Maximilian Späth
1456880

Betriebssysteme, Übungsblatt 1, Winter 2024

Link zum Repository : Siktik/OperatingSystemsLecture: A repository containing solutions for the Operating Systems Lecture (WS 2024/2025) by Prof. Dr. Sturm at University Trier

Aufgabe1:

Den SystemCall Wrapper für read() kann man in der GNU C Library finden. Die Methode ruft SYSCALL_CANCEL(*read, fd, buf, nbytes*) auf, welche ein makro für den eigentlichen SystemCall syscall(*SYS_read, fd, buf, nbytes*) ist. Damit findet der Wechsel zum Kernel Modus statt und syscall() leitet über zur sys_read methode.

```
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
	struct fd f = fdget_pos(fd);
	ssize_t ret = -EBADF;

	if (fd_file(f)) {
		loff_t pos, *ppos = file_ppos(fd_file(f));
		if (ppos) {
			pos = *ppos;
			ppos = &pos;
		}
		ret = vfs_read(fd_file(f), buf, count, ppos);
		if (ret >= 0 && ppos)
			fd_file(f)->f_pos = pos;
		fdput_pos(f);
	}
	return ret;
}

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
	ssize_t ret;

	if (!(file->f_mode & FMODE_READ))
		return -EBADF;
	if (!(file->f_mode & FMODE_CAN_READ))
		return -EINVAL;
	if (unlikely(!access_ok(buf, count)))
		return -EFAULT;

	ret = rw_verify_area(READ, file, pos, count);
	if (ret)
		return ret;
	if (count > MAX_RW_COUNT)
		count =  MAX_RW_COUNT;
```

```
        if (file->f_op->read)
                ret = file->f_op->read(file, buf, count, pos);
        else if (file->f_op->read_iter)
                ret = new_sync_read(file, buf, count, pos);
        else
                ret = -EINVAL;
        if (ret > 0) {
                fsnotify_access(file);
                add_rchar(current, ret);
        }
        inc_syscr(current);
        return ret;
}
```

Innerhalb des Kernels verifiziert *sys_read* den Integer *fd*, der z.B. eine bestimme File identifiziert und konvertiert *fd* durch den Aufruf von *fdget(fd)* in eine Kernel Datei Struktur. Folgend wird die virtual file system layer function *vfs_read* aufgerufen, welche den read Befehl an das richtige Dateisystem gibt. Das entsprechende Dateisystem liest Daten in den Buffer *buf* und gibt den Buffer zurück an *sys_read*. Abschließend wird wieder in den Benutzermodus zurückgewechselt und das Ergebnis an *__libc_read* übergeben.

Um besser zu verstehen was in dem KernelCode passiert wurde ChatGPT zur Unterstützung verwendet.

Aufgabe2:

Ansatz:

Der experimentelle Ansatz, um die minimale Latenz bei der Ausführung eines System-Calls zu ermitteln basiert auf der Verwendung des QueryPerfomanceCounters von Microsoft in C++. Der Performance Counter gibt einen Zeitstempel <1µs zurück, wodurch sehr genaue Zeiten ermittelt werden können. Demnach wird ein Zeitstempel vor dem SystemCall und ein Zeitstempel nach dem SystemCall genommen und die Differenz von Start und Ende als Zeitspanne bzw. minimale Latenz für den SystemCall verwendet. Es werden zwei Methoden verwendet deren Latenz mithilfe des QueryPerfomanceCounters in zwei Experimenten gemessen.

Experimente:

Im ersten Experiment wird die Methode GetSystemTimeAsFileTime verwendet, welche die Systemzeit ausliest. Die Methode arbeitet „in memory“ und es sind daher keine Zugriffe auf Hardware notwendig, was Kontextwechsel zur Folge haben könnte. Es werden 50 Durchläufe gemacht, in denen die Methode einmal angefragt wird. In einem zweiten Experiment wird die Methode *CreateFile* verwendet. Diese benötigt den Zugriff auf externe Speicher wodurch eine höhere Latenz zu erwarten ist. Des Weiteren können Kontextwechsel eine Rolle spielen. Es werden zwei Datensätze erzeugt, die aus je 50 Durchläufen also 50 Datenpunkten bestehen. Im ersten Datensatz wird die Methode einmalig aufgerufen und die Zeit als Datenpunkt abgelegt. Da zu erwarten ist, dass während dem Aufruf Kontextwechsel stattfinden, wird im zweiten Datensatz mithilfe einer Schleife

ein Mittelwert über zehn Aufrufe gebildet. Für jeden dieser Aufrufe wird die Latenz ermittelt und der Mittelwert dieser 10 Anfragen bildet einen Datenpunkt im zweiten Datensatz.

