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) Welche Aussage zum Thema Adressräume ist richtig? 2 Punkte

- ○ Ein Programm kann ohne weiteres auf den Inhalt logischer Adressräume anderer Prozesse zugreifen.
- ○ Virtuelle Adressräume sind Voraussetzung für die Realisierung logischer Adressräume.
- ○ Die Größe eines virtuellen Adressraums darf die Größe des vorhandenen Hauptspeichers nicht überschreiten.
- ○ Wird in einem System die Nutzung virtueller Adressräume unterstützt, so können Teile des Arbeitsspeichers in den Hintergrundspeicher (swap) ausgelagert sein.

b) Welche Aussage zu Zeigern ist richtig? 2 Punkte

- ○ Zeiger können in C nicht als Parameter an Funktionen übergeben werden.
- ○ Der Compiler erkennt bei der Verwendung eines ungültigen Zeigers die problematische Code-Stelle und generiert Code, der zur Laufzeit die Meldung „Segmentation fault“ ausgibt.
- ○ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet werden.
- ○ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.

c) Welche Aussage zu Prozessen und Threads ist richtig? 2 Punkte

- ○ Threads, die mittels `pthread_create()` erzeugt wurden, besitzen jeweils einen eigenen Adressraum.
- ○ Mittels `fork()` erzeugte Kindprozesse können in einem Multiprozessor-System nur auf dem Prozessor ausgeführt werden, auf dem auch der Elternprozess ausgeführt wird.
- ○ Die Veränderung von Variablen und Datenstrukturen in einem mittels `fork()` erzeugten Kindprozess beeinflusst auch die Datenstrukturen im Elternprozess.
- ○ Der Aufruf von `fork()` gibt im Elternprozess die Prozess-ID des Kindprozesses zurück, im Kindprozess hingegen den Wert 0.

d) Welche Aussage zum Thema Systemaufrufe ist richtig? 2 Punkte

- ○ Durch einen Systemaufruf wechselt das Betriebssystem von der Systemebene auf die Benutzerebene, um unprivilegierte Operationen ausführen zu können.
- ○ Nach der Bearbeitung eines beliebigen Systemaufrufes ist es für das Betriebssystem nicht mehr möglich, zu dem Programm zu wechseln, welches den Systemaufruf abgesetzt hat.
- ○ Benutzerprogramme dürfen keine Systemaufrufe absetzen, diese sind dem Betriebssystem vorbehalten.
- ○ Mit Hilfe von Systemaufrufen kann ein Benutzerprogramm privilegierte Operationen durch das Betriebssystem ausführen lassen, die es im normalen Ablauf nicht selbst ausführen dürfte.

e) Welche Aussage über den Linux $\mathcal{O}(1)$-Scheduler ist richtig? 2 Punkte

- ○ Alle vom $\mathcal{O}(1)$-Scheduler genutzten Datenstrukturen werden zwischen allen CPU-Kernen geteilt, um die maximale Performance zu erreichen.
- ○ Der Linux $\mathcal{O}(1)$-Scheduler wurde 2002 von einem $\mathcal{O}(n)$-Scheduler abgelöst.
- ○ Der $\mathcal{O}(1)$-Scheduler nutzt Bitmaps zur Abbildung der Prozessprioritäten und verkettete Listen zum Speichern der Prozessstrukturen und ermöglicht so ein Scheduling mit konstantem Laufzeitaufwand.
- ○ Der $\mathcal{O}(1)$-Scheduler unterstützt keine Prozessprioritäten, da die Umsetzung von Prioritäten (rechen-)aufwändig ist.

f) Welche Aussage zu Semaphoren ist richtig? 2 Punkte

- ○ Die V-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ○ Die P-Operation eines Semaphors erhöht den Wert des Semaphors um 1 und deblockiert gegebenenfalls wartende Prozesse.
- ○ Ein Semaphor kann nur zur Signalisierung von Ereignissen, nicht jedoch zum Erreichen gegenseitigen Ausschlusses verwendet werden.
- ○ Die V-Operation eines Semaphors kann ausschließlich von einem Thread aufgerufen werden, der zuvor mindestens eine P-Operation auf dem selben Semaphor aufgerufen hat.

g) Welche Aussage bezüglich der Seitenersetzungsstrategie Least Recently Used (LRU) ist richtig? 2 Punkte

- ○ Als Auswahlkriterium für die Ersetzung einer Seite wird die Zeit seit dem letzten Zugriff auf die Seite verwendet.
- ○ Die LRU Strategie gewährleistet, dass immer die Seiten eingelagert sind, auf die in der Zukunft zugegriffen wird.
- ○ Die LRU Strategie benötigt ein Referenzbit in jedem Eintrag der Seitenkacheltabelle.
- ○ Zur Implementierung von LRU benötigt man eine sehr genaue Systemuhr.

h) Welche Aussage bezüglich Seitenersetzungsstrategien ist richtig?

