Chapter 18

1. Before doing any translations, let's use the simulator 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?

I predict the following:
Page table size grows linearly with address space size as number of pages = address space size / page size.
Page table size decreases as page size increases as larger pages mean fewer pages needed to cover the same address space.

The results matched my predictions.

For the first 3 commands, the page table grow from 1024 (0-1023) -> 2048 (0-2047) -> 4096 (0-4065)

For the second 3 commands, the page table grow from 1024 (0-1023) -> 512 (0-511) -> 256 (0-255)

Not use big pages can prevent internal fragmentation: If we only need a few bytes but allocate a whole 4KB page, we waste 4KB - a few bytes.

On the other hand, it does offer benefits like:
Smaller page tables: Less memory overhead for page tables
Fewer TLB misses: Each TLB entry covers more memory

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?

As we increase the -u (utilization) percentage from 0% to 100%, more pages in the page table become valid.

For instance, when -u is 0:

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

```
  [       0]  0x00000000
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x00000000
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x00000000
  [       9]  0x00000000
  [      10]  0x00000000
  [      11]  0x00000000
  [      12]  0x00000000
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x00000000
```

Virtual Address Trace

```
  VA 0x00003a39 (decimal:    14905) -->  Invalid (VPN 14 not valid)
  VA 0x00003ee5 (decimal:    16101) -->  Invalid (VPN 15 not valid)
  VA 0x000033da (decimal:    13274) -->  Invalid (VPN 12 not valid)
  VA 0x000039bd (decimal:    14781) -->  Invalid (VPN 14 not valid)
  VA 0x000013d9 (decimal:     5081) -->  Invalid (VPN 4 not valid)
```

when -u = 50, about 50% of the VAs became valid.

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

```
  [       0]  0x80000018
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000000c
  [       4]  0x80000009
  [       5]  0x00000000
  [       6]  0x8000001d
  [       7]  0x80000013
  [       8]  0x00000000
  [       9]  0x8000001f
  [      10]  0x8000001c
  [      11]  0x00000000
  [      12]  0x8000000f
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x80000008
```

Virtual Address Trace
VA 0x00003385 (decimal: 13189) --> 00003f85 (decimal 16261) [VPN 12]
VA 0x0000231d (decimal: 8989) --> Invalid (VPN 8 not valid)
VA 0x000000e6 (decimal: 230) --> 000060e6 (decimal 24806) [VPN 0]
VA 0x00002e0f (decimal: 11791) --> Invalid (VPN 11 not valid)
VA 0x00001986 (decimal: 6534) --> 00007586 (decimal 30086) [VPN 6]

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

4. Use the program to try out some other problems. Can you find the limits of where the program doesn't work anymore? For example, what happens if the address-space size is bigger than physical memory?

What happens when virtual address space is larger than physical memory?

python3 paging-linear-translate.py -P 1k -a 64k -p 32k -v -u 100 -c

This should failed at u=100 because we need 64 pages but only have 32 frames

ARG seed 0
ARG address space size 64k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1
Error: physical memory size must be GREATER than address space size (for this simulation)

What if we need more pages than we have physical frames?

python3 paging-linear-translate.py -P 1k -a 64k -p 8k -v -u 100 -c

Should fail with error or warning

Need 64 pages (64k / 1k) but only have 8 frames (8k / 1k)

ARG seed 0
ARG address space size 64k
ARG phys mem size 8k
ARG page size 1k
ARG verbose True
ARG addresses -1
Error: physical memory size must be GREATER than address space size (for this simulation)

What if a single page can't fit in physical memory?

python3 paging-linear-translate.py -P 64k -a 32k -p 32k -v -u 100 -c

Should fail immediately as it can't allocate even one page!

