CH20

1. With a linear page table, you need a single register to locate the page table, assuming that hardware does the lookup upon a TLB miss. How many registers do you need to locate a two-level page table? A three-level table?
We need one register to store the base of the page directory for the multi-level page table and locate it.

2. Use the simulator to perform translations given random seeds 0, 1, and 2, and check your answers using the -c flag. How many memory references are needed to perform each lookup?

PDBR: 108 (decimal) [This means the page directory is held in this page]

```
Virtual Address 611c:
  --> pde index:0x18 [decimal 24] pde contents:0xa1 (valid 1, pfn 0x21 [decimal 33])
    --> pte index:0x8 [decimal 8] pte contents:0xb5 (valid 1, pfn 0x35 [decimal 53])
      --> Translates to Physical Address 0x6bc --> Value: 08
Virtual Address 3da8:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
    --> pte index:0xd [decimal 13] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 17f5:
  --> pde index:0x5 [decimal 5] pde contents:0xd4 (valid 1, pfn 0x54 [decimal 84])
    --> pte index:0x1f [decimal 31] pte contents:0xce (valid 1, pfn 0x4e [decimal 78])
      --> Translates to Physical Address 0x9d5 --> Value: 1c
Virtual Address 7f6c:
  --> pde index:0x1f [decimal 31] pde contents:0xff (valid 1, pfn 0x7f [decimal 127])
    --> pte index:0x1b [decimal 27] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0bad:
  --> pde index:0x2 [decimal 2] pde contents:0xe0 (valid 1, pfn 0x60 [decimal 96])
    --> pte index:0x1d [decimal 29] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 6d60:
  --> pde index:0x1b [decimal 27] pde contents:0xc2 (valid 1, pfn 0x42 [decimal 66])
    --> pte index:0xb [decimal 11] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 2a5b:
  --> pde index:0xa [decimal 10] pde contents:0xd5 (valid 1, pfn 0x55 [decimal 85])
    --> pte index:0x12 [decimal 18] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 4c5e:
  --> pde index:0x13 [decimal 19] pde contents:0xf8 (valid 1, pfn 0x78 [decimal 120])
    --> pte index:0x2 [decimal 2] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
```

```
      --> Fault (page table entry not valid)
Virtual Address 2592:
  --> pde index:0x9 [decimal 9] pde contents:0x9e (valid 1, pfn 0x1e [decimal 30])
    --> pte index:0xc [decimal 12] pte contents:0xbd (valid 1, pfn 0x3d [decimal 61])
      --> Translates to Physical Address 0x7b2 --> Value: 1b
Virtual Address 3e99:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
    --> pte index:0x14 [decimal 20] pte contents:0xca (valid 1, pfn 0x4a [decimal 74])
      --> Translates to Physical Address 0x959 --> Value: 1e
```

For example:

```
Virtual Address 611c:
  --> pde index:0x18 [decimal 24] pde contents:0xa1 (valid 1, pfn 0x21 [decimal 33])
    --> pte index:0x8 [decimal 8] pte contents:0xb5 (valid 1, pfn 0x35 [decimal 53])
      --> Translates to Physical Address 0x6bc --> Value: 08
```

Our page size is 32 bytes.
The page table is 32 KB, which can divided into 256 * 4 * 32 / 32 = 1024 32-byte pages.
There are 1024 entries,
2^10 = 1024. The VPN holds 10 bits. 5 bits is offset.

Virtual Address 611c: 24860(decimal) 11000(PDE = 24th) 01000(PTE = 8th) 11100 (binary)

Because the PDBR is 108, which is:
page 108:83(0th) fe e0 da 7f d4 7f eb be 9e d5 ad e4 ac 90 d6 92 d8 c1 f8 9f e1 ed e9 a1(24th) e8 c7 c2 a9 d1 db ff

The PDE index should be the 24th byte, which is a1.
0xa1 is 1(valid)0100001 (binary)
0100001 is 33 (decimal)

Let’s check page33:
page 33:7f(0th) 7f 7f 7f 7f 7f 7f 7f b5(8th) 7f 9d 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f f6 b1 7f 7f 7f 7f
0xb5 is 1(valid)0110101 (binary)
0110101 is 53
11100 is 28

Thus the physical address is 011010111100, which is 0x6bc

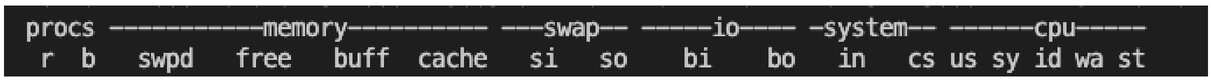
page 53: 0f 0c 18 09 0e 12 1c 0f 08 17 13 07 1c 1e 19 1b 09 16 1b 15 0e 03 0d 12 1c 1d 0e 1a 08(28th) 18 11 00

Thus the value is 08.

3. Given your understanding of how cache memory works, how do you think memory references to the page table will behave in the cache? Will they lead to lots of cache hits (and thus fast accesses?) Or lots of misses (and thus slow accesses)?

The addresses in the simulations are random cases, and there miss rate is high. In gernel usages, because the processes are expected to run frequently, TLB should be hitted in most cases. The top-level page directory tables fits temporal locality and the page entry table fits spatial locality.

