Technische Universität Dortmund

# Klausur „Betriebssysteme“

| | erreichbare Punkte | erhaltene Punkte | | | | |
|---|---|---|---|---|---|---|
| **Aufgabe 1** | 12 | **a** | **b** | **c** | **d** | |
| | | | | | | |
| **Aufgabe 2** | 13 | **a** | **b** | | | |
| | | | | | | |
| **Aufgabe 3** | 14 | **a** | **b** | **c** | | |
| | | | | | | |
| **Aufgabe 4** | 10 | **a** | **b** | **c** | | |
| | | | | | | |
| **Aufgabe 5** | 10 | **a** | **b** | | | |
| | | | | | | |
| **Aufgabe 6** | 31 | **a** | **b** | | | |
| | | | | | | |
| **Summe** | 90 | | | | | |
| **Spickzettel vorhanden?** | | | | | | |

| | | | | | |
|---|---|---|---|---|---|
| | | | | | |

(Matrikel-Nr.)

(Name) (Vorname)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (22 Seiten inklusive Deckblatt),
- den Empfang der Manualseiten (zwei Blätter mit 5 Manualseiten: ctime, malloc, opendir/readdir/closedir, qsort und stat),
- die Kenntnisnahme der Hinweise auf Seite 2.

Dortmund, 22.07.2025

(Unterschrift)

# Hinweise

Bitte lesen Sie die folgenden Informationen **aufmerksam** und unterschreiben Sie die Erklärung auf der ersten Seite.

- Es können **90 Punkte** erreicht werden; 50% der Gesamtpunktzahl sind zum Bestehen der Klausur hinreichend. Zur Bearbeitung stehen insgesamt **90 Minuten** zur Verfügung.
- Als Hilfsmittel ist lediglich ein **von Ihnen eigenhändig geschriebenes** (Handschrift, kein Ausdruck oder Kopie) **A4-Blatt** (beidseitig), welches eingesammelt wird. Ansonsten dürfen **keine** weiteren Hilfsmittel verwendet werden.
- Die Antworten sind in **deutscher** oder **englischer** Sprache zu verfassen.
- Notieren Sie Ihre Lösungen direkt in der Klausur unter Verwendung eines dokumentenechten, **schwarzen oder blauen Stifts**. Entfernen Sie **nicht** die Heftung der Blätter. Falls der vorgesehene Platz nicht ausreichen sollte, können Sie auch die Rückseiten der Blätter und die Reserveseiten am Ende der Klausur verwenden. Verweisen Sie an der für die Lösung vorgesehenen Stelle auf die genutzte Seite.
- Wir bewerten ausdrücklich gemäß der aus Vorlesung und Übungen sowie der Begleitliteratur bekannten Definitionen (Algorithmen und Konzepten) und Begrifflichkeiten.

**Aufgabe 1: Ankreuzfragen (12 Punkte)**

Bei den Einfachauswahlfragen in dieser Aufgabe ist jeweils nur **eine** richtige Antwort eindeutig anzukreuzen. Auf die richtige Antwort gibt es die angegebene Punktzahl.

Wollen Sie eine Antwort korrigieren, streichen Sie bitte die falsche Antwort mit drei waagrechten Strichen durch (☒) und kreuzen die richtige an.

Lesen Sie die Frage genau, bevor Sie antworten.

a) **Ablaufplanung (Scheduling):** Welche Aussage ist korrekt?

3 Punkte

- ☐ First-Come-First-Served (FCFS) garantiert, dass kurze Prozesse immer schneller fertig sind als lange Prozesse.
- ☐ Bei prioritätsorientierter Ablaufplanung kann es zum Verhungern (Starvation) von Prozessen mit niedriger Priorität kommen.
- ☐ Beim präemptiven Scheduling wird ein Prozess niemals unterbrochen, solange er noch Rechenzeit benötigt.
- ☐ Round-Robin-Scheduling behandelt E/A-intensive Prozesse bevorzugt.

b) **Virtueller Speicher:** Welchen Vorteil hat die Einteilung des Speichers in Seitenrahmen (Kacheln)?

