Technische Universität Dortmund

Klausur „Betriebssysteme“

	erreichbare Punkte	erhaltene Punkte				
Aufgabe 1	8	**a**	**b**	**c**	**d**	
Aufgabe 2	9	**a**	**b**			
Aufgabe 3	8.5	**a**	**b**	**c**		
Aufgabe 4	6	**a**	**b**			
Aufgabe 5	6	**a**	**b**			
Aufgabe 6	22.5	**a**	**b**			
Summe	60					
Spickzettel vorhanden?						

(Matrikel-Nr.)

(Name) (Vorname)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (17 Seiten inklusive Deckblatt),
- die Kenntnisnahme der Hinweise auf Seite 2.

Dortmund, 07.08.2024

(Unterschrift)

Hinweise

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

- Es können **60 Punkte** erreicht werden; 50% der Gesamtpunktzahl sind zum Bestehen der Klausur hinreichend. Zur Bearbeitung stehen insgesamt **60 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 (8 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) **Speicher:** Welche der folgenden Aussagen beschreibt den Unterschied zwischen Seitenadressierung (Paging) und Segmentierung korrekt?

2 Punkte

- ☐ Segmentierung unterteilt den Speicher entsprechend der logischen Bereiche eines Programms in Blöcke unterschiedlicher Größe, während Paging Blöcke fester Größe nutzt.
- ☐ Paging unterteilt den Speicher entsprechend den logischen Bereichen des Programms in Blöcke unterschiedlicher Größe, während Segmentierung Blöcke fester Größe nutzt.
- ☐ Beide, Paging und Segmentierung, unterteilen den Speicher entsprechend den logischen Bereichen des Programms in Blöcke unterschiedlicher Größe.
- ☐ Beim Paging ist oftmals eine Umlagerung der Speicherblöcke erforderlich, bei der Segmentierung ist dies nicht nötig.

b) **Virtueller Speicher:** Seitenflattern (Thrashing) tritt auf, wenn . . .

2 Punkte

- ☐ . . . in einer Schleife wiederkehrend die gleichen Daten gespeichert und gelöscht werden, ohne dadurch nützliche Arbeit zu leisten.
- ☐ . . . das Betriebssystem eine hohe Anzahl von Speicheranforderungen erzeugt.
- ☐ . . . zu viele Programme gleichzeitig im System laufen.
- ☐ . . . das System überwiegend damit beschäftigt ist, Seiten zwischen dem Haupt- und dem Hintergrundspeicher auszutauschen.

c) **Systemaufrufe:** Was ist der Zweck des Systemaufrufs **exec()**?

2 Punkte

- ☐ Er beendet einen Prozess.
- ☐ Er startet ein Programm in einem neuen Prozess.
- ☐ Er ersetzt den aktuellen Prozess durch ein neues Programm.
- ☐ Er synchronisiert Prozesse.

d) **Prozesse und Fäden:** Was ist der Unterschied zwischen einem Prozess und einem Faden (Thread)?

2 Punkte

- ☐ Jeder Prozess besitzt seinen eigenen Stack (Stapelspeicher), während sich Fäden einen gemeinsamen Stack teilen.
- ☐ Ein Prozess hat einen eigenen Adressraum, während sich die Fäden eines Prozesses diesen teilen.
- ☐ Prozesse haben stets Zugriff auf den gesamten Arbeitsspeicher des Systems, Fäden hingegen nur auf einen Teil.
- ☐ Prozesse werden vom Betriebssystem verwaltet, Fäden hingegen von Benutzern.

