Operating Systems and Networs SoSe 25 Solutions

Igor Dimitrov

2024-12-18

Table of contents

Preface

1 Blatt 01

1.1 Aufgabe 1

Learning how to Learn:

- **Zwei Denkmodi aus „Learning How to Learn“**
  - **Fokussierter Modus**: Zielgerichtetes, konzentriertes Denken. Gut für bekannte Aufgaben und Übung.
  - **Diffuser Modus**: Entspanntes, offenes Denken. Hilft bei neuen Ideen und kreativen Verknüpfungen.
- **Aufgaben und passende Denkmodi**

  a) Fokussierter Modus
  Warum: Erfordert Konzentration und gezieltes Einprägen.
  b) Zuerst diffuser, dann fokussierter Modus
  Warum: Erst Überblick und Verständnis aufbauen, dann vertiefen.

  c) Fokussierter Modus
  Warum: Klare, schrittweise Übung – ideal für fokussiertes Denken.
  d) Beide Modi
  Warum: Fokussiert für Details & Übungen, diffus für Überblick & Vernetzung.

John Cleese:

- **Zwei Denkmodi**:

  1. **Offener Modus**: Locker, spielerisch, kreativ.
  Beispiel: Ideen für eine Geschichte sammeln.
  Warum: Offenheit fördert neue Einfälle.

  2. **Geschlossener Modus**: Zielgerichtet, angespannt, entscheidungsfreudig.
  Beispiel: Bericht überarbeiten und fertigstellen.
  Warum: Präzises Arbeiten und klare Entscheidungen nötig.

- **Vergleich mit „Learning How to Learn“**

  - **Offen** ⇔ **Diffus**: Für Kreativität und Überblick.

  - **Geschlossen** ⇔ **Fokussiert**: Für Detailarbeit und Umsetzung.

- **Alexander Fleming**:
  - **Modus**: Offen
  - **Warum**: Fleming entdeckte Penicillin zufällig, weil er offen und entspannt war – neugierig statt zielgerichtet. Im geschlossenen Modus hätte er die verschimmelte Petrischale wohl einfach weggeschmissen – zu fokussiert für zufällige Entdeckungen.
- **Alfred Hitchcock**:
  - **Modus**: Offen

  - **Wie**: Er erzählte lustige Anekdoten, um das Team zum Lachen zu bringen – so schuf er eine entspannte Atmosphäre, die kreatives Denken förderte.

1.2 Aufgabe 2

i)
- x64: 16 64 Bit GPRs[1] ⇒ 16 x 64 b = 16 x 8 B = 2^7 B.
- AVX2: 16 256 Bit GPRs[2] ⇒ 16 x 256 b = 16 x 32 B = 2^9 B

ii)
- x64: $\frac{2^7}{2^{30}} = \frac{1}{2^{23}}$
- AVX2: $\frac{2^9}{2^{30}} = \frac{1}{2^{21}}$

allgemein gilt: $10^3 \approx 2^{10}$, und $\frac{2^x}{2^y} = \frac{1}{2^{y-x}}$

1.3 Aufgabe 3

- Der Zugriff scheitert, weil der Arbeitsspeicher durch die **Memory Protection** (z. B. Paging mit Zugriffsrechten) vom Betriebssystem isoliert wird. Nur der Kernel darf die Speicherbereiche aller Prozesse sehen und verwalten.
- Ein Prozess kann trotzdem auf Ressourcen anderer Prozesse zugreifen über kontrollierte Schnittstellen wie IPC (Inter-Process Communication), Dateisysteme, Sockets oder Shared Memory, die vom Betriebssystem verwaltet und überwacht werden.
- Welche Risiken entstehen bei höchstem Privileg für alle Prozesse?
  - **Sicherheitslücken**: Jeder Prozess könnte beliebige Speicherbereiche lesen/schreiben.

[1] https://www.wikiwand.com/en/articles/X86-64
[2] https://www.wikiwand.com/en/articles/Advanced_Vector_Extensions

- **Stabilitätsprobleme**: Fehlerhafte Prozesse könnten das System zum Absturz bringen.

- **Keine Isolation**: Malware hätte vollen Systemzugriff, keine Schutzmechanismen.