3 Punkte

- ☐ Sie macht es überflüssig, zwischen logischen und physischen Adressen zu unterscheiden.
- ☐ Sie sorgt dafür, dass Prozesse unabhängig von ihrer Größe stets in einen einzigen zusammenhängenden Speicherbereich geladen werden.
- ☐ Sie stellt sicher, dass jeder Prozess immer den gesamten physischen Speicher (RAM) nutzen kann.
- ☐ Sie ermöglicht eine effiziente Nutzung des Speichers durch Vermeidung von externem Verschnitt.

c) **Interprozesskommunikation:** Welche Aussage über UNIX-Signale ist korrekt?

3 Punkte

- ☐ Signale sind softwarebasierte Unterbrechungen, die wie Hardware-Interrupts wirken und dem Prozess beim nächsten Übergang in den User Mode zugestellt werden.
- ☐ Ein Signal dient der Interprozesskommunikation und enthält neben der Signalnummer immer auch einen Datenpuffer.
- ☐ Ein Prozess kann Signale nur empfangen wenn er gerade keine Systemaufrufe (z.B. `open`) ausführt, da diese sonst abgebrochen werden könnten.
- ☐ Die Standardbehandlung eines Signals führt immer zur Terminierung des Prozesses.

d) **Synchronisation:** Welche der folgenden Aussagen zu *aktivem Warten* ist richtig?

3 Punkte

- ☐ Aktives Warten garantiert, dass eine Ressource schneller freigegeben wird, weil der wartende Prozess sofort reagieren kann.
- ☐ Beim aktiven Warten überprüft ein Prozess eine Bedingung in einer Endlosschleife und verbraucht dabei Rechenzeit wenn er nicht fortschreiten kann.
- ☐ Aktives Warten wird in modernen Betriebssystemen grundsätzlich vermieden und ist technisch nicht mehr möglich.
- ☐ Aktives Warten tritt nur bei Mehrkernprozessoren auf und ist auf Einkernsystemen ausgeschlossen.

**Aufgabe 2: Prozesse und Scheduling (13 Punkte)**

a) UNIX-Systemaufrufe

5 Punkte

Geben Sie die Ausgabe des folgenden C-Programms an.
**Hinweis:** Das Einbinden der Header-Dateien sowie ein Teil der Fehlerbehandlung wurden ausgelassen. Gehen Sie von einem fehlerfreien Ablauf aus. Das Symbol ␣ steht für ein Leerzeichen.

```
int x = 16;
int subtrahend = 8;

void do_minus(char* msg, int subtrahend){
    if (x >= subtrahend) {
        x -= subtrahend;
        printf("%s%d,␣", msg, x);
    } else {
        printf("%s?,␣", msg);
    }
}
int main(){
    pid_t pid = fork();
    if (pid == 0){
        subtrahend = 14;
        do_minus("j", 4);
    } else if (pid > 0){
        int x = 99;
        wait(NULL);
        do_minus("k", subtrahend);
    } else {
        do_minus("g", subtrahend);
    }
    do_minus("l", subtrahend);
    return 0;
}
```

**Ausgabe:**

---

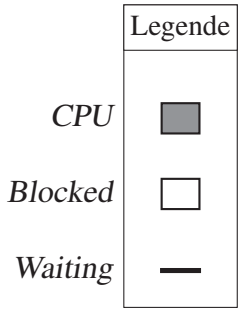
b) Scheduling-Verfahren

8 Punkte

Gegeben sind drei zyklisch arbeitende Prozesse P1, P2 und P3. Die Prozesse treffen mit der in der folgenden Tabelle angegebenen Ankunftszeit ein und führen erst einen (vollständigen) CPU-Stoß und dann einen (vollständigen) E/A-Stoß aus.

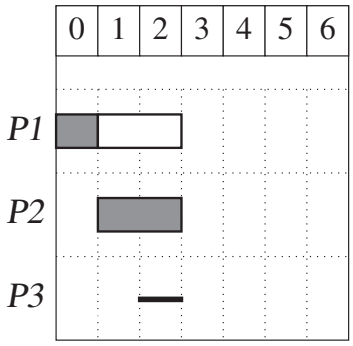
| Prozess | Ankunftszeit | CPU-Zeit | E/A-Zeit |
|---|---|---|---|
| P1 | 0 | 1 | 3 |
| P2 | 1 | 3 | 2 |
| P3 | 2 | 4 | 1 |

Das Scheduling arbeitet nach dem in der jeweiligen Teilaufgabe angegebenen Strategie. Die ersten drei Zeiteinheiten sind von uns bereits vorgegeben. Zeichnen Sie entsprechend der Legende in das folgende Gantt-Diagramm ein, wie P1, P2 und P3 im Folgenden abgearbeitet (Prozesszustände) werden.



