Operating Systems – spring 2024
Tutorial-Assignment 6

Instructor: Hans P. Reiser

Submission Deadline: Monday, February 26th, 2024 – 23:59

A new assignment will be published every week. It must be completed before its submission deadline (late policy for programming assignments: up to two days, 10% penalty/day)

Lab Exercisess are theory and programming exercises discussed in the lab class. They are not graded, but should help you solve the graded questions and prepare for the final exam. Make sure to read and think about possible solutions before the lab class.

T-Questions are theory homework assignments and need to be answered directly on Canvas (quiz).

P-Questions are programming assignments. Download the provided template from Canvas. Do not fiddle with the compiler flags. Submission instructions can be found in the introductory section below.

The topic of this assignment is a deeper dive into page fault handling and page replacement, to large part revisiting lecture material.

Lab 6.1: Hardware- vs. Software-Walked Page Tables

a. What's the difference between the use of hardware-walked page tables and software-walked page tables? How does this relate to TLBs?

b. What difference can you make out in the contents of software-walked vs. hardware-walked multi-level page tables?

c. Under what circumstances are TLB miss handlers or page fault handlers invoked?

Lab 6.2: Page Fault Handling

a. Explain the terms demand-paging and pre-paging. What are the respective strengths and weaknesses?

b. When a thread touches (either reads or writes) a page for the first time with demand-paging, a page fault will occur. Classify the page fault according to where the data for the unmapped page has to be fetched from by the page fault handler.

c. If the process has been running for some time, modifying data along its way, there is one additional case that needs to be covered on a page fault. Which?

d. Discuss which information is required by the page fault handler to correctly setup (or restore) the contents of accessed pages.

e. Can you reuse page table entries to store some of this information? Is it a good idea?

f. What is Copy-on-Write? How can it be implemented?

g. Recap: Describe the steps necessary to handle a page fault in an application's address space.

Lab 6.3: Page Replacement Basics

a. The pager (e.g., kswapd on Linux) of some systems tries to always offer a certain amount of free page frames to improve paging. What is the basic idea behind such a pager?

b. Describe the difference between a global and a local page replacement algorithm. Discuss the advantages and disadvantages of each of them.

c. Does a virtual memory system implementing equal allocation require a global or a local page replacement policy? Justify your answer.

d. What is thrashing? When does it occur?

e. What is the working set of a process? How can the working set be used to prevent thrashing?

Lab 6.4: Page Replacement Policies

a. A task has four page frames $(0, \ldots, 3)$ allocated to it. The virtual page number of each page frame, the time of the last loading of a page into each page frame, the time of the last access to the page frame, and the referenced (R) and modified (M) bits of each page frame are shown in the following table.

frame	virtual page	load time	access time	referenced	modified
0	2	60	161	0	1
1	1	130	160	0	0
2	0	26	162	1	0
3	3	20	163	1	1

A pagefault upon access to virtual page 4 occurs. Which page frame will have its contents replaced for the *FIFO*, *LRU*, *Clock* and *Optimal* (with respect to the number of page replacements) replacement policies?

For the Clock algorithm assume that you have a circular list of frames in the order 0, 1, 2, 3 and that the next-frame pointer refers to frame 2.

For the Optimal algorithm use the following string for subsequent references:
4, 0, 0, 0, 2, 4, 2, 1, 0, 3, 2.

b. For this part, you will be doing experiments with page replacement strategies using the simulator from the OSTEP book (see `https://github.com/xxyzz/ostep-hw/blob/master/22/paging-policy.py` for the "original", there is a version available on Canvas with minor adjustments for Python 3). Please note the terminology used in the script (see also book chapter 22): The main memory of a process is seen as a "cache" of the virtual memory of a process. Consequently, the number of physical pages mapped in an application's virtual address space is called the "cache size".

