#uni/semester3/Betriebssysteme/chapter19/code-fragen

1. For timing, you'll need to use a timer (e.g., gettimeofday()). How precise is such a timer? How long does an operation have to take in order for you to time it precisely? (this will help determine how many times, in a loop, you'll have to repeat a page access in order to time it successfully)

➜ The function **clock_getres**() finds the resolution (precision) of the specified clock clockid. (CLOCK_REALTIME, CLOCK_MONTONIC_RAW)

➜ Was macht clock_getres() ?

➜ Gibt die Zeitauflösung einer Funktion an. Zeitauflösung (resolution, precision) ist die kleinste Zeiteinheit, um die gesteigert werden kann. Die Funktion setzt dann tv_sec auf 0, und die resolution in tv_nsec. How do Computer Clocks work?

➜ For instance, if the clock hardware for CLOCK_REALTIME uses a quartz crystal that oscillates at 32.768 kHz (32,768kHZ), then its resolution would be 30.518 microseconds, and 'clock_getres (CLOCK_REALTIME, &r)' would set r.tv_sec to 0 and r.tv_nsec to 30518. Getting the Time (The GNU C Library)

$$\frac{1}{32768} s \cdot 10\hat{}9 = 30518ns$$

➜ How precise is such a timer?

```
vl161bra@ct-bsys-ws20-12:~/htwg/S3/BS/BSCode/c19_tlb$ ./get_precision
Resolution of CLOCK_REALTIME is 1 nano seconds.
Resolution of CLOCK_MONOTONIC is 1 nano seconds.
Resolution CLOCK_MONOTONIC_RAW is 1 nano seconds.
Resolution CLOCK_PROCESS_CPUTIME_ID is 1 nano seconds.
```

➜ How long does an operation have to take in order for you to time it precisely?

➜ An operation has to to take at least 1 nano second.

2. Write the program, called tlb.c, that can roughly measure the cost of accessing each page. Inputs to the program should be: the number of pages to touch and the number of trials.