Aufgabe 2: Prozesse und Scheduling (9 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 = 25;
int subtrahend = 10;

void do_minus(char* msg, int subtrahend){
    if (x >= 15) {
        x -= subtrahend;
        printf("%s%d,␣", msg, x);
    } else {
        printf("%s?,␣", msg);
    }
}
int main(){
    pid_t pid = fork();
    if (pid > 0){
        wait(NULL);
        do_minus("a", subtrahend);
    } else if (pid == 0){
        x = 50;
        subtrahend = 50;
        do_minus("b", 40);
    } else {
        do_minus("c", subtrahend);
        printf("Oh␣nein!\n");
    }
    do_minus("d", subtrahend);
    return 0;
}
```

Ausgabe:

---

b) Scheduling-Verfahren

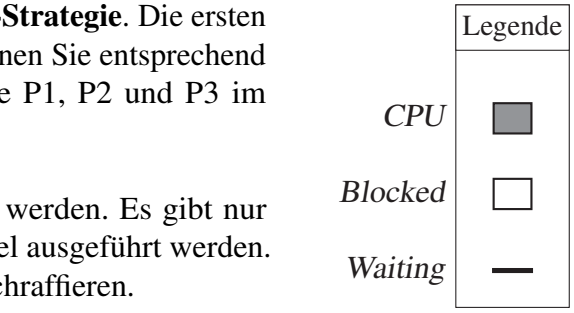
4 Punkte

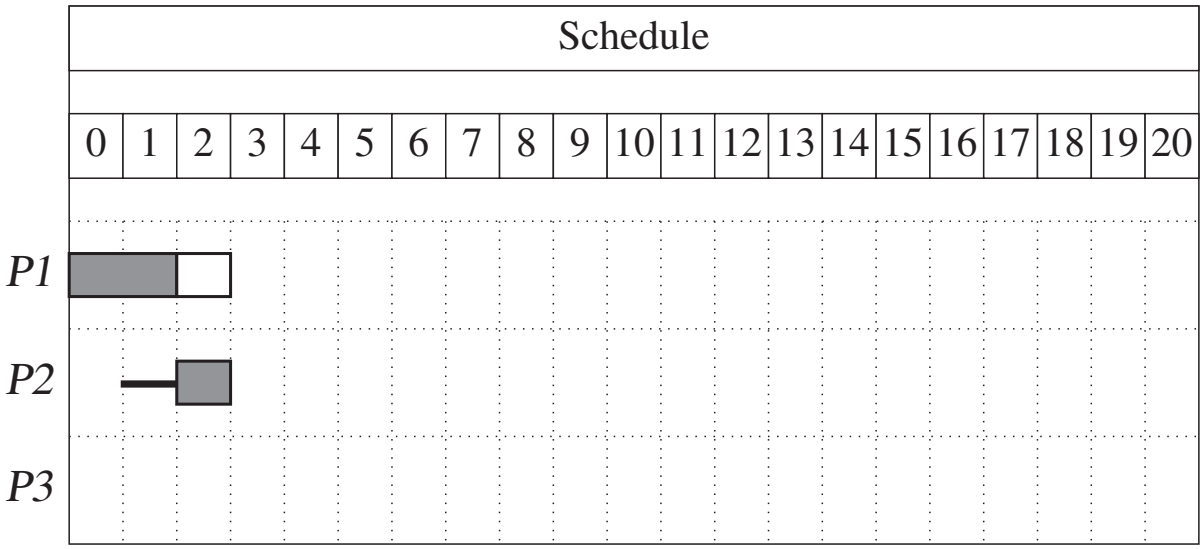
Gegeben sind drei zyklisch arbeitende Prozesse P1, P2 und P3. Die Prozesse treffen mit der in der folgenden Tabelle angegebenen Ankunftszeit ein und führen erst einen (vollständigen) CPU-Stoß und dann einen (vollständigen) E/A-Stoß aus.

Prozess	Ankunftszeit	CPU-Zeit	E/A-Zeit
P1	0	2	3
P2	1	4	4
P3	4	3	6

Das Scheduling arbeitet nach der **Shortest-Process-Next-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.





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

Aufgabe 3: Synchronisation und Verklemmungen (8.5 Punkte)

a) Race Condition

2 Punkte

Erklären Sie den Begriff “Race Condition“ (oder auch „Wettlaufsituation“) und wie sie vermieden werden kann.

Antwort:

b) Verklemmungsbedingungen

2 Punkte

Welche *drei* notwendigen und *eine* hinreichenden Bedingungen müssen erfüllt sein, damit es zu einer Verklemmung kommt?

Antwort:

c) Rangfolgen

4.5 Punkte

In Ihrem System müssen die drei Prozesse A, B und C in einer Sequenz (Verarbeitungskette) ausgeführt werden.

Hinweise:

- Die Abfolge ist: Prozess A (Produzent) → B (Verarbeitungsschritt) → C (Konsument).
- Diese Abfolge wiederholt sich (A beginnt erst nach Fertigstellung von C).
- Alle Prozesse sind zum Zeitpunkt 0 laufbereit.
- Die Prozesse nutzen gemeinsamen Speicher für die Daten.
- Die Funktionen `arbeite[A,B,C](int *)` verändern die Variable `i` und müssen sich gegenseitig ausschließen.

Aufgabenstellung:

- Nutzen Sie die drei vorgegebenen Semaphoren. Initialisieren Sie diese entsprechend (Pseudocode ist ausreichend).
- Ergänzen Sie das vorgegebene Programm um die nötigen Semaphor-Operationen.

