

Operating Systems and Networks SoSe 25 Solutions

Igor Dimitrov

2024-12-18

Table of contents

| | |
|---|-----------|
| Preface | 5 |
| 1 Blatt 01 | 6 |
| 1.1 Aufgabe 1 | 6 |
| 1.2 Aufgabe 2 | 7 |
| 1.3 Aufgabe 3 | 7 |
| 1.4 Aufgabe 4 | 8 |
| 1.5 Aufgabe 5 | 8 |
| 1.6 Aufgabe 6 | 9 |
| 1.7 Aufgabe 7 | 9 |
| 2 Blatt 02 | 10 |
| 2.1 Aufgabe 1 | 10 |
| 2.2 Aufgabe 2 | 10 |
| 2.3 Aufgabe 3 | 12 |
| Erklärung zur Ausgabe von <code>ps -T -H</code> | 12 |
| Process state Codes | 13 |
| Tiefe der Aktuellen Sitzung | 14 |
| 2.4 Aufgabe 4 | 15 |
| 2.5 Aufgabe 5 | 15 |
| 2.6 Aufgabe 6 | 16 |
| 3 Blatt 03 | 19 |
| 3.1 Aufgabe 1 | 19 |
| 3.2 Aufgabe 3 | 20 |
| 3.3 Aufgabe 4 | 24 |
| Pseudocode: | 25 |
| 4 Blatt 04 | 28 |
| 4.1 Aufgabe 1 | 28 |
| 4.2 Aufgabe 2 | 28 |
| 4.3 Aufgabe 3 | 29 |
| 4.4 Aufgabe 4 | 30 |
| 4.5 Aufgabe 5 | 31 |
| 4.6 Aufgabe 6 | 31 |

| | | |
|----------|--|-----------|
| 4.7 | Aufgabe 7 | 32 |
| | Berechnung der Seitennummern und Offsets: | 32 |
| | C-Code: | 32 |
| 5 | Blatt 05 | 34 |
| 5.1 | Aufgabe 1 | 34 |
| 5.2 | Aufgabe 2 | 34 |
| | Adressübersetzung bei Paging mit Seitengröße 2^k | 34 |
| | Umkehrung der Formel | 34 |
| | Gegebene Zuordnungen | 35 |
| | Endgültige rekonstruierte Seitentabelle | 35 |
| | Python Implementierung | 36 |
| 5.3 | Aufgabe 3 | 36 |
| 5.4 | Aufgabe 4 | 40 |
| 5.5 | Aufgabe 6 | 41 |
| 5.6 | Aufgabe 7 | 41 |
| 5.7 | Aufgabe 8 | 43 |
| 6 | Blatt 06 | 45 |
| 6.1 | Aufgabe 1 | 45 |
| 6.2 | Aufgabe 2 | 47 |
| 6.3 | Aufgabe 3 | 48 |
| 6.4 | Aufgabe 4 | 51 |
| 6.5 | Aufgabe 5 | 52 |
| 6.6 | Aufgabe 6 | 52 |
| | Beispielausführung | 53 |
| 7 | Blatt 07 | 55 |
| 7.1 | Aufgabe 1 | 55 |
| 7.2 | Aufgabe 2 | 55 |
| 7.3 | Aufgabe 3 | 55 |
| 7.4 | Aufgabe 4 | 56 |
| 7.5 | Aufgabe 5 | 56 |
| 7.6 | Aufgabe 6 | 57 |
| 7.7 | Aufgabe 7 | 57 |
| | FCFS (First-Come, First-Serve) | 58 |
| | SSTF (Shortest Seek Time First) | 58 |
| | SCAN (Fahrstuhl-Algorithmus) | 60 |
| | C-SCAN (Circular SCAN) | 60 |
| | C-LOOK (Circular LOOK) | 61 |
| | Vergleichstabelle: Disk Scheduling | 62 |
| 7.8 | Aufgabe 8 | 62 |
| 7.9 | Aufgabe 9 | 63 |

| | | |
|----------|---------------------|-----------|
| 8 | Blatt 8 | 64 |
| 8.1 | Aufgabe 1 | 64 |
| 8.2 | Aufgabe 2 | 66 |
| 8.3 | Aufgabe 4 | 68 |
| 8.4 | Aufgabe 5 | 70 |
| 8.5 | Aufgabe 6 | 71 |
| 8.6 | Aufgabe 7 | 76 |

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** \Leftrightarrow **Diffus**: Für Kreativität und Überblick.
- **Geschlossen** \Leftrightarrow **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 GPRs¹ $\Rightarrow 16 \times 64 \text{ b} = 16 \times 8 \text{ B} = 2^7 \text{ B}$.
 - AVX2: 16 256 Bit GPRs² $\Rightarrow 16 \times 256 \text{ b} = 16 \times 32 \text{ B} = 2^9 \text{ B}$
 - 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>

²https://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 Rechteauserweiterung (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 <code>open(path, O_CREAT O_WRONLY O_TRUNC, mode)</code> . |
| dup | Entspricht vollständig <code>fcntl(fd, F_DUPFD, 0)</code> . |

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 \approx 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:

```
#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 insgesamt **6** Prozesse erzeugt. Graph der entstehenden 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()` ist 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:

```
#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+** bedeutet: 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 |
|----------------------|---|
| <code>execl</code> | <code>int execl(const char *path, const char *arg0, ..., NULL);</code> |
| <code>execle</code> | <code>int execle(const char *path, const char *arg0, ..., NULL, char *const envp[]);</code> |
| <code>execlp</code> | <code>int execlp(const char *file, const char *arg0, ..., NULL);</code> |
| <code>execv</code> | <code>int execv(const char *path, char *const argv[]);</code> |
| <code>execvp</code> | <code>int execvp(const char *file, char *const argv[]);</code> |
| <code>execvpe</code> | <code>int execvpe(const char *file, char *const argv[], char *const envp[]);</code> |
| <code>execve</code> | <code>int execve(const char *filename, char *const argv[], char *const envp[]);</code> |

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 |
|----------------------|---|
| <code>execl</code> | Fester Pfad und Argumente direkt im Code als Liste |
| <code>execle</code> | Wie <code>execl</code> , aber mit eigener Umgebung |
| <code>execlp</code> | Wie <code>execl</code> , aber PATH-Suche aktiviert (z. B. <code>ls</code> statt <code>/bin/ls</code>) |
| <code>execv</code> | Pfad bekannt, Argumente liegen als Array vor (z. B. aus <code>main</code>) |
| <code>execvp</code> | Wie <code>execv</code> , aber mit PATH-Suche (typisch für Shells) |
| <code>execvpe</code> | Wie <code>execvp</code> , aber mit eigener Umgebung (GNU-spezifisch) |
| <code>execve</code> | 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`

```
#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:

```

#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.

3 Blatt 03

3.1 Aufgabe 1

- a) Die Ausgabe ist **inkonsistent** – bei mehreren Programmausführungen erscheinen unterschiedliche Werte für **counter**. Dies liegt an einer **Race Condition**, da beide Threads gleichzeitig und ohne Synchronisation auf die gemeinsame Variable **counter** zugreifen. Dadurch können Zwischenergebnisse überschrieben oder verloren gehen, je nachdem, wie der Scheduler die Threads abwechselnd ausführt.
- b) Synchronisierte Lösung (Java-Code):

```
public class Counter {
    static int counter = 0;

    public static class Counter_Thread_A extends Thread {
        public void run() {
            synchronized (Counter.class) {
                counter = 5;
                counter++;
                counter++;
                System.out.println("A-Counter: " + counter);
            }
        }
    }

    public static class Counter_Thread_B extends Thread {
        public void run() {
            synchronized (Counter.class) {
                counter = 6;
                counter++;
                counter++;
                counter++;
                counter++;
                System.out.println("B-Counter: " + counter);
            }
        }
    }
}
```

```

    }
}

public static void main(String[] args) {
    Thread a = new Counter_Thread_A();
    Thread b = new Counter_Thread_B();
    a.start();
    b.start();
}
}

```

Erklärung:

Dieses Programm vermeidet das Race Condition-Problem, indem beide Threads einen `synchronized`-Block verwenden, der auf `Counter.class` synchronisiert ist. Das bedeutet:

- Nur ein Thread darf gleichzeitig den Block betreten.
- Der andere Thread muss warten, bis der erste fertig ist und den Lock freigibt.
- Dadurch wird sichergestellt, dass keine gleichzeitigen Zugriffe auf die gemeinsame Variable `counter` stattfinden.

3.2 Aufgabe 3

- a) Unten folgt der Quellcode zur verbesserten Lösung des Producer-Consumer-Problems (`pc2.c` am Ende des Dokuments). In dieser Version wird Busy Waiting durch eine effiziente Synchronisation mithilfe eines Mutexes und einer Condition Variable ersetzt.

Der Code befindet sich auch im beigefügten Zip-Archiv im Ordner A3. Dort kann das Programm wie folgt kompiliert und ausgeführt werden:

```

make
./pc2

```

Diese Implementierung gewährleistet eine korrekte und effiziente Koordination zwischen Producer- und Consumer-Threads:

- Die gemeinsame Warteschlange wird durch einen Mutex geschützt.
- Threads, die auf eine Bedingung warten, verwenden `pthread_cond_wait()` innerhalb einer `while`-Schleife, um Spurious Wakeups korrekt zu behandeln.
- Ist die Warteschlange leer, schlafen die Consumer, bis sie ein Signal erhalten; ist sie voll, wartet der Producer entsprechend.

- Durch das gezielte Aufwecken via `pthread_cond_signal()` oder `pthread_cond_broadcast()` wird unnötiger CPU-Verbrauch durch aktives Warten vermieden.

Insgesamt ist diese Lösung robuster und skalierbarer als die ursprüngliche Variante mit Busy Waiting – insbesondere bei mehreren Consumer-Threads und höherer Auslastung.

b) Laufzeitvergleich von `pc` und `pc2`

Zur Überprüfung der Effizienzverbesserung durch den Einsatz von Condition Variables wurde folgendes Bash-Skript verwendet, das beide Programme je 10-mal ausführt und die durchschnittliche Laufzeit berechnet:

```
#!/bin/bash

RUNS=10
PC="./pc"
PC2="./pc2"

measure_average_runtime() {
    PROGRAM=$1
    TOTAL=0
    echo "Running $PROGRAM..."
    for i in $(seq 1 $RUNS); do
        START=$(date +%s.%N)
        $PROGRAM > /dev/null
        END=$(date +%s.%N)
        RUNTIME=$(echo "$END - $START" | bc)
        echo "  Run $i: $RUNTIME seconds"
        TOTAL=$(echo "$TOTAL + $RUNTIME" | bc)
    done
    AVG=$(echo "scale=4; $TOTAL / $RUNS" | bc)
    echo "Average runtime of $PROGRAM: $AVG seconds"
    echo
}

echo "Measuring $RUNS runs of $PC and $PC2..."
echo
measure_average_runtime $PC
measure_average_runtime $PC2
```

Ausgeführt wurde das Skript mit:

```
./benchmark_pc.sh
```

Dabei ergaben sich folgende Laufzeiten:

Measuring 10 runs of ./pc and ./pc2...

Running ./pc...

```
Run 1: 5.471139729 seconds
Run 2: 5.545249360 seconds
Run 3: 5.359090183 seconds
Run 4: 5.366634866 seconds
Run 5: 5.459910579 seconds
Run 6: 5.531161091 seconds
Run 7: 5.738575161 seconds
Run 8: 5.835055657 seconds
Run 9: 5.496744966 seconds
Run 10: 5.641529848 seconds
```

Average runtime of ./pc: 5.5445 seconds

Running ./pc2...

```
Run 1: 5.244080521 seconds
Run 2: 5.237442233 seconds
Run 3: 5.220517776 seconds
Run 4: 5.281094089 seconds
Run 5: 5.261722379 seconds
Run 6: 5.363685993 seconds
Run 7: 5.276107150 seconds
Run 8: 5.091557858 seconds
Run 9: 5.073267276 seconds
Run 10: 5.164472482 seconds
```

Average runtime of ./pc2: 5.2213 seconds

Die Ergebnisse zeigen, dass pc2 im Schnitt etwas schneller ist als pc (5.22s gegenüber 5.54s), was den Effizienzgewinn durch den Verzicht auf aktives Warten bestätigt.

Die Dateien `benchmark_pc.sh` und `benchmark_results.txt` befinden sich im Ordner A3 des ZIP-Archivs.

