

Technische Universität Dortmund
Klausur „Betriebssysteme“

	erreichbare Punkte	erhaltene Punkte				
Aufgabe 1	12	a	b	c	d	
Aufgabe 2	12	a	b			
Aufgabe 3	12					
Aufgabe 4	12	a	b	c		
Aufgabe 5	12	a	b			
Aufgabe 6	30	a	b			
Summe	90					
Spickzettel? {ja, nein}						

--	--	--	--	--	--

(Matrikel-Nr.)

 (Nachname)

 (Vorname)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (20 Seiten inklusive Deckblatt),
- den Empfang der Manualseiten (zwei Blätter mit 7 Manualseiten: CHMOD, feof/ferror/fileno, fgetc, fopen/fdopen/fileno, opendir/readdir/closedir, pthread_create/pthread_exit und pthread_join),
- die Kenntnisnahme der Hinweise auf Seite 2.

Dortmund, 24.09.2026

 (Unterschrift)

Hinweise

Lesen Sie die **Hinweise** aufmerksam und **füllen Sie das Deckblatt vollständig aus** (Matrikelnummer, Name, Vorname, Spickzettel: *ja/nein*, Unterschrift).

– **Ablauf und Bewertung:**

- **90 Minuten**, maximal **90 Punkte**; 50% reichen zum Bestehen.
- Antworten Sie auf **Deutsch oder Englisch** direkt auf den Aufgabenblättern. **Heftung nicht entfernen!**
- Bei Platzmangel Rückseiten nutzen und wechselseitig auf Aufgabe und Lösung verweisen. Ersatzabbildungen finden Sie ggf. in der Aufgabe oder am Ende der Klausur.
- Verworfenne Lösungen deutlich durchstreichen; bei mehreren nicht gestrichenen Lösungen wird die schlechteste gewertet.
- Es gelten die Begriffe, Algorithmen und Varianten aus Vorlesung, Übungen und Tutorien.

– **Täuschung und Ordnungsverstoß:** Täuschung und Täuschungsversuche (insbesondere unerlaubte Hilfsmittel, Abschreiben, Abschreibenlassen oder unerlaubte Zusammenarbeit) führen auch *ohne vorherige Ermahnung* zur Bewertung der Prüfung mit **nicht ausreichend (5,0)**. Bei Störung des Prüfungsablaufs kann in der Regel nach Ermahnung ein Ausschluss erfolgen; die Prüfung gilt dann ebenfalls als **nicht ausreichend (5,0)** bewertet (vgl. § 18 Abs. 3 PO).

– **Erlaubte Hilfsmittel und Gegenstände:**

- Amtlicher **Lichtbildausweis** (Personalausweis, Reisepass, Führerschein, Aufenthaltstitel oder Flüchtlingsausweis; nicht digital) und **dokumentenechte Stifte** (schwarz/blau).
- **Spickzettel:** Ein beidseitig ausschließlich handschriftlich beschriebenes DIN-A4-Blatt (keine Ausdrucke/Kopien), oben rechts mit vollständigem Namen und Matrikelnummer. **Am Ende in die Klausur legen!**
- Kleiner Snack und Getränk (ohne andere zu stören).
- Wörterbuch der deutschen Sprache in begründeten Fällen nach vorheriger Anmeldung bei der Aufsicht.
- **Technische Geräte** (insbesondere Mobiltelefone, Smartwatches, Fitnessarmbänder, Smart-Brillen und Kopfhörer) sind **auszuschalten und nicht griffbereit oder am Körper zu verstauen**. Bereits das Mitführen eines unerlaubten, *betriebsbereiten* Geräts kann als **Täuschungsversuch** gewertet werden. Ausgenommen sind vorab genehmigte medizinische Hilfsmittel und bewilligte Nachteilsausgleiche.
- **Jegliche Kommunikation** mit anderen Prüfungsteilnehmenden ist **untersagt**, auch kurze Gespräche, Flüstern und Zeichen. Bereits ein Kommunikationsversuch kann als **Täuschungsversuch** gewertet werden. Bei Fragen oder Problemen wenden Sie sich ausschließlich an die Aufsicht.
- **Tipp:** Bleiben Sie ruhig! Bearbeiten Sie bei Schwierigkeiten zunächst eine andere Aufgabe. Auch Teillösungen geben Punkte. Viel Erfolg!

