Aufgabe 2

1 Lernziel

- Vertiefung Assemblerprogrammierung
- Prozesswechsel verstehen
- Konzept der aktiven Objekte kennenlernen

2 Aufgabenbeschreibung

In dieser Aufgabe soll eine einfache Prozessverwaltung für CoStuBS entstehen. Zunächst werden koorperative Prozesse realisiert, welche die CPU freiwillig abgeben.

Ein Prozess ist durch seinen aktuellen Ausführungszustand gekennzeichnet. Dies beinhaltet u.a. Daten, die er verwendet, sowie die gerade ausgeführte Instruktion. Bei einem Prozesswechsel müssen diese Informationen gesichert werden, damit der Prozess später weiterarbeiten kann. Um welche Informationen es sich dabei handelt, hängt von der verwendeten Hardware ab. Verwaltet werden die Daten von Coroutinen.

Die Prozessverwaltung umfasst mehr als das Speichern und Wiederherstellen von Ausführungszuständen. Welcher Prozess gerade aktiv ist, wird im Dispatcher vermerkt. Wird ein Prozess angehalten, so muss der nächste lauffähige ermittelt werden. Dafür besitzt der Scheduler eine Liste von lauffähigen Prozessen. Da es in CoStuBS keine dynamische Speicherverwaltung gibt, ist es nicht so einfach, Listen zu erstellen. Deshalb gibt es eine Datenstruktur, die ein Listenelement repräsentiert. Jede Klasse, deren Instanzen in einer Liste verwaltet werden, muss die Klasse, die das Listenelement repräsentiert, erweitern.

Aktivitäten kombinieren Coroutinen und Listenelemente, die vom Scheduler verwaltet werden. Zusätzlich besitzen sie einen Ausführungszustand und stellen weitere häufig benutzte Methoden zur Verfügung.

3 Theoretische Aufgaben

3.1 Konstruktoren und Destruktoren

Betrachtet den nachfolgenden (vereinfachten) Quelltext. Angenommen, mit Hilfe von out.print können Zeichenketten auf dem Bildschirm ausgegeben werden. In welcher Reihenfolge erscheinen welche Ausgaben?

```
1  int main() {
2      out.print("main1;");
3      Hello anton("Anton");
4      Hello berta("Berta");
5      out.print("main2;");
6      anton.body();
7      out.print("main3;");
8  }
```

```
1  class Hello {
2      Hello(const char* name) : name(name) {
3          out.print(name);
4          out.print(" ctor;");
5      }
6
7      ~Hello() {
8          out.print(name);
9          out.print(" dtor;");
10     }
11
12     void body() {
13         out.print(name);
14         out.print(" body;");
15     }
16
17     const char* name;
18 };
```

Hinweis:

Es gibt mindestens eine Ausgabe, die die Zeichenkette dtor enthält.

3.2 Gültigkeitsbereiche von Variablen

Betrachtet den nachfolgenden (vereinfachten) Quelltext. Angenommen, mit Hilfe von out.print können Zeichenketten auf dem Bildschirm ausgegeben werden. Wie könnte die folgende Ausgabe zu erklären sein?

```
Anton ctor; Anton dtor; Berta ctor; Berta body; Berta dtor;
```

```
1  Hello* front;
2
3  void createHello() {
4      Hello anton("Anton");
5      front = &anton;
6  }
7
8  int main() {
9      createHello();
10     Hello berta("Berta");
11     front->body();
12 }
```

```
1  class Hello {
2      Hello(const char* name) : name(name) {
3          out.print(name);
4          out.print(" ctor;");
5      }
6
7      ~Hello() {
8          out.print(name);
9          out.print(" dtor;");
10     }
11
12     void body() {
13         out.print(name);
14         out.print(" body;");
15     }
16
17     const char* name;
18 };
```

3.3 Kontrollfragen

Versucht, die folgenden Fragen zu beantworten **bevor** ihr mit der Implementierung beginnt.

