Machine Problem 2 - Demand Paging and Swapping

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100 + 10 Bonus
Release Date:	March 22
Due Date:	April 4, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Tue. & Thu. 13:00-14:00 before due date, CSIE Building R442 or 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 Launching Docker Image of MP2

1. Download the `MP2-rev-2.zip` from NTUCOOL, unzip it, and enter it.

   ```
   $ unzip MP2-rev-2.zip
   $ cd mp2
   ```

2. Pull Docker image from Docker Hub

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

   Note that the image only supports the following CPU architecture

   - `x86_64` or `amd64`
   - `arm64`

   You can check the architecture of your CPU by running the following command

   ```
   $ arch
   ```

   If you are not able to use the image we provided, please send an email to the TA and let us know the output of your `arch` command.

3. Use `docker run` to start the process in a container and allocate a TTY for the container process

   ```
   $ docker run -it -v $(pwd):/home/os_mp2/xv6 ntuos/mp2
   ```

4. Check the environment in the Docker container

   ```
   $ cat /etc/os-release
   ```

3.2 Running Test Programs

There are 4 public test programs, respectively named `mp2_N` with $N = 1, \ldots, 4$. Their source code can be found at `user/mp2_N.c`. Run the command to launch all tests at once, and the output will be saved to `mp2_N.out` files.

```
$ ./run_mp2.py
...
== Test mp2_1 == (1.3s)
== Test mp2_2 == (0.4s)
== Test mp2_3 == (1.0s)
== Test mp2_4 == (1.0s)
```

To run an individual test program instead, run `make qemu` to enter the `xv6` shell and run `mp2_N` command.

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

xv6 kernel is booting

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

4 Assignment

4.1 Scoring

The scoring is divided into the following parts.

- Public program tests (70%)
- Private program tests (15%)
- Required report (15%)
- Bonus report (10%)

The source code for public program tests are shipped with the `MP2-rev-2.zip`, while the private program tests are disclosed after the deadline.

(update 2) Please read appendix materials in Section 6 before writing your code. It will save your time.

