Aufgabe 1: Ankreuzfragen (30 Punkte)

1) Einfachauswahlfragen (22 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) Wie wird erkannt, dass eine Seite eines virtuellen Adressraums gerade ausgelagert ist? 2 Punkte

- ○ Bei Programmen, die in virtuellen Adressräumen ausgeführt werden sollen, erzeugt der Compiler speziellen Code, der vor Betreten einer Seite die Anwesenheit überprüft und ggf. die Einlagerung veranlasst.
- ○ Im Seitendeskriptor wird ein spezielles Bit geführt, das der MMU zeigt, ob eine Seite eingelagert ist oder nicht. Falls die Seite nicht eingelagert ist, löst die MMU einen Trap aus.
- ○ Das Betriebssystem erkennt die ungültige Adresse vor Ausführung eines Maschinenbefehls und lagert die Seite zuerst ein bevor ein Fehler passiert.
- ○ Die MMU erkennt bei der Adressumsetzung, dass die physikalische Adresse ungültig ist und löst einen Trap aus.

b) Welche der folgenden Aussagen über Schedulingverfahren ist richtig? 2 Punkte

- ○ Bei preemptivem Scheduling sind Prozessumschaltungen unmöglich, wenn ein Prozess in einer Endlosschleife läuft.
- ○ Bei kooperativem Scheduling sind Prozessumschaltungen unmöglich wenn ein Prozess in einer Endlosschleife läuft, selbst wenn er bei jedem Schleifendurchlauf einen Systemaufruf macht.
- ○ Preemptives Scheduling ist für Mehrbenutzerbetrieb geeignet.
- ○ Kooperatives Scheduling ist für Steuerungssysteme mit Echtzeitanforderungen völlig ungeeignet.

c) Welche der folgenden Aussagen zum Thema „Aktives Warten“ ist richtig? 2 Punkte

- ○ Aktives Warten vergeudet gegenüber passivem Warten immer CPU-Zeit.
- ○ Bei verdrängenden Scheduling-Strategien verzögert aktives Warten nur den betroffenen Prozess, behindert aber nicht andere.
- ○ Aktives Warten auf andere Prozesse darf bei nicht-verdrängenden Scheduling-Strategien auf einem Monoprozessorsystem nicht verwendet werden.
- ○ Auf Mehrprozessorsystemen ist aktives Warten unproblematisch und deshalb dem passiven Warten immer vorzuziehen.

d) Welche Seitennummer (*page number*) und welcher Versatz (*offset*) gehören bei einstufiger Seitennummerierung und einer Seitengröße von 2048 Bytes zu folgender logischer Adresse: 0xba1d 2 Punkte

- ○ Seitennummer 0xb, Versatz 0xa1d
- ○ Seitennummer 0x17, Versatz 0x21d
- ○ Seitennummer 0x2e, Versatz 0x21d
- ○ Seitennummer 0xba, Versatz 0x1d

e) Welche Problematik kann durch das Philosophenproblem beschrieben werden? 2 Punkte

- ○ Ein Erzeuger und ein Verbraucher greifen gleichzeitig auf gemeinsame Datenstrukturen zu.
- ○ Exklusive Bearbeitung durch mehrere Bearbeitungsstationen.
- ○ Potenzielle Verklemmung durch eine ungünstige Anforderungsreihenfolge geteilter Betriebsmittel durch mehrere Prozesse.
- ○ Mehrere Prozesse greifen lesend und schreibend auf gemeinsame Datenstrukturen zu.

f) Welche Aussage über den Rückgabewert von `fork(2)` ist richtig? 2 Punkte

- ○ Der Rückgabewert ist in jedem Prozess (Kind und Vater) jeweils die eigene Prozess-ID.
- ○ Der Kind-Prozess bekommt die Prozess-ID des Vater-Prozesses.
- ○ Im Fehlerfall wird im Kind-Prozess -1 zurückgeliefert.
- ○ Dem Vater-Prozess wird die Prozess-ID des Kind-Prozesses zurückgeliefert.

