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) Bei Programmunterbrechungen (Ausnahmen) unterscheidet man zwischen Traps und Interrupts. Welche Aussage zu Traps ist richtig? 2 Punkte

- ☐ Der Zugriff auf eine physikalische Speicheradresse kann zu einem Trap führen.
- ☐ Normale Ganzzahl-Rechenoperationen (z. B. Addition, Division) können nicht zu einem Trap führen.
- ☐ Ein Trap steht nicht zwangsläufig in ursächlichem Zusammenhang mit dem unterbrochenen Programm.
- ☐ Traps werden immer nach dem Beendigungsmodell/Terminierungsmodell behandelt.

b) 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 steht bei ausgelagerten Seiten eine Adresse des Hintergrundspeichers und der Speichercontroller leitet den Zugriff auf den Hintergrundspeicher um.
- ☐ Bei ausgelagerten Seiten ist im Seitendeskriptor das „present bit“ nicht gesetzt. Das Betriebssystem erkennt dies und löst bei einer Adressauflösung für solch eine Seite einen Trap aus.
- ☐ 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.

c) Welche der folgenden Aussagen über UNIX-Dateisysteme ist richtig? 2 Punkte

- ☐ Wenn der letzte symbolic link, der auf eine Datei verweist, gelöscht wird, wird auch der zugehörige Dateikopf (inode) gelöscht.
- ☐ Hard links können innerhalb des selben Datenträgers auf beliebige Blöcke zeigen.
- ☐ In einem Verzeichnis darf es keinen Eintrag geben, der auf das Verzeichnis selbst verweist.
- ☐ Für Zugriff über verschiedene Hard links auf die selbe Datei gelten identische Zugriffsrechte

d) Bei einer prioritätengesteuerten Prozessauswahl-Strategie (Scheduling-Strategie) kann es zu Problemen kommen. Welches der folgenden Probleme kann auftreten? 2 Punkte

- ☐ Das Phänomen der Prioritätsumkehr hungert niedrigpriore Prozesse aus.
- ☐ Ein hochpriorer Prozesse muss evtentuell auf ein Betriebsmittel warten, das von einem niedrigprioren Prozess exklusiv benutzt wird. Der niedrigpriore Prozess kann das Betriebsmittel jedoch wegen eines mittelhochprioren Prozesses nicht freigeben (Prioritätenumkehr).
- ☐ 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.

e) Welche der folgenden Aussagen zum Thema persistenter Datenspeicherung sind richtig? 2 Punkte

- ☐ Bei kontinuierlicher Speicherung von Daten ist es unter Umständen mit enormem Aufwand verbunden, eine bestehende Datei zu vergrößern.
- ☐ Bei indizierter Speicherung kann es prinzipbedingt nicht zu Verschnitt kommen.
- ☐ Im Vergleich zu den anderen Verfahren ist bei indizierter Speicherung die Positionierzeit des Festplatten-Armes beim Zugriff auf alle Datenblöcke einer Datei minimal.
- ☐ Extents finden aus Performanzgründen keine Anwendung in modernen Dateisystemen.

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

- ☐ Auf Multiprozessorsystemen kann die Umschaltung von Kern-Threads ohne Mitwirken des Systemkerns erfolgen.
- ☐ Kern-Threads teilen sich den kompletten Adressraum und verwenden daher den selben Stack.
- ☐ Bei User-Threads ist die Schedulingstrategie keine Funktion des Betriebssystemkerns.
- ☐ Die Umschaltung von Threads muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl).

g) Welche der folgenden Aussagen zum Thema Synchronisation sind richtig? 2 Punkte

- ☐ Ein Semaphor kann ausschließlich für mehrseitige Synchronisation verwendet werden.
- ☐ Zur Synchronisation eines kritischen Abschnitts ist passives Warten immer besser geeignet als aktives Warten.
- ☐ Für nicht-blockierende Synchronisationsverfahren ist spezielle Unterstützung durch das Betriebssystem notwendig.
- ☐ Monitore sind Datentypen mit impliziten Synchronisationseigenschaften.

h) 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 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.

i) Welche der folgenden Aussagen zum Thema Seitenfehler (page fault) ist richtig?