➜ use getpagesize(2) - Linux manual page to get the page size

➜ gdb tlb ➜ run 10 1000

➜ valgrind --leak-check=yes ./tlb 10 1000

➜ gcc -o tlb tlb.c -Wall -Wextra -Wpedantic

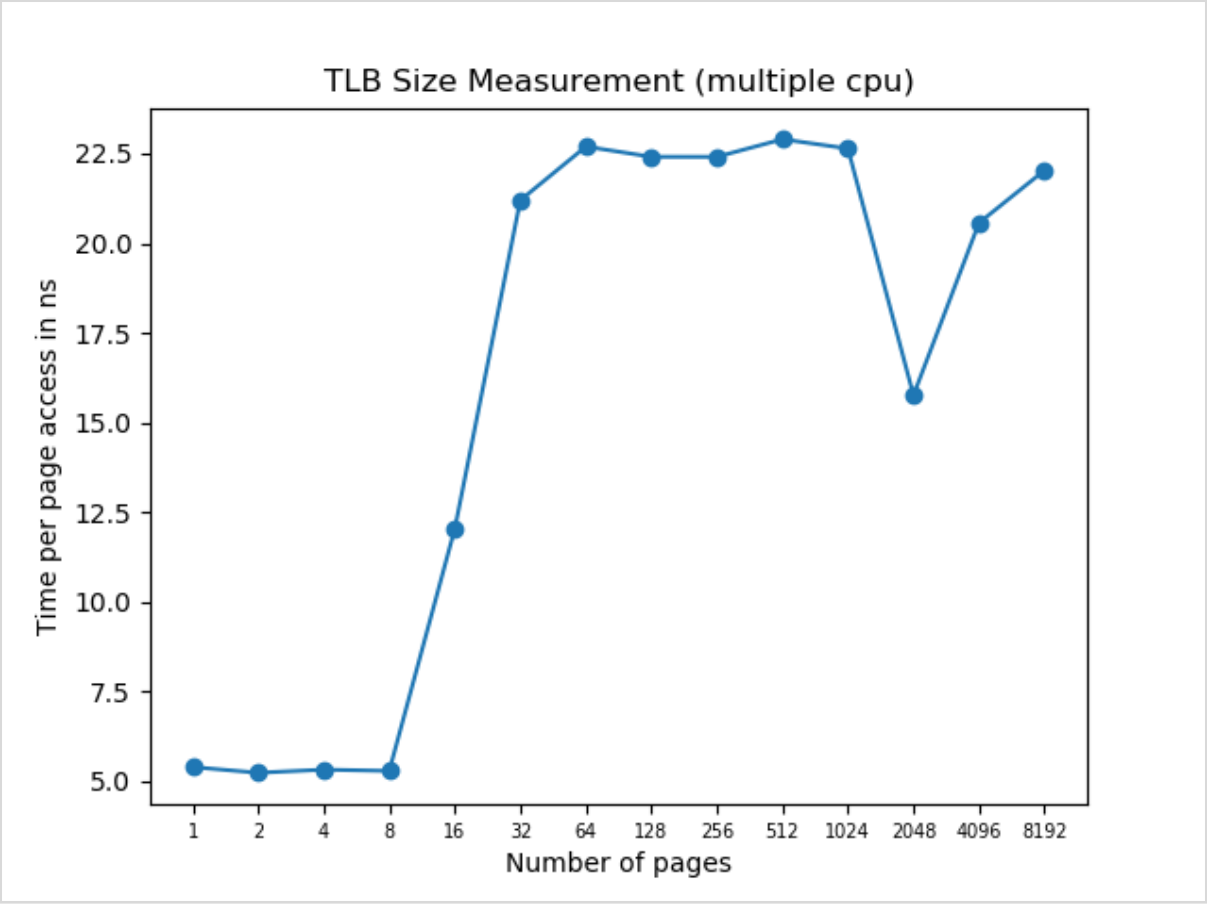
➜ ./plot_access.py --trials 3000

3. Now write a script in your favorite scripting language (bash?) to run this program, while varying the number of pages accessed from 1 up to a few thousand, perhaps incrementing by a factor of two per iteration. Run the script on different machines and gather some data. How many trials are needed to get reliable measurements?

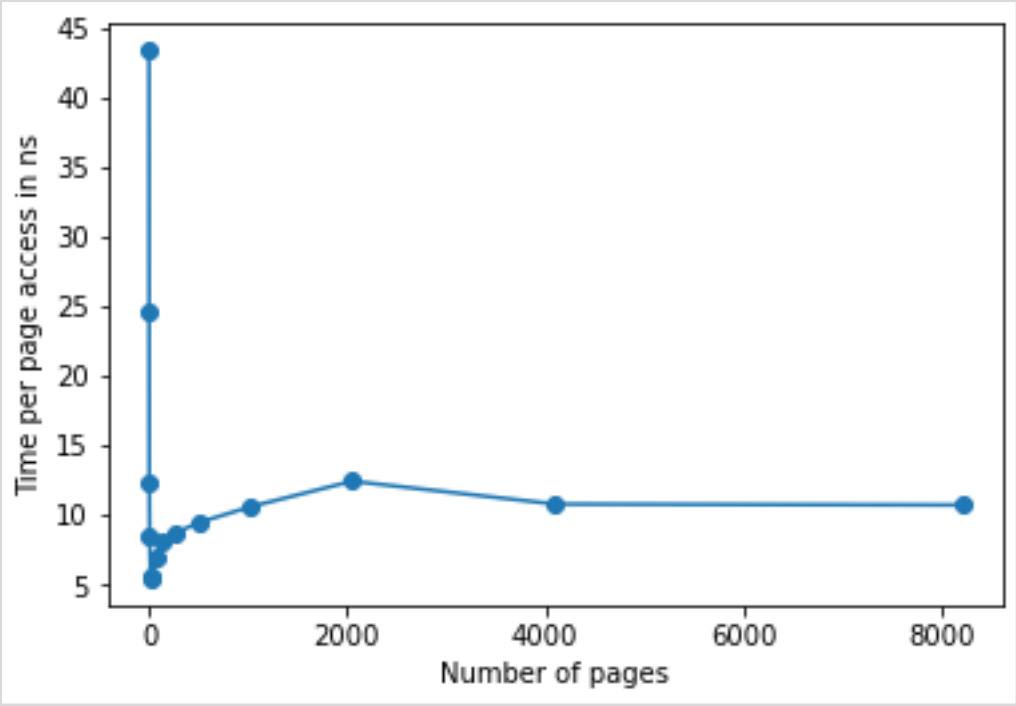
   ➜ Für trials zwischen 1000 und 10000 auf eigenem laptop und container, hat es für uns am besten gepasst.

```
vl161bra@ct-bsys-ws20-12:~/z-drive/S2/S3/BS/BSCode/c19_tlb$ ./plot_access.py --trial
s 10000
Unable to init server: Could not connect: Connection refused
Unable to init server: Could not connect: Connection refused

(plot_access.py:29434): Gdk-CRITICAL **: 16:28:39.757: gdk_cursor_new_for_display: a
ssertion 'GDK_IS_DISPLAY (display)' failed

(plot_access.py:29434): Gdk-CRITICAL **: 16:28:39.758: gdk_cursor_new_for_display: a
ssertion 'GDK_IS_DISPLAY (display)' failed
1 : b'5.409700'
2 : b'5.187300'
4 : b'5.322625'
8 : b'5.289125'
16 : b'11.963256'
32 : b'20.181944'
64 : b'21.731262'
128 : b'22.301032'
256 : b'21.950524'
512 : b'21.973813'
1024 : b'22.574884'
2048 : b'15.558018'
4096 : b'19.830065'
8192 : b'21.827494'
```