g) Welche Aussage über Funktionen der `exec(3)`-Familie ist richtig? 2 Punkte

- ○ Dem Vater-Prozess wird die Prozess-ID des neu erzeugten Kind-Prozesses zurückgeliefert.
- ○ Beim Aufruf von `exec()` wird das im aktuellen Prozess laufende Programm durch das angegebene Programm ersetzt.
- ○ Nach einem erfolgreichen Aufruf von `exec()` kann weiterhin auf Datenstrukturen im Adressraum des Aufrufers zugegriffen werden.
- ○ Der an `exec()` übergebene Funktionszeiger wird durch einen neuen Thread im aktuellen Prozess ausgeführt.

h) Welche Aussage über Schedulingverfahren ist richtig?

2 Punkte

- ○ Bei kooperativem Scheduling kann es zur Monopolisierung der CPU kommen.
- ○ Round-Robin bevorzugt E/A-intensive Prozesse zu Gunsten von rechenintensiven Prozessen.
- ○ Der Konvoieffekt kann bei kooperativen Schedulingverfahren wie First-Come-First-Served nicht auftreten.
- ○ Beim Einsatz preemptiver Schedulingverfahren kann laufenden Prozessen die CPU nicht entzogen werden.

i) Welche der folgenden Aussagen zum Thema Threads und Prozesse ist richtig?

2 Punkte

- ○ Zu jedem Thread (*Light-weight Process*) gehört ein eigener isolierter Adressraum.
- ○ Threads (*Light-weight Processes*) teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- ○ User-level Threads (*Feather-weight Processes*) blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ○ Die Umschaltung von User-level Threads (*Feather-weight Processes*) ist eine privilegierte Operation und muss deshalb im Systemkern erfolgen.

j) Was versteht man unter der Second-Chance- (oder Clock-) Policy?

2 Punkte

- ○ Eine Seitenersetzungsstrategie, bei der jeweils die älteste Seite ausgelagert wird.
- ○ Eine Speicherallokationsstrategie, bei der im Fehlerfall ein zweiter Allokationsversuch stattfindet.
- ○ Eine Seitenersetzungsstrategie, die mit Hilfe eines Referenz-Bits eine einfacher zu implementierende Annäherung an LRU realisiert.
- ○ Eine Scheduling-Strategie, bei der Prozesse vor der Verdrängung eine zweite Chance erhalten.

k) Was versteht man unter virtuellem Speicher?

2 Punkte

- ○ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ○ Unter einem virtuellen Speicher versteht man einen physikalischen Adressraum, dessen Adressen durch eine MMU vor dem Zugriff auf logische Adressen umgesetzt werden.
- ○ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- ○ Speicher, der einem Prozess durch entsprechende Hardware (MMU) und durch Ein- und Auslagern von Speicherbereichen vorgespiegelt wird, aber möglicherweise größer als der verfügbare physikalische Hauptspeicher ist.

2) Mehrfachauswahlfragen (8 Punkte)

Bei den Mehrfachauswahlfragen in dieser Aufgabe sind jeweils m Aussagen angegeben, davon sind n ($0 \leq n \leq m$) Aussagen richtig. Kreuzen Sie alle richtigen Aussagen an.
Jede korrekte Antwort in einer Teilaufgabe gibt einen Punkt, jede falsche Antwort einen Minuspunkt. Eine Teilaufgabe wird minimal mit 0 Punkten gewertet, d. h. falsche Antworten wirken sich nicht auf andere Teilaufgaben aus.

Wollen Sie eine falsch angekreuzte Antwort korrigieren, streichen Sie bitte das Kreuz mit drei waagrechten Strichen durch (☒).

Lesen Sie die Frage genau, bevor Sie antworten.

a) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet sind richtig?

4 Punkte

- ☐ UNIX-Prozesse sind hierarchisch organisiert.
- ☐ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.
- ☐ Ein Programm kann durch mehrere Prozesse gleichzeitig ausgeführt werden.
- ☐ Der UNIX-Systemaufruf `fork(2)` erzeugt, von wenigen Aspekten abgesehen, eine Kopie des aufrufenden Prozess.
- ☐ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ☐ Ein Prozess kann mithilfe von Threads mehrere Programme gleichzeitig ausführen.
- ☐ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- ☐ Der Compiler erzeugt aus einer oder mehreren Objekt-Dateien (Modulen) einen Prozess.

b) Welche der folgenden Aussagen zum Thema UNIX-Signale sind richtig?

4 Punkte

- ☐ UNIX-Signale sind stets ein Hinweis auf ein kritisches Problem, das zwingend zur Beendigung des aktuell laufenden Programms führen muss.
- ☐ Durch Signale können Nebenläufigkeitsprobleme in grundsätzlich nicht-parallelen Programmen entstehen.
- ☐ Signale können dazu führen, dass blockierende Systemaufrufe mit der errno `EINTR` abgebrochen werden.
- ☐ Signale haben keine praktische Relevanz, da neben der Signalnummer keinerlei weitere Nutzinformation übermittelt werden kann.
- ☐ Geräte nutzen UNIX-Signale, um der aktuell laufenden Anwendung oder dem Betriebssystem das Vorliegen neuer Ereignisse mitzuteilen.
- ☐ Programme können für die meisten Signale eine eigene Funktionen zur deren Behandlung bereitstellen.
- ☐ UNIX erlaubt es nicht, Signale an blockierte Prozesse zuzustellen, da dies dazu führen würde, dass wichtige Systemaufrufe unterbrochen würden.
- ☐ Signale, die an bereite, aber nicht laufende Prozesse zugestellt werden, werden abgearbeitet sobald der Prozess die CPU zugeteilt bekommt.

Aufgabe 2: parrots (60 Punkte)

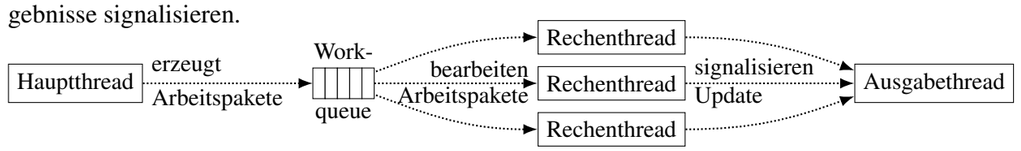
Sie dürfen diese Seite zur besseren Übersicht bei der Programmierung heraustrennen!

Schreiben Sie ein Programm `parrots` (**par**allel **ro**w-based **t**riangle **s**ummer), das zeilenweise Koordinaten von Dreiecken aus einer per Befehlszeilenargument übergebenen Datei einliest und die Anzahl der ganzzahligen Koordinaten auf den Kanten und innerhalb der Dreiecke berechnet.

Beispielhafter Aufruf von `parrots`:

```
chris@host:~$ ./parrots triangles.txt
```

`parrots` liest zeilenweise Dreiecke im Format *(x1,y1),(x2,y2),(x3,y3)* ein. Zeilen, die eine maximale Länge von 1024 Zeichen (`MAX_LINE`, exklusive `'\n'/'\0'`) überschreiten, oder nicht dem erwarteten Format entsprechen, werden dabei unter Ausgabe einer entsprechenden Warnmeldung ignoriert. Die Funktionen zur Umwandlung der eingelesen Zeile in ein **`struct triangle`** und zur Berechnung der Punkte sind im Modul `triangle.o` **vorgegebenen** (siehe `parseTriangle()` und `countPoints()` in der angehängte Manpage `triangle(3)`).
Der aktuelle Zwischenstand der Berechnungen wird regelmäßig aktualisiert und ausgegeben.
Zur Steigerung der Berechnungsgeschwindigkeit werden die Berechnungen an `CALC_THREADS` Rechenthreads ausgelagert, die einem dedizierten Ausgabethread das Vorliegen neuer Zwischenergebnisse signalisieren.



Struktur des Programms:

- Der *Hauptthread* initialisiert die benötigten Datenstrukturen und erzeugt `CALC_THREADS` Rechenthreads und den einen Ausgabethread. Dann liest er zeilenweise die per Befehlszeilenargument übergebene Datei ein, erzeugt Arbeitspakete in Form von **`struct triangle`**s und fügt die Arbeitspakete in die Workqueue ein. Nachdem alle Dreiecke eingelesen und abgearbeitet wurden, wartet er auf die Beendigung **aller** Threads, gibt alle allokierte Ressourcen frei und beendet sich.
- Die *Rechenthreads* (in obiger Illustration: 3), entnehmen jeweils Dreiecke aus der Workqueue und führen die eigentliche Zählung der Punkte mittels der vorgegebenen Funktion `countPoints()` durch. Sobald `countPoints()` die berechneten Werte liefert, signalisiert der Rechenthread dem Ausgabethread das Vorliegen neuer Werte.
- Der *Ausgabethread* gibt nach Signalisierung durch die Rechenthreads die akkummulierte Anzahl aller *interior* und *boundary* Koordinaten (siehe `triangle(3)`) auf `stdout` aus.

Die Kommunikation zwischen den Threads soll mittels modulglobaler Variablen geschehen. Es ist keine Fehlerbehandlung für die Ausgabe auf `stdout` und `stderr` nötig.

Ein zentraler Bestandteil dieser Aufgabe ist die **korrekte Synchronisation** mithilfe der vorgegebenen, **passiv wartenden Semaphor-Implementierung** – aktives Warten ist nicht erlaubt. Langsame Funktionen (z.B. `printf(3)`) dürfen nicht in kritischen Abschnitten ausgeführt werden.

Hinweise:

- Speichern Sie den Wert `NULL` in der Queue um die Arbeiterthreads zuverlässig zu beenden.
- Gehen Sie davon aus, dass alle Operationen der Queue ohne weitere Synchronisation nebenläufig genutzt werden können.
- Ihnen steht das aus der Übung bekannte Semaphor-Modul zur Verfügung. Die Schnittstelle finden Sie im folgenden Programmgerüst nach den **`#include`**-Anweisungen.
- Eine Beschreibung der Schnittstelle der Queue und der Funktionen `parseTriangle()` und `countPoints()` finden Sie in den angehängten Manpages.

```c
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <inttypes.h>

#include "sem.h"
#include "triangle.h"
#include "queue.h"

/* Funktionen aus sem.h */
SEM *semCreate(int initVal); // sets errno on failure
void semDestroy(SEM *sem);
void P(SEM *sem);
void V(SEM *sem);

// Funktionen triangle.h (parseTriangle, countPoints): SIEHE MANPAGE
// Funktionen queue.h (qCreate, qPut, qGet, qDestroy): SIEHE MANPAGE

static const size_t MAX_LINE     = 1024;
static const size_t CALC_THREADS = 5;

static void die(const char message[]) {
  perror(message); exit(EXIT_FAILURE);
}

static void usage(void) {
  fprintf(stderr, "Usage: ./parrots <file>\n");  exit(EXIT_FAILURE);
}
```

// Funktions- & Strukturdekl., globale Variablen, etc.