Zur Veranschaulichung wurde mit dem folgenden Python script zusätzlich ein Diagramm erstellt, das die Laufzeiten von pc und pc2 über zehn Durchläufe hinweg zeigt. Die Durchschnittslinien verdeutlichen, dass pc2 im Mittel schneller und konsistenter ist als pc.

```
import matplotlib.pyplot as plt

# Runtime data for each run (in seconds)
pc = [5.471139729, 5.545249360, 5.359090183, 5.366634866, 5.459910579,
      5.531161091, 5.738575161, 5.835055657, 5.496744966, 5.641529848]
```

```

pc2 = [5.244080521, 5.237442233, 5.220517776, 5.281094089, 5.261722379,
       5.363685993, 5.276107150, 5.091557858, 5.073267276, 5.164472482]

# X-axis: run numbers
runs = list(range(1, 11))

# Calculate averages
avg_pc = sum(pc) / len(pc)
avg_pc2 = sum(pc2) / len(pc2)

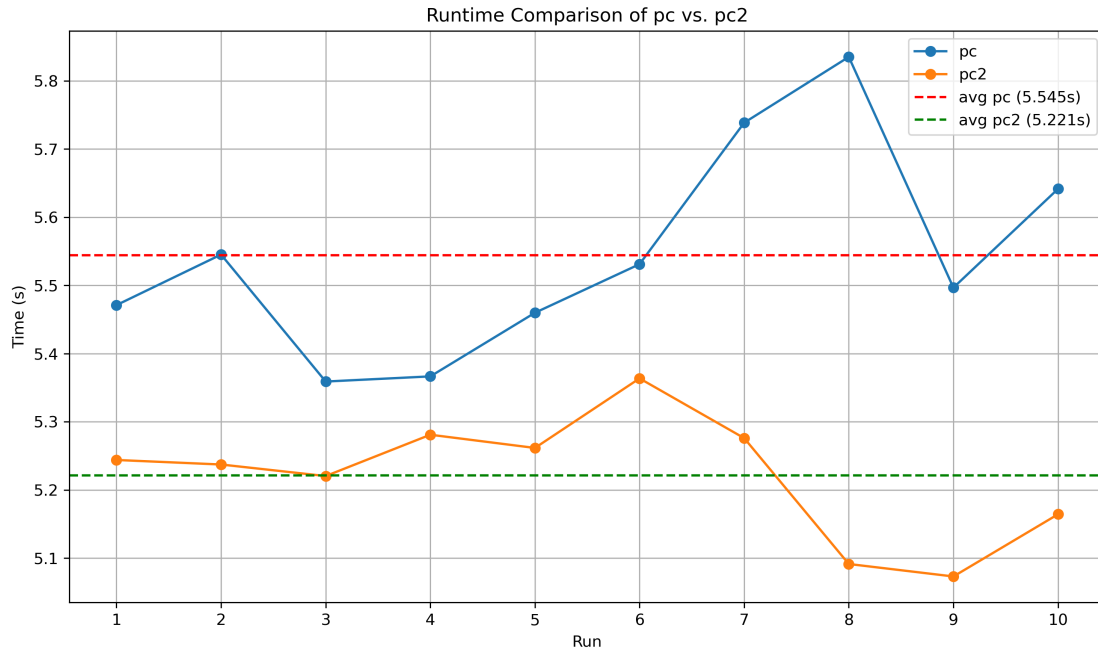
# Plot configuration
plt.figure(figsize=(10, 6))
plt.plot(runs, pc, marker='o', label='pc')
plt.plot(runs, pc2, marker='o', label='pc2')

# Average lines
plt.axhline(avg_pc, color='red', linestyle='--', label=f'avg pc ({avg_pc:.3f}s)')
plt.axhline(avg_pc2, color='green', linestyle='--', label=f'avg pc2 ({avg_pc2:.3f}s)')

# Labels and title
plt.xlabel('Run')
plt.ylabel('Time (s)')
plt.title('Runtime Comparison of pc vs. pc2')
plt.xticks(runs)
plt.grid(True)
plt.legend()
plt.tight_layout()

# Display the plot
plt.show()

```



3.3 Aufgabe 4

- a) Die gegebene Implementierung kann zu einer Verletzung des gegenseitigen Ausschlusses führen, wenn zwei schreibende Threads gleichzeitig in die kritische Sektion gelangen.

Beispiel: Angenommen $N = 5$. Thread A und Thread B rufen gleichzeitig `lock_write()` auf. Da der `for`-Loop nicht durch einen Mutex geschützt ist, können sich ihre `wait(S)`-Aufrufe gegenseitig durchmischen: A nimmt 1 Token $\rightarrow S = 4$ B nimmt 1 Token $\rightarrow S = 3$ A nimmt 1 $\rightarrow S = 2$ B nimmt 1 $\rightarrow S = 1$... und so weiter. Wenn nun zufällig genug Tokens freigegeben werden (z. B. durch `unlock_read()`-Aufrufe), können beide Threads nacheinander die restlichen Semaphore erwerben und ihren Loop abschließen, ohne dass einer von ihnen jemals alle N Tokens exklusiv gehalten hat. Beide betreten anschließend die kritische Sektion, obwohl gegenseitiger Ausschluss nicht mehr gewährleistet ist.

- b) Das Problem wird behoben, indem ein zusätzlicher Mutex eingeführt wird, der verhindert, dass mehrere schreibende Threads gleichzeitig versuchen, die Semaphore S zu erwerben:

```
S = Semaphore(N)
M = Semaphore(1) // neuer Mutex

def lock_read():
    wait(S)
```



```

def unlock_read():
    signal(S)

def lock_write():
    wait(M)
    for i in range(N): wait(S)
    signal(M)

def unlock_write():
    for i in range(N): signal(S)

```

Durch den Mutex *M* ist sichergestellt, dass der Erwerb der Semaphore in `lock_write()` **ausschließlich** von einem Thread durchgeführt wird. So wird verhindert, dass mehrere schreibende Threads gleichzeitig in die kritische Sektion gelangen.

Hinweis: Diese Lösung stellt den gegenseitigen Ausschluss sicher, erlaubt jedoch theoretisch, dass ein schreibender Thread dauerhaft blockiert bleibt, wenn ständig neue Leser auftreten (*Starvation*). Für diese Aufgabe ist jedoch nur die Korrektur der Ausschlussverletzung relevant.

- c) Die Befehle `upgrade_to_write()` und `downgrade_to_read()` ermöglichen es einem Thread, während des laufenden Zugriffs die Art des Read-Write-Locks dynamisch zu wechseln – ohne dabei den kritischen Abschnitt vollständig zu verlassen. Dies verhindert Race Conditions und potenzielle Starvation.

Ein Thread, der `upgrade_to_write()` aufruft, hält bereits einen Lesezugriff (also eine Einheit der Semaphore *S*) und möchte exklusiven Schreibzugriff erhalten. Dafür müssen die verbleibenden $N - 1$ Einheiten erworben werden. Ein zusätzlicher Mutex *M* sorgt dafür, dass nicht mehrere Threads gleichzeitig versuchen, sich hochzustufen, was zu Deadlocks führen könnte.

Ein Thread, der `downgrade_to_read()` aufruft, hält alle *N* Einheiten (Schreibzugriff) und möchte auf geteilten Lesezugriff wechseln. Dazu werden $N - 1$ Einheiten freigegeben – eine Einheit bleibt erhalten.

Hinweis: Das hier verwendete Mutex *M* ist dasselbe wie in Teil b) und stellt sicher, dass nur ein Thread gleichzeitig exklusiven Zugriff auf die Semaphore *S* erwerben kann – sei es über `lock_write()` oder über `upgrade_to_write()`.

Pseudocode:

```

S = Semaphore(N)      // erlaubt bis zu N gleichzeitige Leser oder 1 Schreiber
M = Semaphore(1)      // schützt exklusive Zugriffsversuche

```

```
def upgrade_to_write():
    wait(M)
    for i in range(N - 1):    // hält bereits 1 Einheit als Leser
        wait(S)
    signal(M)

def downgrade_to_read():
    for i in range(N - 1):    // gibt N - 1 Einheiten frei, behält 1
        signal(S)
```

Fazit: Diese Operationen garantieren einen sicheren Übergang zwischen Lese- und Schreibmodus, ohne Race Conditions oder Deadlocks, und basieren auf derselben Semaphor-Struktur wie in Teil b).

Listing 3.1 pc2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "mylist.h"
// Mutex to protect access to the shared queue
pthread_mutex_t queue_lock;
// Single condition variable used for both producers and consumers
pthread_cond_t cond_var;
// Shared buffer (a custom linked list acting as a queue)
list_t buffer;
// Counters for task management
int count_proc = 0;
int production_done = 0;
/*****
/* Function Declarations */
static unsigned long fib(unsigned int n);
static void create_data(elem_t **elem);
static void *consumer_func(void *);
static void *producer_func(void *);
*****/
/* Compute the nth Fibonacci number (CPU-intensive task) */
static unsigned long fib(unsigned int n)
{
    if (n == 0 || n == 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

/* Allocate and initialize a new task node */
static void create_data(elem_t **elem)
{
    *elem = (elem_t*) malloc(sizeof(elem_t));
    (*elem)->data = FIBONACCI_MAX;
}

/* Consumer thread function */
static void *consumer_func(void *args)
{
    elem_t *elem;
    while (1) {
        pthread_mutex_lock(&queue_lock);
        // Wait if the queue is empty and production is not yet complete
        while (get_size(&buffer) == 0 && !production_done) {
            pthread_cond_wait(&cond_var, &queue_lock);
        }
        // Exit condition: queue is empty and production has finished
        if (get_size(&buffer) == 0 && production_done) {
            pthread_mutex_unlock(&queue_lock);
            break;
        }
        // Remove an item from the queue
    }
}
```

4 Blatt 04

4.1 Aufgabe 1

- a) Ein Nachteil benannter Pipes ist, dass sie manuell im Dateisystem erstellt und verwaltet werden müssen (z. B. mit `mkfifo`). Das macht die Handhabung aufwändiger und erfordert gegebenenfalls zusätzliche Aufräumaßnahmen.
- b) Wenn zwei voneinander unabhängige Prozesse (z. B. zwei Terminals) Daten austauschen sollen, ist eine benannte Pipe erforderlich. Anonyme Pipes funktionieren nur zwischen verwandten Prozessen (z. B. Eltern-Kind).

4.2 Aufgabe 2

- a) Im Win32-API ist ein Handle vom Typ:

```
typedef void* HANDLE;
```

Es handelt sich also um einen Zeiger (bzw. zeigerbreiten Wert), der jedoch nicht dereferenziert werden soll. Ein Handle ist ein **undurchsichtiger Verweis** auf eine Ressource, die vom Windows-Kernel verwaltet wird – etwa eine Datei, ein Prozess, ein Event oder ein Fensterobjekt.

Wenn ein Programm zum Beispiel `CreateFile()` aufruft, gibt der Kernel einen solchen Handle zurück. Dieser verweist intern auf ein Objekt in der Handle-Tabelle des Prozesses. Diese Tabelle enthält Informationen wie Zugriffsrechte, aktuelle Dateiposition, Typ des Objekts usw.

Im Unterschied zu Dateideskriptoren unter Unix/Linux (einfache Ganzzahlen) sind Win32-Handles **allgemeiner gehalten** und dienen zum Zugriff auf viele verschiedene Ressourcentypen – nicht nur auf Dateien.

- b) Die Umleitung der Standardausgabe erfolgt im Win32-API in zwei Schritten:
 - 1. Eine Datei wird mit `CreateFile()` geöffnet oder erzeugt.
 - 2. Der Handle für `STD_OUTPUT_HANDLE` wird mit `SetStdHandle()` auf diesen Dateihandle gesetzt.

Beispiel:

```
#include <windows.h>
#include <stdio.h>

int main() {
    HANDLE hFile = CreateFile("output.txt", GENERIC_WRITE, 0, NULL,
                             CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Fehler beim Öffnen der Datei.\n");
        return 1;
    }

    // Standardausgabe umleiten
    SetStdHandle(STD_OUTPUT_HANDLE, hFile);

    // Alles, was an STD_OUTPUT_HANDLE geschrieben wird, geht nun in die Datei
    // DWORD written;
    WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),
              "Hello redirected world!\n", 24, &written, NULL);

    CloseHandle(hFile);
    return 0;
}
```

Diese Umleitung wirkt sich auf Low-Level-Funktionen wie `WriteFile()` aus. Wenn man dagegen höhere Funktionen wie `printf()` oder `std::cout` umleiten will, muss zusätzlich die Laufzeitumgebung angepasst werden – etwa mit `freopen()` oder `std::ios`-Umleitungen.

4.3 Aufgabe 3

Das Program: (Auch im Zip unter dem Verzeichniss A3 als `reverse_pipechat.c` enthalten)

Das C-Programm demonstriert die Kommunikation zwischen zwei Prozessen über anonyme Pipes. Der Elternprozess (A) liest eine Zeichenkette von der Standardeingabe und sendet sie an den Kindprozess (B). Dieser kehrt die Zeichenkette um und schickt sie zurück. Der Elternprozess gibt das Ergebnis anschließend auf der Standardausgabe aus.

Technisch funktioniert das Programm so: Es erstellt zwei Pipes – eine für die Kommunikation von A nach B, die andere für die Rückrichtung. Nach dem Aufruf von `fork()` schließt

jeder Prozess die jeweils nicht benötigten Enden der Pipes. Der Elternprozess sendet die Benutzereingabe an das Kind, das die Zeichenkette verarbeitet und die Antwort zurückschickt. Beide Prozesse verwenden `read()` und `write()` zur Datenübertragung und beenden sich danach.

Kompilieren und ausführen kann man das Programm unter Verzeichniss A3 mit:

```
make
./pipe_example
```

Beispielausgabe:

```
Enter a string: hallo welt
Reversed string: tlew ollah
```

4.4 Aufgabe 4

a) :

- Fragmentierung: Intern. (Eine geringe Anzahl von langlebigen Objekten existieren in einem Page, was zur internen Speicherverschwendung führt)
- Definition der Internen Fragmentierung (in diesem Kontext): Speicherverschwendung innerhalb der Seite

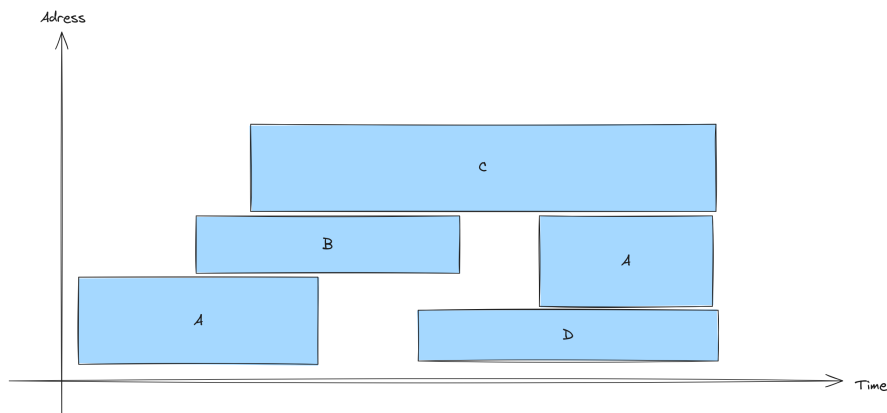


Figure 4.1: timing diagram

b)

c) :

- sehr häufig: fast immer handelt es sich um einen Tradeoff, z.B. beim best fit vs first fit handelt es sich um das Tradeoff Speichereffizienz vs Zeiteffizienz
- Tradeoff: Cache misses vs Interne Fragmentierung (Zeit vs Speicherplatz)
 - Kleine Seiten: Wenig interne Fragmentierung aber häufige Cache misses \Rightarrow Zeitverschwendung
 - Grosse Seiten: Seltene Cache misses aber sehr große interne Fragmentierung (da es häufig langlebige Objekte existieren) \Rightarrow Speicherverschwendung

4.5 Aufgabe 5

1) Interne vs. externe Fragmentierung:

- **Interne Fragmentierung** entsteht, wenn ein Prozess mehr Speicher zugewiesen bekommt, als er tatsächlich benötigt – z.B. bei festen Block- oder Seitengrößen bleibt ungenutzter Speicher *innerhalb* des Blocks.
- **Externe Fragmentierung** tritt auf, wenn der freie Speicher zwar insgesamt groß genug ist, aber in viele kleine, *nicht zusammenhängende* Stücke aufgeteilt ist, sodass größere Prozesse keinen passenden Platz finden.

2) Logische vs. physische Adressen:

- **Logische Adressen** (auch virtuelle Adressen) werden vom Prozess verwendet und beginnen meist bei 0 – sie sind unabhängig vom realen Speicherlayout.
- **Physische Adressen** geben die tatsächliche Position im Hauptspeicher (RAM) an. Das Betriebssystem bzw. die Hardware (MMU) wandelt logische Adressen zur Laufzeit in physische Adressen um.

4.6 Aufgabe 6

Kurze Erklärung zur Notation A:B: Der Segment der Größe A wurde der Speicherlücke der Größe B zugewiesen. (Das ist eindeutig, da die Größen der Segmente und der Lücken jeweils eindeutig sind.)

Dann:

- First fit:

12:20

11:18

3:10

5:7

- Best fit:

12:12
11:15
3:4
5:7

- Worst fit

12:20
11:18
3:15
5:12

4.7 Aufgabe 7

Da die Seitengröße 1 KB = 1024 Bytes = 2^{10} beträgt, entsprechen die unteren 10 Bit des virtuellen Adresse die Offset, die restlichen höheren Bits geben die Seitennummer an.

Berechnung der Seitennummern und Offsets:

| Adresse | Seitennummer | Offset |
|---------|--------------|--------|
| 2456 | 2 | 408 |
| 16382 | 15 | 1022 |
| 30000 | 29 | 304 |
| 4385 | 4 | 289 |

C-Code:

```
// V - virtuelle Adresse, gegeben
int p = V >> 10;           // Seitennummer
int offset = V & 0x3FF;    // Offset (2^10 - 1 = 1023)
```

Durch die Verwendung von Bitoperationen ist die Berechnung effizient, da die Seitengröße eine Zweierpotenz ist.

Listing 4.1 reverse_pipechat.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define BUFFER_SIZE 1024

// Utility: reverse a string in place
void reverse_string(char *str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; ++i) {
        char tmp = str[i];
        str[i] = str[len - 1 - i];
        str[len - 1 - i] = tmp;
    }
}

int main() {
    int pipe_a_to_b[2]; // parent writes to child
    int pipe_b_to_a[2]; // child writes to parent

    if (pipe(pipe_a_to_b) == -1 || pipe(pipe_b_to_a) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        // Child process: Process B
        close(pipe_a_to_b[1]); // Close write end of A→B
        close(pipe_b_to_a[0]); // Close read end of B→A

        char buffer[BUFFER_SIZE];

        // Read message from parent
        ssize_t bytes_read = read(pipe_a_to_b[0], buffer, BUFFER_SIZE - 1);
        if (bytes_read <= 0) {
            perror("child read");
            exit(EXIT_FAILURE);
        }

        buffer[bytes_read] = '\0'; // Null-terminate the string

        reverse_string(buffer); // Reverse the string
    }
}
```

5 Blatt 05

5.1 Aufgabe 1

- a)
- **Matrikelnummer:** Seitennummer (Virtuelle Adresse)
 - **Wohnadresse:** Rahmennummer (Physische Adresse)
 - **Verzeichniss:** Seitentabelle (Index)

keine Entsprechung zum Offset

- b) Matrikelnummer hat 7 Stellen: 10^7 , d.h. 10 Mil Einträge.
Es gibt 4.800 Wohnheimzimmer: relevanter Anteil = $\frac{4.8 \cdot 10^3}{10 \cdot 10^6} \approx 0.5 \cdot 10^{-4} = 0.05\%$

5.2 Aufgabe 2

Adressübersetzung bei Paging mit Seitengröße 2^k

Bei einem Paging-System mit fester Seitengröße von 2^k Byte wird die virtuelle Adresse V wie folgt in eine physische Adresse übersetzt:

$$\text{Physische Adresse} = (F(V \gg k) \ll k) + (V \& (2^k - 1))$$

- $V \gg k$: virtuelle Seitennummer (Integer-Division durch 2^k)
- $V \& (2^k - 1)$: Offset innerhalb der Seite
- $F(n)$: Seitentabelle, die virtuelle Seitennummer n auf Rahmennummer abbildet

Umkehrung der Formel

Wenn die virtuelle Adresse V und die zugehörige physische Adresse P gegeben sind, lässt sich der Seitentableneintrag rekonstruieren:

$$F(V \gg k) = \frac{P - (V \& (2^k - 1))}{2^k}$$

Bei Seitengröße von **4KB** ($2^{12} = 4096$) gilt:

$$F(V \gg 12) = \frac{P - (V \& 0xFFF)}{4096}$$

Gegebene Zuordnungen

- $V = 8203 \rightarrow P = 12229$
- $V = 4600 \rightarrow P = 25080$
- $V = 16510 \rightarrow P = 41086$

1. $V = 8203 \rightarrow P = 12229$

- Virtuelle Seite: $8203 \gg 12 = \lfloor \frac{8203}{4096} \rfloor = 2$
- Offset: $8203 \& 0xFFF = 8203 \bmod 4096 = 8203 - 8192 = 11$
- Frame-Berechnung:

$$F(2) = \frac{12229 - 11}{4096} = \frac{12218}{4096} = 2$$

2. $V = 4600 \rightarrow P = 25080$

- Virtuelle Seite: $4600 \gg 12 = 1$
- Offset: $4600 \& 0xFFF = 504$
- Frame-Berechnung:

$$F(1) = \frac{25080 - 504}{4096} = \frac{24576}{4096} = 6$$

3. $V = 16510 \rightarrow P = 41086$

- Virtuelle Seite: $16510 \gg 12 = \lfloor \frac{16510}{4096} \rfloor = 4$
- Offset: $16510 \& 0xFFF = 16510 - (4 \cdot 4096) = 126$
- Frame-Berechnung:

$$F(4) = \frac{41086 - 126}{4096} = \frac{40960}{4096} = 10$$

Endgültige rekonstruierte Seitentabelle

| Virtuelle Seitennummer | Physischer Rahmen |
|------------------------|-------------------|
| 2 | 2 |
| 1 | 6 |
| 4 | 10 |

Python Implementierung

Die Rekonstruktion der Tabelle kann mit python wie folgt implementiert werden:

```
def reconstruct_page_table(k, mappings):
    page_size = 1 << k # 2^k
    page_mask = page_size - 1

    page_table = []
    for virtual_address, physical_address in mappings:
        virtual_page_number = virtual_address >> k
        offset = virtual_address & page_mask
        frame_number = (physical_address - offset) >> k
        page_table.append((virtual_page_number, frame_number))

    return page_table

k = 12 # page size = 2^12 = 4096
mappings = [
    (8203, 12229),
    (4600, 25080),
    (16510, 41086)
]

page_table = reconstruct_page_table(k, mappings)
for vpn, frame in page_table:
    print(f"Virtuelle Seite {vpn} → Rahmen {frame}")
```

Virtuelle Seite 2 → Rahmen 2
 Virtuelle Seite 1 → Rahmen 6
 Virtuelle Seite 4 → Rahmen 10

5.3 Aufgabe 3

a) Ausgangssituation:

Betrachten wir die folgende C-Schleife auf einem System, bei dem gilt:

- `int` ist 4 Byte groß
- Seitengröße = 4 KB = 4096 Byte
- Der TLB (Translation Lookaside Buffer) hat 64 Einträge (d.h. er kann 64 Seiten zwischenspeichern)

Jede Seite enthält 1024 Integer, da 4 Byte pro Element. ($4096 / 4 = 1024$) Jeder Zugriff auf `X[i]` greift auf eine Speicherseite zu:

$$\text{Seitennummer}(i) = \left\lfloor \frac{i}{1024} \right\rfloor$$

Mit den indizes i :

$$i = 0, M, 2M, 3M, \dots, \text{solange } i < N$$

Setzen wir $T = \left\lfloor \frac{N-1}{M} \right\rfloor$, so gibt es $T + 1$ Iterationen.

Jede Iteration greift also auf die Seite zu:

$$\text{Seite}(j) = \left\lfloor \frac{j \cdot M}{1024} \right\rfloor \quad \text{für } j = 0, 1, \dots, T$$

Die Anzahl der **verschiedenen Seiten**, die beim Schleifendurchlauf berührt werden, ist:

$$\text{TLB_pages}(M, N) = \left| \left\{ \left\lfloor \frac{j \cdot M}{1024} \right\rfloor \mid 0 \leq j \leq \left\lfloor \frac{N-1}{M} \right\rfloor \right\} \right|$$

Dies ist die Anzahl der eindeutig unterschiedlichen Seiten, auf die zugegriffen wird. TLB-Misses treten auf, wenn:

$$\text{TLB_pages}(M, N) > 64$$

Das heißt: Es werden mehr als 64 unterschiedliche virtuelle Seiten benötigt – mehr als der TLB speichern kann.

Beispiele und wichtige Beobachtungen

1. Kleines M (z. B. $M = 1$):

- Aufeinanderfolgende Elemente werden zugegriffen.
- Alle 1024 Zugriffe \rightarrow 1 neue Seite.

- Insgesamt: ca. $N / 1024$ Seiten.
- TLB-Misses bei $N > 64 * 1024 = 65536$.

2. Großes M (z. B. M = 1024):

- Jeder Zugriff springt zu einer neuen Seite.
- Es werden ca. N / M unterschiedliche Seiten berührt.
- Wenn $N / M > 64$, treten TLB-Misses auf.

3. M teilt 1024 (z. B. M = 256, 512):

- Mehrere Schleifendurchläufe landen auf derselben Seite.
- Beispiel: $M = 256 \rightarrow 4$ Zugriffe pro Seite, bevor zur nächsten gewechselt wird.
- Weniger Seiten werden benötigt.
- **Weniger TLB-Misses**, auch bei großem N.

Schlechtester Fall für den TLB

Tritt auf, wenn:

- $M = 1024$ (jeder Zugriff auf eine neue Seite)
- und $N / M > 64$ (mehr als 64 Seiten werden benötigt)

Dann wird bei jedem Zugriff eine andere Seite nachgeladen \rightarrow **viele TLB-Misses**.

Fazit:

Die **Anzahl der verschiedenen Seiten**, auf die beim Schleifendurchlauf zugegriffen wird, lautet:

$$\text{TLB_pages}(M, N) = \left| \left\{ \left\lfloor \frac{j \cdot M}{1024} \right\rfloor \mid 0 \leq j \leq \left\lfloor \frac{N-1}{M} \right\rfloor \right\} \right|$$

- TLB-Misses treten auf, wenn $\text{TLB_pages}(M, N) > 64$
- Kleine M (vor allem wenn $M < 1024$ und M ein Teiler von 1024 ist) \rightarrow mehrere Zugriffe pro Seite \rightarrow **besseres TLB-Verhalten**
- Große M (> 1024) \rightarrow jeder Zugriff auf neue Seite \rightarrow **mehr TLB-Misses**

Dieses Verhalten kann mit der folgenden Python-funktion simuliert werden:

```
def tlb_pages(M, N, ints_per_page=1024):
    """
    Simulates the number of distinct pages accessed in the loop:
        for (int i = 0; i < N; i += M) X[i]++;

    Parameters:
```

```

- M: step size
- N: total number of elements in the array
- ints_per_page: number of integers that fit in a single page (default 4096 bytes / 4 bytes)

Returns:
- The number of distinct pages accessed
"""
pages = set()
for i in range(0, N, M):
    page = i // ints_per_page
    pages.add(page)
return len(pages)

```

Einige Simulationen:

```

print(tlb_pages(1, 65536))      # Should be 64 → fills exactly 64 pages
print(tlb_pages(1024, 65536))  # Should be 64 → each access on a new page
print(tlb_pages(256, 65536))   # Should be 64 / 4 = 16 → reuse of pages
print(tlb_pages(2048, 65536))  # Should be 32 → skips every second page
print(tlb_pages(1, 100000))    # result: 98 → causes TLB misses

```

64
64
64
32
98

b) Das Verhalten des TLB ändert sich deutlich, wenn der Code mehrfach oder regelmäßig ausgeführt wird – etwa in einer oft aufgerufenen Funktion oder in einer heißen Schleife.

1. TLB ist zustandsbehaftet und begrenzt

- Er kann nur eine bestimmte Anzahl an Seitenadressen speichern (z. B. 64).
- Wird die Anzahl der zugreifenden Seiten pro Schleife > 64 , kommt es zu Ersetzungen (evictions), meist nach dem LRU-Prinzip.

2. Wiederholte Ausführung kann TLB verbessern – oder verschlechtern

- Wenn dieselben Seiten wiederverwendet werden (z. B. bei $N = 64 * 1024$), bleiben TLB-Einträge erhalten → nach der ersten Ausführung keine weiteren Misses.
- Wenn mehr als 64 Seiten verwendet oder ständig neue Seiten benötigt werden, werden TLB-Einträge ständig ersetzt → TLB-Misses bei jedem Aufruf.

3. TLB-Arbeitsmenge (working set)

- Die „TLB-Arbeitsmenge“ ist die Menge der Seiten, die eine Funktion während der Ausführung benötigt.
- Passt diese Menge vollständig in den TLB, funktioniert alles effizient.
- Ist sie größer, kommt es zu wiederholten Zugriffen auf die Page Table → langsam.

4. Zugriffsart ist entscheidend

- Kleine Schrittweite M → viele Zugriffe auf dieselbe Seite → hohe Wiederverwendung → TLB effizient.
- Große Schrittweite $M = 1024$ → jeder Zugriff auf eine neue Seite → hoher TLB-Druck, vor allem bei vielen Funktionsaufrufen.

5.4 Aufgabe 4

Beim Wechsel zwischen Prozessen wird der TLB in der Regel geleert, da jeder Prozess einen eigenen virtuellen Adressraum mit einer eigenen Seitentabelle besitzt. Die im TLB gespeicherten Einträge des vorherigen Prozesses wären im neuen Kontext ungültig oder sogar sicherheitskritisch.

