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 über Prozesszustände ist in einem Monoprozessor-Betriebssystem mit blockierenden Ein-/Ausgabeoperationen richtig? 2 Punkte

- ○ Wenn gerade keine Prozessumschaltung stattfindet und ein Prozess im Zustand *laufend* ist, so gibt es mindestens einen Prozess im Zustand *blockiert*.
- ○ Wenn gerade keine Prozessumschaltung stattfindet und kein Prozess im Zustand *laufend* ist, so ist auch kein Prozess im Zustand *blockiert*.
- ○ Ein Prozess im Zustand *laufend* wird in den Zustand *blockiert* überführt, wenn eine seiner Ein-/Ausgabeoperation nicht sofort abgeschlossen werden kann.
- ○ Es befinden sich bis zu zwei Prozesse im Zustand *laufend* und damit in Ausführung auf dem Prozessor (Vordergrund- und Hintergrundprozess).

b) Welche Aussage zum Thema Adressraumschutz ist richtig? 2 Punkte

- ○ Bei allen Verfahren des Adressraumschutzes führt jeder Zugriff auf eine ungültige Speicheradresse zu einem Trap.
- ○ Adressraumschutz durch Abgrenzung erlaubt es, mehrere Benutzerprozesse voneinander zu isolieren.
- ○ Beim Adressraumschutz durch Abgrenzung wird der logische Adressraum in mehrere Segmente mit unterschiedlicher Semantik unterteilt.
- ○ Adressraumschutz durch Abgrenzung eignet sich besonders für Systeme, die von mehreren Nutzern gleichzeitig verwendet werden.

c) Ein Programm will die drei Zeichenketten `char a[] = "path"; char b[] = "to"; char c[] = "video";` 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? 2 Punkte

- ○ 12
- ○ 13
- ○ 14
- ○ 15

d) Welche Antwort trifft für die Eigenschaften eines Linux-Dateideskriptors zu? 2 Punkte

- ○ Ein Dateideskriptor ist eine prozesslokale Integerzahl, die der Prozess zum Zugriff auf eine Datei benutzen kann.
- ○ Dateideskriptoren sind Zeiger auf betriebssystem-interne Strukturen, die von den Systemaufrufen ausgewertet werden, um auf Dateien zuzugreifen.
- ○ 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 Öffnen ein und derselben Datei erhält ein Prozess jeweils die gleiche Integerzahl als Dateideskriptor zum Zugriff zurück.

e) Ein laufender Prozess wird durch den Scheduler verdrängt. Welcher Zustandsübergang findet statt? 2 Punkte

- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *blockiert*.
- ○ Der Prozess wechselt vom Zustand *bereit* in den Zustand *blockiert*.
- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *beendet*.
- ○ Der Prozess wechselt vom Zustand *laufend* in den Zustand *bereit*.

f) Beim Einsatz von RAID-Systemen kann durch zusätzliche Festplatten ein fehlertolerierendes Verhalten erzielt werden. Welche Aussage dazu ist richtig? 2 Punkte

- ○ Bei einem RAID-5-System kommen immer genau 5 Platten zum Einsatz. Die Parity-Information wird dabei auf alle 5 Platten gleichmäßig verteilt.
- ○ Bei allen RAID-Systemen ist ein höherer Lese-Durchsatz als bei einer einzelnen Platte möglich, da mehrere Platten gleichzeitig beauftragt werden können.
- ○ Bei allen RAID-Systemen ist ein höherer Schreib-Durchsatz als bei einer einzelnen Platte möglich, da alle Platten gleichzeitig beauftragt werden können.
- ○ RAID 0 erzielt Fehlertoleranz durch das Verteilen der Daten auf mehrere Platten.

g) Welche Aussage zum Thema Adressräume ist richtig? 2 Punkte

- ○ Der physikalische Adressraum ist durch die gegebene Hardwarekonfiguration definiert.
- ○ Der virtuelle Adressraum eines Prozesses kann nie größer sein als der physikalisch vorhandene Arbeitsspeicher.
- ○ Die maximale Größe des virtuellen Adressraums kann unabhängig von der verwendeten Hardware frei gewählt werden.
- ○ Virtuelle Adressräume sind Voraussetzung für die Realisierung logischer Adressräume.

h) Welche Aussage zu Zeigern ist richtig?

