Betriebssysteme
Übungsblatt 1

Kilian Bartz (Mknr.: 1538561)

Hinweis

Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/bs_ueb01`.

Aufgabe 1: System Call

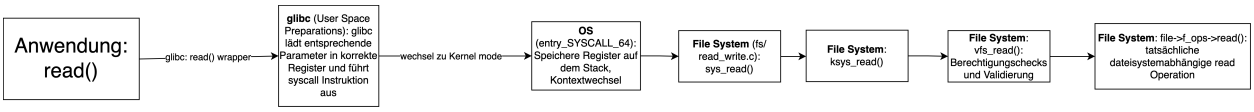

Abbildung 1: Ablauf eines Systemcalls am Beispiel read()

Zur Veranschaulichung eines Systemcalls habe ich mir angeschaut, wie ein *read*-Befehl in einem C Programm, ausgeführt auf einem Linux System, umgesetzt wird. Eine Visualisierung ist in Abbildung 1 zu sehen. Die Anwendung greift dabei in erster Instanz auf den *read*-Wrapper der glibc zu. Dieser initialisiert den **Systemcall**, indem die Systemcall Parameter (Syscall Nummer, Argumente des *read*-Aufrufs) in entsprechende Register geladen werden. Anschließend wird mittels der *syscall* Instruktion ein **Software-Interrupt** ausgeführt, das System wechselt in den Kernel-Modus und springt in die "entry_SYSCALL_64" Routine. Hier wird ein **Kontextwechsel** initiiert, nachdem die aktuellen **Registerinhalte** auf dem Stack **gespeichert** wurden. Nach dem Kontextwechsel wird die *sys_read()*-Routine in fs/read_write.c ausgeführt, welche dann durch eine Schicht von **Abstraktionen** (ksys_read(), vfs_read(), file→fops→read()), in denen jeweils **Validierungen und Errorhandling** stattfinden, den tatsächlichen, dateisystemspezifischen *read*-Code ausführt und einen Filehandle anlegt. Dieser wandert dann wieder in der Abstraktionshierarchie hinauf, bis der vergebene Filehandle schließlich in ein entsprechendes Register gelegt werden kann. Der vorherigen Kontext (Registereinträge, Stackposition etc.) wird wiederhergestellt und das Betriebssystem gibt die Kontrolle zurück an das aufrufende Programm.

Aufgabe 2: Messen der Systemcall Latenz

Zur Messung der Systemcall Latenz wird in den meisten Benchmarks, die ich finden konnte, z. B. OSJitter[1] oder syscall-benchmark[2] initial eine Startzeit gemessen, eine ein billiger Systemcall in einer for-Schleife wiederholt (z. B. 10M mal) aufgerufen und anschließend die durchschnittliche Zeit eines Systemcalls als arithmetisches Mittel berechnet. Für diesen Ansatz spricht, dass relativ grobe Zeitmessungen verwendet werden können, da die Anzahl der Systemcalls so gewählt werden kann, dass die Auflösung ausreicht (dennoch muss sich die gemessene Zeit monoton erhöhen; Methoden wie `gettimeofday()`, welche auf eine Clock zugreifen, die von anderen Services wie *ntpd* zurückgesetzt werden können, sind nicht geeignet). Außerdem ist dieser Code sehr portabel, da keine architekturspezifischen Register / Instruktionen verwendet werden. Die Messung wird zwar dadurch etwas verfälscht, dass pro Iteration ein Overhead von 2 Machineninstruktionen anfällt (`dec`, `jnz`)[3], dies fällt jedoch so wenig ins Gewicht, dass die Vorteile von `gcc -O3` (aktiviert *loop unrolling*) im Bereich der Messtoleranz zu `gcc -O2` liegt.