Hardwarekomponenten:

- Ryzen 3700x, 3.7GHz Basistakt, 4,1 GHZ bei voller Last
- Samsung Evo 970 Pro SSD
- MPG B550 GAMING CARBON WIFI

Ergebnisse:

Die GetSystemAsFileTime Methode hat wie zu erwarten war eine sehr geringe Latenz, die z.T. sogar unterhalb der Auflösung des QueryPerfomanceCounters liegt. Bei manueller Betrachtung fällt auf, dass ungefähr ein Drittel der Datenpunkte 0.0 µs ergeben. Die geringe Latenz ist darauf zurückzuführen, dass die Operation kaum Aufwand benötigt und in memory arbeitet. Der Großteil der Werte konzentriert sich auf 0.0 bis 0.1 µs, wie der Abbildung 1 zu entnehmen ist.

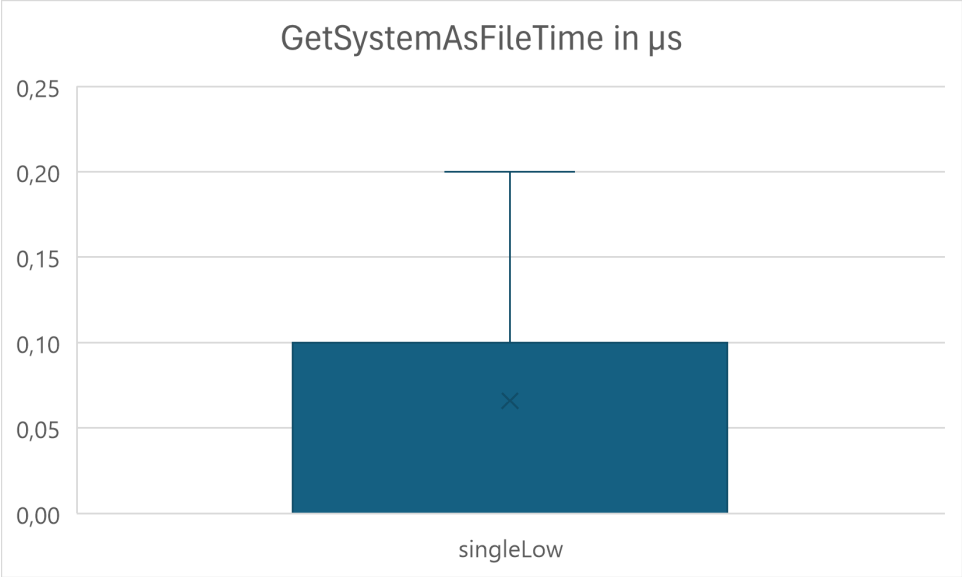


Abbildung 1: Ergebnisse Experiment 1; GetSystemAsFileTime System Call Latenz in µs

Die CreateFile Methode benötigt bei den einzelnen Anfragen (singleHigh) im Mittel 194,4µs mit einer Standardabweichung von 38,53µs während die bereits gemittelten 10 Anfragen (multiHighAVG) des zweiten Datensatzes im Mittel 176,7µs mit einer Standardabweichung von 24,77µs benötigt. Die höheren Werte im Vergleich zu denen der GetSystemAsFiletime sind darauf zurückzuführen, dass bei der Operation Create File der Zugriff und kommunikative Austausch mit dem externen Speicher notwendig ist. Bei einer Dauer von durchschnittlich 176,7µs ist durchaus vorstellbar, dass auch Kontextwechsel stattfinden, in denen anderen Prozessen Informationen, die zum Beispiel eine ähnlich kurze Dauer wie GetSystemAsFileTime haben, zugekommen lassen wird. Dass der erste Datensatz im Mittel ca. 20µs länger benötigt, lässt sich vermutlich mit der geringeren Anzahl an Datenpunkten welche eine mögliche höhere Auslastung des Systems zum Zeitpunkt der Erfassung wiederspiegelt.