Beim Wechsel zwischen Threads desselben Prozesses bleibt der TLB hingegen erhalten, da alle Threads denselben Adressraum und dieselbe Seitentabelle nutzen. Die vorhandenen TLB-Einträge bleiben daher gültig.

Moderne Systeme mit *Address Space Identifiers* (ASIDs) können einen vollständigen TLB-Flush beim Prozesswechsel vermeiden, indem sie TLB-Einträge pro Prozess kennzeichnen und nur die jeweils relevanten aktiv halten.

Kein Flush - Was kann Schiefgehen?

Wenn der TLB beim Kontextwechsel nicht geleert wird, kann es zu schwerwiegenden Sicherheitsproblemen kommen. Das folgende Beispiel zeigt, was konkret passieren kann:

Angenommen, **Prozess A** greift auf die virtuelle Adresse `0x00400000` zu, welche in seiner Seitentabelle korrekt auf die physische Adresse `0x1A300000` abgebildet wird. Diese Übersetzung wird im TLB zwischengespeichert.

Nun findet ein Kontextwechsel zu **Prozess B** statt. Auch Prozess B verwendet die virtuelle Adresse `0x00400000`, aber in seiner eigenen Seitentabelle sollte sie auf eine völlig andere physische Adresse zeigen, z. B. `0x2B400000`.

Wenn der TLB **nicht geleert** wird, verwendet der Prozessor beim Zugriff durch Prozess B weiterhin die alte TLB-Eintragung von Prozess A. Das führt dazu, dass Prozess B auf den physischen Speicher von Prozess A zugreift.

Die Folgen:

- **Sicherheitslücke:** Prozess B kann sensible Daten von Prozess A einsehen.
- **Datenkorruption:** Schreibzugriffe von Prozess B verändern versehentlich die Daten von Prozess A.
- **Verletzung der Speicherisolation:** Ein zentrales Prinzip des Betriebssystems wird untergraben.

Um das zu verhindern, wird der TLB beim Wechsel des Prozesses entweder vollständig geleert oder — bei moderner Hardware — es werden **ASIDs** (Address Space Identifiers) verwendet, die TLB-Einträge pro Prozess kennzeichnen und voneinander trennen.

5.5 Aufgabe 6

1. Es gibt $\frac{2^{32}}{2^{12}} = 2^{20}$ Einträge in der Tabelle. Jeder Eintrag ist 4 Byte groß

\Rightarrow ca. 4 MB Größe der Seitentabelle pro Prozess.

2. Bei einer invertierten Seitentabelle gibt es genau einen Eintrag **pro Frame**. Deshalb entspricht das Verhältnis der Tabellengröße zum physischen Speicher exakt dem Verhältnis der Größe eines Eintrags zur Frame- bzw. Seitengröße:

$$\Rightarrow \frac{4 \text{ B}}{4 \text{ KB}} = \frac{1}{1024}$$

5.6 Aufgabe 7

Geg. sei ein System mit einem TLB und einer hierarchischen Seitentabelle mit k Stufen. Die TLB-Trefferquote sei h , die Zugriffszeit auf den TLB sei t_{TLB} , und ein RAM-Zugriff dauere t_{RAM} . Um eine Seite im Speicher zu lesen, muss zunächst die Adresse übersetzt und anschließend auf die eigentlichen Daten zugegriffen werden. Es gibt zwei Fälle:

- Bei einem TLB-Treffer (Wahrscheinlichkeit h) erfolgt die Übersetzung über den TLB, was t_{TLB} dauert, gefolgt von einem Datenzugriff mit t_{RAM} . Gesamtzeit: $t_{\text{TLB}} + t_{\text{RAM}}$
- Bei einem TLB-Fehlzugriff (Wahrscheinlichkeit $1 - h$) muss die Seitentabelle durchlaufen werden, wobei k RAM-Zugriffe nötig sind. Anschließend folgt der Zugriff auf die Daten mit weiteren t_{RAM} . Gesamtzeit: $(k + 1) \cdot t_{\text{RAM}}$

Die erwartete Zugriffszeit ergibt sich zu:

$$E = h(t_{\text{TLB}} + t_{\text{RAM}}) + (1 - h)(k + 1)t_{\text{RAM}}$$

Wenn h klein ist, dominiert der zweite Term, und der Zugriff ist im Mittel etwa $(k + 1)$ -mal so teuer wie bei einem TLB-Treffer.

In der Praxis tritt dieses Problem jedoch kaum auf, da reale Programme ausgeprägte Lokalität aufweisen. Aufgrund **temporaler Lokalität** (wiederholte Zugriffe auf kürzlich genutzte Seiten) und **spatialer Lokalität** (benachbarte Adressen werden gemeinsam genutzt, z.B. in Arrays) ist die TLB-Trefferquote typischerweise sehr hoch (oft über 95 %). Deshalb amortisiert sich die Existenz eines TLB deutlich.

Das zugrunde liegende Modell ist jedoch in mehreren Punkten idealisiert und in der Praxis eingeschränkt:

- Es nimmt **gleichverteilte Zugriffe** auf alle Seitentabelleneinträge an, ignoriert also die reale Zugriffslokalität.
- **Seitentabelleneinträge werden ggf. auch intern gecacht**, was die Zahl tatsächlicher RAM-Zugriffe reduziert.
- Es berücksichtigt keine Nebeneffekte wie **TLB-Flushes bei Kontextwechseln**, **Prefetching**, oder andere Optimierungen der Speicherhierarchie.

Daher ist das Modell gut geeignet für eine theoretische Analyse, aber es bildet die tatsächliche Effizienz realer Systeme nur vereinfacht ab.

5.7 Aufgabe 8

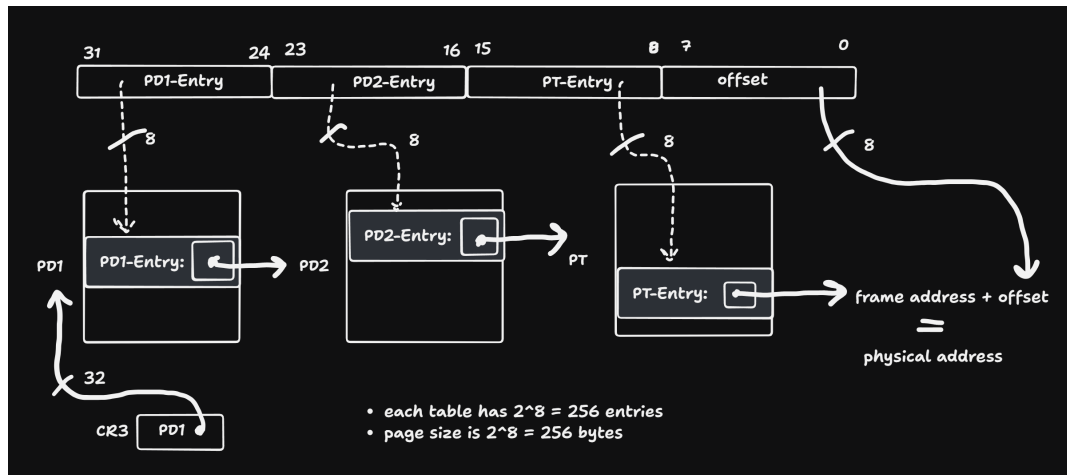


Figure 5.1: 3-level Seitentabelle

a) Das Diagramm zeigt die Übersetzung einer 32-Bit-Adressierung in einem hypothetischen System mit einer dreistufigen Seitentabellenhierarchie und einer Seitengröße von 256 Byte (also 2^8). Die logische Adresse wird dabei in vier gleich große Abschnitte zu je 8 Bit aufgeteilt:

- Bits 31–24: Index in die oberste Tabelle (PD1)
- Bits 23–16: Index in die zweite Tabelle (PD2)
- Bits 15–8: Index in die dritte Tabelle (PT)
- Bits 7–0: Offset innerhalb der Seite

Jede Tabelle hat $2^8 = 256$ Einträge. Dies ergibt sich daraus, dass jeder Tabellenindex 8 Bit umfasst und damit 256 mögliche Positionen adressieren kann. Da alle drei Tabellenebenen mit 8-Bit-Indizes angesprochen werden, besitzen alle drei Stufen genau 256 Einträge. Die Offset-Breite von 8 Bit entspricht der Seitengröße von 256 Byte.

b) Die 32-Bit-Adresse wird in vier Abschnitte zu je 8 Bit unterteilt. Jeder dieser Abschnitte dient als Index in eine bestimmte Stufe der Seitentabellenhierarchie:

- Der erste Abschnitt (Bits 31–24) indexiert einen Eintrag in der obersten Tabelle (PD1).
- Der zweite Abschnitt (Bits 23–16) indexiert die mittlere Tabelle (PD2).
- Der dritte Abschnitt (Bits 15–8) indexiert die unterste Tabelle (PT).
- Der vierte Abschnitt (Bits 7–0) ist der Offset innerhalb der Zielseite.

Jeder Eintrag in den Tabellen enthält eine physische Adresse, die zur nächsten Stufe führt:

- Ein **PD1-Eintrag** enthält die physische Startadresse eines PD2-Tabellenrahmens.
- Ein **PD2-Eintrag** enthält die Adresse eines PT-Tabellenrahmens.
- Ein **PT-Eintrag** enthält die Adresse eines tatsächlichen physischen Seitenrahmens, also einer 256-Byte-Seite im Speicher.

Durch schrittweises Nachschlagen entlang der drei Tabellenstufen wird so der physische Rahmen gefunden, in dem sich die gewünschte Adresse befindet. Der Offset gibt schließlich die genaue Position innerhalb dieser Seite an.

Gibt es signifikante Unterschiede zwischen den Stufen?

Ja, in der Funktion der Stufen:

- Die ersten beiden Stufen (PD1 und PD2) dienen rein der Navigation: Sie verweisen jeweils auf weitere Tabellen.
- Erst die dritte Stufe (PT) enthält den tatsächlichen Verweis auf den physischen Speicherrahmen mit den Daten.
- Auch bei der Interpretation der Einträge kann es Unterschiede geben (z. B. zusätzliche Statusbits oder Flags auf unteren Ebenen), aber im Grundprinzip enthalten alle Einträge physische Adressen von Seitenrahmen — entweder von Tabellen oder von Daten.

c) Jede Tabelle hat maximal $2^8 = 256$ Einträge pro Instanz. Da es sich um eine 3-stufige Hierarchie handelt, ergibt sich im **Extremfall** (voll belegter Adressraum):

- **Stufe 1 (PD1):** 1 Tabelle \times 256 Einträge
- **Stufe 2 (PD2):** 256 Tabellen \times 256 Einträge = 65 536
- **Stufe 3 (PT):** 256 \times 256 Tabellen \times 256 Einträge = 16 777 216

Kumulative Maximalanzahl:

$$256 + 65\,536 + 16\,777\,216 = 16\,843\,008 \text{ Einträge}$$

Speicherverbrauch bei 4 Byte pro Eintrag:

$$16\,843\,008 \times 4 \text{ Bytes} = 67\,372\,032 \text{ Bytes} \approx 64,25 \text{ MiB}$$

6 Blatt 06

6.1 Aufgabe 1

a) :

FIFO (First-In, First-Out)

| Step | Accessed Page | Frame State | FIFO Queue | Action |
|------|---------------|--------------|--------------|------------------------------|
| 1 | 0 | [0] | [0] | Page fault (load 0) |
| 2 | 1 | [0, 1] | [0, 1] | Page fault (load 1) |
| 3 | 7 | [0, 1, 7] | [0, 1, 7] | Page fault (load 7) |
| 4 | 2 | [0, 1, 7, 2] | [0, 1, 7, 2] | Page fault (load 2) |
| 5 | 3 | [3, 1, 7, 2] | [1, 7, 2, 3] | Page fault (evict 0, load 3) |
| 6 | 2 | [3, 1, 7, 2] | [1, 7, 2, 3] | No fault |
| 7 | 7 | [3, 1, 7, 2] | [1, 7, 2, 3] | No fault |
| 8 | 1 | [3, 1, 7, 2] | [1, 7, 2, 3] | No fault |
| 9 | 0 | [3, 0, 7, 2] | [7, 2, 3, 0] | Page fault (evict 1, load 0) |
| 10 | 3 | [3, 0, 7, 2] | [7, 2, 3, 0] | No fault |

Erklärung: Der FIFO-Algorithmus verwaltet Seiten nach dem Prinzip „First-In, First-Out“. Wenn der Speicher voll ist, wird stets die älteste Seite (ganz vorne in der Queue) entfernt. Hier entstehen **6 Seitenfehler** bei den Zugriffen: 1, 2, 3, 4, 5, 9.

LRU (Least Recently Used):

| Step | Accessed Page | Frame State | Action |
|------|---------------|--------------|------------------------------|
| 1 | 0 | [0] | Page fault (load 0) |
| 2 | 1 | [0, 1] | Page fault (load 1) |
| 3 | 7 | [0, 1, 7] | Page fault (load 7) |
| 4 | 2 | [0, 1, 7, 2] | Page fault (load 2) |
| 5 | 3 | [3, 1, 7, 2] | Page fault (evict 0, load 3) |
| 6 | 2 | [3, 1, 7, 2] | No fault |
| 7 | 7 | [3, 1, 7, 2] | No fault |
| 8 | 1 | [3, 1, 7, 2] | No fault |

| Step | Accessed Page | Frame State | Action |
|------|---------------|--------------|------------------------------|
| 9 | 0 | [0, 1, 7, 2] | Page fault (evict 3, load 0) |
| 10 | 3 | [0, 1, 7, 3] | Page fault (evict 2, load 3) |

Erklärung: Der LRU-Algorithmus entfernt immer die Seite, die am längsten **nicht verwendet** wurde. Er berücksichtigt dabei die Reihenfolge der letzten Zugriffe. In diesem Beispiel entstehen **7 Seitenfehler** bei den Zugriffen: 1, 2, 3, 4, 5, 9, 10.

b) :

Vergleich: LRU vs. Clock – Erstes unterschiedliches Opfer

Wir suchen eine möglichst kurze Zugriffssequenz, bei der LRU und Clock beim ersten Page-Fault mit Ersetzung unterschiedliche Seiten auslagern. Dies zeigt konkret, wie Clock als Annäherung an LRU funktioniert, aber nicht exakt gleich entscheidet.

