Operating Systems – spring 2024
Tutorial-Assignment 7

Instructor: Hans P. Reiser

Submission Deadline: Monday, March 4th, 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 paging-based address translation, TLBs, and caches.

Lab 7.1: Caches

a. Describe the principle and the benefits of a memory hierarchy. How can a memory hierarchy provide both fast access and large capacity? On what typical program behavior does it depend?

b. Cache memory is divided into (and loaded in) blocks, also called cache lines. Why is a cache divided into these cache lines? What might limit the size of a cache line?

c. Enumerate and explain the different kinds of cache misses.

d. Explain the difference between the write-through and write-back strategy.

e. How can writes to a data item that is currently not in the cache be handled?

f. What are the two main problems that can arise with caches that are virtually-indexed and virtually-tagged? What can be done to avoid or solve these problems?

g. Does using virtually-indexed, physically-tagged caches solve the two problems described in the previous question?

Lab 7.2: Function Pointers in C

a. In C you can use function pointers to invoke functions (instead of invoking them directly by their name). We have seen one example of function pointers already before (in the pthreads part), but let's revisit them and practice how to declare function pointers. You will need function pointers in this week's assignment.

Simple example: Call hello() or goodby(), depending on the number of command line arguments. (This simplistic example is just good for illustrating the concept of function pointers)

Find the right way to declare and call a function via a function pointers (see comments in code!)

```
#include <stdio.h>

// These are the two function we will "point to"
void hello() {
        printf("Hello!\n");
}
void goodbye() {
        printf("Goodbye!\n");
}

int main(int argc, char *argv[]) {
        // Can you correctly declare "funcptr" as a pointer to a function that has no argument
        // and no return value (void)? The following "void *" is not the correct data type here!
        void *funcptr;

        if(argc<2) { funcptr = hello; }
        else { funcptr = goodbye; }

        // Let's call a function via the function pointer! Two of the three statements
        // below are correct. Can you find out which, and explain why?
        funcptr();
        *funcptr();
        (*funcptr)();
}
```

b. The second example is about how to declare a function pointer for functions that have parameters and/or return values. Can you add the correct type declaration to this code?

```
#include <stdio.h>
#include <stdlib.h>

// These are the two function we will "point to"
int add(int a, int b) {
        return a + b;
}
int sub(int a, int b) {
        return a - b;
}

int main(int argc, char *argv[]) {
        // Can you change this pointer declaration such that it accepts two int parameters
        // and returns an int?
        void (*funcptr)();

        if(argc<4) {
                printf("Usage: %s <operation> <value1> <value>\n"
                        "operation can be a (add) or s (subtract)\n", argv[0]);
                exit(1);
        }

        if(argv[1][0]=='a') { funcptr = add; }
        else { funcptr = sub; }

        int arg1 = atoi(argv[2]);
        int  arg2 = atoi(argv[3]);
        int result = funcptr( arg1, arg2 );
        printf("Result of %s(%d,%d) is %d\n",
                funcptr == add ? "add" : "sub", arg1, arg2, result);
}
```

T-Question 7.1: Paging

a. Consider a system that translates virtual addresses to physical addresses using hierarchical page tables. Every page table page comprises 512 entries, with each entry having a size of 8 bytes (all levels identical). The maximum size of the virtual address spaces is 512 GiB. The page size is 4096 bytes.

   How many levels L do we need in a hierarchical (multi-level) page table that can map the complete virtual address space to physical addresses?

   At each level, what is the maximum of page table pages that we can have?

   Note: Level 1 designates the lowest level, Level L the highest level. For example, standard 32 bit Intel x86 has one page table page at L2 (="Page Directory") and (up to) 1024 page table pages at L1 (="Page Table") **2 T-pt**

b. In the lecture (Chapter 8, slide 30) two programs showed significantly different performance when used with standard 4 KiB pages, primarily due to TLB misses. Note that also P6's trace-stydemo was from one of these two programs (the one on the left), and the memory trace of the one on the right was shown in the lecture last week.

   Modern Intel x86 CPUs support page tables with large pages, skipping L1 and directly storing the physical address of the page frame at L2.

   For this specific application, analyze whether using large pages can yield:

   (A) Advantages for the two programs?
   Explain which program might benefit and provide justifications.

   (B) Disadvantages for the two programs?
   Explain which program might face drawbacks and provide justifications.

   (C) Estimate the number of TLB misses:
   Assuming no preemption by the scheduler during execution, predict how many TLB misses you expect for each of the two programs. **3 T-pt**

c. What difficulties exist when implementing the CLOCK algorithm in a virtual memory system that uses paging with hardware support that sets a bit in the page table entry (PTE) when a page is accessed, and supports shared memory?

   (Assume standard multi-level page tables) **2 T-pt**

T-Question 7.2: CPU Caches

a. Pick the (most) correct choices for the following statements about physically-tagged CPU caches: **2 T-pt**

   true/false: Physically tagged CPU caches have to be invalidated whenever the dispatcher switches to a different virtual address space

   true/false: Physically tagged CPU caches are faster (lower latency), as they operate closer to the hardware, directly using physical addresses without virtual-to-physical translation overhead

   XX is the indexing / tagging combination that is of no practical relevance in real systems

   XX is used as the tag in a physically-tagged CPU cache.

b. Which of the following statements about the Meltdown attack is true (select exactly one – the most valid statement)? **1 T-pt**

true	false	
☐	☐	Meltdown is a type of side-channel attack that targets cryptographic keys, exposing sensitive information stored in secure enclaves (like Intel SGX).
☐	☐	Meltdown relies on injecting malicious code into the victim's system through phishing attacks, compromising the kernel's integrity.
☐	☐	Meltdown leverages out-of-order execution, enabling a user-mode application to infer information about sensitive data in kernel memory.
☐	☐	Meltdown leverages speculative branch prediction on the CPU, enabling a user-mode application to escape out of its sandbox environment.

