Betriebssysteme
Übung 1

Tillmann Faust

1.12.2024

Aufgabe 1

Für diese Aufgabe wurde der read() System-Call und der zugehörige Wrapper für Linux in c gewählt. In unistd.h wird ein allgemeines Syscall-Makro vordefiniert, welches anschließend systemspezifisch, für Linux unter 'glibc/sysdeps/unix/sysv/linux/read.c'. Der Ablauf eines read() System-Calls in glibc ist:

1. Ein Anwendung ruft die Funktion 'read(fd,buf,nbytes)' auf. (Dazu muss 'unistd.h' eingebunden sein)

```
extern ssize_t __read (int __fd, void *__buf, size_t __nbytes);
libc_hidden_proto (__read)
```

2. Über den dynamischen Linker wird der System-Call-Wrapper der glibc aufgerufen. Dieser ist in 'glibc/sysdeps/unix/sysv/linux/read.c' definiert.

```
#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)
```

   Hier wird das Makro SYSCALL_CANCEL mit den nötigen Informationen zum spezifischen System-Call aufgerufen

3. Das Makro SYSCALL_CANCEL ruft das Makro INLINE_SYSCALL_CALL und soll zusätzlich mögliche Probleme durch Threading behandeln.

4. INLINE_SYSCALL_CALL ruft anschließend INLINE_SYSCALL auf, welches den System-Call an den Kernel übergibt.
5. Das System geht nun in den Kernel-Modus. Die ID des System Calls wird aus einer systemspezifischen Tabelle ausgelesen (0 für read) und die so gefundene Kernel Operation sys_read wird ausgeführt.
6. sys_read befindet sich im Linux Kernel unter 'fs/read_write.c'.

```
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;
}

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

   (a) Das Makro 'SYSCALL_DEFINE3(name,...)' steht für 'sys_##name(...)' und wird in diesem Kontext zu 'sys_read' aufgelöst.
   (b) ksys_read definiert den System-Call im Kernel und führt, sofern die Datei existiert, den read über vfs_read aus.
   (c) vfs steht virtual file system. vfs_read dient als Abstraktion und nutzt einen Zeiger auf den read Befehl der systemspezifischen Dateistruktur um den read-Befehl letztendlich auszuführen.

7. Nach einem abgeschlossenen read, wird der Rückgabewert geschrieben und das System wechselt wieder in den User-Mode. Dort wird in glibc der Wert von INLINE_SYSCALL, INLINE_SYSCALL_CALL und SYSCALL_CANCEL letztlich an die Wrapper-Funktion ___libc_read übergeben und von dort an die Anwendung.

Aufgabe 2

Der in der Bearbeitung verfolgte Ansatz war es ein c-Programm zu schreiben, welches wiederholt getpid() Aufrufe ausführt und die benötigte Zeit pro Aufruf ermittelt. Dazu wurde das Programm auf einem Raspberry Pi 2 Model B

mit 1GB RAM und einem Broadcom BCM2835 Quad Core Processor@900MHz ausgeführt.

Programm

```c
struct timespec start, end;
long long single_elapsed_ns, elapsed_ns = 0;
long long elapsed_ns_min = 1LL << 60;

// Ermittle geringste benötigte Zeit für einen getpid() Aufruf
for (long i = 0; i < ITERATIONS; i++) {
    clock_gettime(CLOCK_MONOTONIC,&start);
    getpid();
    clock_gettime(CLOCK_MONOTONIC,&end);
    single_elapsed_ns = (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
    if (single_elapsed_ns < elapsed_ns_min) elapsed_ns_min = single_elapsed_ns;
    elapsed_ns += single_elapsed_ns;
}

// Zeitdifferenz in Nanosekunden berechnen
double avg_ns = elapsed_ns / (double) ITERATIONS;
```

Abbildung 1: Hauptteil des Testprogramms

Das Testprogramm führt ITERATIONS(= 10.000.000)-mal einen getpid() Befehl aus und ermittelt die benötigte Zeit. Sollte die benötigte Zeit geringer als die aktuell geringste Zeit sein, so wird die minimale Zeit angepasst. Anschließend wird, zur Ermittlung des Durchschnitts die Zeit zusätzlich aufsummiert. Zuletzt werden die ermittelten Ergebnisse ausgegeben. Das Programm befindet sich in der Datei ”info_msc_os24_ueb1_a2_getpid_latency_test.c”.

Ergebnis

Das Ergebnis dieser experimentellen Ermittlung war eine minimale Latenz von 468 Nanosekunden. Diese Latenz hat sich über den Verlauf mehrerer Ausführungen des Programms nicht verändert, was eine untere Schranke vermuten lässt. Die ermittelten Zeitkosten eines einzelnen Aufrufs haben mit bis zu +100ns variiert. Nicht eingerechnet wurde die zusätzliche Latenz der Zeitabfragen, welche bei Tests ca. 3ns betrug.

```
Gesamtlaufzeit für 10000000 Iterationen: 5161362394 ns, Durchschnitt: 516.136239 ns
Minimale Latenz: 468 ns
Latenz letzter Einzelaufruf: 521 ns
```

Abbildung 2: Ergebnisse eines Programmdurchlaufs

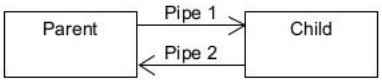
Weiterhin wurde in einem zweiten Ansatz versucht, die durchschnittliche Latenz durch mehreren direkt aufeinander folgenden Systemaufrufen zu testen. Hierbei sind die durchschnittlichen Kosten bei Steigender Zahl von Wiederholungen von ca. 700ns bei 1e6 Iterationen auf ca. 270ns bei 1e10 Interationen konvergiert, weswegen die Vermutung bestand, dass die for-Schleife einen Zeitlich signifikanten Overhead generiert. Korrigiert wurde dieser durch das vorherige messen der Dauer von ITERATIONS vielen Durchläufen einer leeren for-Schleife. Das Ergebnis war eine durchschnittliche Latenz von ca. 264ns. Die Frage, warum die durchschnittliche Latenz bei der Messung von n Aufrufen direkt hintereinander so weit unter dem minimum bei n Einzelmessungen liegt bleibt offen.

Aufgabe 3

Der Kontextwechsel beschreibt einen Vorgang im Betriebssystem, in welchem die CPU von einer Ausführung eines Prozesses oder Threads zur Ausführung eines anderen wechselt. Bei einem Kontextwechsel speichert das Betriebssystem den Zustand des aktuell ausgeführten Prozesses, also Register, Programmzähler, Speicherinformationen und lädt anschließend den Zustand eines anderen Prozesses um diesen auszuführen.

Experimenteller Ansatz

Um die Dauer eines Kontextwechsels zu ermitteln kann man versuchen, in einem (möglichst) kontrollierten Umfang, das Betriebssystem zum Wechsel zwischen zwei Prozessen oder Threads zu bewegen. Die Threads sollen dabei so voneinander abhängig sein, dass immer nur einer zeitgleich laufen kann. Dies soll sicherstellen, dass die Menge der Wechsel zwischen beiden Prozessen klar bekannt ist und so eine Abschätzung überhaupt möglich ist. Dazu können beispielsweise in C ein Child- und ein Parent-Prozess mit Pipelines verknüpft werden und anschließend Nachrichten untereinander austauschen.



Während eines Durchlaufs schickt der Parent-Prozess ITERATIONEN oft eine Nachricht an den Child-Prozess und wartet anschließend auf eine Antwort. Der Child-Prozess wartet ebenfalls auf Nachrichten des Parent-Prozesses und antwortet anschließend auf diese. Der Zeitverbrauch in ns der gesamten ITERATIONEN-vielen Interaktionen wird gemessen und der Durchschnitt durch die Formel

$$\text{average in ns} = \frac{\text{time in ns}}{2 \cdot \text{ITERATIONS}}$$

errechnet. Die so ermittelte Zeit beträgt ca. 25.5 us.

```
Gesamtlaufzeit: 50.986 s
Durchschnittliche Zeit pro Kontextwechsel: 25.49317 us
```

Die Zeit ist ungenauer, da zum Einen die anderen Instruktionen, also das Senden der Nachricht, insbesondere die for-Schleife und die Zeitmessung selbst, wenn auch nur wenig, Zeit benötigen und zum Anderen während der gesamten Laufzeit des Programms auch einige andere Kontextwechsel zu anderen Threads stattfinden können. Das Programm befindet sich in der Datei "info_msc_os24_ueb1_a3_context_switch_latency_test.c"

Analysetool: Perf

Zur Analyse der Dauer von Kontextwechseln eignet sich das umfangreiche Linux Analysetool perf. Dieses ist in der Lage eine Umfangreiche menge an Daten über

den Verbrauch verschiedenster Systemressourcen aufzuzeichnen und auszuwerden. Mittels 'sudo perf sched record ./context_switch_latency_test 5' wurde die Arbeit des Schedulers in den ersten 5 Sekunden der Ausführung des Experimentellen Testprogramms aufgezeichnet und anschließend mit 'perf sched latency' verarbeitet und betrachtet. Das Ergebnis war eine Durchschnittszeit von ungefähr 0.025ms, was sehr nah an den experimentell ermittelten Zeit liegt.

```
context_switch_:(2)   |  69236.460 ms |  1767406 | avg:   0.025 ms |
```

Profiling tools wie perf berücksichtigen zusätzliche Systembelastungen, weswegen die ermittelten Zeiten genauer sein können, während der experimentelle Ansatz effektiv beinahe die Gesamtlaufzeit des Programms nimmt und als Gesamt-Kontextwechselzeit interpretiert.

Indirekte Kosten

Neben den Kosten für die, für den Kontextwechsel notwendigen Operationen können weitere, schwer messbare Kosten durch beispielsweise Cache-Invalidierung oder Kernel-Overheads entstehen.