- Was ist ein Prozess im Sinne der Informatik? Durch welche Teile definiert er sich?
- Warum ist die Abstraktion eines Prozesses sinnvoll? Betrachte hierbei den Aspekt der Monopolisierung einer CPU!
- Was versteht man konkret unter einem Prozesswechsel (welche notwendigen Schritte müssen unternommen werden)?
- Wie sieht ein Prozesswechsel dementsprechend auf der x86 Hardware aus?
- Wie sieht der Stack beim ersten Wechsel zu einem Prozesses aus?
- Was ist ein Prozesskontrollblock und was beschreibt er?
- In welchem Kontext macht ein Prozesskontrollblock Sinn?
- Welche Arten von Prozessverwaltung gibt es?
- Nenne und erkläre grundlegende Algorithmen der Prozessverwaltung!

- Was ist eine Ready-Liste und wozu dient sie?
- Müssen Prozesse beendet werden? Wenn ja, wann und wie? Wenn nein, warum nicht? Wie lange existieren Prozesse dann?
- Was ist eine Coroutine?

4 Praktische Aufgaben

4.1 Einarbeitung in die Vorgabe

Wie in jeder Vorgabe sind neue Dateien hinzugekommen oder bereits bestehende wurde verändert. Schaut euch die vorgegebenen Dateien an, damit ihr deren Inhalt versteht. Einzige Ausnahme sind die Dateien im Ordner machine/boot - diese müsst ihr nicht verstehen.
Zur Unterstützung werden manchmal UML-Diagramme zur Verfügung gestellt. Bitte beachtet, dass diese nicht immer vollständig sind.

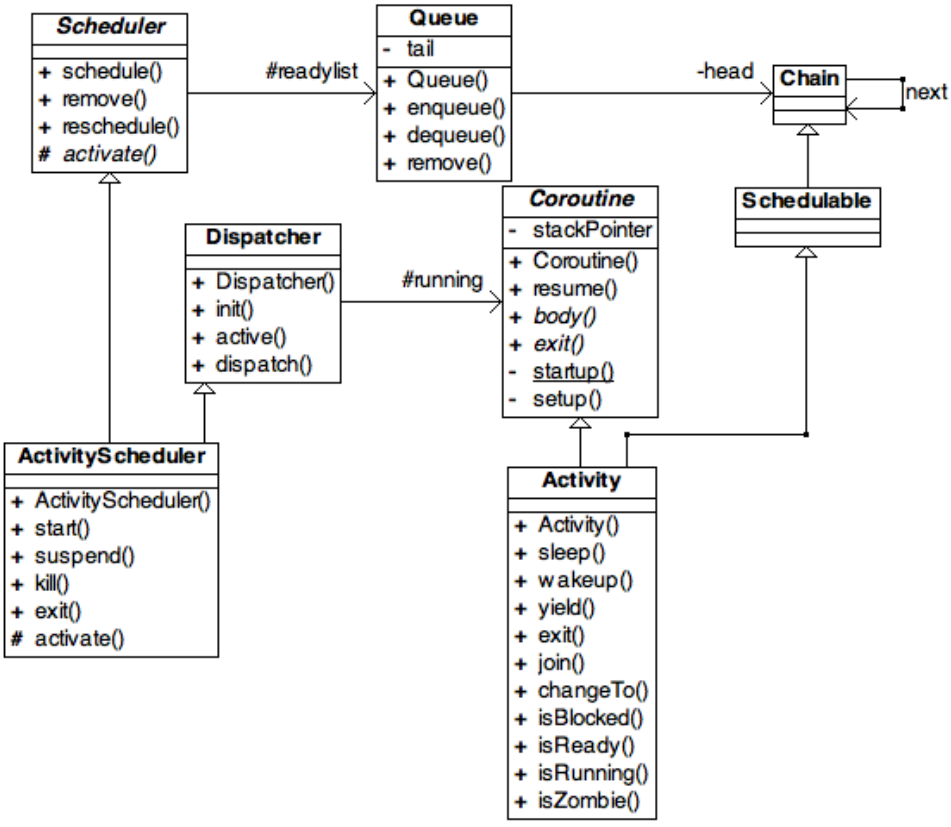

Abbildung 1: Klassendiagramm

Gegeben ist das Klassendiagramm in Abbildung 1. Außerdem stellen wir eine Listenimplementierung (Queue) zur Verfügung, welche hilfreich bei der Implementierung des Schedulers ist. Kopiert die Dateien in eurer Projekt an die vorgesehen Stellen und nehmt die erforderlichen Modifikationen am Makefile vor.