Maximilian Späth
1456880

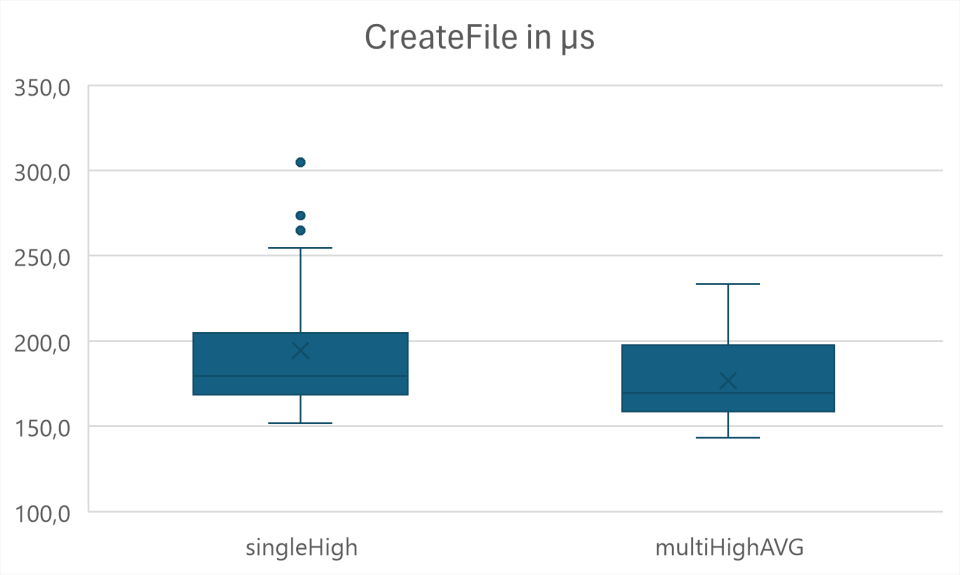


Abbildung 2: Ergebnisse Experiment 2; CreateFile System Call Latenz in µs

Die Schlussfolgerung für die niedrigeren Ergebnisse bezüglich des Mittelwerts und der Standardabweichung würde sich demnach mit der höheren Anzahl an Datenpunkten erklären lassen. Da die Erfassung über einen längeren Zeitraum als die des esten Datensatzes, wurden Datenpunkte unter höherer und geringerer Last aufgenommen. Daraus ließe sich schlussfolgern, das die Werte der zweiten Erfassung eine generelle Aussage über die Dauer der Methode auf dem System treffen. Allgemein wurden jedoch beide Datenerfassungen mit den gleichen sichtbar geöffneten Programmen durchgeführt, wodurch sich nicht sagen lässt, welche Hintergrundprozesse ins Gewicht gefallen sein könnten.

Aufgabe3:

Generell werden drei Arten von Kontextwechseln betrachtet.

1. Prozess/Task Scheduling: Laufende Prozesse werden angehalten und deren Zustand abgelegt um einen oder mehrere andere Prozesse durchzuführen. Dies wird durch Prioritäten und Zeitfenster bestimmt die Prozessen zugeordnet sind.
2. Benutzer-Modus/ Kernel-Modus Wechsel: Wenn ein Prozess einen SystemCall macht, z.B. das Lesen systeminterner Daten oder des virtuellen Speichers findet ein wechsel vom Benutzer Modus zum Kernel Modus statt. Ein weiterer Wechsel zurück zum Benutzer-Modus folgt, wenn die Daten von der Kernel zurückgegeben werden.
3. Interrupts: Treten Fehler in der Hardware auf wechselt der Kontext automatisch zu der einer Komponente der Hardware welche den Fehler schneller behandeln kann.