2 Punkte

- ○ Zeiger können verwendet werden, um in C eine call-by-reference Übergabesemantik nachzubilden.
- ○ Zeiger können in C nicht als Parameter an Funktionen übergeben werden.
- ○ Ein Zeiger kann zur Manipulation von schreibgeschützten Datenbereichen verwendet 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.

i) Welche Aussage über fork() ist richtig?

2 Punkte

- ○ Der Kind-Prozess bekommt die Prozess-ID des Eltern-Prozesses zurückgegeben.
- ○ Der Aufruf von fork() ersetzt das im aktuellen Prozess laufende Programm durch das als Parameter angegebene Programm.
- ○ Im Fehlerfall wird im Kind-Prozess -1 zurückgeliefert.
- ○ Dem Eltern-Prozess wird die Prozess-ID des neu erzeugten Kind-Prozesses zurückgeliefert.

j) Der Speicher eines UNIX-Prozesses ist in Text-, Daten- und Stack-(Stapel-)Segment untergliedert. Welche Aussage zur Platzierung von Daten in diesen Segmenten ist richtig?

2 Punkte

- ○ Globale Variablen liegen im Daten-Segment.
- ○ Dynamisch allozierte Zeichenketten liegen im Text-Segment.
- ○ Lokale static-Variablen werden bei jedem Betreten der zugehörigen Funktion neu initialisiert.
- ○ Bei einem Aufruf von malloc(3) wird das Stack-Segment dynamisch erweitert.

k) Welche Aussage zu Speicherzuteilungsstrategien ist richtig?

2 Punkte

- ○ Bei worst-Fit ist das Verschmelzen freier Blöcke besonders einfach.
- ○ first-fit setzt eine aufsteigende Sortierung der Freispeicherliste nach Adresse der Speicherblöcke voraus.
- ○ Bei allen listenbasierten Zuteilungsverfahren (First-, Next-, Best-, Worst-Fit) kann externer Verschnitt auftreten.
- ○ Best-fit ist in jedem Fall das beste Verfahren.

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 ist richtig?

4 Punkte

- ☐ Bei *Kernel-Level Threads* ist die Schedulingstrategie durch das Betriebssystem implementiert.
- ☐ *Kernel-Level Threads* blockieren sich bei blockierenden Systemaufrufen gegenseitig.
- ☐ *Kernel-Level Threads* teilen sich den kompletten Adressraum und verwenden daher denselben Stack.
- ☐ *Kernel-Level Threads* setzen den Einsatz von verdrängenden Schedulingverfahren voraus.
- ☐ *Kernel-Level Threads* können Multiprozessoren ausnutzen.
- ☐ Die Umschaltung von *User-Level Threads* ist sehr effizient.
- ☐ Die Umschaltung von *User-* und *Kernel-Level Threads* muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl).
- ☐ Bei *User-Level Threads* ist die Schedulingstrategie durch die Anwendung bzw. die von ihr genutzten Bibliotheken vorgegeben.

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

4 Punkte

- ☐ In einem UNIX-Dateisystem ist es möglich, dass derselbe Inode unter mehreren Dateinamen erreichbar ist.
- ☐ Es ist möglich, eine symbolische Verknüpfung (symbolic link) mit Verweis auf eine nicht existierende Datei anzulegen.
- ☐ Zum Anlegen oder Löschen von Dateien sind die Schreibzugriffsrechte auf das übergeordnete Verzeichnis irrelevant.
- ☐ Auf das Wurzelverzeichnis (root directory, "`/`") darf immer nur genau ein hardlink verweisen.
- ☐ Um den Inhalt einer Datei einlesen zu können, benötigt man Leserechte für das übergeordnete Verzeichnis.
- ☐ In einem hierarchisch organisierten Dateisystem dürfen gleiche Dateinamen in unterschiedlichen Verzeichnissen enthalten sein.
- ☐ Wird die letzte feste Verknüpfung (hard link) auf eine Datei entfernt, so wird auch die Datei selbst gelöscht.
- ☐ Wird die Datei gelöscht, auf die eine symbolische Verknüpfung (symbolic link) verweist, so wird auch die symbolische Verknüpfung selbst gelöscht.

