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) 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

b) Man unterscheidet die Begriffe Programm und Prozess. Welche der folgenden Aussagen zu diesem Themengebiet ist richtig? 2 Punkte

- ○ Der Binder erzeugt aus mehreren Programmteilen (Module) einen Prozess.
- ○ Ein Prozess ist ein Programm in Ausführung - ein Prozess kann während seiner Lebenszeit aber auch mehrere verschiedene Programme ausführen.
- ○ Das Programm ist der statische Teil (Rechte, Speicher, etc.), der Prozess der aktive Teil (Programmzähler, Register, Stack).
- ○ Der UNIX-Systemaufruf `fork(2)` lädt eine Programmdatei in einen neu erzeugten Prozess.

c) Was passiert, wenn Sie in einem C-Programm über einen ungültigen Zeiger versuchen 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“.
- ○ Die MMU erkennt die ungültige Adresse bei der Adressumsetzung und löst einen Trap aus.
- ○ Das Betriebssystem erkennt die ungültige Adresse bei der Weitergabe des Befehls an die CPU (partielle Interpretation) und leitet eine Ausnahmebehandlung ein.
- ○ Der Compiler erkennt die problematische Code-Stelle und generiert Code, der zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.

d) Was versteht man unter Virtuellem Speicher? 2 Punkte

- ○ Virtueller Speicher kann größer sein als der physikalisch vorhandene Arbeitsspeicher. Gerade nicht benötigte Speicherbereiche können auf Hintergrundspeicher ausgelagert werden.
- ○ Virtueller Speicher ist in unbegrenzter Menge vorhanden.
- ○ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.
- ○ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.

e) Welche Problematik wird auf das Philosophenproblem abgebildet? 2 Punkte

- ○ Ein Erzeuger und ein Verbraucher greifen gleichzeitig auf gemeinsame Datenstrukturen zu.
- ○ Exklusive Bearbeitung durch mehrere Bearbeitungsstationen.
- ○ Mehrere Prozesse greifen lesend und schreibend auf gemeinsame Datenstrukturen zu.
- ○ Gleichzeitiges Belegen mehrerer Betriebsmittel.

f) Welche der folgenden Aussagen zum Thema Threads sind richtig? 2 Punkte

- ○ Bei federgewichtigen Prozessen ist die Schedulingstrategie durch das Betriebssystem vorgegeben.
- ○ Bei Kern-Threads ist die Schedulingstrategie meist durch das Betriebssystem vorgegeben.
- ○ Die Umschaltung von Threads muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl).
- ○ Kern-Threads blockieren sich bei blockierenden Systemaufrufen gegenseitig.

g) Wodurch kann es in einem System zu Nebenläufigkeit kommen? 2 Punkte

- ○ Durch Multithreading auf einem Monoprozessorsystem.
- ○ Durch Seitenflattern.
- ○ Durch langfristiges Scheduling.
- ○ Durch Traps.

TODO

h) Bei einer prioritätengesteuerten Prozess-Auswahlstrategie (Schedulingstrategie) kann es zu Problemen kommen. Welches der folgenden Probleme kann auftreten?

2 Punkte

- ○ Eine prioritätenbasierte Auswahlstrategie arbeitet sehr ineffizient, wenn viele Prozesse im Zustand bereit sind.
- ○ Prioritätenbasierte Auswahlstrategien führen zwangsläufig zur Aushungerung von Prozessen, wenn mindestens zwei verschiedene Prioritäten vergeben werden.
- ○ Ein hochpriorer Prozesse muss eventuell auf ein Betriebsmittel warten, das von einem niedrigpriorien Prozess exklusiv benutzt wird. Der niedrigpriore Prozess kann das Betriebsmittel jedoch wegen eines mittelhochprioren Prozesses nicht freigeben (Prioritätenumkehr).
- ○ Das Phänomen der Prioritätsumkehr hungert niedrig-priore Prozesse aus.

i) Ein Programm will die drei Zeichenketten

2 Punkte

```
char a[] = "dire";
char b[] = "cto";
char c[] = "ry";
```

mit der Funktion `sprintf(3)` wie folgt in einen Puffer buffer speichern:

```
sprintf(buffer, "%s/%s/%s", a, b, c);
```

Mit welcher Länge (in Bytes) muss der Puffer buffer mindestens angelegt werden, damit kein Überlauf entstehen kann?