// Hauptfunktion (main)

if(argc != 2) { usage(); }

// Initialisierung

// Threads starten

// Arbeitspakete aus der Datei auslesen

```
  // "Haupt"schleife
```



```
  // Threads + Ressourcen aufräumen
```

```
// Ende Hauptfunktion
```

M:

```
// Funktion Rechenthread
```

```
// Ende Rechenthread
```

R:

```
// Ausgabethread
```

```
// Ende Ausgabethread
```

P:

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `parrots` unterstützt werden, welches das Programm `parrots` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. `parrots.o`) zurück. Gehen Sie davon aus, dass die vorgegebenen Module `sem.o`, `queue.o` und `triangle.o` stets vorliegen und daher nicht erzeugt werden müssen.

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `parrots` löschen.

Definieren und nutzen Sie dabei die Variablen `CC` und `CFLAGS` konventionskonform. Achten Sie darauf, dass das Makefile ohne eingebaute Variablen und Regeln (Aufruf von `make -Rr`) funktioniert!

Mk:

Aufgabe 3: Adressräume & Freispeicherverwaltung (23 Punkte)

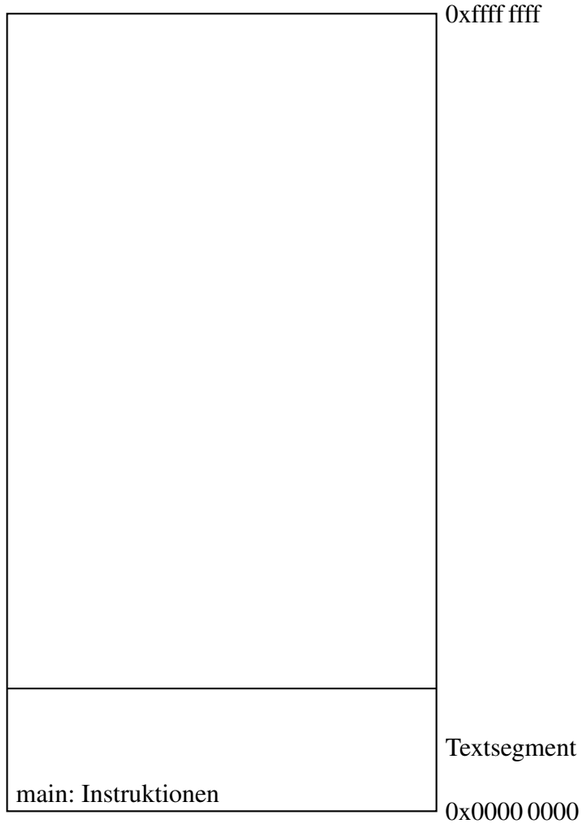
1) Gegeben sei das nachfolgende Programm. Skizzieren Sie den Aufbau des logischen Adressraums eines Prozesses, der dieses Programm ausführt. Tragen Sie die Segmente und deren Namen (analog zum schon vorgegebenen Textsegment) in unten stehende Zeichnung ein. Unterscheiden Sie hierbei die Bereiche zur Speicherung von initialisierten und nicht initialisierten Variablen. Zeichnen Sie für jede Variable ein, wo diese ungefähr im logischen Adressraum zu finden sein wird und welchen Wert sie enthält. Illustrieren Sie im Falle von Zeigervariablen mittels Pfeil, auf welches Datum die Variable jeweils zeigt.

Vermerken Sie zudem, in welche Richtung Segmente variabler Größe wachsen. Gehen Sie hierbei von einem x86-System aus. (8 Punkte)

```
static int a = 3;
static int b = 0;

const char *s = "Hello World\n";
static int t = 0;

int main(void) {
    int g = 5;
    static int h = 12;
    void *arr = malloc(5);
    int (*func)(void) = main;
}
```

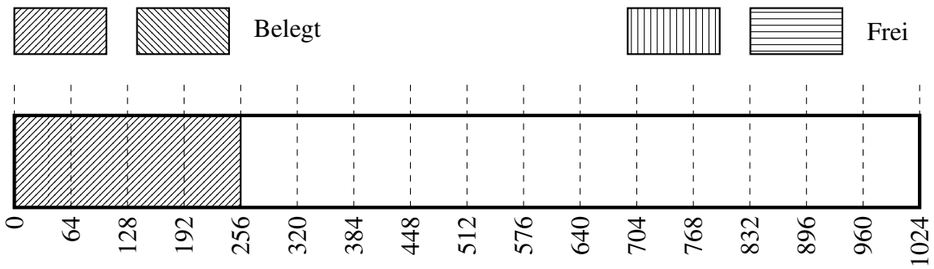


2) Ein in der Praxis häufig eingesetztes Verfahren zur Verwaltung von freiem Speicher ist das *Buddy*-Verfahren.