Aufgabe 2: `hope` (60 Punkte)

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

Schreiben Sie das Programm `hope` (**H**yper-**O**ptimized **P**roxy **E**ngine), das zur Anonymisierung von HTTP-Anfragen genutzt werden kann. `hope` ist ein Proxy-Dienst, welcher zur Identitätsverschleierung des Clients dessen HTTP-Anfragen stellvertretend durchführt und die Antwort des Servers zurücksendet. Des Weiteren werden immer **genau** 8 **Anfragen** mit Hilfe eines Arbeiter-Thread-Pools parallel abgearbeitet. Angenommene Verbindungen werden über einen entsprechend synchronisierten Ringpuffer an die Threads weitergeleitet. Die Bearbeitung einer Anfrage durch einen Thread lässt sich in 3 Schritte untergliedern: 1) Anfrage empfangen 2) Kommunikation mit dem Zielserver 3) Antwort weiterleiten. Um das Beobachten des Datenverkehrs zu erschweren, sollen Schritte 2 und 3 der parallelen Anfragen jeweils gleichzeitig zueinander ausgeführt werden. Die dafür notwendige Synchronisation erfolgt mit Hilfe eines speziellen Semaphors, der seinen internen Zähler atomar um beliebige Werte inkrementieren bzw. dekrementieren lässt. Dafür erfordern die Funktionen `P()` und `V()` im Vergleich zu den aus der Übung bekannten Semaphoren einen zusätzlichen Parameter (siehe zweite Teilaufgabe).

Implementieren Sie folgende Funktionen:

`int main(void)` Initialisiert das Programm, setzt einen TCP-Socket für alle IPv6-Adressen auf Port 2025 auf und wartet auf eingehende Verbindungen. Angenommene Verbindungen werden via `bbPut` über den Ringpuffer an den Arbeiter-Thread-Pool weitergereicht. Nachdem 8 Anfragen weitergeleitet wurden, sorgt `main` für die weitere Synchronisation der Abarbeitung durch die Arbeiter-Threads. Hierzu wartet `main` auf die Beendigung eines Schritts durch alle 8 Threads und signalisiert diesen anschließend die Weiterarbeit über den Semaphor des jeweils nächsten Schritts. Erst nachdem der dritte Schritt freigegeben wurde, beginnt `main` wieder mit der Annahme von neuen Verbindungen. Verbindungsabbrüche dürfen das Programm nicht beenden.

`void *worker(void *)` Dient als Einstiegsfunktion der Arbeiter-Threads. Sie entnimmt mittels `bbGet` eine Anfrage aus dem Puffer und bearbeitet diese in drei, mit den anderen Arbeitern synchronisierten Schritten:

1. Einlesen aller Zeilen und Auswerten der ersten Zeile (`setupConnection` und `parseRequest` sind vorgegeben und nicht zu implementieren!)
2. Anfrage senden (ab Zeile 2) und Antwort abwarten (siehe `forwardRequest`)
3. Weiterleiten der Antwort an den Klienten (siehe `sendLines`)

Zwischen jedem dieser Schritte sollen sich die Threads mit Hilfe von Semaphoren synchronisieren. Hierzu signalisieren sie per entsprechendem Semaphor jeweils den Abschluss eines Schritts und warten darauf, dass der Beginn des folgenden Schritts freigegeben wird. Gehen Sie davon aus, dass auftretende Fehler bereits in den aufgerufenen Funktionen protokolliert werden. In jedem Fall muss an der Synchronisation aller Schritte teilgenommen werden.

`char **receiveLines(FILE *rx)` Liest von `rx` Zeile für Zeile ein und liefert einen `NULL`-terminierten Vektor mit Zeigern auf die gelesenen Zeilen. Im Fehlerfall soll `NULL` zurückgegeben werden und der Fehlergrund auf `stderr` ausgegeben. Sie dürfen davon ausgehen, dass ein Zeile insgesamt maximal 8192 Zeichen lang ist.

