

3.2 Run the Preliminary Judge

The zip provides a `judge` program to perform 4 public tests, respectively named `mp2_N` where $N = 1, \ldots, 4$. The source code can be found at `user/mp2_N.c`. The commands below runs all tests at once and produces a report in the same directory.

```
$ make STUDENT_ID=d10922013 zip  # set your ID here
$ ./judge d10922013.zip
```

3.3 Run Individual Tests

To run one of `mp2_N` individually, run `make qemu` to enter the shell and run the `mp2_N` command, where $N = 1, \ldots, 4$

```
$ make clean
$ make qemu
...

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mp2_1
```

In the case that the test program hangs, start a new shell and run:

```
killall qemu-system-riscv64
```

4 Scoring

The judge program for public tests is shipped with `MP2.zip`. The private test is disclosed after the deadline.

- Public program tests (70%)
- Private program tests (15%)
- Required report (15%)

5 Assignment

It is recommended to read materials in Section 7 before writing your code. It will save your time. Also, do your homework early. :)

5.1 Preliminary

5.1.1 Print a Page Table (Public Test 5% + Report 5%)

Most of the operating systems implement a separate page table for each process. If a process occupies its page table with the size analogous to the size of its virtual memory, the amount of memory occupied by the page tables can be huge, and is unacceptable as main memory is a scarce resource.
Hence, modern operating systems incorporate with *multilevel* page tables. It has a higher level page table, where each entry points to a lower level page table. Page tables of each level are structured similarly except that the lowest page tables point to actual pages. Lower level page tables are allocated only when needed. The RISC-V `xv6` page table has 3 levels. In this section, we are going to implement `vmprint()` to dump the page table tree.

Program Part: (5%) Implement the `vmprint()` function in `kernel/vm.c`. It takes a page table of type `pagetable_t` and print that page table in the format below.

```
*** Now run 'gdb' in another window.

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mp2_1
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
```

```
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
$ qemu-system-riscv64: terminating on signal 15 from pid 45662 (make)
```

In the example above, the top-level page table has mappings in entries at indexes 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.
The printed tree structure satisfies the rules.

1. The first line shows the address passed to `vmprint`.
2. Print the page table entries (PTEs) since the second line. One line per entry.
3. The entries are indented to show the tree structure of the page table, four spaces per level. Use the `+-` prefix to indicate an entry.

Each page table entry (PTE) contains the following items.

1. It has an index in its page-table page for that entry. e.g. `+- 255:` is the 255th entry in that tree level.
2. `pte=` writes the physical address of the entry.
3. `va=` writes the virtual memory address recorded on the entry.
4. `pa=` writes the physical memory address recorded on the entry.
5. It has flag bits `V, R, W, X, U` for the entry. Note that PTEs without `PTE_V` bit are not printed.
6. Each entry ends with a line break ('\n') and has no trailing spaces.

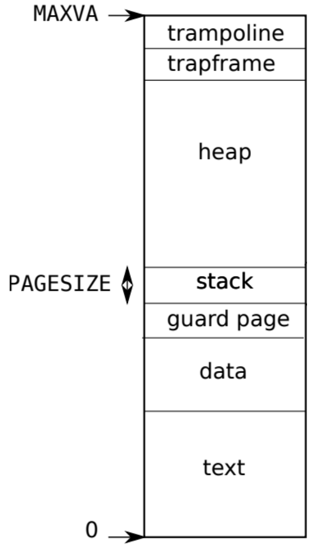
Hints:

1. Your code might emit different physical addresses than those shown above. The number of entries and the virtual addresses should be the same.
2. You may check the source files.
   - `kernel/memlayout.h` defines the memory layout.
   - `kernel/vm.c` contains most virtual memory related code.
3. Use the macros at the end of the file `kernel/riscv.h`.
4. Use `%p` in `printf` calls to print out full 64-bit hex addresses.

Report Part: (5%) Please answer the following questions:

1. Explain how `pte`, `pa` and `va` values are obtained in detail. Write down the calculation formula for `va`.
2. Write down the correspondences from the page table entries printed by `mp2_1` to the memory sections in Figure 1. Explain the rationale of the correspondence. Please take virtual addresses and flags into consideration.
3. Make a comparison between the *inverted page table* in textbook and *multilevel page table* in the following aspects:
   (a) Memory space usage
   (b) Lookup time / efficiency of the implementation.