[1] `https://github.com/gsauthof/osjitter/blob/master/bench_syscalls.cc`
[2] `https://github.com/arkanis/syscall-benchmark`
[3] `https://github.com/arkanis/syscall-benchmark/blob/master/02_getpid_syscall.asm`

Ein zweiter Ansatz zur Messung wäre es stattdessen, jeden Systemcall einzeln zu messen, dazu ist jedoch ein Zurückgreifen auf architekturspezifische High-Performance-Counter oder nötig, da nur diese die nötige Auflösung zur Messung von Nanosekunden-Intervallen mitbringen. Unter macOS bietet meinen Recherchen zufolge `mach_absolute_time` die akkurateste Zeitmessung ohne Möglichkeit, direkt auf entsprechende Register zuzugreifen; hier ist die Auflösung jedoch nur 41.67ns, was die Messung deutlich verfälschen könnte. Trotzdem könnte für diese Messmethode sprechen, dass es statistisch sehr unwahrscheinlich ist, dass das Zeitquantum während eines einzelnen Systemcalls ausläuft. Somit sollte dieser Ansatz die Interferenz durch andere Prozesse, die nach Ablaufen des Zeitquantums gescheduled und somit mitgemessen werden würden, minimieren.
Schließlich spielt auch die Wahl des konkreten Systemcalls eine wichtige Rolle. Intuitiv ist klar, dass ein Systemcall wie `getpid()`, welcher lediglich einen einzigen Wert nachschlagen muss, deutlich billiger ist als ein Systemcall, der an das Dateisystem gerichtet ist (etwa `read`, `write`). Somit ist dieser in der Theorie deutlich besser dazu geeignet, die Latenz, die durch das Systemcall Prinzip selbst verursacht wird (Wechsel in den Kernelmodus, Kontextwechsel) zu messen. Allerdings wird es hier schnell komplex, da man differenzieren muss zwischen normalen Systemcalls und *vDSO* Systemcalls, welche mittels einer *shared library* den Moduswechsel bei speziellen Systemcalls (`gettimeofday`, `clock_gettime`, `clock_getres`, `getrandom`) überflüssig machen und somit kaum fair zu vergleichen sind. Auf der anderen Seite ist es wahrscheinlich, dass sehr sicherheitskritische Systemcalls durch zusätzliche Mitigationen gegen CPU Bugs wie Meltdown und Spectre durch zusätzlichen (nicht einsehbaren) Mikrocode der seitens der CPU Hersteller übermäßig teuer sind und so selbst innerhalb derselben Prozessorarchitektur schwer vergleichbar sind [4].

Ergebnisse

Initial entschied ich mich mittels Ansatz 1 die Zeit für 10M Iterationen des `getpid` Systemcalls zu messen. Diese Messungen wurden 10x wiederholt und gemittelt. Für die Zeitmessungen zuständig ist `clock_gettime` im Modus `CLOCK_MONOTONIC_RAW`, da dies der einzige Modus ist, der nicht nur eine monotone Zeit, sondern auch Freiheit von NTP und *adjtime*-Anpassungen garantiert.
Als Testsysteme kamen zum Einsatz:

1. MacBook Air 2020 (M1) mit macOS 15.0.1 im Netzbetrieb; Apple clang Compiler
2. PC mit AMD Ryzen 7 7700X (Ubuntu 24.04.1); gcc
3. Raspberry Pi 5B (Raspberry Pi OS 12, Kernel 6.6.47); gcc

Der Code findet sich in `main.c` und wurde auf jedem System mittels `gcc -O2` kompiliert.
Die Ergebnisse für die `getpid`-Messungen finden sich in Tabelle 1. Da die Latenz unter macOS deutlich zu

	System 1	**System 2**	**System 3**
Latenz	1.6	99.5	174.3

Tabelle 1: Durchschnittliche Latenz einer `getpid`-Messung in ns

klein erscheint, habe ich des Weiteren die Systemcalls `getuid`, `gettimeofday` und `open` gemessen (Tabelle 2). Wenngleich sich die Latenzen für diese drei Systemcalls je nach System deutlich unterscheiden (`getuid` auf System 2 am schnellsten, System 1 am schnellsten für die restlichen Systemcalls), liegen sie doch in der gleichen Größenordnung. Die Unterschiede rühren dabei vermutlich nicht nur von den verbauten Prozessoren, sondern auch von den Implementierungen der Systemcalls (macOS vs. Linux).