Mit meiner **Unterschrift** bestätige ich die **Kenntnisnahme der Klausurhinweise** und erkläre, dass mir zu Prüfungsbeginn **keine Umstände bekannt sind, die meine Prüfungsfähigkeit beeinträchtigen**.

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) **Abstraktionen:** Welche Gleichung beschreibt die Zeit pro verarbeitetem Datenblock eines Wechselfuffers am besten?

3 Punkte

- ☐ $\max(\text{DauerDerLeseoperation}, \text{RechenzeitFürDieVerarbeitung} + \text{DauerDesKopiervorgangsAusDemPuffer})$
- ☐ $\text{DauerDerLeseoperation} + \text{RechenzeitFürDieVerarbeitung}$
- ☐ $\max(\text{DauerDerLeseoperation}, \text{RechenzeitFürDieVerarbeitung}) + \text{DauerDesKopiervorgangsAusDemPuffer}$
- ☐ $\max(\text{DauerDerLeseoperation} + \text{DauerDesKopiervorgangsAusDemPuffer}, \text{RechenzeitFürDieVerarbeitung})$

b) **Ablaufplanung:** Was bezeichnet *Thrashing*?

3 Punkte

- ☐ Die kurzfristige Aus- und Wiedereinlagerung benutzter Seitenrahmen.
- ☐ Das automatische Aufräumen von ungenutzten Dateien.
- ☐ Eine Strategie um im Mehrprogrammbetrieb die CPU-Auslastung zu maximieren.
- ☐ Das dynamische Aufteilen eines Prozesses in mehrere Threads.

c) **Synchronisation:** Welche der folgenden Maßnahmen kann **NICHT** zur Auflösung eines Deadlocks dienen?

3 Punkte

- ☐ Die Anwendung des Vogel-Strauß-Algorithmus (*ostrich algorithm*).
- ☐ Das systemseitige (virtuelle) Entziehen von Ressourcen.
- ☐ Das Laden eines *Checkpoints* des Prozesses.
- ☐ Das Abbrechen der betroffenen Prozesse.

d) **Interprozesskommunikation:** Welche Aussage zu federgewichtigen Prozessen (*User-Level Threads*) stimmt?

3 Punkte

- ☐ Ein Fehler in einem *User-Level Thread* kann einen ganzen Prozess (Task) blockieren.
- ☐ Eine Verwendung erfordert die Bereitstellung spezifischer Systemcalls durch das Betriebssystem.
- ☐ Jeder *User-Level Thread* erhält bei der Erstellung eine eigene Prozess-ID.
- ☐ Kontextwechsel werden über einen *Trap* im Kern realisiert.

Aufgabe 2: Threads und Scheduling (12 Punkte)

a) Programmverständnis

6 Punkte

Welche Ausgabe erzeugt das folgende C-Programm?

Hinweis: Header-Dateien und Teile der Fehlerbehandlung fehlen. Nehmen Sie einen fehlerfreien Ablauf an. Jede Ausgabe wird sofort sichtbar; Ausgabepufferung spielt keine Rolle.

```
1      int x = 10;
2      int y[] = {3, 5, 7};
3
4      int get_x() {
5          return x;
6      }
7
8      void* work(void* arg) {
9          int* p = (int*)arg;
10         *(p + 1) = *p + x;
11         *(p + 2) = *(p + 1) - *(p + 0);
12         printf("%d|%d|%d|",
13             *p,
14             *(p+1),
15             *(p+2)
16         );
17         return NULL;
18     }
19
20     int main() {
21         pthread_t thread;
22         static int x = 42;
23         pthread_create(&thread, NULL, &work, (void*)y);
24         pthread_join(thread, NULL);
25         printf("%d|%d|%d|%d|%d",
26             x,
27             get_x(),
28             y[0],
29             y[1],
30             y[2]
31         );
32         return 0;
33     }
```

Ausgabe:

b) Scheduling-Verfahren

6 Punkte

Gegeben sind vier zyklische Prozesse P1 bis P4. Jeder Prozess führt wiederholt einen CPU-Stoß und anschließend einen E/A-Stoß aus. Nach Abschluss des E/A-Stoßes ist er wieder bereit.