You can simulate the execution of an arbitrary access pattern using the following command: (example: access pattern 1,2,3,4,5,4,3,2,1 on a system with cache size 4 and policy FIFO):

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

- Find a single, arbitrary access pattern with 20 elements that has an worst-case performance on all of these scheduling policies: FIFO, LRU, CLOCK and OPT.
- For each of the three policies FIFO, LRU and MRU, find one access pattern **(reference string with 20 elements)** that uses **at most 5 different** virtual memory pages (cache size) and has a worst-case behaviour (maximizing page faults) on a system **with four physical frames.**

T-Question 6.1: Paging

a. Let's assume you have a system that uses page tables for virtual address spaces. Two processes, A and B, want to have region of shared memory in their address space. Which of the following statements are true? **2 T-pt**

true	false	
☐	☐	The page table entries for the shared memory region must use identical virtual page numbers at process A and B.
☐	☐	The PTEs for the shared memory have to be configured as read-only mapping.
☐	☐	The same physical page frame can be mapped at different addresses in the virtual address spaces of A and B.
☐	☐	Shared memory between A and B has to be implemented using copy-on-write (COW), enabling deduplication in case one process modifies the content of the shared memory.

b. Assume you want to implement shared data with private modifications: As long two processes use some original data, they should access the same physical memory (shared memory). As soon as one process modifies the data, that process will get its own, private copy of the data, and the modifications are performed on that private copy, not on the shared data. (For example, you can implement such strategy with the `MAP_PRIVATE` option of the `mmap` system call).

What page protection bit in a PTE is most suitable for implementing such a copy-on-write (COW) mechanism? (select exactly one true answer) **1 T-pt**

true	false	
☐	☐	Read only bit (Write bit)
☐	☐	Valid bit (Present bit)
☐	☐	Dirty bit.
☐	☐	Supervisor bit.

T-Question 6.2: Page Replacement

a. A page fault handler has to deal differently with two types of pages when it comes to eviction and loading of pages from/into physical memory:

- *named mappings* (memory mapped files, such as code of a program)
- *anonymous mappings* (page not backed by a file, such as heap and stack pages)

Which of the following statements is true? **2 T-pt**

true	false	
☐	☐	Named entries contain important data that always needs to be written back to disk before dropping the page.
☐	☐	Memory mapped files are always mapped read only, whereas anonymous mappings can be used for writable memory.
☐	☐	Anonymous mappings (with modifications) generally have to be written to a page file or swap device before eviction.
☐	☐	A page that was evicted from physical memory needs to be loaded to exactly the same physical memory address if a user program accesses that page again after the eviction.

b. Consider a system with a total of 4 page frames and an application accessing memory according to this reference string of virtual page numbers (VPNs): **4 1 2 7 3 5**. Complete the mapping table for a process at different times (t_0: initial mapping, t_1: mapping after accessing VPN 4, etc.) if **clock page replacement** is used. Assume that the circular buffer of the clock is in ascending order (i.e., frame 0, 1, 2, 3), that the clock hand at t_0 is positioned at **frame 0** and that the reference bit for **page 7** (in frame 2) is set, and cleared for the other pages. **3 T-pt**

frame	$VPN(t_0)$	$VPN(t_1)$	$VPN(t_2)$	$VPN(t_3)$	$VPN(t_4)$	$VPN(t_5)$	$VPN(t_6)$
0	3						
1	1						
2	7						
3	6						

c. Page faults for different algorithms **2 T-pt**

Note: The access sequence for this question was generated by the following command (see P part): `paging-policy.py -s 6 -n 15 -m 5`

Consider the page access sequence (reference string):

```
3 4 2 1 0 3 2 3 1 3 1 4 3 2 2
```

For each of the three policies FIFO, LRU and OPT, determine which of these access are page faults ("cache misses") in a system with a total of four physical frames ("cache size")! What is the total number of page faults?

P-Question 6.1: Page Replacement Simulation

For this assignment, there is no C programming required. Instead you will be doing experiments with page replacement strategies using the simulator from the OSTEP book (see `https://github.com/xxyzz/ostep-hw/blob/master/22/paging-policy.py` for the "original", there is a version available on Canvas/on noskel/in the template with minor adjustments for Python 3 and performance enhancements for the OPT simulation).

Please note the terminology used in the script – of which I am not a big fan: The main memory of a process is seen as a "cache" of the virtual memory of a process. Consequently, the number of physical pages mapped in an application's virtual address space is called the "cache size" (see also book chapter 22).

a. Preparation: Get real traces **0 P-pt**

For the following parts, you will need memory access traces of real applications. You can create those traces on your own using the lackey plugin of valgrind. Lets assume you have a program `main`. This command will create a trace (in `trace-main.txt`):
`valgrind --tool=lackey --trace-mem=yes ./main 2> trace-main.txt`

A (very short) sample trace is in the template. Find more traces on noskel.mooo.com in `/home/sty24/traces`. You should have at least 2 traces for testing.

