بسم الله الرحمن الرحيم

EE 463 : Operating Systems
Faculty Of Engineering
King Abdulaziz University
Department Of Electrical and Computer Engineering

LAB #8

Virtual Memory Management

Linear Page Translation

Objective:

All modern operating systems provide virtual memory management in some way. One very commonly used method for managing computer virtual memory is called paging. Almost all modern computers have hardware support (MMU) for paging. Virtual memory systems make it possible to easily load and run programs regardless of their size compared to actual memory size, or where they are loaded in memory. This is done by having two independent address spaces: virtual (logical) and real (physical), and providing a way to dynamically translate a virtual address generated by a running program to the corresponding real address. In this lab, you are going to use a program that will show you how this translation is done using a paging system. You will exercise the calculations involved in translating virtual addresses to real addresses using a one-level page table, and calculating the amount of memory required to store the page table in memory for a single process.

Background:

There are two programs for this lab, written in *Python*, and you can run them on any computer which has *Python* interpreter version *3.8* or above installed. The two programs are called *pagetrans.py*, and *pagetablesize.py*. You can download them from the lab's page of the course web site. The first one generates virtual address translation problems for systems using a one-level page table, where you can specify the system parameters at the command-line, or they can be automatically generated for you in random, depending on a random seed that you specify with the command, thus, each time you change the random seed, a new problem is generated. The second program is used to exercise calculating the page-table size, thus the memory size required to store the table for a single user program in a given system.

How to use the programs:

1) Make sure you have *Python 3.8* or above installed in your system.
2) Download *paging.zip* from the course web site. Unzip the file and note where you saved it.
3) Use the command interface and change to the directory you noted above.
4) Begin with *pagetrans.py*, the command to run it has the following format:

```
> python ./pagetrans.py [options]
```

The command-line options are:

```
-h,            --help                show this help message and exit
-A ADDRESSES,  --addresses=ADDRESSES a set of comma-separated pages to access;
                                     -1(one) means randomly generate - Default= -1
-s SEED,       --seed=SEED           the random seed - Default= 0
-a ASIZE,      --asize=ASIZE         address space size (e.g., 16, 32k, 64m, 1g)
-r RSIZE,      --realmem=RSIZE       real memory size (e.g., 16, 128k, 32m, 1g)
-p PAGESIZE,   --pagesize=PAGESIZE   page size (e.g., 4k, 8k, … etc.), - Default= 4k
```

```
-n NUM,        --numaddrs=NUM        no. virtual addresses to generate – Default= 5
-u USED,       --used=USED           percent of virtual address space used– Dflt= 50
```

Example:

5) Enter the following command:

```
> python ./pagetrans.py -a 16k -r 64k -s 110
```

The program will automatically generate the following system. Note that if you later want to generate this exact same system, you have to specify these exact same parameters and seed again. You should see:

```
ARG seed 110
ARG address space size 16k
ARG real memory size 64k
ARG page size 4k
ARG verbose True
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.

Page Table (from entry 0 down to the max size)
  [       0]   0x80000006
  [       1]   0x80000003
  [       2]   0x00000000
  [       3]   0x00000000

Virtual Address Trace
  VA 0x000022b0 (decimal:     8880) --> RA or invalid address?
  VA 0x000004cc (decimal:     1228) --> RA or invalid address?
  VA 0x00001aab (decimal:     6827) --> RA or invalid address?
  VA 0x0000143a (decimal:     5178) --> RA or invalid address?
  VA 0x00000290 (decimal:      656) --> RA or invalid address?

For each virtual address, write down the real address it translates to
OR write down that it is an out-of-bounds address (e.g., Not Valid).
```

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 the address space of one process, thus a single page table). The page table tells you, for each virtual page number (VPN) of the user's program address space, that the virtual page is mapped to a particular real Page Frame Number (PFN) and thus valid, or invalid according to its existence in a real memory page frame or not, respectively.

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

In the above example, the page table maps VPN 0 to PFN 0x6 (decimal 6), VPN 1 to PFN 0x3 (decimal 3), and leaves the other two virtual pages, VPN 2 and VPN 3, as not valid.

Your job, then, is to use this page table to translate the virtual addresses (**VA**s) given to you in the trace to real addresses (**RA**s). It requires **4 steps**. Let's look at the first and second virtual addresses:

- VA: 0x000022b0:
  1. Break the VA up into its constituent components: a Virtual Page Number and an offset (Displacement) [VPN | D]. 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 (2^{14} a very small address space) and the page size is 4KB (2^{12}). Thus, we know that there are 14 bits in the VA, and that D is 12 bits, leaving 2 bits for the VPN.
  2. Thus, with our address 0x22b0, which is binary 10 0010 1011 0000, we know the top two bits specify the VPN. Thus, VA 0x22b0 is on virtual page 2 with an offset of 0x2b0.
  3. Look in the page table to see if VPN 2 is valid and mapped to some real frame or invalid. We see that it is indeed invalid (the high bit is 0).
  4. Thus 0x22b0 is **invalid** virtual address.