Es gibt genau eine CPU; E/A-Vorgänge können parallel laufen. Prozesswechsel benötigen keine Zeit. Bei nicht anderweitig festgelegter Gleichheit gilt P1 vor P2 vor P3 vor P4.

Ergänzen Sie die folgenden beiden Scheduling-Verfahren die Gantt-Diagramme ab Zeitpunkt 6 (die ersten sechs Zeiteinheiten sind vorgegeben). Kennzeichnen Sie für jeden Prozess die Zustände *Running*, *Ready*, *Blocked* und *Not Arrived* (Ankunftszeit siehe Tabelle) entsprechend der Legende.

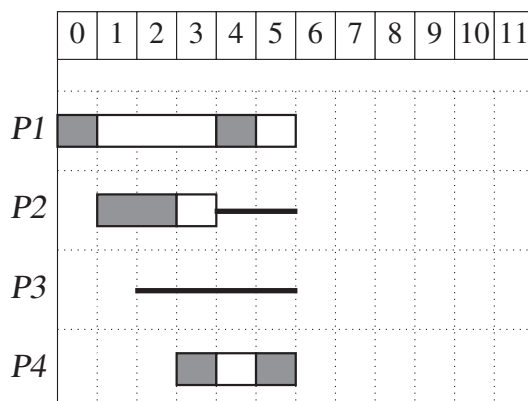
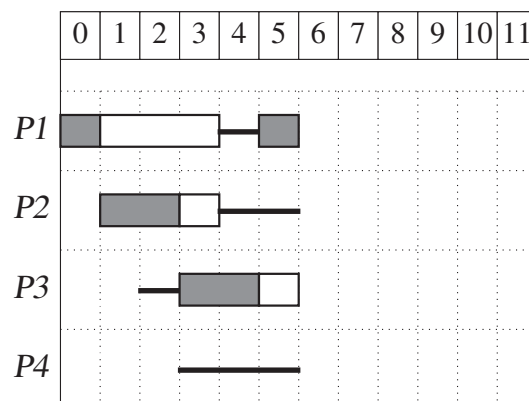
Für Korrekturen können Sie die Ersatzdiagramme verwenden.

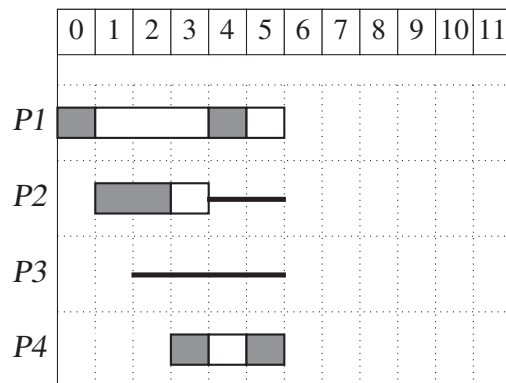
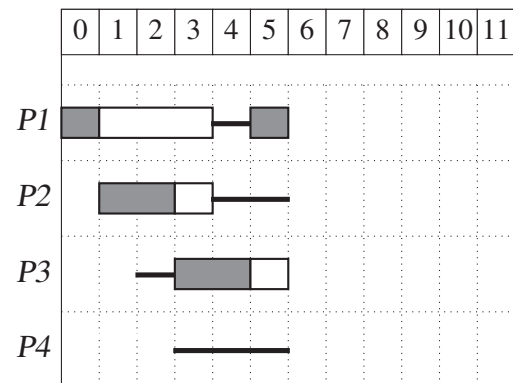
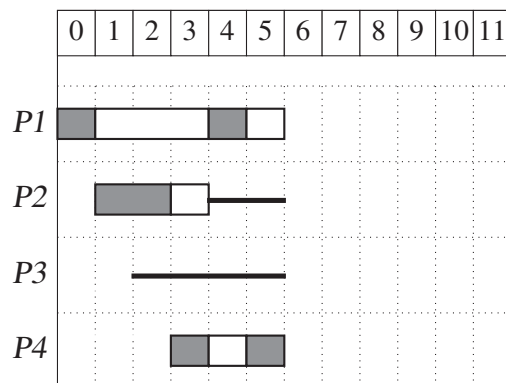
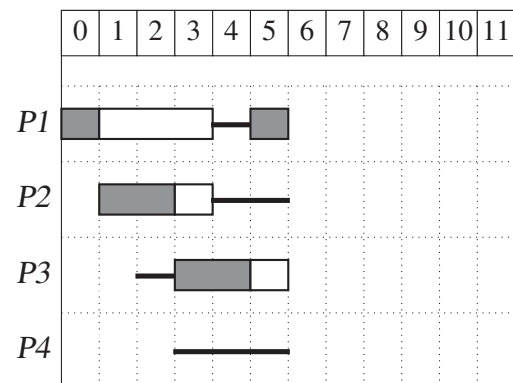
Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	1	3
P2	1	2	1
P3	2	2	2
P4	3	1	1

