Institut für Informatik **Ludwig-Maximilians-Universität München**
Prof. Dr. Francois Bry

Higher level languages: Rust, WS 18/19
Tutorial 6

November 21, 2018

Exercise 6-1 C memory allocator benchmark

Given is the archive file `allocator.tar.gz`. The program consisting of files `clib.c` and `main.rs` provides a Rust interface to the system memory allocator `malloc(3)` which is defined in the standard library `glibc`. The subtasks below aim at getting precise measurements of the number of CPU cycles that are required for allocating memory blocks of different sizes using the system allocator.

a) Compile the program in order to check that all required dependencies are installed and the program works as intended.

   Look at the output of the program and try to retrace program execution. What happens before and after each line of output?

b) Try to understand how building the program works. In order to get a better understanding, rename the custom C library source code `clib.c` into `alloc.c` and the static library built from it from `libclib.a` to `liballoc.a`.

   The following files require changes: `Cargo.toml`, `build.rs`, and `src/main.rs`.

The CPU instruction RDTSC (short for ReaD Time Stamp Counter, available on all recent Intel and AMD CPUs) determines the number of elapsed CPU cycles since the the CPU was brought online. Hence, by the repeated use of RDTSC fine-granular time measurements are possible. We use this CPU instruction to measure the time required to allocate memory blocks.

The number of elapsed cycles can be represented by a 64-bit unsigned integer. However, the CPU instruction RDTSC does not return a full 64-bit value: The high-order 32 bits of this value needs to be read from register EDX, the low-order 32 bits from register EAX. Rust code to determine the complete 64-bit value looks like this:

```
// Rust Nightly required in order to use the asm! macro.
fn rdtsc() -> u64 {
    unsafe {
        let (cycles_lo, cycles_hi) = (0u32, 0u32);
        asm!("RDTSC" : "=a" (cycles_lo), "=d" (cycles_hi));
        (cycles_hi as u64) << 32 | cycles_lo
    }
}
```

Unfortunately, the `asm!` macro is only available on the Rust Nightly platform which we do not use in the tutorial. Consequently, the code above was ported to C. Invoking this C function from Rust also requires the use of the `unsafe` keyword.

c) Write a custom data type `Cycles` and define an associated function `fn start() -> Cycles` and a method `fn stop(self) -> u64` which exposes a safe API for the unsafe `rdtsc` function. Calling the method `stop` should return the number of elapsed cycles since the call to `start`.

d) Use the `Cycles` structure from above to determine how good the system allocator works on smaller and larger memory blocks. To do this, implement the following pseudo-code:

```
let mut blocks: Vec<Mem> = Vec::new();     // store all allocated memory blocks
let mut cycles: Vec<u64> = vec![0; 14];    // store accumulated time
let mut count:  Vec<u64> = vec![0; 14];    // # of blocks of particular size
for _ = 0..100_000                         // repeat for reliable measurement
    let exponent = a random number between 4 and 13 (inclusive)
    let blocksize = 2^exponent             // blocksize between 16 bytes and 8K
    let cycles = Cycles::start();          // start the clock
    let block = Mem::new(blocksize);       // request allocation

    // now, trick the compiler and actually force allocation by
    // accessing the memory block and printing its content
    // however, printing is slow and we actually do not want to do it
    // (but the compiler cannot know: we compare against a random number)
    if exponent == 100 {
       for byte in block.0.iter() {
           print!("{:02x} ", byte);
       }
    }
    count [exponent] += 1;                 // record count for block-size
    cycles[exponent] += cycles.stop();     // record run-time
    blocks.push(block);                    // store block
println!("{}", cycles/count);              // pseudo-code, understood point-wise
```

e) Finally, benchmark the standard memory allocator. To do this, comment-out the `printf`-statements in the C library; output to the console would invalidate the measurements. Then, compile your finished program in release mode (`cargo build --release`) and run it.

Running the program is a bit tricky because we are utilizing the RDTSC processor instruction to determine the number of passed cycles. However, this value is not global but specific to each CPU in a machine. On the other hand, modern operating systems reserve the right to move programs from one CPU to another if deemed necessary. In order to get reliable measurements, one has to limit the execution of the binary to one processor. This can be done with the `taskset` command on Linux (available from package `util-linux`, installed by default on Debian, Ubuntu, ArchLinux), the thread affinity API or by other means on OS X, and the task manager in Windows.

In order to limit the program to the first processor, the `taskset` tool is called with these two command line arguments: `taskset 1 target/release/allocators`.