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 der folgenden Aussagen über kooperative (ohne Verdrängung) und preemptive (mit Verdrängung) Schedulingverfahren ist richtig? 2 Punkte

- ○ Kooperatives Scheduling ist für Steuerungssysteme mit Echtzeitanforderungen völlig ungeeignet.
- ○ Preemptives Scheduling ist für den Mehrbenutzerbetrieb geeignet.
- ○ 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.

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

- ○ 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.
- ○ Der Linux $\mathcal{O}(1)$-Scheduler wurde 2002 von einem $\mathcal{O}(n)$-Scheduler abgelöst.
- ○ Alle vom $\mathcal{O}(1)$-Scheduler genutzten Datenstrukturen werden zwischen allen CPU-Kernen geteilt, um die maximale Performance zu erreichen.

c) Gegeben sei folgende C-Funktion: 2 Punkte

```
void func(void) {
 static int a = 42;
 //[...]
}
```

Welche Aussage zu Lebensdauer und Sichtbarkeit der Variablen `a` ist korrekt?

- ○ Lebensdauer: Funktionslaufzeit, Sichtbarkeit: global
- ○ Lebensdauer: Programmlaufzeit, Sichtbarkeit: global
- ○ Lebensdauer: Funktionslaufzeit, Sichtbarkeit: funktionslokal
- ○ Lebensdauer: Programmlaufzeit, Sichtbarkeit: funktionslokal

d) Gegeben seien die folgenden Präprozessor-Makros: 2 Punkte

```
#define ADD(a, b) a + b
#define DIV(a, b) a / b
```

Was ist das Ergebnis des folgenden Ausdrucks?

```
3 * DIV(ADD(4, 8), 2)
```

- ○ 16
- ○ 24
- ○ 18
- ○ 10

e) Welche Seitennummer (*page number*) und welcher Versatz (*offset)* gehören bei einstufiger Seitennummerierung und einer Seitengröße von 1024 Bytes zu folgender logischen Adresse: 0xc01a 2 Punkte

- ○ Seitennummer 0x30, Versatz 0x1a
- ○ Seitennummer 0xc01, Versatz 0xa
- ○ Seitennummer 0xc0, Versatz 0x1a
- ○ Seitennummer 0xc, Versatz 0x1a

f) Wodurch kann es zu Seitenflattern (*page thrashing*) kommen? 2 Punkte

- ○ Wenn ein Prozess immer abwechselnd physikalische und virtuelle Seiten vom Betriebssystem anfordert.
- ○ Wenn sich zu viele Prozesse im Zustand *blockiert* befinden.
- ○ Durch Programme, die eine Defragmentierung auf der Platte durchführen.
- ○ Wenn ein Prozess zum Weiterarbeiten immer gerade die Seiten benötigt, die durch das Betriebssystem im Rahmen einer globalen Ersetzungsstrategie gerade erst ausgelagert wurden.

g) Welche der genannten Attribute sind in einer Inode eines UNIX-Dateisystems gespeichert? 2 Punkte

- ○ Eigentümer, Dateigröße und Dateityp
- ○ Dateityp, Eigentümer und Dateiname
- ○ Gruppenzugehörigkeit, Anzahl der Verweise und bei Verzeichnissen zusätzlich die Anzahl der enthaltenen Unterverzeichnisse
- ○ Referenzzähler mit der Anzahl der Symbolic Links, die auf die Inode verweisen

h) Welche der Aussage zum Thema Dateisysteme ist richtig?

2 Punkte

- ○ Bei indizierter Speicherung (*Indexed Allocation*) von Dateien müssen unter Umständen mehrere Blöcke geladen werden, bevor der Dateiinhalt gelesen werden kann.
- ○ Journaling-Dateisysteme sind immun gegen defekte Plattenblöcke.
- ○ Bei indizierter Speicherung (*Indexed Allocation*) kann es prinzipbedingt nicht zu Verschnitt kommen.
- ○ Bei kontinuierlicher Speicherung (*Contiguous Allocation*) ist es immer problemlos möglich, bestehende Dateien zu vergrößern.

i) Welche der folgenden Aussagen über UNIX-Dateisysteme ist richtig?

2 Punkte

- ○ Auf ein Verzeichnis darf immer nur genau ein *hard link* verweisen.
- ○ *Hard links* auf Dateien können nur innerhalb des Dateisystems angelegt werden, in dem auch die Datei selbst liegt.
- ○ Auf eine Datei in einem Dateisystem verweisen immer mindestens zwei *hard links*.
- ○ Wenn der letzte *symbolic link*, der auf eine Datei verweist, gelöscht wird, wird auch die Datei und deren Inode gelöscht.

j) Beim Einsatz von RAID-Systemen kann durch zusätzliche Festplatten Fehlertoleranz erzielt werden. Welche Aussage dazu ist richtig?

2 Punkte

- ○ Bei RAID 6 darf eine bestimmte Menge von Festplatten nicht überschritten werden, da es sonst nicht mehr möglich ist, die Paritätsinformation zu bilden.
- ○ Bei RAID 4 Systemen wird Paritätsinformation gleichmäßig über alle beteiligten Platten verteilt.
- ○ RAID 0 erzielt Fehlertoleranz durch das Verteilen der Daten auf mehrere Platten.
- ○ Bei RAID-Systemen ist ein höherer Lese-Durchsatz als bei einer einzelnen Platte möglich, da mehrere Platten parallel Leseanfragen bearbeiten können.

k) Welche der Aussage zum Thema Threads ist richtig?

2 Punkte

- ○ Bei Threads (*Light-weight Process*) ist die Schedulingstrategie durch das Betriebssystem vorgegeben.
- ○ Zu jedem Thread (*Light-weight Process*) gehört ein eigener Adressraum.
- ○ Bei User-level Threads (*Feather-weight Processes*) ist die Schedulingstrategie durch das Betriebssystem vorgegeben.
- ○ Zu jedem User-level Thread (*Feather-weight Processes*) gehört ein eigener Adressraum.

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 Threads und Prozesse sind richtig?

4 Punkte

- ☐ Die Veränderung von Variablen und Datenstrukturen in einem mittels `fork()` erzeugten Kindprozess beeinflusst auch die Datenstrukturen im Elternprozess.
- ☐ Threads (*Light-weight Processes*) können Multiprozessoren ausnutzen.
- ☐ User-level Threads (*Feather-weight Processes*) blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ☐ Zur Umschaltung von User-level Threads (*Feather-weight Processes*) ist ein Adressraumwechsel erforderlich.
- ☐ Mittels `fork()` erzeugte Kindprozesse können in einem Multiprozessor-System nur auf dem Prozessor ausgeführt werden, auf dem auch der Elternprozess ausgeführt wird.
- ☐ Die Einplanung und Einlastung von User-level Threads (*Feather-weight Processes*) findet ohne Wissen des Betriebssystems in der Anwendung statt.
- ☐ Der Aufruf von `fork()` gibt im Elternprozess die Prozess-ID des Kindprozesses zurück, im Kindprozess hingegen den Wert 0.
- ☐ Threads (*Light-weight Processes*) teilen sich den kompletten Adressraum und verwenden daher den selben Stack.

b) Welche der folgenden Aussagen zu prioritätsbasierten Scheduling-Verfahren sind richtig?

4 Punkte

- ☐ Prioritätsumkehr (*priority inversion*) meint, dass ein Prozess mit niedriger Priorität abgebrochen wird (*umkehrt*), damit ein Prozess mit höherer Priorität laufen kann.
- ☐ Prioritätsumkehr (*priority inversion*) meint, dass ein Prozess mit niedriger Priorität einen Prozess mit höherer Priorität durch das Belegen einer geteilten Ressource blockiert.
- ☐ Prioritätsumkehr (*priority inversion*) kann nur mit dynamischen Prioritäten auftreten.
- ☐ Prioritätsumkehr (*priority inversion*) kann durch Prioritätsgrenzen (*priority ceiling protocols*) verhindert werden.
- ☐ Bei Multilevel Feedback Queues können Prozesse ggf. in höhere Queues zurückwechseln, falls sie lange laufen (*anti-aging*).
- ☐ Bei Multilevel Feedback Queues werden Prozesse, die zu lange laufen, abgebrochen (*anti-aging*), um sicherzustellen, dass andere Prozesse nicht verhungern (*starvation*).
- ☐ Kurze Prozesse werden bei Multilevel Feedback Queues generell bevorzugt.
- ☐ Queues in Multilevel Feedback Queues werden generell per Round Robin verwaltet.

Aufgabe 2: `witch` (60 Punkte)

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

Schreiben Sie ein POSIX-1.2008 und C11-konformes Programm `witch`, welches parallel die in der Umgebungsvariable `PATH` aufgeführten Verzeichnisse nach Dateien mit einem bestimmten Namen durchsucht. Wird zusätzlich beim Aufruf von `witch` das Befehlszeilenargument `--hunt` übergeben, soll versucht werden die gefundenen Dateien zusätzlich auch zu löschen.

Die Umgebungsvariable `PATH` kann mittels `getenv()` (siehe angehängte Manpages) ausgelesen werden und enthält beliebig viele, durch Doppelpunkt `:` getrennte Pfade, die alle von `witch` durchsucht werden sollen. Beispiel für den Inhalt der `PATH`-Variable:
`/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`

Zugehöriger, beispielhafter Aufruf von `witch`:

```
herzog@manwe:~$ ./witch --hunt ls
/usr/bin/ls
/bin/ls
```

Das Programm soll folgendermaßen strukturiert sein:

- Funktion `main()`: Prüft die Befehlszeilenargumente, initialisiert benötigte Datenstrukturen und liest die Umgebungsvariable `PATH` aus. Für jedes Verzeichnis in `PATH` wird ein Thread gestartet, der die Suche innerhalb des Verzeichnisses durchführt. Der Einstiegspunkt für einen Such-Thread ist die Funktion `thread_start()`. Nach dem Starten der Such-Threads, wartet der Hauptthread darauf, dass sich alle Such-Threads beendet haben. Die Such-Threads hinterlegen die Pfade aller gefundenen Dateien in einer Liste, deren Implementierung gegeben ist. Nachdem sich alle Such-Threads beendet haben, entnimmt der Hauptthread per `removeElement()` alle Pfade aus der Liste und gibt diese aus. Anschließend gibt er alle Ressourcen frei. Wurde mindestens eine Datei gefunden, beendet sich der Hauptthread mit dem Exitstatus `EXIT_SUCCESS`, ansonsten mit `EXIT_FAILURE` und einer Fehlermeldung.
- Funktion `void *thread_start(void *arg)`:
  Konvertiert das übergebene Argument `arg` (Zeiger auf den gegebenen Typ `search_t`) und ruft die Funktion `search_dir()` passend auf. Anschließend gibt die Funktion `NULL` zurück.
- Funktion `void search_dir(char *search, char *dir, bool hunt)`:
  Iteriert über alle Einträge des Verzeichnisses `dir`. Falls ein Verzeichniseintrag eine *reguläre Datei* ist, wird geprüft, ob der Name der Datei dem Suchstring `search` entspricht. Falls der Name dem Suchstring entspricht, wird der Pfad zu der gefundenen Datei per `insertElement()` in die Liste eingehängt. Falls der Parameter `hunt` wahr ist, wird anschließend versucht die Datei zu löschen (`unlink()`, siehe angehängte Manpage), wenn der Name dem Suchstring entspricht. Falls ein Verzeichniseintrag ein Verzeichnis ist, wird rekursiv `search_dir()` aufgerufen, um das Verzeichnis ebenfalls zu durchsuchen.
  Sollten Funktionen mit Dateisystembezug fehlschlagen, so soll `witch` eine entsprechende Fehlermeldung ausgeben und mit der Bearbeitung des nächsten Eintrags fortsetzen.

Hinweise:

- Die Listenfunktionen `insertElement()` und `removeElement()` dürfen **nicht** nebenläufig aufgerufen werden, entsprechend **ist auf korrekte Synchronisation zu achten**.
- Ihnen steht das aus der Übung bekannte Semaphor-Modul zur Verfügung (sh. Manpages).
- Fehler bei der Ausführung von dateisystembezogenen Funktionen soll nicht zum Abbruch des Programms führen.
- Es ist **keine** Fehlerbehandlung/`fflush()` für Ausgaben auf `stdout` nötig. Achten Sie ansonsten auf korrekte und vollständige Fehlerbehandlung.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdbool.h>
#include <dirent.h>
#include <pthread.h>

#include "list.h"
#include "sem.h"

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

static void err(const char msg[]) {
  fprintf(stderr, "%s\n", msg);
  exit(EXIT_FAILURE);
}

static void usage(void) {
  err("Usage: witch [--hunt] <search>");
}

typedef struct {
  char *search; // Suchstring
  char *dir;    // zu durchsuchendes Verzeichnis
  bool hunt;    // Befehlszeilenargument --hunt übergeben
} search_t;
```

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

// Funktion main()

// Deklarationen etc.

// Befehlszeilenargumente prüfen

// Initialisierungen und PATH auslesen

```
// Verzeichnisse aus PATH extrahieren
```

```
// Such-Threads starten und auf Beendigung warten
```



```
// Ausgabe der gefundenen Dateien
```

```
// Aufräumen und Beenden
```

```
// Ende Funktion main()
```

M:

```
// Funktion thread_start()
```

```
// Ende Funktion thread_start()
```

T:

```
// Funktion search_dir()

  // Verzeichnis öffnen

  // Über Verzeichniseinträge iterieren
```



```
  // Aufräumen

// Ende Funktion search_dir()
```

S:

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

Das Target `clean` soll alle erzeugten Zwischenergebnisse und das Programm `witch` 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: Synchronisation (16 Punkte)

Skizzieren Sie in Programmiersprachen-ähnlicher Form, wie mit Hilfe eines zählenden Semaphors die folgenden Szenarien korrekt synchronisiert werden können. Ihnen stehen dabei folgende Semaphor-Funktionen zur Verfügung:

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

Beachten Sie, dass nicht unbedingt alle freien Zeilen für eine korrekte Lösung nötig sind. **Kennzeichnen Sie durch /, wenn Ihre Lösung in einer freien Zeile keine Operation benötigt.** Jede korrekt beschriftete Zeile gibt einen halben Punkt, jede falsch oder nicht beschriftete einen halben Punkt Abzug. Jede Teilaufgabe wird minimal mit 0 Punkten gewertet. Fehlerbehandlung ist in dieser Aufgabe **nicht** notwendig. Achten Sie jedoch auf das Freigeben von angeforderten Ressourcen.

1) Zu jedem Zeitpunkt sollen so viele Arbeiterthreads wie möglich, maximal jedoch 8 gleichzeitig, laufen. Ein Arbeiterthread zählt bis zur Rückkehr aus `doWork()` in das Limit der maximal laufenden Arbeiterthreads. (3 Punkte)

Hauptthread:

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

  while(!finished) {

    startWorkerThread(threadFunc);

  }

}
```

Arbeiterthread:

```
void threadFunc(void) {

  doWork();

}
```

2) Der Hauptthread soll nach jeder Statistik-Änderung durch einen Arbeiterthread (`stat = l_stat`) die aktuellen Statistiken ausgeben (`printStats()`) und dazwischen passiv warten. Achten Sie darauf, dass sich weder `doWork()` noch `printStats()` in einem kritischen Abschnitt befinden, um eine möglichst effiziente Abarbeitung des Programms zu gewährleisten.

Achten Sie außerdem darauf **alle** nicht benötigten Zeilen durch `/` zu kennzeichnen. (7 Punkte)

Hauptthread:

```
static SEM *mutex;
static SEM *notify;
static stat_t stat;

int main(void){

-------------------------------------

-------------------------------------
  startAllWorkerThreads()
  while(!finished) {
-------------------------------------

-------------------------------------
    stat_t l_stat = stat;

-------------------------------------

-------------------------------------
    printStats(l_stat);
-------------------------------------
  }

-------------------------------------

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

Arbeiterthread:

```
void threadFunc(void) {

-------------------------------------
  stat_t l_stat = doWork();
-------------------------------------

-------------------------------------
  stat = l_stat;

-------------------------------------

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

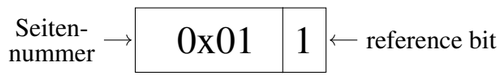

3) Was versteht man unter einer Verklemmung (*Deadlock*) und was sind die (hinreichenden und notwendigen) Bedingungen, damit eine Verklemmung auftreten kann? (6 Punkte)

Aufgabe 4: Speicherverwaltung (14 Punkte)

