Machine Problem 2 - Virtual Memory, Swapping and Page Replacement

CSIE3310 - Operating Systems
National Taiwan University

Total Points:	100
Release Date:	March 19
Due Date:	April 1, 23:59:00
TA e-mail:	`ntuos@googlegroups.com`
TA hours:	Mon. 11:00-12:00, Wed. 14:00-15:00, CSIE Building B04 (Please knock the door)

Contents

1 Summary

The virtual memory system provides each process with an isolated and abstracted memory space. Through per-process page tables, only a portion of virtual memory pages are actively mapped to physical memory. This sophisticated memory management enables the operating system to manage processes that have large virtual memory spaces while utilizing a relatively small amount of physical memory.

To further enhance memory efficiency, modern operating systems implement page replacement algorithms. Page replacement is a technique used to select which memory pages to swap out to disk when physical memory is full. This homework introduces two classic page replacement strategies: First In, First Out (FIFO) and Least Recently Used (LRU). FIFO evicts the oldest page in memory, while LRU removes the page that has been unused for the longest time. These algorithms aim to optimize the use of available memory and the performance of the system by keeping the most relevant pages in physical memory and swapping out the less frequently used ones.

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 with 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
   ```

4. Enter `home` directory

   ```
   $ cd
   ```

5. Run Individual Tests
   You can refer the correct output in mp2_output directory for each public test cases.

   (a) For `mp2_1`, $\ldots, 3$, run `make qemu` before `mp2_N` command, where $N = 1, \ldots, 3$

   ```
   $ make clean
   $ make qemu
   ...
   $ mp2_1
   ```

   (b) For `mp2_ 4`, run `make fifo` before `mp2_4` command

   ```
   $ make clean
   $ make fifo
   ...
   $ mp2_4
   ```

   (c) For `mp2_5`, run `make lru` before `mp2_5` command

```
      $ make clean
      $ make lru
      ...
      $ mp2_5
```

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

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

4 Scoring and Submission

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

- Public program tests (60%)
- Private program tests (40%)

No partial points will be given !!

4.1 Run the Preliminary Judge

You can get 60 points (100 points in total) if you pass all public test cases. You can judge the code by running the following command in the docker container (not in xv6; this should run in the same place as make qemu).

```
$ make grade
```

If you successfully pass all the public test cases, the output should be similar to the one below.

```
== Test mp2_1 (6%) ==
mp2_1: OK (3.4s)
== Test mp2_2 (12%) ==
mp2_2: OK (1.1s)
== Test mp2_3 (12%) ==
mp2_3: OK (1.0s)
== Test mp2_4 (12%) ==
mp2_4: OK (4.6s)
== Test mp2_5 (18%) ==
mp2_5: OK (4.5s)
Score: 60/60
```

4.2 Submission

Run this command to pack your code into a zip named in your lowercase student ID, for example, `d10922013.zip`. Submit this file to *Machine Problem 2* section on NTUCOOL.

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

4.3 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 penalty. The score is deduced by 20 points per 24 hours later on. For example, 25-hour late submission leads to 40 points deduction.

5 Assignment

It is recommended to read materials in Section 6 before writing your code. It will save your time. Also, do your homework early. :)

5.1 Print a Page Table (Public 6%+Private 4%)

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 the `vmprint()` function in `kernel/vm.c` to dump the page table tree. It takes a page table of type `pagetable_t` and print that page table in the format below.

```
$ mp2_1
page table 0x0000000087f49000
+-- 0: pte=0x0000000087f49000 va=0x0000000000000000 pa=0x0000000087f45000 V
|   +-- 0: pte=0x0000000087f45000 va=0x0000000000000000 pa=0x0000000087f44000 V
|       +-- 0: pte=0x0000000087f44000 va=0x0000000000000000 pa=0x0000000087f46000 V R W X U
|       +-- 1: pte=0x0000000087f44008 va=0x0000000000001000 pa=0x0000000087f43000 V R W X
|       +-- 2: pte=0x0000000087f44010 va=0x0000000000002000 pa=0x0000000087f42000 V R W X U D
+-- 255: pte=0x0000000087f497f8 va=0x0000003fc0000000 pa=0x0000000087f48000 V
    +-- 511: pte=0x0000000087f48ff8 va=0x0000003fffe00000 pa=0x0000000087f47000 V
        +-- 510: pte=0x0000000087f47ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W D
        +-- 511: pte=0x0000000087f47ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
```

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, D` 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.

5.2 Swapping (Public 24%+Private 16%)

Page replacement with swapping is a technique to manage memory pages in a secondary storage, typically a hard disk. This technique involves replacing pages in main memory with pages from secondary storage when the available physical memory is insufficient to accommodate all the required pages. This approach aims to optimize memory utilization and performance. 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.

5.2.1 Handle Page Faults

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 `handle_page_fault()` in `/kernel/paging.c` and call that function whenever a page fault occurs. Check Figure 1 for the exception codes of scause.

- Always mark `PTE_U, PTE_R, PTE_W, PTE_X` flags on newly allocated pages.

- In `handle_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.

5.2.2 Swapping / Pinning Pages

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.

Figure 1: Supervisor cause register (scause) values after trap. *Andrew Waterman, Krste Asanovic, The RISC-V Instruction Set Manual*

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	≥10	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	9–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	≥16	*Reserved*

– `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. This feature is provided by TAs, thus you can reference the implementation to implement the feature of `MADV_WILLNEED`, `MADV_PIN`, and `MADV_UNPIN`. Note that you need to extend the `vmprint()` function to show the `PTE_S` bit if a page is swapped. Show block numbers for swapped page entries.
Sampled output:

```
page table 0x0000000087f49000
+-- 0: pte=0x0000000087f49000 va=0x0000000000000000 pa=0x0000000087f45000 V
|   +-- 0: pte=0x0000000087f45000 va=0x0000000000000000 pa=0x0000000087f44000 V
|       +-- 0: pte=0x0000000087f44000 va=0x0000000000000000 pa=0x0000000087f46000 V R W X U
|       +-- 1: pte=0x0000000087f44008 va=0x0000000000001000 pa=0x0000000087f43000 V R W X
|       +-- 2: pte=0x0000000087f44010 va=0x0000000000002000 blockno=0x00000000000002f0 R W X U S
+-- 255: pte=0x0000000087f497f8 va=0x0000003fc0000000 pa=0x0000000087f48000 V
    +-- 511: pte=0x0000000087f48ff8 va=0x0000003fffe00000 pa=0x0000000087f47000 V
        +-- 510: pte=0x0000000087f47ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W D
        +-- 511: pte=0x0000000087f47ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
```

– `MADV_PIN`: Make pages within the region that cannot be page out. Note that you need to extend the `vmprint()` function to show the `PTE_P` bit if a page is pinned.
Sampled output:

```
page table 0x0000000087f49000
+-- 0: pte=0x0000000087f49000 va=0x0000000000000000 pa=0x0000000087f45000 V
|   +-- 0: pte=0x0000000087f45000 va=0x0000000000000000 pa=0x0000000087f44000 V
|       +-- 0: pte=0x0000000087f44000 va=0x0000000000000000 pa=0x0000000087f46000 V R W X U
|       +-- 1: pte=0x0000000087f44008 va=0x0000000000001000 pa=0x0000000087f43000 V R W X
|       +-- 2: pte=0x0000000087f44010 va=0x0000000000002000 pa=0x0000000087f42000 V R W X U P
+-- 255: pte=0x0000000087f497f8 va=0x0000003fc0000000 pa=0x0000000087f48000 V
    +-- 511: pte=0x0000000087f48ff8 va=0x0000003fffe00000 pa=0x0000000087f47000 V
        +-- 510: pte=0x0000000087f47ff0 va=0x0000003fffffe000 pa=0x0000000087f65000 V R W D
        +-- 511: pte=0x0000000087f47ff8 va=0x0000003ffffff000 pa=0x0000000080007000 V R X
```

– `MADV_UNPIN`: Make pages within the region that can be page out if necessary.

It leverages `balloc_page()` and `write_page_to_disk()` to allocate a page on the storage device. More utility functions can be found at Section 6.

5.2.3 madvise MADV_NORMAL Option

Handle the MADV_NORMAL option in madvise() syscall. It does nothing and just return.
This option is combined into the test case of swapping in, swapping out, please read 5.2.4.

5.2.4 Swap In Pages

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 may goes through the following steps:

1. Call vmprint() to check current page table.
2. Increase the process memory by malloc(x * PGSIZE) where x >= 16.
3. Call madvise() with MADV_NORMAL option to the correspond pages, and call vmprint() to check current page table.
4. Call madvise() with MADV_DONTNEED option to swap the correspond pages to the disk, and call vmprint() to check current page table.
5. 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.

5.2.5 Pin/Unpin Pages

The concept of "pinning" a page table entry (PTE) is related to memory management strategies, particularly in scenarios where certain pages of memory need to be prevented from being moved or swapped out by the operating system. Handle the MADV_PIN option in madvise() syscall. It ensures the pages within the memory region to be pined or unpined. The PTE_P bit is set on those affected page table entries.
The testing program may goes through the following steps:

1. Increase the process memory by malloc(x * PGSIZE) where x >= 16.
2. Call vmprint() to check current page table.
3. Call madvise() with MADV_PIN option to pin the correspond pages, and call vmprint() to check current page table.
4. Call madvise() with MADV_UNPIN option to unpin the correspond pages. Then, Call vmprint() to check the page table.

5.3 Page Replacement (Public 30%+Private 20%)

Implement your algorithms in fifo.c and lru.c file. We have provided a template of data structure which to implement FIFO(fifo.c) and LRU(lru.c) algorithms. You can use the template (encouraged) or change it if you do not want to use it. Note that the fields inside the structure must be primitive type and array type.

- This preprocessor directives are used to differentiate FIFO or LRU page replacement algorithm. You will see multiple preprocessor directives like this and you should implement your page replacement algorithm in corresponding block.