4.2 Preliminary

4.2.1 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%) Define a function called ‘vmprint()‘. It takes a ‘pagetable_t‘ argument and print that page table in format described 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 11618 (make)
```

Note:

1. The first line displays the argument to ‘vmprint‘.
2. After that, there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree.
3. Each PTE line is indented by a number of ‘" ├"‘ and ‘" └‘ that indicates its depth in the tree. (It's ‘" ├"‘, not ‘" |−"‘. Otherwise, you may not pass the judge.
4. Each PTE line shows:
   (a) The PTE index in its page-table page
   (b) The PTE flag bits `V, R, W, X, U`.
   (c) The physical address extracted from the PTE.
   (d) Don't print PTEs without `PTE_V` bits.
5. (update 2) The provided output examples contain non-ASCII space characters (0xC2 0xA0). You can use ASCII space character instead (0x20). The judge will tolerate the difference.
6. (update 2) Make sure each line ends with a line break ('\n') and has no trailing spaces.

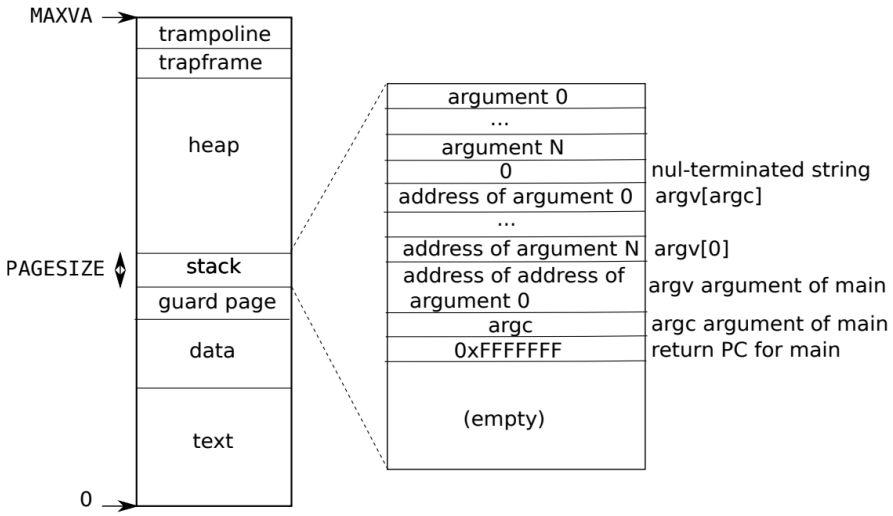
Hint:

1. Your code might emit different physical addresses than those shown above. But the number of entries and the virtual addresses should be the same.
2. 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.
3. Read "Chapter 3" of the [xv6 book], and related files:
   - ‘kernel/memlayout.h‘, which captures the layout of memory.
   - ‘kernel/vm.c‘, which contains most virtual memory code.
4. Use the macros at the end of the file ‘kernel/riscv.h‘.
5. Use ‘%p‘ in your ‘printf‘ calls to print out full 64-bit hex PTEs and addresses.

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

1. (update 2) Explain how pte, pa values are obtained in detail, and the calculation of va in the `vmprint()` output. Literally how they are done in your code.
2. (update 2) Write down the correspondences from the page table entries printed by `mp2_1` to the memory layout in Figure 1. Explain the rationale of the correspondence. You may take virtual addresses and flags into consideration.
3. Make a comparison between inverted page table in textbook and multilevel page table in these aspects:
   (a) Memory space usage
   (b) (update 2) Lookup time / efficiency of the implementation.

Figure 1: The virtual memory layout



4.2.2 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()‘` allocates physical memory and maps it into the process's virtual address space. It can take a long time for to allocate the memory if the size is large. For example, consider a gigabyte consisting of 262,144 pages. That's a huge number to allocate. In addition, programs can allocate more memory than the amount they actually use (e.g., to implement sparse arrays), 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 physical memory, but simply increases the heap size. When the process first tries to use a page of lazily-allocated memory, it triggers a page fault, which is handled by the kernel to allocate physical memory, zeroing it, and mapping it to the corresponding virtual address. In this section, you will need to prepare for lazy allocation feature on ‘sbrk()‘, eliminate allocation from ‘sbrk()‘.

Program Part (I): Change the behavior of the sbrk() (6%) In the original `xv6` kernel, `‘sbrk()‘` allocates physical memory and maps it into the process's virtual address space. Please modify the `sbrk()` in order not to allocate physical memory. The testing program will investigate the page table using `vmprint()` implemented earlier after the `‘sbrk()‘` call, and check that the page table is not touched.

Hint:

- Delete page allocation from the `‘sbrk(n)‘` system call implementation (`‘sys_sbrk()‘` in `‘kernel/sysproc.c‘`). The `‘sbrk(n)‘` system call modifies `myproc()->sz` to grow the process memory size by `‘n‘` bytes, and then returns the old process memory size.

- Read "Section 4.6" of the [xv6 book] and related file `‘kernel/trap.c‘`.
- You may delete or change the call to `‘growproc()‘`.
- (update) You may change `proc_freepagetable()` in `proc.c` and `uvmunmap()` in `vm.c` so that they don't fail on unallocated pages.

Program Part (II): Allocate physical space in Page Fault Handler (7%) To implement the lazy page allocation, please implement the page fault handler function. The page fault handler must finish the following works:

1. Find the virtual address which cause the page fault.
2. Allocate a physical memory page for that virtual address.

Hint:

- Modify `usertrap()` in "/kernel/trap.c" to handle page fault. You should create the page fault handler by yourself in "/kernel/paging.c" to handle the page fault.
- `walk()` in "/kernel/vm.c", `mappages()` in "/kernel/vm.c", and `memset()` in "/kernel/string.c" are your good friends.Hint
- (update) Newly allocated pages always have `PTE_U, PTE_R, PTE_W, PTE_X` flags.
- (update) The testing program always triggers page fault on legal memory addresses.
- (update) Use `r_scause() == 13 || r_scause() == 15` to catch page fault events in `usertrap()`.

Program Part (III) : Free physical space in Page Fault Handler (7%) Please modify the `sbrk(n)` system call to free the physical memory when `n` is negative.

Running the Test: The testing program `mp2_2` runs the following actions to the check if `sbrk()` and `handle_pgfault()` functions work properly.

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.

Testing program output example:

```
xv6 kernel is booting

hart 1 starting
hart 2 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 12110 (make)
```

4.3 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, usually in a hard disk, and the pages are brought to main memory when it is needed by the process. This is also known as a lazy swapper. Those pages that are never accessed are never loaded into the physical memory.

In this section, we are going to implement the `madvise()` syscall to let users to suggest 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 load swapped disk in the disk. Besides the programming part, you also need to briefly explain how you implement demand paging and swapping in your report.

Program Part (I): Swap Out Pages (17%) The madvise() system call tells how a memory region is expected to use to the kernel. The kernel can choose appropriate paging policy and caching techniques from that advice. This is the function signature:

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

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

- The `addr` and `length` describes the range of the memory region.
- If a portion of the memory region exceeds the process memory size (`myproc()->sz`), it returns -1.
- The `advice` is an option how the region is expected to use. It accepts the following values:
  - `MADV_NORMAL` : No special treatment. Nothing to be done.
  - (update 2) `MADV_WILLNEED`: Expect access in the near future. It allocates the physical memory pages for the affected memory region, including swapped pages and unallocated pages.
  - (update 2) `MADV_DONTNEED`: Do not expect access in the near future. Any page is still in physical memory within the memory region will be moved to the swap space.
- If the operation succeeds, return 0.

The following works should be done in this part:

- Extend the the `vmprint()` function to show the `PTE_S` bit and the block number of each page table entry (PTE), if the entry points to a swapped page.
- Implement the `MADV_NORMAL` option in `madvise()` in "/kernel/vm.c". It should check the given memory region is valid or not.
- Implement 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 affected page table entries.

The testing 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 memory regions, and check if returns 0 or -1 accordingly. (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=0x00000000000002e0 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 12141 (make)
```

Program Part (II): Swap In Pages (28%) Add the `MADV_WILLNEED` option to the `madvise()` syscall. It causes every page within memory region are allocated in physical memory. The `PTE_V` bit is set on 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. (update 2) ~~Again trigger a page fault to allocate a physical memory.~~ No page fault will triggered on any swapped page in this test.
5. (update 2) 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: (update 2) 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 to workaround if you encounter this issue.
The testing program `mp2_4` output example (changed):

```
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=0x00000000000002e8 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 12173 (make)
```

Program Part (III): Page Fault on Swapped Pages (Private Test 15%) Change the page fault handler to handle swapped pages. Allocates physical memory pages for the swapped pages and let the corresponding virtual memory pages to be mapped to the physical memory pages.

The private testing program runs through the following steps.

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. (update 2) Undisclosed additional tests with the combination of `madvise()` and page fault. (7%)

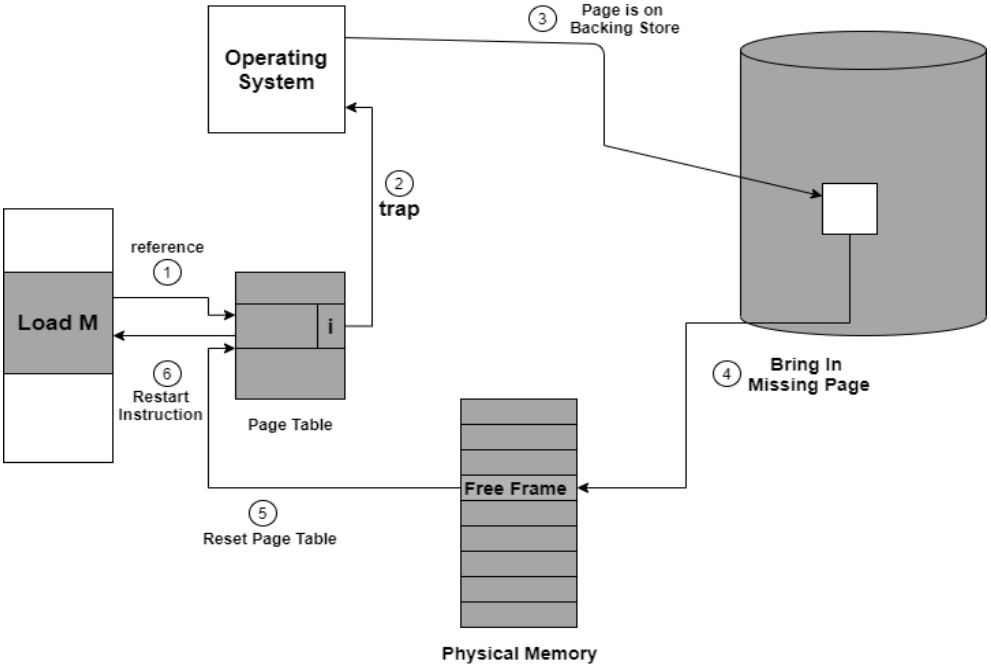
(update 2) The testing program output may look like this. The actual private test output can be different. The page fault is always triggered on valid heap addresses. You can choose to kill the process, panic or ignore the error on invalid virtual addresses.

```
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=0x00000000000002e8 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 page fault 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
│       └── 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
$ qemu-system-riscv64: terminating on signal 15 from pid 12204 (make)
```

Report Part (10%): (update 2) 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?