Rahmenbedingungen:

- Anzahl Frames: 3
- Zugriffssequenz: 1, 2, 3, 1, 4
- Seitenzahlen stammen aus {1, ..., 9}

LRU (Least Recently Used)

LRU wählt die Seite aus, die am längsten nicht mehr verwendet wurde.

| Step | Accessed Page | Frame State (after access) | Action |
|------|---------------|----------------------------|-------------------------|
| 1 | 1 | [1] | Page fault |
| 2 | 2 | [1, 2] | Page fault |
| 3 | 3 | [1, 2, 3] | Page fault |
| 4 | 1 | [1, 2, 3] | No fault (update usage) |
| 5 | 4 | [1, 4, 3] | Page fault → evict 2 |

Erklärung: LRU entfernt Seite 2, da sie seit Schritt 2 nicht mehr verwendet wurde.

Clock (Second-Chance)

Clock gibt Seiten mit gesetztem R-Bit eine „zweite Chance“. Der Zeiger dreht sich zirkulär durch die Frames.

- Neue Seiten: $R = 1$
- Zugriff: $R \leftarrow 1$
- Beim Ersetzen:

- Wenn $R = 1$: $R \leftarrow 0$, weiter
- Wenn $R = 0$: auslagern

Anfangszustand vor Schritt 5:

- Frame: [1, 2, 3]
- R bits: [1, 1, 1]
- Pointer: $\rightarrow 3$ (zeigt auf Seite 3)

| Step | Accessed Page | Frame State (after access) | R Bits | Pointer | Action |
|------|---------------|----------------------------|-----------|-----------------|----------------------------------|
| 1 | 1 | [1] | [1] | $\rightarrow 1$ | Page fault |
| 2 | 2 | [1, 2] | [1, 1] | $\rightarrow 2$ | Page fault |
| 3 | 3 | [1, 2, 3] | [1, 1, 1] | $\rightarrow 3$ | Page fault |
| 4 | 1 | [1, 2, 3] | [1, 1, 1] | $\rightarrow 3$ | No fault |
| 5 | 4 | [1, 2, 4] | [0, 0, 1] | $\rightarrow 1$ | Page fault \rightarrow evict 3 |

Erklärung: Clock entfernt Seite 3, da beim Durchlauf alle R-Bits auf 1 gesetzt waren und beim zweiten Umlauf zuerst Seite 3 mit $R = 0$ erreicht wurde.

Vergleich

| Algorithmus | Erste ersetzte Seite |
|-------------|----------------------|
| LRU | 2 |
| Clock | 3 |

Diese kurze Sequenz zeigt präzise, wie Clock trotz ähnlicher Zielsetzung (ältere Seiten auslagern) durch seine heuristische Umsetzung (R-Bits und Zeiger) zu anderen Entscheidungen als LRU kommt.

6.2 Aufgabe 2

Obwohl LRU – im Gegensatz zu OPT – die Zukunft nicht kennt, kann es mit exakten Zeitstempeln vergangener Seitenzugriffe fundierte Entscheidungen treffen. OPT hingegen nutzt zukünftige Zugriffe zur Minimierung der Seitenfehler und stellt damit die theoretisch beste Strategie dar. Zwischen beiden Algorithmen besteht ein enger mathematischer Zusammenhang: **Die Anzahl der Seitenfehler von LRU ist im schlimmsten Fall höchstens k -mal so groß wie die von OPT**, wobei k die Anzahl der verfügbaren Seitenrahmen ist. Dieser Zusammenhang stammt aus der kompetitiven Analyse von Online-Algorithmen und

ist zwar theoretisch interessant, jedoch **nicht praktisch nutzbar**, da er keine konkreten Verbesserungen im realen Betrieb ermöglicht.

6.3 Aufgabe 3

Im folgenden Python-programm haben wir den **LRU-Algorithmus** implementiert (auch im Zip als 'lru_sim.py' enthalten):

```
import sys

def print_frame_state(step, page, frames, action):
    print(f"Step {step:2}: Page {page:2} → {action:6} | Frames: {frames}")

def simulate_lru(num_frames, access_sequence):
    frames = []
    lru_order = [] # Tracks least to most recently used pages

    for step, page in enumerate(access_sequence, 1):
        if page in frames:
            action = "Hit"
            # Move page to most recently used
            lru_order.remove(page)
            lru_order.append(page)
        else:
            action = "Fault"
            if len(frames) < num_frames:
                # Free space available
                frames.append(page)
            else:
                # Evict least recently used page
                lru_page = lru_order.pop(0)
                index = frames.index(lru_page)
                frames[index] = page
                lru_order.append(page)
            print_frame_state(step, page, frames.copy(), action)

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: python lru_sim.py <num_frames> <page1> <page2> ...")
        sys.exit(1)
```



```

try:
    num_frames = int(sys.argv[1])
    access_sequence = list(map(int, sys.argv[2:]))
except ValueError:
    print("Error: All inputs must be integers.")
    sys.exit(1)

simulate_lru(num_frames, access_sequence)

```

Algorithmusprinzip (LRU)

Beim LRU-Verfahren wird stets die **am längsten nicht benutzte Seite** entfernt, wenn ein neuer Seitenzugriff erfolgt und kein freier Rahmen mehr verfügbar ist. Das Verfahren benötigt daher eine Struktur, um die **Zugriffshistorie** zu verfolgen.

Umsetzung in Python:

Für die Python-Implementierung wurden folgende Datenstrukturen verwendet:

- **frames**: Eine Liste, die den aktuellen Inhalt der Seitenrahmen repräsentiert.
- **lru_order**: Eine separate Liste, die die Zugriffsreihenfolge der Seiten hält – von **ältestem** zu **jüngstem** Zugriff.

Ablauf:

- Bei einem **Treffer (Hit)** wird die Seite in **lru_order** nach hinten verschoben (neuester Zugriff).
- Bei einem **Seitenfehler (Fault)**:
 - Wenn Platz frei ist → Seite wird einfach geladen.
 - Wenn kein Platz mehr frei ist → Die Seite ganz vorne in **lru_order** wird entfernt (am längsten unbenutzt) und im Rahmen ersetzt.

Beispielaufruf:

```
python lru_sim.py 3 7 0 1 2 0 3 0 4 2 3
```

Der erste Input ist die Anzahl der Rahmen und die restlichen Zahlen stellen die Referenzfolge dar.

Hinweis:

Die Implementierung nutzt bewusst nur grundlegende Datenstrukturen (**list**), um die LRU-Logik transparent und nachvollziehbar zu gestalten. Für größere Datenmengen könnten effizientere Strukturen wie **collections.OrderedDict** verwendet werden.

Ausgabeblogs für Referenzfolgen A und B:

- A:

```
A="7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1"
python lru_sim.py 3 $A
```

output (auch im ZIP als A-log.txt enthalten):

```
Step 1: Page 7 → Fault | Frames: [7]
Step 2: Page 0 → Fault | Frames: [7, 0]
Step 3: Page 1 → Fault | Frames: [7, 0, 1]
Step 4: Page 2 → Fault | Frames: [2, 0, 1]
Step 5: Page 0 → Hit   | Frames: [2, 0, 1]
Step 6: Page 3 → Fault | Frames: [2, 0, 3]
Step 7: Page 0 → Hit   | Frames: [2, 0, 3]
Step 8: Page 4 → Fault | Frames: [4, 0, 3]
Step 9: Page 2 → Fault | Frames: [4, 0, 2]
Step 10: Page 3 → Fault | Frames: [4, 3, 2]
Step 11: Page 0 → Fault | Frames: [0, 3, 2]
Step 12: Page 3 → Hit   | Frames: [0, 3, 2]
Step 13: Page 2 → Hit   | Frames: [0, 3, 2]
Step 14: Page 1 → Fault | Frames: [1, 3, 2]
Step 15: Page 2 → Hit   | Frames: [1, 3, 2]
Step 16: Page 0 → Fault | Frames: [1, 0, 2]
Step 17: Page 1 → Hit   | Frames: [1, 0, 2]
Step 18: Page 7 → Fault | Frames: [1, 0, 7]
Step 19: Page 0 → Hit   | Frames: [1, 0, 7]
Step 20: Page 1 → Hit   | Frames: [1, 0, 7]
```

- B:

```
B="2 3 2 1 5 2 4 5 3 2 5 2"
python lru_sim.py 3 $B
```

output (auch im ZIP als B-log.txt enthalten):

```
Step 1: Page 2 → Fault | Frames: [2]
Step 2: Page 3 → Fault | Frames: [2, 3]
Step 3: Page 2 → Hit   | Frames: [2, 3]
Step 4: Page 1 → Fault | Frames: [2, 3, 1]
Step 5: Page 5 → Fault | Frames: [2, 5, 1]
Step 6: Page 2 → Hit   | Frames: [2, 5, 1]
Step 7: Page 4 → Fault | Frames: [2, 5, 4]
Step 8: Page 5 → Hit   | Frames: [2, 5, 4]
Step 9: Page 3 → Fault | Frames: [3, 5, 4]
```

| | | |
|---------------|-----------|-------------------|
| Step 10: Page | 2 → Fault | Frames: [3, 5, 2] |
| Step 11: Page | 5 → Hit | Frames: [3, 5, 2] |
| Step 12: Page | 2 → Hit | Frames: [3, 5, 2] |

6.4 Aufgabe 4

- a) Im Worst Case sind alle 20 Prozesse aktiv und belegen jeweils den gesamten virtuellen Adressraum des IA32-Systems. Bei einer Seitengröße von 4 KiB ergibt sich für jeden Prozess eine Seitentabelle mit

$$\frac{2^{32}}{2^{12}} = 2^{20} = 1.048.576 \text{ Seiten}$$

Insgesamt müssen also

$$20 \times 2^{20} = 20.971.520 \text{ Seiteneinträge}$$

betrachtet werden. Da laut Aufgabenstellung das Lesen und Zurücksetzen des R-Bits pro Eintrag im Mittel 10 Taktzyklen benötigt, ergibt sich ein Gesamtaufwand von

$$20.971.520 \times 10 = 209.715.200 \text{ Taktzyklen}$$

Bei einem Prozessor mit 1 GHz Taktfrequenz entspricht das einer Zeitdauer von

$$\frac{209.715.200}{1.000.000.000} = 0,2097 \text{ Sekunden} \approx 210 \text{ ms}$$

Im Worst Case benötigt das System also rund **210 ms pro Epoche**, um alle R-Bits der Seitentabellen zu überprüfen und zurückzusetzen.

- b) Damit das regelmäßige Scannen der R-Bits die Gesamtleistung des Systems nicht spürbar beeinträchtigt, sollte die Epochendauer so gewählt werden, dass der dabei entstehende Rechenaufwand nur einen kleinen Teil der Zeit ausmacht. Eine sinnvolle Faustregel ist, dass **der Verwaltungsaufwand maximal etwa 10 % der gesamten Rechenzeit** betragen sollte.

Wenn das Scannen der R-Bits im Worst Case rund 210 ms dauert, ergibt sich daraus eine minimale sinnvolle Epochendauer von:

$$\frac{210 \text{ ms}}{0,1} = 2100 \text{ ms}$$

Eine Epoche von etwa **2 Sekunden** ist somit eine sinnvolle Wahl. Dadurch bleibt der relative Aufwand für die Speicherverwaltung auch im ungünstigsten Fall moderat. In der Praxis würde dieser Aufwand meist noch deutlich geringer ausfallen, da typischerweise weniger Prozesse aktiv sind und nicht alle Seitentabellen vollständig gefüllt sind.

6.5 Aufgabe 5

Ja, eine Seite kann gleichzeitig zu mehreren Working Sets gehören, allerdings nur, wenn es sich um eine gemeinsam genutzte Seite handelt (zum Beispiel bei Shared Memory oder gemappten Dateien) und sie von jedem der betreffenden Prozesse kürzlich verwendet wurde. Da das Working Set prozessbezogen ist, enthält es nur Seiten, die der jeweilige Prozess innerhalb des festgelegten Zeitfensters selbst genutzt hat. Wird eine geteilte Seite von mehreren Prozessen aktiv verwendet, so gehört sie gleichzeitig zu den Working Sets dieser Prozesse.

6.6 Aufgabe 6

Problemstellung

Gegeben ist:

- eine Referenzfolge von Seitenzugriffen (z. B. als Datei gespeichert),
- eine natürliche Zahl Δ (Delta), die die Größe des betrachteten Fensters angibt.

Gesucht ist:

Ein größtes Working Set, d. h. eine Menge von Seiten, die in einem **beliebigen** Fenster der letzten Δ Speicherzugriffe aufgetreten sind, wobei das Fenster über die gesamte Referenzfolge verschoben wird.

Das Ziel ist also nicht das Working Set **am Ende** der Referenzfolge, sondern das Working Set, das an **irgendeiner Stelle** die **größte Anzahl unterschiedlicher Seiten** enthält.