Wie erklärt sich die große Diskrepanz bei getpid?

Ein Vergleich mit Online-Ressourcen zu diesem Thema [4, 5] zeigt, dass die Latenzen, die ich gemessen habe, in einem plausiblen Bereich liegen (zwischen 80ns-400ns für echte Systemcalls, ¡10ns für *vDSO*-Calls). Einzig `getpid` ist auf dem macOS-System unverhältnismäßig schnell. Um herauszufinden, worin dies begründet

Systemcall	System 1	System 2	System 3
`getuid`	146.1	96.1	167.7
`gettimeofday`	1.9	2.8	4.2
`open`	211.8	255.1	248.9

Tabelle 2: Durchschnittliche Latenzen der Systemcalls in ns

liegt, habe ich den Apple Instruments[4] Profiler verwendet, um die CPU-Zyklen und Systemcalls, welche von einer vereinfachten Version des Tests (`simple_getpid.c`) verursacht wurden, zu inspizieren. Ein Vergleich eines Profilings von `simple_getpid.c` und `simple_getuid.c` findet sich in Abbildung 2. Augenscheinlich

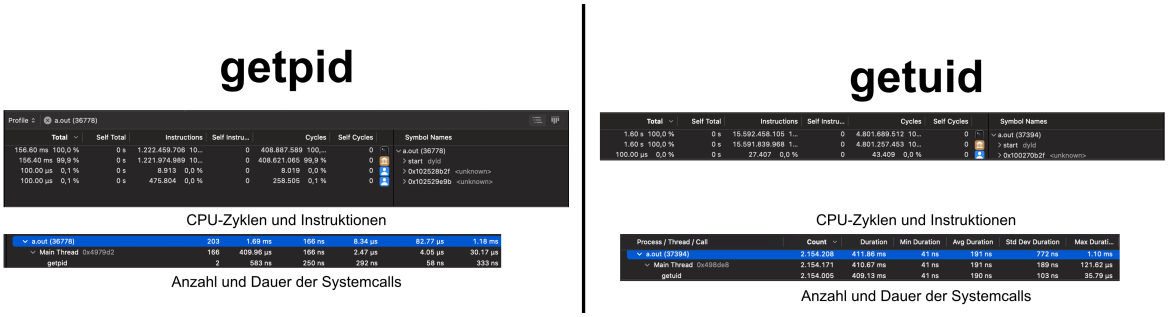

Abbildung 2: Profiling (CPU-Zyklen, Instruktion, Systemcalls) von `getpid` und `getuid`

verursacht `simple_getpid.c` nur etwa 1/13 der Instruktionen von `simple_getuid.c` (bei ansonsten gleichem Code). Obgleich auch bei `getuid` nur 2.1M von 10M Systemcalls aufgezeichnet wurden (dies könnte durch Batching oder die Ungenauigkeit des Sampling Profilers begründet sein), ist doch ein großer Unterschied zu den 2(!) `getpid` Systemcalls erkennbar, die das erste Programm verursachte, was den Verdacht nahelegt, dass die PID hier gecached wurde. Ein Vergleich mit den gemessenen Plattformen durch [4] zeigt, dass diese Optimierung auch unter (älteren) Linux Versionen[5] eingesetzt wurde.