Um zuverlässig Kontextwechsel messen zu können ist es sinnvoll diese gezielt auslösen zu können. Das Problem am Prozess Scheduling ist, dass auf einem Rechner unzählige Prozesse auch abseits von geöffneten Programmen laufen. Bei einer Operation wie CreateFile aus Aufgabe 2 z.B. findet zum einen der Wechsel vom Benutzer-Modus zum Kernel-Modus und zurück statt, jedoch werden auch Subprozesse ausgeführt wie das allokieren von Speicherplatz auf dem externen Speicher. Wie in Aufgabe 2 zu erkennen ist, benötigt dies im Schnitt ca 174ms, während Operationen mit geringem Overhead wie z.B. der GetSystemTimeAsFileTime sehr geringe Latenzen von 0,1µs haben. Es ist daher durchaus wahrscheinlich, dass während der createFile Operationen einige Kontextwechsel

stattfinden, in welchen der Prozess der Datei Erstellung pausiert wird und andere Prozesse mit Informationen versorgt werden, da diese z.B. eine höhere Priorität und geringere Durchführungszeit haben. Es ist hier daher schwer die durchschnittliche Zeit eines Kontextwechsels zu bestimmen, da beliebig viele auftreten können. Interrupts zu triggern um einen Kontextwechsel auszulösen und dann noch die entsprechende Zeit des Wechsel zu bestimmen obwohl das System einen Fehler handelt dürfte ebenfalls wenig zielführend sein. Am sinnvollsten, zur Beantwortung der Frage wie lange im Schnitt ein Kontextwechsel dauert, ist es gezielt nur einen, bzw. zwei, Kontextwechsel zu triggern. Fordern wir zum Beispiel eine systeminterne Information an, z.B. durch GetSystemTimeAsFileTime oder die Abfrage der ID des derzeitigen Prozesses mit GetCurrentProcessId, wird ein Wechsel vom Benutzer-modus zum kernel-modus getriggert, die Information besorgt und der Kontext wieder zum benutzer-modus gewechselt. Da diese Operation in-memory erfolgt und daher sehr schnell erledigt ist, kann ausgeschlossen werden, dass Kontextwechsel aufgrund von Taskscheduling stattfinden. Etwaige Verzögerungen sind entweder darauf zurückzuführen, dass ein anderer Prozess Vorrang vor der Anfrage durch getSystemTimeAsFileTime hat, oder die Zeit, die der Kontextwechsel mitsamt der Besorgung der Information benötigt. Wie bereits in Aufgabe 2 betrachtet, liegt die Zeit die eine getSystemTimeAsFileTime benötigt jedoch häufig unter der Auflösung von 0,1µs des QueryPerfomanceCounters. Ähnliche Resultate zeigen frühe Versuche für GetCurrentProcessId im gleichen Versuchsaufbau wie mit GetSystemTimeAsFileTime, welche ebenfalls ein „in memory“ System Call ist. Um die Zeit des Kontextwechsels von Benutzer-modus zu Kernel-modus zu bestimmen wäre daher eine noch feinere Auflösung notwendig. Eine noch feinere Auflösung könnte man mit dem „Read Time-Stamp Counter“ (rdtsc) erhalten. Auf x86 Prozessoren kann so der Time-Stamp Counter der CPU ausgelesen und, mithilfe der Taktfrequenz die Zeit bestimmt werden die vergangen ist. Dies ermöglicht eine Auflösung abhängig von der Taktfrequenz was bei 4Ghz ca. 0.25ns sind.
Aufgrund weiterer Recherche bezüglich Kontextwechseln die einen Referenzwert für die durchschnittliche Dauer von Kontextwechseln von 5µs nennen, und einige dieser Quellen den angesprochenen Kontextwechsel zwischen User- und System-Kernel nicht, wohl aber das Taskscheduling und Interrupting aufführen, wird eine Möglichkeit gesucht einzelne Kontextwechsel im Sinne des TaskScheduling durchzuführen.