1.4 Aufgabe 4

Kernel-Code benötigt einen sicheren, kontrollierten Speicherbereich (seinen eigenen Stack), um zu vermeiden:

- Beschädigung durch Benutzerprozesse
- Abstürze oder Rechteausweitung (Privilege Escalation)

Daher hat jeder Prozess:

- Einen User-Mode-Stack (wird bei normaler Ausführung verwendet)
- Einen Kernel-Mode-Stack (wird bei System Calls und Interrupts verwendet)

1.5 Aufgabe 5

Entfernte Systemaufrufe

Systemaufruf	Grund für Entfernung
creat	Entspricht vollständig `open(path, O_CREAT | O_WRONLY | O_TRUNC, mode)`.
dup	Entspricht vollständig `fcntl(fd, F_DUPFD, 0)`.

Alle übrigen Systemaufrufe bieten **essenzielle Funktionen**, die nicht exakt durch andere ersetzt werden können.
Sie decken ab:

- Datei- und Verzeichnisoperationen (`open`, `read`, `write`, `unlink`, `mkdir`, etc.)
- Prozessmanagement (`fork`, `exec`, `wait`, `exit`, etc.)
- Metadatenverwaltung (`chmod`, `chown`, `utime`, etc.)
- Kommunikation und Steuerung (`pipe`, `kill`, `ioctl`, etc.)
- Zeit- und Systemabfragen (`time`, `times`, `stat`, etc.)

Ohne sie wären bestimmte Kernfunktionen unmöglich.

1.6 Aufgabe 6

`script.sh` auch im Zip:

```
cd $1
while :
do
    echo "5 biggest files in $1:"
    ls -S | head -5
    echo "5 last modified files starting with '$2' in $1:"
    ls -t | grep ^$2 | head -5
    sleep 5
done
```

1.7 Aufgabe 7

Vorteile:

- **Komplexitätsreduktion:** Abstraktionen verbergen technische Details und erleichtern das Entwickeln und Verstehen von Systemen.
- **Wiederverwendbarkeit:** Einmal geschaffene Abstraktionen (z.B. Dateisystem, Prozesse) können flexibel in verschiedenen Programmen genutzt werden.

Nachteile:

- **Leistungsaufwand:** Abstraktionsschichten können zusätzliche Rechenzeit und Speicherverbrauch verursachen.
- **Fehlerverdeckung:** Probleme in tieferen Schichten bleiben oft verborgen und erschweren Fehlersuche und Optimierung.

2 Blatt 02

2.1 Aufgabe 1

Die Datenstruktur `task_struct` ist im Linux-Kernel-Quellcode (Linux kernel Version **6.15.0**) definiert unter:

include/linux/sched.h
Die Definition erstreckt sich über die Zeilen **813 bis 1664**.

Darin befinden sich etwa **320 Member-Variablen**.
Bei einer Annahme von 8 Byte pro Variable ergibt sich eine geschätzte Größe von:

2.560 Byte ≈ 2,5 KB

2.2 Aufgabe 2

Der Systemaufruf `fork()` erzeugt einen neuen Prozess, der eine Kopie des aufrufenden Prozesses ist (Kindprozess).

Rückgabewert:

- **0** im Kindprozess
- **PID des Kindes** im Elternprozess
- **−1** bei Fehler

a) Mit dem program:

```c
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int i = 0;
    if (fork() != 0) i++;
    if (i != 1) fork();
    fork();
```

```
    return 0;
}
```

werden insgesammt **6** Prozesse erzeugt. Graph der enstehenden Prozess hierarchie:

```
P1
  P1.1
    P1.1.1
      P1.1.1.1
    P1.1.2
  P1.2
```

Schrittweise Erzeugung der Prozesse:

1. **P1** startet das Programm. Der Wert von `i` ist anfangs `0`.
2. Die erste `fork()`-Anweisung wird ausgeführt:
   - **P1** ist der Elternprozess, der einen neuen Kindprozess **P1.1** erzeugt.
   - Im Elternprozess (**P1**) ist das Rückgabewert von `fork()` $0 \rightarrow$ `i` wird auf `1` gesetzt.
   - Im Kindprozess (**P1.1**) ist das Rückgabewert `0` $\rightarrow$ `i` bleibt `0`.
3. Danach folgt die Bedingung `if (i != 1) fork();`:
   - **P1** hat `i == 1` $\rightarrow$ keine Aktion.
   - **P1.1** hat `i == 0` $\rightarrow$ führt eine `fork()` aus $\rightarrow$ erzeugt **P1.1.1**.
4. Schließlich wird eine letzte `fork();` von **allen existierenden Prozessen** ausgeführt:
   - **P1** erzeugt **P1.2**
   - **P1.1** erzeugt **P1.1.2**
   - **P1.1.1** erzeugt **P1.1.1.1**

b) Das Programm führt `fork()` aus, bis ein Kindprozess mit einer durch 10 teilbaren PID entsteht. Jeder `fork()` erzeugt ein Kind, das sofort endet (die Rückgabe von `fork()` is 0 bei einem Kind), außer die Bedingung ist erfüllt. Da etwa jede zehnte PID durch 10 teilbar ist, liegt die **maximale Prozessanzahl** (inkl. Elternprozess) typischerweise bei **etwa 11**.

Da PIDs vom Kernel **in aufsteigender Reihenfolge als nächste freie Zahl** vergeben werden, ist garantiert, dass früher oder später eine durch 10 teilbare PID erzeugt wird. Das Programm terminiert daher immer. Wären PIDs zufällig, könnte es theoretisch unendlich laufen.

Startende oder endende Prozesse können die PID-Vergabe beeinflussen, da sie die Reihenfolge freier PIDs verändern – dadurch variiert die genaue Prozessanzahl je nach Systemzustand.

2.3 Aufgabe 3

Erklärung zur Ausgabe von `ps -T -H`

Das C-Programm:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char const *argv[])
{
    if (fork() > 0) sleep(1000);
    else exit(0);
    return 0;
}
```

erzeugt einen Kindprozess. Das Kind beendet sich sofort (`exit(0)`), während der Elternprozess 1000 Sekunden schläft (`sleep(1000)`).

Ablauf der Kommandos:

1. Das Ausführen von `./test &`:
   - Das Programm läuft im Hintergrund.
   - Die Shell gibt `[1] 136620` aus → Prozess-ID (PID) 136620.
   - Der Kindprozess wird erzeugt und terminiert sofort.
   - Der Elternprozess schläft weiter.
   - Da `wait()` **nicht** aufgerufen wird, wird der Kindprozess zu einem **Zombie-Prozess**.
2. Das Ausführen von `./test` und das drücken von `<Strg>+Z` danach:
   - Das Programm startet im Vordergrund.
   - Mit `<Strg>+Z` wird es gestoppt.
   - Die Shell zeigt: `[2]+  Stopped ./test`.
   - Auch hier terminiert der Kindprozess sofort → Zombie-Prozess entsteht erneut.

Ausgabe von `ps -T -H`:

```
   PID TTY      STAT   TIME COMMAND
  1025 pts/0    Ss     0:00 /bin/bash --posix
136620 pts/0    S      0:00   ./test
136621 pts/0    Z      0:00     [test] <defunct>
136879 pts/0    T      0:00   ./test
136880 pts/0    Z      0:00     [test] <defunct>
136989 pts/0    R+     0:00   ps T -H
```

Erklärung:

- `1025`: Die Shell (`bash`), läuft im Terminal `pts/0`.
- `136620`: Erstes `./test`-Programm, läuft im Hintergrund, schläft (`S`).
- `136621`: Dessen Kindprozess (Zombie, `Z`), da `exit()` aufgerufen wurde, aber vom Elternprozess nicht abgeholt.
- `136879`: Zweites `./test`-Programm, wurde mit `<Strg+Z>` gestoppt (`T`).
- `136880`: Auch hier: Kindprozess wurde beendet, aber nicht „abgeholt“ → Zombie.
- `136989`: Der `ps`-Prozess selbst, der gerade die Ausgabe erzeugt (`R+` = laufend im Vordergrund).

Die Spalten

- **PID**: Prozess-ID.
- **TTY**: Terminal, dem der Prozess zugeordnet ist.
- **STAT**: Prozessstatus:
  - `S`: sleeping – schläft.
  - `T`: stopped – gestoppt (z. B. durch `SIGSTOP`).
  - `Z`: zombie – beendet, aber noch nicht „aufgeräumt“.
  - `R`: running – aktuell laufend auf der CPU.
  - `+`: Teil der Vordergrund-Prozessgruppe im Terminal.
- **TIME**: CPU-Zeit, die der Prozess verbraucht hat.
- **COMMAND**: Der auszuführende Befehl.
  - `[test] <defunct>` heißt, es handelt sich um einen Zombie-Prozess, dessen Kommandozeile nicht mehr verfügbar ist.

Process state Codes

Prozesszustände (erste Buchstaben):

Code	Meaning	Description
`R`	Running	Currently running or ready to run (on CPU)

Code	Meaning	Description
`S`	Sleeping	Waiting for an event (e.g., input, timer)
`D`	Uninterruptible sleep	Waiting for I/O (e.g., disk), cannot be killed easily
`T`	Stopped	Process has been stopped (e.g., `SIGSTOP`, Ctrl+Z)
`Z`	Zombie	Terminated, but not yet cleaned up by its parent
`X`	Dead	Process is terminated and should be gone (rarely shown)

Zusätzliche flags:

Flag	Meaning
`<`	High priority (not nice to others)
`N`	Low priority (nice value > 0)
`L`	Has pages locked in memory
`s`	Session leader
`+`	In the foreground process group
`l`	Multi-threaded (using CLONE_THREAD)
`p`	In a separate process group

Z.B. `Ss+` beduetet: Sleeping (S), Session leader (s) & Foreground process (+).

Tiefe der Aktuellen Sitzung

Zuerst finden wir die PID der Aktuellen sitzung mit

```
echo $$
```

heraus. Output: `1025`.

Danch führen wir das Command `ps -eH | less` aus und suchen im pager nach "1025". In unserer Sitzung befand sich "bash" unter der Hierarchie:

```
1 systemd
    718 ssdm
        766 ssdm-helper
            859 i3
                884 kitty
                    1025 bash
```

Das entspricht der Tiefe **5** des Prozessbaums.

2.4 Aufgabe 4

Übersicht der Varianten mit Signaturen:

Funktion	Signatur
`execl`	`int execl(const char *path, const char *arg0, ..., NULL);`
`execle`	`int execle(const char *path, const char *arg0, ..., NULL, char *const envp[]);`
`execlp`	`int execlp(const char *file, const char *arg0, ..., NULL);`
`execv`	`int execv(const char *path, char *const argv[]);`
`execvp`	`int execvp(const char *file, char *const argv[]);`
`execvpe`	`int execvpe(const char *file, char *const argv[], char *const envp[]);`
`execve`	`int execve(const char *filename, char *const argv[], char *const envp[]);`

Wichtige Unterschiede:

- `l` = Argumente als **Liste** (z. B. `execl`)
- `v` = Argumente als **Array (vector)** (z. B. `execv`)
- `p` = **PATH-Suche** aktiv (z. B. `execvp`)
- `e` = **eigene Umgebung (envp[])** möglich (z. B. `execle`, `execvpe`)
- Kein `p` = voller Pfad zur Datei nötig
- Kein `e` = aktuelle Umgebungsvariablen werden übernommen

Wann welche Variante?

Variante	Typischer Einsatzzweck
`execl`	Fester Pfad und Argumente direkt im Code als Liste
`execle`	Wie `execl`, aber mit **eigener Umgebung**
`execlp`	Wie `execl`, aber **PATH-Suche** aktiviert (z. B. `ls` statt `/bin/ls`)
`execv`	Pfad bekannt, Argumente liegen **als Array** vor (z. B. aus `main`)
`execvp`	Wie `execv`, aber **mit PATH-Suche** (typisch für Shells)
`execvpe`	Wie `execvp`, aber mit **eigener Umgebung** (GNU-spezifisch)
`execve`	Low-Level, **volle Kontrolle** über Pfad, Argumente und Umgebung

2.5 Aufgabe 5

Ein Prozesswechsel (Context Switch) tritt auf, wenn das Betriebssystem (OS) die Ausführung eines Prozesses stoppt und zu einem anderen wechselt. Dabei entsteht Overhead, weil:

- Der aktuelle CPU-Zustand (Register, Programmzähler etc.) gespeichert werden muss
- Dieser Zustand im Prozesskontrollblock (PCB) abgelegt wird
- Der Zustand des neuen Prozesses aus seinem PCB geladen wird
- Die Speicherverwaltungsstrukturen (z. B. Seitentabellen der MMU) aktualisiert werden müssen
- Der TLB (Translation Lookaside Buffer) meist ungültig wird und geleert werden muss
- Weitere OS-Daten wie Datei-Deskriptoren oder Signale angepasst werden müssen

Der PCB enthält:

- Prozess-ID, Zustand
- Register, Programmzähler
- Speicherinfos, geöffnete Dateien
- Scheduling-Infos

Beim Prozesswechsel speichert das OS den PCB des alten Prozesses und lädt den neuen, um eine korrekte Fortsetzung zu ermöglichen. Da jeder Prozess einen eigenen Adressraum besitzt, ist der Aufwand für das Umschalten entsprechend hoch.

Threads desselben Prozesses teilen sich hingegen denselben Adressraum (also denselben Code, Heap, offene Dateien etc.). Das bedeutet:

- Es ist kein Wechsel des Adressraums nötig
- Die MMU- und TLB-Einträge bleiben gültig
- Nur der Thread-spezifische Kontext (Register, Stack-Pointer etc.) muss gespeichert werden

Fazit: Ein Threadwechsel ist viel leichter und schneller**, da kein teurer Speicherverwaltungswechsel nötig ist.

2.6 Aufgabe 6

1. In der ursprünglichen Version werden alle Threads schnell hintereinander gestartet, ohne aufeinander zu warten. Da die Ausführung der Threads vom Scheduler (Betriebssystem) abhängt und parallel erfolgt, kann die Ausgabe beliebig vermischt erscheinen – z. B. kann ein Thread seine Nachricht „number: i“ ausgeben, noch bevor die Hauptfunktion „creating thread i“ gedruckt hat.

   In der überarbeiteten Version hingegen wird jeder Thread direkt nach dem Start mit `pthread_join` wieder eingesammelt. Dadurch läuft immer nur ein Thread zur Zeit, und seine Ausgabe erfolgt vollständig, bevor der nächste beginnt. So entsteht eine streng sequentielle Ausgabe:

   - „creating thread i“

- „number: i“
- „ending thread i“

Diese einfache Struktur vermeidet Race Conditions und benötigt keine zusätzlichen Synchronisationsmechanismen wie Semaphoren oder Locks.

Überarbeitete Version (auch im zip als `threads_example.c` enthalten):

Listing 2.1 `threads_example.c`

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS 200000

void* TaskCode (void* argument)
{
   int tid = *((int*) argument);
   printf("number: %d\n", tid);
   printf("ending thread %d\n", tid);
   return NULL;
}

int main()
{
   pthread_t thread;
   int thread_arg;

   for (int i = 0; i < NUM_THREADS; i++) {
      thread_arg = i;
      printf("creating thread %d\n", i);
      int rc = pthread_create(&thread, NULL, TaskCode, &thread_arg);
      assert(rc == 0);
      rc = pthread_join(thread, NULL);
      assert(rc == 0);
   }

   return 0;
}
```

2. In unserem System $N_{\max} \approx 200000$.
3. Im folgenden Program wird `TaskCode()` $N_{\max}$ mal in einer einfachen Schleife aufgerufen:

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS 200000

void* TaskCode (void* argument)
{
   int tid = *((int*) argument);
   printf("number: %d\n", tid);
   printf("ending thread %d\n", tid);
   return NULL;
}

int main()
{
   for (int i = 0; i < NUM_THREADS; i++) {
      TaskCode(&i);
   }

   return 0;
}
```

Die Ausführung dieses Programs dauerte c. 2 Sekunden auf unserem System. D.h. die fehlenden zwei `pthread_*` aufrufe kosten

c. 8 Sekunden für 200000 Schleifen. Das entspricht c. 20 millisekunden pro `pthread_*` Aufruf.