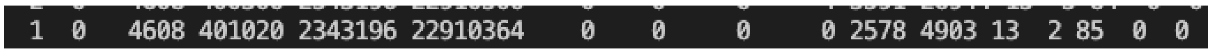
CH21:
1. First, open two separate terminal connections to the same machine, so that you can easily run something in one window and the other.
Now, in one window, run vmstat 1, which shows statistics about machine usage every second. Read the man page, the associated README, and any other information you need so that you can understand its output. Leave this window running vmstat for the rest of the exercises below. Now, we will run the program mem.c but with very little memory usage. This can be accomplished by typing ./mem 1 (which uses only 1 MB of memory). How do the CPU usage statistics change when running mem? Do the numbers in the user time column make sense? How does this change when running more than one instance of mem at once?



```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
```

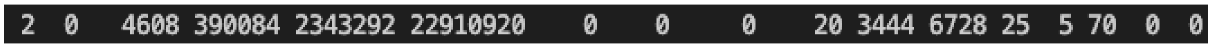
us: show the percentage of CPU cycles spent on user processes

When running 1 ./mem 1, us goes up to 13



```
 1  0   4608 401020 2343196 22910364    0    0     0     0 2578 4903 13  2 85  0  0
```

When running 2 ./mem 1, us goes up to 25

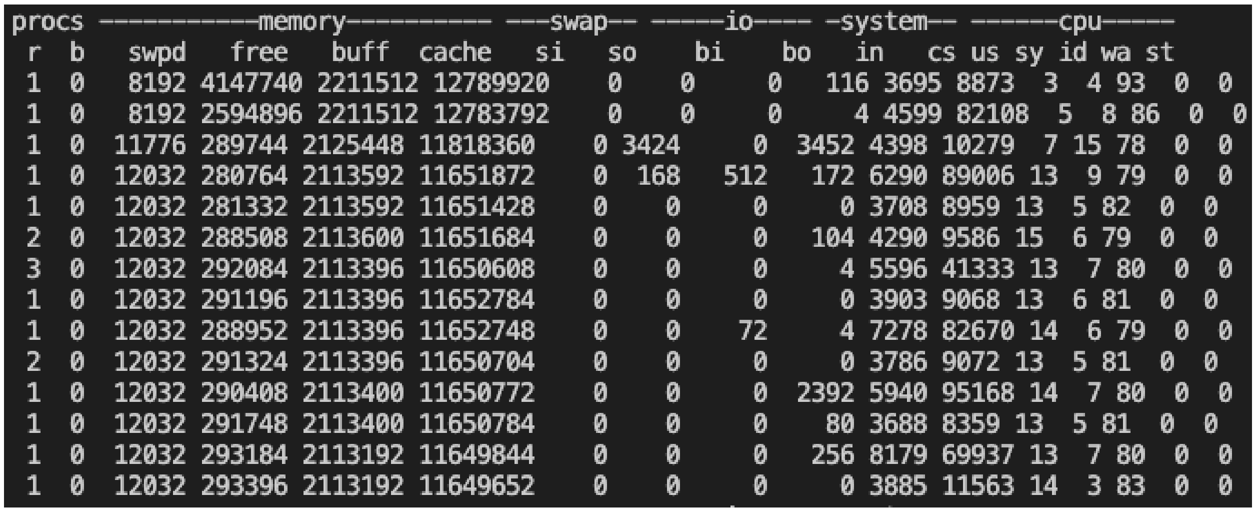


```
 2  0   4608 390084 2343292 22910920    0    0     0    20 3444 6728 25  5 70  0  0
```

2. Let’snowstartlookingatsomeofthememorystatisticswhilerunningmem. We’ll focus on two columns: swpd (the amount of virtual memory used) and free (the amount of idle memory). Run ./mem 1024 (which allocates 1024 MB) and watch how these values change. Then kill the running program (by typing control-c) and watch again how the values change. What do you notice about the values? In particular, how does the free column change when the program exits? Does the amount of free memory increase by the expected amount when mem exits?

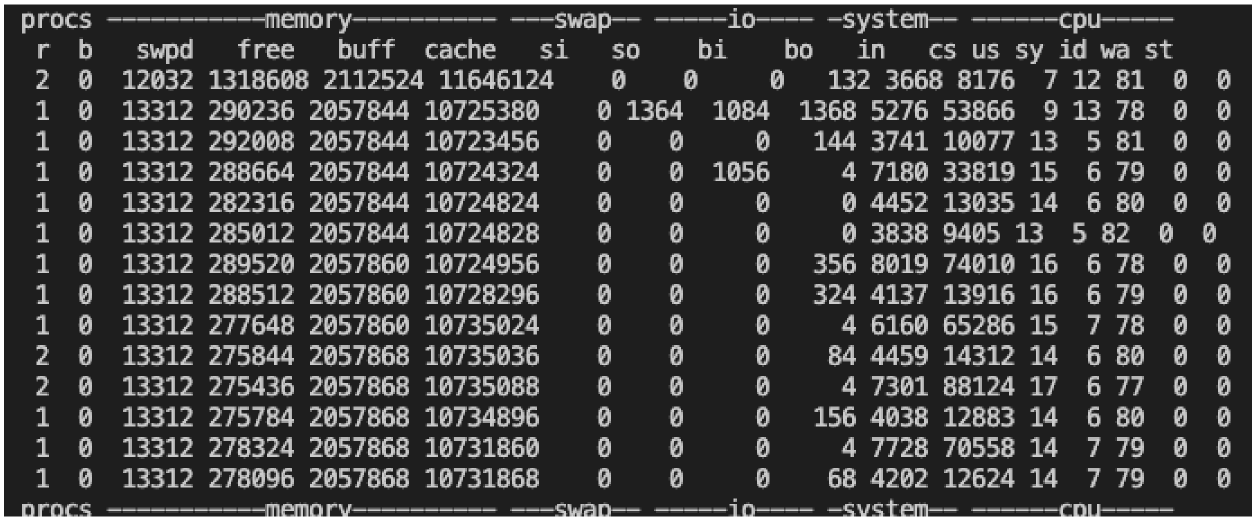
Run vmstat 1 --unit M, then it will use MB as the unit
When running ./mem 1024, the free column is around 183000
After killing ./mem 1024, the free column is around 184000
Does the amount of free memory increase by the expected amount when mem exits? Yes

