Aufgabe 5

1 Lernziel

- Grobgranulare, nichtblockierende Synchronisationsmechanismen
- Semaphorenbasierte Synchronisation für Anwendungen

2 Aufgabenbeschreibung

In dieser Aufgabe bekommt CoStubs einen Monitor, welcher den Kern schützt. Damit dies funktioniert, müssen alle Zugriffe auf Funktionen des Kerns über eine definierte Systemschnittstelle erfolgen.

3 Vorbereitungsfragen

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

- Wie erfolgt die Interrupt-Behandlung bisher? Wenn ein Interrupt eintritt, ab wann können neue Interrupts auftreten?
- Wozu dienen die Interrupt-Sperren (IntLock) und wann ist ihr Einsatz bisher notwendig?
- Wozu dienen bei der Interrupt-Behandlung Prolog und Epilog?
- Worin unterscheiden sie sich?
- Welchen Vorteil bringt die Aufteilung?
- Was ist ein Monitor?
- Wozu wird er verwendet? Welches Konzept ersetzt er?

- Was ist eine Semaphore?
- Was ist private Vererbung?

4 Implementierungshinweise

4.1 Nichtblockierende Synchronisation

Zunächst müsst ihr die Synchronisation in CoStubs umarbeiten, da diese von nun an nicht mehr durch Interrupt-Sperren realisiert wird. Die Interrupt-Behandlung wird in Prolog und Epilog aufgeteilt.

Betrachtet zuerst die vorgegebene Monitor-Implementierung. Versucht sie zu verstehen, bevor ihr mit der Implementierung beginnt. Achtet insbesondere auf den Zusammenhang von Monitor und Interrupt-Behandlung.

Zerlegt die Interrupt-Behandlung in Prolog und Epilog. Diese entsprechenden Methoden prologue und epilogue muss jedes Gate implementieren. (Die Methode handle entfällt dadurch.) Während des Prologes sind die Interrupts gesperrt. Der Rückgabewert des Prologes bestimmt, ob der Epilog aufgerufen werden soll. Beim Epilog sind die Interrupts aktiv.

Bei der Tastatur soll die Methode analyzeScanCode im Epilog ausgeführt werden. Überlegt, warum dies sinnvoll ist. Dadurch müssen nicht nur die analysierten Tastatur-Eingaben gepuffert werden, sondern auch die einzelnen ScanCodes. Dazu benötigt ihr einen zweiten Puffer.

Überlegt, wie die Behandlung von Timer-Interrupts aufgeteilt werden sollte. Sollten Prozesswechsel im Prolog oder Epilog erfolgen?

Bisher konnten 'Benutzer-Programme' (wie Hello) Datenstrukturen des Kerns (etwa die ReadyList) manipulieren. Nun sollen Benutzer- und System-Teil voneinander getrennt werden. Als Schnittstelle dient die Klasse Thread, von der sich alle Anwendungsprogramme ableiten. Da Thread privat von Activity erbt, haben abgeleitete Klassen nicht mehr direkt Zugriff auf Basisklassen im Kern. Thread ist für das Betreten und Verlassen des Kerns verantwortlich. Zur leichteren Handhabung gibt es analog zur Klasse IntLock eine Klasse KernelLock.

Da jetzt nicht mehr über Interruptsperren synchronisiert wird, müsst ihr auch die Coroutinenimplementierung entsprechend anpassen.

4.2 Blockierende Synchronisation auf System- und Nutzerebene

Bei vorherigen Aufgaben habt ihr vielleicht schon bemerkt, dass es ungünstig ist, während der Bildschirmausgabe unterbrochen zu werden. Wenn der nächste Prozess auch Daten ausgibt, ist das Ergebnis kaum lesbar.

Die Vorgabe enthält Beispiel-Anwendungen (Hello-Threads). Diese geben Informationen auf dem Bildschirm aus und erwarten danach Eingaben des Benutzers. Dazwischen darf kein anderer Prozess Ein-/Ausgabeoperationen machen. Stellt euch zur Veranschaulichung vor, dass Anton nach einem Passwort fragt. Direkt nach der Ausgabe 'Bitte Passwort eingeben:' wird Anton unterbrochen und der Scheduler aktiviert Berta. Berta liest nun das Passwort und gelangt damit an geheime Informationen.

Um solche Unterbrechungen zu vermeiden, wird als Ein-/Ausgabeschnittstelle eine Console implementiert. Der Zugriff auf die Console darf erst erfolgen, nachdem diese reserviert (attach) wurde. Nach der Benutzung muß die Console wieder freigegeben werden (detach). Die innere Synchronisation erfolgt dabei über eine Semaphore.

Implementiert zunächst die KernelSemaphore für den kooperativen Fall. Wie der Name andeutet, wird diese innerhalb des Kerns benutzt. Überlegt euch, was dies für den Schutz von kritischen Datenstrukturen bedeutet.

Benutzerprogramme, die die Console verwenden, befinden sich nicht im Kern. Für sie ist die Nutzung der KernelSemaphore damit nicht angebracht. Deshalb soll für sie die Semaphore implementiert werden. In dieser wird nicht die gesamte Funktionalität der KernelSemaphore erneut implementiert (beachtet die Vererbungshierarchie). Überlegt euch, welche Erweiterungen vorgenommen werden müssen und setzt diese um. Verwendet den Monitor für die interne Synchronisation.

Implementiert diese Klassen am Besten ohne in das Skript o.ä. zu schauen.

4.3 Problemlösungen

Falls ihr die Fehlermeldung 'undefined reference to __stack_chk_fail' bekommt, so müsst ihr das Makefile im bin-Ordner anpassen. Fügt zu den CXXFLAGS das Flag '-fno-stack-protector' hinzu. Beachtet, dass die CXXFLAGS sowohl für Cygwin als auch für Linux gesetzt werden. Ändert die für euch passende Stelle oder zur Sicherheit beide.

Viel Erfolg!