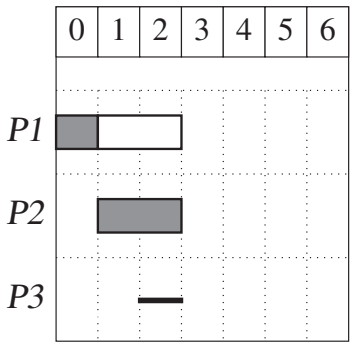

**Hinweis:** Die Prozessumschaltzeit kann vernachlässigt werden. Es gibt nur eine CPU, die E/A-Vorgänge der Prozesse können parallel ausgeführt werden. Es genügt, wenn Sie die Flächen für Running geeignet schraffieren.

I) First Come First Serve (2 Punkte)



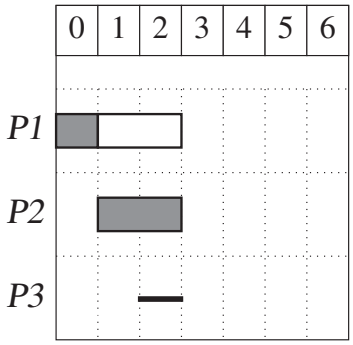

II) Round Robin (2 Punkte)

Die Länge der Zeitschreiben beträgt 2 Zeiteinheiten.

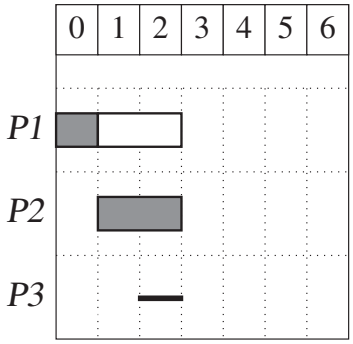


III) Virtual Round Robin (2 Punkte)

Die Länge der Zeitschreiben beträgt 2 Zeiteinheiten.




IV) Shortest-Process-Next-Strategie (2 Punkte)




(Ersatzdiagramme auf dem Reserveblatt: Streichen Sie ungültige Lösungen deutlich durch!)

**Aufgabe 3: Synchronisation und Verklemmungen (14 Punkte)**

a) Dead- vs. Livelock

4 Punkte

Erklären Sie den **Unterschied** zwischen **Deadlock** und **Livelock**. Gehen Sie dabei auch darauf ein, warum Livelocks in der Regel das größere von beiden Problemen darstellen.

**Antwort:**

b) Verklemmungsbekämpfung

3 Punkte

In der Veranstaltung wurden verschiedene konkreten Gegenmaßnahme zur **Bekämpfung von Verklemmungen** genannt. Diese lassen sich in drei übergeordnete Kategorien einteilen. Nennen Sie **zwei** dieser Kategorien und je eine Maßnahme pro Kategorie.

**Antwort:**

c) Achterbahn

7 Punkte

Es gibt `n` Passagiere und einen Zug mit Kapazität `C` (`C < n`). Zug und Passagiere sind jeweils Threads. Der Zug darf nur abfahren, wenn er voll ist.

**Ablauf**

- Passagier-Threads: wiederholen jeweils für sich `board` → `unboard`
- Zug-Thread: wiederholt `load` → `run` → `unload`

**Bedingungen**

- `board` erst nach `load`
- `run` erst, wenn genau `C` Passagiere `board` ausgeführt haben
- `unboard` erst nach `unload`
- neues `load` erst, wenn alle `C` Passagiere `unboard` ausgeführt haben

**Hinweis**

- `signal(C)` erhöht einen Semaphor direkt um `C`

**Aufgabe**

- Ergänzen Sie nur das Programm eines **Passagier-Threads** mit den nötigen Synchronisationsanweisungen und Bedingungen (`boarders` / `unboarders`) für Zu- bzw. Ausstieg.
- Halten Sie alle obigen Regeln ein.
- Nutzen Sie alle vorgegebenen Variablen im gemeinsamen Speicher.
- Die Ausführung muss beliebig oft wiederholbar sein, ohne Verklemmungen.

```
/* gemeinsamer Speicher */
Semaphore mutex = 1;
int boarders = 0;
int unboarders = 0;
Semaphore boardQueue = 0;
Semaphore unboardQueue = 0;
Semaphore allAboard = 0;
Semaphore allAshore = 0;

/* Zug /*
/* NICHT verändern! /*
while(1) {
 load()
 boardQueue.signal(C)
 allAboard.wait()

 run()

 unload()
 unboardQueue.signal(C)
 allAshore.wait()
}
```

```
/* Passagier */
while(1) {

 board();

 boarders += 1;
 if(boarders == C) {

  boarders = 0;
 }

 unboard();

 unboarders += 1;

}
```