P-Question 7.1: TLB and Paging Measurements

Download the template **p1** for this assignment from Canvas. You may only modify and upload the file `measure_tlb.c`, together with files `info.txt`, `plot-noskel.png` and `plot-other.png` – see also part (e), and the file `groupinfo.txt`

In this question you will implement code to measure the impact of various mechanisms on memory access performance.

a. **Implement a function to measure the execution time of another function.** **2 P-pt**

For this purpose, the template contains the function `measureFunction`. This function takes a function pointer (`function`) and an argument pointer (`arg`) as parameters.

You should implement code that

- invokes the function referenced by `function`, passing `arg` as parameter;
- before and after the invocation, uses the API function `clock_gettime` (see man page) to obtain a high-precision time stamp; and finally
- if `function` returns an error code (value < 0), returns -1;
- otherwise, returns the elapsed time (difference between the two time stamps) in nanoseconds.

```
int64_t measureFunction(  int(*function)(void *), void *arg );
```

b. **Implement a function that accesses main memory.** **3 P-pt**

This is the function that you want to examine (measure the execution time of it).

The function takes three arguments:

- `memsize`: The size of the memory to use. Allocate (and deallocate when done) this memory on your program's heap.
- `count`: In total the function should access (read) the memory `count` times (read a single `uint64_t` value from memory each time).
- `step`: After each read access, the program advances `step` bytes for the next access, wrapping back to the beginning when it reaches the end of the allocated memory.

Example: With memsize=3000, step=1000, and count=10, your program shall read from these offsets in the allocated memory: 0, 1000, 2000, 0, 1000, 2000, 0, 1000, 2000, 0.

In case of insufficient available memory, your function shall print an error message and return the error code -1. On success, return 0.

```
int accessMemory(uint64_t memsize, uint64_t count, uint64_t step);
```

c. **Implement a wrapper function to measure your function.** **1 P-pt**

Looking at those two functions from the previous parts, there is a small problem: The generic measurement function takes a single `void *` as parameter, but the function you want to measure needs three parameters.

This problem can be solved with a simple wrapper function. The header contains a structure (`MeasurementParameters`) that can hold the three parameters. Implement a wrapper function that retrieves the three parameters from the structure and calls your function. This wrapper function can then be used as parameter for your `measureFunction` implementation.

```
void accessMemoryWrapper(MeasurementParamters *param);
```

d. **Implement a function that executes the meaasurements.** **2 P-pt**

Based on the skeleton in the template, implement a function that measures the execution time for various memory sizes, doing `COUNT` memory accesses in a loop.

- Start with a memory size large enough for two pages, then iteratively increment the size by a factor for two (i.e., you will be using 2, 4, 8, 16, ... pages).
- Terminate when the system is out of memory (on noskel, never use more than 1 GiB of memory).
- For each memory size, run your measurements with step size 64 and with step size 4096 (this is the size of a cache line and the size of a memory page on Intel x86).
- Choose a good value for `COUNT`. Good means: larger values are good for the precision of the measurement, but too large values will result in too large run time (in particular for large memory sizes).
- If you want to enhance your solution, repeat the measurement multiple times and use the minimum (!) value (minimum is good, as the smallest runtime is the run with the least perturbations due to other software running on the same CPU). You could dynamically adjust the number of repetitions (many for small memory (which is fast), less for large memory (which is slow)).
- Do NOT change the output (modify `printf` statements or add additional output)!

e. **Plot the results.** **2 P-pt**

Run the measurement on two different computers (your own and noskel). Plot the results with a tool of your choice (Excel, Python script with pyplot, etc.)

- X-Axis: memory size (log scale, i.e. all the measurement with memory size 2^x shall be evenly distributed on the X axis
- Y-Axis: Measured time
- Two curves (different colors) for step size 64 and 4096.

Include two images `plot-noskel.png` and `plot-other.png` in your solution.

Furthermore, add a simple text file `info.txt` briefly describing the other system ("your own") that you used (OS, CPU info, RAM size).

f. **Bonus: Explain the results** **2 P-pt**

The program code does the same thing for all configurations: In a loop, it accesses the memory `COUNT` times. Why does the run time differ between various memory sizes? You can obtain up to two bonus points for explaining the results. Prerequisite for a bonus is an excellent quality of the solution of the previous parts:

- Your `accessMemory` function indeed measures primarily the memory access, without unnecessary other clutter.
- Your `executeMeasurement` functions implements the enhancement listed in (d)

Which concepts of the lecture provide a good explanation of the increase of the access time with increasing memory size? Can you derive any estimate on information such as TLB size, cache size, main memory size, or CPU clock frequency?

Furthermore, add `memset(memory, 0x55, memsize);` immediately after your malloc call. Explain the effect his has on the runtime!

(Put your answers into `info.txt` – plain utf-8 text or markdown)

Total:
10 T-pt
12 P-pt