Operating Systems and Networs SoSe 25 Solutions

Igor Dimitrov

2024-12-18

Table of contents

Preface

1 Blatt 01

1.1 Aufgabe 1

Learning how to Learn:

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

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

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

John Cleese:

- **Zwei Denkmodi**:

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

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

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

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

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

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

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

1.2 Aufgabe 2

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

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

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

1.3 Aufgabe 3

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

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

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

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

1.4 Aufgabe 4

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

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

Daher hat jeder Prozess:

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

1.5 Aufgabe 5

Entfernte Systemaufrufe

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

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

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

Ohne sie wären bestimmte Kernfunktionen unmöglich.

1.6 Aufgabe 6

`script.sh` auch im Zip:

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

1.7 Aufgabe 7

Vorteile:

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

Nachteile:

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

2 Blatt 02

2.1 Aufgabe 1

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

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

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

2.560 Byte ≈ 2,5 KB

2.2 Aufgabe 2

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

Rückgabewert:

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

a) Mit dem program:

```c
#include <stdio.h>

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

```
    return 0;
}
```

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

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

Schrittweise Erzeugung der Prozesse:

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

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

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

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

2.3 Aufgabe 3

Erklärung zur Ausgabe von `ps -T -H`

Das C-Programm:

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

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

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

Ablauf der Kommandos:

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

Ausgabe von `ps -T -H`:

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

Erklärung:

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

Die Spalten

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

Process state Codes

Prozesszustände (erste Buchstaben):

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

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

Zusätzliche flags:

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

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

Tiefe der Aktuellen Sitzung

Zuerst finden wir die PID der Aktuellen sitzung mit

```
echo $$
```

heraus. Output: `1025`.

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

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

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

2.4 Aufgabe 4

Übersicht der Varianten mit Signaturen:

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

Wichtige Unterschiede:

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

Wann welche Variante?

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

2.5 Aufgabe 5

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

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

Der PCB enthält:

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

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

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

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

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

2.6 Aufgabe 6

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

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

   - „creating thread i“

- „number: i“
- „ending thread i“

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

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

Listing 2.1 `threads_example.c`

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

#define NUM_THREADS 200000

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

int main()
{
   pthread_t thread;
   int thread_arg;

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

   return 0;
}
```

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

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

#define NUM_THREADS 200000

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

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

   return 0;
}
```

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

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

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`). 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
```

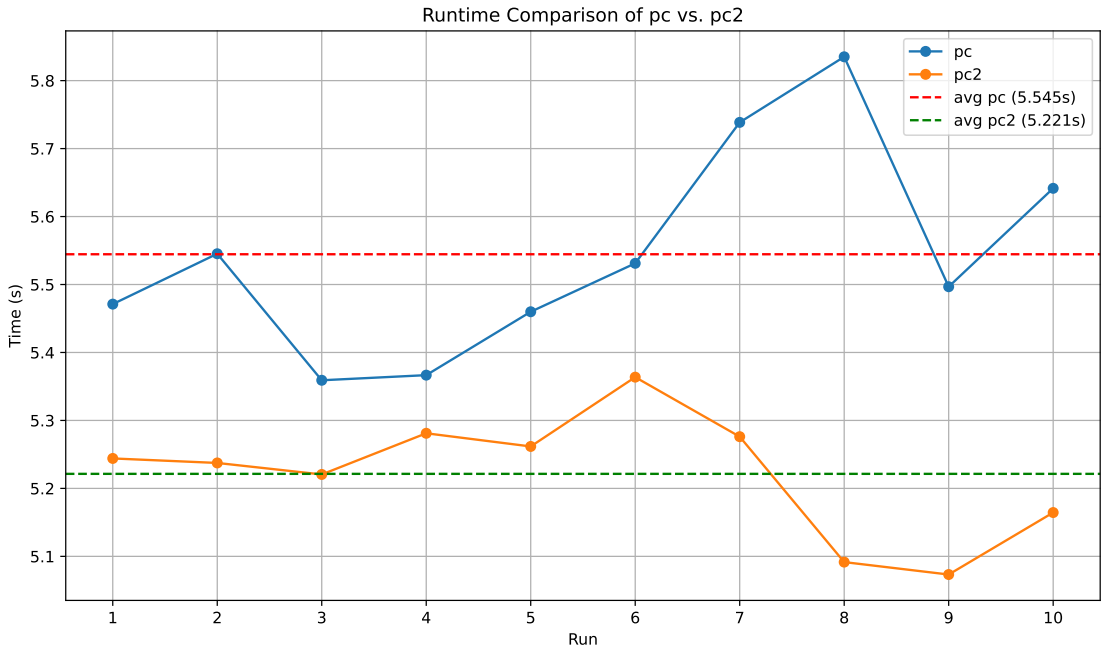
Die Ergebnisse zeigen, dass `pc2` im Schnitt etwas schneller ist als `pc` (5.22 s gegenüber 5.54 s), was den Effizienzgewinn durch den Verzicht auf aktives Warten bestätigt.

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

Zur Veranschaulichung wurde mit dem folgendnen 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`.

```python
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 3

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 → `S = 4` B nimmt 1 Token → `S = 3` A nimmt 1 → `S = 2` B nimmt 1 → `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`

```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);
```