Hinweise:

- Die Funktion `parseRequest` speichert lediglich Zeiger in den Eingabepuffer
- Falls nicht anders angegeben, setzen vorgegebene Funktionen im Fehlerfall die `errno`. Erfolg wird über einen gültigen Zeiger oder den `int` Wert 0 mitgeteilt.
- Auftretende Fehler sollen in allen Fällen auf `stderr` protokolliert werden und nach Möglichkeit nicht zur Beendigung von `hope` führen. Nur wenn ein ordnungsgemäßer Weiterbetrieb ausgeschlossen werden kann, soll sich `hope` beenden!
- Nutzen Sie die Funktion `clearLines` zum Freigeben eines `NULL`-terminierten Arrays.

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

```c
#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include "jbuffer.h"
#include "sem.h"

#define THREAD_COUNT 8
#define MAX_LINE_LEN 8192

// Globale Variablen
static SEM *sig_step2, *sig_step3, *sig_main;
static BNDBUF *bb;

// Datenstrukturen
struct connection { FILE *rx, *tx; };
struct request { char *domain, *path; };

// Vorgegebene Funktionen:
// Terminiert das laufende Program mit einer errno-basierten Fehlerausgabe
static void die(const char *msg);

// Wandelt "fd" in die FILE Streams von "c" um. Schließt "fd" immer.
static int setupConnection(struct connection *c, int fd);
// Funktion zum Freigeben eines NULL-terminierten Arrays "lines"
static void clearLines(char **lines);
// Übermittelt alle Zeilen des NULL-terminierten Arrays "lines" an "fp"
static int sendLines(FILE *fp, char **lines);
// Zerlegt "line" in die für "rq" relevanten Teile
static int parseRequest(struct request *rq, char *line);
// Übermittelt die in "lines" stehenden Zeilen den in "rq" beschriebenen Server
static char **forwardRequest(const struct request *rq, char **lines);

// Erstellt einen Ringpuffer, setzt im Fehlerfall errno und gibt NULL zurück
BNDBUF *bbCreate(size_t size);
// Legt den Wert "value" in "bb"; blockiert falls dieser gerade voll ist
void bbPut(BNDBUF *bb, int value);
// Liest einen Wert aus "bb"; blockiert falls dieser gerade leer ist
int bbGet(BNDBUF *bb);

// Erstellt einen Semaphor
SEM *semCreate(int initVal);
// Blockiert bis "count" Plätze im Semaphor allokiert werden konnten
void P(SEM *sem, unsigned count);
// Gibt "count" Plätze im Semaphor frei
void V(SEM *sem, unsigned count);
```

```
int main(void) {
  // Initialisieren des globalen Zustands
      sig_main = semCreate(0);
      if (!sig_main) {
          die("semCreate");
      }
      sig_step2 = semCreate(0);

      if (sig_step2 == NULL) die("sCreate");
      sig_step3 = semCreate(0);
      if (!sig_step3) die("sCreate");
      bb = bbCreate(8);
      if (!bb) die("bbCreate");









  // Erstellen des passiven Sockets
       int sock = socket(AF_INET6, SOCK_STREAM, 0);
       if (sock == -1) die("socket");
       struct sockaddr_in6 addr = {
            .sin6_family = AF_INET6,
            .sin6_port = htons(2025),
            .sin6_addr = in6addr_any // nicht unbedingt nötig
       };
       // reuse
       if (bind(sock, &addr, sizeof(addr)) < 0) die("bind");
       if (listen(sock, SO_MAXCONN) < 0) die("listen");
```

```
  // Erstellen des Arbeiter-Thread-Pools
        pthread_t tids[THREAD_COUNT];

        for (int i = 0; i < THREAD_COUNT; ++i) {
            errno = pthread_create(&tids[i], NULL, worker, NULL);
            if (errno) die("pthread_create");
        }

  // Annehmen der Verbindungen
        int counter = 0;
        while (1) {
            int client = accept(sock, NULL, NULL);
            if (client == -1) continue;
            bbPut(bb, client);

            counter++;
            if (counter == 8) {
                V(sig_step2, 8);
                P(sig_main, 8);

                V(sig_step3, 8);
                P(sig_main, 8);
            }

        }

} // Ende von main
```