- VA: 0x000004cc:
  1. Again, to translate this virtual address into a real address, we break it up into its two components: [2-bits VPN | 12-bits D] = [0x0 | 0x4cc] = [00 | 0100 1100 1100].
  2. Thus 0x04cc is on virtual page 0 with an offset of 0x4cc.
  3. Look in the page table to see if VPN 0 is valid and mapped to some real frame or invalid. We see that it is valid (the high bit is 1) and mapped to Page Frame Number, PFN 6.
  4. Form the final Real Address (**RA**) by taking the PFN 6 and adding it onto the offset, D, as follows:

     0x6000 (real page frame number, shifted into the proper spot) **OR** 0x4cc (the offset), yielding the final real address 0x64cc. Thus, we can see that virtual address (VA): 0x04cc translates to real address (RA) 0x64cc in this example.

6) Compute the rest of the solutions below; note down the answers here; this is ***part1*** of your report.

```
Virtual Address Trace
  VA 0x000022b0 (decimal:     8880) --> Invalid  (VPN 2 not valid)
  VA 0x000004cc (decimal:     1228) --> RA 000064cc (decimal    25804) [VPN 0]
  VA 0x00001aab (decimal:     6827) --> _____________________________________
  VA 0x0000143a (decimal:     5178) --> _____________________________________
  VA 0x00000290 (decimal:      656) --> _____________________________________
```

The above program can generate many interesting problems for you, if you change any of its parameters; try it out and exercise other possibilities.

7) Do another simulation with different parameters. Write the VA trace as above as ***part2*** of the report.

Operating systems that use paging, especially, one-level paging, tend to use a large portion of memory just to store page tables of all the loaded programs. Thus, the size of the page table is really an important issue, especially when the systems get larger and larger virtual address spaces, like 64-bit or even 32-bit.

The second part of this lab concerns this issue. The *pagetablesize.py* program allows you to exercise calculating the size of a one-level page table for a given system. It can be used as follows:

```
> python ./pagetablesize.py [options]
```

The command-line options are:

```
  -h,             --help                show this help message and exit
  -v VSIZE,       --vasize=VSIZE        bits in virtual address (e.g., 16, 20, 32)
  -e PTESIZE,     --ptesize=PTESIZE     size of the page table entry (e.g., 4 bytes)
  -p PAGESIZE,    --pagesize=PAGESIZE   size of the page (e.g., 4k, 4096, 1m)
```

Example:

8) Enter the following command:

```
> python ./pagetablesize.py -v 38 -p 4k -e 4
```

The program will automatically generate the following system. You should see:

```
ARG bits in virtual address 38
ARG page size 4k
ARG pte size 4

Compute how big a linear page table is with the characteristics you see above.

REMEMBER: There is one linear page table *per process*.
Thus, the more running processes, the more of these tables we will need.
However, for this problem, we are only concerned about the size of *one* table.
```

Recall that a virtual address (VA) has two components: [Virtual Page Number (VPN) | Offset (D)]. In the above example, we have:

The number of *bits* in the VA:	38
The page size:	4096 (2^{12}) *bytes*
Thus, the number of *bits* needed in the offset:	$\log_2(4096) = 12$
Which leaves this many *bits* for the VPN:	$38 - 12 = 26$

Thus, a virtual address looks like this:

```
                3                   2                   1                   0
  7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
[ V V V V V V V V V V V V V V V V V V V V V V V V V V | O O O O O O O O O O O O ]
|<------------------------26 bits------------------------>|<----------12 bits---------->|
```

Where, V is for a VPN bit and O is for an offset bit. To compute the size of the linear page table, we need to know:

- The number of entries in the table, which is: $2^{(\text{no. VPN bits})} = 2^{26}$ = 67,108,864 Entry
- The size of each page table entry, which is: 4 Bytes

And then multiply them together. The final result:

4 x 67108864	= 268,435,456	Bytes
	= 262,144	KB
	= 256	MB

Exercise:

9) Run the program again twice with some other parameters! (i.e. different number of bits in the address space, different page size, different page table entry size).

10) Calculate the page table size in each of the above two cases, and note the parameters you used along with the calculation results for both cases, as ***part3*** of this lab's report.