Hinweis: *Running* wird durch ausgefüllte Boxen, *Blocked* durch leere Boxen, *Ready* durch horizontale Striche und *Not Arrived* durch leere Flächen symbolisiert (s. Legende, rechts). Für Korrekturen **streichen Sie ungültige Lösungen deutlich durch** und nutzen Sie stattdessen die Ersatzdiagramme am Ende der Aufgabe.

Legende
<i>Running</i> 
<i>Blocked</i> 
<i>Ready</i> 
<i>Not Arrived</i>

I) Shortest Process Next (3 Punkte)

II) Virtual Round Robin (3 Punkte)
Zeitscheibe: 2 Zeiteinheiten.

Ersatzdiagramme:**Verfahren: Shortest Process Next****Verfahren: Virtual Round Robin****Zeitscheibe: 2 Zeiteinheiten.****Verfahren: Shortest Process Next****Verfahren: Virtual Round Robin****Zeitscheibe: 2 Zeiteinheiten.**

Aufgabe 3: Synchronisation (12 Punkte)

Das klassische **Cigarette Smokers Problem**¹ besteht aus drei Raucher-Threads (`smoker`) und den drei Ressourcen Tabak (`tobacco`), Zigarettenpapier (`paper`) und Streichhölzern (`match`).

Jeder Raucher-Thread besitzt bereits genau *eine* dieser Ressourcen dauerhaft. Zum Drehen einer Zigarette benötigt er daher nur die beiden jeweils anderen Ressourcen:

- der Raucher mit Tabak (`smokerWithTobacco`) benötigt `paper` und `match`,
- der Raucher mit Papier (`smokerWithPaper`) benötigt `tobacco` und `match`,
- der Raucher mit Streichhölzern (`smokerWithMatch`) benötigt `tobacco` und `paper`.

Ein Versorger-Thread (`supplier`) stellt jeweils zwei der drei Ressourcen bereit. Der Raucher, der dieses Ressourcenpaar benötigt, soll daraufhin aktiviert werden. Bevor der Versorger das nächste Ressourcenpaar bereitstellt, wartet er mittels `supplier`, bis das vorherige Ressourcenpaar verbraucht wurde.

Beispielsweise stellt er `paper` und `tobacco` wie folgt bereit:

Versorger-Thread (`supplier`), Auszug für das Paar `paper` und `tobacco`

```
1    ...
2    wait(supplier)
3    signal(paper)
4    signal(tobacco)
5    ...
```

Sobald ein Raucher das von ihm benötigte Ressourcenpaar erhalten hat, dreht er die Zigarette und gibt anschließend den Versorger frei. Das eigentliche Rauchen kann parallel zu den folgenden Runden erfolgen.