2 Punkte

- ○ Die FIFO-Anomalie tritt nur bei der FIFO-Ersetzungsstrategie auf.
- ○ In den meisten Betriebssystemen wird die OPT-Strategie verwendet, da sie die besten Ergebnisse liefert.
- ○ FIFO-Anomalie bedeutet, dass ein größerer Hauptspeicher (mit mehr Speicherkacheln) immer zu einem schlechteren Verhalten führt.
- ○ Die LRU-Strategie (Least Recently Used) versucht immer die Seite ersetzt wird, welche am längsten unbenutzt war.

i) Sie kennen den Translation-Look-Aside-Buffer (TLB). Welche Aussage ist richtig?

2 Punkte

- ○ Der TLB puffert die Ergebnisse der Abbildung von physikalischen auf logische Adressen, sodass eine erneute Anfrage sofort beantwortet werden kann.
- ○ Der TLB verkürzt die Zugriffszeit auf den physikalischen Speicher, da ein Teil des Speichers in einem sehr schnellen Pufferspeicher vorgehalten wird.
- ○ Der TLB ist eine schnelle Umsetzeinheit der MMU, die logische in physikalische Adressen umsetzt.
- ○ Wird eine Speicherabbildung im TLB nicht gefunden, wird der auf den Speicher zugreifende Prozess mit einer Schutzraumverletzung (Segmentation Fault) abgebrochen.

j) Welche Aussage zu Interrupts ist richtig?

2 Punkte

- ○ Bei der mehrfachen Ausführung eines unveränderten Programms mit gleicher Eingabe treten Interrupts immer an den gleichen Stellen auf.
- ○ Eine Signalleitung teilt dem Prozessor mit, dass er den aktuellen Prozess anhalten und auf das Ende der Unterbrechung warten soll.
- ○ Mit einer Signalleitung wird dem Prozessor eine Unterbrechung angezeigt. Der Prozessor sichert den aktuellen Zustand bestimmter Register, insbesondere des Programmzählers, und springt eine vordefinierte Behandlungsfunktion an.
- ○ Durch eine Signalleitung wird der Prozessor veranlasst, die gerade bearbeitete Maschineninstruktion abzubrechen.

k) Gegeben seien die folgenden Präprozessor-Makros:

2 Punkte

```
#define SUB(a, b) a - b
#define MUL(a, b) a * b
```

Was ist das Ergebnis des folgenden Ausdrucks? `4 * MUL ( SUB(3,5), 2)`

- ○ 2
- ○ -2
- ○ -16
- ○ 16

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) Konzeptionell ist der Speicher eines UNIX-Prozesses in Text-, Daten- und Stack-Segment untergliedert. Welche der folgenden Aussagen treffen zu?

4 Punkte

- ☐ Variablen der Speicherklasse *static* liegen im Daten-Segment.
- ☐ Vor Ausführung einer Funktion wird das Daten-Segment vergrößert, um den Speicher für lokale Variablen zu reservieren.
- ☐ Zeigervariablen können auf Daten aus allen Segmenten verweisen.
- ☐ Das Text-Segment enthält sowohl den Programmcode als auch konstante Zeichenketten.
- ☐ Lokale *„automatic“* Variablen einer Funktion werden im Stack-Segment abgelegt.
- ☐ Dynamisch allozierte Zeichenketten werden in das Text-Segment gelegt.
- ☐ Variablen der Speicherklasse *„automatic“* werden durch den Übersetzer mit dem Wert 0 initialisiert.
- ☐ Lokale Variablen der Speicherklasse *static* werden beim Betreten der zugehörigen Funktion neu initialisiert.

b) Welche der folgenden Aussagen zum Thema Threads und Prozesse sind richtig?

4 Punkte

- ☐ Die Umschaltung von User-level Threads (*Feather-weight Processes*) ist eine privilegierte Operation und muss deshalb im Systemkern erfolgen.
- ☐ User-level Threads (*Feather-weight Processes*) blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ☐ Threads (*Light-weight Processes*) können Multiprozessoren ausnutzen.
- ☐ Threads (*Light-weight Processes*) teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- ☐ Zu jedem Thread (*Light-weight Process*) gehört ein eigener isolierter Adressraum.
- ☐ Der Synchronisationsbedarf im Anwendungsprogramm kann von der Ablaufplanung der Kernfäden abhängen.
- ☐ Die Umschaltung von Threads (*Light-weight Processes*) muss im Systemkern erfolgen.
- ☐ Zur Umschaltung von User-level Threads (*Feather-weight Processes*) ist ein Adressraumwechsel erforderlich.