- ○ 12
- ○ 10
- ○ 11
- ○ 13

j) Ein Prozess wird in den Zustand bereit überführt. Welche Aussage passt zu diesem Vorgang?

2 Punkte

- ○ Der Prozess wartet auf eine Tastatureingabe.
- ○ Ein anderer Prozess blockiert sich an einem Semaphor.
- ○ Der Prozess hat einen Seitenfehler für eine Seite, die aber noch im Hauptspeicher vorhanden ist.
- ○ Der Prozess hat auf Daten von der Festplatte gewartet und die Daten stehen nun zur Verfügung.

k) Gegeben sei folgendes Szenario: zwei Fäden werden auf einem Monoprozessorsystem mit der Strategie „First Come First Served“ verwaltet. In jedem Faden wird die Anweisung „i++;“ auf die gemeinsame, globale Variable i ausgeführt. Welche der folgenden Aussagen ist richtig:

2 Punkte

- ○ In einem Monoprozessorsystem ohne Verdrängung ist keinerlei Synchronisation erforderlich.
- ○ Während der Inkrementoperation müssen Interrupts vorübergehend unterbunden werden.
- ○ Die Inkrementoperation muss mit einer CAS-Anweisung nicht-blockierend synchronisiert werden.
- ○ Die Operation i++ ist auf einem Monoprozessorsystem immer atomar.

文本

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) Gegeben sei folgendes Programm:

4 Punkte

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define PI 3.1415
5
6  extern int x;
7
8  int main(int argc, char *argv[]) {
9      static int a;
10     int b = PI;
11
12     x = a + b;
13
14     printf("%f\n", b);
15
16     return EXIT_SUCCESS;
17 }
```

Welche der folgenden Aussagen bzgl. dieses Programms sind korrekt?

- ☐ Die Variable `a` ist uninitialisiert und enthält daher einen zufälligen Wert.
- ☐ `argv` ist ein Array aus Zeigern, die jeweils auf ein Array aus `char`s zeigen.
- ☐ Beim Binden des Programms kann ein Fehler auftreten.
- ☐ Beim Überschreiben der Variable `x` in Zeile 12 tritt ein Fehler auf, weil externe Variablen nicht überschrieben werden dürfen.
- ☐ Die globale Variable `PI` enthält den Wert `3.1415`.
- ☐ Der Inhalt der Datei *stdlib.h* wird vor dem Übersetzen an die Stelle des includes einkopiert.
- ☐ An Index `0` des `argv`-Arrays liegt ein Zeiger auf den Programmnamen oder -pfad.
- ☐ Der Aufruf von `printf` in Zeile 14 gibt den Wert `3.1415` auf `stdout` aus.

Ein Befehlscache (Instruction Cache) ist ein spezieller Cache, der die zuletzt ausgeführten Maschinenbefehle zwischenspeichert, um den Zugriff auf den Befehlsspeicher zu beschleunigen.

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

4 Punkte

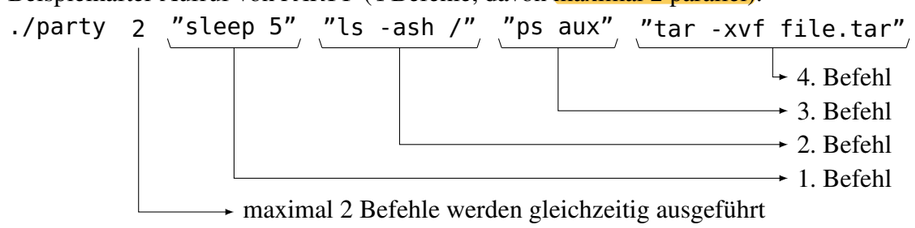
- ☐ Einen speziellen Cache der CPU, der die zuletzt ausgeführten Maschinenbefehle zwischenspeichert (beschleunigt vor allem den Ablauf von Schleifen).
- ☐ Wird eine Speicherabbildung im TLB nicht gefunden, wird die Abbildung in den Seitentabellen nachgeschlagen und im TLB eingetragen.
- ☐ Verändert sich die Speicherabbildung von logischen auf physikalische Adressen aufgrund einer Adressraumumschaltung, so werden auch die Daten im TLB ungültig.
- ☐ Der TLB beinhaltet einen voll-assoziativen Cache, der zur Adressumsetzung genutzt wird.
- ☐ Der TLB verkürzt die Zugriffszeit auf den physikalischen Speicher, da ein Teil des möglichen 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.
- ☐ Der TLB puffert die Ergebnisse der Abbildung von physikalische auf logische Adressen, sodass eine erneute Anfrage sofort beantwortet werden kann.

Aufgabe 2: PARTY - PARallel Task Player (45 Punkte)

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

Schreiben Sie ein Programm *PARTY*, das per Befehlszeile übergebene Befehle parallel ausführt. Dabei soll darauf geachtet werden, dass maximal n Befehle gleichzeitig laufen. Die Obergrenze n soll dabei ebenfalls per Befehlszeile übergebenen werden.

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



Das Programm soll folgendermaßen strukturiert sein:

– Funktion `main()`: Initialisiert zunächst alle benötigten Datenstrukturen und prüft die Befehlszeilenargumente. Nutzen Sie zum Umwandeln der Obergrenze n die Funktion `strtol`, um eine Fehlerprüfung zu ermöglichen.

  Im Anschluss daran werden die als Befehlszeilenargumente übergebenen Befehle parallel ausgeführt. Dabei sollen, solange noch nicht ausgeführte Befehle vorhanden sind, stets genau n Befehle parallel laufen. Zur Verwaltung der gestarteten Prozesse soll ein **`struct`** process-Array verwendet werden. Nachdem alle Befehle gestartet wurden, soll auf die noch laufenden Prozesses gewartet werden. *Hinweis:* Es steht Ihnen frei, die Definition der Struktur um zusätzliche Einträge zu erweitern.

– Funktion `pid_t run(char *cmdline)`: Führt die übergebene Befehlszeile aus. Dazu werden das auszuführende Programm und die Parameter aus der Befehlszeile extrahiert und mithilfe einer Funktion der `exec()`-Familie ausgeführt. Tritt im Kindprozess ein Fehler auf, dann wird eine aussagekräftige Fehlermeldung ausgegeben und der Kindprozess beendet. Das Hauptprogramm selbst läuft im Falle von Fehlern im Kindprozess weiter. 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(`**`struct`** `process *processes, size_t size)`: Wartet passiv auf einen beliebigen der zuvor gestarteten Befehle. Nach Terminierung des Prozesses gibt `waitProcess` die PID, die Befehlszeile und (falls zutreffend) den Exitcode aus.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen zu ergänzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist überall sehr großzügig Platz gelassen, damit Sie auch weitere notwendige Programmanweisungen entsprechend Ihrer Programmierung einfügen können.

Einige wichtige Manual-Seiten liegen bei – es kann aber durchaus sein, dass Sie bei Ihrer Lösung nicht alle diese Funktionen oder gegebenenfalls auch weitere Funktionen benötigen.

Hinweis: Diese Übungsaufgabe ist etwas kürzer (45 Minuten) als die Programmieraufgabe in der "richtigen" Klausur (60 Minuten).


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

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

static void usage(void) {
  fprintf(stderr, "Usage: party <#parallel processes> <commandlines>\n");
  exit(EXIT_FAILURE);
}

struct process {
  pid_t  pid; // PID des Prozesses

};

// Makros, Funktionsdeklarationen, globale Variablen

// Funktion main

  // Befehlszeilenargument(e) prüfen
```

```
  // Befehlszeilenarg. <n> mit strtol parsen
```

```
// Ende Funktion main
```

M:


```
// Funktion run
```

```
    // Befehlszeile parsen
```

```
// Ende Funktion run
```