ARG seed 0
ARG address space size 32k
ARG phys mem size 32k
ARG page size 64k
ARG verbose True
ARG addresses -1
Error: physical memory size must be GREATER than address space size (for this simulation)

What if pages are bigger than the entire address space?

python3 paging-linear-translate.py -P 64k -a 16k -p 128k -v -u 100 -c

It should fail. Would result in less than 1 page!

ARG seed 0
ARG address space size 16k
ARG phys mem size 128k

ARG page size 64k
ARG verbose True
ARG addresses -1
Error in argument: address space must be a multiple of the pagesize

What happens with extremely small page sizes?

python3 paging-linear-translate.py -P 1 -a 4k -p 8k -v -u 50 -c

Should work but creates a huge page table (4096 entries for 4KB address space!) This demonstrates page table overhead problem

Results are too long to fully include here, there are 4096 entries.

```
  [    4092]  0x00000000
  [    4093]  0x00000000
  [    4094]  0x00000000
  [    4095]  0x00000000
Virtual Address Trace
  VA 0x00000fa0 (decimal:     4000) -->  Invalid (VPN 4000 not valid)
  VA 0x00000dc0 (decimal:     3520) -->  Invalid (VPN 3520 not valid)
  VA 0x0000095a (decimal:     2394) --> 00000a87 (decimal 2695) [VPN 2394]
  VA 0x00000d70 (decimal:     3440) -->  Invalid (VPN 3440 not valid)
  VA 0x00000816 (decimal:     2070) --> 000006d9 (decimal 1753) [VPN 2070]
```

What's the minimum viable paging system?

python3 paging-linear-translate.py -P 16k -a 16k -p 32k -v -u 100 -c

Should work. There is only a 1 page table entry. All addresses map to same page (VPN = 0)chan

ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 16k
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.
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)
  [0] 0x80000001
Virtual Address Trace
  VA 0x00001aea (decimal: 6890) --> 00005aea (decimal 23274) [VPN 0]
  VA 0x00001092 (decimal: 4242) --> 00005092 (decimal 20626) [VPN 0]
  VA 0x000020b8 (decimal: 8376) --> 000060b8 (decimal 24760) [VPN 0]
  VA 0x000019ea (decimal: 6634) --> 000059ea (decimal 23018) [VPN 0]
  VA 0x00003229 (decimal: 12841) --> 00007229 (decimal 29225) [VPN 0]

Chapter 19

1. For timing, you'll need to use a timer (e.g., gettimeofday()). How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)

After doing a bit of research, I found that the precision level is hardware dependent, and there is no official documentation in the man page in Linux.
Some estimates that gettimeofday() has microsecond (μs) precision — about 1 μs.
But due to system overhead and noise, its practical accuracy is ~1–10 μs.
So we need the operation to take much longer than that to measure reliably.
I think I should repeat the memory access loop many times (millions of iterations) to minimize the noises, in other words, making the precision level less significant with respect to the total run time.

2. Write the program, called tlb.c, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.

See tlb.c in github

3. Now write a script in your favorite scripting language (bash?) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?

See run_tlb.sh in github
chmod +x run_tlb.sh
./run_tlb.sh > results.csv
Results:

Pages	ns_per_access
1	3.51
2	3.94
4	4.06
8	3.83
16	9.63
32	11.91
64	11.65
128	9.76
256	6.39