Profiler: Unterschiede in der Implementierung

Nun, da ich den Profiler bereits installiert hatte, habe ich auch in den anderen Systemcalls, die ich gemessen habe, untersucht, ob sich die zeitlichen Diskrepanzen eventuell durch die Komplexität der Implementierungen erklären lassen. In Tabelle 3 habe ich dazu die Instruktionen der Systemcalls, gemessen mit den Profilern Apple Instruments bzw. `perf`, aufgelistet. Die Werte sind nicht zwingend direkt vergleichbar, sollten jedoch ausreichen, um die Größenordnungen einzuschätzen.
Man sieht sehr schön, dass *vDSO* durch das Einsparen des Mode-Switches auch etwa 90% der Komplexität eines normalen Systemcalls einspart, was sich überproportional in der Latenz niederschlägt. Noch mehr Komplexität spart macOS durch das Caching nur in `getpid` ein, was auch die noch schnellere Latenz erklärt.

Systemcall	System 1	System 2
`getpid`	1.222.459.706	37.982.751.636
`gettimeofday`	2.522.162.155	3.102.971.788
`getuid`	15.592.458.105	37.680.717.552

Tabelle 3: Anzahl der Instruktionen, verursacht durch 100M `getpid`, `gettimeofday` und `getuid` Systemcalls.

[4] `https://developer.apple.com/tutorials/instruments`
[5] `https://manpath.be/f34/2/getpid`

Vergleich: Ansatz 1 vs. Ansatz 2

Um festzustellen, wie akkurat die Werte des ersten Ansatzes (Messungen mittels`clock_gettime` + arithmetisches Mittel) ist, habe ich auf System 2 100M `getpid` Systemcalls mittels dem High-Performance-Counter `rtdsc` (Ansatz 2; Datei: `rtdsc.c`) gemessen und die Latenzen mit `clock_gettime`-Messungen verglichen (`rtdsc_comp.c`).
Mithilfe von `rtdsc` konnte ich etwa 437 Zyklen pro Systemcall messen, was bei 5.422 GHz (gemessen mittels `perf`) etwa **79.7ns** pro Systemcall entspricht. Demgegenüber ergaben die `clock_gettime`-Messungen etwa **98.6ns** pro Systemcall. Erwartungsgemäß ist das Ergebnis mit Ansatz 1 größer, da zumindest die zusätzliche Latenz der Schleife mitgemessen werden. Die restliche Differenz erklärt sich mutmaßlich dadurch, dass die Zeit, die mittels `clock_gettime` gemessen wird, auch weiterläuft, während ein Kontextwechsel stattfindet und ein anderer Prozess gescheduled wird (`perf` meldet 29 Kontextwechsel über die gesamte Laufzeit), während die kleinen Messungen mit Ansatz 2 vermutlich meist so "atomar" sind, dass wie vermutet wahrscheinlich nicht oft ein Kontextwechsel in einer laufenden Messung passiert.
Obwohl dies bezeugt, dass Ansatz 2 akkuratere Ergebnisse liefert, reicht Ansatz 1 auf jeden Fall aus, um die Latenzen nach oben abzuschätzen. Außerdem kann ich, durch den limitierten Zugriff auf entsprechende Hardware-Register auf ARM Plattformen, nur mit Ansatz 1 Vergleichbarkeit herstellen.

