#uni/semester3/Betriebssysteme/chapter22/homework

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, to OPT. Compute whether each access in said address traces are hits or misses.
   ➜ See Ordner

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

   Worst case FIFO:

```
./paging-policy.py -a 0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9 -c -p FIFO -C 5
```

   ➜ to approach OPT the cache size must be increased by 5, so it is 10.
   ➜ Number of unique pages = 10

   Worst case LRU:

```
./paging-policy.py -a 0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9,0,1,2,3,4,5,6,7,8,9,1,2,3,4,5,6,7,8,9 -c -p LRU -C 5
```

   ➜ to approach OPT the cache size must be increased by 5, so it is 10.
   ➜ Number of unique pages = 10

   Worst case MRU:

```
./paging-policy.py -a 0,1,2,3,4,5,4,5,4,5,4,5  -c -p MRU -C 5
```

   ➜ Worst case when you have (cache size + 1) unique pages and then you oscillate between two

- ➡ Increasing the cache size by one solves the problem
- ➡ New cache size is 6, which approaches OPT

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

```
./paging-policy.py -a 3, 1, 6, 0, 5, 3, 0, 5, 0, 4 -c -p FIFO -C 5
./paging-policy.py -a 3, 1, 6, 0, 5, 3, 0, 5, 0, 4 -c -p LRU -C 5
./paging-policy.py -a 3, 1, 6, 0, 5, 3, 0, 5, 0, 4 -c -p RAND -C 5
./paging-policy.py -a 3, 1, 6, 0, 5, 3, 0, 5, 0, 4 -c -p MRU -C 5
./paging-policy.py -a 3, 1, 6, 0, 5, 3, 0, 5, 0, 4 -c -p OPT -C 5
```

   - ➡ I expect all of them to perform roughly the same. With OPT being the best
   - ➡ They all get a hit rate of 40%

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

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

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

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

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

./paging-policy.py -s 0 -n 10 -c -p MRU
FINALSTATS hits 2   misses 8   hitrate 20.00
```

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?

➡ run generate-locality.py

```
./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p FIFO
FINALSTATS hits 5   misses 5   hitrate 50.00

./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p LRU
FINALSTATS hits 6   misses 4   hitrate 60.00

./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p RAND
FINALSTATS hits 6   misses 4   hitrate 60.00

./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p CLOCK -b 0
FINALSTATS hits 6   misses 4   hitrate 60.00

./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p CLOCK -b 1
FINALSTATS hits 6   misses 4   hitrate 60.00

./paging-policy.py -a 9,9,1,9,5,5,9,9,8,9 -c -p CLOCK -b 2
FINALSTATS hits 6   misses 4   hitrate 60.00
```

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.

```
valgrind --tool=lackey --trace-mem=yes ls &> ref-trace.txt
```

➡ Wenn man ref-trace.txt analysiert, sieht man dass die ersten 5 hex Zahlen zur VPN gehören, und die letzten 3 zum offset