3. We’llnextlookattheswapcolumns(siandso),whichindicatehowmuch swapping is taking place to and from the disk. Of course, to activate these, you’ll need to run mem with large amounts of memory. First, examine how much free memory is on your Linux system (for example, by typing cat /proc/meminfo; type man proc for details on the /proc file system and the types of information you can find there). One of the first entries in /proc/meminfo is the total amount of memory in your system. Let’s as- sume it’s something like 8 GB of memory; if so, start by running mem 4000 (about 4 GB) and watching the swap in/out columns. Do they ever give non-zero values? Then, try with 5000, 6000, etc. What happens to these values as the program enters the second loop (and beyond), as compared to the first loop? How much data (total) are swapped in and out during the second, third, and subsequent loops? (do the numbers make sense?)
./mem 5000



```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0   8192 4147740 2211512 12789920    0    0     0   116 3695 8873  3  4 93  0  0
 1  0   8192 2594896 2211512 12783792    0    0     0     4 4599 82108  5  8 86  0  0
 1  0  11776 289744 2125448 11818360    0 3424     0  3452 4398 10279  7 15 78  0  0
 1  0  12032 280764 2113592 11651872    0  168   512   172 6290 89006 13  9 79  0  0
 1  0  12032 281332 2113592 11651428    0    0     0     0 3708 8959 13  5 82  0  0
 2  0  12032 288508 2113600 11651684    0    0     0   104 4290 9586 15  6 79  0  0
 3  0  12032 292084 2113396 11650608    0    0     0     4 5596 41333 13  7 80  0  0
 1  0  12032 291196 2113396 11652784    0    0     0     0 3903 9068 13  6 81  0  0
 1  0  12032 288952 2113396 11652748    0    0    72     4 7278 82670 14  6 79  0  0
 2  0  12032 291324 2113396 11650704    0    0     0     0 3786 9072 13  5 81  0  0
 1  0  12032 290408 2113400 11650772    0    0     0  2392 5940 95168 14  7 80  0  0
 1  0  12032 291748 2113400 11650784    0    0     0    80 3688 8359 13  5 81  0  0
 1  0  12032 293184 2113192 11649844    0    0     0   256 8179 69937 13  7 80  0  0
 1  0  12032 293396 2113192 11649652    0    0     0     0 3885 11563 14  3 83  0  0
```

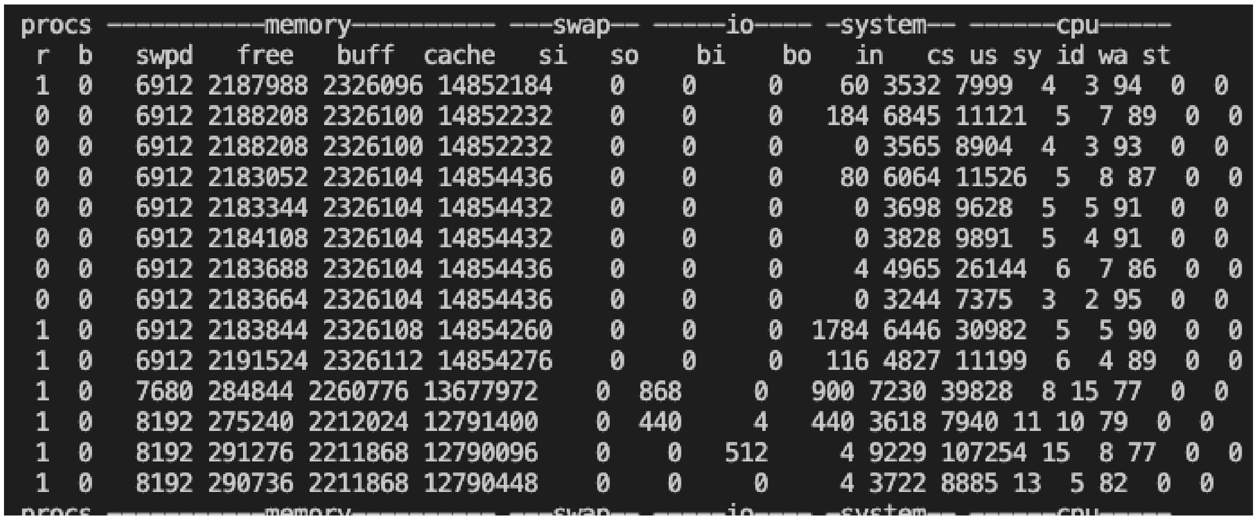
./mem 6000



```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  0  12032 1318608 2112524 11646124    0    0     0   132 3668 8176  7 12 81  0  0
 1  0  13312 290236 2057844 10725380    0 1364  1084  1368 5276 53866  9 13 78  0  0
 1  0  13312 292008 2057844 10723456    0    0     0   144 3741 10077 13  5 81  0  0
 1  0  13312 288664 2057844 10724324    0    0  1056     4 7180 33819 15  6 79  0  0
 1  0  13312 282316 2057844 10724824    0    0     0     0 4452 13035 14  6 80  0  0
 1  0  13312 285012 2057844 10724828    0    0     0     0 3838 9405 13  5 82  0  0
 1  0  13312 289520 2057860 10724956    0    0     0   356 8019 74010 16  6 78  0  0
 1  0  13312 288512 2057860 10728296    0    0     0   324 4137 13916 16  6 79  0  0
 1  0  13312 277648 2057860 10735024    0    0     0     4 6160 65286 15  7 78  0  0
 2  0  13312 275844 2057868 10735036    0    0     0    84 4459 14312 14  6 80  0  0
 2  0  13312 275436 2057868 10735088    0    0     0     4 7301 88124 17  6 77  0  0
 1  0  13312 275784 2057868 10734896    0    0     0   156 4038 12883 14  6 80  0  0
 1  0  13312 278324 2057868 10731860    0    0     0     4 7728 70558 14  7 79  0  0
 1  0  13312 278096 2057868 10731868    0    0     0    68 4202 12624 14  7 79  0  0
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
```