```
/* gemeinsamer Speicher */
int i = 0;
Semaphore semA =
Semaphore semB =
Semaphore semC =
```

/* Prozess A */	/* Prozess B */	/* Prozess C */
arbeiteA(&i);	arbeiteB(&i);	arbeiteC(&i);

Aufgabe 4: Speicherverwaltung und virtueller Speicher (6 Punkte)

a) *Platzierungsstrategien*

4 Punkte

Die folgenden Tabellen zeigen die Belegung eines 16 MiB Speichers der in 1 MiB Blöcke eingeteilt ist. Zum Zeitpunkt t=0 belegt Prozess A beispielsweise 3 MiB Speicher (B und C haben bereits Speicher erhalten). Vervollständigen Sie die Speicherbelegung für t=1 bis t=4 gemäß der angegebenen Aktionen. Wenden Sie dabei die Platzierungsstrategie **Worst-Fit** an.

Hinweis: Falls eine Belegung nicht erfüllt werden kann, kennzeichnen Sie die betreffende Operation deutlich.

t	Prozess	Aktion	Größe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	...	...	...					B	B							C	C	C	
0	A	alloc	3 MiB					B	B	A	A	A				C	C	C	
1	D	alloc	3 MiB																
2	E	alloc	1 MiB																
3	B	free																	
4	F	alloc	2 MiB																

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

b) Segmentbasierte Addressberechnung

2 Punkte

Geben Sie für die logischen Adressen $\underline{0300\ 53bb}_{16}$ und $\underline{0000\ 1111}_{16}$ 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}	$FA57\ 0000_{16}$	$01\ DDDD_{16}$
02_{16}	$F00D\ 0000_{16}$	$50\ 0000_{16}$
03_{16}	$8080\ 0000_{16}$	$00\ FFFF_{16}$
...		
10_{16}	$99C0\ 0000_{16}$	$00\ FFFF_{16}$

logische Adresse: $\underline{0300\ 53bb}_{16}$

→ physikalische Adresse:

logische Adresse: $\underline{0000\ 1111}_{16}$

→ physikalische Adresse:

Aufgabe 5: Dateisysteme (6 Punkte)

a) Dateispeicherung

4 Punkte

Erklären sie was kontinuierliche Speicherung und verkettete Speicherung im Bezug auf Dateien auf einer **Festplatte** bedeuten.

Nennen sie für beide Verfahren je einen Vorteil gegenüber dem jeweils anderen Verfahren.

Kontinuierliche Speicherung

Verkettete Speicherung

Vorteil der kontinuierlichen Speicherung

Vorteil der verketteteten Speicherung

b) Dateisysteme mit Fehlererholung

2 Punkte

Nennen Sie einen Vor- und einen Nachteil von Journaled File Systems.

Vorteil:

Nachteil:

Aufgabe 6: Programmieraufgabe: Hashwerte von Dateien berechnen (22.5 Punkte)

Vervollständigen Sie das Programm `hash_dir`, welches SHA256-Hashwerte für einen übergebenen Pfad berechnet und ausgibt. Wir haben für Sie bereits die `main()` sowie eine Reihe von Hilfsfunktionen (siehe unten) vorbereitet.

Aufgabenstellung:

Implementieren Sie die folgenden Funktionen:

a) **`get_file_hash()`**, welche den SHA256-Hashwert für eine Datei berechnet.

b) **`process_path()`**, welche den übergebenen Pfad rekursiv nach Dateien durchsucht und für diese den SHA256-Hashwert ausgibt.

Details zu diesen Funktionen erhalten Sie an den passenden Stellen.

Hilfsfunktionen:

Folgende Hilfsfunktionen stehen Ihnen zur Verfügung (Details an passender Stelle):

- `dir_entry_type()`: Ermittelt den Typ eines Verzeichniseintrags (Datei, Verzeichnis).
- `print_hash()`: Gibt SHA256-Hashwert und Dateiname aus.
- `concat_paths()`: Konkateniert ein Pfadpräfix mit einem Suffix zu einem vollständigen Pfad. (*Achtung:* Alloziert Speicher)
- `error()`: Gibt eine Fehlermeldung aus und bricht das Programm ab (Fehlerfall).

Wichtige Hinweise:

- *Belegte Ressourcen* (d.h. geöffnete Datein/Verzeichnisse und allozierter Speicher) müssen im Normalfall wieder *freigegeben* werden. Ausnahme: Fehlerfall!
- Achten Sie in Ihrer Implementierung auf *korrekte Fehlerbehandlung* (siehe Handbuchseiten und unsere Dokumentation der Hilfsfunktionen). Nutzen Sie `error()` um Schreibarbeit zu sparen (Achtung: ggf. ist noch mehr zu tun).
- Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die semantische Umsetzung.
- Relevante Systemaufrufe mit beigefügten Handbuchseiten: `SHA256`, `fopen`, `fread`, `ferror`, `fclose`, `opendir`, `readdir` und `closedir`

Vorgegebene Funktionen:

```
// Gehen Sie davon aus, dass alle notwendigen Header inkludiert sind

// Kurzschreibweise Fehlerbehandlung (Programmabbruch)
void error(char* msg) {perror(msg); exit(errno);}

// Bestimmt Typ eines Verzeichniseintrags.
// Rückgabewerte:
//  1 für reguläre Datei
//  2 für Verzeichnis
//  -1 und errno gesetzt im Fehlerfall
static int dir_entry_type(char* path) {
    /* Implementierungsdetails nicht relevant */
}

// Schreibt Hashwert und Dateipfad nach stdout
static void print_hash(unsigned char* hash, char* path) {
    /* Implementierungsdetails nicht relevant */
}

// Konkateniere Prä- und Suffix zu einem Pfad.
// Speicher muss später mit free freigegeben werden.
static char* concat_paths(char* path, char* filename) {
    /* Implementierungsdetails nicht relevant */
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Mindestens ein Argument benötigt");
        return 1;
    }
    for (int i=1; i<argc; i++) {
        process_path(argv[i]);
    }
    return 0;
}
```

Die Funktion **`get_file_hash()`** berechnet den SHA256-Hash einer Datei. Gehen sie dabei wie folgt vor:

- Initialisieren sie den SHA256 Algorithmus mittels `SHA256_Init`.
- Öffnen Sie die Datei und lesen Sie deren Inhalt in 64 Byte Schritten in den Block-Puffer (`fopen`, `fread` und `fclose`).
- Lassen Sie dabei jeden neuen Block in den SHA256-Hashwert einfließen (`SHA256_Update`). Achten Sie auf die Länge des letzten Blocks!
- Speichern Sie den berechneten SHA256-Hashwert in den übergebenen Rückgabepuffer mittels `SHA256_Final`.

a) `get_file_hash`-Funktion (11 Punkte)

