

Betriebssysteme

Übung 1 - System Calls

Narek Grigoryan

Table of Contents

Betriebssysteme.....	1
Narek Grigoryan	1
Aufgabe 1: System-Call	2
Einleitung.....	2
Vorgehen.....	2
Aufgabe 2: System-Call-Latenz	6
Einleitung.....	6
Vorgehen.....	6
Ergebnisse.....	7
Diskussion	7
Aufgabe 3: Kontextwechsel.....	8
Einleitung.....	8
Vorgehensweise	9
Ergebnisse.....	10
Diskussion	11
Persönliches Fazit.....	11

Aufgabe 1: System-Call

Einleitung

In dieser Aufgabe bestand die Zielsetzung darin, den Quellcode eines System-Call-Wrappers (hier: `read()`) zu analysieren, die zugehörige Implementierung im Linux-Kernel zu finden und die wesentlichen Schritte und Abläufe eines System-Calls auf Anwendungsebene darzustellen. Zur Lösung der Aufgabe wurde eine Linux-Umgebung mithilfe eines Docker-Containers eingerichtet und der Quellcode von glibc sowie des Linux-Kernels untersucht.

Vorgehen

1. Einrichtung der Linux-Umgebung

Da die Aufgabe unter macOS bearbeitet wurde, wurde eine Linux-Umgebung mit Docker (`docker run -it ubuntu`) erstellt. Anschließend wurden wichtige Tools installiert:

- **build-essential**: Compiler und Build-Tools.
- **manpages-dev**: Dokumentation für Systemaufrufe.
- **git und wget**: Für Quellcode-Downloads und weitere Pakete.

Damit war die Umgebung für die Quellcode-Analyse bereit.

2. Herunterladen des glibc-Quellcodes

Die GNU C Library (glibc) enthält die Wrapper-Funktionen für Systemaufrufe wie `read()`. Der Quellcode wurde heruntergeladen und entpackt:

```
`wget http://ftp.gnu.org/gnu/libc/glibc-2.37.tar.gz`  
`tar -xvzf glibc-2.37.tar.gz && cd glibc-2.37`
```

3. Identifikation des `read()`-Wrappers in glibc

Nach Durchsuchen des Quellcodes mit:

```
`find . -name "*read*"`
```

wurde die Datei ``sysdeps/unix/sysv/linux/read.c`` gefunden, die die Implementierung des `read()`-Wrappers enthält. Die Funktion wurde in dieser Datei geöffnet und untersucht:

```

root@dfa70f0eee00:/glibc-2.37# cat sysdeps/unix/sysv/linux/read.c
/* Linux read syscall implementation.
   Copyright (C) 2017-2023 Free Software Foundation, Inc.
   This file is part of the GNU C Library.

   The GNU C Library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   The GNU C Library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with the GNU C Library; if not, see
   <https://www.gnu.org/licenses/>. */

#include <unistd.h>
#include <sysdep-cancel.h>

/* Read NBYTES into BUF from FD. Return the number read or -1. */
ssize_t
__libc_read (int fd, void *buf, size_t nbytes)
{
  return SYSCALL_CANCEL (read, fd, buf, nbytes);
}
libc_hidden_def (__libc_read)

libc_hidden_def (__read)
weak_alias (__libc_read, __read)
libc_hidden_def (read)
weak_alias (__libc_read, read)
root@dfa70f0eee00:/glibc-2.37# █

```

- Analyse des Wrappers:

Der Wrapper leitet den Aufruf mit SYSCALL_CANCEL an den Kernel weiter.

SYSCALL_CANCEL führt den eigentlichen Systemaufruf (read) aus.

4. Untersuchung des Linux-Kernels

Der Linux-Kernel wurde von GitHub heruntergeladen, um die Implementierung des System-Calls read zu untersuchen:

```
`git clone https://github.com/torvalds/linux.git && cd linux`
```

Mithilfe von `grep -r "SYSCALL_DEFINE3(read)" .` wurde die Datei `fs/read_write.c` identifiziert, die den read-System-Call implementiert.

5. Analyse der Kernel-Implementierung

Zentrale Codeabschnitte für die Implementierung des read-System-Calls sind:

```

...

SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
  return ksys_read(fd, buf, count);
}
...

