Technische Universität Berlin Fachgebiet Komplexe und Verteilte IT-Systeme Sommersemester 2020	**Aufgabenblatt 1** zu – Systemprogrammierung – Prof. Dr. Odej Kao
Abgabetermin:	**17.05.2020 23:55 Uhr**

Aufgabe 1.1: Von-Neumann-Architektur (Tafelübung)

Die am weitesten verbreitete Rechnerarchitektur wurde nach John von Neumann benannt.

a) Beschreiben Sie die Funktionen der vier grundlegenden Komponenten der Von-Neumann-Architektur:

- CPU
- Speicher
- Ein-/Ausgabegeräte
- Gemeinsamer Bus

b) Wie wird in einem Rechner auf Basis dieser Architektur ein Programm prinzipiell abgearbeitet (Takte)?

c) Nennen Sie zwei Nachteile gegenüber einer parallelen Architektur wie der Harvard-Architektur. Gehen Sie dabei auch auf den *Von-Neumann-Flaschenhals* ein.

Aufgabe 1.2: Verkettete Listen (Tafelübung)

Diese Aufgabe wiederholt die Datenstruktur verkettete Liste. Sie gilt als direkte Vorbereitung auf die Praxisaufgabe. Diese Aufgabe behandelt die FIFO-Liste.

a) Was ist eine einfach verkettete Liste? Was bedeutet FIFO?

b) Was machen die Funktionen *enqueue()*, *dequeue()* und *front()*? Wie verändert sich die Liste?

c) Machen Sie eine Handsimulation der folgenden Aufrufe. Wie sieht die Liste nach jeder Funktion aus?

- enqueue(B)
- enqueue(A)
- enqueue(C)
- dequeue()
- enqueue(D)
- dequeue()
- dequeue()
- front()

Aufgabe 1.3: Prozessormodi (2 Punkt) (Theorie)

a) Welche Prozessormodi existieren? Nennen Sie zwei privilegierte Modi und einen nicht privilegierten Modus. (0.6 P)

b) Worin unterscheiden sich nicht privilegierte und privilegierte Modi (Rechte und Anwendung)? Nennen Sie 3 Unterschiede. (0.6 P)

c) Wie wird zwischen privilegierten und nicht privilegierten Modus gewechselt? Nennen Sie drei Beispiele zum Wechseln in einen privilegierten Modus und ein Beispiel zum Wechseln in den unprivilegierten Modus. (0.8 P)

Aufgabe 1.4: Interrupts (2 Punkt) (Theorie)

a) Was ist ein Hardware-Interrupt? Wie reagiert das Betriebssystem und der Prozessor auf einen Hardware-Interrupt? (0.4 Punkte)

b) Wie erkennt der Prozessor, dass ein Hardware-Interrupt vorliegt? (0.4 Punkte)

c) Nennen Sie zwei Beispiele, wie Hardware-Interrupts entstehen können. (0.4 Punkte)

d) Wie unterscheiden sich sequentielle Unterbrechungsbehandlung und geschachtelte Unterbrechungsbehandlung? Was passiert, wenn ein Interrupt eintritt, während noch ein vorheriger Interrupt behandelt wird? (0.8 Punkte)

Aufgabe 1.5: Priority Queue (5 Punkte) (Praxis)

In dieser Aufgabe soll eine Priority Queue mit Hilfe einer verketteten Liste implementiert werden. Hierzu soll die Datenstruktur und deren Elemente als structs abgebildet werden und die gängigen Funktionen *prio_q_enqueue()*, *prio_q_front()* und *prio_q_dequeue()* implementiert werden. Des weiteren sollen Funktionen zum Erstellen einer neuen Priority Queue *prio_q_create()* und zum Löschen einer Priority Queue *prio_q_destroy()* implementiert werden. Optional kann die Funktion *prio_q_print* zu Debugging-Zwecken implementiert werden. Nähere Informationen zu diesen Funktionen finden Sie in der vorgegebenen *prio_q.h*.

Die Struktur der Queue und Elemente ist vorgegeben. Die Queue speichert stets das erste (*front*) Element und zählt in der Variable *size* die aktuelle Anzahl an Elementen mit. Das Erstellen der Queue-Elemente soll beim Aufruf der Funktion *prio_q_enqueue()* geschehen. Die Elemente sollen untereinander einfach verkettet werden und einen Prioritätswert als Integer speichern. In dieser Version bedeutet eine größere Prioritätszahl eine höhere Priorität: Ein Element mit der Priorität 3 hat also Vorrang gegenüber einem Element mit der Priorität 1. Bei gleicher Priorität soll das FIFO-Prinzip (*First In First Out*) gelten, sodass also ein neues Element hinter die Elemente mit gleicher Priorität eingefügt wird. Die Speicherverwaltung der Elemente und Queue erfolgt dynamisch über *malloc()* bzw. *free()*. Die Elemente speichern generische Daten in Form eines *void*-Pointers. Es ist somit dem Anwender überlassen, welche Art von Nutzdaten die einzelnen Elemente enthalten. Um die Referenzen auf die Nutzdaten beim Löschen der Queue nicht zu verlieren, erhält die Funktion *prio_q_destroy()* einen zusätzlichen Parameter, welcher ein Array repräsentiert, in das die Daten-Pointer der einzelnen Elemente hineingeschrieben werden sollen. Der Aufrufer kann somit den (eventuell) für die Daten dynamisch allozierten Speicher wieder freigeben (siehe *main*-Funktion).

Die Implementierung erfolgt ausschließlich in der Datei *prio_q.c*, hier können zudem nach Bedarf weitere Hilfsfunktionen implementiert werden. In der Datei *main.c* können weitere Testfälle hinzugefügt werden, die Header-Datei *prio_q.h* soll jedoch nicht geändert werden.

Die folgenden Fragen können Sie **zur Orientierung** nutzen (eine schriftliche Beantwortung dieser Fragen ist nicht notwendig).

- Was ist ein struct in C und wie wird es genutzt?
- Wie funktioniert die dynamische Speicherverwaltung über malloc() und free()?
- Was sind Pointer und wie werden diese genutzt?
- Was ist eine einfach verkettete Liste und wie kann diese in C implementiert werden?

Hinweise:

- Zur Abgabe gehören eure bearbeitete *prio_q.c* sowie alle weiteren Dateien der Vorgabe.
- **Vorgaben:** Bitte halten Sie sich bei der Programmierung immer an die Code-Vorgaben und Abgaberichtlinien, die Sie auf ISIS finden. Eine Missachtung kann zu Punktabzug führen.
- **Makefile:** Bitte verwenden Sie für diese Aufgabe das Makefile aus der Vorgabe.
- **Dynamischer Speicher:** Um zu evaluieren ob der gesamte von der Queue allokierte Speicher wieder freigegeben wurde, empfiehlt sich das Kommandozeilen Werkzeug **valgrind**[1] (unter linux über apt-get install valgrind). **Memory leaks führen zu Punktabzug.**
- Bitte testen Sie ihre Priority Queue sorgfältig! Das erfolgreiche ausführen der vorgegebenen main Funktion bedeutet nicht, dass Euer Priority Queue fehlerfrei ist! Sie dürfen die Main Funktion frei bearbeiten.

[1] http://valgrind.org/