```
    #ifdef (PG_REPLACEMENT_USE_LRU)
        \\ LRU behavior
    #elif defined(PG_REPLACEMENT_USE_FIFO)
        \\ FIFO behavior
    #endif
```

- How to select page replacement algorithm when booting xv6?
  - FIFO

```
    $make fifo
```

  – LRU

    ```
    $make lru
    ```

- You should implement `pgprint()` in `kernel/vm.c` to show final page replacement buffer for grading public test cases mp2_{4, 5}. Note that the next most likely victim should output first.
  Sampled output:

  ```
  Page replacement buffers
  ------Start------------
  pte: 0x0000000087f44040
  pte: 0x0000000087f44048
  pte: 0x0000000087f44050
  pte: 0x0000000087f44058
  pte: 0x0000000087f44060
  pte: 0x0000000087f44068
  pte: 0x0000000087f44078
  pte: 0x0000000087f44080
  ------End--------------
  ```

Hints:

- Pinned pages should not be replaced
- Swapped out pages should be removed from page replacement buffer
- The page replacement buffer size will be 8 for grading
- You have to ignore the first three PTEs
- The page replacement occurs in your page replacement buffer, do not try to modify the process's `pagetable_t` directly

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_D, PTE_P`.

- `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.
- `PTE_D`
  If set, the page at the address is written.
- `PTE_P`
  If set, the page at the address is pined.

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);
}
```

6.2 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.

6.3 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()`
  - `pgprint()`
- `kernel/paging.c`
  - `handle_pgfault()`
- `kernel/trap.c`
  - `usertrap()`
- `kernel/fifo.c`
  - `q_init()`
  - `q_push()`
  - `q_pop_idx()`
  - `q_empty()`
  - `q_full()`
  - `q_clear()`
  - `q_find()`
- `kernel/lru.c`
  - `lru_init()`
  - `lru_push()`
  - `lru_pop()`
  - `lru_empty()`
  - `lru_full()`
  - `lru_clear()`
  - `lru_find()`

  Note that for fifo/lru.[ch] file, if you want to rename the function, you must need to change the header file accordingly. For example, if you change the function in fifo.c, then you also need to change the fifo.h.

6.4 Generate your own test cases

You may want to write own test cases, you may use `malloc()`, but just ignore `free()`

- `user/umalloc.c`
  - `malloc()`, the `vmprint()` is your implementation in 5.1.1. Your test case must store in user/ directory. Then, open Makefile and search UPROGS variable, add your file to the last entry of this variable. For example, if your file called mytest.c, add U/_mytest to the last entry:

```
UPROGS=\
        $U/_cat\
        $U/_echo\
        $U/_forktest\
        $U/_grep\
        $U/_init\
        $U/_kill\
        $U/_ln\
        $U/_ls\
        $U/_mkdir\
        $U/_rm\
        $U/_sh\
        $U/_stressfs\
        $U/_usertests\
        $U/_grind\
        $U/_wc\
        $U/_zombie\
        $U/_mp2_1\
        $U/_mp2_2\
        $U/_mp2_3\
        $U/_mp2_4\
        $U/_mp2_5\
        $U/_mytest
```

  Sample test case:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"
#include "kernel/vm.h"

#define PG_SIZE 4096
#define NR_PG 18

int main(int argc, char *argv[]) {
  char *ptr = malloc(NR_PG * PG_SIZE);
  vmprint();
  exit(0);
}
```

6.5 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();
```

6.6 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 in page table(not allocted or swapped out).

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()`.

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.

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/

[6] RISC-V privileged manual (table 4.2)
https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf