Aufgabe 3

1 Lernziel

- Behandlung von Interrupts
- präemptives Scheduling
- Erkennung und Schutz kritischer Abschnitte

2 Aufgabenbeschreibung

In dieser Aufgabe wird CoStubs um präemptives Scheduling erweitert. Wir wollen damit erreichen, dass unsere Prozesse quasiparallel ablaufen können, ohne dass sie explizit die CPU abgeben müssen. Dazu müssen wir in der Lage sein, bei Ablauf einer Zeitscheibe ein Rescheduling zu veranlassen und einen Prozesswechsel durchzuführen. Wir brauchen dazu die Unterstützung eines Timers, der uns nach festgelegten Zeitintervallen regelmäßig unterbricht. Diese Aufgabe übernimmt im PC der **Programmable Interval Timer (PIT)** , ein Baustein, der so programmiert werden kann, dass er regelmäßig einen Interrupt auslöst.

Wir müssen den Interrupt geeignet behandeln und dabei beachten, dass wir uns von nun an Gedanken über den nebenläufigen Zugriff auf gemeinsame Daten machen müssen. Hier gilt es, alle kritischen Abschnitte in CoStubs zu identifizieren und zu schützen.

3 Vorbereitungsfragen

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

- Was ist präemptives Scheduling und wie kann es realisiert werden?
- Wie kann man eine quasiparallele Abarbeitung von Prozessen durch ein präemptives Scheduling erreichen?

- Was wäre ein passendes Beispiel für einen nebenläufigen Zugriff auf gemeinsame Daten?
- Was ist ein Programmable Interval Timer?
- Wie funktioniert der Programmable Interval Timer (PIT)?
- Wie kann man diesen konfigurieren (Beschreibt eine Konfigurationssequenz!)?
- Was ist ein Interrupt?
- Wie kann die CPU Interrupts unterscheiden?
- Wie werden Interrupts behandelt?
- Können neue Interrupts während einer Interruptsbehandlung auftreten?
- Wenn ja, werden diese dann umgehend behandelt?
- Wenn nein, werden die blockierten Interrupts durch das Gerät später nochmal ausgelöst?
- Zwei Prozesse A und B sollen in einer Endlosschleife immer wieder ihren Namen ausgeben. Diese Ausgabe dauert 10 Takte. Wenn Prozess A eine Zeitscheibe von 10 Takten und Prozess B eine Zeitscheibe von 100 Takten zugewiesen wird, welche Ausgaben sind zu erwarten?

4 Implementierungshinweise

Die Aufgabe besteht im Wesentlichen aus drei Teilen.

Teil A

Hier muss zunächst der Timer-Baustein programmiert werden. Dies geschieht in der Klasse PIT.h. Aufbauend auf dieser Klasse ist dann eine Klasse Clock.h zu implementieren, welche den vom PIT.h generierten Interrupt behandelt. Clock.h verwaltet unsere Systemzeit (in Ticks) und aktiviert regelmäßig den Scheduler. Programmiert die Uhr so, dass alle 20ms ein Interrupt auftritt.

Testet die Uhr zunächst, ohne den Scheduler zu aktivieren. Gebt dazu einfach innerhalb der Uhr

ein einziges Zeichen (ohne Zeilenvorschub) aus. Wenn ihr dieses Zeichen regelmäßig seht, habt ihr den Baustein prinzipiell korrekt programmiert und den Interrupt soweit richtig behandelt. Überprüft nun die korrekte Zeiteinstellung, indem ihr an einer fixen Bildschirmposition einen kleinen Propeller jede Sekunde ein Stück weiterdreht. Schaut zur Kontrolle auf eine Uhr. Der Propeller ist während der Abgabe zu zeigen!

Zur Behandlung der Interrupts: Jeder Interrupt-Quelle ist eine Vektornummer zugeordnet. Diese Nummern sind in der vorgegebenen Enumeration InterruptVector .h deklariert. Die Clock-Klasse muss sich von der vorgegebenen Klasse Gate.h ableiten, dem Gate-Konstruktor die Vektornummer mitteilen und eine handle() Methode definieren, um Interrupts behandeln zu können. Gate sorgt dann automatisch dafür, dass ein Eintrag an der richtigen Stelle der Vektortabelle erfolgt. Wenn alles korrekt ist, wird dann bei jedem Timer-Interrupt automatisch Clock::handle() aufgerufen.

Schaut euch an dieser Stelle noch einmal genau an, wie die Interruptverarbeitung vonstatten geht.