Aufgabe 3

Der Kontext bezeichnet in modernen Betriebssystemen den virtuellen Adressraum eines Prozesses inklusive des Stacks, Heaps und etwaiger Registerinhalte. Ein Kontextwechsel findet folglich immer dann statt, wenn ein laufender Prozess pausiert wird und nun ein anderer Prozess oder das Betriebssystem (z. B. bei manchen Systemcalls) auf dem selben Prozessor Berechnungen durchführen möchte. Technisch erfordert dieser Vorgang also, dass der alte Kontext (v. a. die Registerinhalte) gespeichert wird und der neue Kontext geladen wird [2]. Der Aufwand eines Kontextwechsels kommt also größtenteils durch Kopiervorgänge (Register $\leftrightarrow$ Speicher, evtl. auch Festplatte $\leftrightarrow$ Speicher, wenn Swap verwendet werden muss) zu Stande.
Ein Kontextwechsel kann absichtlich vom gerade ausgeführten Programm herbeigeführt werden (z. B. `sleep`, `thread_yield`, o. ä.), wenn es aktuell keinen Bedarf mehr hat, weitere Berechnungen auf der CPU durchzuführen (oder Ressourcen benötigt, die hinter einem Systemcall liegen, z. B. `open`). Auf der anderen Seite versucht das Betriebssystem die verwalteten Ressourcen möglichst fair auf die auszuführenden Programme zu verteilen, wenn ein Programm Ressourcen zu lange beansprucht (also ein Zeitquantum überschreitet), wird es vom Betriebssystem pausiert und stattdessen eine wartende Anwendung gescheduled. Schließlich kann ein Kontextwechsel durch *Preemption* erzwungen werden, also wenn ein anderes Programm mit höherer Priorität so schnell wie möglich Zugriff auf die CPU benötigt.

Messherausforderungen

Dadurch, dass der Scheduler des Betriebssystems jedes Mal, wenn ein Prozess die Kontrolle über die CPU abgibt, die freie Wahl hat, welcher Prozess als nächstes auf dieser CPU ausgeführt werden soll, ist das akkurate Messen von Kontextwechselzeiten nicht trivial. Idealerweise würde man sich wünschen, in einer Anwendung A die Zeit messen zu können, dann absichtlich die Kontrolle abzugeben und sofort wieder als nächstes gescheduled zu werden. Würde nun erneut die Zeit gemessen, beträgt die Zeitdifferenz genau einem Kontextwechsel $A \rightarrow A$. Es ist zwar durchaus möglich, die Wahl des Schedulers mittels der Priorität eines Prozesses zu beeinflussen; verhindern kann man jedoch kaum, dass nach einem Kontextwechsel einige der aktuell wartenden Prozesse ausgeführt werden (ein Blick in den Taskmanager offenbart bei sehr leichter Nutzung bereits 223 Prozesse mit 3820 Threads) und somit in der Zeit bis zur nächsten Messung inklusive sind.
Ein weiterer Aspekt, der eine akkurate Messung schwierig macht, ist das Scheduling auf Multicore-Prozessoren. Eventuell muss der zuletzt verwendete CPU-Kern von einem anderen Prozess mit höherer Priorität verwendet werden (z. B. für den Zugriff auf IO), ein anderer Kern wird frei, woraufhin der Prozess, in dem die Messung ausgeführt wird, auf diesen freien Kern migriert wird (was mit hohen Kosten verbunden ist); mehr dazu später. Standardmäßig wird im Scheduling eine *weak affinity* berücksichtigt, ein Prozess wird also vorzugsweise auf dem letzten Kern, auf dem er gelaufen ist, gescheduled, dies ist jedoch keine harte Begrenzung.

Mittels `taskset` bzw. `sched_setaffinity` kann zwar eine feste CPU Affinität festgelegt werden, was jedoch auch für noch längere Intervalle sorgen kann, in denen der Messprozess warten muss.

Erste Messungen

Gemessen wurden mittels *rtdsc* die CPU-Zyklen vor und nach einem Aufruf einer Kontextwechselinstruktion. Diese Messung wurde 1M mal durchgeführt und das Ergebnis über 10 Läufe gemittelt. Der Code wurde erneut mit `gcc -O2` kompiliert und auf System 2 getestet (nur hier hatte ich Zugriff auf `perf`, um die Zahl der tatsächlichen Kontextwechsel zu verifizieren). Die Ergebnisse finden sich in Tabelle 4.

