Technische Universität Dortmund

Klausur „Betriebssysteme“

	erreichbare Punkte	erhaltene Punkte				
Aufgabe 1	8	**a**	**b**	**c**	**d**	
Aufgabe 2	8	**a**	**b**			
Aufgabe 3	8	**a**	**b**			
Aufgabe 4	10	**a**	**b**	**c**		
Aufgabe 5	7	**a**	**b**			
Aufgabe 6	19	**a**	**b**			
Summe	60					
Spickzettel vorhanden?						

(Matrikel-Nr.)

(Name) (Vorname)

Durch meine Unterschrift bestätige ich

- den Empfang der vollständigen Klausur (16 Seiten inklusive Deckblatt),
- den Empfang der Manualseiten (zwei Blätter mit 6 Manualseiten: pthread_create/pthread_exit, pthread_join, scanf, sem_init, sem_post und sem_wait),
- die Kenntnisnahme der Hinweise auf Seite 2.

Dortmund, 02.08.2023

(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 erforderlich. 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 auf den ausgeteilten Aufgabenblättern 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 verwenden. Notieren Sie in diesem Fall aber an der für die Lösung vorgesehenen Stelle einen Verweis auf die Seite.
- Da es unterschiedliche Sprachgebräuche sowie Algorithmen- und Konzept-Ausprägungen gibt, werden hier ausdrücklich die aus Vorlesung und Übungen bekannten Ausdrucksweisen und Ausprägungen zu Grunde gelegt.

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) **Virtueller Speicher:** Was passiert auf einem x86-Linux-System, wenn ein C-Programm über einen ungültigen Zeiger versucht auf Speicher zuzugreifen? 2 Punkte

- ☐ Beim Laden des Programms wird die ungültige Adresse erkannt und der Speicherzugriff durch einen Sprung auf eine Abbruchfunktion ersetzt. Diese Funktion beendet das Programm mit der Meldung „Segmentation fault“.
- ☐ Das Betriebssystem erkennt die ungültige Adresse und leitet eine Ausnahmebehandlung ein.
- ☐ Der Übersetzer erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.
- ☐ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.

b) **Prozesse:** Welche Aussage zu Prozesszuständen ist auf einem Monoprozessorsystem richtig? 2 Punkte

- ☐ Ist zu einem Zeitpunkt kein Prozess im Zustand bereit, so ist auch kein Prozess im Zustand laufend.
- ☐ Ein Prozess im Zustand blockiert muss warten, bis der laufende Prozess den Prozessor abgibt und kann dann in den Zustand laufend überführt werden.
- ☐ Es befindet sich zu jedem Zeitpunkt maximal ein Prozess im Zustand laufend.
- ☐ In den Zustand blockiert gelangen Prozesse nur aus dem Zustand bereit.

c) **Systemaufrufe:** Welche Aussage zum Thema Systemaufrufe ist richtig? 2 Punkte

- ☐ Durch die Bereitstellung von Systemaufrufen, kann ein Benutzerprogramm das Betriebssystem um eigene Funktionen erweitern.
- ☐ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.
- ☐ Die Bearbeitung eines Systemaufrufs findet immer in dem Adressraum des aufrufenden Prozesses statt.
- ☐ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.

d) **Prozesse und Fäden:** Welche Aussage zu Prozessen und Fäden (Threads) ist richtig?

2 Punkte

- ☐ Der Aufruf von fork(2) gibt im Elternprozess die Prozess-ID des Kindprozesses zurück, im Kindprozess hingegen den Wert 0.
- ☐ Die Veränderung von Variablen und Datenstrukturen in einem mittels fork(2) erzeugten Kindprozess beeinflusst auch die Datenstrukturen im Elternprozess.
- ☐ Mittels fork(2) erzeugte Kindprozesse müssen stets vor dem Elternprozess beendet werden.
- ☐ Fäden, die mittels pthread_create(3) erzeugt wurden, besitzen jeweils einen eigenen Adressraum.

Aufgabe 2: Prozesse (8 Punkte)

a) UNIX-Systemaufrufe

6 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 = 1;
void next() {
    printf("%d,␣", x++);
}
void* five() {
    x += 5;
    printf("%d,␣", x++);
    return 0;
}
int main() {
    next();
    pid_t pid = fork();
    if (pid > 0) {
            wait(NULL);
            next();
            int x = 0;
            next();
    } else if (pid == 0) {
            pthread_t t;
            if(pthread_create(&t, NULL, &five, NULL)) {printf("Schade\n");}
            else {pthread_join(t, NULL);}
            next();
    } else {
            next();
            printf("Ohje\n");
    }
    return 0;
```

Ausgabe:

---

b) Fehlerbehandlung

2 Punkte

Wir nehmen nun an, dass unmittelbar nach Start des Programms (noch vor dem fork-Aufruf) die maximal erlaubte Prozesszahl des ausführenden Nutzers auf 1 beschränkt wird. Es darf also nur ein einziger Prozess dieses Nutzers gleichzeitig existieren. Geben Sie hier die Ausgabe an, die in diesem Szenario vom Programm ausgegeben wird.

Ausgabe:

Aufgabe 3: Synchronisation und Verklemmungen (8 Punkte)

a) Korrekte Synchronisation

5 Punkte

Im Folgenden sind C-Programmausschnitte von drei nebenläufigen Programmen gegeben, die auf einem gemeinsamen Speicher arbeiten. Die Funktionen `arbeite1(int *)` und `arbeite2(int *, int *)` verändern die referenzierten Parameter und setzen dabei voraus, dass diese während der Ausführung nicht von anderen Prozessen verändert werden (gegenseitiger Ausschluss). Die bereits initialisierten Mutex-Variablen `m1` und `m2` sind vorgegeben und sollen verwendet werden, um dies sicherzustellen. Sorgen Sie durch Einfügen von Aufrufen der Mutex-Operationen `lock(Mutex *)` und `unlock(Mutex *)` mit passendem Parameter dafür, dass der Zugriff auf den gemeinsamen Speicher synchronisiert wird. Schränken Sie dabei die Nebenläufigkeit nur so weit ein, wie gerade eben nötig. Die Funktion `andere_arbeit()` symbolisiert eine beliebige Funktion, die nicht auf die Variablen `a`, `b` oder `c` zugreift.

```
/* gemeinsamer Speicher */
int a = 0, b = 0, c = 0;
Mutex m1;
Mutex m2;
```

/* Ausschnitt von Programm 1 */	/* Ausschnitt von Programm 2 */	/* Ausschnitt von Programm 3 */
arbeite1(&a);	arbeite1(&b);	arbeite2(&b,&a);
andere_arbeit();	andere_arbeit();	andere_arbeit();
arbeite2(&a,&b);	arbeite1(&a);	arbeite2(&a,&c);

b) Synchronisationsmuster

3 Punkte

Vervollständigen Sie den Programmausschnitt gemäß der **einseitigen Synchronisierung mit gegenseitigem Ausschluss**. Platzieren Sie hierzu die entsprechenden Semaphor-Operationen (`P()` und `V()` bzw. `wait()` und `signal()`) in dem Ausschnitt und initialisieren Sie die Semaphore geeignet.
Hierbei darf nur dann ein Element aus der Warteschlange entnommen werden, wenn diese nicht leer ist. Zudem dürfen `dequeue` und `enqueue` nicht gleichzeitig ausgeführt werden. Gehen Sie davon aus, dass die Warteschlange Platz für beliebig viele Elemente bietet und vermeiden Sie Verklemmungen. Zu Beginn ist die Warteschlange leer.