Figure 1: The virtual memory layout for xv6



5.1.2 Generate a Page Fault (Public Test 20%)

Lazy allocation of user space heap memory is a neat trick on page table. `xv6` applications ask the kernel for heap memory using the `sbrk()` system call, which is implemented in the function `sys_sbrk()` in `kernel/sysproc.c`. In the original `xv6` kernel, `sbrk()` always allocates physical pages and maps them to the process virtual address space. It can take a long time for to allocate the memory if the size is as large as 1GB (=262,144 pages). In addition, programs can allocate more memory than the amount they actually use or allocate far before in advance.

To remedy this issue, `sbrk()` can be changed to allocate user memory lazily. That is, `sbrk()` doesn't allocate actual physical memory at the beginning, but simply increases the heap size. When the process first access the lazily-allocated page, it triggers a page fault, which is handled by the kernel to allocate physical memory. In this section, you will change the default behavior of `on sbrk()` to prepare for lazy allocation feature.

Program Part (I): Change the behavior of the sbrk() (6%) Modify the (`sys_sbrk()` function in `kernel/sysproc.c`). Remove the physical page allocation part. Instead, it modifies `myproc()->sz` to grow the process memory size by `n` bytes without actual allocation, and return the old process memory size.

Also, `proc_freepagetable()` in `proc.c` and `uvmunmap()` in `vm.c` assumes the orginal behavior of `sbrk()`. Please modify them so that they don't fail on unallocated pages.

Program Part (II): Allocate physical space in Page Fault Handler (7%) To enable lazy page allocation, the page fault handler must be modified to find the virtual address triggering page fault in the process, and allocate a physical memory page for that offending virtual address. The page fault will be captured by `usertrap()` in `kernel/trap.c`. Change the function to handle page fault events. The modification goes as follows.

- Use `r_scause() == 13 || r_scause() == 15` condition to catch page fault events in `usertrap()` in `kernel/trap.c`. Create a page fault handling function `hanle_page_fault()` in `/kernel/paging.c` and call that function whenever a page fault occurs.
- Always mark `PTE_U, PTE_R, PTE_W, PTE_X` flags on newly allocated pages.
- In `hanle_page_fault()`, call `uint64 va = r_stval()` to find the offending virtual address, and round the address to page boundary using `PGROUNDDOWN()`.
- Make use of the functions below to allocate a physical page for the offending virtual address.
  - The `walk()` in `kernel/vm.c` traverses entries in a page table.
  - The `mappages()` in `/kernel/vm.c` is used to assign a physical to virual address mapping in the page table.
  - The `memset()` in `/kernel/string.c` zeros out a fraction of memory.

Program Part (III) : Free physical space in Page Fault Handler (7%) Modify `sys_sbrk(n)` to free the physical pages when `n` is negative and decreases `myproc()->sz`.

Run the Test for Program Part I, II and III The test program `mp2_2` performs the actions below to the check the correctness of `sbrk()` and `handle_pgfault()` functions. It always triggers page fault on valid memory addresses.

1. Call `sbrk(PGSIZE * 2)` before `vmprint()` to check your original page table.
2. Incur a page fault before calling `vmprint()` to check if your page table have correctly produce entries.
3. Call `sbrk(-PGSIZE * 2)` before `vmprint()` to check if your page table have correctly reduce entries.

The `mp2_2` output looks like this.

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mp2_2
# Before sbrk(PGSIZE * 2)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After sbrk(PGSIZE * 2)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After sbrk(-PGSIZE * 2)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After sbrk(PGSIZE * 2) again
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
```