**Aufgabe 4: Speicherverwaltung und virtueller Speicher (10 Punkte)**

a) *Platzierungsstrategien*

4 Punkte

Die folgenden beiden Tabellen zeigen die Belegung eines 32 MiB Speichers an, der in 2 MiB große Blöcke eingeteilt ist.

Anfragen, die nicht in das Muster passen werden auf die nächsten 2 MiB aufgerundet.

Im Folgenden werden zwei Anfragen A und B nacheinander Speicher anfragen.

Vervollständigen Sie die jeweiligen Tabellen so, dass die nachfolgenden Zeilen der Tabelle die Speicherbelegung nach der Anfrage von A und nach der Anfrage von B repräsentieren.

Verwenden Sie dabei jeweils die angegebene Platzierungsstrategie. Kann eine Anfrage nicht beantwortet werden, kennzeichnen Sie die entsprechende Zeile der Tabelle mit einem **E** in der ersten Spalte und lassen den Rest leer.

**Best Fit-Verfahren:**

| Aktion | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Grundzustand | | | X | X | X | | | Y | Y | | | | | Z | Z | Z |
| A reserviert 4 MiB | | | | | | | | | | | | | | | | |
| B reserviert 5 MiB | | | | | | | | | | | | | | | | |

**Worst Fit-Verfahren:**

| Aktion | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Grundzustand | | | X | X | X | | | Y | Y | | | | | Z | Z | Z |
| A reserviert 4 MiB | | | | | | | | | | | | | | | | |
| B reserviert 5 MiB | | | | | | | | | | | | | | | | |

(Ersatztabelle auf dem Reserveblatt: Streichen Sie ungültige Lösungen deutlich durch!)

b) Verschnitt

3 Punkte

In der Situation in Aufgabe a) ensteht interner Verschnitt. Erklären Sie kurz, was interner Verschnitt ist und wodurch er in der obrigen Situation entsteht.

c) Segmentbasierte Addressberechnung

3 Punkte

Geben Sie für die logischen Adressen $0200\ 005D_{16}$ und $FF01\ 1111_{16}$ jeweils die physikalische Adresse unter Anwendung der segmentbasierten Adressabbildung an. Die höchstwertigen 8 Bit der logischen Adresse geben die Position innerhalb der Segmenttabelle an. Falls eine Speicheranfrage eine Zugriffsverletzung auslöst, machen Sie dies kenntlich.

Segmenttabelle:

| | Startadresse | Länge |
|---|---|---|
| $00_{16}$ | $0815\ 0000_{16}$ | $00\ 000F_{16}$ |
| $01_{16}$ | $1337\ 0000_{16}$ | $00\ 00FF_{16}$ |
| $02_{16}$ | $F00D\ CFB0_{16}$ | $00\ 08FF_{16}$ |
| $03_{16}$ | $8080\ 0000_{16}$ | $00\ FFFF_{16}$ |
| ... | | |
| $FE_{16}$ | $01CE\ 0000_{16}$ | $10\ 0000_{16}$ |
| $FF_{16}$ | $1FFF\ 0000_{16}$ | $01\ FFFF_{16}$ |

logische Adresse: $0200\ 005D_{16}$

→ physikalische Adresse:

logische Adresse: $FF01\ 1111_{16}$

→ physikalische Adresse:

**Aufgabe 5: Dateisysteme (10 Punkte)**

a) UNIX Block Buffer Cache

6 Punkte

Erläutern Sie kurz den UNIX Block Buffer Caches. Beantworten Sie hierfür die folgenden Punkte:

- Wozu dient dieser?
- Addressierung und Verwaltung der belegten und freien Blocke im Cache?
- Wie werden Lesevorgänge optimiert?
- Wann erfolgt das Schreiben?

b) Speicherung von Dateien

4 Punkte

Wie unterscheidet sich die Speicherung von Dateien im klassichen Windows FAT und UNIX Inode System? Benennen Sie jeweils das zugrundeliegende Verfahren und geben Sie einen Vor- und Nachteil an.