Pseudocode:

```
funktion berechne_groesstes_working_set(delta, dateiname):  
    initialisiere leere Liste window  
    initialisiere leere Hashtabelle page_count  
    initialisiere leere Menge current_ws  
    initialisiere leere Menge max_ws_snapshot  
  
    lese Inhalt der Datei mit Dateiname in ein Array pageAccesses ein
```

```

für jede Seite page in pageAccesses:
    // Seite zum Fenster hinzufügen
    window.append(page)
    falls page nicht in page_count:
        page_count[page] = 1
        current_ws.add(page)
    sonst:
        page_count[page] += 1

// Wenn das Fenster zu groß wird, älteste Seite entfernen
falls window.größe > delta:
    oldest = window.pop_links()
    page_count[oldest] -= 1
    falls page_count[oldest] == 0:
        entferne page_count[oldest]
        current_ws.remove(oldest)

// Maximales Working Set ggf. aktualisieren
falls current_ws.größe > max_ws_snapshot.größe:
    max_ws_snapshot = kopie_von(current_ws)

gib max_ws_snapshot aus

```

Verwendete Datenstrukturen und Variablen:

| Name | Typ | Beschreibung |
|-----------------|------------------------------|---|
| window | Warteschlange / Liste | Enthält die letzten Seitenzugriffe |
| page_count | Hashtabelle (Seite → Anzahl) | Speichert, wie oft jede Seite im aktuellen Fenster vorkommt |
| current_ws | Menge (Set) | Enthält die aktuell im Fenster vorhandenen unterschiedlichen Seiten |
| max_ws_snapshot | Menge (Set) | Enthält ein Working Set mit der maximalen bisher gefundenen Größe |

Beispielausführung

Eingabeparameter:

- = 5
- Referenzfolge (z. B. in Datei gespeichert):

1
2
3
4
2
1
5
2
6
7
2
3

Schritt-für-Schritt-Auswertung:

Wir verschieben ein Fenster der Größe 5 über die Folge und bestimmen dabei das Working Set (WS), also die Menge der **verschiedenen** Seiten im aktuellen Fenster.

| Fenster (Position) | Fensterinhalt | Aktuelles WS | Größe | Größtes WS bisher |
|--------------------|-----------------|-----------------|-------|------------------------|
| 1–5 | [1, 2, 3, 4, 2] | {1, 2, 3, 4} | 4 | {1, 2, 3, 4} |
| 2–6 | [2, 3, 4, 2, 1] | {1, 2, 3, 4} | 4 | — |
| 3–7 | [3, 4, 2, 1, 5] | {1, 2, 3, 4, 5} | 5 | {1, 2, 3, 4, 5} |
| 4–8 | [4, 2, 1, 5, 2] | {1, 2, 4, 5} | 4 | — |
| 5–9 | [2, 1, 5, 2, 6] | {1, 2, 5, 6} | 4 | — |
| 6–10 | [1, 5, 2, 6, 7] | {1, 2, 5, 6, 7} | 5 | — (ebenfalls max.) |
| 7–11 | [5, 2, 6, 7, 2] | {2, 5, 6, 7} | 4 | — |
| 8–12 | [2, 6, 7, 2, 3] | {2, 3, 6, 7} | 4 | — |

Ausgabe des Programms:

Das Programm gibt eines der größten Working Sets aus, z. B.:

{1, 2, 3, 4, 5}

Größe = 5

Das entspricht der ersten Stelle im Verlauf, an der ein Fenster mit 5 verschiedenen Seiten auftritt.

7 Blatt 07

7.1 Aufgabe 1

- **Vorteil von Hard Links:** Bleiben gültig, auch wenn die Originaldatei gelöscht wird, da sie direkt auf die Inode verweisen.
- **Vorteil von Soft Links:** Können Dateien über verschiedene Dateisysteme hinweg verlinken, da sie den Pfad statt der Inode speichern.

7.2 Aufgabe 2

7.3 Aufgabe 3

a) Analogien:

- **logische Seiten (logische Adressen) Datei-Block-Offsets (logische Datei-Adressen)** : Eine logische Adresse besteht aus einer Seitennummer und einem Offset innerhalb der Seite; ebenso wird eine Datei über Blocknummer und Offset adressiert. Beide sind abstrakte Adressen, die auf reale Speicherorte abgebildet werden.
- **physische Frames (physische Adressen) Festplattenblöcke** : Ein Frame im RAM und ein Block auf der Festplatte sind physische Einheiten fester Größe (z. B. 4KB), in denen tatsächliche Daten gespeichert werden.
- **Frame-Nummer Blocknummer** : Die Seitentabelle enthält Frame-Nummern (Index im physischen RAM); der i-node enthält Blocknummern (Index auf der Festplatte). Beide zeigen, wo genau sich die Daten physisch befinden.
- **Seitentabelle i-node** : Die Seitentabelle eines Prozesses ordnet virtuelle Seiten physischen Frames zu; der i-node einer Datei ordnet logische Blöcke physischen Festplattenblöcken zu. Beide sind zentrale Strukturen für die Adressübersetzung.

- b) Das Problem der übergroßen Seitentabellen entspricht bei I-Nodes dem Problem, dass große Dateien nicht allein mit direkten Blockzeigern adressiert werden können, da die I-Nodes sonst zu groß würden. Die Lösung ist in beiden Fällen ähnlich: Hierarchische Seitentabellen in der Speicherverwaltung entsprechen ein- oder mehrstufigen indirekten Blöcken bei I-Nodes. Welche Form der „Seitentabelle“ (direkt oder indirekt) verwendet wird, hängt bei I-Nodes von der Dateigröße ab – kleine Dateien nutzen nur direkte Zeiger, große benötigen zusätzliche Indirektion.

7.4 Aufgabe 4

Jeder Zeiger adressiert Blöcke der Größe 1024 Byte, und jede Blockadresse belegt 4 Byte.

Die 14 direkten Zeiger verweisen direkt auf Datenblöcke und ermöglichen somit den Zugriff auf $14 \times 1024 = 14.336$ Byte.

Jeder der beiden indirekten Zeiger zeigt auf einen weiteren Block, der ausschließlich Blockadressen enthält. Da ein Block 1024 Byte groß ist und jede Adresse 4 Byte belegt, passen $1024 / 4 = 256$ Adressen in einen solchen Block. Diese 256 Adressen verweisen jeweils auf Datenblöcke zu je 1024 Byte, sodass pro indirektem Zeiger $256 \times 1024 = 262.144$ Byte adressiert werden können. Zwei indirekte Zeiger ergeben somit $2 \times 262.144 = 524.288$ Byte.

Die maximale Dateigröße beträgt daher $14.336 + 524.288 = \mathbf{538.624 \text{ Byte}}$ $\approx 0.5 \text{ MB}$.

7.5 Aufgabe 5

Zur Auflösung des Pfades `/usr/aa/lehre/ibn/ueb7-ibn-2017.pdf` müssen nacheinander die Verzeichniseinträge gelesen und die entsprechenden I-Nodes geladen werden. Der Pfad besteht aus fünf Komponenten: `usr`, `aa`, `lehre`, `ibn` und `ueb7-ibn-2017.pdf`.

Da der I-Node des Wurzelverzeichnisses bereits im Hauptspeicher liegt, entfällt dafür eine Operation. Für jede der übrigen Komponenten müssen jedoch:

1. Der Plattenblock des Verzeichnisses gelesen werden (um den Dateinamen \rightarrow I-Node-Nummer zu finden),
2. Der entsprechende I-Node geladen werden (um zum nächsten Element zu gelangen).

Somit fallen für jede der 5 Komponenten **2 Plattenoperationen** an: einmal Lesen des Verzeichnisblocks, einmal Lesen des I-Nodes.

\Rightarrow Es werden insgesamt $5 \times 2 = 10$ Plattenoperationen benötigt.

7.6 Aufgabe 6

- a) Die Drehzahl beträgt 1500 Umdrehungen pro Minute, also dauert eine vollständige Umdrehung $60 \text{ s} / 1500 = 0,04 \text{ s} = 40 \text{ ms}$. Da die mittlere Drehlatenz einer halben Umdrehung entspricht, ergibt sich $40 \text{ ms} / 2 = \mathbf{20 \text{ ms}}$. Die mittlere Drehlatenz beträgt somit **20 Millisekunden**.

- b) Um diese Aufgabe zu lösen, gilt: Die durchschnittliche Zugriffszeit ergibt sich aus der Summe von mittlerer Spurwechselzeit und mittlerer Drehlatenz.

Gegeben ist eine mittlere Spurwechselzeit von 85 ms und eine mittlere Drehlatenz von 20 ms. Damit ergibt sich eine durchschnittliche Zugriffszeit von $85 \text{ ms} + 20 \text{ ms} = 105 \text{ ms}$.

Für den Zugriff auf 10 unabhängig und zufällig ausgewählte Datenblöcke ergibt sich eine mittlere Gesamtdauer von $10 \times 105 \text{ ms} = 1050 \text{ ms} = 1,05 \text{ s}$.

- c) Gegeben ist eine mittlere Drehlatenz von 8,333 ms. Da dies der Hälfte einer Umdrehung entspricht, dauert eine vollständige Umdrehung $2 \times 8,333 \text{ ms} = 16,666 \text{ ms}$.

Die Anzahl der Umdrehungen pro Sekunde beträgt daher $1 \text{ s} / 0,016666 \text{ s} = 60$. Das entspricht 60 Umdrehungen pro Sekunde bzw. $60 \times 60 = 3600 \text{ U/min}$.

Die Platten drehen sich also mit 3600 Umdrehungen pro Minute.

7.7 Aufgabe 7

Wir haben:

- **Zylinderbereich:** 0 bis 4999
- **Startposition:** Kopf ist aktuell bei **143**, kam gerade von **125** (d.h. aktuelle Bewegungsrichtung ist **aufwärts**)
- **Anfragen in Ankunftsreihenfolge:** 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Wir berechnen nun für jeden Algorithmus:

1. **Reihenfolge der bedienten Zylinder**
2. **Gesamte Bewegung des Kopfes (in Zylindern)**

FCFS (First-Come, First-Serve)

Verarbeitung in Ankunftsreihenfolge, egal wie weit entfernt:

- Start bei 143
- Folge: 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Berechnung der Bewegungen:

- $|143 - 86| = 57$
- $|86 - 1470| = 1384$
- $|1470 - 913| = 557$
- $|913 - 1774| = 861$
- $|1774 - 948| = 826$
- $|948 - 1509| = 561$
- $|1509 - 1022| = 487$
- $|1022 - 1750| = 728$
- $|1750 - 130| = 1620$

Summe: $57 + 1384 + 557 + 861 + 826 + 561 + 487 + 728 + 1620 = 7081$ Zylinder

SSTF (Shortest Seek Time First)

Hier wird immer die **nächstgelegene** Anfrage (bezogen auf aktuelle Kopfposition) bedient.

- **Startposition:** 143
- **Offene Anfragen:** 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Wir wählen immer den **nächstgelegenen Zylinder** zur aktuellen Position. Schritte:

1. Aktuell bei 143 Nächster: 130 ($|143-130| = 13$)
→ neue Position: 130
Übrig: 86, 1470, 913, 1774, 948, 1509, 1022, 1750
2. Aktuell bei 130
Nächster: 86 ($|130-86| = 44$)
→ neue Position: 86
Übrig: 1470, 913, 1774, 948, 1509, 1022, 1750
3. Aktuell bei 86
Nächster: 913 ($|86-913| = 827$)
→ neue Position: 913
Übrig: 1470, 1774, 948, 1509, 1022, 1750

4. Aktuell bei 913
 Nächster: 948 ($|913-948| = 35$)
 → neue Position: 948
 Übrig: 1470, 1774, 1509, 1022, 1750
5. Aktuell bei 948
 Nächster: 1022 ($|948-1022| = 74$)
 → neue Position: 1022
 Übrig: 1470, 1774, 1509, 1750
6. Aktuell bei 1022
 Nächster: 1470 ($|1022-1470| = 448$)
 → neue Position: 1470
 Übrig: 1774, 1509, 1750
7. Aktuell bei 1470
 Nächster: 1509 ($|1470-1509| = 39$)
 → neue Position: 1509
 Übrig: 1774, 1750
8. Aktuell bei 1509
 Nächster: 1750 ($|1509-1750| = 241$)
 → neue Position: 1750
 Übrig: 1774
9. Aktuell bei 1750
 Nächster: 1774 ($|1750-1774| = 24$)
 → neue Position: 1774
 Übrig: —

Reihenfolge: 143 → 130 → 86 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774

Bewegungen:

- $|143-130| = 13$
- $|130-86| = 44$
- $|86-913| = 827$
- $|913-948| = 35$
- $|948-1022| = 74$
- $|1022-1470| = 448$
- $|1470-1509| = 39$
- $|1509-1750| = 241$
- $|1750-1774| = 24$

Summe: $13 + 44 + 827 + 35 + 74 + 448 + 39 + 241 + 24 = 1745$ Zylinder

SCAN (Fahrstuhl-Algorithmus)

Der Lesekopf bewegt sich in eine Richtung (hier: **aufwärts**, weil er von 125 nach 143 kam), bedient dabei alle Anfragen auf dem Weg, und kehrt dann am Ende um.

- **Startposition:** 143
- **Bewegungsrichtung:** aufwärts
- **Anfragen:** 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Wir teilen in zwei Gruppen:

- Aufwärts (≥ 143): 1470, 913, 1774, 948, 1509, 1022
- Abwärts (< 143): 86, 130

Zuerst behandeln wir alle Anfragen in aufsteigender Reihenfolge 143: \rightarrow 913, 948, 1022, 1470, 1509, 1774 Dann kehrt der Kopf um und bedient absteigend: \rightarrow 130, 86

Reihenfolge des Besuchs: 143 \rightarrow 913 \rightarrow 948 \rightarrow 1022 \rightarrow 1470 \rightarrow 1509 \rightarrow 1774 \rightarrow 130 \rightarrow 86

Bewegungen:

- $|143-913| = 770$
- $|913-948| = 35$
- $|948-1022| = 74$
- $|1022-1470| = 448$
- $|1470-1509| = 39$
- $|1509-1774| = 265$
- $|1774-130| = 1644$
- $|130-86| = 44$

Summe: $770 + 35 + 74 + 448 + 39 + 265 + 1644 + 44 = 3319$ Zylinder

C-SCAN (Circular SCAN)

Der Kopf bewegt sich in **eine Richtung** (hier: aufwärts) und bedient alle Anfragen auf dem Weg. Am Ende (bei höchstem Zylinder) springt der Kopf **ohne Bedienung** zurück zum Anfang und beginnt erneut.