Für den Raucher mit Streichhölzern gilt beispielsweise:

Raucher mit Streichhölzern (`smokerWithMatch`)

```
1    wait(smokerWithMatch)
2    makeCigarette()
3    signal(supplier)
4    smoke()
```

Vorgabe: Die Implementierungen des Versorgers und der Raucher dürfen nicht verändert werden. Insbesondere kann der Versorger die Raucher nicht direkt aktivieren.

Der naive Ansatz, bei dem die Raucher-Threads einfach nebenläufig auf ihre beiden benötigten Ressourcen warten, kann zu einer Verklemmung führen: Ähnlich wie beim Dining-Philosophers-Problem können Raucher jeweils eine Ressource erhalten und anschließend auf die zweite warten.

Eine bekannte Lösung verwendet daher drei zusätzliche Pusher-Threads (`pusher`). Jeder Pusher wartet auf eine der drei Ressourcen und koordiniert sich über gemeinsame Zustandsvariablen mit den anderen Pushern. Sobald ein vollständiges Ressourcenpaar vorliegt, weckt er den passenden Raucher durch seine eigene Semaphore (`smokerWith...`).

Wichtig: Die Pusher sind nach der Ressource benannt, auf die sie warten, und *keinem* bestimmten Raucher zugeordnet.

¹S. Patil: *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes*, Computation Structures Group Memo 57, Project MAC, MIT, Cambridge, Februar 1971.

Aufgabe:

Vervollständigen Sie den Pusher-Thread für die Ressource match. Beachten Sie dabei:

- Warten sie auf match und prüfen Sie anhand der Zustandsvariablen, ob bereits eine zweite Ressource verfügbar ist.
- Falls ja: Wecken Sie den passenden Raucher und aktualisieren Sie die Zustandsvariablen.
- Falls nein: Vermerken Sie match als verfügbar.
- Schützen Sie alle kritischen Teile der Implementierung durch gegenseitigen Ausschluss.
- *Tipp:* Der Versorger wird bereits durch die Raucher aktiviert.

Alle globalen Variablen und Semaphore

```
1 Semaphore supplier = 1 // Zum wecken des Versorgers
2
3 // Vom Versorger bereitgestellte Ressourcen / zum Wecken der Pusher:
4 Semaphore match, paper, tobacco = 0
5
6 // Von den Pushern gemeinsam genutzte Zustandsvariablen für die Ressourcen
7 bool isMatch = false
8 bool isPaper = false
9 bool isTobacco = false
10
11 Semaphore mutex = 1 // Gegenseitiger Ausschluss
12
13 // Zum wecken der Raucher
14 Semaphore smokerWithMatch = 0
15 Semaphore smokerWithPaper = 0
16 Semaphore smokerWithTobacco = 0
```

match-Pusher-Thread:

Aufgabe 4: Speicherverwaltung (12 Punkte)

