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) Was versteht man unter einer Unterbrechung (*Interrupt*) bei der Ausführung von Instruktionen durch einen Prozessor? 2 Punkte

- ☐ 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.
- ☐ Der Prozessor wird veranlasst eine Unterbrechungsbehandlung durchzuführen. Der gerade laufende Prozess kann die Unterbrechungsbehandlung ignorieren.
- ☐ Eine Signalleitung teilt dem Prozessor mit, dass er den aktuellen Prozess anhalten und auf das Ende der Unterbrechung warten soll.
- ☐ Durch eine Signalleitung wird der Prozessor veranlasst, die gerade bearbeitete Maschineninstruktion zu unterbrechen und in den Benutzermodus umzuschalten.

b) Für welchen Zweck wird der Systemaufruf listen() benutzt? 2 Punkte

- ☐ Der Aufruf von listen() wartet solange an einem Socket, bis eine einkommende Verbindungsanfrage vorliegt.
- ☐ Damit das Betriebssystem überhaupt Systemaufrufe annimmt, muss es erst mit listen() in einen Modus des „Zuhörens“ gebracht werden.
- ☐ Der Aufruf von listen() erzeugt eine leere verkettete Liste, die zum Speichern von Daten verwendet werden kann.
- ☐ Mit listen() wird ein Socket für die Verbindungsannahme vorbereitet. Ein Parameter gibt an, wieviele Verbindungsanfragen vor deren Annahme gepuffert werden können.

c) Für lokale Variablen, Aufrufparameter, etc. einer Funktion wird bei vielen Prozessoren ein Stack-Frame angelegt. Welche Aussage ist richtig? 2 Punkte

- ☐ Bei rekursiven Funktionsaufrufen kann der Speicher des Stack-Frames in jedem Fall wiederverwendet werden, weil die gleiche Funktion aufgerufen wird.
- ☐ Wenn in einem UNIX-Prozess mehrere Threads parallel laufen, benötigt jeder von ihnen einen eigenen Stack.
- ☐ Ein Pufferüberlauf eines lokalen Arrays wird immer zu einem Segmentation Fault führen und kann somit keine sicherheitskritischen Auswirkungen haben.
- ☐ Der Compiler legt zur Übersetzungszeit fest, an welcher Position im Stack-Frame der main-Funktion die globalen Variablen angelegt werden.

d) Wozu benötigt man Bedingungsvariablen (condition variables)? 2 Punkte

- ☐ Zur Erkennung von Verklemmungen.
- ☐ Bei if-Abfragen in C-Programmen.
- ☐ Zum aktiven Warten in kritischen Abschnitten.
- ☐ Um in einem kritischen Abschnitt auf ein Ereignis zu warten und den kritischen Abschnitt während der Wartezeit freizugeben.

e) Welche Antwort trifft für die Eigenschaften eines UNIX/Linux-Dateideskriptors zu? 2 Punkte

- ☐ Ein Dateideskriptor ist eine Verwaltungsstruktur, die auf der Festplatte gespeichert ist und Informationen über Größe, Zugriffsrechte, Änderungsdatum usw. einer Datei enthält.
- ☐ Ein Dateideskriptor ist ein Zeiger auf Betriebssystem-interne Strukturen, der von den Systemaufrufen ausgewertet wird, um auf Dateien zuzugreifen.
- ☐ Beim Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Filedeskriptor zum Zugriff zurück.
- ☐ Ein Dateideskriptor ist eine prozesslokale Ganzzahl, die der Prozess zum Zugriff auf eine geöffnete Datei benutzen kann.

f) Wodurch kann es zu Seitenflattern kommen? 2 Punkte

- ☐ Durch Programme, die eine Defragmentierung auf der Platte durchführen.
- ☐ Wenn die Zahl der residenten Seiten die Größe des physikalischen Speichers überschreitet.
- ☐ Wenn bei einer verdrängenden Scheduling-Strategie die Zahl der von den Prozessen aktiv genutzten Seiten die Zahl der verfügbaren Seitenrahmen übersteigt.
- ☐ Wenn bei der Ersetzungsstrategie Second Chance (SC) bei einem Zugriff auf eine Seite das Referenzbit gesetzt und die Seite danach längere Zeit nicht angesprochen wird, so dass das Bit wieder gelöscht wird. Wenn sich diese Situation mehrfach wiederholt, so dass das Bit ständig den Wert ändert, spricht man von Seitenflattern.

g) Gegeben sei folgendes Szenario: zwei Fäden werden auf einem Monoprozessorsystem mit der Strategie „First Come First Served“ ohne Verdrängung verwaltet. In jedem Faden wird die Anweisung `i++;` auf die gemeinsame, globale Variable `i` ausgeführt. Welche der folgenden Aussagen ist richtig? 2 Punkte

- ☐ 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.
- ☐ In einem Monoprozessorsystem ohne Verdrängung ist keinerlei Synchronisation erforderlich.

h) Was versteht man unter Virtuellem Speicher?

