

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:

`#define SUB(a, b) a - b
#define MUL(a, b) a * b`

Was ist das Ergebnis des folgenden Ausdrucks? $4 * \text{MUL}(\text{SUB}(3, 5), 2)$

- 2
- 2
- 16
- 16

$$\begin{array}{rcl} 4 & \times & 3 - 5 \times 2 \\ & = & 12 - 10 = 2 \end{array}$$

2 Punkte

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

- 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)` kopiert eine Programmdatei in einen neu erzeugten Prozess.

2 Punkte

c) Was passiert, wenn Sie in einem C-Programm über einen ungültigen Zeiger versuchen auf Speicher zuzugreifen?

- 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.

Compiler knows Nothing

2 Punkte

d) Was versteht man unter Virtuellem Speicher?

- 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 Daten speichern lassen, weil er physikalisch nicht vorhanden ist.

e) Welche Problematik wird auf das Philosophenproblem abgebildet?

alex

- Ein Erzeuger und ein Verbraucher greifen gleichzeitig auf gemeinsame Datenstrukturen zu. mehr Verbraucher! kein Erzeuger falsch
- Exklusive Bearbeitung durch mehrere Bearbeitungsstationen.
- Mehrere Prozesse greifen lesend und schreibend auf gemeinsame Datenstrukturen zu. kein Schreiben
- Gleichzeitiges Belegen mehrerer Betriebsmittel (Hintergrund的分子, 还需要多人的分子)

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

geo_12b

- Bei federgewichtigen Prozessen ist die Schedulingstrategie durch das Betriebssystem vorgegeben. 联邦 User-Threads
- Bei Kern-Threads ist die Schedulingstrategie meist durch das Betriebssystem vorgegeben. Scheduler Process, 不是 threads
- Die Umschaltung von Threads muss immer im Systemkern erfolgen (privilegierter Maschinenbefehl). User-Threads user-level threads (feather-weight processes) 不被 Systemkern 支持
- Kern-Threads blockieren sich bei blockierenden Systemaufrufen gegenseitig.

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

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

2 Punkte

Projekt mit V6

(c) MMU (memory management unit) V6 - Memory Management

↑ hardware support

← Lösungen zu den Programmieraufgaben

Anzeige in geschachterter Form



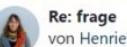
frage von Shipan Liu - Freitag, 3. Februar 2023, 16:12

ich weiss nicht welche Option ist richtig, kann jemand mir vielleicht erklären?
ich denke, 4. ist falsch,
und 1. ist richtig, aber weiss nicht warum 2. und 3. sind falsch.

c) Was passiert, wenn Sie in einem C-Programm über einen ungültigen Zeiger suchen auf Speicher zuzugreifen?

- Beim Laden des Programms wird die ungültige Adresse erkannt und der Adresszugriff 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 setzt einen Trap aus.
- Das Betriebssystem erkennt die ungültige Adresse bei der Weitergabe des Adressfehlers an die CPU (partielle Interpretation) und leitet eine Ausnahmebehandlung ein.
- Der Compiler erkennt die problematische Code-Stelle und generiert Code zur Laufzeit bei dem Zugriff einen entsprechenden Fehler auslöst.

Dauerlink Antworten



Re: frage von Henriette Paulina Hofmeier - Freitag, 3. Februar 2023, 17:07

Hallo Shipan,

bei der Frage hast du richtig erkannt, dass 4. falsch ist -- der Compiler hat an der Stelle keine Information über die Gültigkeiten von den in Zeigern gespeicherten Adressen.

Genauso falsch ist auch Antwort 3, denn die Adresse, also der Inhalt des Pointers, wird zuerst an die MMU übergeben, die zunächst die Adresse in eine physikalische Adresse übersetzt. Das ist dann auch die Stelle an der der Zugriff auf einen ungültigen Zeiger auffällt. Sprich, Antwort Nr. 2 ist richtig.

Antwort 1 ist falsch. Ein Segmentation Fault wird durch Betriebssystem zugestellt, nachdem ein Fehler durch die MMU signalisiert wurde. Das passiert bei Zugriff auf den Zeiger, im Rahmen der Adressübersetzung und damit zur Laufzeit und nicht beim Laden des Programms.

MMU Fehler
↓
↓

Ich hoffe, die richtige Antwort ist so klar geworden.