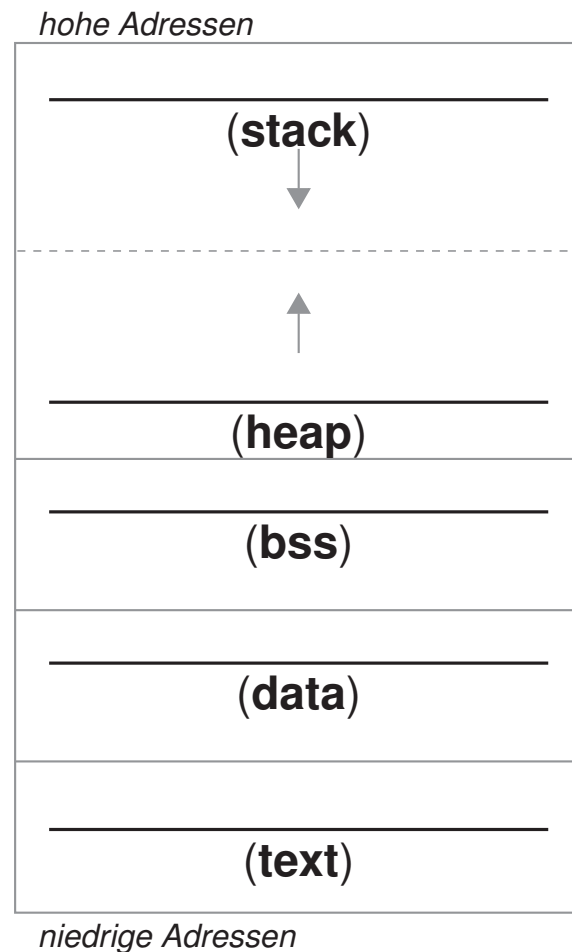
a) Programmsegmente und Variablen

4 Punkte

Ordnen Sie die Bezeichner `a`, `b`, `x`, `main` und `set` den dargestellten Speicherbereichen zu, indem Sie den Bezeichner auf die Linie des jeweiligen Speicherbereichs schreiben.

Ordnen Sie zusätzlich die Speicherstelle zu, auf die `a` in Zeile 11 zeigt (`*a`).

```
1 int* a;
2 int b = 42;
3
4 void set(int x) {
5     b = x;
6 }
7
8 int main() {
9     set(67);
10    a = &b;
11    // Wo liegt *a?
12
13    return 0;
14 }
```



b) Platzierungsstrategien

6 Punkte

Gegeben ist ein Speicher mit 32 MiB (**1 Feld = 2 MiB**). Jede Teilaufgabe enthält genau eine Speicheranforderung. Die erste Tabellenzeile zeigt die Belegung vor der Anforderung. **Füllen Sie jeweils die leere Zeile mit der Belegung danach.** Tragen Sie den tatsächlich reservierten Bereich ein und beachten Sie die angegebene Platzierungsstrategie. Bei Fit-Verfahren wird im gewählten freien Bereich an der kleinsten Adresse platziert.

Hinweise:

- Änderungen in der *vorher*-Zeile sind automatisch falsch!
- Es muss die vollständige Belegung in der *nachher*-Zeile angegeben werden.
- Für Korrekturen streichen Sie die *nachher*- und nutzen stattdessen die **Ersatz**-Zeile.
- Für **unerfüllbare** Anforderungen **kreuzen** Sie **nur** das entsprechende **Kästchen an**.

Best-Fit-Verfahren: Prozess E fordert 3 MiB an:

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
<i>vorher</i>		A			B				C	C					D	D
<i>nachher</i>																
Ersatz																

☐ Anforderung unerfüllbar!

Buddy-Verfahren: Prozess F fordert 7 MiB an:

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
<i>vorher</i>	A		B								C					D
<i>nachher</i>																
Ersatz																

☐ Anforderung unerfüllbar!

Worst-Fit-Verfahren: Prozess G fordert 2 MiB an:

	0	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
<i>vorher</i>	A	A	B						C	C					D	D
<i>nachher</i>																
Ersatz																

☐ Anforderung unerfüllbar!

c) Segmentbasierte Adressberechnung

2 Punkte

Berechnen Sie für die logischen Adressen 0x05 567063 und 0x09 315584 jeweils die physikalische Adresse mit segmentbasierter Adressabbildung. Die höchstwertigen 8 Bit der logischen Adresse wählen den Eintrag in der Segmenttabelle. Falls der Zugriff eine Zugriffsverletzung auslöst, kennzeichnen Sie dies.

Segmenttabelle:

	Startadresse	Länge
0x00	0x002B1CC9	0x28B9A0
0x01	0x23F10017	0x7A20D6
0x02	0x278B3062	0xD54669
0x03	0x31C1BACC	0x3207AF
0x04	0x42FB84E6	0xD0B23F
0x05	0x6112713C	0x62D952
0x06	0x73E112F2	0xA77CD4
0x07	0x761572A2	0x9A25E0
0x08	0x8134055B	0xB9CA23
0x09	0x8B8E9641	0x463AF0
0x0A	0x9309FE77	0x8C4B67
0x0B	0xA79F6BE8	0xF6C135
0x0C	0xA92F3BBB	0x5BB391
0x0D	0xCF52800B	0x2CF357
0x0E	0xD24618B3	0xE035D9
0x0F	0xE0F3EF30	0x81FFA5
0x10	0xECD731E0	0x9BCA81

logische Adresse: 0x05 567063→ **physikalische Adresse:** **logische Adresse:** 0x09 315584→ **physikalische Adresse:**

Aufgabe 5: E/A und Dateisysteme (12 Punkte)

a) E/A-Scheduling

6 Punkte

Gegeben sei ein Plattenspeicher mit 20 Spuren (0–19). Der Schreib-/Lesekopf befindet sich anfangs über Spur 0 und bewegt sich zunächst in Richtung steigender Spurnummern.