My machine has 32G memory. The amount of memory swapped to disk increased in the first loop and sometimes the second loop, then become zero.

4. Do the same experiments as above, but now watch the other statistics(such as CPU utilization, and block I/O statistics). How do they change when mem is running?



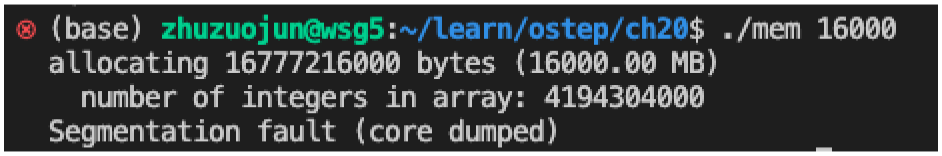
```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0   6912 2187988 2326096 14852184    0    0     0    60 3532 7999  4  3 94  0  0
 0  0   6912 2188208 2326100 14852232    0    0     0   184 6845 11121  5  7 89  0  0
 0  0   6912 2188208 2326100 14852232    0    0     0     0 3565 8904  4  3 93  0  0
 0  0   6912 2183052 2326104 14854436    0    0     0    80 6064 11526  5  8 87  0  0
 0  0   6912 2183344 2326104 14854432    0    0     0     0 3698 9628  5  5 91  0  0
 0  0   6912 2184108 2326104 14854432    0    0     0     0 3828 9891  5  4 91  0  0
 0  0   6912 2183688 2326104 14854436    0    0     0     4 4965 26144  6  7 86  0  0
 0  0   6912 2183664 2326104 14854436    0    0     0     0 3244 7375  3  2 95  0  0
 1  0   6912 2183844 2326108 14854260    0    0     0  1784 6446 30982  5  5 90  0  0
 1  0   6912 2191524 2326112 14854276    0    0     0   116 4827 11199  6  4 89  0  0
 1  0   7680 284844 2260776 13677972    0  868     0   900 7230 39828  8 15 77  0  0
 1  0   8192 275240 2212024 12791400    0  440     4   440 3618 7940 11 10 79  0  0
 1  0   8192 291276 2211868 12790096    0    0   512     4 9229 107254 15  8 77  0  0
 1  0   8192 290736 2211868 12790448    0    0     0     4 3722 8885 13  5 82  0  0
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
```

We can see the user time(us) and Blocks received from a block device (blocks/s) (bi) and Blocks sent to a block device (blocks/s) (bo) are increased.

5. Now let’s examine performance. Pick an input for mem that comfortably fits in memory (say 4000 if the amount of memory on the system is 8 GB). How long does loop 0 take (and subsequent loops 1, 2, etc.)? Now pick a size comfortably beyond the size of memory (say 12000 again assuming 8 GB of memory). How long do the loops take here? How do the bandwidth numbers compare? How different is performance when constantly swapping versus

fitting everything comfortably in memory? Can you make a graph, with the size of memory used by mem on the x-axis, and the bandwidth of accessing said memory on the y-axis? Finally, how does the performance of the first loop compare to that of subsequent loops, for both the case where everything fits in memory and where it doesn't?.

