Ergebnisbericht Übungsblatt 1 Betriebssysteme

Simon Haebenbrock, s4sihaeb@uni-trier.de, dc: SimonH (simon_7672)

Aufgabe 1 System-Call

Für die Folgende Analyse wurde der System-Call **read()** ausgewählt. Er ermöglicht es, Daten von einer Datei oder einem anderen Eingabestream in einen Speicherpuffer einzulesen.

Lokalisierung des Wrappers in der GNU-C-Bibliothek (glibc)
Die glibc enthält den System-Call-Wrapper für read() in der Datei io/read.c. Der Wrapper stellt die Schnittstelle zwischen der Anwendung und dem Kernel dar und nutzt die Funktion syscall, um den Übergang in den Kernel zu initiieren.

Quellcode des Wrappers:

```
ssize_t
__libc_read (int fd, void *buf, size_t nbytes)
{
  if (nbytes == 0)
    return 0;
  if (fd < 0)
    {
      __set_errno (EBADF);
      return -1;
    }
  if (buf == NULL)
    {
      __set_errno (EINVAL);
      return -1;
    }

  __set_errno (ENOSYS);
  return -1;
}
```

Parameterprüfung: Der Wrapper prüft, ob die Eingabeparameter gültig sind (fd, buf, nbytes).

Fehlerbehandlung: Fehler setzen einen entsprechenden errno-Wert (z. B. EBADF, EINVAL).

Wechsel in den Kernel-Modus: In der endgültigen Version des Codes wird der Übergang vom User-Space in den Kernel über eine spezielle syscall-

Anweisung durchgeführt. Der gezeigte Code ist ein Platzhalter, der darauf hinweist, dass die Architektur-spezifische Umsetzung dieses System-Calls an anderer Stelle in den glibc-Quellen zu finden ist. Das bedeutet, die genaue Ausführung der syscall kann je nach verwendeter Hardware-Architektur unterschiedlich sein und wird in anderen Teilen der glibc-Bibliothek definiert.

Implementierung im Linux-Kernel
Der System-Call read() wird im Kernel durch das Makro SYSCALL_DEFINE definiert.

Der read()-System-Call ist in der Datei fs/read_write.c definiert:

```
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;
}

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

Parameter: Die Definition zeigt, dass der System-Call drei Parameter erwartet:
fd: Dateideskriptor.
buf: Zeiger auf den Speicherpuffer im User-Space.
count: Anzahl der zu lesenden Bytes.

Funktionsaufruf: Diese Funktion ruft ksys_read() auf, um die eigentliche Verarbeitung durchzuführen.

Kernelfunktion ksys_read():
Parameterprüfung: fdget_pos(fd) überprüft den Dateideskriptor und lädt die zugehörige Datei.
Abstraktionsebene: vfs_read() wird verwendet, um die Datei zu lesen, unabhängig vom spezifischen Dateisystem.

Ablauf des System-Calls
Der Ablauf eines read()-System-Calls kann wie folgt beschrieben werden:

1. **Anwendungsebene:** Eine Anwendung ruft read(fd, buf, count) auf, um Daten zu lesen.
2. **System-Call-Wrapper (glibc):** Die glibc kapselt den Aufruf mit dem Wrapper __libc_read. Dieser initialisiert den Übergang in den Kernel mithilfe von syscall.
3. **Kernel-Übergang:** Der Kernel wird über einen Software-Interrupt (z. B. int 0x80 bei älteren x86-Architekturen oder syscall bei neueren) aktiviert, und die CPU wechselt in den Kernel-Modus.
4. **Kernel-Verarbeitung:**
    - SYSCALL_DEFINE3(read, ...) wird ausgeführt, die Parameter werden überprüft.
    - ksys_read() ruft vfs_read() auf, das den Lesevorgang auf Dateisystem-Ebene durchführt.
5. **Rückgabe an die Anwendung:** Der Kernel liefert das Ergebnis (z. B. Anzahl der gelesenen Bytes oder einen Fehlercode) zurück an die Anwendung.

Quellen A1

- Kernel-Quellen: Kernel.org
- glibc-Quellen: Sourceware Glibc Repository
- OS01_Architecture Folien
- GitHub Copilot für Fehleranalyse, Refactoring und Generierung von Boilerplate-Code

Aufgabe 2 System-Call-Latenz

Messung der System-Call-Latenz für den read()-System-Call

Ziel der Messung war es, die Latenz des read()-System-Calls bei der Ausführung auf einer kleinen Textdatei zu bestimmen. Die Latenz gibt an, wie viel Zeit das Betriebssystem benötigt, um eine Datei zu lesen und die darin enthaltenen Daten in den Arbeitsspeicher zu übertragen.
Für die Messung wurden **drei Durchläufe** der Datei durchgeführt, um die Latenz des Lesevorgangs unter möglichst stabilen Bedingungen zu ermitteln.

Durchführung der Messung
Die Messung wurde mit einer kleinen Textdatei durchgeführt, die 23 Bytes enthält. Die Latenz für jeden Lesevorgang wurde in Nanosekunden gemessen. Zu diesem Zweck wurde ein kleines Java-Programm geschrieben, das die Datei in drei Durchläufen liest. Die folgenden Ergebnisse wurden für diese drei Durchläufe der Datei ermittelt.

Ergebnisse
Die Messung für die Datei testfile.txt ergab die folgenden Werte:

- **Durchlauf 1:**
  Latenz: **6542 ns**
  Gelesene Bytes: 23
- **Durchlauf 2:**
  Latenz: **4708 ns**
  Gelesene Bytes: 23
- **Durchlauf 3:**
  Latenz: **2000 ns**
  Gelesene Bytes: 23

Erklärung: Die sinkende Latenz ist durch Caching-Effekte bedingt. Beim ersten Lesevorgang mussten die Daten von der Festplatte geladen werden, während sie bei den folgenden Durchläufen aus dem Cache des Betriebssystems gelesen wurden, was die Latenz deutlich reduzierte.

Diskussion
Die Ergebnisse zeigen, dass Caching die Latenz bei wiederholten Lesevorgängen erheblich reduziert. Einfache Leseoperationen auf kleinen Dateien profitieren stark vom Cache des Systems.

Fazit
Die Latenz des read()-System-Calls sinkt mit wiederholtem Zugriff auf eine kleine Datei, da die Daten zunehmend aus dem Cache und nicht mehr von der Festplatte gelesen werden.

Aufgabe 3 Kontextwechsel

Die durchschnittliche Kontextwechselzeit auf meinem **MacBook Pro** (Apple M1 Pro) wurde mit einem **Java-Programm** gemessen, das 100 Iterationen zwischen zwei Threads durchführt. Die Threads simulieren **CPU-intensive Aufgaben**, um den Prozessor zu beschäftigen, und **erzwungene Kontextwechsel** durch **Thread.sleep()**. Die **durchschnittliche Zeit** für einen **Kontextwechsel** beträgt: **12472046 ns**.

Systeminformationen

- **Modell**: MacBook Pro (Apple M1 Pro)
- **Prozessor**: 8 Kerne (6 Performance, 2 Effizienz)
- **Arbeitsspeicher**: 16 GB
- **Betriebssystem**: macOS 15.1.1

Indirekte Kosten eines Kontextwechsels

Cache Misses:
Kontextwechsel leeren oft den Cache, was zu langsamerem Zugriff auf Daten im Hauptspeicher führt.
Auswirkung: Auf dem Apple M1 Pro führt der Kontextwechsel zu Cache Misses und verlangsamt die Anwendung.

Synchronisationskosten:
Häufige Kontextwechsel erhöhen den Overhead durch notwendige Synchronisation (z. B. Sperren von Ressourcen).
Auswirkung: Mehr Synchronisationsaufwand verringert die Performance.

Fazit
Die gemessene **Kontextwechselzeit** von **12472046 ns** zeigt, dass der M1 Pro eine effiziente Handhabung von Kontextwechseln bietet, aber indirekte Kosten wie **Cache Misses** können die Performance beeinflussen.

Quellen A3

- Hazelcast. (2024, November 12). What's a cache miss? Policies that reduce cache misses | Hazelcast. https://hazelcast.com/glossary/cache-miss/
- Sturm, P. & Universität Trier. (2001). *Systemsoftware*. https://www.uni-trier.de/fileadmin/fb4/prof/INF/SVS/Download/Wintersemester_2001_2002/Systemsoftware_I/Vorlesung/05_Memory-Based_IPC.pdf
- GitHub Copilot für Fehleranalyse, Refactoring und Generierung von Boilerplate-Code