```

- SYSCALL_DEFINE3: Definiert den read-Systemaufruf mit drei Parametern (fd, buf, count).

- ksys_read(fd, buf, count): Führt die eigentliche Leseoperation aus.

```

...
ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
{
    CLASS(fd_pos, f)(fd);
    ssize_t ret = -EBADF;
    if (!fd_empty(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;
    }
    return ret;
}
...

```

Die Funktion `ksys_read` übernimmt grundlegende Prüfungen und leitet die Leseoperation an die Funktion `vfs_read` weiter, die die Leseoperation durchführt.

6. Zusammenfassung des Datenflusses

Warum wurden glibc und Kernel untersucht?

Um den gesamten Ablauf eines Systemaufrufs wie `read()` zu verstehen, ist es notwendig, sowohl die Implementierung in der **glibc** als auch im **Linux-Kernel** zu untersuchen. Diese beiden Ebenen arbeiten zusammen, um sicherzustellen, dass Anwendungen effizient und sicher auf Systemressourcen zugreifen können.

1. **glibc – Die Rolle der Standard-C-Bibliothek:** Die glibc stellt Funktionen wie `read()` als **Wrapper** bereit. Diese Wrapper dienen als Schnittstelle zwischen der Anwendung und dem Kernel. Die glibc:
 - **Abstrahiert den Übergang:** Anwendungen müssen keine Systemaufrufe direkt über low-level Mechanismen wie `syscall` ausführen.
 - **Ermöglicht Portabilität:** Die Bibliothek kümmert sich um die Details, wie ein Systemaufruf für unterschiedliche Plattformen implementiert ist.

- **Leitet die Anfrage an den Kernel weiter:** Der Wrapper in der glibc übergibt die Parameter (Dateideskriptor, Puffer, Anzahl der Bytes) an den Kernel und sorgt für den Übergang in den Kernel-Space.
2. **Kernel – Die Verarbeitung der Anfrage:** Der Kernel führt die eigentliche Arbeit aus, wenn der Systemaufruf über den Wrapper in der glibc ausgelöst wurde. Für den read()-Systemaufruf übernimmt der Kernel:
- **Sicherheitsprüfungen:** Überprüft, ob die Datei existiert, ob die Datei lesbar ist, und ob der angegebene Speicherbereich im User-Space gültig ist.
 - **Datenzugriff:** Greift auf die Datei (oder das Gerät) zu, von der gelesen werden soll, und liest die angeforderten Daten.
 - **Rückgabe der Daten:** Kopiert die gelesenen Daten aus dem Kernel-Speicher in den User-Space-Puffer und gibt die Anzahl der gelesenen Bytes oder einen Fehlercode zurück.
3. **Zusammenspiel von glibc und Kernel:** Der Systemaufruf verläuft in mehreren Schritten:
1. Anwendungsebene:
 - Die Anwendung ruft read(fd, buf, count) auf.
 - Der Aufruf wird an die glibc-Funktion __libc_read() weitergeleitet.
 2. Übergang in den Kernel:
 - __libc_read nutzt SYSCALL_CANCEL, um in den Kernel-Modus zu wechseln.
 - Der Kernel nimmt den Aufruf mit SYSCALL_DEFINE3(read) entgegen.
 3. VFS-Schicht:
 - ksys_read prüft die Eingaben und delegiert die Leseoperation an vfs_read.
 - vfs_read führt die eigentliche Leseoperation aus.
 4. Rückgabe:
 - Die gelesenen Daten oder ein Fehlercode werden an die Anwendung zurückgegeben.

Durch die Analyse sowohl der glibc als auch des Linux-Kernels wird deutlich, wie der Übergang vom User-Space (Anwendungsebene) in den Kernel-Space (Systemressourcenebene) funktioniert.

Aufgabe 2: System-Call-Latenz

Einleitung

Das Ziel dieser Aufgabe war es, die minimale Latenz eines Systemaufrufs experimentell zu ermitteln. Die System-Call-Latenz beschreibt die Zeit, die ein Systemaufruf benötigt, um vom User-Space in den Kernel-Space zu wechseln, die gewünschte Operation auszuführen und in den User-Space zurückzukehren.