My machine has 32G memory. Since my machine has a large memory, if I input a larger parameter 16000(4000 * 4). The program can not malloc such big address. The program always crash when I run it.



```
⊗ (base) zhuzuojun@wsg5:~/learn/ostep/ch20$ ./mem 16000
  allocating 16777216000 bytes (16000.00 MB)
    number of integers in array: 4194304000
  Segmentation fault (core dumped)
```

So I changed to another server with the memory of 4G. Run ./mem 2000

```
bash-4.2$ ./mem 2000
allocating 2097152000 bytes (2000.00 MB)
  number of integers in array: 524288000
loop 0 in 1649.30 ms (bandwidth: 1212.63 MB/s)
loop 1 in 1231.56 ms (bandwidth: 1623.95 MB/s)
loop 2 in 1261.97 ms (bandwidth: 1584.82 MB/s)
loop 3 in 1236.88 ms (bandwidth: 1616.98 MB/s)
loop 4 in 1240.88 ms (bandwidth: 1611.76 MB/s)
loop 5 in 1247.62 ms (bandwidth: 1603.05 MB/s)
loop 6 in 1261.99 ms (bandwidth: 1584.79 MB/s)
loop 7 in 1237.08 ms (bandwidth: 1616.70 MB/s)
loop 8 in 1233.93 ms (bandwidth: 1620.84 MB/s)
loop 9 in 1241.38 ms (bandwidth: 1611.11 MB/s)
loop 10 in 1267.11 ms (bandwidth: 1578.39 MB/s)
```

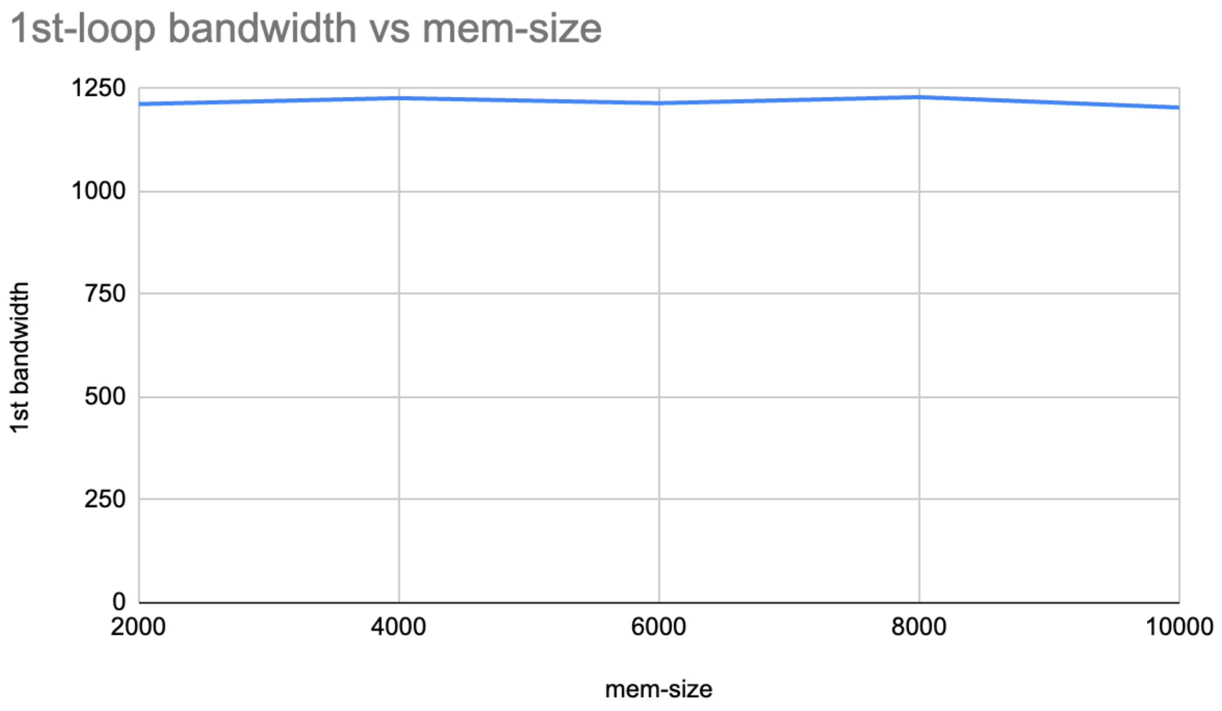
Loop 0 took 1649.3 ms, Loop 1 took 1231.56ms and Loop 2 took 1261.97ms. Bandwidth became larger as more loops.
Free is arroud 182000.

./mem 6000

```
bash-4.2$ ./mem 6000
allocating 6291456000 bytes (6000.00 MB)
  number of integers in array: 1572864000
loop 0 in 5046.42 ms (bandwidth: 1188.96 MB/s)
loop 1 in 4064.37 ms (bandwidth: 1476.25 MB/s)
loop 2 in 3820.89 ms (bandwidth: 1570.31 MB/s)
loop 3 in 3835.69 ms (bandwidth: 1564.26 MB/s)
loop 4 in 3857.22 ms (bandwidth: 1555.52 MB/s)
loop 5 in 3837.41 ms (bandwidth: 1563.55 MB/s)
loop 6 in 3842.00 ms (bandwidth: 1561.69 MB/s)
loop 7 in 3867.54 ms (bandwidth: 1551.37 MB/s)
loop 8 in 3882.39 ms (bandwidth: 1545.44 MB/s)
loop 9 in 3846.00 ms (bandwidth: 1560.06 MB/s)
loop 10 in 3850.11 ms (bandwidth: 1558.40 MB/s)
```

Loops took more time and bandwidth is a little shorted.
Free is arround 178000, which is smaller than the situation of smaller mem.