Instruktion	**durchschnittliche Zyklen**	**durchschnittliche Zeit (ns)**
`sched_yield`	1382	254
`sleep(0)`	254903	46857
`sleep(0)` (CPU-Affinität)	246889	45384
`sleep(0)` (hohe Priorität)	65609	12060
`sleep(0)` (CPU-Aff + hohe Priorität)	65200	11985

Tabelle 4: Durchschnittliche Zeit für eine Kontextwechselinstruktion, gemessen in Zyklen. Die durchschnittliche Zeit in ns wurde berechnet mit einem gemittelten Takt von 5.44GHz.

In der Theorie würde sich die `sched_yield` Instruktion am besten für diesen Test eignen, da sie dafür gedacht ist, den aufrufenden Thread in die Warteschlange einzureihen und einem anderen Thread die CPU zu überlassen. Jedoch führt sie, wie bereits an der sehr kurzen Latenz ablesbar, meist nicht zu einem Kontextwechsel, wenn kein Prozess mit einer höheren Priorität in der Warteschlange ist. So wurden laut `perf stat` über den gesamten Test nur 39 der geplanten 10M Kontextwechsel ausgeführt.
Im Gegensatz dazu hat die `sleep(0)` Instruktion laut `perf stat` tatsächlich bei jedem Aufruf für einen Kontextwechsel gesorgt. Die Messungen (ohne erhöhte Priorität + feste CPU-Affinität benötigt eine Messung fast 4 mal so lange) bestätigen in diesem Fall auch meine Annahmen, dass beide Maßnahmen notwendig sind, um akkurate Messungen zu erhalten, da ansonsten zu viel Interferenz von außen (CPU-Wechsel + andere Prozesse) mitgemessen wird. Somit stellen die 12μs, die ich in meinem letzten Test messen konnte, die genauste Messung dar, die ich mit diesem Ansatz erzielen konnte.

Kontextwechsel durch IPC

In einem entsprechenden Stackoverflow Thread vermutete Mekki[6], dass ein Kontextwechsel mittels `sleep` länger braucht, als ein Kontextwechsel durch aufgebrauchtes Zeitquantum oder *preemption*, da in diesem Fall der Overhead des Timers und zusätzlicher Schedulingoperationen (für die Dauer des Timers muss der Prozess komplett aus der Warteschlange entfernt, danach wieder eingefügt werden) hinzukommen. Da sich in den *manpages* zum `sleep`-Systemcall keine Notizen zu einer Sonderbehandlung bei 0s als Argument finden, kommt dieser Overhead wahrscheinlich auch bei meinen Messungen zu Stande. Stattdessen schlug Mekki daher ein *Pingpong*-Szenario vor, in dem 2 Threads sich abwechselnd gegenseitig aus einem Wartezustand aufwecken. Der erste Thread weckt mittels `pthread_cond_signal` den zweiten Thread auf und wartet anschließend. Der zweite Thread wartet, bis er dieses Signal erhält und sendet dann ein Signal an den ersten Thread, um ihn wieder zu wecken. Für eine feste Zeit wird dann gemessen, wie viele Iterationen stattgefunden haben. Auch in diesem Fall kommt natürlich ein unvermeidbarer Overhead für Mutex und IPC zu Stande, der aber hoffentlich deutlich kleiner ist.
Den vorgeschlagenen Test (`cs_pipe.c` bzw. `cs_pipe_mac.c`) habe ich dann jeweils 10s auf allen Systemen ausprobiert (Tabelle 5). Mittels `perf stat` konnte ich verifizieren, dass die Zahl der gemessenen Kontextwechsel tatsächlich akkurat war.
Spannend ist hier, dass zwar alle Messungen in der selben Größenordnung sind, die beiden ARM Systeme (System 1 und 3) schneller sind als das x86 System, obwohl hier eine deutlich potentere CPU verbaut ist. Dieser Vorteil von ARM wird relativ gesehen noch größer, da die ARM CPUs nur mit 2.4GHz (System 3) bzw.