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



- Describe the procedure of each step in plain English in Figure 2. Also, include the functions to be called in your implementation *if any*.

4.4 Bonus Reports

4.4.1 Pros and Cons of Demand Paging (Bonus +5%):

Explain the benefits and drawbacks of "demand paging" in following aspects.

- Memory Utilization.
- External fragmentation.
- Input/Output for demand paging.
- The size of physical memory.
- Share the pages with demand paging.
- (update 2) Overheads due to interrupts, page tables access, and memory access time in demand paging. (disadvantage only)

4.4.2 Effective Memory Access Time Analysis (Bonus +5%)

Given the single level paging EMAT with TLB:

EMAT (effective memory access time) = $P\times$(TLB access time + memory access time)+$(1-P)\times$(TLB access time + $2\times$ memory access time)

- Question 1 : Derived the EMAT formula of "Multi-level Paging" (k level paging), assumed that no page fault occur in "Multi-level Page" access.
- Question 2 : Following from the question above, if we want to have a EMAT less then 180 nanoseconds, what is the maximum memory access time could be design in this system? Knowing that:
  - The system have 3 level paging.
  - TLB access time is 20 nanoseconds.
  - The TLB hit ratio is 80%.

5 Submission

5.1 Academic Honesty

1. Write the code on your own and write the report in your own words.
2. If your explanation does not correspond to your code, you are also suspected of committing plagiarism. TAs will decide whether you can get partial score for corresponding section, or fail this course.
3. List your assumptions, if any. Of course, any assumption should not violate the SPECs or (`./mp2_output`).
4. If your code was based on some critical assumptions and you lack them in report, TAs will decide whether you get partial score or get 0 for corresponding section.In some severe cases, you may be suspected of committing plagiarism.

5.2 Report

- Submit your report to "Report" section in NTUCOOL in one PDF file named `d08922025_mp2_report.pdf`, for example.
- Submit your bonus report to "Bonus Report" section in NTUCOOL in one PDF file named `d08922025_mp2_bonus.pdf`, for example. This item is judged only when the PDF is submitted.

5.3 Source Code

- Make sure your xv6 part can be compiled and run `make clean` before submission.
- Please compress your xv6 source code as `<Lowercase Student ID>.zip` (Example: d08922025.zip) and upload it to NTUCOOL.
- Never compress files we do not request, such as `.o`, `.d`, `.asm` files.
- Submit the zip file to the "Machine Problem 2" on NTUCOOL.
- (update 2) Safe ways to submit code: If you use git, remove untracked files in the repository. If you don't use git, run `make clean` before submission.

5.4 Folder Structure after Unzip

We will unzip your submission by running `unzip *.zip`. The unzipped folder structure should look like this. The common error is missing the top level directory named `<student_id>`.

```
<student_id>
|
+-- xv6
    |
    +-- kernel
    |   |
    |   +-- defs.h
    |   +-- exec.c
    |   +-- paging.c
    |   +-- sysproc.c
    |   +-- trap.c
    |   +-- (remaining files ...)
    |
    +-- Makefile
```

5.5 Grading Policy

- Compile error leads to 0 points in the submission.
- Erroneous folder structure incurs 10 point penalty. Using uppercase in the `<student_id>` is also a kind of wrong folder structure.
- If your submission is late for n days, your score will be $raw\_score \times (100\% - 20\% \times \lceil n \rceil)$. Note that you will not get any points if $\lceil n \rceil >= 5$.

6 Appendix

6.1 Macros and Builtin Types

6.1.1 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"
```

6.1.2 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

6.1.3 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`.

6.1.4 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` isn't. 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);
}
```

6.2 Functions Used in The Homework

The following functions are used to de/allocate pages on a 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 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()`.

6.3 Functions to be Modified in the Homework

(update) 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()`

6.4 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();
```

6.5 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()`.

7 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/