Aufgabe 2: hupsi - Highly Unreliable Parallel Software Igniter (59 Punkte)

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

Schreiben Sie ein POSIX-1.2008 und C11-konformes Programm *hupsi*, das Befehle zeilenweise (ein Befehl pro Zeile, Zeilen mit `'\n'` getrennt) von der Standardeingabe einliest und parallel ausführt. Die maximale Anzahl n an parallel ausgeführten Befehlen soll dabei als Befehlszeilenargument übergeben werden.

Beispielhafter Aufruf von *hupsi* (4 Befehle, davon maximal 2 parallel):

```
chris@legio:~$ ./hupsi 2 < befehle.txt
```

Beispielinhalt befehle.txt:

```
gcc hupsi.c -o hupsi
sleep 5
find / -name bs-klausur.tex
wsort klausurnoten.txt
```

Das Programm soll folgendermaßen strukturiert sein:

- Funktion `main()`: Prüft zunächst die Befehlszeilenargumente und initialisiert ggf. benötigte Datenstrukturen. Nutzen Sie zum Umwandeln der Obergrenze n die Funktion `strtol`, um eine saubere Fehlerprüfung zu ermöglichen.
  Im Anschluss daran werden die auf der Standardeingabe übergebenen Befehle durch Aufruf der nachfolgend beschriebenen `run`-Funktion parallel ausgeführt. Dabei sollen, solange noch nicht ausgeführte Befehle vorhanden sind, stets genau n Befehle parallel laufen. Falls die maximale Zahl an parallel laufenden Prozessen erreicht ist, soll mittels der unten beschriebenen Funktion `waitProcess()` auf das Terminieren eines zuvor gestarteten Prozesses gewartet werden. Sie dürfen (ohne weitere Prüfung) davon ausgehen, dass die übergebenen Befehle eine maximale Länge von 100 (`CMD_MAX`) Zeichen nicht überschreiten. Zur Verwaltung der gestarteten Prozesse soll eine einfach verkettete Liste aus **`struct`** `process`-Elementen verwendet werden; das Hinzufügen von Elementen geschieht ebenfalls in der Funktion `main()`. Nach dem Start aller Befehle wird abschließend auf die noch laufenden Prozesses gewartet.
- Funktion `pid_t run(`**`char`** `*cmdline)`:
  Führt die übergebene Befehlszeile aus und gibt die Prozess-ID des erzeugten Kindes zurück. Dazu wird ein neuer Prozess erzeugt, das auszuführende Programm und die Parameter aus der Befehlszeile extrahiert und die Ausführung mittels einer Funktion der `exec()`-Familie gestartet. Tritt bei der Ausführung mittels `exec()` ein Fehler auf, wird eine aussagekräftige Fehlermeldung ausgegeben und der Kindprozess beendet. Zur Vereinfachung dürfen Sie annehmen, dass die zu extrahierenden Parameter durch Leerzeichen voneinander getrennt sind und sich innerhalb der einzelnen Parameter keine weiteren Leerzeichen befinden.
- Funktion **`void`** `waitProcess(`**`void`**`)`:
  Wartet passiv auf einen beliebigen der zuvor gestarteten Befehle. Nach Terminierung des Prozesses gibt `waitProcess` die PID, die Befehlszeile, den Exitcode (falls zutreffend) und die Ausführdauer aus. Nutzen Sie hierfür die Funktion `time()` (siehe Manpage).

Hinweise:

- Neue Prozesse sollen am Anfang der Prozessliste eingehängt werden.
- Die Definition der **`struct`** `process` darf um zusätzliche Einträge erweitert werden.
- Das `fflush()` am Ende des Programms ist in dieser Aufgabe nicht notwendig.
- Achten Sie auf **korrekte und vollständige Fehlerbehandlung**.

Auf den folgenden Seiten finden Sie ein grobes Gerüst für das beschriebene Programm. Es ist überall großzügig Platz gelassen, damit Sie nachträgliche Anpassungen durchführen können. Einige wichtige Manual-Seiten liegen bei – es kann aber durchaus sein, dass Sie nicht alle Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.


```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

#define CMD_MAX 100

static void die(const char *msg) {
  perror(msg);
  exit(EXIT_FAILURE);
}

static void usage(void) {
  fprintf(stderr, "Usage: hupsi <n>\n");
  exit(EXIT_FAILURE);
}

struct process {
  pid_t  pid; // PID des Prozesses
  // Eigene Mitglieder
};

// Makros, Funktionsdeklarationen, globale Variablen
```

G:

```
// Funktion main

  // Befehlszeilenargument(e) prüfen

  // Befehlszeilenarg. <n> mit strtol parsen

  // Befehle von stdin einlesen und verarbeiten
```



```
// Ende Funktion main
```

M:

```
// Funktion run
```

```
// Ende Funktion run
```

R:


```
// Funktion waitProcess
  // Auf beliebigen Prozess warten
  // struct process zu PID aus Prozessliste ermitteln
  // (Vorbereiten der) Ausgabe
```

```
  // Prozess aus Liste entfernen & Speicher freigeben
```

```
// Ende Funktion waitProcess
```

W:

2) Makefile (8 Punkte)

Schreiben Sie ein Makefile, welches die Targets `all` und `clean` unterstützt. Ebenfalls soll ein Target `hupsi` unterstützt werden, welches das Programm `hupsi` baut. Greifen Sie dabei stets auf Zwischenprodukte (z.B. hupsi.o) zurück.