Viele Grüße
Henriette

Dauerlink Ursprungsbeitrag Antworten

Frage

tfolio OpenRUB Support Deutsch (de)

Shipan Liu

Anzeige in geschachterter Form



Frage von Shipan Liu - Freitag, 3. Februar 2023, 16:46

hallo,

in diesem Bild habe ich keine Ahnung, welche ist eigentlich richtig. "Threads" in 2. Option bedeutet "kern-Threads" oder? ich glaube die 1. ist falsch, da Scheduling ist eigentlich nur für Process?

- Bei Kern-Threads ist die Schedulingstrategie meist durch das Betriebssystem vorgegeben.
- Die Umschaltung von Threads muss immer im Systemkern erfolgen (unterstützt durch spezielle Maschinenbefehle).

Dauerlink Antworten



Re: Frage

von Henriette Paulina Hofmeier - Freitag, 3. Februar 2023, 17:23

Hallo Shipan,

an der Stelle ist die erste Aussage, also, dass bei Kernel-Threads die Schedulingstrategie meist durch das Betriebssystem vorgegeben ist, richtig.

Wirf dazu am besten nochmal einen Blick in die Übungsfolien mit der Gegenüberstellung der verschiedenen Threadarten (Übungsfoliensatz 5).

Kernel-Threads werden durch den Scheduler des Betriebssystems koordiniert.

Es gibt aber auch Threads, z.B. User-Threads, bei deren Scheduling das Betriebssystem gar nicht involviert ist. Daher ist die zweite Aussage falsch.

An der Stelle noch eine Anmerkung -- wir versuchen solche Unklarheiten, ob bei Threads jetzt einfach nur die Vorsilbe fehlt oder das so Absicht ist, bei den Klausuren zu vermeiden. Also hier würden wir stattdessen eine Formulierung im Sinne von „Die Umschaltung bei allen Arten von Threads muss immer im Systemkern erfolgen“ oder so ähnlich.

Die Frage kommt noch aus der Probeklausuren, die gerade bei solchen Formulierungen und der ausschließlichen Verwendung von deutschen Begriffen nicht ganz repräsentativ ist. 😊

Viele Grüße
Henriette

Dauerlink Ursprungsbeitrag Antworten

← frage

CLOCK Aufgabe Klausur SOSE 2022

← Ankündigungen

Direkt zu:

Fragen zu A1 (filo)

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

- 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.
processes with long execution time are prone to starvation
- Ein hochpriorer Prozesse muss eventuell auf ein Betriebsmittel warten, das von einem niedrigprioren Prozess exklusiv benutzt wird. Der niedrigpriorere Prozess kann das Betriebsmittel jedoch wegen eines mittelprioriern Prozesses nicht freigeben (Prioritätenumkehr).
快想把资源给大高,但是被中将抢了
- Das Phänomen der Prioritätsumkehr hungert niedrig-priore Prozesse aus.

2 Punkte

i) Ein Programm will die drei Zeichenketten

```
char a[] = "dire"; 4
char b[] = "cto"; 3
char c[] = "ry"; 2
```

地址乱序→buffer溢出

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
中间的3个'/' 不要忽略!
- 10
- 11
- 13

2 Punkte

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

- Der Prozess wartet auf eine Tastatureingabe.
(warte auf Betriebsmittel)
- Ein anderer Prozess blockiert sich an einem Semaphor.
- Der Prozess hat einen Seitenfehler für eine Seite, die aber noch im Hauptspeicher vorhanden ist.
(same as :)
- Der Prozess hat auf Daten von der Festplatte gewartet und die Daten stehen nun zur Verfügung.
prozess für Festplatte ist fertig.

2

Hallo Shipan,

zu j)

Hier ist die 4. Antwort richtig, da es sich um den Zustand "bereit" handelt. Bereit bedeutet, dass der Prozess direkt wieder die Ausführung aufnehmen kann sobald er CPU-Zeit durch den Scheduler zugewiesen bekommt. Im Falle der 4. Aussage war der Prozess blockiert, da er auf Daten von der Festplatte gewartet hat - in der Zeit konnte er seine Ausführung nicht fortsetzen. Da die Daten jetzt aber bereit stehen ist auch der Prozess "bereit" weiter zu laufen.

Die 2. Aussage ist falsch, da wie auch schon durch den Satz angedeutet, der Prozess blockiert ist. Also der Prozess hat an einer Semaphore P() aufgerufen und der kritische Abschnitt ist aber schon belegt und der Prozess muss warten, bis ein anderer die Semaphore frei gibt (V()). In der Zeit könnte der Prozess auch bei Zuteilung von CPU Zeit ja nicht mehr weiter laufen. Daher wird er in den Zustand "blockiert" überführt und ist erst wieder "bereit", sobald ein anderer Prozess V() an der Semaphore aufgerufen hat.

k) Gegeben sei folgendes Szenario: zwei Fäden werden auf einem Monoprozessor-System 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:

- 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.