Experiment Sleep(0):
Eine Möglichkeit besteht darin, den aktuellen Thread für 0 Sekunden schlafen zu lassen. Dies gibt ein Signal an die CPU, das der aktuelle Thread pausiert und gewechselt werden kann, während sich dieser sogleich wieder in die Warteschlange für die CPU einreiht. In C++ wird dies mithilfe der thread library und dem Befehl std::this_thread::sleep_for(0) erreicht. In einem von ChatGPT erstellten Programm, wird zudem die chrono library verwendet, welche eine Auflösung im Nanosekunden Bereich ermöglicht.
Hardwarekomponenten:

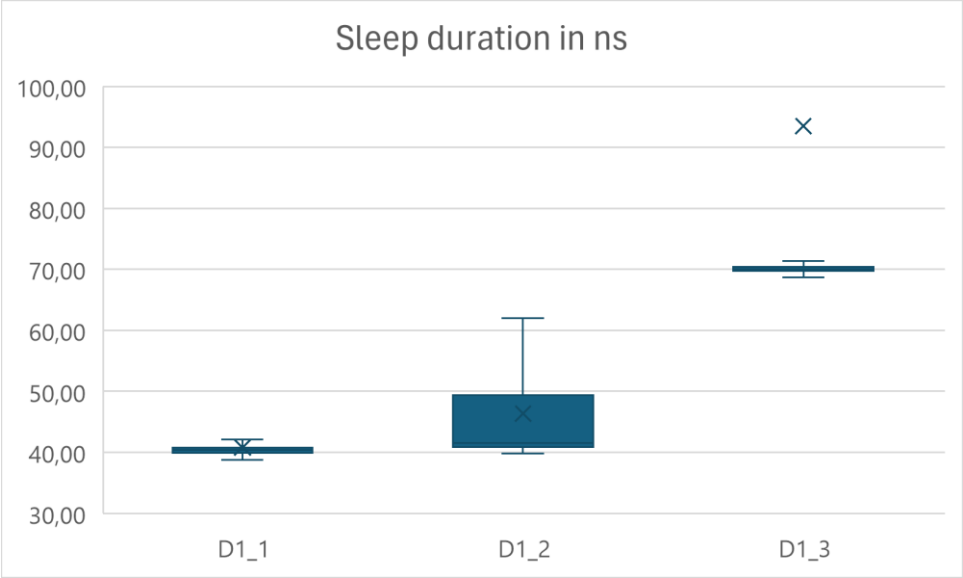
- Ryzen 3700x, 3.7GHz Basistakt, 4,1 GHZ bei voller Last
- Samsung Evo 970 Pro SSD
- MPG B550 GAMING CARBON WIFI

Aufbau:

Maximilian Späth
1456880

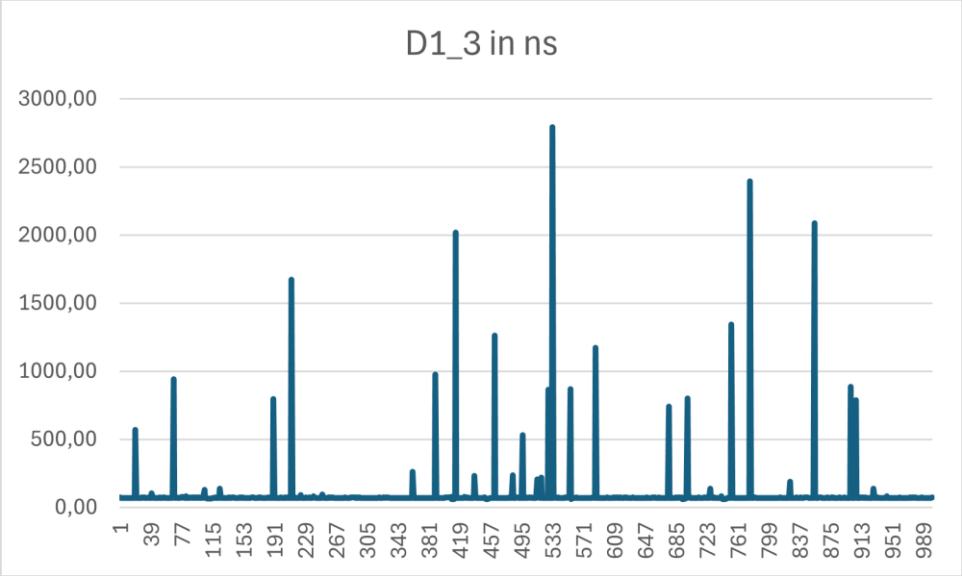
Um die Dauer der sleep(0) Operation und dem damit verbundenen Kontextwechsel zu testen, werden drei Durchläufe durchgeführt. Im ersten Durchlauf (D1.1) wird das System keinerlei zusätzlichem Stress ausgesetzte, es sind nur notwendige Programme geöffnet und Hintergrundprozesse möglichst reduziert. Im zweiten Durchlauf (D1.2) wird in einem Python Programm in der PyCharmIDE in einer Endlosschleife der String „help“ geprintet. In einem dritten Durchlauf (D1.3) wird mithilfe eines Programms „StressMyPC“ v5.44 alle Kerne der CPU auf 100% gestresst.
In jedem Durchlauf werden 1000 Datenpunkte bestimmt. Ein Datenpunkt ist der Mittelwert aus 10000 Iterationen. Der Code für die Erfassung ist simpel und beinhaltet eine Schleife, welche die 10000 Iterationen durchführt, in jeder Iteration eine Startzeit mithilfe der chrono library festhält, die sleep Methode ausführt und die Endzeit festhält und so die Gesamtzeit hochzählt. Am Ende wird der Durchschnitt gebildet und ein Datenpunkt zurückgegeben.