The first loop usually took more time and the bandwidth is smaller compared with the subsequent case. Because the first loop is usually miss in TLB.



6. Swap space isn't infinite. You can use the tool swapon with the -s flag to see how much swap space is available. What happens if you try to run mem with increasingly large values, beyond what seems to be available in swap? At what point does the memory allocation fail?

```
./mem 2000
bash-4.2$ swapon -s
Filename                    Type         Size    Used    Priority
/dev/dm-1                   partition    4194300 862236  -2
```

When it hits to ./mem 190000. The memory allocation would fail.

```
bash-4.2$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  0 878876 1403160   6284 174447376    0    0     0     0    0    0  3  1 96  0  0

bash-4.2$ swapon -s
Filename                    Type         Size    Used    Priority
/dev/dm-1                   partition    4194300 878876  -2
```

7. Finally, if you're advanced, you can configure your system to use different swap devices using swapon and swapoff. Read the man pages for details. If you have access to different hardware, see how the performance of swapping changes when swapping to a classic hard drive, a flash-based SSD, and even a RAID array. How much can swapping performance be improved via newer devices? How close can you get to in-memory performance?

Flash-based SSD would run faster with a smaller memory and a classic hard drive would run slower with a larger available memory.

CH22:
1. Generate random addresses with the following arguments: -s 0 -n 10, -s 1 -n 10, and -s 2 -n 10. Change the policy from FIFO, to LRU, toOPT. Computewhether each access in said address traces are hits or misses.

```
./paging-policy.py -s 0 -n 10 -c
1e18@Zuojuns-MacBook-Air vm-beyondphys-policy % ./paging-policy.py -s 0 -n 10 -c
ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy FIFO
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 0
ARG notrace False

Solving...

Access: 8  MISS FirstIn ->          [8] <- Lastin  Replaced:- [Hits:0 Misses:1]
Access: 7  MISS FirstIn ->       [8, 7] <- Lastin  Replaced:- [Hits:0 Misses:2]
Access: 4  MISS FirstIn ->    [8, 7, 4] <- Lastin  Replaced:- [Hits:0 Misses:3]
Access: 2  MISS FirstIn ->    [7, 4, 2] <- Lastin  Replaced:8 [Hits:0 Misses:4]
Access: 5  MISS FirstIn ->    [4, 2, 5] <- Lastin  Replaced:7 [Hits:0 Misses:5]
Access: 4  HIT  FirstIn ->    [4, 2, 5] <- Lastin  Replaced:- [Hits:1 Misses:5]
Access: 7  MISS FirstIn ->    [2, 5, 7] <- Lastin  Replaced:4 [Hits:1 Misses:6]
Access: 3  MISS FirstIn ->    [5, 7, 3] <- Lastin  Replaced:2 [Hits:1 Misses:7]
Access: 4  MISS FirstIn ->    [7, 3, 4] <- Lastin  Replaced:5 [Hits:1 Misses:8]
Access: 5  MISS FirstIn ->    [3, 4, 5] <- Lastin  Replaced:7 [Hits:1 Misses:9]
FINALSTATS hits 1   misses 9   hitrate 10.00

./paging-policy.py -s 0 -n 10 -c --policy=LRU
1e18@Zuojuns-MacBook-Air vm-beyondphys-policy % ./paging-policy.py -s 0 -n 10 -c
--policy=LRU
ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy LRU
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 0
ARG notrace False

Solving...

Access: 8  MISS LRU ->          [8] <- MRU Replaced:- [Hits:0 Misses:1]
Access: 7  MISS LRU ->       [8, 7] <- MRU Replaced:- [Hits:0 Misses:2]
Access: 4  MISS LRU ->    [8, 7, 4] <- MRU Replaced:- [Hits:0 Misses:3]
Access: 2  MISS LRU ->    [7, 4, 2] <- MRU Replaced:8 [Hits:0 Misses:4]
Access: 5  MISS LRU ->    [4, 2, 5] <- MRU Replaced:7 [Hits:0 Misses:5]
Access: 4  HIT  LRU ->    [2, 5, 4] <- MRU Replaced:- [Hits:1 Misses:5]
Access: 7  MISS LRU ->    [5, 4, 7] <- MRU Replaced:2 [Hits:1 Misses:6]
Access: 3  MISS LRU ->    [4, 7, 3] <- MRU Replaced:5 [Hits:1 Misses:7]
Access: 4  HIT  LRU ->    [7, 3, 4] <- MRU Replaced:- [Hits:2 Misses:7]
```

```
Access: 5  MISS LRU ->    [3, 4, 5] <- MRU Replaced:7 [Hits:2 Misses:8]

FINALSTATS hits 2   misses 8   hitrate 20.00
```

```
./paging-policy.py -s 0 -n 10 -c --policy=OPT

ARG addresses -1
ARG addressfile
ARG numaddrs 10
ARG policy OPT
ARG clockbits 2
ARG cachesize 3
ARG maxpage 10
ARG seed 0
ARG notrace False

Solving...

Access: 8  MISS Left  ->          [8] <- Right Replaced:- [Hits:0 Misses:1]
Access: 7  MISS Left  ->       [8, 7] <- Right Replaced:- [Hits:0 Misses:2]
Access: 4  MISS Left  ->    [8, 7, 4] <- Right Replaced:- [Hits:0 Misses:3]
Access: 2  MISS Left  ->    [7, 4, 2] <- Right Replaced:8 [Hits:0 Misses:4]
Access: 5  MISS Left  ->    [7, 4, 5] <- Right Replaced:2 [Hits:0 Misses:5]
Access: 4  HIT  Left  ->    [7, 4, 5] <- Right Replaced:- [Hits:1 Misses:5]
Access: 7  HIT  Left  ->    [7, 4, 5] <- Right Replaced:- [Hits:2 Misses:5]
Access: 3  MISS Left  ->    [4, 5, 3] <- Right Replaced:7 [Hits:2 Misses:6]
Access: 4  HIT  Left  ->    [4, 5, 3] <- Right Replaced:- [Hits:3 Misses:6]
Access: 5  HIT  Left  ->    [4, 5, 3] <- Right Replaced:- [Hits:4 Misses:6]

FINALSTATS hits 4   misses 6   hitrate 40.00
```