2 Punkte

- ☐ Speicher, der einem Prozess durch Ein- und Auslagern von Speicherbereichen durch das Betriebssystem und die Hardware vorgespiegelt wird.
- ☐ Adressierbarer Speicher in dem sich keine Daten speichern lassen, weil er physikalisch nicht vorhanden ist.
- ☐ Einen logischen Adressraum.
- ☐ Speicher, der nur im Betriebssystem sichtbar ist, jedoch nicht für einen Anwendungsprozess.

i) Wodurch kann es in einem System zu Nebenläufigkeit kommen?

2 Punkte

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

j) Was versteht man unter einem Translation Lookaside Buffer (TLB)?

2 Punkte

- ☐ Einen speziellen Cache der CPU, der die zuletzt ausgeführten Maschinenbefehle zwischenspeichert (beschleunigt vor allem den Ablauf von Schleifen).
- ☐ Einen speziellen Cache der MMU, der Informationen aus den zuletzt genutzten Seitendeskriptoren vorhält.
- ☐ Der TLB ist eine schnelle Umsetzeinheit der MMU, die physikalische in logische Adressen umsetzt.
- ☐ Einen speziellen Cache der MMU, der den Inhalt der zuletzt angesprochenen Speicherzellen vorhält.

k) Welche der folgenden Aussagen über Einplanungsverfahren ist richtig?

2 Punkte

- ☐ Beim Einsatz präemptiver Einplanungsverfahren kann laufenden Prozessen die CPU nicht entzogen werden.
- ☐ Bei kooperativer Einplanung kann es zur Monopolisierung der CPU kommen.
- ☐ Asymmetrische Einplanungsverfahren können ausschließlich auf asymmetrischen Multiprozessor-Systemen zum Einsatz kommen.
- ☐ Probabilistische Einplanungsverfahren müssen die exakten CPU-Stoßlängen aller im System vorhandenen Prozesse kennen.

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) Welche der folgenden Aussagen zum Thema Adressraumkonzepte sind richtig?

4 Punkte

- ○ Ein Adressraumwechsel ist eine privilegierte Operation und kann daher nur durch das Betriebssystem vorgenommen werden.
- ○ Mit Hilfe des Systemaufrufes `malloc(3)` kann ein Programm zusätzliche Speicherblöcke von sehr feinkörniger Struktur vom Betriebssystem anfordern.
- ○ Mit `malloc(3)` angeforderter Speicher, welcher vor Programmende nicht freigegeben wurde, kann vom Betriebssystem nicht mehr an andere Prozesse herausgegeben werden und ist damit bis zum Neustart des Systems verloren.
- ○ Da das Laufzeitsystem auf die Betriebssystemschnittstelle zur Speicherverwaltung zurückgreift, ist die Granularität der von `malloc(3)` zurückgegebenen Speicherblöcke vom Betriebssystem vorgegeben.
- ○ Ein Speicherbereich, der mit Hilfe der Funktion `free(3)` freigegeben wurde, verbleibt im logischen Adressraum des zugehörigen Prozesses.
- ○ Bei einer seitenorientierten Adressraumorganisation muss die Größe jeder Seite in der Seitentabelle gespeichert werden.
- ○ Ein Seitenfehler wird abhängig von der Ursache nach dem Fortsetzungs- oder dem Beendigungsmodell behandelt.
- ○ Zur Implementierung von logischen Adressräumen ist spezielle Hardwareunterstützung nötig.

b) Welche der folgenden Aussagen zu UNIX-Dateisystemen sind richtig?

4 Punkte

- ◯ Auf das Wurzelverzeichnis (root directory, "/") darf immer nur genau ein Hardlink verweisen.
- ◯ Auf ein Verzeichnis darf immer nur genau ein Hardlink verweisen.
- ◯ Wird der letzte Hardlink auf eine Datei entfernt, so wird auch die Datei selbst gelöscht.
- ◯ Ein symbolischer Link kann auf eine Datei innerhalb eines anderen Dateisystems verweisen.
- ◯ In einem hierarchisch organisierten Dateisystem dürfen gleiche Dateinamen in unterschiedlichen Verzeichnissen enthalten sein.
- ◯ Ein Inode in einem UNIX-Dateisystem enthält den Dateinamen und weitere Metadaten zu einer Datei.
- ◯ Ein Hardlink kann auf eine Datei innerhalb eines anderen Dateisystems verweisen.
- ◯ Es ist möglich, einen symbolischen Link mit Verweis auf eine nicht-existierende Datei anzulegen.

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

Aufgabe 2: reci (60 Punkte)

Schreiben Sie einen **Re**liable **CI**-Runner (reci) der in einem Verzeichnis abgelegte Aufträge parallel ausführt und jeweils einen Server über das Ergebnis per TCP informiert.

Als einziges Argument erhält das Programm ein Verzeichnis, in dem beliebige Dateien abgelegt sind. Aufträge sind Dateien mit einer gültigen Jobbeschreibung im Format "`CPULOAD,HOST,BINARY\n`". reci liest die Dateien und startet für jeden Auftrag jeweils einen Prozess mit dem Programm `BINARY` sowie einen Begleiter- und einen Timeout-Thread. Das Starten soll erst erfolgen, wenn `CPULOAD` Kerne verfügbar sind. Der Begleiter-Thread wartet auf das Terminieren des gestarteten Prozesses und informiert `HOST` im Anschluss über die Abarbeitung. Der Timeout-Thread sendet nach 10 Sekunden `SIGKILL` an den gestarteten Prozess.

Die von allen Jobs zusammen genutzte Anzahl an CPU-Kernen soll nie größer sein, als die verfügbare Anzahl an Kernen (`NR_CPUS`). Fehler durch ungültige Verzeichnis- oder Auftragsinhalte sollen immer nur zu einer Warnung auf `stderr` und dem Abbruch des jeweiligen Jobs führen. Nur fatale Fehler, die auf allgemeine Ressourcenknappheit hinweisen, sollen zum Abbruch des gesamten Programms mit einem Fehlercode führen.

Implementieren Sie folgende Funktionen:

`int main(int argc, char *argv[])` Iteriert mit `scandir()` über die Dateien, die im als Argument übergebenen Verzeichnis liegen und ruft für jede sichtbare Datei (Name beginnt nicht mit einem Punkt) `startJob()` auf. Anschließend wartet die Funktion auf das Terminieren aller Begleiter- und Timeout-Threads und schreibt (mit Fehlerbehandlung) die Dateinamen der erfolgreichen Jobs auf `stdout` (d.h. die Jobs bei denen der jobspezifische Server erfolgreich informiert wurde).

`int filter(const struct dirent *e)` Callback-Funktion zum Filtern der Dateien.

`int startJob(job_t *j, FILE *f)` Liest aus `f` die Jobbeschreibung ein (max. `LMAX` Nutzzeichen und zusätzliches `\n`, durch Kommas getrennte Tokens) und ergänzt `j` um die entsprechenden Werte. `CPULOAD` wird mittels der vorgegebenen Funktion `parsePositiveInt()` in eine Zahl umgewandelt die kleiner als die verfügbare Anzahl an Kernen sein muss. Startet das spezifizierte `BINARY` ohne weitere Argumente sowie den Begleiter- und Timeout-Thread. Wird ein Fehler beim Einlesen des Jobs bemerkt wird dieser mit einer Warnung auf `stderr` abgebrochen und alle mit dem Job assoziierten Ressourcen werden freigegeben (Rückgabewert `-1`). Geben Sie keine offenen Dateien bis auf `stdin`, `stdout` und `stderr` an den Kindprozess weiter (auch keine Sockets aus `tcpNotify()`).

`void *attendant(void *voidJob)` Der Begleiter-Thread wartet auf das Terminieren des Jobprozesses und gibt die von diesem verwendete Zahl an CPU-Kernen wieder frei. Hat der Prozess sich mit dem Exitstatus `0` regulär beendet, wird der jobspezifische Server mittels `tcpNotify()` darüber informiert. Gibt bei jobspezifischen Fehlern `NULL`, und bei Erfolg `voidJob` zurück.

`void *timeout(void *voidJob)` Wartet mittels `sleep()` 10 Sekunden um dann dem Jobprozess das Signal `SIGKILL` zu senden. An den Thread gerichtete Signale dürfen kein frühzeitiges senden von `SIGKILL` hervorrufen können. Gehen Sie davon aus, dass PIDs vom System nie wiederverwendet werden.

`void P(SEM *sem, unsigned n)` und `void V(SEM *sem, unsigned n)` Implementieren Sie eine Variante des allgemeinen Semaphors (mit passivem Warten) der das atomare Erhöhen/Verringern um einen beliebigen positiven Wert `n` erlaubt (d.h., nicht nur um 1).

`int tcpNotify(char *host, char *name)` Sucht die IP-Adresse von `host` und baut eine TCP-Verbindung zum Server auf (Port `1604`). Informiert dann den Server mit der vorgegebenen Funktion `writeJobNotification()` über das erfolgreiche Abarbeiten des Jobs. Gibt im Fehlerfall eine Warnung aus und `-1` zurück, im Erfolgsfall `0`.