Der E/A-Scheduler erhält die folgenden Leseaufträge. Neue Aufträge treffen jeweils unmittelbar nach Abschluss der angegebenen Anzahl bearbeiteter Aufträge und vor der Auswahl des nächsten Auftrags ein.

Bearbeitete Aufträge	Neu eintreffende Aufträge
0 (zu Beginn)	$L_0 = \{4, 5, 8\}$
1	$L_1 = \{1, 3, 15\}$
4	$L_4 = \{10, 18\}$
7	$L_7 = \{15, 19\}$
9	$L_9 = \{2, 16\}$

Aufgabe:

Tragen Sie die Reihenfolge der gelesenen Spuren ein, wenn der E/A-Scheduler nach der **Fahrstuhlstrategie** (Elevator) arbeitet.

Für Korrekturen streichen Sie ungültige Lösungen deutlich durch und nutzen Sie stattdessen die Ersatztabelle.

--	--	--	--	--	--	--	--	--	--	--	--

Ersatztabelle:

--	--	--	--	--	--	--	--	--	--	--	--

b) UNIX-Dateirechte und Man-Pages

6 Punkte

Für das Verzeichnis `rand_dir` liefert `ls -ld rand_dir` folgende Ausgabe:

```
drwxr-xr-x 2 user group 1024 Jan 30 00:00 rand_dir
```

Das führende `d` kennzeichnet ein Verzeichnis. `user` ist der Eigentümer und `group` der zugehörige Gruppenname.

Anschließend wird folgender Befehl (**siehe `chmod(1)`**) ausgeführt:

```
chmod g+w-x,o-rx rand_dir
```

Aufgabe:

Geben Sie für **alle Nutzergruppen** explizit an, **welche Berechtigungen** bezüglich des Verzeichnisses `rand_dir` nach Ausführung des Befehls bestehen bzw. nicht bestehen.

Aufgabe 6: Programmieraufgabe: Datei mit den meisten Zeilen bestimmen (30 Punkte)

Vervollständigen Sie das Programm `dir_workers`, das die Datei mit den meisten Zeilen im aktuellen Verzeichnis mithilfe von Worker-Threads bestimmt.

Aufgabenstellung:

- Das Hauptprogramm liest das aktuelle Verzeichnis und verteilt die Einträge über einen *größenbegrenzten* Ringpuffer an `WORKER_COUNT` Worker-Threads.
- Jeder Worker (insgesamt `WORKER_COUNT` Threads) untersucht *reguläre* Dateien aus dem gemeinsamen Puffer und speichert den Dateinamen mit den meisten Zeilen in seinem `result_t`.
- Nach Beendigung aller Worker-Threads bestimmt das Hauptprogramm den globalen Gewinner und gibt ihn einmalig aus.
- Nicht reguläre Dateien werden übersprungen. Kann eine reguläre Datei nicht geöffnet werden, wird eine Fehlermeldung ausgegeben und der Auftrag übersprungen.
- Gibt es keine erfolgreich bearbeitete Datei, erfolgt keine Ausgabe. Bei gleicher maximaler Zeilenzahl darf eine beliebige der betreffenden Dateien ausgegeben werden.

Vorgaben:

Hier die Übersicht über die vorgegebenen Typen und Funktionen. Sie finden die Implementierung im Detail auf der nächsten Seite.