Vorgehen

1. Nutzung des bestehenden Docker-Containers

Für diese Aufgabe wurde der bereits erstellte **Docker-Container** aus **Aufgabe 1** wiederverwendet. Der Container stellt eine Ubuntu-basierte Linux-Umgebung bereit, die für die Durchführung dieser Aufgabe geeignet ist. Dadurch konnten bestehende Konfigurationen und Installationen weiterhin genutzt werden. Zusätzlich wurden im Container Werkzeuge wie nano und gcc mit `"apt update && apt install -y nano gcc"` installiert:

2. Begründung der Implementierung in C

Das Programm zur Messung der System-Call-Latenz wurde in der Programmiersprache **C** implementiert, da:

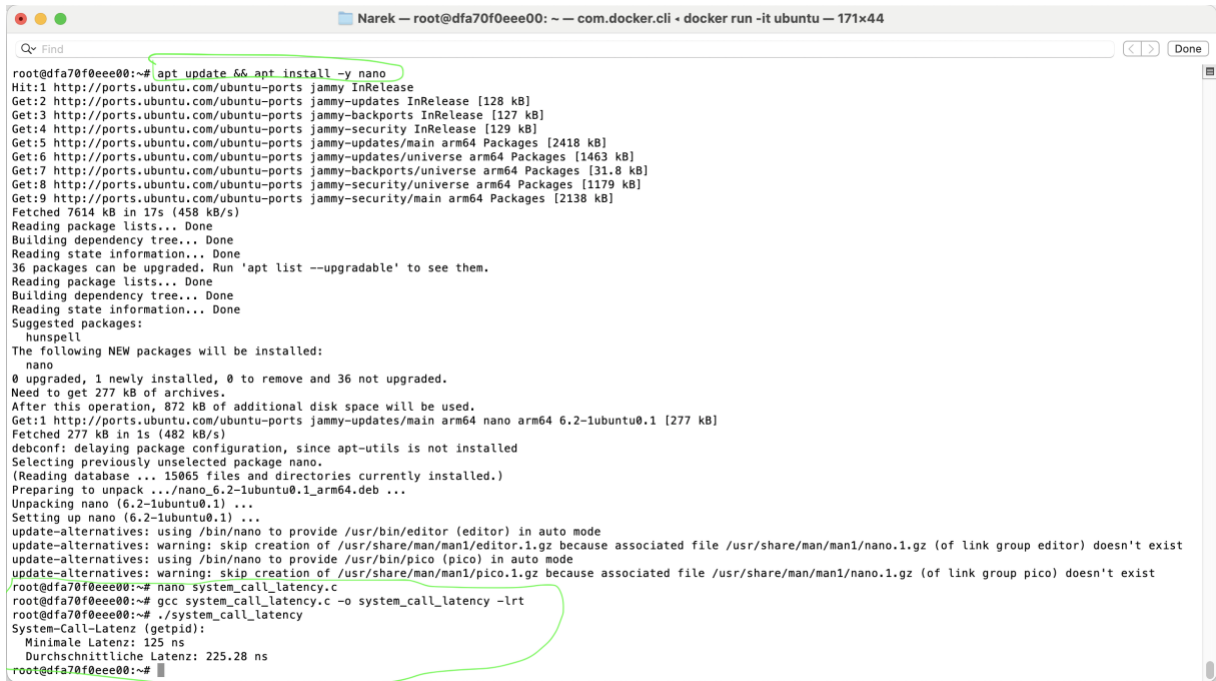
- **C** die Sprache ist, die nahe am Betriebssystem arbeitet und direkten Zugriff auf Systemaufrufe bietet.
- Zeitmessungen sowie der Zugriff auf Funktionen wie `getpid()` und `clock_gettime()` lassen sich in C einfach und effizient umsetzen.
- Der Overhead durch die Programmiersprache minimal gehalten wird, um möglichst präzise Messungen durchzuführen.