M:

```
// Funktion für den Arbeiterfaden
    void *worker(void* arg) {
        while (1) {
            P(sig_step2, 1);
            char req_line[MAX_LINE_LEN + 2];
            int client = bbGet(bb);
            struct connection conn;

            setupConnection(&conn, client);
            if (!fgets(req_line, sizeof(req_line) conn.rx)) continue;

            struct request req;
            parseRequest(&req, req_line);

            char **lines = receiveLines(conn.rx);
            fclose(conn.rx);

            if (!lines) {
                V(sig_main, 1);
                P(sig_step3, 1);
                V(sig_main, 1);
                continue;
            }
            char **res = forwardRequest(&rq, lines);
            V(sig_main, 1);


            P(sig_step3, 1);
            if (res != NULL) {
                sendLines(conn.tx, res);
                clearLines(res);
            }
            fclose(conn.tx);
            V(sig_main, 1);
        }
    }
```

W:

```
// Funktion zum Einlesen aller Zeilen eines FILE-Streams
char ** receiveLines(FILE* rx) {
    char **ret = NULL;

    for (int i = 0;; i++) {
        ret = realloc(ret, (i + 1) * sizeof(char*));
        char* cur = malloc(MAX_LINE_LEN + 1);
        if (!fgets(cur, MAX_LINE_LEN + 1, rx)) {
            // Fehlerbehandlung ferror
            if (ferror(rx)) {
                perror("fgets");
                free(cur);
                free(ret);
                return NULL;
            }

            ret[i] = NULL;
            break;
        }
        ret[i] = cur;
    }

    return ret;

}
```

Z:

Semaphor (10 Punkte)

In dieser Teilaufgabe sollen Sie die erweiterten Operationen `P()` und `V()` eines Semaphors implementieren, so wie diese von `hope` benötigt werden. Abweichend von einem normalen Semaphor, bei dem der Wert des Semaphors jeweils immer um 1 erniedrigt bzw. erhöht wird, nehmen `P()` und `V()` in dieser Variante neben dem Zeiger auf den Semaphor noch einen weiteren Parameter `count` entgegen. Dieser gibt an, um welchen Wert der Semaphor auf einmal erhöht bzw. erniedrigt wird. Hierbei ist wichtig, dass der Semaphor nur erniedrigt werden darf, wenn er dabei nicht kleiner `0` wird, sonst blockiert die Operation (übertragen auf eine Puffer-Synchronisation bedeutet dies, dass man entweder alle angeforderten Pufferplätze auf einmal bekommt oder gar keinen nimmt).

```
#include "sem.h"
#include <pthread.h>

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

void P(SEM *sem, unsigned count) {

}

void V(SEM *sem, unsigned count) {

}
```

S:

Aufgabe 3: Ausnahmen (12 Punkte)

1) Sie haben in der Vorlesung zwei Arten von Ausnahmen von der normalen Programmausführung kennengelernt. Wie lauten diese verschiedenen Arten von Ausnahmen? Nennen sie zwei Eigenschaften, durch die sie sich voneinander unterscheiden. (3 Punkte)

2) Nennen Sie die zwei Modelle der Ausnahmebehandlung. Wie unterscheiden sich diese Modelle? Nennen Sie für jedes der Modelle je einen Auslösungsgrund. (5 Punkte)

3) Untenstehender Code wird auf einem x86_64-Linux-System ausgeführt. (4 Punkte)

```
char *ptr = NULL;
*ptr = 'c';
```

a) Wodurch tritt hier ein Fehler auf? Was wird der Anwendung signalisiert? (2 Punkte)

b) Welche Hardwarekomponente entdeckt diesen Fehler? Wie signalisiert diese Komponente den Fehler an das Betriebssystem? (2 Punkte)

Aufgabe 4: Speicherverwaltung (12 Punkte)

1) 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)