4.2 Coroutine

Hier soll die Klasse Coroutine.h implementiert und in einer kleinen Anwendung getestet werden. Dazu ist u.A. eine Assemblerroutine switchContext zu implementieren, welche den Prozesswechsel durchführt. Ferner muss hier ein initialer Stackaufbau für eine Coroutine vorgenommen werden. Beide Punkte werden intensiv in den Vorlesungen bzw. Übungen erläutert. Testet Eure Coroutinen mit dem vorgegebenen Programm mainCo.cc und skizziert euch was eigentlich während des Programmablaufs passiert.

4.3 Activity

Hier soll die Klasse Activity.h implementiert werden. Diese Klasse stellt de facto einen Prozesskontrollblock für unsere Prozessverwaltung dar. Besondere Aufmerksamkeit verlangt die Verwaltung des Ausführungszustandes dahingehend, dass eine Aktivität eventuell auf die Beendigung von sich selbst oder eines anderen Objektes warten muss, bevor es zerstört werden darf. Dazu dienen die Methoden

```
void Activity :: join ()
virtual ~Activity ()
```

Durch den Aufruf der Methode join wird der gerade laufende Prozess suspendiert. Seine Ausführung wird erst dann fortgesetzt, wenn der Prozess, dessen join-Methode aufgerufen wurde, beendet wird.

Beispiel: Angenommen Anton ist der gerade aktive Prozess. Durch Ausführung von berta.join() wird Anton suspendiert und fährt erst dann fort, wenn Berta beendet wird.

Alle abgeleiteten Klassen sollten einen Destruktor definieren, der seinerseits entweder join() oder kill() aufruft, um sicherzustellen, dass der betreffende Prozess nicht mehr läuft! Für den Fall, dass ein Anwender vergisst join() oder kill() im Destruktor einer abgeleiteten Klasse aufzurufen, soll der Activity Destruktor auf jeden Fall ein kill() ausführen. Damit ist sichergestellt, dass keine 'Prozessleichen' auf der Ready-Liste zu finden sind.

Scheduler

Als nächstes kommt der Scheduler.h. Er verwaltet alle lauffähigen Objekte auf einer Liste. In der ersten Aufgabe ist ein simpler FIFO-Scheduler zu implementieren. Er wird über die Methoden schedule, remove und reschedule benutzt. Seinerseits benutzt er die abstrakte Methode activate, um dem abgeleiteten Scheduler mitzuteilen, welches Objekt er zur Aktivierung ausgesucht hat. Er entfernt dabei das ausgesuchte Objekt aus seiner Ready-Liste.

ActivityScheduler

Zum Schluss kommt der ActivityScheduler.h. Er verwaltet die Ausführungszustände der Prozesse. Er sorgt dafür, dass ein Prozess wieder in die Ready-Liste kommt, wenn es der Ausführungszustand erlaubt. Er sorgt auch dafür, dass wenn Prozesse beendet werden, eventuelle Wartebedingungen eingehalten werden. Außerdem kann der ActivityScheduler.h Prozesse komplett aus dem Scheduling entfernen.

Testet zum Schluss Euer System zunächst mit dem in mainAct.cc vorgegebenen Programm und testet zusätzlich mit einem eigenen Programm die Funktionalität von sleep, wakeup und das Verhalten beim Beenden von Prozessen. Dies ist natürlich bei der Abgabe zu demonstrieren.

4.4 Vorbereitung der Abnahme

Testet eure Implementierung ausführlich und vervollständigt Kommentare. Bereitet euch auf die Abnahme vor. Informationen zu deren Ablauf findet ihr auf der Lehrstuhl-Webseite.

Viel Erfolg!