```
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After page fault at 0x0000000000004000
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 pa=0x0000000087f58000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After sbrk(-PGSIZE * 2) again
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
$ qemu-system-riscv64: terminating on signal 15 from pid 46166 (make)
```

5.2 Demand Paging and Swapping (Public Test 45% + Private Test 15% + Report 10%)

Demand paging with swapping is a technique to store memory pages in a secondary storage, which is usually a hard disk. The pages are brought to main memory when they are needed by a process. This technique is also known as a lazy swapper. In this section, we are going to implement the new `madvise()` syscall. It allows the users to hint that a sequence of pages, namely a *memory region*, should be *swapped out* to the disk or be *swapped in* to the physical memory. The page fault handler will be modified to handle swapped pages on the disk. In the report, briefly explain how you implement demand paging and swapping in your code.

Program Part (I): Swap Out Pages (17%) The user passes a virtual memory address range to `madvise()` syscall to hint how a memory region is expected to use to the kernel. The memory region is the minimum set of pages covering the address range from user. The kernel can choose an appropriate paging policy and caching techniques according to the user's advice. The function signature is defined as follows.

int madvise(void *addr, size_t length, int advice);

The behavior of `madvise()` obeys the following rules.

- The `addr` and `length` specifies a range of memory address $[addr, addr+length)$. The byte at $addr+length$ is not included. The corresponding *memory region* is the minimum set of pages covering the address range.
- If a portion of the memory region exceeds the process memory size (`myproc()->sz`), it returns -1. Otherwise, it performs appropriate actions and returns 0.
- The `advice` option describes how the region is expected to be used, which can be one of the three values.

– MADV_NORMAL : No special treatment. Nothing to be done.
– MADV_WILLNEED: Expect an access in the near future. It swaps in the pages on the disk to the memory, and allocates new physical memory pages for unallocated pages.
– MADV_DONTNEED: Do not expect any access in the near future. Any pages within the region that are still in the physical memory will be swapped out to the disk.

To implement this feature, use balloc_page() and write_page_to_disk() to allocate a page on the storage device. The steps go as follows. More utility functions can be found at Section 7.

- Extend the the vmprint() function to show the PTE_S bit if a page is swapped. Show block numbers for swapped page entries.
- In the madvise() in /kernel/vm.c, handle the MADV_NORMAL option. It does nothing but simply check the given memory region is valid or not.
- Handle the MADV_DONTNEED option. Those pages still in physical memory within the region are moved to the disk. It sets the PTE_S bit and cancels the PTE_V bit on PTEs for those pages.

The test program mp2_3 will run the following actions.

1. Increase the process memory by sbrk(PGSIZE * 3).
2. Call madvise() with MADV_NORMAL and different address ranges, and check if returns 0 or -1 according to the addresses. (5%)
3. Trigger a page fault to allocate a physical memory and call vmprint() to check the page table. (4%)
4. Call madvise() with MADV_DONTNEED values option to swap a memory region into the disk, and call vmprint() to check the page table. It should indicate the swapped page with swapped bit "S". (8%)

The mp2_3 program output example:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mp2_3
# Before madvise()
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 pa=0x0000000087f58000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After madvise()
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 blockno=0x00000000000002f8 R W X U S
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
```

```
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
$ qemu-system-riscv64: terminating on signal 15 from pid 46206 (make)
```

Program Part (II): Swap In Pages (28%) Handle the `MADV_WILLNEED` option in `madvise()` syscall. It ensures the pages within the memory region to be physically allocated. The `PTE_V` bit is set on those affected page table entries.
The testing program `mp2_4` goes through the following steps:

1. Trigger a page fault to allocate a physical memory.
2. Call `vmprint()` to check current page table.
3. Call `madvise()` with `MADV_DONTNEED` option to swap the correspond pages to the disk, and call `vmprint()` to check current page table. (13%)
4. Call `madvise()` with `MADV_WILLNEED` option to place the correspond the physical memory from the disk back to physical memory and allocate pages for those not allocated yet. Then, Call `vmprint()` to check the page table. (15%)

Notes The program panics when it is exiting and `uvmunmap()` tries to free swapped pages on the disk. This is a known issue and is not trivial to fix. You can ignore swapped pages in `uvmunmap()` to workaround it.
The test program `mp2_4` output looks like this.

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ mp2_4
# After page fault
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 pa=0x0000000087f58000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After madvise(DONTNEED)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 blockno=0x0000000000000300 R W X U S
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After madvise(WILLNEED)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
```

