# Portfolio Prüfung

17. November 2024

## Inhaltsverzeichnis

# 1 System-Calls

## 1.1 Definition eines System-Calls

Ein **System-Call** (Systemaufruf) ist eine Schnittstelle zwischen Programmen im Benutzermodus (User Mode) und dem Betriebssystemkern (Kernel Mode). Er ermöglicht es Anwendungen, Betriebssystemdienste wie Dateiverwaltung, Speicherverwaltung oder Prozesssteuerung zu nutzen. Dabei wird der Prozessor vom Benutzermodus in den Kernelmodus geschaltet, da diese Operationen nur im Kernelmodus sicher ausgeführt werden können.
Beispiele für System-Calls sind `read()`, `write()`, `open()` und `close()`.

## 1.2 Analyse des `read_write()`-System-Calls

### 1.2.1 Vorgehensweise zur Analyse

Für die Analyse wurde der Sourcecode von `read()` im Linux-Kernel untersucht. Dabei wurden folgende Schritte durchgeführt:

1. **Kernel-Sourcecode herunterladen:**

```
simon@Swifty:~/Documents/Vorlesungen/Winter_24/Betriebssysteme/Uebungen/Linux/linux-6.11.8$ ls
arch   COPYING  Documentation  include  ipc      kernel    MAINTAINERS  net      samples   sound  virt
block  CREDITS  drivers        init     Kbuild   lib       Makefile     README   scripts   tools
certs  crypto   fs             io_uring Kconfig  LICENSES  mm           rust     security  usr
```

Abbildung 1: Linux Version 6.11.8

2. **Wrapper-Funktion in der libc finden:** Der System-Call `read()` wird von der C-Standardbibliothek (glibc/libc) über eine Wrapper-Klasse bereitgestellt:

```
simon@Swifty:~/Documents/Vorlesungen/Winter_24/Betriebssysteme/Uebungen/Linux/linux-6.11.8$ find . -name 'read_write.c'
./fs/read_write.c
./fs/ecryptfs/read_write.c
```

3. **Kernel-Implementierung identifizieren:** Die eigentliche Implementierung des System-Calls `read()` im Kernel ist durch die Funktion `sys_read` definiert.

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

Diese Definition befindet sich in der Datei `fs/read_write.c`.

### 1.2.2 Ablauf des System-Calls `read()`

Der Aufruf eines System-Calls durchläuft die folgenden Schritte:

1. **Anwendungsebene:** Die Anwendung ruft die Funktion `read()` auf.
2. **glibc-Wrapper:** Die libc-Funktion `read()` verwendet z. B. den Assembler-Befehl `syscall`, um die System-Call-Nummer und die Parameter an den Kernel zu übergeben.
3. **Kernel-Dispatcher:** Der Kernel identifiziert den System-Call anhand der System-Call-Nummer in der `sys_call_table`:

   ```
   arch/x86/entry/syscalls/syscall_32.tbl
   ```

   Die Nummer verweist auf die Funktion `sys_read`.
4. **Kernel-Modus:** Der Kernel prüft die Parameter (z. B. Dateideskriptor und Speicherzugriff) und führt die gewünschte Aktion durch.
5. **Rückgabe:** Die Ergebnisse (z. B. gelesene Bytes oder Fehlercodes) werden zurück in den Benutzermodus übertragen.

# 2 System-Call-Latenz

## 2.1 Messung der Latenz von `read()`, `write()` und `getpid()`

### 2.1.1 Ziel des Versuchs

Das Ziel des Experiments war es, die minimale Latenz der System-Calls `read()`, `write()` und `getpid()` zu bestimmen. Diese drei System-Calls repräsentieren unterschiedliche Typen von Operationen: I/O-Operationen (`read()` und `write()`) sowie eine einfache Abfrage des aktuellen Prozess-IDs (`getpid()`). Ihre Latenzen geben Aufschluss über die Effizienz der Kernel-Schnittstelle in unterschiedlichen Kontexten.

### 2.1.2 Versuchsaufbau

Der Versuch wurde mit einem Programm durchgeführt, das die Zeitmessung für die System-Calls `read()`, `write()` und `getpid()` wie folgt organisierte:

1. **Zielobjekt:** Es wurde das virtuelle Gerät `/dev/null` verwendet, um tatsächliche physische I/O-Operationen zu vermeiden und die reine System-Call-Latenz für `read()` und `write()` zu messen. Für `getpid()` wurden keine zusätzlichen Objekte benötigt, da dieser System-Call lediglich die Prozess-ID abruft.
2. **Zeitmessung:** Die Zeit vor und nach der Ausführung von `read()`, `write()` und `getpid()` wurde mithilfe von Zeitfunktionen gemessen.
3. **Wiederholung:** Die Messungen wurden eine Million Mal wiederholt, um eine statistisch belastbare Aussage zu ermöglichen.
4. **Auswertung:** Durchschnitts- und Minimalwerte der Latenzen wurden berechnet, um die System-Call-Effizienz zu bewerten.