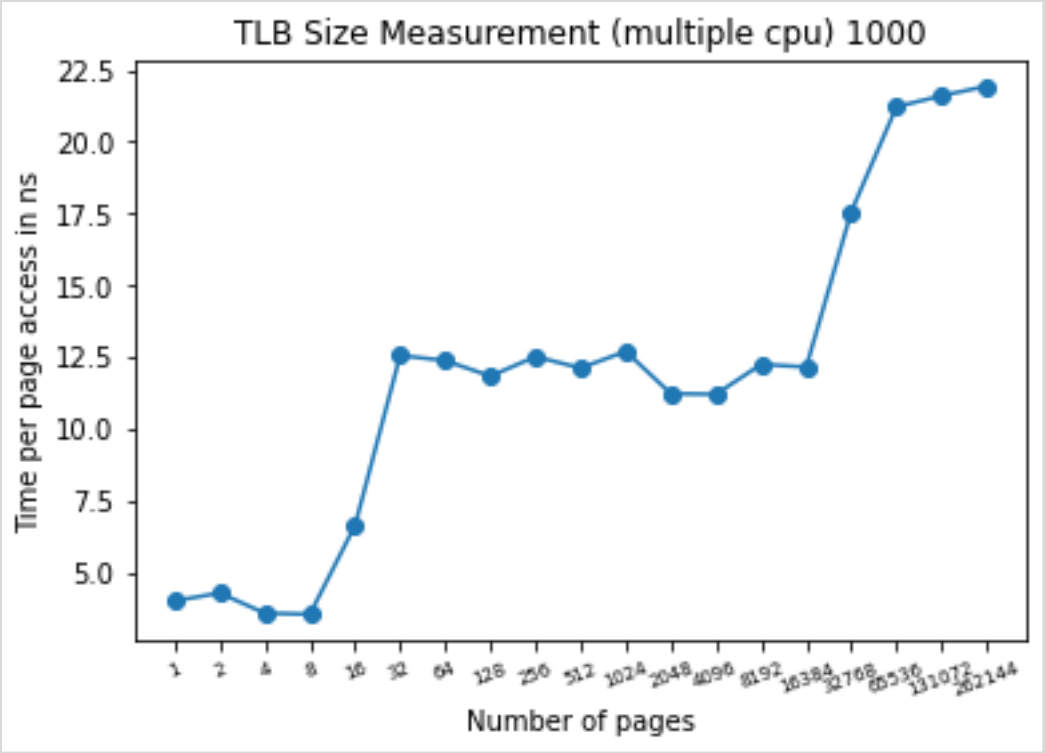
   ➜ Beobachtung: Bei Pageanzahl 2048 ist die Zeit immer kleiner.



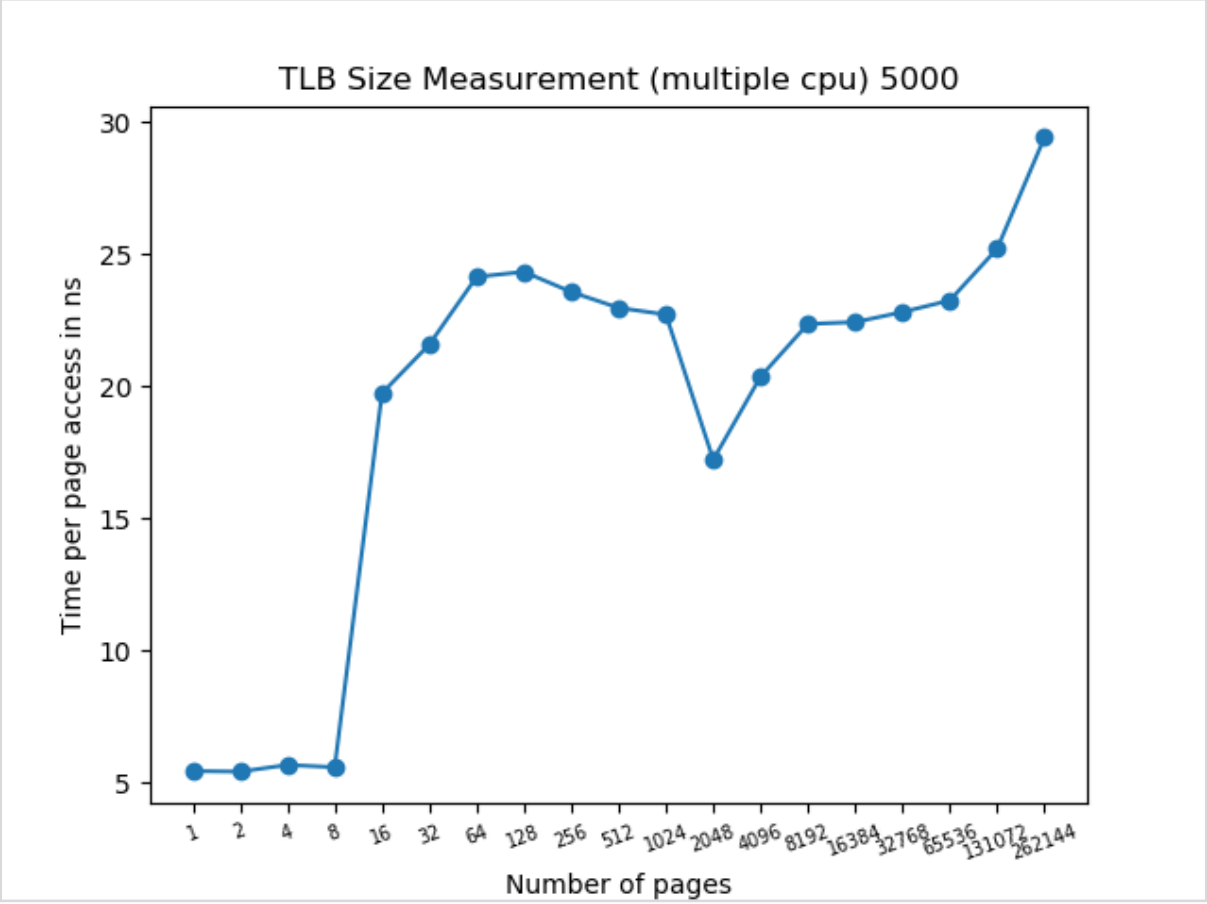
➜ Aufruf auf container mit Trials = 10000

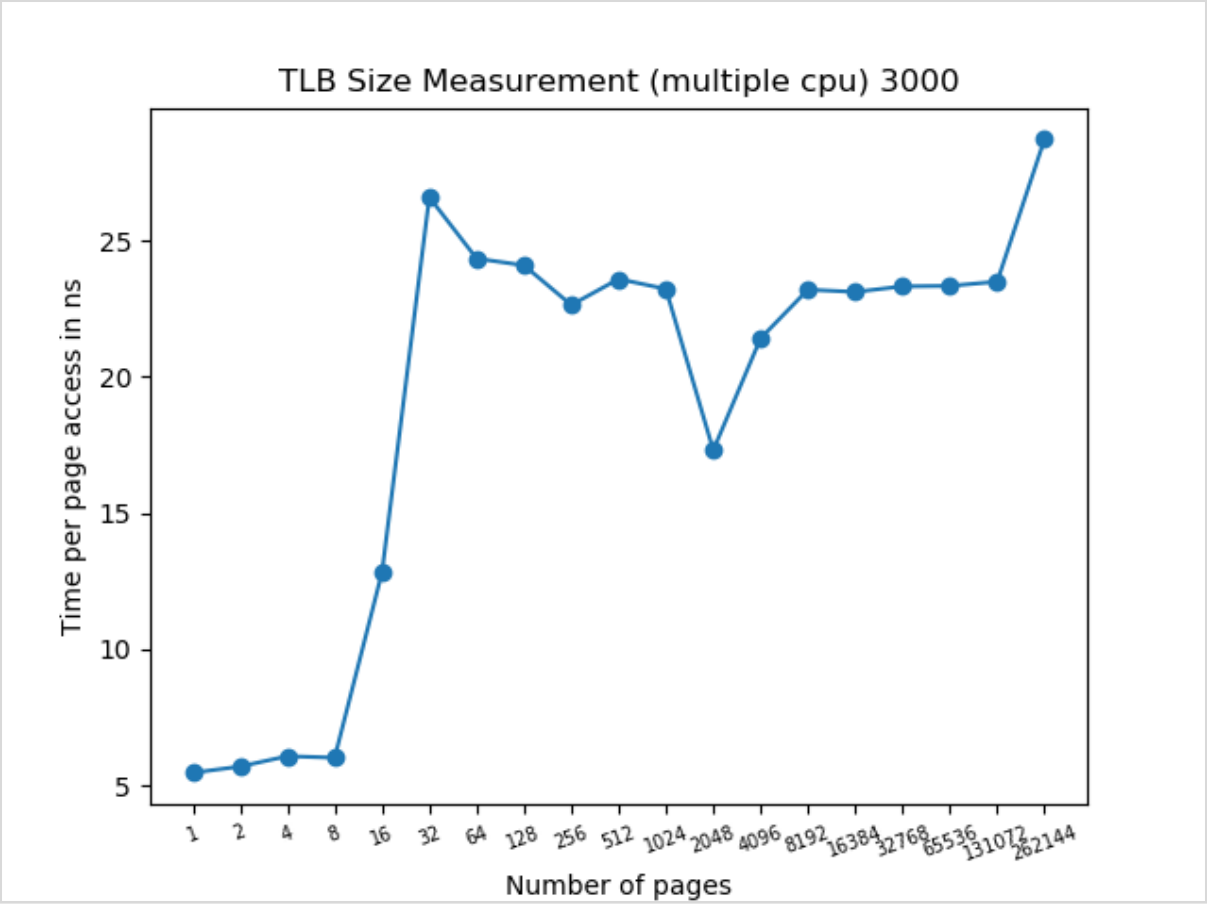


➜ Aufruf auf container, wenn man die innere for schleife misst, und nicht die äußere



➜ Auf meinem PC mit 'run plot_access.py --trials 1000'





5. One thing to watch out for is compiler optimization. Compilers do all sorts of clever things, including removing loops which increment values that no other part of the program subsequently uses. How can you ensure the compiler does not remove the main loop above from your TLB size estimator?

➜ gcc -O0 -o tlb tlb.c (-O0 ist default) und deaktiviert Optimierung
➜ Mit -O0, -O1, -O2 kann Optimierung aktiviert werden. Umso höher die Zahl desto mehr wird optimiert.

6. Another thing to watch out for is the fact that most systems today ship with multiple CPUs, and each CPU, of course, has its own TLB hierarchy. To really get good measurements, you have to run your code on just one CPU, instead of letting the scheduler bounce it from one CPU to the next. How can you do that? (hint: look up "pinning a thread" on Google for some clues) What will happen if you don't do this, and the code moves from one CPU to the other?

➜ sched_setaffinity(0, sizeof(mask), &mask);
➜ Die Null heißt der aufrufende Prozess. Kinder die mir fork() erstellt werden, vererben die mask der Eltern.
➜ Affinität, also das Binden eines Prozesses zu einem Prozessor kann zu Ersteigerung

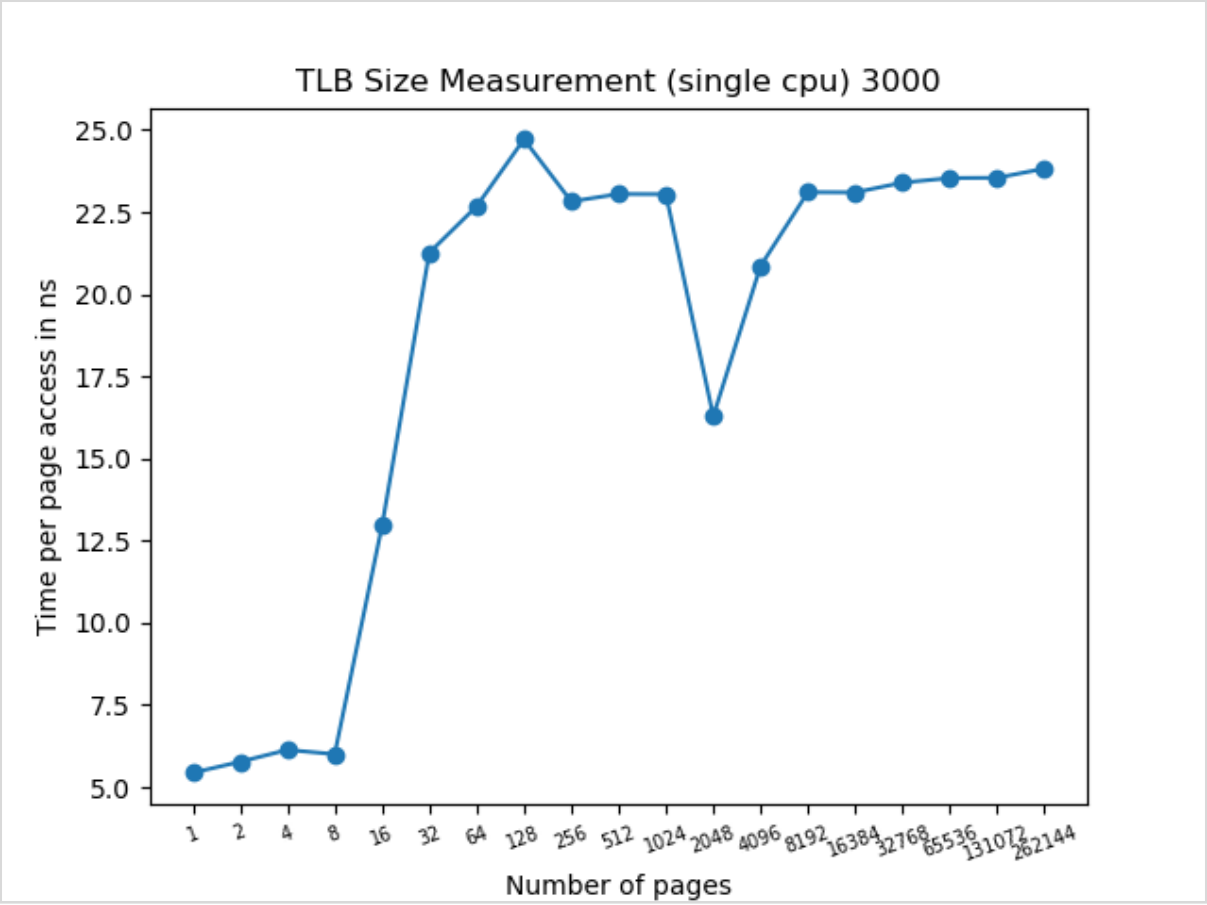
der Performance führen, weil es in manchen Fällen cache invalidation und trashing reduziert

➜ For threads on macOS Binding Threads to Cores on OSX

➜ (If you are using the POSIX threads API, then use pthread_setaffinity_np(3) instead of **sched_setaffinity**().)

OVERHEAD

```
    return 0;
}



double overhead_timer(void)
{
    struct timespec start, end, result;
    long loops = 10000;
    double res;

    for (int i = 0; i < loops; i++)
    {
        clock_gettime(CLOCK_MONOTONIC_RAW, &start);
        clock_gettime(CLOCK_MONOTONIC_RAW, &end);
        result = eval(start, end);
        res += ((result.tv_sec * BILLION) + result.tv_nsec);
    }
    res = res / loops;
    //fo = for_overhead();
    //res = res - fo;
    return res;
}
```



➜ Das bekommen wir, wenn wir Prozess nur an einem Prozessor binden. Unterschied ist dass für 262144 jetzt nicht mehr nach oben geht.

➜ Setting the mask on a multiprocessor system can improve performance. For example, setting the mask for one process to specify a particular CPU, and then setting the mask of all other processes to exclude the CPU, dedicates the CPU to the process so that the process runs as fast as possible. This technique also prevents loss of performance in case the process terminates on one CPU and starts again on another, invalidating cache. sched_setaffinity

➜ A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run.

On a multiprocessor system, setting the CPU affinity mask can be used to obtain performance benefits. For example, by dedicating one CPU to a particular thread (i.e.,

setting the affinity mask of that thread to specify a single CPU, and setting the affinity mask of all other threads to exclude that CPU), it is possible to ensure maximum execution

speed for that thread. Restricting a thread to run on a single CPU also avoids the performance cost caused by the cache invalidation that occurs when a thread ceases to

execute on one CPU and then recommences execution on a different CPU. Ubuntu Manpage: sched_setaffinity, sched_getaffinity - set and get a thread's CPU affinity mask

7. Another issue that might arise relates to initialization. If you don't initialize the array a above before accessing it, the first time you access it will be very expensive, due to initial access costs such as demand zeroing. Will this affect your code and its timing? What can you do to counterbalance these potential costs?

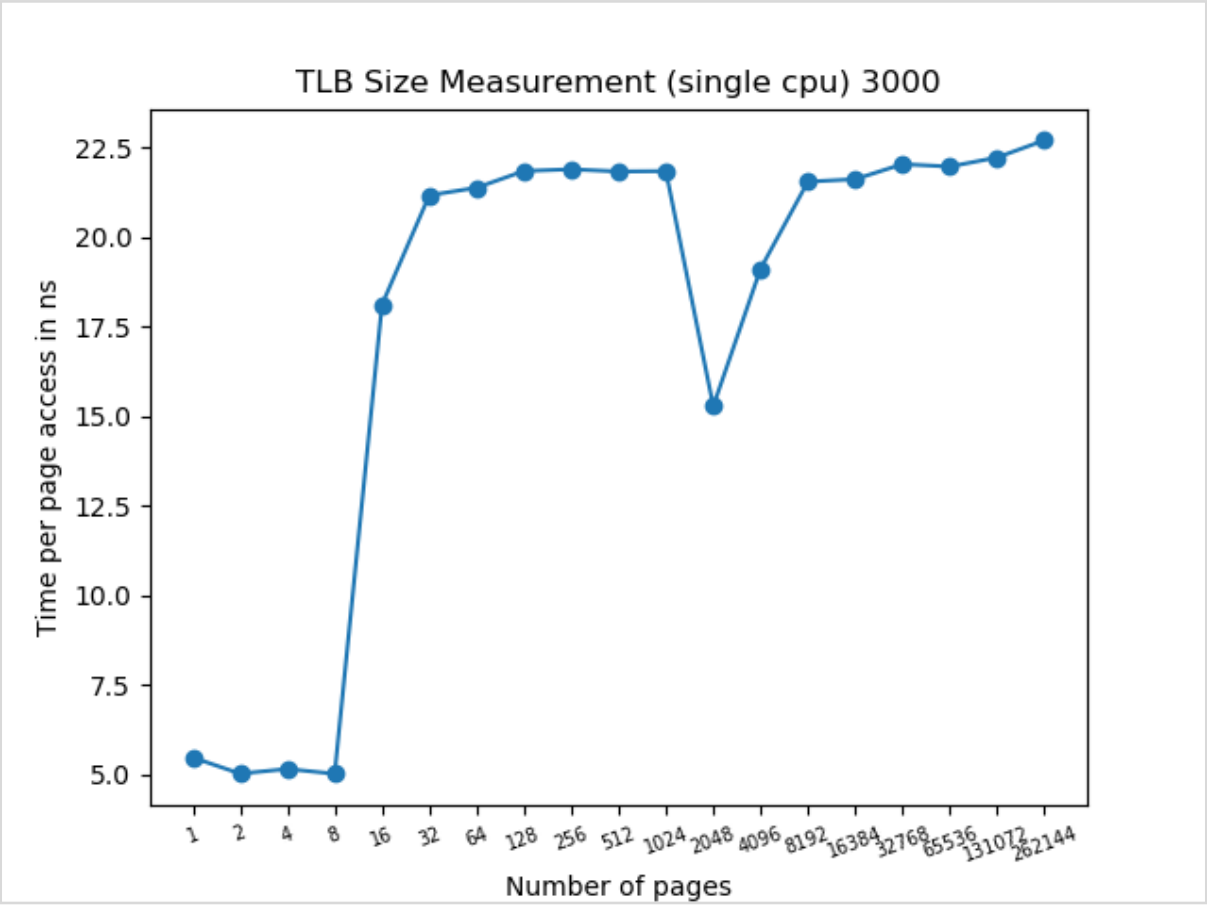
➜ A dynamic memory allocator maintains an area of a process's virtual memory known as the heap. Details vary from system to system, but without loss of generality, we will assume that the **heap is an area of demand-zero memory** that begins immediately after the uninitialized bis area and grows upward (toward higher addresses).

➜ http://pages.cs.wisc.edu/~remzi/OSTEP/vm-vax.pdf Seite 6

➜ lies es nochmal durch mit demand zeroing

> VMS had two other now-standard tricks: demand zeroing and copy-on-write. We now describe these **lazy** optimizations.
>
> One form of laziness in VMS (and most modern systems) is **demand zeroing** of pages. To understand this better, let's consider the example of adding a page to your address space, say in your heap. In a naive implementation, the OS responds to a request to add a page to your heap by finding a page in physical memory, zeroing it (required for security; otherwise you'd be able to see what was on the page from when some other process used it!), and then mapping it into your address space (i.e., setting up the page table to refer to that physical page as desired). But the naive implementation can be costly, particularly if the page does not get used by the process.
>
> With demand zeroing, the OS instead does very little work when the page is added to your address space; it puts an entry in the page table that marks the page inaccessible. If the process then reads or writes the page, a trap into the OS takes place. When handling the trap, the OS notices (usually through some bits marked in the "reserved for OS" portion of the page table entry) that this is actually a demand-zero page; at this point, the OS then does the needed work of finding a physical page, zeroing it, and mapping it into the process's address space. If the process never accesses the page, all of this work is avoided, and thus the virtue of demand zeroing.

➜ General approach is to zero a page before assigning it to an address space, which is slow, and not beneficial when the page won't be used. The zeroing is done by the os. With demand zeroing, the OS does less work. It puts the translation in the page table a new entry with this page, and marks it as inaccessible. When process tries to use the page, it traps to the OS, the OS figures out it is a demand-zero page, and then zeroes it and maps it to the process address space. If the process doesn't use this page, then all the work will be avoided. Zeroing is still slow, so by initialising it with calloc beforehand, improves the performance later.



➜ Wir haben mit calloc davor den Speicher initialisiert. Alles im allgemein sieht ein bisschen schneller aus, aber der Unterschied ist nicht groß.