3. Warum `getpid()` gewählt wurde

Der Systemaufruf **`getpid()`** wurde gewählt, weil:

- Er ist ein sehr einfacher Systemaufruf, der nur die **Prozess-ID des aktuellen Prozesses** zurückgibt.
- `getpid()` führt keine komplexen Berechnungen oder Interaktionen mit Geräten durch. Dadurch misst man hauptsächlich die Latenz des Übergangs

zwischen **User-Space** und **Kernel-Space**, ohne dass zusätzliche Verzögerungen durch andere Faktoren entstehen.



```
root@dafa70f0eee00:~# apt update && apt install -y nano
Hit:1 http://ports.ubuntu.com/ubuntu-ports jammy InRelease
Get:2 http://ports.ubuntu.com/ubuntu-ports jammy-updates InRelease [128 kB]
Get:3 http://ports.ubuntu.com/ubuntu-ports jammy-backports InRelease [127 kB]
Get:4 http://ports.ubuntu.com/ubuntu-ports jammy-security InRelease [129 kB]
Get:5 http://ports.ubuntu.com/ubuntu-ports jammy-updates/main arm64 Packages [2418 kB]
Get:6 http://ports.ubuntu.com/ubuntu-ports jammy-updates/universe arm64 Packages [1463 kB]
Get:7 http://ports.ubuntu.com/ubuntu-ports jammy-backports/universe arm64 Packages [31.8 kB]
Get:8 http://ports.ubuntu.com/ubuntu-ports jammy-security/universe arm64 Packages [1179 kB]
Get:9 http://ports.ubuntu.com/ubuntu-ports jammy-security/main arm64 Packages [2138 kB]
Fetched 7614 kB in 17s (458 kB/s)
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
36 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
hunspell
The following NEW packages will be installed:
nano
0 upgraded, 1 newly installed, 0 to remove and 36 not upgraded.
Need to get 277 kB of archives.
After this operation, 872 kB of additional disk space will be used.
Get:1 http://ports.ubuntu.com/ubuntu-ports jammy-updates/main arm64 nano arm64 6.2-1ubuntu0.1 [277 kB]
Fetched 277 kB in 1s (482 kB/s)
debconf: delaying package configuration, since apt-utils is not installed
Selecting previously unselected package nano.
(Reading database ... 15065 files and directories currently installed.)
Preparing to unpack .../nano.6.2-1ubuntu0.1_arm64.deb ...
Unpacking nano (6.2-1ubuntu0.1) ...
Setting up nano (6.2-1ubuntu0.1) ...
update-alternatives: using /bin/nano to provide /usr/bin/editor (editor) in auto mode
update-alternatives: warning: skip creation of /usr/share/man/man1/editor.1.gz because associated file /usr/share/man/man1/nano.1.gz (of link group editor) doesn't exist
update-alternatives: using /bin/nano to provide /usr/bin/pico (pico) in auto mode
update-alternatives: warning: skip creation of /usr/share/man/man1/pico.1.gz because associated file /usr/share/man/man1/nano.1.gz (of link group pico) doesn't exist
root@dafa70f0eee00:~# nano system_call_latency.c
root@dafa70f0eee00:~# gcc system_call_latency.c -o system_call_latency -lrt
root@dafa70f0eee00:~# ./system_call_latency
System-Call-Latenz (getpid):
Minimale Latenz: 125 ns
Durchschnittliche Latenz: 225.28 ns
root@dafa70f0eee00:~#
```

Ergebnisse

Das Programm lieferte folgende Ausgabe:

System-Call-Latenz (getpid):

- **Minimale Latenz:** Die schnellste gemessene Zeit für den getpid()-Systemaufruf beträgt **125 Nanosekunden**.
- **Durchschnittliche Latenz:** Im Durchschnitt benötigt der Systemaufruf **225.28 Nanosekunden**.

Diskussion

1. Was misst das Programm?

Das Programm misst die Zeit, die benötigt wird, um:

1. Den Systemaufruf getpid() auszuführen.
2. Vom User-Space in den Kernel-Space zu wechseln.
3. Die Prozess-ID im Kernel zu lesen.

4. Vom Kernel-Space zurück in den User-Space zu wechseln.

Die Differenz zwischen der minimalen und der durchschnittlichen Latenz zeigt den Einfluss von Faktoren wie CPU-Interrupts, Cache-Misses oder Scheduling-Verzögerungen.