2. For a cache of size 5, generate worst-case address reference streams for each of the following policies: FIFO, LRU, andMRU (worst-case reference streams cause themostmisses possible. For theworst case reference streams, howmuch bigger of a cache is needed to improve performance dramatically and approach OPT?

```
./paging-policy.py --addresses=0,1,2,3,4,5,0,1,2,3,4,5 --policy=FIFO --cachesize=5 -c
./paging-policy.py --addresses=0,1,2,3,4,5,0,1,2,3,4,5 --policy=LRU --cachesize=5 -c
./paging-policy.py --addresses=0,1,2,3,4,5,4,5,4,5,4,5 --policy=MRU --cachesize=5 -c
```

We only need one cache.

For the work processes : 0,1,2,3,4,5,0,1,2,3,4,5. The cache size should be 6 that can fit the workflow better.

3. Generate a random trace (use python or perl). How would you expect the different policies to perform on such a trace?

```
./paging-policy.py -s 0 -n 10 -c
FINALSTATS hits 1   misses 9   hitrate 10.00

./paging-policy.py -s 0 -n 10 -c --policy=LRU
FINALSTATS hits 2   misses 8   hitrate 20.00

./paging-policy.py -s 0 -n 10 -c --policy=OPT
FINALSTATS hits 4   misses 6   hitrate 40.00

./paging-policy.py -s 0 -n 10 -c --policy=UNOPT
FINALSTATS hits 0   misses 10   hitrate 0.00

./paging-policy.py -s 0 -n 10 -c --policy=RAND
FINALSTATS hits 0   misses 10   hitrate 0.00

./paging-policy.py -s 0 -n 10 -c --policy=CLOCK
FINALSTATS hits 1   misses 9   hitrate 10.00
```

OPT has the best performance.