```
/* gemeinsamer Speicher */

Semaphore S1 =

Semaphore S2 =
```

/* Prozess 1 */	/* Prozess 2 */
item = dequeue(); // entnehmen	enqueue(item); // einfügen

Aufgabe 4: Speicherverwaltung und virtueller Speicher (10 Punkte)

a) Programmsegmente und Variablen

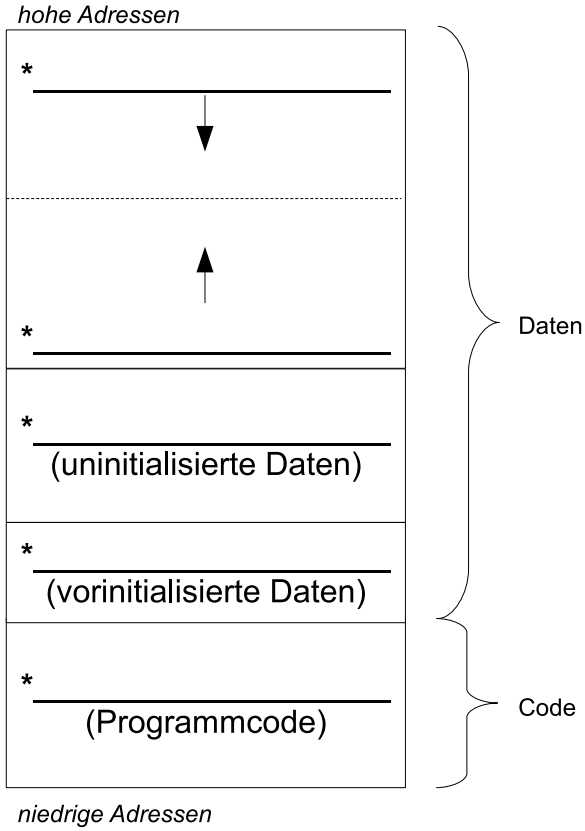
4 Punkte

Auf einem x86-System wird das untengenannte C-Programm übersetzt und gebunden. Beschriften Sie das Speicherlayout-Diagramm mit den passenden Namen der Speichersegmente (die mit * markierten Felder) und ordnen Sie mit Pfeilen die im Programm angelegten Variablen (x, a) und die Funktion (sq) den Segmenten zu.

```
int x = 1;

void sq(int a) {
  x = a * a;
}

int main(void) {
  sq(4);
  return x;
}
```



b) Segmentbasierte Addressberechnung

2 Punkte

Geben Sie für die logischen Adressen 0101_FFFF_{16} und 0201_ABC0_{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: $0101\ FFFF_{16}$

→ physikalische Adresse:

logische Adresse: $0201\ ABC0_{16}$

→ physikalische Adresse:

c) *Seitenersetzung*

4 Punkte

In einem System mit Seitenadressierung und der Seitenersetzungsstrategie *LRU* (*Least Recently Used*) tätigt ein Prozess Seitenzugriffe entsprechend folgender

Referenzfolge 4,5,1,2,3,2,5,4,3.

Das Betriebssystem sieht für diesen Prozess eine feste Anzahl von drei Hauptspeicherkacheln vor. Tragen Sie in die Tabelle unter „Hauptspeicher“ jeweils die Nummer der Seite ein, die zum gegebenen Zeitpunkt in der jeweiligen Kachel eingelagert ist. Die Felder unter „Kontrollzustände“ können Sie zum Notieren von Hilfsinformationen verwenden.

Referenzfolge		4	5	1	2	3	2	5	4	3
Hauptspeicher	Kachel 1									
	Kachel 2									
	Kachel 3									
Kontrollzustände	Kachel 1									
	Kachel 2									
	Kachel 3									

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

Aufgabe 5: Sicherheit (7 Punkte)

a) UNIX-Zugriffsrechte (5 Punkte)

I) Zugriffsrechte verstehen

3 Punkte

In einem UNIX-Dateisystem liegt eine Datei, deren Zugriffsrechte von '`ls -l`' folgendermaßen angezeigt werden:

```
-rwxr-x---  1 kepler  studi   1448 2016-08-08 14:30 eclipse.sh
```

Das Verzeichnis, in dem diese Datei liegt, ist für alle Benutzer:innen lesbar. Die Benutzerin 'juno' ist in der Gruppe 'studi', der Benutzer 'guest' dagegen nicht. Welche Benutzer:innen dürfen diese Datei lesen, welche schreiben, und welche ausführen? Schreiben Sie in jedes Tabellenfeld 'J' (ja) oder 'N' (nein), je nachdem, ob der Benutzer das jeweilige Recht besitzt.

Benutzername	**lesen**	**schreiben**	**ausführen**
juno			
kepler			
guest			

II) Zugriffsrechte abändern

2 Punkte

Ändern Sie die Zugriffsrechte der oben angegebenen Datei so ab, dass alle Benutzer:innen, die nicht die Nutzerin `juno` sind oder Mitglieder der Gruppe `studi` sind, die Datei schreiben und ausführen dürfen. Ferner sollen Mitglieder der Gruppe `studi` die Datei nur lesen dürfen. Verwenden Sie dazu die oben bereits abgebildete Notation: `-rwxr-x---`.

---

b) Begriffsdefinitionen

2 Punkte

Was ist der Unterschied zwischen „Safety“ und „Security“?

Aufgabe 6: Programmierung (19 Punkte)

Implementieren Sie in drei Teilen das Programm `numreader`, welches ganze Zahlen von der Benutzer:in einliest, an einen Thread weitergibt, welcher diese schliesslich ausgibt.

Dieser Beispielaufruf „`./numreader`“ würde die nachfolgende Ausgabe erzeugen:

```
writer: Please enter a number:7
writer: Please enter a number:8
writer: Please enter a number:-1
reader: Read value '7' from the buffer
reader: Read value '8' from the buffer
writer: Negative value. Terminating
reader: Read value '-1' from the buffer
reader: Negative value. Terminating
```

Funktion `main`

- Zunächst sollen die drei Semaphoren `mutex`, `cnt_empty` und `cnt_used` korrekt initialisiert werden. Die Bedeutung der einzelnen Semaphoren ist in den Kommentaren im Quelltext zu finden.
- Anschließend soll ein Thread erstellt werden, der die Funktion `reader_thread` ausführt.
- In einer Endlosschleife soll die Nutzer:in aufgefordert werden, eine Zahl einzugeben. Von der Standardeingabe wird diese eingelesen und in den Puffer geschrieben (siehe `put_buffer()`).
- Beim Zugriff auf den Puffer muss mittels der drei Semaphoren sichergestellt werden, dass nur ein Thread auf den Puffer zugreift und auch ein Element frei ist.
- Handelt es sich bei dem eingelesenen Wert um eine negative Zahl, so wird die Schleife verlassen.
- Zuletzt soll auf die Terminierung des oben erstellten Threads gewartet werden.

Funktion `reader_thread`

- In einer Endlosschleife wird die Zahl mit `get_buffer()` aus dem Puffer gelesen und durch die Standardausgabe ausgegeben. Auch hier muss mittels der drei Semaphoren sichergestellt werden, dass nur ein Thread auf den Puffer zugreift und auch ein Element belegt ist.
- Wird eine negative Zahl ausgelesen, so wird die Schleife verlassen und der Thread beendet.

Fehlerbehandlung: Beachten Sie, dass Fehler im gesamten Programm auftreten können und entsprechend behandelt werden müssen. Das Programm darf beim Auftreten eines Fehlers jederzeit abgebrochen werden.

Relevante Manual-Seiten: `sem_init`, `sem_wait`, `sem_post`, `pthread_create`, `pthread_join`, `scanf`

Hinweis: Einfache syntaktische Fehler (z.B. vergessene Strichpunkte) führen nicht zu Punktabzug, es geht um die semantische Umsetzung.

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

#define BUF_SIZE 20

// Sorgt für gegenseitigen Ausschluss
sem_t mutex;
// Zählt die belegten Elemente in dem Buffer
sem_t cnt_used;
// Zählt die leeren Elemente in dem Buffer
sem_t cnt_empty;

struct buffer {
  int data[BUF_SIZE];
  int read; // Aktuelle Leseposition
  int write;  // Aktuelle Schreibposition
};
struct buffer char_buf;

void put_buffer(int value) {
  char_buf.data[char_buf.write] = value;
  char_buf.write = (char_buf.write + 1) % BUF_SIZE;
}

int get_buffer(void) {
  int value;
  value = char_buf.data[char_buf.read];
  char_buf.read = (char_buf.read + 1) % BUF_SIZE;
  return value;
}
void* reader_thread(void *arg);
```

a) `main`-Funktion (11.5 Punkte)

```c
int main(int argc, const char *argv[]) {
  pthread_t thread;
  char_buf.read = 0;
  char_buf.write = 0;

```

```
  return EXIT_SUCCESS;
}
```

b) Funktion `writer_thread` (7.5 Punkte)

```
void* reader_thread(void *arg) {

}
```

Reserveblatt

Ersatztabelle für Aufgabe 4

Referenzfolge		5	1	2	3	4	3	1	5	4
Hauptspeicher	Kachel 1									
	Kachel 2									
	Kachel 3									
Kontrollzustände	Kachel 1									
	Kachel 2									
	Kachel 3									