```
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 3: pte=0x0000000087f52018 va=0x0000000000003000 pa=0x0000000087f58000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 pa=0x0000000087f76000 V R W X U
|       +-- 5: pte=0x0000000087f52028 va=0x0000000000005000 pa=0x0000000087f75000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
$ qemu-system-riscv64: terminating on signal 15 from pid 46245 (make)
```

Program Part (III): Page Fault on Swapped Pages (Private Test 15%) Change the page fault handler `handle_pgfault()` to handle swapped pages. If a page fault is triggered on a virtual address which page was swapped in to the disk, move the swapped page back to the physical memory. The feature is tested in a private test, which goes through the following process.

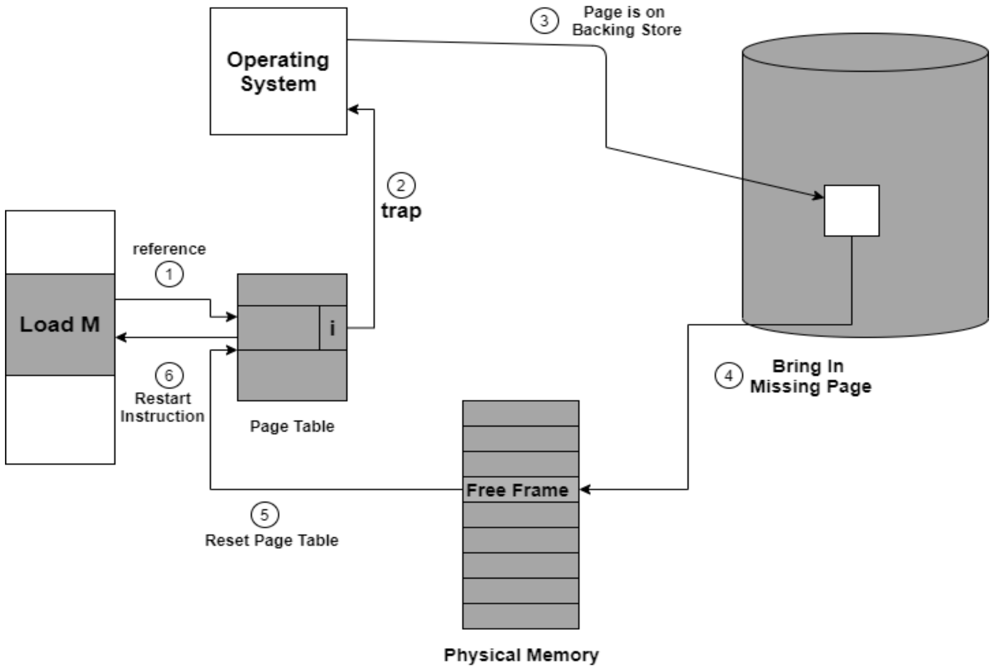
1. Trigger a page fault to allocate a physical memory and call `vmprint()` to check the page table.
2. Call `madvise()` with `MADV_DONTNEED` option to swap the pages to the disk and call `vmprint()` to check the page table.
3. Trigger page fault on a swapped page and call `vmprint()` to check if a new physical memory page is allocated. (8%)
4. Undisclosed additional tests with the combination of `madvise()` and page fault. (7%)

The output may look like this. The actual output can be different given that the actual private test is disclosed after deadline.

```

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ mp2_5
# After page fault
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 pa=0x0000000087f58000 V R W X U
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
        +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
        +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X

# After madvise(DONTNEED)
page table 0x0000000087f57000
+-- 0: pte=0x0000000087f57000 va=0x0000000000000000 pa=0x0000000087f53000 V
|   +-- 0: pte=0x0000000087f53000 va=0x0000000000000000 pa=0x0000000087f52000 V
|       +-- 0: pte=0x0000000087f52000 va=0x0000000000000000 pa=0x0000000087f54000 V R W X U
|       +-- 1: pte=0x0000000087f52008 va=0x0000000000001000 pa=0x0000000087f51000 V R W X
|       +-- 2: pte=0x0000000087f52010 va=0x0000000000002000 pa=0x0000000087f50000 V R W X U
|       +-- 4: pte=0x0000000087f52020 va=0x0000000000004000 blockno=0x0000000000000300 R W X U S
+-- 255: pte=0x0000000087f577f8 va=0x0000003fc0000000 pa=0x0000000087f56000 V
    +-- 511: pte=0x0000000087f56ff8 va=0x0000003fffe00000 pa=0x0000000087f55000 V
```

```
    +-- 510: pte=0x0000000087f55ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W
    +-- 511: pte=0x0000000087f55ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
```

Figure 2: The worlflow of demand paging. *Silberschatz, A., Galvin, P. B., & Gagne, G. Operating system concepts*



Report Part (10%): Figure 2 shows the workflow of demand paging in several steps. Please answer the following questions:

- In which steps the page table is changed? How are the addresses and flag bits modified in the page table?
- Describe the procedure of each step in plain English in Figure 2. Also, include the functions to be called in your implementation *if any*.

6 Submission

6.1 Report

Submit your report to *Machine Problem 2* on Gradescope course site (https://www.gradescope.com/courses/494146) in one PDF file. We have registered your Gradescope account with your university email address.

6.2 Source Code

Run this command to pack your code into a zip named in your lowercase student ID, for example, `d08922025.zip`. Submit this file to *Machine Problem 2* section on NTUCOOL.