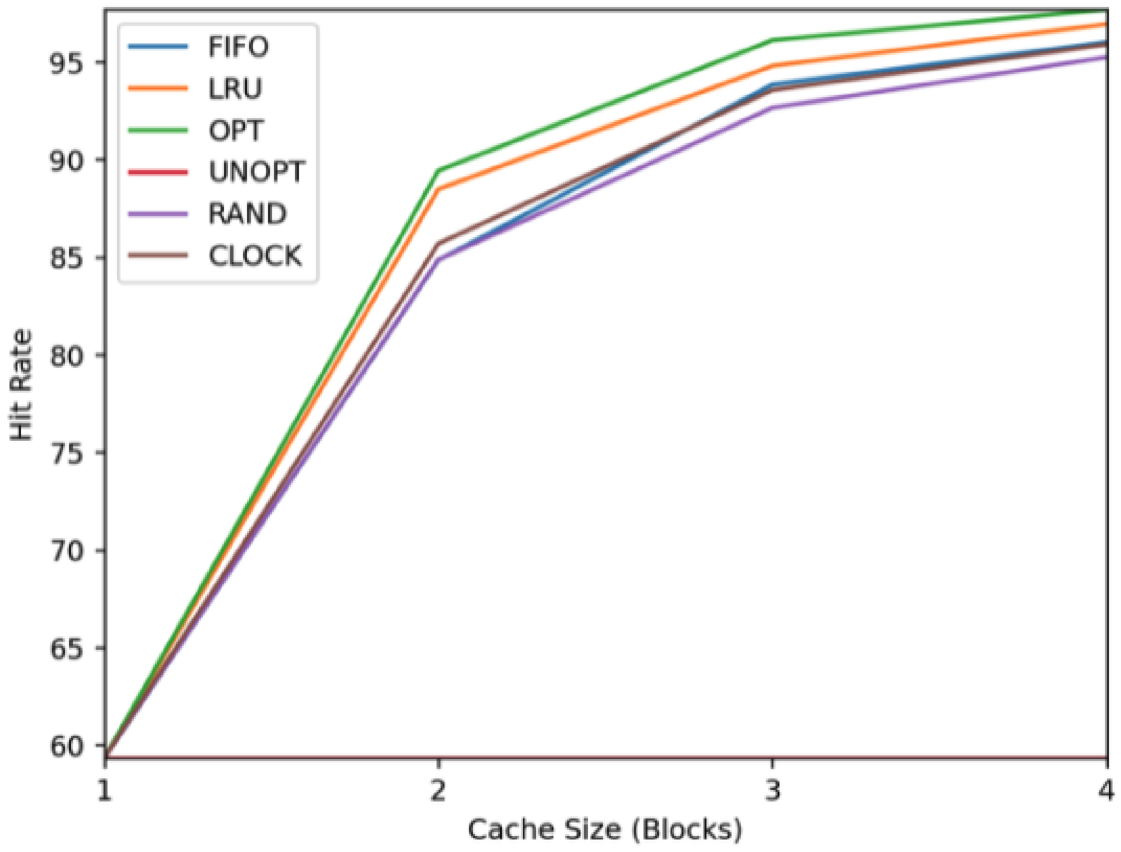
4. Now generate a trace with some locality. How can you generate such a trace? How does LRU perform on it? How much better than RAND is LRU? How does CLOCK do? How about CLOCK with different numbers of clock bits?
I generate a trace which is 3,0,6,6,6,6,7,0,6,6

```
$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=LRU -c
FINALSTATS hits 6   misses 4   hitrate 60.00

$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=RAND -c
FINALSTATS hits 5   misses 5   hitrate 50.00

$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=CLOCK -c -b 2
Access: 3  MISS Left  ->          [3] <- Right Replaced:- [Hits:0 Misses:1]
Access: 0  MISS Left  ->       [3, 0] <- Right Replaced:- [Hits:0 Misses:2]
Access: 6  MISS Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:0 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:1 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:2 Misses:3]
```

```
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:3 Misses:3]
Access: 7  MISS Left  ->    [3, 6, 7] <- Right Replaced:0 [Hits:3 Misses:4]
Access: 0  MISS Left  ->    [3, 7, 0] <- Right Replaced:6 [Hits:3 Misses:5]
Access: 6  MISS Left  ->    [7, 0, 6] <- Right Replaced:3 [Hits:3 Misses:6]
Access: 6  HIT  Left  ->    [7, 0, 6] <- Right Replaced:- [Hits:4 Misses:6]
FINALSTATS hits 4   misses 6   hitrate 40.00

$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=CLOCK -c -b 0
Access: 3  MISS Left  ->          [3] <- Right Replaced:- [Hits:0 Misses:1]
Access: 0  MISS Left  ->       [3, 0] <- Right Replaced:- [Hits:0 Misses:2]
Access: 6  MISS Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:0 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:1 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:2 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:3 Misses:3]
Access: 7  MISS Left  ->    [3, 0, 7] <- Right Replaced:6 [Hits:3 Misses:4]
Access: 0  HIT  Left  ->    [3, 0, 7] <- Right Replaced:- [Hits:4 Misses:4]
Access: 6  MISS Left  ->    [3, 7, 6] <- Right Replaced:0 [Hits:4 Misses:5]
Access: 6  HIT  Left  ->    [3, 7, 6] <- Right Replaced:- [Hits:5 Misses:5]
FINALSTATS hits 5   misses 5   hitrate 50.00

$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=CLOCK -c -b 1
Access: 3  MISS Left  ->          [3] <- Right Replaced:- [Hits:0 Misses:1]
Access: 0  MISS Left  ->       [3, 0] <- Right Replaced:- [Hits:0 Misses:2]
Access: 6  MISS Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:0 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:1 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:2 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:3 Misses:3]
Access: 7  MISS Left  ->    [3, 0, 7] <- Right Replaced:6 [Hits:3 Misses:4]
Access: 0  HIT  Left  ->    [3, 0, 7] <- Right Replaced:- [Hits:4 Misses:4]
Access: 6  MISS Left  ->    [3, 7, 6] <- Right Replaced:0 [Hits:4 Misses:5]
Access: 6  HIT  Left  ->    [3, 7, 6] <- Right Replaced:- [Hits:5 Misses:5]
FINALSTATS hits 5   misses 5   hitrate 50.00

$ ./paging-policy.py --addresses=3,0,6,6,6,6,7,0,6,6 --policy=CLOCK -c -b 3
Access: 0  MISS Left  ->       [3, 0] <- Right Replaced:- [Hits:0 Misses:2]
Access: 6  MISS Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:0 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:1 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:2 Misses:3]
Access: 6  HIT  Left  ->    [3, 0, 6] <- Right Replaced:- [Hits:3 Misses:3]
Access: 7  MISS Left  ->    [3, 6, 7] <- Right Replaced:0 [Hits:3 Misses:4]
Access: 0  MISS Left  ->    [6, 7, 0] <- Right Replaced:3 [Hits:3 Misses:5]
Access: 6  HIT  Left  ->    [6, 7, 0] <- Right Replaced:- [Hits:4 Misses:5]
Access: 6  HIT  Left  ->    [6, 7, 0] <- Right Replaced:- [Hits:5 Misses:5]
FINALSTATS hits 5   misses 5   hitrate 50.00
```

The result of CLOCK doesn't matter with the CLOCK BITS in my case.

5. Use a program like valgrind to instrument a real application and generate a virtual page reference stream. For example, running valgrind --tool=lackey --trace-mem=yes ls will output a nearly-complete reference trace of every instruction and data reference made by the program ls. To make this useful for the simulator above, you'll have to first transform each virtual memory reference into a virtual page-number reference (done by masking off the offset and shifting the resulting bits downward). How big of a cache is needed for your application trace in order to satisfy a large fraction of requests? Plot a graph of its working set as the size of the cache increases.



4 caches would be quite good.