2. Warum ist getpid() geeignet?

- getpid() ist ein einfacher Systemaufruf, der keine umfangreiche Verarbeitung erfordert.
- Die gemessene Zeit repräsentiert daher hauptsächlich den Overhead des Übergangs zwischen User- und Kernel-Space.

3. Einflussfaktoren

Die Ergebnisse können von mehreren Faktoren beeinflusst werden:

- **Systemlast:** Hintergrundprozesse können die durchschnittliche Latenz erhöhen.
- **Hardware:** Prozessorleistung und Speicherlatenzen beeinflussen die Ergebnisse.
- **Docker-Umgebung:** Da die Messung in einem Docker-Container durchgeführt wurde, kann ein kleiner Overhead durch die Virtualisierung auftreten.

4. Bedeutung der Ergebnisse

Die niedrige minimale Latenz von 125 Nanosekunden zeigt, wie effizient Linux einfache Systemaufrufe verarbeitet. Die durchschnittliche Latenz von 225.28 Nanosekunden reflektiert reale Bedingungen mit Hintergrundaktivitäten und anderen Prozessen. Dieses Experiment demonstriert, wie schnell Systemaufrufe wie getpid() in einem Linux-System ausgeführt werden können. Die Unterschiede zwischen minimaler und durchschnittlicher Latenz verdeutlichen den Einfluss von Systeminterferenzen. Dieses Wissen ist besonders wertvoll, um die Effizienz von Betriebssystemen zu bewerten oder Optimierungen durchzuführen.

Aufgabe 3: Kontextwechsel

Einleitung

Das Ziel der Aufgabe war es, die durchschnittliche Dauer eines Kontextwechsels auf einem System experimentell zu ermitteln. Ein Kontextwechsel tritt auf, wenn die CPU zwischen zwei Threads oder Prozessen wechselt, wobei der Zustand des vorherigen Prozesses gespeichert und der Zustand des neuen Prozesses geladen wird. Für diese Aufgabe wurde ein experimenteller Ansatz entwickelt, bei dem mithilfe eines in C

implementierten Programms die durchschnittliche Kontextwechselzeit gemessen wurde. Ergänzend dazu wurden die Ergebnisse mit Standardwerkzeugen wie /usr/bin/time validiert und analysiert.

Vorgehensweise

1. **Entwicklung eines C-Programms:** Das Programm wurde entwickelt, um mithilfe von zwei Threads und Mutexes freiwillige Kontextwechsel zu erzeugen. Die durchschnittliche Zeit für einen Kontextwechsel wurde auf Grundlage der benötigten Zeit für eine große Anzahl von Iterationen berechnet.

Kernpunkte des Programms:

- Zwei Threads wurden verwendet: Ein Hauptthread und ein zusätzlicher Thread, die über Mutexes synchronisiert wurden.
- Die Funktion `clock_gettime(CLOCK_MONOTONIC)` wurde genutzt, um präzise Zeitmessungen zu ermöglichen.
- Die durchschnittliche Zeit wurde berechnet, indem die Gesamtzeit durch die Anzahl der Kontextwechsel geteilt wurde.



```
GNU nano 6.2 context_switch.c
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <stdint.h>
#include <unistd.h>

#define ITERATIONS 1000000 // Anzahl der Kontextwechsel

// Globale Variablen für Synchronisation
pthread_mutex_t lock1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t lock2 = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg) {
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&lock2); // Warte auf den Main-Thread
        pthread_mutex_unlock(&lock1); // Signalisiere den Main-Thread
    }
    return NULL;
}

int main() {
    struct timespec start, end;
    pthread_t thread;

    // Lock 2 zuerst sperren, damit der zweite Thread blockiert startet
    pthread_mutex_lock(&lock2);

    // Thread erstellen
    pthread_create(&thread, NULL, thread_func, NULL);

    // Messung starten
    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < ITERATIONS; i++) {
        pthread_mutex_lock(&lock1); // Warte auf den Thread
        pthread_mutex_unlock(&lock2); // Signalisiere den Thread
    }
    clock_gettime(CLOCK_MONOTONIC, &end);

    // Warte auf Thread-Ende
    pthread_join(thread, NULL);

    // Dauer berechnen (in Nanosekunden)
    uint64_t total_time = (end.tv_sec - start.tv_sec) * 1000000000 +
        (end.tv_nsec - start.tv_nsec);

    double avg_time = total_time / (double)ITERATIONS * 2; // 2 Wechsel pro Iteration

    // Ergebnisse ausgeben
    printf("Durchschnittliche Kontextwechselzeit: %.2f ns\n", avg_time);

    return 0;
}
```