### 2.1.3 Ergebnisse und Diskussion

Die Ergebnisse des Experiments zeigen die Latenzen der System-Calls `read()`, `write()` und `getpid()`:

- **Minimale Latenz:** Die schnellste gemessene Ausführungszeit für `read()` betrug **676 ns**, für `write()` lag die minimale Latenz bei **652 ns**, und `getpid()` erreichte eine minimale Latenz von **567 ns**.
- **Durchschnittliche Latenz:** Der Durchschnittswert der Latenz lag bei **691 ns** für `read()`, **667 ns** für `write()` und **576 ns** für `getpid()`.

Die minimale Latenz zeigt, wie schnell ein System-Call im besten Fall durch den Kernel abgearbeitet wird, während die durchschnittliche Latenz typische Betriebssystem- und Hardware-Effekte, wie Kontextwechsel und Caching, reflektiert. Die Ergebnisse zeigen:

- Die Latenz von `read()` und `write()` ist höher als die von `getpid()`, da bei `read()` und `write()` zusätzliche Prüfungen und I/O-Operationen innerhalb des Kernels notwendig sind, während `getpid()` lediglich die Prozess-ID abruft.
- Schwankungen in der durchschnittlichen Latenz sind durch Systemlast und Hardwaregegebenheiten bedingt.
- Die Verwendung von `/dev/null` als Zielobjekt minimiert externe Einflüsse und erlaubt die Fokussierung auf die reine System-Call-Latenz für `read()` und `write()`.

Die Ergebnisse verdeutlichen, dass einfache System-Calls wie `getpid()` deutlich schneller sind als komplexere System-Calls, die I/O-Operationen erfordern. Dies unterstreicht die Bedeutung von Optimierungen innerhalb des Kernels, um die Gesamtleistung von Anwendungen zu verbessern.

# 3 Kontextwechsel

Ein Kontextwechsel tritt auf, wenn der Prozessor zwischen verschiedenen Threads oder Prozessen wechselt. Dabei speichert das Betriebssystem den Zustand des aktuellen Threads (z. B. Register, Stack-Pointer) und lädt den Zustand eines anderen Threads, um diesen weiter auszuführen. Kontextwechsel sind notwendig, um Multitasking zu ermöglichen, verursachen jedoch einen gewissen Overhead, da sie Zeit und Systemressourcen beanspruchen.
Das Programm misst indirekt die Anzahl der Kontextwechsel, indem es Threads erstellt, die in einer Schleife `sched_yield()` aufrufen. Der Aufruf von `sched_yield()` teilt dem Scheduler des Betriebssystems mit, dass der aktuelle Thread freiwillig auf die Ausführung verzichtet. Dies zwingt das Betriebssystem, einen Kontextwechsel zu einem anderen Thread durchzuführen.

## Funktionsweise der Kontextwechselzählung

Die Anzahl der Kontextwechsel wird wie folgt gezählt:

1. Jeder Thread führt in einer Schleife `NUM_ITERATIONS` Iterationen aus.
2. In jeder Iteration ruft der Thread `sched_yield()` auf, wodurch ein Kontextwechsel angestoßen wird.
3. Ein Mutex (`context_switch_mutex`) sorgt dafür, dass die globale Variable `context_switches` threadsicher inkrementiert wird. Dies verhindert Race Conditions, bei denen mehrere Threads gleichzeitig auf die Zählvariable zugreifen könnten, womit wir auf knapp 400 Nanosekunden für einen Kontextwechel kommen.

Da die Anzahl der Aufrufe von `sched_yield()` proportional zur Anzahl der Kontextwechsel ist, kann die Anzahl der Kontextwechsel direkt aus der Variable `context_switches` abgeleitet werden.

## Ergebnisse des Programms

Nach der Ausführung des Programms wurden die folgenden Ergebnisse ermittelt:

- **Kontextwechsel-Dauer:** Die Gesamtzeit für die Ausführung des Programms beträgt 823.900.919 Nanosekunden. Dies umfasst die Zeit, die für das Starten, Ausführen und Beenden der Threads benötigt wurde.
- **Geschätzte Anzahl der Kontextwechsel:** Während der Programmausführung wurden geschätzt 2.000.000 Kontextwechsel durchgeführt. Dies entspricht der Summe der Aufrufe von `sched_yield()` über alle Threads hinweg.