```
void get_file_hash(char* filepath, unsigned char* resultbuf) {
    SHA256_CTX ctx;  // Zustand der Prüfsumme
    unsigned char blockbuf[64];  // Puffer zum Einlesen
    FILE* f; // Handle zur Arbeit mit der Datei

    // 1. Schritt: SHA256-Kontext initialisieren
```

```
}
```

Diese Funktion **`process_path`** nimmt einen Pfad als Eingabeparameter. Prüfen Sie zunächst was sich hinter diesem Pfad verbirgt und gehen Sie dabei wie folgt vor:

- Falls es sich um eine Datei handelt: Berechnen (`get_file_hash`) und geben Sie den SHA256-Hashwert aus (`print_hash`).
- Falls es sich um ein Verzeichnis handelt:
  - Durchlaufen sie das Verzeichnis (`opendir`, `readdir` und `closedir`).
  - Rufen Sie für die Verzeichniseinträge *rekursiv* (`process_path`) auf.
  - Ignorieren Sie dabei alle Einträge die mit einem „.“ beginnen.
  - Beachten Sie, dass Sie stets einen vollständigen Pfad für den (rekursiven) Aufruf von (`process_path`) erzeugen (z.B. mit `concat_path`).

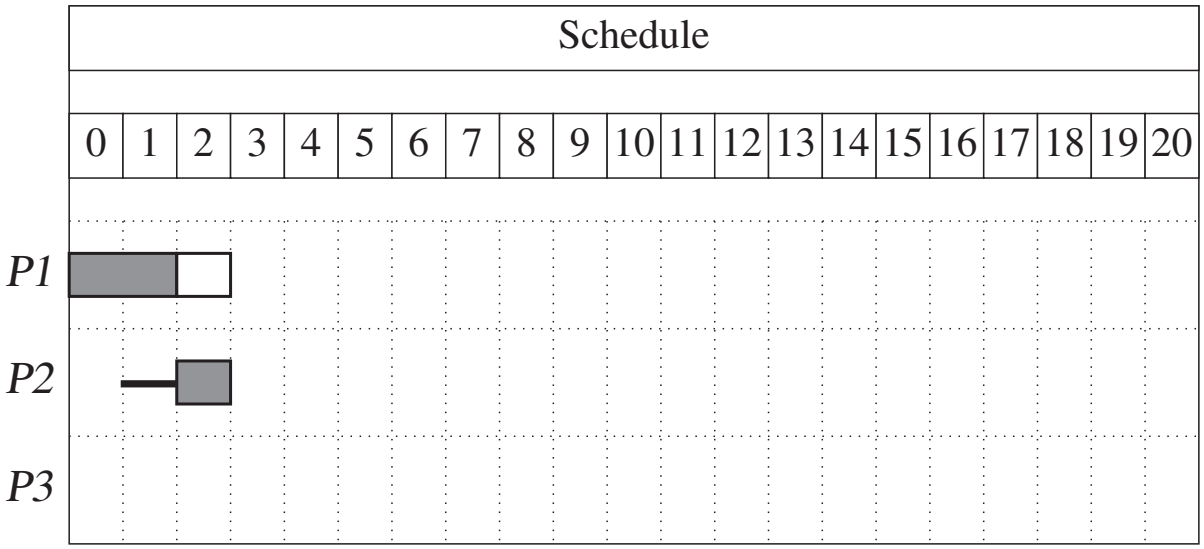
b) Funktion `process_path` (11.5 Punkte)

```
void process_path(char* path) {
    unsigned char buf[32]; // Puffer für SHA256-Prüfsumme
    DIR* dir;  // Handle zur Arbeit mit dem Verzeichnis
    struct dirent* de; // de->d_name Enthält den Dateinamen

    // 1. Schritt: Fallunterschiedung Datei / Verzeichnis / Fehler
```

```
}
```

Reserveblatt



t	Proz.	Akt.	Größe	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-1	...	...	...					B	B							C	C	C	
0	A	alloc	3 MiB					B	B	A	A	A				C	C	C	
1	D	alloc	3 MiB																
2	E	alloc	1 MiB																
3	B	free																	
4	F	alloc	2 MiB																