2 Punkte

- ☐ Ein Seitenfehler zieht eine Ausnahmebehandlung nach sich. Diese wird dadurch ausgelöst, dass die MMU das Signal SIGSEGV an den aktuell laufenden Prozess schickt.
- ☐ Seitenfehler können auch auftreten, obwohl die entsprechende Seite gerade im physikalischen Speicher vorhanden ist.
- ☐ Wenn der gleiche Seitenrahmen in zwei verschiedenen Seitendeskriptoren eingetragen wird, löst dies einen Seitenfehler aus (Gefahr von Zugriffskonflikten!).
- ☐ Ein Seitenfehler wird ausgelöst, wenn der Offset in einer logischen Adresse größer als die Länge der Seite ist.

j) Welches der folgenden Verfahren trägt in der Praxis am besten dazu bei, die Auswirkungen eines Seitenfehlers zu minimieren?

2 Punkte

- ☐ Man lagert regelmäßig länger nicht genutzte Seiten aus und trägt sie in einem Freiseitenpuffer ein.
- ☐ Man ermittelt, welche der Seiten eines Prozesses in Zukunft am längsten nicht angesprochen wird und lagert genau diese aus (OPT Strategie).
- ☐ Man setzt eine Segmentierung in Kombination mit Seitenadressierung ein.
- ☐ Man übergibt Prozesse, die einen Seitenfehler verursachen der mittelfristigen Prozesseinplanung, damit sie in nächster Zeit nicht wieder aktiv werden.

k) Welche der folgenden Aussagen über Einplanungsverfahren ist richtig?

2 Punkte

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

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 zu UNIX/Linux-Dateideskriptoren sind korrekt?

4 Punkte

- ○ Dateideskriptoren sind Zeiger auf Betriebssystem-interne Strukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- ○ Ein Dateideskriptor ist eine Verwaltungsstruktur, die auf der Festplatte gespeichert ist und Informationen über Größe, Zugriffsrechte, Änderungsdatum usw. einer Datei enthält.
- ○ Ist das Flag `FD_CLOEXEC` eines Dateideskriptors gesetzt, dann wird dieser Dateideskriptor geschlossen, sobald der Prozess eine Funktion der `exec`-Familie aufruft.
- ○ Beim Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Dateideskriptor zum Zugriff zurück.
- ○ Ein Dateideskriptor ist eine Integerzahl, die über gemeinsamen Speicher an einen anderen Prozess übergeben werden kann und von letzterem zum Zugriff auf eine geöffnete Datei verwendet werden kann.
- ○ Beim Aufruf von `fork()` werden zuvor geöffnete Dateideskriptoren in den Kindprozess vererbt.
- ○ Auch Netzwerkverbindungen werden über einen Dateideskriptor referenziert.
- ○ Ein Dateideskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei benutzen kann.

b) Welche der folgenden Aussagen zum Thema Prozesszustände sind richtig?

4 Punkte

- ○ Es können sich maximal genauso viele Prozesse gleichzeitig im Zustand laufend befinden, wie Prozessorkerne vorhanden sind.
- ○ Im Rahmen der mittelfristigen Einplanung kann ein Prozess von Zustand laufend in den Zustand schwebend laufend wechseln.
- ○ Bei Eintreffen eines Interrupts wird der aktuell laufende Prozess für die Dauer der Interrupt-Abarbeitung in den Zustand blockiert überführt.
- ○ Ein Prozess kann nur durch seine eigene Aktivität vom Zustand laufend in den Zustand blockiert überführt werden.
- ○ Das Auftreten eines Seitenfehlers kann dazu führen, dass der aktuell laufende Prozess in den Zustand beendet überführt wird.
- ○ Greift ein laufender Prozess lesend auf eine Datei zu und der entsprechende Datenblock ist nicht im Hauptspeicher vorhanden, dann wird der Prozess in den Zustand bereit überführt.
- ○ Bei kooperativem Scheduling ist kein direkter Übergang vom Zustand laufend in den Zustand bereit möglich.
- ○ Die V-Operation eines Semaphors kann bewirken, dass ein Prozess vom Zustand blockiert in den Zustand bereit überführt wird.

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

Aufgabe 2: `rethceaW` (60 Punkte)

1) `Waechter` (50 Punkte)

Schreiben Sie ein Programm das auf IPv6 Port 2020 auf eingehende Verbindungen wartet und diese in einem Kindprozess bearbeitet. Die konkrete Bearbeitung ist in der Funktion `int handle(int fd)` gekapselt und für die Implementierung von `Waechter` vorerst nicht weiter von Interesse. Der Rückgabewert diser Funktion soll von dem Kindprozess als Exit-Status verwendet werden. Das Annehmen einer neuen Verbindungen ist nur erlaubt, falls weniger als 5 Verbindungen gerade behandelt werden. Des Weiteren ist vorgeschrieben, dass der Kontakt zwischen Klienten und Kindprozess maximal 60 Sekunden andauern darf. Dazu soll `Waechter` die Funktion `set_deadline` verwenden, die als Parameter einen Zeitstempel erwartet und dem Prozess zu dieser Zeit das Signal `SIGALRM` zustellt. Ein Zeitstempel kann durch die Funktion `make_timestamp` erzeugt werden. Nutzen Sie dies, um im passenden Signalhandler alle Kinder, die Ihre Kontaktzeit überschritten haben mit dem Signal `SIGKILL` zu beenden. Beachten Sie, dass ein Aufruf von `set_deadline` einen vorher gesetzten Alarm überschreibt. Stellen Sie außerdem sicher, dass das Standardverhalten von `SIGALRM` beim Aufruf von `handle` wieder hergestellt ist.

Zur Verwaltung der Verbindungs-Slots ist eine Listenimplementierung für `struct contact` Strukturen vorgegeben. Vor der ersten Verwendung muss jede Liste mit Hilfe von `list_init` in einen gültigen Zustand gebracht werden. Die Funktionen `list_enqueue` und `list_dequeue` erlauben die Liste nach dem FIFO-Prinzip zu nutzen. Anhand der `pid` kann ein spezifischer Eintrag via `list_remove` aus der Liste entfernt werden. Ohne Modifikation der Liste ist mit `list_peek` der Zugriff auf das erste Element möglich. Die Implementierung erlaubt keinen parallelen Zugriff auf die selbe Listeninstanz.

Die ggf. notwendige Synchronisierung muss durch Sie erfolgen.

Implementieren Sie die folgenden Funktionen:

`void handle_sigalrm(int)` Sendet an Kinder, deren maximale Kontaktzeit abgelaufen ist `SIGKILL` und verschiebt deren Verwaltungsstruktur in die Liste `killed`.

`void handle_sigchld(int)` Sammelt alle beendeten Kindprozesse auf und stellt sicher, dass die zugehörigen `struct contact` Objekte wiederverwendet werden können.

`struct contact *wait_for_slot(void)` Sollten noch nicht alle erlaubten Verbindungen belegt sein, wird der nächste Eintrag aus der Liste `unused` zurückgegeben. Andernfalls wird **passiv gewartet** bis eine neue Verbindung erlaubt ist.

`void server_loop(int ls)` Nimmt wiederholt Verbindungen auf `ls` an und lagert deren Behandlung in einen Kindprozess aus. Verwendet die Funktion `wait_for_slot` um das Verbindungslimit einzuhalten. Falls nötig, wird ein Alarm mit `set_deadline` aufgesetzt und das zugehörige `struct contact` an die Liste `running` angehängt.

`int main(void)` Initialisiert die globalen Strukturen und hängt 5 freie Slots in Form von `struct contact` Objekten in die Liste `unused`. Desweiteren wird die Signalbehandlung sowie ein Socket, der auf Port 2020 lauscht konfiguriert. Schließlich wird der Socket an die Funktion `server_loop` zur Annahme von Verbindungen weitergereicht.

Hinweise:

- Das Programm darf nur beenden werden, wenn kein sinnvoller Weiterbetrieb mehr möglich ist.
- Signalbehandlungsfunktionen dürfen alle vorgegben Funktion verwenden (async-signal-safe).
- In einem Kindprozess wird der mit `set_deadline` aktivierte Alarm deaktiviert, ausstehende Signale werden aber weiterhin zugestellt.
- Die Standardbehandlung von `SIGALRM` terminiert den Prozess.

