Gruppe 15 | Tobias Schoch, Luis Nothvogel

Aufgaben wurden im HTWG Container & Linux VM erledigt (details unten)

19.1

Zitat aus der Man Page von gettimeofday(): „Gives the number of seconds and microseconds since the Epoch. “ Dabei kann man mit tv_sec oder tv_usec arbeiten. Tv_sec misst die Zeit in Sekunden und tv_usec in Millisekunden.
Daher ist die Präzision des Timers auf Millisekunden genau.

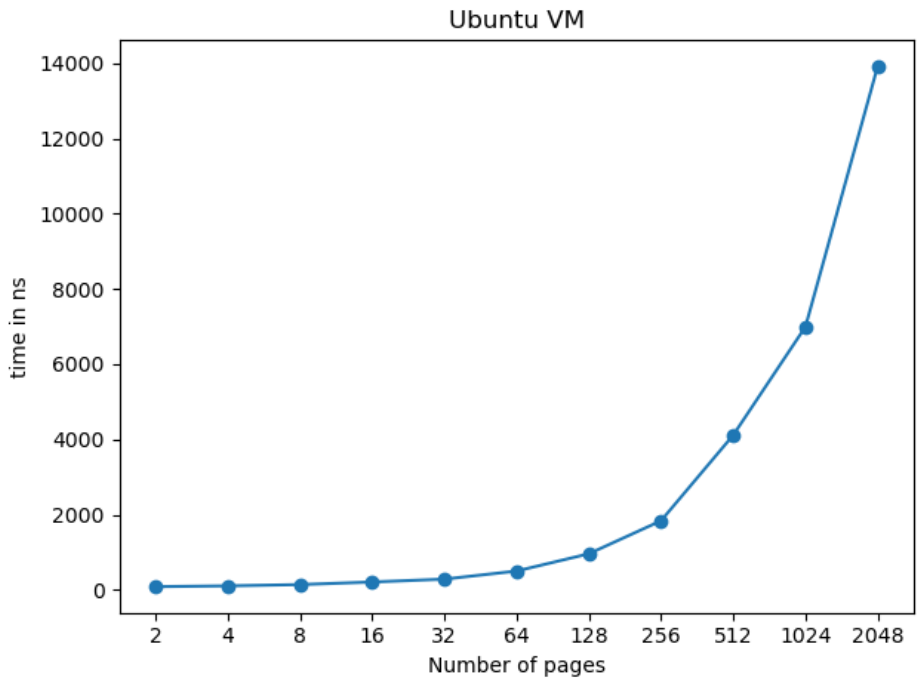
Um eine Zeit zu erfahren wie lange eine Operation braucht, sind viele Faktoren im Spiel. Selbst wenn man das Programm auf demselben System am nachfolgenden Tag ausführt, erhält man ein anderes Ergebnis. Zudem spielen noch Faktoren wie das Betriebssystem, die Hardware oder die Auslastung des Computers eine Rolle.
Um eine annähernde Zeit zu erfahren, könnte man auf demselben System das Programm sehr oft ausführen um eine Durchschnittszeit zu erhalten. Zudem könnte man statt gettimeofday() andere Funktionen benutzen, wie zum Beispiel CLOCK_MONOTONIC(), die die absolute Zeit seit einem starting point misst und damit einer der C Funktionen ist, die am genausten misst.

19.3

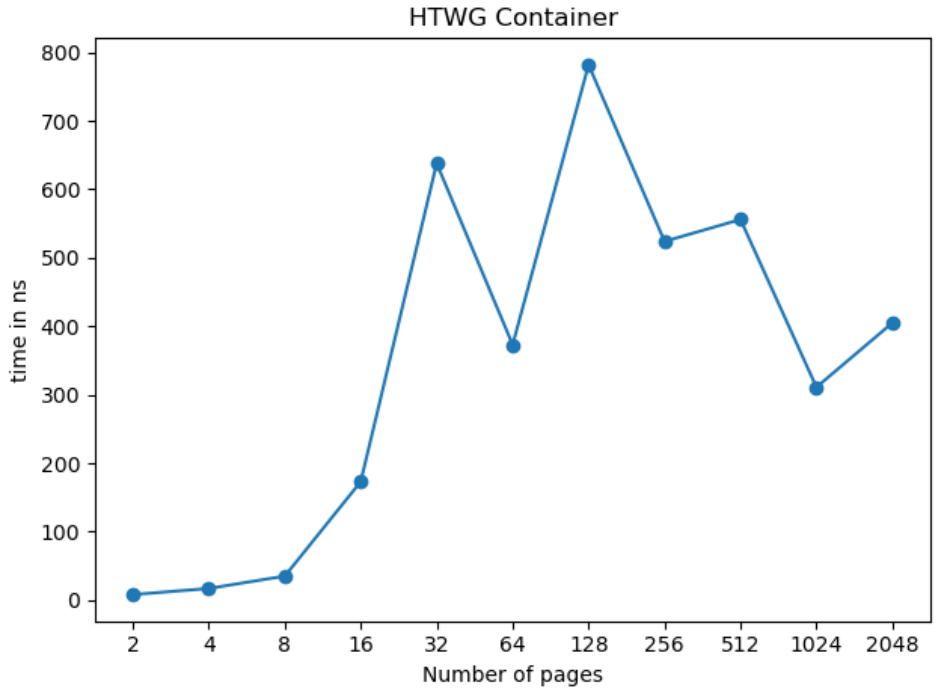
Bei einer frischen Ubuntu VM mit 8GB Arbeitsspeicher, 2 x 2 CPU-Kernen I7 4790K und einer 980TI hat jeder Versuch geklappt, so dass wir sofort zuverlässige Werte bei diesem System erhalten haben ohne einen Aufruf nochmal zu starten. Beim HTWG Container hat dieser Versuch auch geklappt, allerdings mussten wir mehrmals die Versuche wiederholen, um zuverlässige Werte zu erhalten. Dabei wurde das Skript 6-mal ausgeführt, da unser Programm oft negative Werte ausgegeben hat.

19.4

Bei einer frischen Ubuntu VM mit 8GB Arbeitsspeicher, 2 x 2 CPU-Kernen I7 4790K und einer 980TI sah der Versuch folgendermaßen aus.



Beim HTWG Container sah der Versuch so aus:



19.5

gcc -O3 ist die höchste Form der Optimierung, während gcc -O0 jegliche Optimierung deaktiviert. Daher würde die Flag -O0 das Programm nicht verändern und es somit zulassen, dass die main loop nicht gelöscht wird. Dadurch, dass nichts optimiert wird, verbessert sich auch die Compilation Zeit.

19.6

Nahezu diesen Code haben wir in Aufgabe 6.2 bereits verwendet, wo wir einen Context Switch nur auf einem Prozessor ausführen lassen durften.

cpu_set_t cpu_set;
CPU_ZERO(&cpu_set);
CPU_SET(0, & cpu_set);
sched_affinity(0, sizeof(cpu_set_t), & cpu_set);

Dabei wird bei cpu_set_t erst einmal definiert. Im Anschluss mit CPU_ZERO mit 0 initialisiert. Danach nimmt man eine Zahl die einem Prozessorthread zugewiesen ist, bei uns nehmen wir den Prozessorthread 0.
Danach wird mit sched_affinity der Wert übergeben, welcher Prozessorthread für das Programm verwendet werden soll.

Wenn eine CPU ein TLB Miss hat und im Anschluss zum nächsten CPU springt, wird sie dort erneut wahrscheinlich auch einen TLB Miss haben, sofern in der TLB der nächsten CPU noch kein Eintrag für die Page vorhanden ist. Deshalb ist es von Vorteil für den Versuch nur eine CPU zu verwenden, dass wir so einen unverfälschten Eindruck erhalten. Denn beim Benutzen von nur einer CPU, kann es auch nur einen TLB Miss pro Page geben.

Links die bei unserer Recherche verwendet wurden:

- https://stackoverflow.com/questions/10490756/how-to-use-sched-getaffinity-and-sched-setaffinity-in-linux-from-c
- https://linux.die.net/man/2/sched_setaffinity

19.7

Ja, es beeinflusst code und timing. Denn wie schon in der Aufgabenstellung beschrieben, muss der Array initialisiert werden (z.B. mit 0) und diese Initialisierung kostet Zeit.

Um diesem Problem entgegenzuwirken, könnte man direkt statt den Befehl malloc() aufzurufen, den Befehl calloc() aufrufen, der direkt bei der Speicherreservierung die Speicherstellen mit 0 initialisiert.
Dieser Prozess wäre allerdings beim Aufrufen „teuer“, da im Gegensatz zu malloc() bei calloc() die Memory beschrieben wird und evtl. noch Platz frei gemacht werden muss.