2. **Ausführung des Programms:** Das Programm wurde innerhalb der selben Docker-Ubuntu-Container ausgeführt, um die gemessenen Zeiten unabhängig von macOS zu bestimmen. Mit dem Befehl gcc wurde das Programm kompiliert und mit ./context_switch ausgeführt.
3. **Nutzung von /usr/bin/time:** Zusätzlich wurde /usr/bin/time verwendet, um die Systemzeit (sys) und die Anzahl der freiwilligen und unfreiwilligen Kontextwechsel zu messen. Dieses Tool bietet eine zusätzliche Validierung und eine breitere Perspektive.

Ergebnisse

1. Aus dem Programm:

- Durchschnittliche Kontextwechselzeit: **14.840,14 ns (14,84 µs)**.
- Dies repräsentiert die Zeit für einen vollständigen Wechsel zwischen zwei Threads, einschließlich des Speicherns und Wiederherstellens von Register- und Speicherinhalten.

2. Aus /usr/bin/time:

- **Systemzeit (sys):** 31,34 Sekunden.
- **Freiwillige Kontextwechsel:** 1.998.647.
- **Unfreiwillige Kontextwechsel:** 103.
- Die Anzahl der freiwilligen Kontextwechsel passt gut zur Anzahl der Iterationen im Programm (1.000.000 Iterationen × 2 Kontextwechsel pro Iteration = 2.000.000 Wechsel).

```
root@dfa70f0eee00:~# /usr/bin/time -v ./context_switch
Durchschnittliche Kontextwechselzeit: 14840.14 ns
Command being timed: "./context_switch"
User time (seconds): 4.57
System time (seconds): 31.34
Percent of CPU this job got: 121%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:29.68
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 1428
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 70
Voluntary context switches: 1998647
Involuntary context switches: 103
Swaps: 0
File system inputs: 0
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
root@dfa70f0eee00:~#
```

3. Diskrepanzen:

- Die Zeitmessung des Programms ist präziser für die spezifische Operation (Kontextwechsel). Die Systemzeit aus `/usr/bin/time` umfasst zusätzliche Kerneloperationen, wie Scheduler-Aufgaben, die die Messung beeinflussen.

Diskussion

1. Direkte Kosten eines Kontextwechsels:

- Die gemessene Zeit von etwa 14,84 μ s ist typisch für Kontextwechsel und verdeutlicht den Overhead solcher Operationen. Diese Zeit beinhaltet:
 - **Speichern und Wiederherstellen von Registern:** Der Zustand des vorherigen Threads wird gesichert und der neue Zustand geladen.
 - **Speicherverwaltungsaufgaben:** Die CPU muss den Zugriffskontext für den Speicher ändern.

2. Indirekte Kosten eines Kontextwechsels:

- **Cache-Misses:** Kontextwechsel leeren oft den CPU-Cache, was dazu führt, dass der nächste Thread auf Daten aus dem Hauptspeicher statt aus dem Cache zugreifen muss.
- **Speicherzugriff:** Der Wechsel zwischen Threads erfordert zusätzliche Arbeit für den Memory Management Unit (MMU), um den Speicherzugriff zu validieren.
- **Scheduler-Overhead:** Der Kernel benötigt Zeit, um den nächsten auszuführenden Thread zu bestimmen, insbesondere bei vielen Prozessen.

Persönliches Fazit

Die Übungen haben mir geholfen, ein besseres Verständnis für die Abläufe in einem Betriebssystem zu bekommen. Besonders interessant war es zu sehen, wie System Calls und Kontextwechsel in der Praxis funktionieren und welche Zeitkosten sie mit sich bringen. Durch die praktischen Experimente habe ich viel über die Effizienz und Optimierung moderner Systeme gelernt, was mir sicher auch in zukünftigen Projekten nützlich sein wird.