```
make STUDENT_ID=d10922013 zip  # set your ID here
```

6.3 Folder Structure after Unzip

We will unzip your submitted file using `unzip` command. The unzipped folder structure should look like this. The common error is missing the top level directory `<student_id>` and missing a `xv6` directory.

```
<student\_id>
‘-- xv6
    |-- Makefile
    |-- conf
```

```
    |-- kernel
    |-- mkfs
    |-- user
    `-- ...
```

6.4 Grading Policy

- It is allowed to re-submit your code and report. The renaming due to resubmission will not result in point deduction. Only the latest submission is judged, even it's over the deadline.
- Compilation error leads to 0 points.
- Erroneous folder structure incurs 10 point penalty. Using uppercase in `<student_id>` is also treated as wrong folder structure.
- Late submission will incur immediate 20 points penaly. The score is deduced by 20 points per 24 hours later on. For example, 25-hour late submission leads to 40 points deduction.

7 Appendix

7.1 Macros and Builtin Types

7.1.1 Headers

To use the macros and builtin types on XV6 kernel code, the following headers must be included.

```
#include "types.h"
#include "param.h"
#include "riscv.h"
```

7.1.2 Naming Conventions

- PA - Physical address
- VA - Virtual address
- PG - Page
- PTE - Page table entry
- BLOCKNO - Block number on a device or a disk
- PGTBL - Page table

7.1.3 Page Alignment

The following macros convert a memory address to a page aligned value.

- `PGSIZE`
  The page size, which is 4096.
- `PGSHIFT`
  The number of offset bits in memory address, which is 12.
- `PGROUNDUP(sz)`
  Round the memory address to multiple of 4096 greater than or equal to `sz`.
- `PGROUNDDOWN(sz)`
  Round the memory address to multiple of 4096 less than or equal to `sz`.

7.1.4 Page table entry (PTE) Constants and Macros

A page table entry is a 64-bit integer, consisting of 10 low flag bits and remaining high address bits. The flag bits includes `PTE_V, PTE_R, PTE_W, PTE_X, PTE_U, PTE_S`.

- `PTE_V`
  If set, the high bits represent a valid memory address.
- `PTE_R`
  If set, the page at the address can be read.
- `PTE_W`
  If set, the page at the address can be written.
- `PTE_X`
  If set, the code on the page at the address can be executed.
- `PTE_U`
  If set, the page at the address is visible to userspace.
- `PTE_S`
  If set, the high bits represent the block number of a swapped page.

They can be used to check, set or unset flag bits on a page table entry.

```
pte_t *pte = walk(pagetable, va, 0);

/* Check if PTE_V bit is set */
if (*pte & PTE_V) { /* omit */ }

/* Set the PTE_V bit */
*pte |= PTE_V;

/* Unset the PTE_V bit */
*pte &= ~PTE_V;
```

The high bits must be a valid address if `PTE_V` bit is set. The following macros are used to convert a physical address to the high bits of PTE, and vice versa.

```
pte_t *pte = walk(pagetable, va, 0);

if (*pte & PTE_V) {
  /* Get the PA from a PTE */
  uint64 pa = PTE2PA(*pte);

  /* Create a PTE from a PA and flag bits */
  *pte = PA2PTE(pa) | PTE_FLAGS(*pte);
}
```

If a PTE points to a swapped page, the `PTE_S` bit is set but `PTE_V` should be eliminated. The high bits represents the block number on `ROOTDEV` device.

```
pte_t *pte = walk(pagetable, va, 0);

if (*pte & PTE_S) {
  /* Get the BLOCKNO from a PTE */
  uint64 blockno = PTE2BLOCKNO(*pte);

  char *pa = kalloc();  /* Assume pa != 0 */
  read_page_from_disk(ROOTDEV, pa, blockno);