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

```c
#include <signal.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#define MAX_CONTACT_TIME 60
#define MAX_CONTACTS     5
#define PORT             2020

typedef unsigned long time_t;
typedef struct contact {
  pid_t  pid;
  time_t deadline;
} contact_t;
struct list { /* ... */ };

// @list initialisieren.
void list_init(struct list *list);

// Rückgabe: Erstes Element in @list oder NULL, falls leere Liste.
struct contact *list_peek(struct list *list);

// @item an das Ende von @list anfügen.
void list_enqueue(struct list *list, struct contact *item);

// Entfernt das erste Element aus @list.
// Rückgabe: Erstes Element in @list oder NULL, falls leere Liste.
struct contact *list_dequeue(struct list *list);

// Entfernt das erste Element von @list mit passender @pid.
// Rückgabe: entferntes Element oder NULL, falls kein Treffer.
struct contact *list_remove(struct list *list, pid_t pid);

// Behandlung einer angenommenen Verbindung.
int handle(int fd);

// Erzeugt eine Zeitstempel, der @seconds nach dem Aufrufzeitpunkt ist.
time_t make_timestamp(unsigned seconds);

// Setzt einen neuen Zeitstempel für SIGALRM. Gibt den zuletzt gesetzten
// Zeitstempel zurueck oder 0 falls inaktiv.
time_t set_deadline(time_t timestamp);

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

static struct list running, killed, unused;
```



```c
static void handle_sigalrm(int sig) {
}

static void handle_sigchld(int sig) {
}
```

H:

```
static struct contact *wait_for_slot(void) {
}

static void server_loop(int ls) {
  struct contact *slot = NULL;
  for (;;) {
    // Warten auf erlaubte Verbindung
```

W:


```
    // Verbindungsbehandlung auslagern
```

```
  }
}

int main(void) {
  // globale Daten initialisieren

  // Behandlung von Signalen aufsetzen
```

S:


```
  // Erstellen des Netzwerksockets und server_loop Aufruf

}
```

M:

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

2) `Inversion` (10 Punkte)

In dieser Teilaufgabe sollen Sie ein Modul schreiben, das die von `Wächter` genutzte Funktion `int handle(int fd)` implementiert. Funktion des Modules soll sein, die Eingabe der Gegenstelle entgegenzunehmen und die Zeichen in umgekehrter Reihenfolge zurückzusenden. Dazu wird bis zum `EOF` Zeichen die Eingabe gelesen und zwischengespeichert.

Hinweis: Da `handle` in einem Kindprozess ausgeführt wird, kann im Fehlerfall auch die vorgegebene Funktion `die` verwendet werden um sich direkt zu beenden!

```c
#include <stdio.h>
#include <stdlib.h>

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



```c
int handle(int fd) {
  // Erstellen der FILE* Abstraktionen


  // Einlesen der Eingabe


  // Ausgabe der Daten


}
```

I:

Aufgabe 3: Koordinierung (13 Punkte)

Zur Koordinierung von nebenläufigen Vorgängen, die auf gemeinsame Betriebsmittel zugreifen, unterscheidet man zwischen einseitiger und mehrseitiger Synchronisation.

1) Beschreiben Sie die Unterschiede zwischen einseitiger und mehrseitiger Synchronisation. (2 Punkte)

2) Synchronisation erfolgt in vielen Fällen mittels blockierender Verfahren. Welche Probleme sind mit dem Einsatz blockierender Synchronisation verbunden? (4 Punkte)

3) In manchen Situationen kann man statt mit blockierender Synchronisation mit optimistischen, nicht-blockierenden Synchronisationsverfahren arbeiten. Beschreiben Sie an einem kleinen Beispiel (z.B. unter Verwendung eines CAS-Befehls), wie solch ein Verfahren vom Prinzip her arbeitet. Nennen Sie jeweils zwei Vor- und Nachteile solcher Verfahren. (7 Punkte)

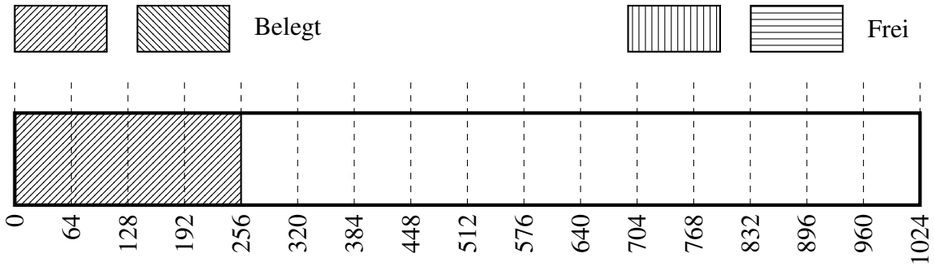
Aufgabe 4: Adressräume & Freispeicherverwaltung (17 Punkte)

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 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, und tragen Sie in der Tabelle den aktuellen Zustand der Lochliste 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								
2^8	256							
2^9	512							
2^{10}								

2) 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)

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)