Tutorial 6 **CS3103 Operating Systems**

Student Name: _____

Student No. : | | | | | | | | |

Day: ☐ Monday ☐ Tuesday

Time: ☐ 10:00 - 10:50 ☐ 11:00 - 11:50 ☐ 12:00 - 12:50 ☐ 14:00 - 14:50 ☐ 18:00 - 18:50

Paging

Introduction

Topics to be covered in this tutorial include:

- Explore how simple virtual-to-physical address translation works with linear page tables;
- Explore how multi-level page tables works.

Acknowledgement

This tutorial was adapted from OSTEP book written by Remzi and Andrea Arpaci-Dusseau at the University of Wisconsin. This free OS book is available at http://www.ostep.org.

Getting Started

1. Logging in to the Linux server

- Start the SSH client, e.g., MobaXterm or Xshell.
- Log in to the Linux server using the following details:

  Host Name: gateway.cs.cityu.edu.hk

  User Name: your EID (e.g., cctom2)

  Password: your password

Your password will not be shown on the screen as you type it, not even as a row of stars (******).

NOTE: The shell will always give you a prompt if it is ready to accept commands. The shell prompt normally ends in a `$` sign as we use in this tutorial. Some shell prompts end in `%` or `>` instead. Never copy/type the shell prompt used in this tutorial.

NOTE: Please don't forget to log out (use the `exit` command) after you finish your work.

2. Getting the Simulators

This tutorial lets you explore how simple virtual-to-physical address translation works with a linear page table or a multi-level page table. The simulators are available at */public/cs3103/tutorial6* on the gateway server:

- `paging-linear-translate.py`: it simulates virtual-to-physical address translation based on linear page tables.
- `paging-multilevel-translate.py`: it simulates virtual-to-physical address translation based on multi-level page tables.

As before, follow the same copy procedure as you did in the previous tutorials to get the simulators.

Introduction to paging-linear-translate.py

In this tutorial, you will first use a simple program, which is known as `paging-linear-translate.py`, to see if you understand how simple virtual-to-physical address translation works with linear page tables. To run the program, remember to either type just the name of the program (`./paging-linear-translate.py`) or possibly this (`python paging-linear-translate.py`).

First, run the program without any arguments:

```
$ ./paging-linear-translate.py
ARG seed 0
ARG address space size 16k
ARG phys mem size 64k
ARG page size 4k
ARG verbose False
ARG addresses -1

The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
   0x8000000c
   0x00000000
   0x00000000
   0x80000006

Virtual Address Trace
  VA 0x00003229 (decimal:    12841) --> PA or invalid address?
  VA 0x00001369 (decimal:     4969) --> PA or invalid address?
  VA 0x00001e80 (decimal:     7808) --> PA or invalid address?
  VA 0x00002556 (decimal:     9558) --> PA or invalid address?
  VA 0x00003a1e (decimal:    14878) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

As you can see, what the program provides for you is a page table for a particular process (remember, in a real system with linear page tables, there is one page table per process; here we just focus on one process, its

address space, and thus a single page table). The page table tells you, for each virtual page number (VPN) of the address space, that the virtual page is mapped to a particular physical frame number (PFN) and thus valid, or not valid.

The format of the page-table entry is simple: the left-most (high-order) bit is the valid bit; the remaining bits, if the valid bit is 1, are the PFN.

In the example above, the page table maps VPN 0 to PFN 0xc (decimal 12), VPN 3 to PFN 0x6 (decimal 6), and leaves the other two virtual pages, 1 and 2, as not valid.

Because the page table is a linear array, what is printed above is a replica of what you would see in memory if you looked at the bits yourself. However, it is sometimes easier to use this simulator if you run with the verbose flag (-v); this flag also prints out the VPN (index) into the page table. From the example above, run with the -v flag:

```
Page Table (from entry 0 down to the max size)
  [       0]   0x8000000c
  [       1]   0x00000000
  [       2]   0x00000000
  [       3]   0x80000006
```

Your job, then, is to use this page table to translate the virtual addresses given to you in the trace to physical addresses. Let's look at the first one: VA 0x3229. To translate this virtual address into a physical address, we first have to break it up into its constituent components: a virtual page number and an offset. We do this by noting down the size of the address space and the page size. In this example, the address space is set to 16KB (a very small address space) and the page size is 4KB. Thus, we know that there are 14 bits in the virtual address, and that the offset is 12 bits, leaving 2 bits for the VPN. Thus, with our address 0x3229, which is binary 11 0010 0010 1001, we know the top two bits specify the VPN. Thus, 0x3229 is on virtual page 3 with an offset of 0x229.

We next look in the page table to see if VPN 3 is valid and mapped to some physical frame or invalid, and we see that it is indeed valid (the high bit is 1) and mapped to physical page 6. Thus, we can form our final physical address by taking the physical page 6 and adding it onto the offset, as follows: 0x6000 (the physical page, shifted into the proper spot) OR 0x0229 (the offset), yielding the final physical address: 0x6229. Thus, we can see that virtual address 0x3229 translates to physical address 0x6229 in this example.

The rest of the solutions is given as follows:

```
VA  0: 00003229 (decimal: 12841) --> 00006229 (25129) [VPN 3]
VA  1: 00001369 (decimal:  4969) --> Invalid (VPN 1 not valid)
VA  2: 00001e80 (decimal:  7808) --> Invalid (VPN 1 not valid)
VA  3: 00002556 (decimal:  9558) --> Invalid (VPN 2 not valid)
VA  4: 00003a1e (decimal: 14878) --> 00006a1e (27166) [VPN 3]
```

Of course, you can change many of these parameters to make more interesting problems. Run the program with the -h flag to see what options there are:

- The -s flag changes the random seed and thus generates different page table values as well as different virtual addresses to translate.
- The -a flag changes the size of the address space.
- The -p flag changes the size of physical memory.
- The -P flag changes the size of a page.
- The -n flag can be used to generate more addresses to translate (instead of the default 5).
- The -u flag changes the fraction of mappings that are valid, from 0% (-u 0) up to 100% (-u 100). The default is 50, which means that roughly 1/2 of the pages in the virtual address space will be valid.
- The -v flag prints out the VPN numbers to make your life easier.

Now go answer the first three questions then come back and continue.

Introduction to paging-multilevel-translate.py

Now you have the basics of how simple virtual-to-physical address translation works with linear page tables, let's go ahead and see how a multi-level page table works.

The program is called: `paging-multilevel-translate.py`.

Some basic assumptions:

- The page size is an unrealistically-small 32 bytes;
- The virtual address space for the process in question (assume there is only one) is 1024 pages, or 32 KB;
- physical memory consists of 128 pages.

Thus, a virtual address needs 15 bits (5 for the offset, 10 for the VPN). A physical address requires 12 bits (5 offset, 7 for the PFN).

The system assumes a multi-level page table. Thus, the upper five bits of a virtual address are used to index into a page directory; the page directory entry (PDE), if valid, points to a page of the page table. Each page table page holds 32 page-table entries (PTEs). Each PTE, if valid, holds the desired translation (physical frame number, or PFN) of the virtual page in question.

The format of a PTE is thus:

```
VALID | PFN6 ... PFN0
```

and is thus 8 bits or 1 byte.

The format of a PDE is essentially identical:

```
VALID | PT6 ... PT0
```

You are given two pieces of information to begin with.

First, you are given the value of the page directory base register (PDBR), which tells you which page the page directory is located upon.

Second, you are given a complete dump of each page of memory. A page dump looks like this:

```
page 0: 08 00 01 15 11 1d 1d 1c 01 17 15 14 16 1b 13 0b ...
page 1: 19 05 1e 13 02 16 1e 0c 15 09 06 16 00 19 10 03 ...
page 2: 1d 07 11 1b 12 05 07 1e 09 1a 18 17 16 18 1a 01 ...
...
```

which shows the 32 bytes found on pages 0, 1, 2, and so forth. The first byte (0th byte) on page 0 has the value 0x08, the second is 0x00, the third 0x01, and so forth.

You are then given a list of virtual addresses to translate.

Use the PDBR to find the relevant page table entries for this virtual page. Then find if it is valid. If so, use the translation to form a final physical address. Using this address, you can find the VALUE that the memory reference is looking for.

Of course, the virtual address may not be valid and thus generate a fault.

Some useful options:

```
-s SEED, --seed=SEED  the random seed
-a ALLOCATED, --allocated=ALLOCATED
                      number of virtual pages allocated
-n NUM, --addresses=NUM
                      number of virtual addresses to generate
```

Change the seed to get different problems, as always. Change the number of virtual addresses generated to do more translations for a given memory dump.

Read question 4 to study a multi-level page table based virtual-to-physical address translation in more depth.

Questions

All questions should be answered on the separate answer sheet provided.

1. Before doing any translations, let's use the simulator `paging-linear-translate.py` to study how linear page tables change size given different parameters. Compute the size of linear page tables as different parameters change. Some suggested inputs are below; by using the `-v` flag, you can see how many page-table entries are filled. First, to understand how linear page table size changes as the address space grows, run with these flags:

```
-P 1k -a 1m -p 512m -v -n 0
-P 1k -a 2m -p 512m -v -n 0
-P 1k -a 4m -p 512m -v -n 0
```

   Then, to understand how linear page table size changes as page size grows:

```
-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

   Before running any of these, try to think about the expected trends. How should page-table size change as the address space grows? As the page size grows? Why not use big pages in general?

2. Now let's do some translations. Start with some small examples and change the number of pages that are allocated to the address space with the `-u` flag. For example:

```
-P 1k -a 16k -p 32k -v -u 0
-P 1k -a 16k -p 32k -v -u 25
-P 1k -a 16k -p 32k -v -u 50
-P 1k -a 16k -p 32k -v -u 75
-P 1k -a 16k -p 32k -v -u 100
```

   What happens as you increase the percentage of pages that are allocated in each address space?

3. Now let's try some different random seeds, and some different (and sometimes quite crazy) address-space parameters, for variety:

```
-P 8 -a 32 -p 1024 -v -s 1
-P 8k -a 32k -p 1m -v -s 2
-P 1m -a 256m -p 512m -v -s 3
```

   Which of these parameter combinations are unrealistic? Why?

4. Use the simulator `paging-multilevel-translate.py` to perform translations. Run:

```
$ ./paging-multilevel-translate.py -s 3103
```

You are given the value of the PDBR, a complete dump of each page of memory, and a list of virtual addresses to translate. Solutions to the first two addresses are given below for reference:

- Virtual Address 4a14: Translates to what Physical Address (and fetches what value)? Or Fault?

```
Virtual Address 4a14:
  --> pde index:0x12 [decimal 18] pde contents:0x9a (valid 1, pfn 0x1a [decimal 26])
    --> pte index:0x10 [decimal 16] pte contents:0xd8 (valid 1, pfn 0x58 [decimal 88])
      --> Translates to Physical Address 0xb14 --> Value: 17
```

- Virtual Address 685e: Translates to what Physical Address (and fetches what value)? Or Fault?

```
Virtual Address 685e:
  --> pde index:0x1a [decimal 26] pde contents:0xbf (valid 1, pfn 0x3f [decimal 63])
    --> pte index:0x2 [decimal 2] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
```

For each of the following virtual addresses, write down the physical address it translates to or write down that it is a fault (e.g., page directory entry not valid, page table entry not valid).

- Virtual Address 5a23
- Virtual Address 14ab
- Virtual Address 257e
- Virtual Address 7988
- Virtual Address 75cf
- Virtual Address 3350
- Virtual Address 0a70
- Virtual Address 55f9