- **Startposition:** 143
- **Bewegungsrichtung:** aufwärts
- **Anfragen:** 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Anfragen 143 (in aufsteigender Reihenfolge): $\rightarrow 913, 948, 1022, 1470, 1509, 1750, 1774$

Anfragen <143 (werden erst nach Rücksprung behandelt, ebenfalls aufsteigend): $\rightarrow 86, 130$

Reihenfolge des Besuchs: $143 \rightarrow 913 \rightarrow 948 \rightarrow 1022 \rightarrow 1470 \rightarrow 1509 \rightarrow 1750 \rightarrow 1774 \rightarrow$
Sprung zu $0 \rightarrow 86 \rightarrow 130$

Bewegungen:

- $|143-913| = 770$
- $|913-948| = 35$
- $|948-1022| = 74$
- $|1022-1470| = 448$
- $|1470-1509| = 39$
- $|1509-1750| = 241$
- $|1750-1774| = 24$
- $|1774-0| = 1774$ (Sprung ohne Bedienung)
- $|0-86| = 86$
- $|86-130| = 44$

Summe: $770 + 35 + 74 + 448 + 39 + 241 + 24 + 1774 + 86 + 44 = 3535$ Zylinder

C-LOOK (Circular LOOK)

- **Startposition:** 143
- **Bewegungsrichtung:** aufwärts
- **Anfragen:** 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Zuerst alle Anfragen 143 (sortiert): $\rightarrow 913, 948, 1022, 1470, 1509, 1750, 1774$

Dann Sprung zurück zum **niedrigsten angefragten Zylinder** (<143): $\rightarrow 86, 130$ (ebenfalls aufsteigend)

Reihenfolge des Besuchs: $143 \rightarrow 913 \rightarrow 948 \rightarrow 1022 \rightarrow 1470 \rightarrow 1509 \rightarrow 1750 \rightarrow 1774 \rightarrow$
 $86 \rightarrow 130$

Bewegungen:

- $|143-913| = 770$
- $|913-948| = 35$
- $|948-1022| = 74$
- $|1022-1470| = 448$
- $|1470-1509| = 39$
- $|1509-1750| = 241$
- $|1750-1774| = 24$
- $|1774-86| = 1688$ (Sprung zurück)

- $|86-130| = 44$

Summe: $770 + 35 + 74 + 448 + 39 + 241 + 24 + 1688 + 44 = \mathbf{3363 \text{ Zylinder}}$

Vergleichstabelle: Disk Scheduling

| Algorithmus | Reihenfolge der besuchten Zylinder | Gesamtdistanz (Zylinder) |
|---------------|---|-----------------------------|
| FCFS | 143 → 86 → 1470 → 913 → 1774 → 948 → 1509 → 1022 → 1750 → 130 | 7081 |
| SSTF | 143 → 130 → 86 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 | 1897 |
| SCAN | 143 → 913 → 948 → 1022 → 1470 → 1509 → 1774 → 130 → 86 | 3319 |
| C-SCAN | 143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 0 → 86 → 130 | 3535 |
| C-LOOK | 143 → 913 → 948 → 1022 → 1470 → 1509 → 1750 → 1774 → 86 → 130 | 3363 |

7.8 Aufgabe 8

Ein RAID 0-System verteilt die Daten ohne Redundanz auf zwei Platten. Fällt auch nur eine Platte aus, ist das gesamte System unbrauchbar. Die Wahrscheinlichkeit, dass **beide Platten überleben**, beträgt $0,9 \times 0,9 = 0,81$. Daraus ergibt sich eine **Ausfallwahrscheinlichkeit von $1 - 0,81 = 0,19$, also 19 %**.

Ein RAID 1-System spiegelt die Daten auf zwei Platten. Es bleibt funktionsfähig, solange **mindestens eine** Platte intakt ist. Nur wenn beide gleichzeitig ausfallen ($0,1 \times 0,1 = 0,01$), kommt es zum Datenverlust. Die **Ausfallwahrscheinlichkeit liegt also bei 1 %**.

Eine weitere Reduktion der Ausfallwahrscheinlichkeit ist **nur bei RAID 1** sinnvoll möglich. Dies gelingt z. B. durch den Einsatz von mehr als zwei Festplatten: Bei drei Platten beträgt die Wahrscheinlichkeit, dass alle gleichzeitig ausfallen, nur $0,1^3 = 0,001$. Auch sogenannte **Hot-Spare-Platten** können automatisch einspringen, wenn eine Platte ausfällt.

Bei RAID 0 hingegen erhöht **jede zusätzliche Platte** sogar das Risiko, da das System schon beim Ausfall einer einzigen Platte versagt. Eine Verbesserung der Sicherheit ist daher hier nicht möglich.

7.9 Aufgabe 9

Ja, in bestimmten Szenarien kann ein RAID 1-System eine höhere Leseleistung erreichen als RAID 0. Da die Daten auf mehreren Platten identisch vorliegen, kann der Controller parallele Lesezugriffe auf verschiedene Platten verteilen oder jeweils die am schnellsten zugängliche Platte nutzen.

Dies bringt Vorteile bei vielen gleichzeitigen, verteilten Lesezugriffen – etwa bei Datenbankanfragen oder Webservern. Vorausgesetzt ist, dass der Controller intelligentes Load Balancing unterstützt.

Bei rein sequentiellm Lesen großer Dateien ist RAID 0 meist schneller, da Striping die Datenrate erhöht.

⇒ RAID 1 kann bei paralleler Last schneller sein, RAID 0 eher bei sequenziellen Zugriffen.

8 Blatt 8

8.1 Aufgabe 1

a) **Allgemeiner Beweis: Es kann kein Deadlock auftreten**

Gegeben sind:

- Ein Ressourcentyp mit insgesamt $E = 4$ Instanzen,
- Drei Prozesse P_1, P_2, P_3 ,
- Jeder Prozess kann maximal 2 Instanzen der Ressource anfordern.

Sei C_i die Anzahl der aktuell einem Prozess P_i zugeteilten Ressourceninstanzen, und R_i der verbleibende Bedarf, so dass gilt:

$$C_i + R_i \leq 2 \quad \text{für alle } i = 1, 2, 3$$

und

$$\sum_{i=1}^3 C_i \leq 4$$

Es soll gezeigt werden, dass **unter diesen Bedingungen kein Deadlock entstehen kann**.

Fall 1: Mindestens ein Prozess wartet nicht Falls für ein i gilt: $R_i = 0$, dann kann P_i sofort fertigstellen und bis zu 2 Ressourcen freigeben. Dadurch erhöht sich die Anzahl der verfügbaren Instanzen A , was anderen wartenden Prozessen die Fortsetzung erlaubt. Durch Induktion führt dies zu einer vollständigen Abarbeitung aller Prozesse. Ein Deadlock ist somit ausgeschlossen.

Fall 2: Alle Prozesse warten auf Ressourcen Angenommen, jeder Prozess hält mindestens eine Instanz und fordert eine weitere an, also:

$$C = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad R = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, \quad A = [1]$$

Dies ist die **kritischste zulässige Konfiguration**: Alle 4 Ressourcen sind verteilt (3 belegt, 1 frei), und jeder Prozess wartet auf eine weitere Instanz.

In diesem Fall kann das System dennoch einer Anfrage (z. B. von P_1) nachkommen, sodass dieser Prozess abschließen kann. Danach gibt er seine 2 gehaltenen Ressourcen frei, und A steigt auf 3. Damit können die verbleibenden Prozesse nacheinander ebenfalls fertigstellen. Es kommt also **auch im schlimmsten Fall nicht zu einem Deadlock**.

Schlussfolgerung In allen gültigen Konfigurationen – ob mindestens ein Prozess direkt fertigstellen kann oder alle warten – hat das System stets genug Ressourcen, um zumindest einem Prozess die Beendigung zu ermöglichen. Dadurch ist Fortschritt immer möglich, und eine zirkuläre Wartebedingung kann nie entstehen. Es folgt: **Ein Deadlock ist unter den gegebenen Voraussetzungen ausgeschlossen**.

b) **Aufgabe 1b – Beispiel mit erfüllten Deadlock-Bedingungen, aber ohne Deadlock**

Dieses Beispiel stammt direkt aus den Vorlesungsfolien und illustriert eine Systemkonfiguration, in der **alle vier notwendigen Coffman-Bedingungen für einen Deadlock erfüllt sind**, jedoch **kein Deadlock vorliegt**. Der Bankieralgorithmus bestätigt, dass der Zustand sicher ist.

Systemdefinition

Ressourcentypen: R_1 und R_2 **Gesamtressourcenvektor:** $E = (2, 2)$ **Verfügbarkeitsvektor:** $A = (0, 0)$

Alle Ressourcen sind momentan vollständig vergeben.

Zuteilungsmatrix C :

$$C = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

P_1 hält R_2 , P_2 und P_3 halten jeweils R_1 , P_4 hält R_2 .

Anforderungsmatrix R :

$$R = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

P_1 fordert R_1 , P_3 fordert R_2 .

Überprüfung der Coffman-Bedingungen

1. Wechselseitiger Ausschluss: Ressourcen können nicht gemeinsam genutzt werden. **2. Hold-and-Wait:** Prozesse halten Ressourcen und fordern weitere an (z.B. P_1 hält R_2 , fordert R_1). **3. Keine Präemption:** Ressourcen können Prozessen nicht zwangsweise entzogen werden. **4. Zyklisches Warten:** Es existiert ein Zyklus im Ressourcen-Wartegraphen: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$.

→ Alle vier Bedingungen sind erfüllt.

Anwendung des Bankieralgorithmus

| Schritt | Prozess i | R_i | Verfügbar A | Belegt C_i | Markierte Prozesse (P1, P2, P3, P4) |
|---------|-------------|--------|------------------|--------------|--|
| 1 | 1 | (1, 0) | (0, 0) | (0, 1) | × × × × |
| 2 | 2 | (0, 0) | (0, 0) | (1, 0) | × × × |
| 3 | 3 | (0, 1) | (1, 0) | (1, 0) | × × × |
| 4 | 4 | (0, 0) | (1, 0) | (0, 1) | × × |
| 5 | 1 | (1, 0) | (1, 1) | (0, 1) | × |
| 6 | 3 | (0, 1) | (1, 2) | (1, 0) | |

Nach jedem Schritt wird der Verfügbarkeitsvektor A aktualisiert, sobald ein markierter Prozess seine gehaltenen Ressourcen freigibt. Am Ende sind alle Prozesse markiert → der Zustand ist **sicher**.

Schlussfolgerung

Trotz erfüllter Coffman-Bedingungen und zyklischer Abhängigkeiten liegt **kein Deadlock vor**, da der Bankieralgorithmus eine vollständige Abarbeitungsreihenfolge findet. Dieses Beispiel erfüllt somit exakt die Aufgabenstellung: Ein System mit potenzieller Deadlock-Struktur, das sich jedoch als sicher herausstellt.

8.2 Aufgabe 2

a) Beschreiben Sie ein Szenario, in dem ein Deadlock entstehen kann:

Ein Deadlock kann entstehen, wenn die beiden Prozesse **P1** und **P2** die Ressourcen **A** und **B** in entgegengesetzter Reihenfolge anfordern, und beide jeweils bereits eine Ressource belegt haben, während sie auf die zweite warten.

Ablauf eines möglichen Szenarios:

1. **P1** startet zuerst und führt `wait(sem_A)` aus → Ressource **A** wird gesperrt.
 2. **P2** startet kurz danach und führt `wait(sem_B)` aus → Ressource **B** wird gesperrt.
 3. Nun versucht **P1**, `wait(sem_B)` auszuführen, blockiert aber, da **B** bereits durch **P2** gesperrt ist.
 4. Gleichzeitig versucht **P2**, `wait(sem_A)` auszuführen, blockiert aber, da **A** durch **P1** gehalten wird.
- Beide Prozesse warten nun jeweils auf eine Ressource, die vom jeweils anderen gehalten wird.
 - Keiner kann weitermachen oder Ressourcen freigeben.
 - **Dies ist ein klassischer Deadlock-Zustand.**

Dieses Szenario entspricht genau dem Beispiel auf **Folie 15 der Vorlesung 15**, bei dem zwei Prozesse in dieser Weise mit `wait(A)/wait(B)` agieren. Der Deadlock tritt bei einer ungünstigen Ausführungsreihenfolge auf.

- b) Erklären Sie präzise, welche der notwendigen Bedingungen für einen Deadlock hier erfüllt sind.

In der dargestellten Situation (siehe Teil a) sind **alle vier Coffman-Bedingungen** erfüllt, die gemeinsam notwendig für das Entstehen eines Deadlocks sind:

1. **Wechselseitiger Ausschluss (Mutual Exclusion)** Beide Ressourcen (**A** und **B**) sind durch binäre Semaphore geschützt, d. h. sie können **immer nur von einem Prozess gleichzeitig** verwendet werden.
2. **Hold-and-Wait (Belegen und Warten)** Beide Prozesse belegen eine Ressource (z. B. **P1** hält **A**, **P2** hält **B**) und **fordern gleichzeitig eine weitere Ressource an**, auf die sie warten.
3. **Keine Präemption (No Preemption)** Ressourcen können **nicht gewaltsam entzogen** werden. Ein Prozess gibt eine Ressource nur durch ein `signal()` freiwillig frei — aber das geschieht **erst nach der kritischen Sektion**, also **nicht während des Wartens**.
4. **Zyklische Wartebedingung (Circular Wait)** Es entsteht ein Zyklus im Ressourcen-Wartegraphen:
 - **P1** hält **A** und wartet auf **B**,
 - **P2** hält **B** und wartet auf **A**. → Zyklus: $P1 \rightarrow B \rightarrow P2 \rightarrow A \rightarrow P1$

⇒ Alle vier notwendigen Bedingungen für einen Ressourcen-Deadlock sind in diesem Szenario gleichzeitig gegeben. Das erklärt, **warum hier ein Deadlock möglich ist**, wenn die Prozesse in genau dieser Reihenfolge ausgeführt werden.

- c) Schlagen Sie eine einfache konzeptionelle Änderung vor, die das Risiko eines Deadlocks in diesem Beispiel beseitigt.

Eine einfache, aber **sehr effektive** konzeptionelle Änderung besteht darin, **eine einheitliche Reihenfolge für die Ressourcenzugriffe** festzulegen.

Konkret: Beide Prozesse sollen die Ressourcen **immer in derselben Reihenfolge anfordern** — zum Beispiel:

```
wait(sem_A);  
wait(sem_B);  
// kritische Sektion  
signal(sem_B);  
signal(sem_A);
```

Sowohl **P1** als auch **P2** würden dann **zuerst sem_A und danach sem_B** anfordern.

Durch die feste Anforderungsreihenfolge wird die **zyklische Wartebedingung (circular wait)** **prinzipiell ausgeschlossen**. Denn:

- Wenn ein Prozess bereits **sem_A** belegt, ein anderer Prozess aber auf **sem_B** wartet, wird dieser niemals gleichzeitig **sem_A** anfordern, da er **sem_A** ja **zuerst** anfordern müsste.
- Somit kann kein zyklisches Warten entstehen.