  /* Create a PTE from a BLOCKNO and flag bits */
  *pte = BLOCKNO2PTE(pa) | PTE_FLAGS(*pte);
}
```

7.2 Functions Used in The Homework

The following functions can de/allocate pages on a storage device.

- `uint balloc_page(uint dev)`
  - Allocate a 4096-byte page on device `dev`.
  - Return the block number of the page.
- `uint bfree_page(uint dev, uint blockno)`
  - Deallocate the 4096-byte page at block number `blockno` on device `dev`.
  - The `blockno` must be returned from `balloc_page()`.

The following functions are used to load/save a memory page from/to a page on a storage device.

- `void write_page_to_disk(uint dev, char *page, uint blk)`
  - Write 4096 bytes from `page` to the page at block number `blk` on device `dev`.
  - The address `page` must be 4096-aligned and is returned from `kalloc()`.
  - The `blk` must be returned from `balloc_page()`
- `void read_page_from_disk(uint dev, char *page, uint blk)`
  - Read 4096 bytes from the page at block number `blk` on device `dev` to `page` .
  - The address `page` must be 4096-aligned and is returned from `kalloc()`.
  - The `blk` must be returned from `balloc_page()`

The following functions are related to the page table.

- `pte_t *walk(pagetable_t pagetable, uint64 va, int alloc)`
  - Look up the virtual address `va` in `pagetable`.
  - Return the pointer to the PTE if the entry exists, otherwise return zero.
  - If `alloc` is nonzero, it allocates page tables for each level for the given virtual address.
  - Note that it can return a non-null PTE pointer but without `PTE_V` bit set on the entry.
- `int mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)`
  - Map a virtual memory range of `size` bytes starting at virtual address `va` to the physical address `pa` on `pagetable`.
  - Return 0 if successful, otherwise nonzero.
  - The corresponding PTEs for the given virtual memory range must not set `PTE_V` flag.

The following functions are used to de/allocate physical memory.

- `void *kalloc()`
  - Allocate a 4096-byte physical memory page and return the address.
- `void kfree(void *pa)`
  - Deallocate the physical memory page at `pa`.
  - `pa` must be returned from `kalloc()`.

7.3 Functions to be Modified in the Homework

The following functions are potential candidates to be modified in this homework. You are free to roll out your own implementation.

- `kernel/vm.c`
  - `vmprint()`
  - `madvise()`
- `kernel/paging.c`
  - `handle_pgfault()`
- `kernel/sysproc.c`
  - `sys_sbrk()`
- `kernel/trap.c`
  - `usertrap()`
- `kernel/vm.c`
  - `walkaddr()`
  - `uvmunmap()`

7.4 Notes on Device I/O

When working calling device I/O functions, such as `balloc_page()` and `bfree_page()`, it must be encapsulated with `begin_op()` and `end_op()` to work properly.

```
begin_op();
read_page_from_disk(ROOTDEV, pa, blockno);
bfree_page(ROOTDEV, blockno);
end_op();
```

7.5 Troubleshooting

`panic: invalid file system` when starting xv6 Download the fresh zip source code from NTUCOOL. Copy the `fs.img` from the zip and overwrite the `fs.img` in your homework directory. Make sure your `write_page_to_disk()` writes to a valid `blockno`, and `balloc_page()`/`bfree_page()` are used in the right way.

The page fault is triggered on `kerneltrap()` It occurs when your kernel code accidentally reads or writes to to an address which page is not allocated yet.

remap panic in `mappages()` `mappages()` expects the received virtual address does not have `PTE_V` on the corresponding page table entry. It is usually caused by setting `PTE_V` on the entry before calling `mappages()`.

Test program panics when it exits The cause is that the kernel deallocates existing pages of the exiting process, while `uvmunmap()` is not implemented correctly. It mostly happens when `sbrk()` is changed in problem 2, but corresponding modification is not yet done in `uvmunmap()`.

If you free swapped pages in `uvmunmap()`, the process will panic when exits. The fix is not trivial. You can choose to ignore swapped pages in this case.

Panic in `kfree()` Make sure the address passed to `kfree()` is 4096-byte aligned and points to a page allocated by `kfree()`.

8 References

[1] POSIX — IEEE Standard Portable Operating System Interface for Computer Environments
https://ieeexplore.ieee.org/document/8684566](https://ieeexplore.ieee.org/document/8684566/

[2] Inverted Page Table — Computer Architecture: A Quantitative Approach
https://ict.iitk.ac.in/wp-content/uploads/CS422-Computer-Architecture-patterson-5th-edition.pdf

[3] xv6 — A simple, Unix-like teaching operating system by MIT
https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf/

[4] Linux Kernel - The Process Address Space — sathya's Blog
https://sites.google.com/site/knsathyawiki/example-page/chapter-15-the-process-address-space/

[5] Advanced Programming in the UNIX® Environment
https://www.oreilly.com/library/view/advanced-programming-in/9780321638014/