- `job_t`: Auftrag mit Namen eines Verzeichniseintrags; `stop != 0` kennzeichnet einen Stop-Auftrag.
- `result_t`: Thread-lokaler Gewinner mit Zeilenzahl und Gültigkeitskennzeichen `valid`.
- `buffer_init()`, `put_job()`, `get_job()` und `buffer_destroy()`: vollständig implementierter Ringpuffer der Größe `BUFFER_SIZE`.
- `is_regular_file()`: Prüft, ob ein Pfad eine reguläre Datei bezeichnet.
- `error()`: Gibt eine Fehlermeldung aus und beendet das Programm bei fatalen Fehlern.

Relevante System- und Bibliotheksaufrufe:

- Beigefügte Handbuchseiten zu: `opendir` (inkl. `readdir` und `closedir`), `fopen` (inkl. `fgetc`, `ferror` und `fclose`) sowie `pthread_create` (inkl. `pthread_join`).

Wichtige Hinweise:

- *Belegte Ressourcen* müssen im Normalfall wieder *freigegeben* werden. Ausnahme: Fehlerfall!
- Achten Sie auf korrekte Fehlerbehandlung aller System- und Bibliotheksaufrufe. `error()` darf hierfür verwendet werden.
- Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug; es geht um die semantische Umsetzung.

Vorgegebener Code:

// Gehen Sie davon aus, dass alle notwendigen Header eingebunden sind.

```
#define BUFFER_SIZE 8
```

```
#define WORKER_COUNT 3
```

```
typedef struct {  
    char name[NAME_MAX + 1]; // 256  
    int stop;  
} job_t;
```

```
typedef struct {  
    job_t job;  
    size_t lines;  
    int valid;  
} result_t;
```

```
typedef struct {  
    job_t jobs[BUFFER_SIZE];  
    size_t read_pos;  
    size_t write_pos;  
    sem_t mutex;  
    sem_t empty;  
    sem_t used;  
} buffer_t;
```

```
static buffer_t buffer;
```

// Bricht das Programm nach einem Fehler ab.

```
static void error(const char* message) {  
    perror(message);  
    exit(EXIT_FAILURE);  
}
```

// Prüft, ob path eine reguläre Datei bezeichnet.

// Rückgabe: 1 für reguläre Datei, sonst 0

```
static int is_regular_file(char* path);
```

// Initialisiert Indizes und Semaphore des Puffers.

```
static void buffer_init(buffer_t* b);
```

// Fügt einen Auftrag in den Puffer ein.

```
static void put_job(job_t job);
```

// Entnimmt einen Auftrag aus dem Puffer.

```
static job_t get_job(void);
```

// Gibt die Semaphore des Puffers frei.

```
static void buffer_destroy(buffer_t* b);
```

```
static void* worker(void* arg) {
    result_t* result = arg; // Speicher für Ergebnis
    result->valid = 0;
    size_t lines = 0; // Anzahl Zeilen
    int c; // Ein Zeichen
    // Schritt 1: Aufträge bis zum Stop-Auftrag bearbeiten

    // Schritt 2: Reguläre Datei erkennen und öffnen
```

```
// Schritt 3: Zeilen zählen
```

```
// Schritt 4: Bisher besten Auftrag merken
```

```
// Schritt 5: Datei schließen
```

```
return NULL;
```

```
}
```


// Schritt 2: Verzeichnis öffnen und Einträge lesen

// Schritt 3: Aufträge einstellen

// Schritt 4: Stop-jobs und Verzeichnis schließen

```
// Ergebnisse einsammeln und globalen Gewinner ausgeben
result_t best_result = {.lines = 0, .valid = 0};
for (size_t i = 0; i < WORKER_COUNT; i++) {
    int rc = pthread_join(tids[i], NULL);
    if (rc != 0) {
        errno = rc;
        error("pthread_join");
    }
    if (results[i].valid &&
        (!best_result.valid ||
         results[i].lines > best_result.lines)) {
        best_result = results[i];
    }
}

if (best_result.valid) {
    printf("%s:_%zu_Zeilen\n", best_result.job.name, best_result.lines);
}

buffer_destroy(&buffer);
return EXIT_SUCCESS;
}
```