⇒ Die Einführung einer **globalen, konsistenten Reihenfolge** beim Zugriff auf mehrere Ressourcen ist eine einfache, konzeptionelle Methode zur **Vermeidung von Deadlocks**. Sie unterläuft gezielt eine der vier Deadlock-Bedingungen — nämlich die zyklische Wartebedingung.

8.3 Aufgabe 4

- a) RAG:

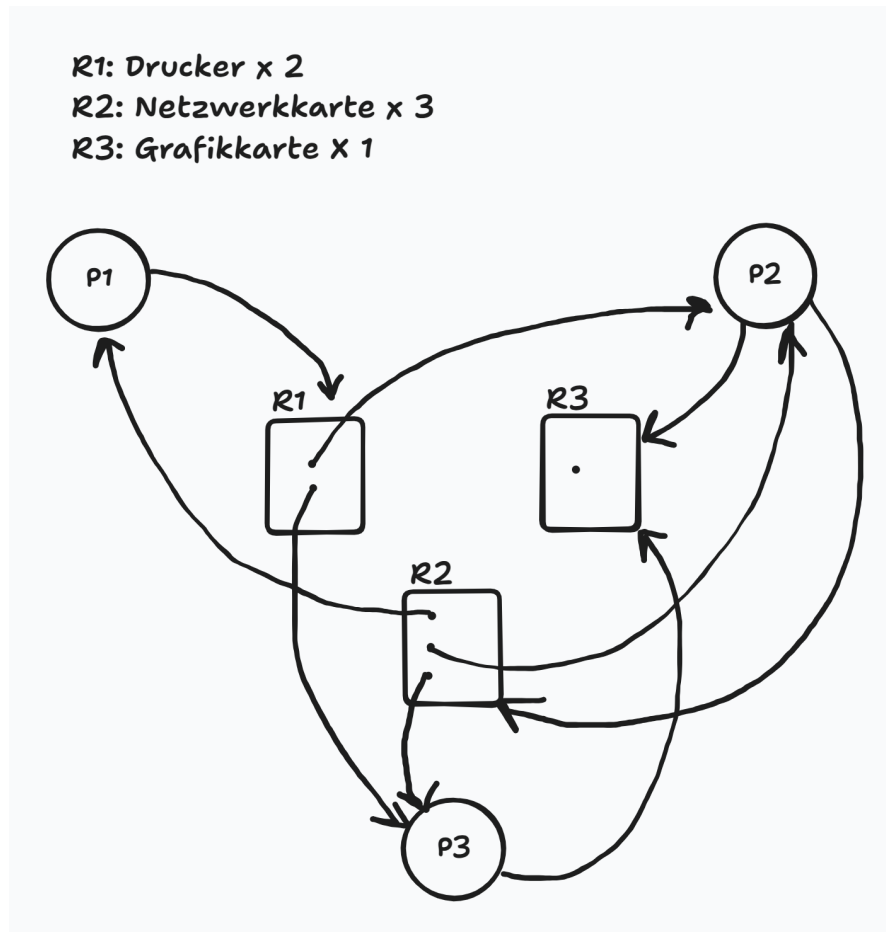


Figure 8.1: RAG

b) E, A, R, und C:

- **Gesamtressourcenvektor:** $E = (2, 3, 1)$
- **Verfügbarkeitsvektor (initial):** $A = (0, 0, 1)$
- **Belegungsmatrix C:**

$$C = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

- **Anforderungsmatrix R:**

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

c) Durchführung des Bankieralgorithmus

Ausführungstabelle des Bankieralgorithmus:

| Schritt | Prozess i | R_i | A | C_i | Anfrage erfüllbar? | Markiert |
|---------|-------------|-----------|-----------|-----------|--------------------|----------|
| 1 | 1 | (1, 0, 0) | (0, 0, 1) | (0, 1, 0) | Nein | × × × |
| 2 | 2 | (0, 1, 1) | (0, 0, 1) | (1, 1, 0) | Nein | × × × |
| 3 | 3 | (0, 0, 1) | (0, 0, 1) | (1, 1, 0) | Ja | × × |
| 4 | 1 | (1, 0, 0) | (1, 1, 1) | (0, 1, 0) | Ja | × |
| 5 | 2 | (0, 1, 1) | (1, 2, 1) | (1, 1, 0) | Ja | |

Alle Prozesse konnten schrittweise markiert werden. Es existiert also eine vollständige sichere Ausführungsreihenfolge.

⇒ **Es liegt kein Deadlock vor.**

Erklärung der Tabelle: Die folgende Tabelle zeigt Schritt für Schritt, wie der Bankieralgorithmus arbeitet:

- In jeder Zeile wird ein Prozess betrachtet.
- Wenn seine Ressourcenanforderung R_i durch den aktuellen Verfügbarkeitsvektor A erfüllbar ist, wird der Prozess **markiert**.
- Anschließend wird seine Ressourcenbelegung C_i zu A addiert (Ressourcenfreigabe nach Terminierung).
- Die nächste Zeile prüft erneut alle Prozesse, bis keine unmarkierten Prozesse mehr übrig sind.

8.4 Aufgabe 5

CPU-bound (rechenintensive) Prozesse führen lange Rechenoperationen mit wenigen I/O-Unterbrechungen aus. Sie profitieren von **längerer CPU-Zuteilung** ohne häufiges Umschalten, da Kontextwechsel mit **Overhead** verbunden sind (z.B. Speichern des Zustands, Leeren von Caches wie dem TLB).

I/O-bound (ein-/ausgabeintensive) Prozesse führen dagegen viele kurze Rechenphasen aus, gefolgt von häufigem Warten auf I/O-Geräte. Würde man diese Prozesse zu lange auf der CPU halten, würde man **wertvolle CPU-Zeit verschwenden**, da sie häufig blockieren. Sie

profitieren daher von **kurzen, aber schnellen Reaktionszeiten**, sodass sie nach Beendigung ihrer I/O-Phasen zügig wieder eingeplant werden können.

Ein intelligenter Scheduler erkennt diese Unterschiede und trifft darauf basierende Entscheidungen, um die **Systemauslastung zu optimieren** und sowohl **Rechenzeit effizient zu verteilen** als auch **I/O-Wartezeiten zu minimieren**.

8.5 Aufgabe 6

a) Gantt-Diagramme für verschiedene Scheduling-Strategien**

Gegeben sind fünf Prozesse mit folgenden CPU-Burst-Zeiten (in Millisekunden):

| Prozess | Burst-Zeit |
|---------|------------|
| P1 | 10 |
| P2 | 1 |
| P3 | 2 |
| P4 | 1 |
| P5 | 5 |

i) FCFS (First-Come, First-Served)** Prozesse werden in ihrer Ankunftsreihenfolge vollständig abgearbeitet:

| | | | | | |
|----|----|----|----|----|----|
| P1 | P2 | P3 | P4 | P5 | |
| 0 | 10 | 11 | 13 | 14 | 19 |

ii) SJF (Shortest Job First)** Prozesse werden nach aufsteigender Burst-Zeit sortiert und dann nacheinander ausgeführt:

| | | | | | |
|----|----|----|----|----|----|
| P2 | P4 | P3 | P5 | P1 | |
| 0 | 1 | 2 | 4 | 9 | 19 |

iii) RR (Round-Robin), Quantum = 1 ms** Prozesse werden in Zeitscheiben von 1 ms ausgeführt und nach jedem Quantum ans Ende der Queue gesetzt. Ergebnis bis Zeit 16 ms:

| | | | | | | | | |
|----|----|----|----|----|----|----|----|---|
| P1 | P2 | P3 | P4 | P5 | P1 | P3 | P5 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

| | | | | | | | | |
|----|----|----|----|----|----|----|----|----|
| P1 | P5 | P1 | P5 | P1 | P1 | P1 | P1 | |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

| P1 | P1 | P1 |
 16 17 18 19

b) Durchlaufzeiten:

Die **Durchlaufzeit** eines Prozesses ist definiert als:

$$\text{Durchlaufzeit} = \text{Fertigstellungszeit} - \text{Ankunftszeit}$$

Da alle Prozesse bei $t = 0$ starten, entspricht die Durchlaufzeit hier einfach der **Zeit der letzten Ausführung** eines Prozesses im jeweiligen Gantt-Diagramm.

i) FCFS (First-Come, First-Served)

Gantt-Diagramm:

| P1 | P2 | P3 | P4 | P5 |
 0 10 11 13 14 19

| Prozess | Durchlaufzeit |
|---------|---------------|
| P_1 | 10 |
| P_2 | 11 |
| P_3 | 13 |
| P_4 | 14 |
| P_5 | 19 |

ii) SJF (Shortest Job First)**

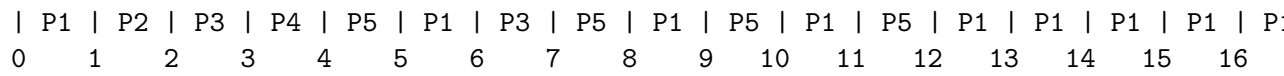
Gantt-Diagramm:

| P2 | P4 | P3 | P5 | P1 |
 0 1 2 4 9 19

| Prozess | Durchlaufzeit |
|---------|---------------|
| P_1 | 19 |
| P_2 | 1 |
| P_3 | 4 |
| P_4 | 2 |
| P_5 | 9 |

iii) RR (Round-Robin), Quantum = 1 ms**

Gantt-Diagramm:



Fertigstellungszeitpunkte (letzte Ausführung pro Prozess):

- P_2 : endet bei $t = 2$
- P_4 : endet bei $t = 4$
- P_3 : endet bei $t = 7$
- P_5 : endet bei $t = 14$
- P_1 : endet bei $t = 19$

| Prozess | Durchlaufzeit |
|---------|---------------|
| P_1 | 19 |
| P_2 | 2 |
| P_3 | 7 |
| P_4 | 4 |
| P_5 | 14 |

Vergleich der Durchlaufzeiten:

| Prozess | FCFS | SJF | RR (Q = 1 ms) |
|---------|------|-----|---------------|
| P_1 | 10 | 19 | 19 |
| P_2 | 11 | 1 | 2 |
| P_3 | 13 | 4 | 7 |
| P_4 | 14 | 2 | 4 |
| P_5 | 19 | 9 | 14 |

c) Wartezeit (Waiting Time)

Die Wartezeit berechnet sich aus:

$$\text{Wartezeit} = \text{Durchlaufzeit} - \text{Burst-Zeit}$$

Die Burst-Zeiten (CPU-Bedarf) der Prozesse sind:

| Prozess | Burst-Zeit |
|---------|------------|
| P_1 | 10 |
| P_2 | 1 |
| P_3 | 2 |
| P_4 | 1 |
| P_5 | 5 |

Berechnung der Wartezeiten je Strategie

i) FCFS

| Prozess | Durchlaufzeit | Burst | Wartezeit |
|---------|---------------|-------|-----------|
| P_1 | 10 | 10 | 0 |
| P_2 | 11 | 1 | 10 |
| P_3 | 13 | 2 | 11 |
| P_4 | 14 | 1 | 13 |
| P_5 | 19 | 5 | 14 |

ii) SJF

| Prozess | Durchlaufzeit | Burst | Wartezeit |
|---------|---------------|-------|-----------|
| P_1 | 19 | 10 | 9 |
| P_2 | 1 | 1 | 0 |
| P_3 | 4 | 2 | 2 |
| P_4 | 2 | 1 | 1 |
| P_5 | 9 | 5 | 4 |

iii) RR ($Q = 1 \text{ ms}$)

| Prozess | Durchlaufzeit | Burst | Wartezeit |
|---------|---------------|-------|-----------|
| P_1 | 19 | 10 | 9 |
| P_2 | 2 | 1 | 1 |
| P_3 | 7 | 2 | 5 |
| P_4 | 4 | 1 | 3 |
| P_5 | 14 | 5 | 9 |

iv) Durchschnittliche Wartezeiten:

i) FCFS:

Wartezeiten: 0, 10, 11, 13, 14

$$\text{Summe} = 48, \quad \text{Durchschnitt} = \frac{48}{5} = 9,6 \text{ ms}$$

ii) SJF:

Wartezeiten: 9, 0, 2, 1, 4

$$\text{Summe} = 16, \quad \text{Durchschnitt} = \frac{16}{5} = 3,2 \text{ ms}$$

iii) RR (Quantum = 1 ms):

Wartezeiten: 9, 1, 5, 3, 9

$$\text{Summe} = 27, \quad \text{Durchschnitt} = \frac{27}{5} = 5,4 \text{ ms}$$

Vergleich – Durchschnittliche Wartezeit

| Strategie | Durchschnittliche Wartezeit |
|-----------|-----------------------------|
| FCFS | 9,6 ms |
| SJF | 3,2 ms |
| RR (Q=1) | 5,4 ms |

⇒

SJF liefert die **geringste durchschnittliche Wartezeit**, da kurze Prozesse bevorzugt behandelt werden. Allerdings kann SJF zu **Verhungern (Starvation)** führen, wenn lange Prozesse ständig verdrängt werden. Round-Robin bietet bessere Fairness, aber höhere Gesamtlatenz.

8.6 Aufgabe 7

- **FCFS (First-Come, First-Served):** Keine Gefahr von Verhungern, da Prozesse strikt in Ankunftsreihenfolge ausgeführt werden.
- **SJF (Shortest Job First):** Gefahr von Verhungern besteht, da lange Prozesse durch immer neu ankommende kurze Prozesse dauerhaft verdrängt werden können.
- **SRTF (Shortest Remaining Time First):** Ebenfalls anfällig für Verhungern, insbesondere für Prozesse mit langer verbleibender Laufzeit, da kürzere Prozesse bevorzugt werden.
- **RR (Round-Robin):** Keine Gefahr von Verhungern, da alle Prozesse regelmäßig CPU-Zeit erhalten.
- **Multilevel Queue mit fixer Priorität:** Hohe Verhungernsgefahr für Prozesse in unteren Prioritätsklassen, wenn ständig Prozesse mit höherer Priorität ankommen.