[6] `https://stackoverflow.com/a/304925`

Messszenario	System 1	System 2	System 3
regulär	776495 / 1288 ns	371361 / 2693 ns	387361 / 2580 ns
feste Affinität	-	396409 / 2523 ns	387198 / 2583 ns
hohe Priorität	-	442007 / 2262 ns	407267 / 2455 ns
Affinität + Priorität	-	607451 / 1646 ns	627913 / 1593 ns

Tabelle 5: Messungen des IPC Kontextwechsel Tests. Die erste Zahl gibt jeweils die Iterationen pro Sekunde, die zweite die mittlere Latenz pro Kontextwechsel in ns an. Auf macOS (System 1) stehen `taskset` und `chrt` nicht zur Verfügung.

3.2GHz (System 1) takten und damit weit vom Takt des x86 Systems (5.44GHz) entfernt sind. Vermutlich hat man sich mit ARM als neuerer Prozessorarchitektur einige Gedanken gemacht, den Kontextwechsel als potenziellen Flaschenhals zu optimieren. Außerdem ist hier erneut deutlich zu sehen, wie viel Interferenz ohne feste CPU-Affinität und mit niedriger Priorität mitgemessen wird.

Messungen mit einem Profiler

Zur Messung der Kontextwechsellatenz mittels Profiler hatte ich bereits die Datei `profile_sleep.c` angelegt, in der 100k `sleep(0)` Aufrufe erfolgen sollten. Dann habe ich nach Profilern gesucht, die ich einsetzen könnte und leider nicht sonderlich viel Glück:

- Der Profiler *tracy*[7] sah als multiplattform Profiler mit hoher Auflösung (5ns) sehr vielversprechend aus. Leider konnte ich ihn auf keiner Plattform installieren.
- Der speziell für AMD x86 Prozessoren entwickelte AMDμProf Profiler[8] hat sich leider als ziemlich ungenau herausgestellt. Das OS Tracing mit Messung der Kontextwechsel hat leider nicht funktioniert (angeblich immer 0 Kontextwechsel) und die Latenzen wären ohnehin nur in ms angegeben worden.
- `ftrace` scheint auf einem Ubuntu Live USB Stick nicht nutzbar zu sein
- `perf` ist theoretisch in der Lage dazu, direkt mit `sched:switch` Events zu arbeiten. In der standard Ubuntu-Version ist dies mangels *libtraceevent* jedoch nicht möglich.

Schließlich probierte ich *dtrace* aus, was als macOS Implementierung von *strace* / *perf* theoretisch auch auf Kontextwechsel-Events zugreifen können sollte. Erstes Hindernis: Um das Tool überhaupt zum Laufen zu kriegen, muss man zunächst die *system integrity protection* (SPI) für *dtrace* deaktivieren (das funktioniert nur im Recovery Modus). Mit dem dazugehörigen Skript `dtrace.d` erhielt ich dann die Ergebnisse, die in Tabelle 6 abgebildet sind. Da die gesamte Ausführung durch *dtrace* jedoch massiv verlangsamt wird, sind sämtliche Messwerte jedoch deutlich zu hoch und deshalb nicht brauchbar.

Programm	mittlere Kontextwechselzeit in ns
`cs_sleep.c`	13762
`cs_pipe.c`	121947
`cs_sched_yield.c`	16493

Tabelle 6: Mittlere Zeit pro Kontextwechsel (Intervall zwischen `sched::off-cpu` Event und `sched::on-cpu` Event) auf System 1

[7] `https://github.com/wolfpld/tracy`
[8] `https://www.amd.com/en/developer/uprof.html`

