Software Architecture For Enterprises
Übung 1

Aufgabe 1 *System-Call*

Ich habe mich für den System-Call time() entschieden. Da ich einen Windows-Rechner besitze, habe ich mich dazu entschieden, WSL (Ubuntu) für die Aufgabe zu nutzen. Deshalb, musste zunächst der Linux-Kernel heruntergeladen werden, um auf diesen zuzugreifen.

Im Gegensatz zu write() und read() befindet sich time(), neben vielen anderen System-Calls, nicht unter linux/fs, sondern linux/kernel/time. Der System-Call für time() sieht folgendermaßen aus:

```
/*
 * sys_time() can be implemented in user-level using
 * sys_gettimeofday().  Is this for backwards compatibility?  If so,
 * why not move it into the appropriate arch directory (for those
 * architectures that need it).
 */
SYSCALL_DEFINE1(time, __kernel_old_time_t __user *, tloc)
{
        __kernel_old_time_t i = (__kernel_old_time_t)ktime_get_real_seconds();

        if (tloc) {
                if (put_user(i,tloc))
                        return -EFAULT;
        }
        force_successful_syscall_return();
        return i;
}
```

Den System-Call-Header kann man in usr/include/time.h finden. Dies ist eine Besonderheit von time(). Viele andere System-Calls sind unter /usr/include/unistd.h zu finden.

```
#ifndef __USE_TIME_BITS64
/* Return the current time and put it in *TIMER if TIMER is not NULL.  */
extern time_t time (time_t *__timer) __THROW;
```

Bei einem Aufruf von time(null) in einer Anwendung, wird die von der funktion time(null) die aktuelle Zeit zurückgegeben. Die Funktion ruft den time()-System-Call auf und gibt die aktuelle Zeit in Sekunden seit dem Epoch-Zeitpunkt (1. Januar 1970) zurück.

Dieser Ablauf ist für die meisten System-Calls ähnlich. Die meisten grundliegenden System-Call-Wrapper sind unter /usr/include/unistd.h (oder /usr/include/time.h) zu finden und bilden dadurch eine benutzerfreundlichere Schnittstelle für Nutzer/Entwickler zu den eigentlichen System-Calls im Kernel.

Unter /linux/arch/x86/entry/syscalls/ gibt es die beiden Look-Up-Tabellen syscall_32.tbl und syscall_64.tbl. Die Tabellen ordnen die System-Call-Wrapper den entsprechenden Systen-Call-Funktionen des Kernels zu. In syscall_64.tbl sind viele der in syscall_32.tbl definierten System-Calls ebenfalls vorhanden. Time() jedoch ist nur in syscall_32.tbl, was an der Rückwärtskompatibilität liegt. Die moderne Alternative zu time ist z.B. clock_gettime(), da diese präziser ist.

Aufgabe 2 *System-Call-Latenz*

Um die tatsächliche Zeit eines System-Calls zu ermitteln, sollte man einen System-Call verwenden, der einen minimalen Overhead hat, da so die Latenz möglichst gering sein sollte.

Um eine möglichst präxise Zeitmessung zu haben, sollte clock_gettime() statt time() verwendet werden, da hier auch Nonosekunden angezeigt werden

Damit man einen Annäherungswert für den System-Call hat, sollte dieser sehr häufig (1.00.000) ausgeführt werden. Dadurch können Schwankungen und Ungenauigkeiten minimiert werden.

Das Programm das ich hierfür erstellt habe heißt “Betriebssysteme_Aufgabe2” und es berechnet die Zeit, die kürzeste Zeit, die der Programmaufruf bei allen Versuchen benötigt hat. Bei mehrmaligem Ausführen dieses System-Calls habe ich eine Bestzeit von 64 ns und im schlechtesten Fall eine Zeit von 69 ns bekommen.

Führe ich den System-Call ohne Wrapper auf (siehe Betriebssysteme_Aufgabe2_ohne_Header.c”), dann erhalte ich tatsächlich bessere Werte. Nun ist die Zeit nie schlechter als 65 ns gewesen. Die Bestzeit von 64 ns konnte sich jedoch nicht verbessern.