Diese Daten verdeutlichen, dass Kontextwechsel, obwohl notwendig, einen signifikanten Einfluss auf die Gesamtlaufzeit haben können. Besonders bei Anwendungen mit vielen Threads oder intensiver Nutzung von Synchronisationsmechanismen können sie den Gesamtdurchsatz verringern.

## Profiling von Anwendungen zur Ermittlung von Kontextwechseln

Zur umfassenderen Analyse von Kontextwechseln können Standardwerkzeuge für das Profiling von Anwendungen genutzt werden. Diese Werkzeuge erlauben die Beobachtung und Messung von Systemereignissen, einschließlich Kontextwechseln. Zu den häufig genutzten Tools gehören:

- **top/htop (Linux):** Diese Tools zeigen in Echtzeit Statistiken zur Systemauslastung, einschließlich Prozesswechsel.

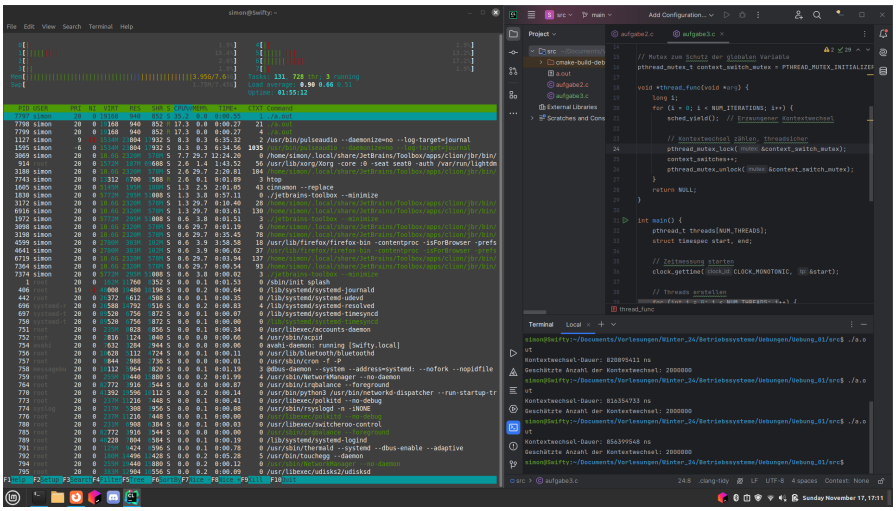

Abbildung 2: Mit `htop` gemessene Kontextwechsel

- **perf (Linux):** Ein leistungsstarkes Werkzeug, das Kontextwechsel und andere Systemereignisse analysieren kann. Es bietet detaillierte Berichte zu Thread-Aktivitäten und Scheduler-Ereignissen. Htop zeigt insgesamt 26 Kontextwechsel während dem Programmlauf an.


Abbildung 3: Mit `perf` gemessene Kontextwechsel

Bei der Untersuchung der Kontextwechsel mittels eines Testprogramms, das `sched_yield()` verwendet, wurden knapp 2.000.000 Kontextwechsel gemessen. Dieses Ergebnis beruht darauf, dass das Programm jeden Aufruf von `sched_yield()` als Kontextwechsel zählt,

unabhängig davon, ob der Scheduler tatsächlich einen Wechsel zwischen Threads durchgeführt hat.
Im Gegensatz dazu zeigten die Messungen mit Tools wie `perf` und `htop` eine deutlich geringere Anzahl von Kontextwechseln (etwa 20). Diese Diskrepanz lässt sich dadurch erklären, dass diese Tools nur tatsächliche Kontextwechsel zählen, bei denen der Scheduler den laufenden Thread von der CPU entfernt und einen anderen Thread ausführt. Falls jedoch keine anderen Threads verfügbar oder bereit zur Ausführung sind, verbleibt der aktuelle Thread auf der CPU, und kein echter Kontextwechsel findet statt.
Die Ergebnisse verdeutlichen, dass `sched_yield()` nicht in jedem Fall zu einem tatsächlichen Kontextwechsel führt. Stattdessen dient der Aufruf als freiwillige Aufgabe der CPU-Zeit, wobei der Scheduler entscheidet, ob ein Wechsel notwendig ist. Dies zeigt die Bedeutung einer präzisen Definition von "Kontextwechsel"bei der Analyse der Systemleistung.