**Aufgabe 6: Programmieraufgabe: Dateiauflistung nach Änderungsdatum (31 Punkte)**

Vervollständigen Sie das Programm `list_dates`, welches alle regulären Dateien in einem Verzeichnis nach ihrer Größe aufsteigend sortiert ausgibt.

**Aufgabenstellung:**

Sie müssen die folgenden Funktionen implementieren:

a) **`list_files()`**, welche ein Array aus Einträgen (`FileInfo`) mit allen regulären Dateien eines Verzeichnisses und deren Größe erstellt.

b) **`main()`**, welche `list_files()` mit einem als Argument übergebenen Pfad aufruft und die Einträge anschließend sortiert auf der Standardausgabe ausgibt.

Details zur Aufgabenstellung erhalten Sie an den passenden Stellen.

**Hilfsfunktionen:**

Folgende Hilfsfunktionen stehen Ihnen zur Verfügung (Details siehe nächste Seite):

- `concat_paths()`: Konkateniert ein Pfadpräfix mit einem Suffix zu einem vollständigen Pfad. (*Achtung:* Alloziert Speicher).
- `fill_fileinfo()`: Füllt die `FileInfo`-Struktur für eine reguläre Datei.
- `compare_file_dates()`: Vergleichsfunktion für `FileInfo`-Datensätze für die Nutzung in `qsort()`.
- `error()`: Gibt eine Fehlermeldung aus und bricht das Programm ab (Fehlerfall).

**Relevante Systemaufrufe:**

- Beigefügten Handbuchseiten: `opendir`, `readdir` und `closedir`, `stat`, `inode`, `qsort`, `malloc`, `ctime`

**Wichtige Hinweise:**

- *Belegte Ressourcen* (d.h. geöffnete Datein/Verzeichnisse und allozierter Speicher) müssen im Normalfall *freigegeben* werden. Ausnahme: Fehlerfall!
- Achten Sie in Ihrer Implementierung auf *korrekte Fehlerbehandlung* aller Systemaufrufe und Hilfsfunktionen. Nutzen Sie `error()`, um Schreibarbeit zu sparen (Achtung: ggf. auf `errno` achten).
- Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die semantische Umsetzung.

**Vorgegebene Funktionen:**

```c
// Gehen Sie davon aus, dass alle notwendigen Header inkludiert sind
// Kurzschreibweise Fehlerbehandlung (Programmabbruch)
void error(char* msg) {perror(msg); exit(errno);}

// Konkateniere Prä- und Suffix zu einem Pfad.
// Rückgabewert muss später mit free freigegeben werden.
// Rückgabe 0: Fehler aufgetreten und errno gesetzt
// Sonst: Rückgabewert ist ein gültiger char*
static char* concat_paths(char* path, char* filename) {
    /* Implementierungsdetails nicht relevant */
}

// Struktur zum Speichern von Dateinamen und Änderungsdatum
typedef struct FileInfo {
    char* path; // muss ggf. seperat freigegeben werden
    time_t mtime;
} FileInfo;

// Speichert den Dateipfad und das Änderungsdatum in ein FileInfo-Struct.
int fill_fileinfo(struct FileInfo* fi, char* filepath, struct stat* sb) {
    fi->path = filepath;
    fi->mtime = sb->st_mtime;
    return 0;
}

// Vergleicht Dateieinträge nach Änderungsdatum.
// Rückgabe: neg. wenn a < b, pos. wenn a > b, 0 wenn gleich.
int compare_file_dates(const void *a, const void *b) {
    double diff = difftime(((FileInfo *)a)->mtime, ((FileInfo *)b)->mtime);
    return (diff > 0.0) - (diff < 0.0);
}
```

Die Funktion **`list_files()`** erstellt ein Array mit Einträgen für alle regulären Dateien in einem Verzeichnis. Gehen sie dabei wie folgt vor:

**Schritt 1:** Öffnen Sie das angegeben Verzeichnis (`path`) und lesen Sie alle Einträge (`opendir()` und `readdir()`; Spezielle Fehlerbehandlung beachten!).