Aufgabe 3 *Kontextwechsel*

Da bei den vorherigen Aufgaben Linux genutzt werden sollte, habe ich mich dazu entschieden, die Kontextwechsel auf Windows 11 zu analysieren. Hierfür habe ich zunächst ein Programm erstellt, dass die Kontextwechsel auf Windows 11 errechnet. Hierfür vergleicht das Programm “Betriebssysteme_Aufgabe3_W11_Unterschied.c” die durchschnittliche Dauer des Aufrufens einer Dummy-Funktion, mit und ohne Kontextwechsel, wobei für den Kontextwechsel ein Thread erstellt wird.

Hierbei ergeben sich extrem hohe Unterschiede. Bei Kontextwechsel wurden Durchschnittlich 63702,90 ns benötigt. Ohne Kontextwechsel wurden nur 21,11 ns pro Aufruf benötigt. Dadurch ergibt sich eine isolierte Kontextwechseldauer von 6381,80 ns.

Indirekte Kosten eines Kontextwechsels

Zusätzlich zu den unmittelbaren Zeitkosten eines Kontextwechsels, die sich aus der Ausführung der notwendigen Aufgaben direkt ergeben, entstehen auch indirekte Kosten, die die Anwendungsperformanz negativ beeinflussen und schwer quantifizierbar sind:

1. **Cache-Verluste**:
    a. Beim Kontextwechsel können CPU-Caches ungültig werden, was zu einer erhöhten Cache-Miss-Rate führt. Dies bedeutet, dass Daten erneut aus dem Hauptspeicher geladen werden müssen, was die Zugriffszeiten verlängert und die Anwendung verlangsamt.
2. **TLB-Einträge (Translation Lookaside Buffer)**:
    a. Der TLB speichert die Übersetzungen von virtuellen zu physischen Speicheradressen. Kontextwechsel können TLB-Einträge ungültig machen, was zusätzliche Speicherzugriffszeiten verursacht und die Effizienz der Speicherzugriffe reduziert.
3. **Erhöhte Systemlast**:
    a. Häufige Kontextwechsel erhöhen die Systemlast, da der Scheduler des Betriebssystems mehr Arbeit leisten muss, um die Prozesse und Threads zu verwalten. Dies kann die Gesamtleistung des Systems beeinträchtigen, insbesondere bei hoher Prozessorauslastung.
4. **Scheduling-Overhead**:
    a. Der Betriebssystem-Scheduler verursacht Overhead, wenn er Entscheidungen darüber trifft, welcher Prozess oder Thread als nächstes ausgeführt wird. Dieser Overhead kann die Effizienz der Anwendungsausführung weiter verringern und zu einer zusätzlichen Belastung des Systems führen.

Auswirkungen auf die Anwendungsperformanz

Die oben genannten indirekten Kosten führen zu einer erhöhten Systemverzögerung und verringern die Effizienz der Anwendungsausführung. Anwendungen, die stark von häufigen Kontextwechseln betroffen sind, können eine schlechtere Performance und längere Reaktionszeiten aufweisen. Das Verständnis und die Minimierung der Kontextwechsel und deren indirekte Kosten sind daher entscheidend für die Optimierung der Anwendungsperformance.

Ich habe für als Standardwerkzeug “Windows Performance Analyzer” genutzt. Die Auswertung hiervon gibt einen anderen Wert für die durchschnittliche Zeit von 81692 ns im Vergleich zu meinem Programm, das beim Laufen des Performance Analyzers auch länger brauchte als zuvor und eine Zeit von 89833 ns benötigte. Man könnte also meinen, dass dieses Standardwerkzeug fast so hohe Kosten hat, wie der Kontextwechsel. Der Unterschied bei der Kontextwechselzeit kann darauf zurückgeschlossen werden, dass das Tool die Kontextwechsel besser loggen kann und somit genauer ist.