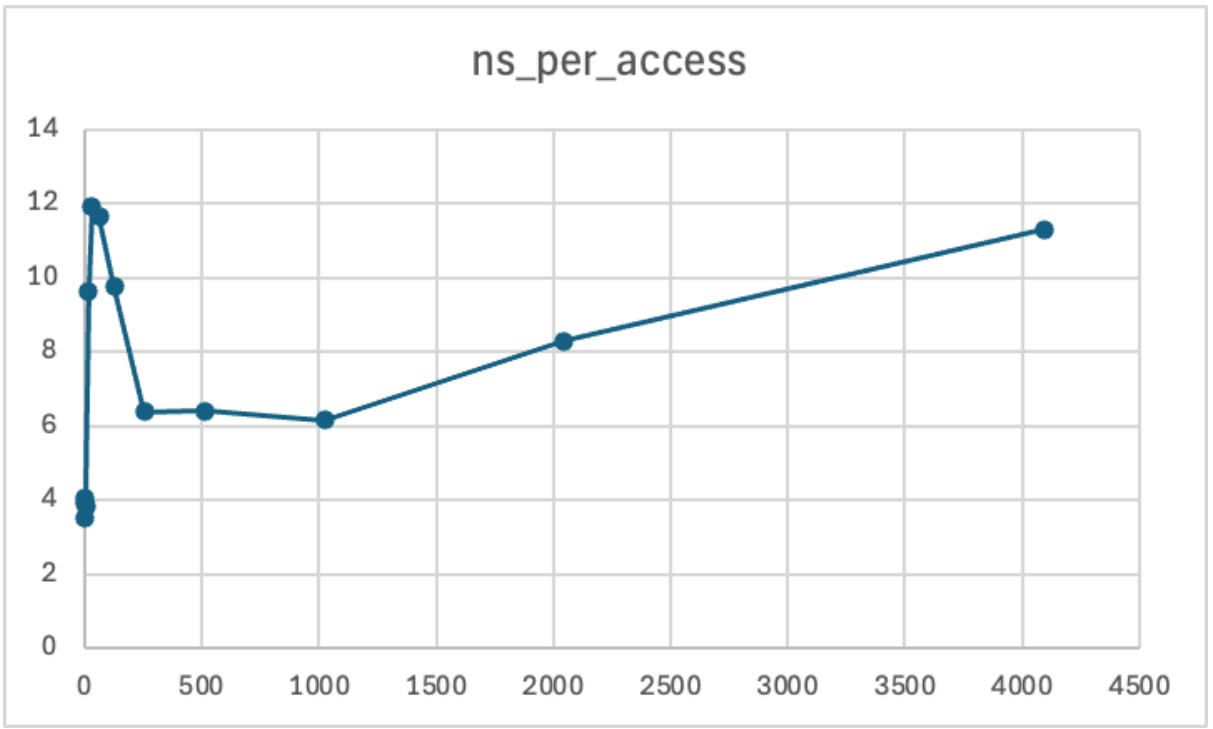
512	6.4
1024	6.16
2048	8.29
4096	11.3

To get stable numbers, we should choose trails large enough that each run lasts several seconds.

In khoury machine 100000 trials generate stable results.

```
[zw335812@login-students homework5]$ ./tlb 16 100000
16 pages, 9.26 ns per access
```

4. Next, graph the results, making a graph that looks similar to the one above. Use a good tool like ploticus or even zplot. Visualization usually makes the data much easier to digest; why do you think that is?



Clearly it is not as stable as the graph in the book, but we are able to see the jump despite the ns per access is not always a perfect linear relationship.

5. One thing to watch out for is compiler optimization. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program

subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?

gcc -O0 -o tlb tlb.c

-O0 - Sets the optimization level to zero

This means no optimization - the compiler generates code that closely matches source code line-by-line

Also we can use volatile so compiler cannot remove the loop

```
volatile int sink = 0;

if (sink == 0x12345678) printf("sink=%d\n", sink);
```

6. Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up “pinning a thread” on Google for some clues) What will happen if you don’t do this, and the code moves from one CPU to the other?

```
// --- Q6: Pin to a single CPU (CPU 0) ---

cpu_set_t set;

CPU_ZERO(&set);

CPU_SET(0, &set);

if (sched_setaffinity(0, sizeof(set), &set) != 0) {

    perror("sched_setaffinity");

}
```

sched_setaffinity()

Pins to one CPU so TLB behavior is consistent

7. Another issue that might arise relates to initialization. If you don’t initialize the array a above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?

We can run a warm-up loop that touches every page once before timing — this ensures pages are in memory and mapped in TLB.