**Schritt 2:** Erzeugen Sie jeweils den vollen Pfad der Dateien (`concat_paths()`) und identifiziert mithilfe von `stat()` die regulären Dateien.

**Schritt 3:** Befüllen Sie fortlaufend ein `FileInfo`-Arrary, dessen Speicherbedarf sie mit `realloc()` dynamisch anpassen. `fill_fileinfo()` kann die einzelnen Einträge initialisieren.

**Schritt 4:** Geben Sie das Array von `FileInfo`-Strukturen als Rückgabewert zurück und setzen Sie `fileCounter` auf die Anzahl der regulären Dateien.

a) `list_files`-Funktion (19.5 Punkte)

```
// Speichert Informationen zu allen Dateien in einem Verzeichnis
FileInfo* list_files(char* path, int* fileCount) {
    DIR* dir;
    struct dirent* entry;
    struct stat sb;
    *fileCount = 0;
    FileInfo *fileArray = NULL;
```

}

Diese Funktion **`main()`** nimmt einen Pfad auf einen Verzeichnis als Eingabeparameter. Geben sie die Dateien im Ziel nach Änderungsdatum sortiert aus und gehen Sie dabei wie folgt vor:

- Diese Funktion verarbeitet die Kommandozeilenargumente. Stellen Sie sicher, dass genau ein Argument (der Pfad zum Verzeichnis) übergeben wird.
- Rufen Sie `list_files()` auf, um ein Array von `FileInfo`-Strukturen zu erhalten
- Sortieren Sie das Array mithilfe von `qsort()` und der Vergleichsfunktion `compare_file_dates()`
- Geben Sie die Dateipfade zusammen mit ihrem Änderungsdatum im lesbarer Form (`ctime()`) über die Konsole aus

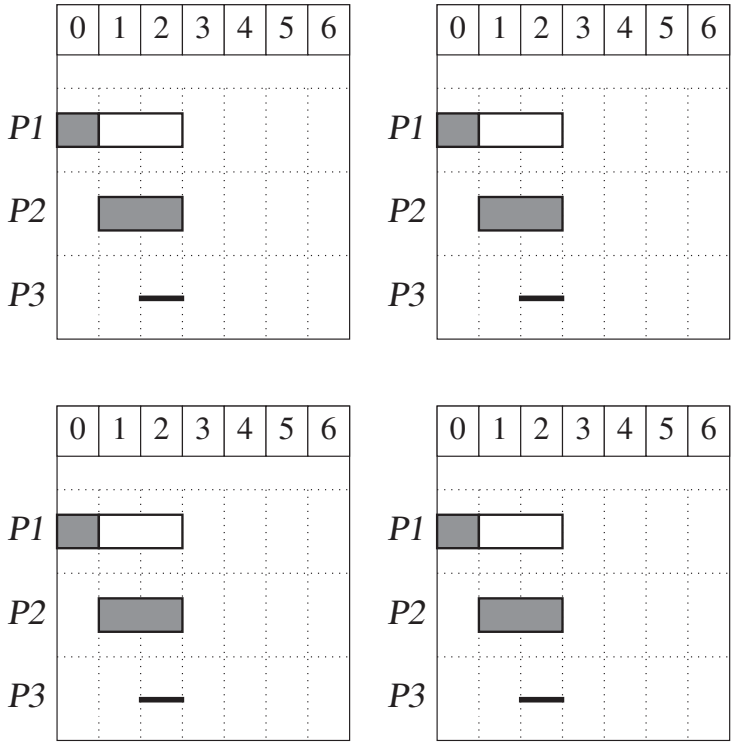
b) `main`-Funktion und Vergleichsfunktion (11.5 Punkte)

```
int main(int argc, char* argv[]) {
```

```
    return 0;
}
```

# Reserveblatt

**Bitte die Strategie mit angeben!**




**Bitte das Verfahren angeben:**

| Aktion | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Grundzustand | | | X | X | X | | | Y | Y | | | | | Z | Z | Z |
| A reserviert 4 MiB | | | | | | | | | | | | | | | | |
| B reserviert 5 MiB | | | | | | | | | | | | | | | | |

**Bitte das Verfahren angeben:**

| Aktion | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Grundzustand | | | X | X | X | | | Y | Y | | | | | Z | Z | Z |
| A reserviert 4 MiB | | | | | | | | | | | | | | | | |
| B reserviert 5 MiB | | | | | | | | | | | | | | | | |