2 Punkte

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:

```

1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #define PI 3.1415
5 //全局变量
6 extern int x;
7
8 int main(int argc, char *argv[]) {
9     static int a; → BSS区
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 chars 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. → include file copy past the content
- 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.

Zeile b已把参数除了, 变成了3
Zeile 14输出 3.1415



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

- 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. (由TLB flush, TLB中内容会过期)

包裝
- 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. (逻辑地址, 因为用MMU来转)

逻辑地址, 因为用MMU来转
- 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.

zum TLB:

Poste am besten mit, aus welcher Klausur du die Frage nimmst und die gesamten Antwortmöglichkeiten. Das ist in dem Falle eine Multi-Choice Aufgabe und es kann mehr als nur eine Antwort korrekt sein.

Die beiden Aussagen, die du dir rausgesucht hast sind hier auch beide richtig.

Zum Thema voll-assoziativer Cache - das Thema Caching und verschiedene Cache Assoziativitäten sollte bereits aus anderen Vorlesungen bekannt sein. Wenn nicht, lies am besten nochmal zu den Assoziativitäten nach. Ganz grundsätzlich gibt die Assoziativität an, in welchem Cacheblock bzw. in welcher Cachezeile ein Speichereintrag im Cache abgelegt wird. Bei einem voll-assoziativem Cache, gibt es keine spezielle Zuordnung und ich nenne es mal flappig eine "freie Platzwahl". Also jeder Speichereintrag kann prinzipiell in jeder Cachezeile abgelegt werden.

从3!
从逻辑地址
到物理地址
nach physikalische Adresse

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):

```
./party 2 "sleep 5" "ls -ash /" "ps aux" "tar -xvf file.tar"
          |         |         |         |
          |         |         |         ↓ 4. Befehl
          |         |         |         ↓ 3. Befehl
          |         |         |         ↓ 2. Befehl
          |         |         |         ↓ 1. Befehl
          |         |         |
          |         ↓         ↓         ↓         ↓
          ↓         ↓         ↓         ↓
          → maximal 2 Befehle werden gleichzeitig ausgeführt
```

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 Prozesse 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).

```
#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
```

A simple rectangular outline with a double border, designed for children to draw or write in.

// Befehlszeile parsen

A blank rectangular box with a thin black border, intended for a child to draw or write in.

```
// Ende Funktion main
```

- M:

```
// Funktion run
```

1

```
// Befehlszeile parsen
```

A small, empty rectangular box with a thin black border, likely intended for a student to draw or write something.

// 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)

Es ist ein Zustand der Verzögerung, bei dem ein Prozess auf einen Ressourcenzugriff warten muss, während andere Prozesse die Ressource nutzen.

- 2) Erläutern Sie das Konzept Mutex. Welche Operationen sind auf Mutexen definiert und was tun diese Operationen? (5 Punkte)

Mutex is a program object that is created so that multiple program threads can take turns sharing the same resource, such as access to a file.

Mutex ensures that only one thread has access to a critical section or data by using operations like "lock" and "unlock".



- 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 *); *mutex, nicht Semaphore*
- 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){
```

m = mutCreate()

```
for(int i = 0; i < 3; ++i) {
    startWorkerThread(threadFunc);
}
```

```
while(1) {
```

```
    int x = calcValue();
```

(lock(m))

```
    accu += x;
```

unlock(x)

```
}
```

```
}
```

Arbeiterthread:

```
void threadFunc(void) {
```

```
/
```

```
int x = calcValue();
```

(lock(m))

```
accu += x;
```

unlock(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.

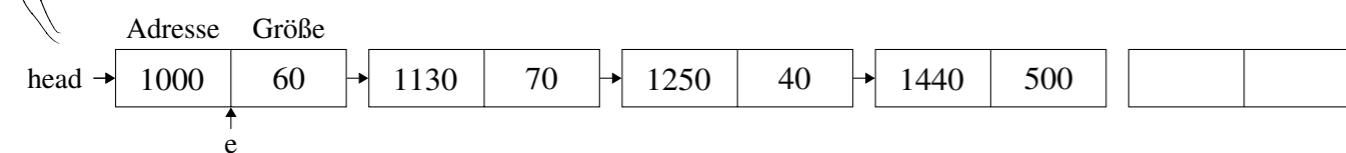
1) Vervollständigen Sie den Zustand der Verwaltungsdatenstrukturen für die untenstehende Folge von `malloc(3)`- und `free(3)`-Aufrufen. Im Rahmen dieser Aufgabe soll dafür das **next-fit**-Verfahren mit Verschmelzung von freien Blöcken angewandt werden. Zeichnen Sie pro Schritt den Einsprungpunkt **e**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