Ergebnisse
Für den ersten Datensatz D1_1 resultierte ein Mittelwert von 40,8ns und eine Standardabweichung von 2,54ns. Für den zweiten Datensatz D1_2 resultierte ein Mittelwert von 46,37ns und 10,491ns. Für den dritten Datensatz D1_3 resultierte ein Mittelwert von 93,51ns und eine Standardabweichung 182,069ns.
Der erste Datensatz hat eine sehr geringe Standardabweichung was darauf hin deutet das entweder immer die gleichen Interferenzen vom System kamen oder kaum Interferenzen vorhanden waren. Der Mittelwert von 40,8ns erscheint ebenfalls sehr gering, da im Internet bei Kontextwechseln von einer Zeit von durchschnittlich 5µs gesprochen wird, scheint die Simulation eines Kontextwechsels durch sleep nicht zu funktionieren wie gewünscht. Es ist theoretisch möglich das die Methode sleep zwar bewirkt das der Thread seinen Slot an der CPU kurzzeitig abgibt, aber diesen sogleich wiederbekommt, wenn keine anderen Prozesse warten. Demnach hätte kein Kontextwechsel stattgefunden, was zu der geringen Standardabweichung und der geringen Zeit für die Durchführung passt.



Im zweiten Datensatz, welcher unter leichter Last entstanden ist, sind Mittelwert und Standardabweichung leicht erhöht. Es zeigt sich hier die Auswirkung der leichten Last durch das Python Programm. Es gibt wenige Ausreißer, die bei ca 180ns liegen, was aber immer

noch weit entfernt von den berichteten 5µs für Kontextwechsel entfernt ist. Es wäre daher möglich das aufgrund des dauerhaften Schreibens eines Outputs durch das Python Programm das System insgesamt langsamer war, ein Kontextwechsel an dem entsprechenden Slot des Threads jedoch nicht stattgefunden hat.
Im dritten Datensatz, welcher unter voller Last erhoben wurde, zeigen sich starke Ausreißer, wie in dem Diagramm für D1_3 dargestellt ist. Es gibt hier einige Messwerte zwischen 500 und 3000. Wenn man betrachtet das dies bereits Mittelwerte aus 10000 Iterationen sind, scheinen hier regelmäßig Kontextwechsel stattgefunden zu haben, welche wohl durchaus im Bereich von 5µs gelegen haben können. Allgemein ist es aber schwer eine genaue Aussage zu treffen, wie lange ein Kontextwechsel dauert, da unklar ist wie viele Kontextwechsel stattgefunden haben.