Note: If your solution handles the large compiler trace for generating the memory access pattern graph (heapmap), you gain +1 bonus points (feasible). If you handle the large compiler trace in the last part (replacement simulation – you may skip "OPT") you gain +2 bonus points (I expect nobody to do this – this is hard).

Note 2: Another bonus point (+1) if you mark code pages (those where instructions are executed) in the memory access pattern with a different background color.

b. Paging policy simulation with real workloads **2 P-pt**

In the Python script `sim-paging.py`, implement the function `get_page_list` that transforms the valgrind/lackey output into a list of page numbers. The function takes a file name as input parameter. This file contains the valgrind/lackey output, which contains lines with type of access (I:code, L:load, S:store, M:modify), virtual address, and access size. You may ignore the size of the access. The traces are from an Intel x86-64 system, where the page size is 4096 bytes (use that to transform virtual addresses into virtual page numbers). Your function shall return a list of page numbers, in the order of access.

c. Visualizing memory access patterns **3 P-pt**

Implement the `plot_memory_access` function that creates an image similar to lecture part 8 slide 22. Suggested approach:

- You need to transform the page access list into a 2D array.
- For the X axis, use equal-sized bins into which you group subsets of the page access list. Example: page access list has 1M elements, you use 1000 bins, so each bin corresponds to 1000 elements of the page access list
- For the Y axis, you first "normalize" the page numbers (these are not contiguous, but it is easier to plot if you have a contiguous numbering). Define an order-preserving mapping from the virtual page numbers to natural numbers in the range 0 to (number of different pages - 1). Example: If your page access list contains (only) the page numbers 598, 4, and 42, then you map 4 to 0, 42 to 1, 598 to 2.
- You will probably have something in the order of a few hundred to a few thousand different pages. The range of your Y axis (and thus this dimension of the 2D array) will equal the number of pages you have.

- For each bin `i` (X axis), and for each page `k` contained in the per-bin subset of the page access list, you set your `array[i][k]` to 1 (all other array elements are 0)
- You plot the result with `np.imshow` (you need transpose the array). Arrange your plot such that page 0 is on the bottom, not on the top.
- Show your output on the screen by default, or write it to a png file if an optional parameter pngfile is passed to your function.
- A good plot has clearly and accurately labeled title and axes.

d. Preparing for the page replacement simulation Implement the `export_page_trace` function. This function shall do two things: **1 P-pt**

- To speed up the simulation, remove adjacent duplicated pages in your page access list. We assume that immediately after accessing a page, if the next access is to the same page, it will not be a page fault. So removing those duplicates does not change the total number of page faults (for FIFO, LRU, RAND, and OPT policies – would have an impact on LFU/MFU). So this is valid to compare different policies.
- Write the page list to a simple file (as expected by the simulation tool) with with one page number per line, in decimal.

e. Page replacement simulation **4 P-pt**

Use the trace from part (d) and the OSTEP simulator to calculate the number of page faults for several sizes of the physical memory, for each of the policies FIFO, LRU, RAND and OPT. Plot a graph with the results: X-axis: number of pages in physical memory (1..total number of accessed blocks), Y-axis: number of page faults, using a log scale.

For the X-axis, if a simulation for all memory sizes in above range is not feasible, select some strategy such that the interesting region where the number of page faults go from some minimum to 10x the minimum is clearly visible.

Submission: A single zip file with this content:

- groupinfo.txt (as in earlier assignments)
- sim-paging.py with implemented functions for parts (b), (c), (d)
- traces.txt A text file with the md5 hashes of the two (or more) traces you have used. Also point out if you want to claim any bonus points. (in case you generated your own trace: add a short explanation how you generated it, like the exact valgrind command)
- mempattern-1.png and mempattern-2.png: Your plots for part (c)
- simresult-1.png and simresult-2.png: for each of the two traces you have used: a single file with your results (all four policies plotted in a single diagram! use different colors for different policies).

Total:
10 T-pt
10 P-pt