Nehmen Sie einen Speicher von 1024 Bytes an und gehen Sie davon aus, dass die Freispeicher-Verwaltungsstrukturen separat liegen. Initial ist bereits ein Datenblock der Größe 256 Bytes vergeben worden. Ein Programm führt nacheinander die im folgenden Bild angegebenen Anweisungen aus. (11 Punkte)

⓪ p0 = malloc(200); // 0 (initial vergebener Block)
① p1 = malloc(32);
② p2 = malloc(120);
③ p3 = malloc(60);
④ free(p2);
⑤ p4 = malloc(250);
⑥ free(p1);
⑦ free(p3);

Tragen Sie hinter den obigen Anweisungen jeweils ein, welches Ergebnis die `malloc()` - Aufrufe zurückliefern. Skizzieren Sie in der folgenden Grafik, wie der Speicher **nach Schritt ⑤** aussieht.



Tragen Sie in unten stehender Tabelle die **Adressen** der freien Blöcke (*left-over holes*) nach **jedem** Schritt ein. Für Blöcke gleicher Größe schreiben Sie die Adressen nebeneinander in die Tabellenzeile (es ist nicht notwendig, verkettete Buddys wie in der Vorlesung beschrieben einzutragen).

Größe der Blocks	initial ⓪	①	②	③	④	⑤	⑥	⑦
2^5								
2^6								
2^7								
2^8	256							
2^9	512							
2^{10}								

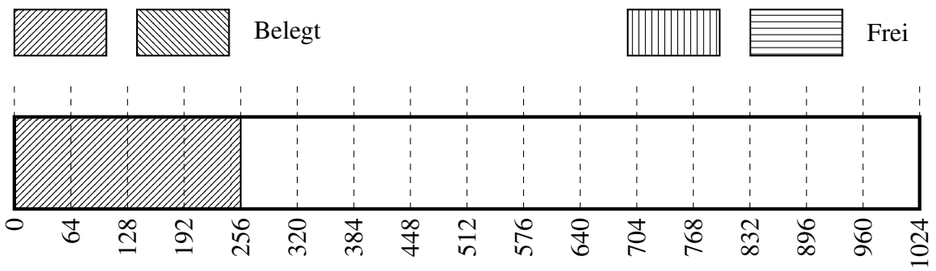
freier Block der Größe 2^9 ab Adresse 512

Hinweis: $2^5 = 32$, $2^6 = 64$, $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, $2^{10} = 1024$

3) Man unterscheidet bei Adressraumkonzepten und bei Zuteilungsverfahren zwischen externer und interner Fragmentierung. Beschreiben Sie beide Arten der Fragmentierung und erklären Sie, ob diese bei der Anwendung des Buddy-Verfahren auftritt. (4 Punkte)

Ersatzgrafik für Teilaufgabe 3.2.

Sie dürfen diese Grafik nutzen, falls Sie sich beim Ausfüllen der Grafik in 3.1. verzeichnet haben. Markieren Sie eindeutig, welche der Grafiken zur Bewertung herangezogen werden soll!



Aufgabe 4: Prozesszustände (7 Punkte)

Beschreiben Sie die Prozesszustände bei der Einplanung von Prozessen sowie die Ereignisse, die jeweils zu Zustandsübergängen führen (Skizze mit kurzer Erläuterung der Zustände und Übergänge).