Hier ist noch zu beachten, dass alle Interrupt-Quellen von einem externen Baustein verwaltet werden, dem Programmable Interrupt Controller (PIC.h). Jede Interrupt-Quelle muss explizit über diesen Baustein angestellt werden, sonst wird er nicht zur CPU durchgeleitet. Ferner muss jeder Interrupt-Handler dem PIC die Behandlung des Interrupts mitteilen. Wenn man diese Bestätigung vergisst, kommt der Interrupt nie wieder durch. In der vorgegebenen Klasse PIC.h sind alle Steuerfunktionen für den PIC.h definiert. Ferner sind dort auch die Interrupt-Quellen vermerkt. PIC::PIT ist die betreffende Nummer unseres Timers.

Teil B

Nun wollen wir ein präemptives Scheduling mit Hilfe des Timers durchführen. Ruft dazu zunächst die reschedule()- Methode des Schedulers an geeigneter Stelle in Clock::handle() auf und sorgt dafür, dass in Coroutine:: startup () explizit für jede Coroutine die Interrupts angestellt werden. Dies muss geschehen, bevor ihr body() aufruft. Wir müssen das so machen, weil wir in einem Interrupt-Handler einen Prozesswechsel durchführen. Die Interrupts sind deshalb solange gesperrt, bis wir wieder aus dem Handler zurückkehren. Da neue Prozesse am Anfang noch niemals losgelaufen sind, sind sie auch noch nie unterbrochen worden. Wir müssen explizit einmal dafür sorgen, dass sie unterbrechbar sind. Die Interrupts können mit Hilfe der Klasse CPU.h angestellt und gesperrt werden.

Wenn unser erstes Scheduling erwartungsgemäß läuft, wollen wir nun jedem Prozess ein individuelles Zeitquantum (gemessen in den Ticks unserer Uhr) zuteilen. Dieses Quantum wird in der vorgegebenen Klasse Schedulable.h verwaltet. Wir müssen nun den Scheduler ein wenig über-

arbeiten. Dem aktiven Prozess soll erst dann die CPU entzogen werden, wenn sein Quantum abgelaufen ist. Führe dazu eine Methode checkSlice () beim Scheduler ein, die überprüft, ob die Zeitscheibe des aktuell laufenden Prozesses abgelaufen ist. Wenn ja, soll checkSlice () ein Rescheduling veranlassen. Ruft von nun an in Clock :: handle () checkSlice () anstelle von reschedule () auf.

Experimentiert mit verschiedenen Zeitscheiben für verschiedene Prozesse und beobachtet die Auswirkungen.

Hinweis

Sofern es Probleme geben sollte, bei denen euer System gewisse Interrupts nicht wahrnimmt, sie dementsprechend verloren gehen, dann ist der folgende Hinweis für euch relevant.

Schreibt eine Assembler Routine, die das Anschalten der Interrupts sowie den Wechsel des CPU Modus (CPU-Halt) vornimmt (siehe cpu.asm). Diese Routine könnt ihr dann in der CPU.hKlasse mithilfe des **extern**-Schlüsselworts ansprechen und benutzen. Dies sollte den oben beschriebenen Fehler vermeiden.

Teil C

Im Moment sind alle unserer Prozesse präemptiv gesteuert und geben die CPU nur über den Ablauf der Zeitscheibe ab. Was ist aber nun, wenn ich kooperative Prozesse hinzufüge, die explizit die CPU abgeben, sich freiwillig schlafen legen und irgendwann wieder aufgeweckt werden?

Überlegt euch, was dies für die Ready-Liste des Schedulers bedeutet. Findet alle kritischen Abschnitte in eurem System und schützt sie entsprechend.

Zu diesem Zweck stehen euch die Klassen CPU.h und IntLock.h zur Verfügung. Zum Testen implementiert ihr in Hello :: body() eine Endlosschleife in der yield () aufgerufen wird.

4.1 Vorgabe / Hinweise

In der Vorgabe zur Aufgabe 3 findet ihr zwei main-Dateien. Diese sind zum Testen eures Codes gedacht.

Die Datei mainInt.cc sorgt dafür, dass eine **while**-Schleife unendlich lang ausgeführt wird. Diese Implementierung eignet sich damit perfekt, um eure interruptgesteuerte Ausgabe vorzunehmen.

Die Datei mainPre.cc sorgt, wie der Name schon vermuten lässt, für eine präemptive Abarbeitung der vorhandenen Prozesse. Die Voraussetzung hierfür ist natürlich die korrekte Implementierung der hierfür notwendigen Klassen durch euch.

In der vorgegebenen mainPre.cc wird für die Prozesse keine Zeitscheibe explizit eingestellt, was dafür sorgt, dass alle mit dem gleichen Quantum initialisiert werden. Sorgt dafür, dass man für jeden Prozess das Quantum individuell einstellen kann. Es soll beobachtbar sein, dass ein Prozess mit einer längeren Zeitscheibe mehr Ausgaben schafft als einer mit einer kürzeren Zeitscheibe.

4.2 Weitere Informationen

- Informationen zum **Programmable Interval Timer (PIT)**
- Klassendiagramm 1
- Sequenzdiagramm 2

Bitte beachtet, dass die Diagramme nicht immer vollständig sind.

Viel Erfolg!

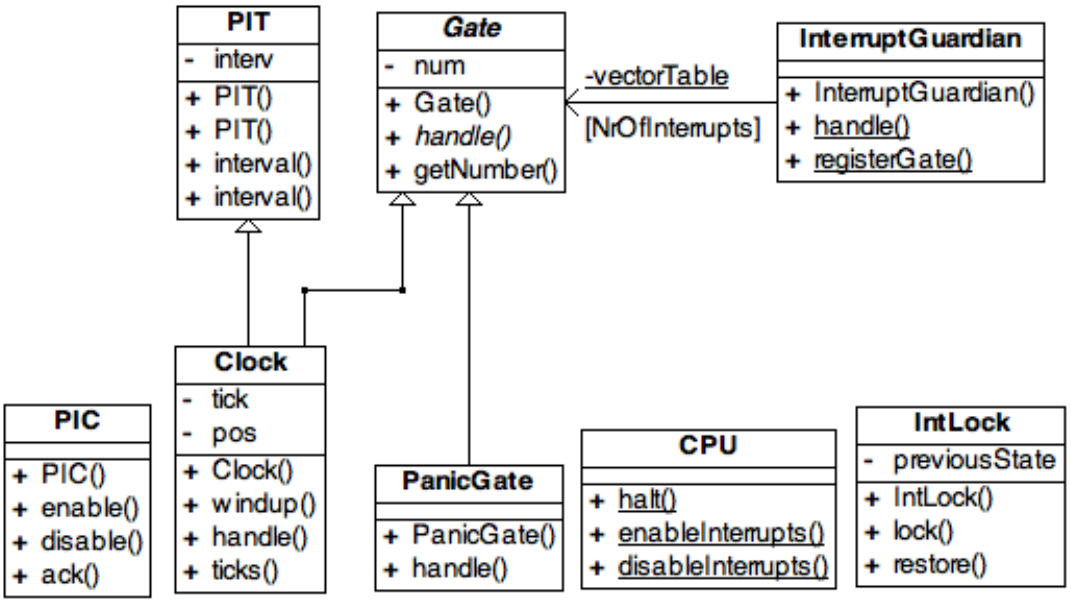

Abbildung 1: Klassendiagramm

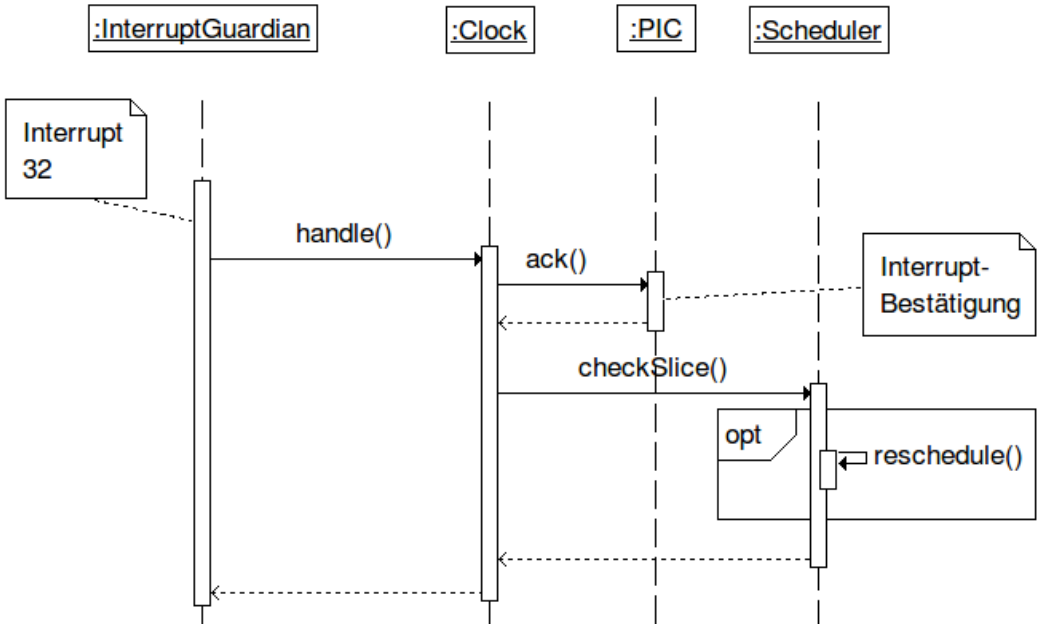

Abbildung 2: Sequenzdiagramm