R:

```
// Funktion waitProcess
```

```
  // Auf beliebigen Prozess warten
```

```
  // Befehlszeile raussuchen
```

```
  // Terminierungsgrund ausgeben
```

W:

2) Makefile (8 Punkte)

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

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

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

```
# Makefile
```

Mk:

Aufgabe 3: Synchronisation (16 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 Mutex. Welche Operationen sind auf Mutexen definiert und was tun diese Operationen? (5 Punkte)

3) Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe eines Mutexes das folgende Szenario korrekt synchronisiert werden kann: Vier Threads führen parallele Berechnungen durch und addieren die berechneten auf den Wert einer globalen Variable auf. Zu jedem Zeitpunkt müssen so viele Threads wie möglich die Funktion `calcValue` ausführen. Ihnen stehen dabei folgende Mutex-Funktionen zur Verfügung: (5 Punkte)

- `MUT * mutCreate();` // nicht gesperrt nach Erzeugung
- `void lock(MUT *);`
- `void unlock(MUT *);`

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.

Hauptthread:

```
static int accu;
static MUT *m;
int main(void){

-------------------------------------
  for(int i = 0; i < 3; ++i) {
    startWorkerThread(threadFunc);
  }
-------------------------------------
  while(1) {
-------------------------------------
    int x = calcValue();
-------------------------------------
    accu += x;
-------------------------------------
  }
}
```

Arbeiterthread:

```
void threadFunc(void) {

-------------------------------------
  while(1) {
-------------------------------------
    int x = calcValue();
-------------------------------------
    accu += x;
-------------------------------------
  }
-------------------------------------
}
```

Aufgabe 4: Freispeicherverwaltung (14 Punkte)

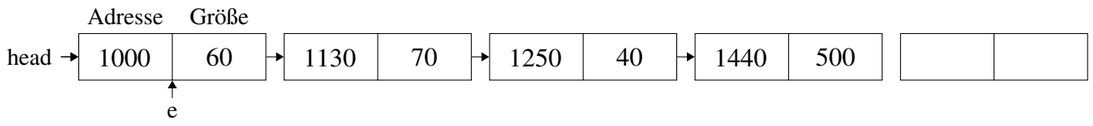
Zur Verwaltung von freiem Speicher (z.B. zur feingranularen Verwaltung in Funktionen wie `malloc(3)` und `free(3)`) gibt es verschiedene Strategien zur Herausgabe des Speichers.

1) Vervollständigen Sie den Zustand der Verwaltungsdatenstrukturen für die untenstehende Folge von `malloc(3)`- und `free(3)`-Aufrufen. Im Rahmen dieser Aufgabe soll dafür das **next-fit**-Verfahren mit Verschmelzung von freien Blöcken angewandt werden. Zeichnen Sie pro Schritt den Einsprungpunkt **e** und den Zustand der Freispeicherliste in die untenstehende Abbildung ein.

Der Einsprungpunkt **e** wird zur Suche des nächsten Blocks genutzt und zeigt auf den zuletzt geteilten Block. Wird ein Block vollständig entnommen, so wird **e** auf dessen Nachfolger gesetzt.

Vermerken Sie zudem, welche Adresse der jeweilige `malloc(3)`-Aufruf zurückliefert. In dem Fall, dass ein Speicherblock aufgeteilt wird, soll der hintere Teil (entspricht dem Speicherbereich mit der höheren Adresse) an den Aufruf Aufrufer zurückgegeben werden.
⓪ kennzeichnet den initialen Zustand der Speicherverwaltung nach der Allokation p_0. (9 Punkte)

⓪ p_0 `= malloc(50); // = 1060` ← Rückgabewert des Aufrufs von malloc



① p_1 `= malloc(70);`

② p_2 `= malloc(20);`

③ p_3 `= malloc(500);`

④ `free(`p_1`);`

⑤ `free(`p_0`);`

2) Nennen Sie je einen Vorteil und einen Nachteil von **next-fit** gegenüber **best-fit**. (1 Punkt)

3) Erläutern Sie den Unterschied zwischen interner und externer Fragmentierung. (2 Punkte)

4) 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)