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

```c
#include <dirent.h>
#include <errno.h>
#include <fnmatch.h>
#include <limits.h>
#include <netdb.h>
#include <netinet/in.h>
#include <pthread.h>
#include <pwd.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <unistd.h>

#define LMAX 256
#define NR_CPUS 8

typedef struct SEM {
  unsigned value;
  pthread_mutex_t mutex;
  pthread_cond_t cond;
} SEM;

typedef struct job {
  char *name;
  bool user_error;
  int load;
  char host[LMAX + 2];
  pid_t pid;
  pthread_t attendant, timeout;
} job_t;

static void die(char *m) { perror(m); exit(EXIT_FAILURE); }
// Gibt einen positiven int zurück, oder -1 im Fehlerfall:
static int parsePositiveInt(char *str) { /* ... */ }
// Informiert einen Server durch schreiben auf ‘socket‘ über den Abschluss des
// Jobs ‘name‘. Schreibt in Fehlerfall eine Warnung auf ‘stderr‘ und gibt -1
// zurück, sonst 0. Schließt ‘socket‘ in jedem Fall:
static int writeJobNotification(int socket, char *name) { /* ... */ }

// Allokiert SEM und initialisiert alle Felder. Gibt im Fehlerfall
// NULL zurück und setzt errno:
static SEM *semCreate(unsigned initVal) { /* ... */ }
// Gibt alle Ressourcen von SEM frei. Kann nicht Fehlschlagen:
static void semDestroy(SEM *sem) { /* ... */ }
```

```
static SEM *load_sem;

int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(stderr, "USAGE: %s DIRECTORY\n", argv[0]);
    return EXIT_FAILURE;
  }

  struct sigaction action = { .sa_handler = SIG_IGN };
  sigemptyset(&action.sa_mask);
  if (sigaction(SIGPIPE, &action, NULL)) die("sigaction");

  // Iterieren über die Dateien:
```

```
  // Warten auf die Begleiterthreads:
```

```
  return EXIT_SUCCESS;
}

static int filter(const struct dirent *e) {
```

```
}
```

M:

```
static int startJob(job_t *j, FILE *f) {
```

```
  // Aufteilen in Tokens:
```

```
  // Starten des Programms:
```

```
  return 0;
}
```

S:

```
static void *attendant(void *voidJob) {
```

```
  return NULL;
}
```

A:

```
static void *timeout(void *voidJob) {
```

```
  return NULL;
}
```

T:

```
void V(SEM *sem, unsigned n) {
```

```
}
```

V:

```
void P(SEM *sem, unsigned n) {
}
```

P:

```
static int tcpNotify(char *host, char *name) {
  return writeJobNotification(fd, name);
}
```

N:

Aufgabe 3: Adressräume & Freispeicherverwaltung (8 Punkte)

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

Erklären Sie kurz wie beim **first-fit**- und **best-fit**-Verfahren die Lochliste sortiert ist und nennen Sie einen Vorteil und Nachteil von **first-fit** im Vergleich zu **best-fit**.

(2 Punkte)

2) Man unterscheidet bei Adressraumkonzepten und bei Zuteilungsverfahren zwischen externer und interner Fragmentierung. Erläutern Sie den Unterschied. Kann man gegen die beiden Arten der Fragmentierung jeweils etwas tun und wenn ja, was? (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)

Aufgabe 4: Prozesse (22 Punkte)

1) Beschreiben Sie die Prozesszustände **bei kurz- und mittelfristiger** Einplanung sowie die Ereignisse, die jeweils zu den Zustandsübergängen führen (Skizze mit kurzer Erläuterung der Zustände und Übergänge). Nutzen Sie zur Erklärung der Ereignisse wann immer möglich die konkreten Systemaufrufe, die diese auslösen. (10 Punkte)

2) Man kann Scheduling-Verfahren nach verschiedenen Kriterien in Kategorien einteilen. Beschreiben Sie für die Kategorisierung kooperatives vs. präemptives Scheduling jeweils die Eigenschaften entsprechender Scheduling-Strategien, nennen Sie konkrete Beispiele für solche Strategien und geben Sie an, in welchen Anwendungssituationen entsprechende Strategien sinnvoll eingesetzt werden. (6 Punkte)

3) Es gibt Prozesse, Kernel Threads und User Threads. Beschreiben Sie in Stichworten wie sich diese voneinander unterscheiden. Nennen sie dazu für jede Art je zwei disjunkte Eigenschaften oder Vor-/Nachteile. (6 Punkte)