Evaluierung von Kontextwechseln mittels PIX
Um Vergleichswerte zu erhalten wurde das Tool PIX von Microsoft verwendet. Die Funktion der Timing Captures ermöglicht die genaue Betrachtung von Kontextwechseln. Hierzu wurde das Programm lediglich dahingehen angepasst, dass der thread.sleep(0) Befehl in einer while Schleife dauerhaft aufgerufen wurde. Indem die native ThreadID zu beginn ausgegeben wird, kann der entsprechende Thread in PIX betrachtet werden. Die Evaluation findet ohne Last statt, Vergleiche werden daher zu dem ersten Datensatz gezogen für den eine durchschnittliche Zeit für einen Kontextwechsel von 40,8ns ermittelt wurde.
Was zuerst auffällt bei der Betrachtung der Kontextwechsel ist, dass der Thread teilweise bis zu einer Sekunde ununterbrochen läuft. Das deutet darauf hin, dass der Ansatz mit sleep(0) häufig nicht zu einem Kontextwechsel geführt hat. Eine Erklärung hierfür ist, dass Thread.sleep(0) zwar ein Zeichen an die CPU gibt dass der Thread seinen Slot frei macht und sich wieder in die Warteschlange einreiht, bei 16 Kernen ohne Last auf dem System es aber allgemein wenige Threads in den Wartelisten gibt. Dadurch wird der Kern für den der betrachtete Thread sich in die Warteschlange eingereiht hat gar nicht erst belegt und kommt sogleich wieder dran. Da eine Wartezeit von 0ms angegeben ist, ist der Übergang entweder so fein granular das kein Kontextwechsel betrachtet wird, oder aber die CPU bzw. der Kern erkennt dass der gleiche Thread wie zuvor geladen wird, und dadurch ebenfalls kein Kontextwechsel wahr genommen wird. Was das eigens geschriebene Programm in

diesem Fall an Zeit stoppen würde, ist jene die notwendig ist um den C++ Befehl zum Kernel zu transportieren und entsprechend die Information zurückzuerhalten, dass der Thread wieder gescheduled worden ist.
Für die Kontextwechsel die durchgeführt wurde fällt auf, dass häufig zugleich mehrere Kontextwechsel innerhalb von 50-200ms, an der Zahl 2-5 durchgeführt wurden. Des Weiteren fällt auf, dass ein Kontextwechsel hier immer zur Folge hat, dass der Thread auf einem anderen Kern weiterläuft. Betrachtet man manuell einzelne Kontextwechselzeiten liegen diese im Schnitt bei 8ns. Ein Ausreißer findet sich bei 81ns. Generell liegen die 8ns weit unter den 40ns die im eigenen Programm gemessen wurden, jedoch ist es wie zuvor benannt durchaus möglich, dass die Annahme, jeder Aufruf von Thread.sleep(0) würde zu einem Kontextwechsel führen falsch. Da PIX vermutlich ein unverbesserlich genaues Messverfahren hat, ist nicht auszuschließen das die verwendete Methode des Chronographen im eigenen Programm overhead erzeugt und nicht so fein granular die Zeiten bestimmt wie PIX.
Wenn man Vergleiche zur Literatur zieht[1][2], benötigen Kontextwechsel im Schnitt 3-7µs. Der Unterschied liegt vor allem darin, dass das eigene Programm keinen wirklichen state hat der gespeichert oder geladen werden muss. Während in der Literatur häufig Arrays allokiert werden und mithilfe von Pipes sich zwei Prozesse gegenseitig aufwecken, hat das eigene Programm keinen sichtbaren workload oder state der gespeichert werden müsste.