Fazit

Am verlässlichsten finde ich für dieses Experiment den IPC Ansatz. Auch, wenn hier Inkrements, Schleifen (vernachlässigbar) und IPC als zusätzlicher Overhead anfallen, ist der Test hier vermutlich recht realitätsnah. Die Ergebnisse (ca. 1-3 μs) decken sich außerdem besser mit den Tests, die man zu diesem Thema im Internet finden kann [6, 1], wo einstellige μs Latenzen ermittelt wurden.
Allerdings sieht man auch an diesen Werten, dass ein Kontextwechsel in der Tat ziemlich teuer ist (laut [1] etwa so teuer, wie das Kopieren von 64KB Daten im Hauptspeicher). Dies liegt unter anderem an den indirekten Kosten, welche u. U. auch auf Geräten ohne jegliche andere Last anfallen würde. Einerseits benötigen Berechnungen in einem Programm, das die Kontrolle über die CPU aufgibt (aufgeben muss) und erst Bruchteile einer Sekunde später wieder gescheduled wird, offensichtlich länger.
Andererseits ist es bei modernen CPUs essenziell, über Caches nachzudenken. Bereits innerhalb eines Algorithmus kann durch gute Cachelokalität (der Daten) die Performanz um ein Vielfaches steigen. Kommt es zu einem Kontextwechsel, infolgedessen der gesamte Cacheinhalt ausgetauscht wird (sowohl gecachte Codeinstruktionen als auch Daten), muss alles wieder aus dem Hauptspeicher (oder im schlimmsten voll sogar aus dem Sekundärspeicher) geladen werden, wodurch selbstverständlich die eigentlich durchzuführenden Berechnungen lange warten müssen. In einem entsprechenden Paper von 2007 [3] wurde gezeigt, dass sich bei einer größeren Menge von Programmdaten, die bei jedem Kontextwechsel wieder in den Cache geladen werden müssen, die mittlere Latenz von 38μs auf 203μs ca. verfünffacht.
Da der Cache auch bei der Migration eines Prozesses auf eine andere CPU neu befüllt werden muss, beeinflusst dies auch die Zeit, die der Kontextwechsel benötigt. Ein Experiment dazu [6] kommt auf eine ganze Größenordnung, die dies ausmacht.

Takeaways

- Ein Kontextwechsel kostet mindestens einige μs kann aber bei großen Datenmengen und häufigen Cache-Invalidierungen auch in den ms Bereich reichen.
- Wenige Kontextwechsel sind für einen Nutzer in den meisten Fällen vermutlich nicht erkennbar, passiert dies zu oft, kann die Performanz eines Programms vermutlich doch merklich darunter leiden.
- Insbesondere sollten nicht mehr Threads für rechenintensive Programme verwendet werden als physische CPUs vorhanden sind, da dies viele Kontextwechsel provoziert. Außerdem können mittels Buffering Kontextwechsel minimiert werden.
- Wenn möglich ist *user level scheduling* oft sinnvoll, da der Programmierer besser weiß, wie verschiedene Threads eines Programms sinnvoll zusammenarbeiten, als das Betriebssystem.

Literatur

[1] Bendersky, E. Measuring context switching and memory overheads for Linux threads - Eli Bendersky's website — eli.thegreenplace.net. `https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/`, 2018. [Accessed 29-11-2024].

[2] Hawthorn, J. e. a. Context Switching - OSDev Wiki — wiki.osdev.org. `http://wiki.osdev.org/Context_Switching`, 2023. [Accessed 23-11-2024].

[3] Li, C., Ding, C., and Shen, K. Quantifying the cost of context switch. In Proceedings of the 2007 Workshop on Experimental Computer Science (New York, NY, USA, 2007), ExpCS '07, Association for Computing Machinery, p. 2–es.

[4] Sauthoff, G. On the Costs of Syscalls — gms.tf. `https://gms.tf/on-the-costs-of-syscalls.html`, 2021. [Accessed 22-11-2024].

[5] Soller, S. Measurements of system call performance and overhead - Arkanis Development — arkanis.de. http://arkanis.de/weblog/2017-01-05-measurements-of-system-call-performance-and-overhead, 2017. [Accessed 22-11-2024].

[6] tsuna. How long does it take to make a context switch? — blog.tsunanet.net. https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html, 2010. [Accessed 29-11-2024].