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

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

Mk:

Aufgabe 3: Synchronisation (14 Punkte)

1) Was versteht man unter einer Verklemmung und was sind die (hinreichenden und notwendigen) Bedingungen, damit eine Verklemmung auftreten kann? (6 Punkte)

2) Erläutern Sie das Konzept Semaphor. Welche Operationen sind auf Semaphoren definiert und was tun diese Operationen? (5 Punkte)

3) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe eines zählenden Semaphors das folgende Szenario korrekt synchronisiert werden kann: Zu jedem Zeitpunkt müssen so viele Threads wie möglich, maximal jedoch 4, die Funktion `doWork()` ausführen. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung: (3 Punkte)

- `SEM * semCreate(int);`
- `void P(SEM *);`
- `void V(SEM *);`

Beachten Sie, dass nicht unbedingt alle freien Zeilen für eine korrekte Lösung nötig sind. Kennzeichnen Sie durch `/`, wenn Ihre Lösung in einer freie Zeile keine Operation benötigt.
Fehlerbehandlung ist in dieser Aufgabe nicht notwendig.

Hauptthread:

```
static SEM *s;
int main(void){

-------------------------------------
  while(1) {

-------------------------------------
  startWorkerThread(threadFunc);

-------------------------------------
  }

-------------------------------------
}
```

Arbeiterthread:

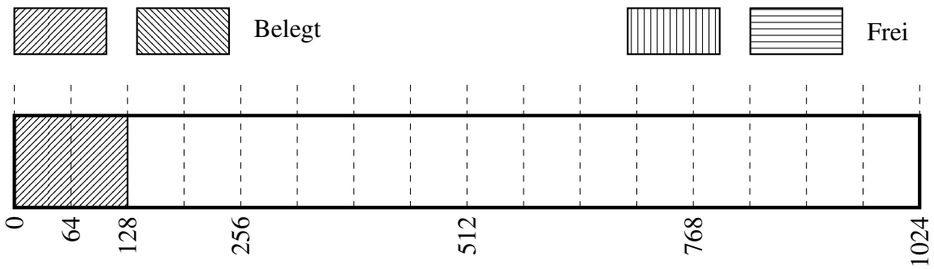
```
void threadFunc(void) {

-------------------------------------
  doWork();

-------------------------------------
}
```


Aufgabe 4: Adressräume & Freispeicherverwaltung (17 Punkte)

1) 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 128 Bytes vergeben worden. Ein Programm führt nacheinander die im folgenden Bild angegebenen Anweisungen aus. (11 Punkte)

```
⓪  p0 = malloc(100);   // 0 (initial vergebener Block)
①  p1 = malloc(50);
②  p2 = malloc(110);
③  p3 = malloc(32);
④  free(p1);
⑤  p4 = malloc(386);
⑥  free(p3);
⑦  free(p4);
```

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, und tragen Sie in der Tabelle den aktuellen Zustand der Bucketliste (*left-over holes*) nach **jedem** Schritt ein. Für Löcher gleicher Größe schreiben Sie die Adressen einfach nebeneinander in die Tabellenzeile (es ist nicht notwendig, verkettete Buddys wie in der Vorlesung beschrieben einzutragen).



Größe der Buckets	initial ⓪	①	②	③	④	⑤	⑥	⑦
2^5								
2^6								
2^7	128							
2^8	256							
2^9	512							
2^{10}								

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

2) Man unterscheidet bei Adressraumkonzepten und bei Zuteilungsverfahren zwischen externer und interner Fragmentierung. Erläutern Sie den Unterschied. Was kann man jeweils gegen die beiden Arten der Fragmentierung tun? (4 Punkte)

3) Im Hinblick auf Adressraumkonzepte gibt es bei interner Fragmentierung einen Nebeneffekt in Bezug auf Programmfehler (vor allem im Zusammenhang mit Zeigern). Beschreiben Sie diesen Effekt. (2 Punkte)

Ersatzgrafik für Teilaufgabe 4.1.

Sie dürfen diese Grafik nutzen, falls Sie sich beim Ausfüllen der Grafik in 4.1. verzeichnet haben. Markieren Sie eindeutig, welche der Grafiken zur Bewertung herangezogen werden soll!