Indirekte Kosten
Zusätzlich zu den unmittelbaren Zeitkosten eines Kontextwechsels – wie dem Speichern und Laden des CPU-Kontexts oder dem Umschalten von Threads – entstehen indirekte Kosten, die schwerer zu quantifizieren sind, aber dennoch eine erhebliche Auswirkung auf die Performanz haben können. Diese indirekten Kosten resultieren aus Effekten, die durch die Mechanismen des Kontextwechsels selbst sowie die Eigenschaften moderner CPU-Architekturen hervorgerufen werden.
Cache Invalidierung. Ein Kontextwechsel kann die Inhalte der CPU-Caches (z. B. L1-, L2- und L3-Cache) ungünstig beeinflussen. Diese Caches sind darauf optimiert, die Daten und Befehle des aktuell laufenden Threads bereitzustellen. Wird ein Thread durch einen Kontextwechsel unterbrochen, können die Daten des neuen Threads die Inhalte des Caches überschreiben. Wenn der ursprüngliche Thread später wieder ausgeführt wird, müssen die zuvor ausgelagerten Daten erneut in den Cache geladen werden. Im Kontext des eigenen Programms könnte die Cache-Invalidierung eine Rolle spielen, wenn das Programm nicht allein auf einem isolierten Kern läuft. Zwar war die CPU in während der Durchführung nicht stark ausgelastet, aber jeder Cache-Reload erzeugt Latenz, die sich auf die gemessenen Werte auswirken könnte.
Scheduler-Entscheidungen. Der Betriebssystem-Scheduler benötigt Zeit, um zu entscheiden, welcher Thread als nächstes ausgeführt wird. Dieser Entscheidungsprozess hängt von der Priorität und dem Status anderer Threads ab. Bei geringer Systemlast kann dies minimal sein, aber es gibt dennoch einen Overhead für das Management der Warteschlangen und den Übergang zwischen Threads. Die gemessenen 40,8ns könnten Scheduler-Overheads enthalten, da das eigene Programm darauf angewiesen ist, dass der Scheduler den Thread

[1] Tsuna's blog: How long does it take to make a context switch?
[2] switch.dvi

erneut einplant. Wenn keine anderen Threads mit hoher Priorität vorhanden sind, bleibt der Overhead minimal.

Effekte des Kernel-Modus-Wechsels. Das Aufrufen von thread.sleep(0) führt zu einem Wechsel in den Kernel-Modus, um die CPU über den Freigabezustand des Threads zu informieren. Ein solcher Moduswechsel erfordert Zeit für den Übergang zwischen User- und Kernel-Space sowie für das Bearbeiten von Interrupts. Das eigene Programm misst diesen Zeitaufwand definitiv mit, während dies bei PIX vermutlich nicht der Fall ist.

Fehlende Last und Thread-Lokalisierung. Da das Experiment in einer Umgebung mit geringer Systemlast durchgeführt wurde, konnten indirekte Effekte minimiert werden. Ein relevanter Punkt ist jedoch, dass thread.sleep(0) auf einem Mehrkernsystem nicht zwangsläufig zu einem Kontextwechsel führt, wie bereits geschrieben wurde. Dies könnte dazu führen, dass die gemessenen Zeiten nicht den tatsächlichen Kosten eines vollständigen Kontextwechsels entsprechen. Die geringen Systemanforderungen im Experiment führen vermutlich dazu, dass die CPU den gleichen Thread auf demselben Kern sofort wieder ausführt. Dadurch bleiben zwar viele der indirekten Kosten (z. B. Cache-Effekte) gering, dies resultiert aber auch in den großen Diskrepanzen zwischen den eigens und von PIX gemessenen Werten.

Abschätzung der Auswirkungen auf die Effizienz der Anwendungsausführung

Unabhängig vom eigenen Programm, welches bezüglich der Ermittlung von Kontextwechselzeiten definitiv Schwächen aufweist, können die indirekten Kosten eines Kontextwechsels die Performanz einer Anwendung erheblich beeinträchtigen, insbesondere wenn viele Threads konkurrieren oder ein hoher Grad an Multitasking vorliegt. In einer realen Anwendungssituation können die folgenden Auswirkungen auftreten:

1. **Verringerte Cache-Effizienz**: Häufige Kontextwechsel können zu einem Anstieg der Cache-Miss-Raten führen, was die CPU-Auslastung und die Latenzzeit erhöht.
2. **Erhöhter Scheduler-Overhead**: In Szenarien mit hoher Thread-Anzahl wächst der Aufwand für die Planung exponentiell, was zu spürbaren Verzögerungen führen kann.
3. **Zusätzliche Speicherzugriffe**: TLB-Flushes und erneute Seitenladeoperationen belasten die Speicherbandbreite und erhöhen die Zugriffszeiten.