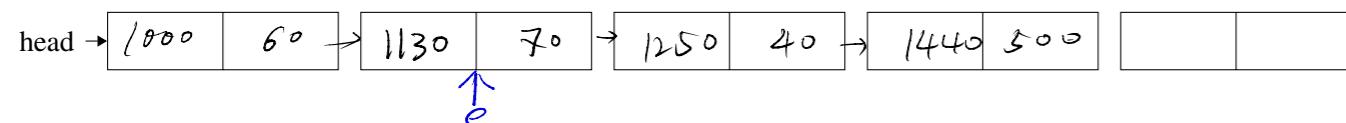
Vermerken Sie zudem, welche Adresse der jeweilige malloc(3)-Aufruf zurückliefert. In dem Fall, dass ein Speicherblock aufgeteilt wird, soll der hintere Teil (entspricht dem Speicherbereich mit der höheren Adresse) an den Aufrufer Aufrufer zurückgegeben werden.

① kennzeichnet den initialen Zustand der Speicherverwaltung nach der Allokation p_0 . (9 Punkte)
不是 进攻的空间，而是 1 个 追起来的计时器

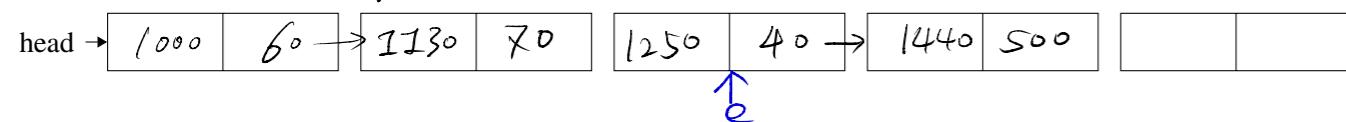
\① p₀ = malloc(50); // = 1060 ← Rückgabewert des Aufrufs von malloc



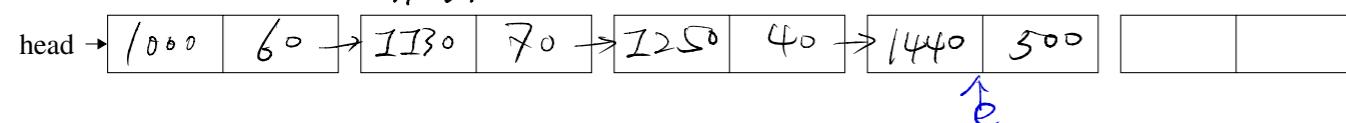
① p₁ = malloc(70); // = 1200



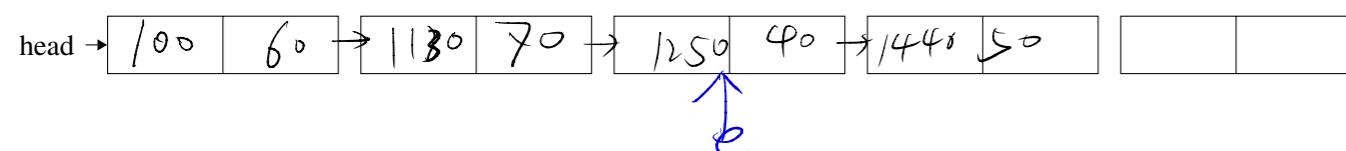
② p₂ = malloc(20); // = 1290



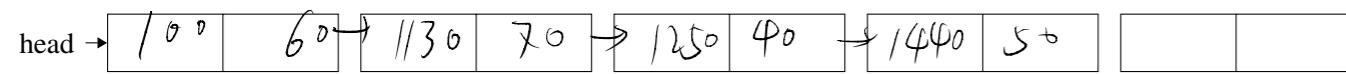
③ `p3 = malloc(500); // 1940`



④ free(p₁);



⑤ free(p_0);



First Fit-# avoid many small holes at the beginning of the list (first matching hole is used)

① like First Fit, But start at the last assigned hole

↓
不从头开始，从
已-分配过
空闲块开始

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

Vorteil: start at the last assigned hole
and don't have to start at the beginning

- Nachteile: can not choose the most suitable hole (too large)

- 3) Erläutern Sie den Unterschied zwischen interner und externer Fragmentierung. (2 Punkte) 15

Agar

- 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)

✓春季 在SS2上