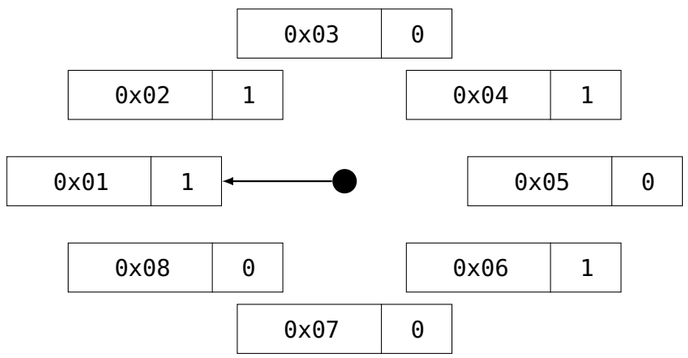
1) Eine in der Praxis gut einsetzbare Strategie ist Second Chance: CLOCK. In dieser Aufgabe soll CLOCK als (Prozess-)lokale Seitenersetzungsstrategie eingesetzt werden. (10 Punkte)

Im Folgenden sind die für die Seitenverwaltung erforderlichen Daten der aktuell anwesenden Seiten eines Prozesses dargestellt. Nehmen Sie eine Seitengröße von 4096 Bytes an (ergibt 12 Bit Offset). Gehen Sie davon aus, dass alle Seiten les- und schreibbar sind und alle zugegriffenen Adressen gültig sind.

Hinweis: Der CLOCK-Zeiger zeigt jeweils auf den Eintrag, der bei der nächsten Suche nach einem freien Seitenrahmen (*page frame*) als erstes überprüft wird.



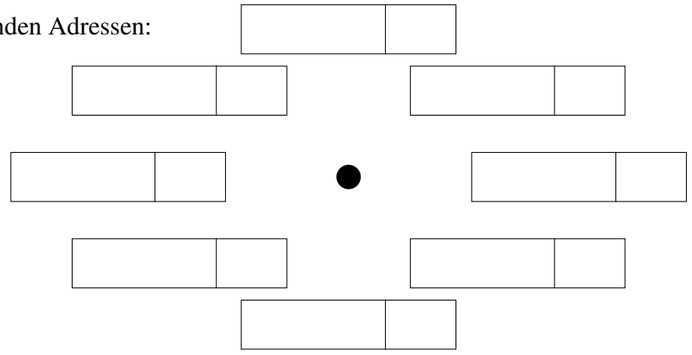
Ausgangszustand:



Nach Zugriff auf die folgenden Adressen:

Lesen von `0x03` `120`

Schreiben nach `0x0a` `42c`

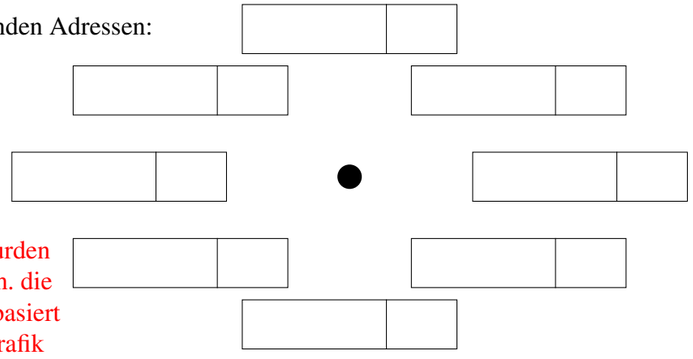


Nach Zugriff auf die folgenden Adressen:

Lesen von `0x08` `120`

Schreiben nach `0x05` `f0c`

Schreiben nach `0x07` `00a`



Hinweis: In der Korrektur wurden Folgefehler berücksichtig, d.h. die Korrektur des unteren Bilds basiert auf den Werten der oberen Grafik

2) Virtualisierte Speicherverwaltung benötigt Unterstützung durch die Hardware. Nennen Sie die benötigte Hardware und nötige Zusatzeinträge in Seitendeskriptoren um Strategien wie CLOCK zu implementieren. (2 Punkte)

3) Falls kein freier Seitenrahmen im Hauptspeicher verfügbar ist, muss eine Seite ausgelagert werden. Nennen Sie zwei mögliche Strategien zur Bestimmung der zu verdrängenden Seite (ausgenommen Second Chance, CLOCK). Beschreiben Sie die Strategien jeweils kurz. (2 Punkte)

Alternativlösung für Schritt 2 von Aufgabe 4.1:
Aus der Aufgabenstellung wird nicht ersichtlich, dass
Daher ist es auch valide Schritt 2 auf Basis vom Ausg

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!