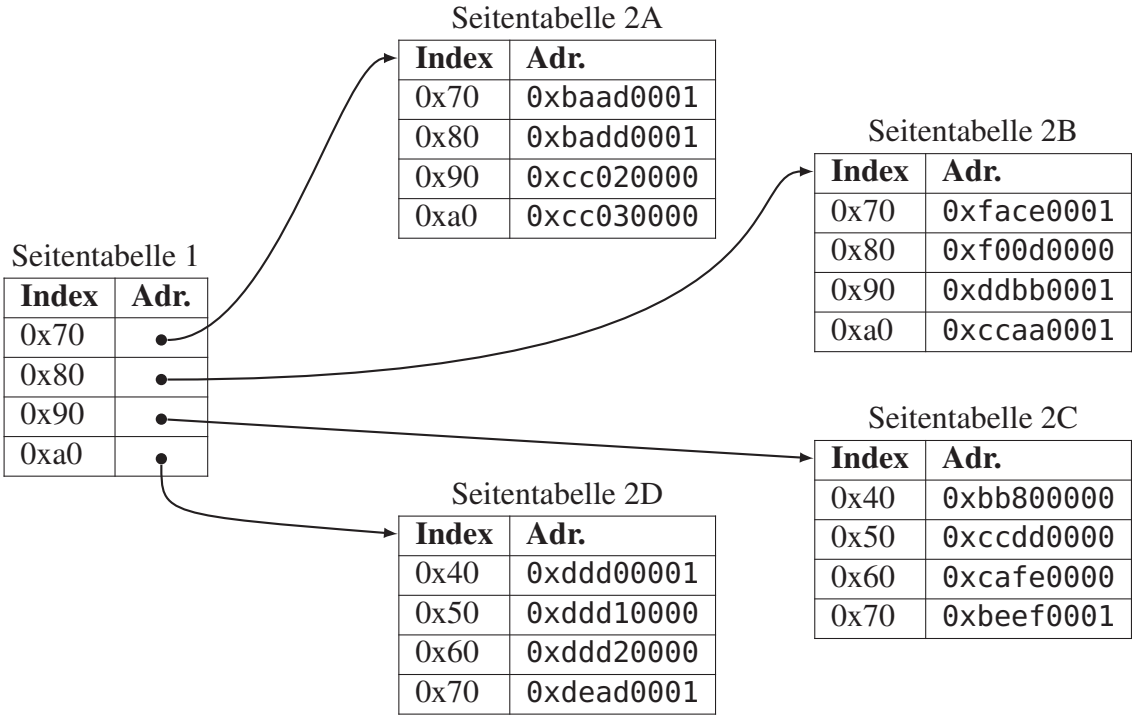
2) Nicht-optimale Ersetzungsstrategien nutzen eine gewisse Programmeigenschaft aus, um sinnvolle Ergebnisse zu erzielen. Nennen Sie diese Eigenschaft und erklären Sie kurz, warum sie auf die meisten Programme zutrifft. (2 Punkte)

3) 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)

4) 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? (6 Punkte)

Aufgabe 5: Paging (6 Punkte)

Gegeben sei unten dargestellte Hierarchie zweistufiger Seitentabellen. Die Adresslänge des genutzten Systems sei $32\,\mathrm{Bit}$, die Größe einer Seite $64\,\mathrm{Kibibyte}$ (d.h., 2^{16} Byte). Für die Indizierung der zweistufigen Abbildung werden pro Stufe $8\,\mathrm{Bit}$ genutzt. Das niederwertigste Bit eines Seitentabelleneintrags fungiert als Anwesenheitsbit: Ist es auf 1 gesetzt, ist die Seite anwesend.



1) Bestimmen Sie die **reale Adresse** zur logischen Adresse `0x9070babe`. Geben Sie hierbei die zur Bestimmung notwendigen Zwischenschritte stichpunktartig an! (4 Punkte)

2) Nehmen Sie an, dass alle gültigen Seitentabelleneinträge oben abgebildet sind. Was würde in diesem Fall mit einem Prozess passieren, der schreibend auf die Adresse `0xddd10000` zugreift und wieso? (2 Punkte)