Protokoll der Praktikumsdurchführung
BS Praktikum 2 – Gruppe 1

Ivan Morozov und Chandrakant Swaneet Kumar Sahoo
sowie
Kai Bursch und Matthias Nitsche

16.11.2013

Wir haben uns mit einer anderen Gruppe zusammengeschlossen und die Aufgaben gemwiensam zu bearbeiten und Probleme gemeinsam zu lösen. Morozov und Sahoo sowie Nitsche und Brusch haben sich zusammengesetzt.

Aufgabe 1 – Makefile:

Unser makefile lautet:

```
CC=/usr/bin/gcc
CFLAGS= -pthread -g -Wall -lpthread

TARGET = $1

$(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c
```

Mit „`make critsect02`“ können wir die Datei erstellen.

Aufgabe 2 - Philosophenproblem:

Entwurf und Lösungsmerkmale:

Unsere Dateien sind „`main.c`“, „`main.h`“, „`monitor.c`“ und „`Makefile`“. In der `main.h` Datei stehen die Deklarationen und globalen (Präprozessor-Konstanten) drinnen. Die `main.c` Datei enthält das Hauptprogramm mit der Initialisierung der Philosophen, dem Lebenslauf der Philosophen(`eat()`, `think()`... etc.) und dem Inputloop welches Benutzereingaben einliest und ausführt.

Bei der Initialisierung werden die die Semaphoren, die Cond_vars und die Defaultwerte für die Philosophen und das Mutex initialisiert. Danach werden nach und nach die Philosophen erzeugt. Sie erhalten einen Pointer auf ihre ID. Die ID wurde vorher beim Initialisieren in ein `p_ids[]` rein geschrieben. Wir können nicht die Adresse der ID selber übergeben, da die Thread-erzeugung verspätet passiert und die Threads dann fehlerhafte Ids erhalten.

Nachdem die main-Prozedur in `main.c` die Threads erzeugt hat, geht sie in einen InputLoop und liest in einer endlosschleife Benutzereingaben ab und delegiert den Input an `handle_quit` und `handle_command`.

Handle_quit prüft ob die Eingabe „`Q`“ bzw. „`q`“ war. In diesem Fall schreibt sie zuerst den `QUIT`-Befehl auf alle Threads(durch das Array) und befreit sie von früheren Benuzter-Blockierungen. Damit kann das Programm beendet werden, selbst wenn einige Philsophen vom Benutzter immernoch blockiert waren. Danach werden alle Threads signalisiert(falls sie `HUNGRY` waren) und werden dazu befohlen zu joinen. Dies verhindert, dass Philosophen im `HUNGRY` feststecken und gar nicht erst zum Beenden kommen können. Die Threads quitten aus ihren Endlosschleife(`philo`-Prozedur) und beenden mit `pthread_exit`. Danach werden die Threads sowie die Synchronisationsobjekte gelöscht und das `main`-Programm beendet. Beim fehlerhaften Beenden wird der Benutzer über die Konsole benachrichtigt. Es werden im Normalfall aufgrund des Aufräumens keine Synchronisationsobjekte hinterlassen.

`Handle_command` prüft ob die Eingabe eine gültige PhilosophenID und ein gültiger Befehl ist und führt diesen aus. Beim Blockier-befehlt der Philosophe zum „`BLOCK`“ aufgefordert. Beim Befreien(„`UNBLOCK`“) wird ein `sem_post` auf dem genannten Thread aufgerufen. Das `sem_post` haben wir nicht im entsprechenden Philothread geschrieben, da es aufgrund

seiner Blockierung sich selbst nicht befreien kann. Beim Proceed wird einfach der Befehl auf das Thread-Befehls-Array geschrieben und delegiert.
Die Philsosophen-Thread-Prozedur speichert zunächst ihr engekommene PhislosophenID damit sie über diesen Index auf die ihn relevanten Information im `philo_state[]`, `stick_state[]`, `cond[]`, `semaphores[]` und `input_commands[]` zugreifen kann. Sie geht danach in eine Endlosschleife in dem sie Denken, Sticks-aufheben, Essen und Sticks-ablegen. Sollten sie ein `QUIT`-Befehl erhalten haben, beenden sie die Schleife und den Thread(`pthread_exit`). Die Think- und Eat-Prozedur sind beides For-Schleifen mit den entsprechend durhc die Konstanten definierten Iterationenanzahl. Bei jeder Iteration auf einen angekommennen Proceed/Block-Befehl überprüft und entsprechend blockiert/übersprungen. Wir haben Prüfung(und Ausführung) der Blockierbefehle hier implementiert, damit der Phislosoph sofort während seiner Aktionen `THINK` und `EAT` blockiert werden kann. Diese Implementation heißt auch, dass der Philsosoph nicht im `HUNGRY` durch einen Semaphor blockiert wird.
In monitor.c stehen `get_sticks`, `put_sticks` und das `disp_philso_states`.
Im `get_sticks` wird, solange beide Sticks besetzt auf der eigenen `cond_var` blockiert. Davor wechselt er noch in den `HUNGRY` –State, da das `THINK` jetzt nich mehr zutrifft. Sobald beide Sticks gleichzeitig frei sind, reserviert er beide Sticks und wechselt in `EAT`. Bei beiden Zustandsänderungen wird `disp_philso_states` aufgerufen. Diese Prozedur ist durch einen cirtical-section gesichert, da sonst die Threads z.B. die Sticks im „`USED`“ hinterlassen könnten ohne selber im Zustand `EAT` zu sein. Oder es könnten die Sticks wieder besetzt sein, obwohl der Thread aus der While-Schleife kam und sie auch besetzten will. Solche Inkonsistenzen werden durch den Mutex verhindert.
Das `put_sticks` legt einfach beide Sticks zurück, setzten den Philsophen auf `THINK` und signalisiert die benachbarten Philosophen. Wir haben dies auch als einen critical-section, da sonst die Philsoophen beim Zurpüklegen unterbrochen werden und evtl. die benachbarten Philosophen nicht signalisieren oder nur einer der beiden Sticks-zurück gelegt wird.
Außerdem ist es nicht sinnvoll Semaphoren hier zu verwenden, da Semaphoren nicht abhängig von einer `cond_var` blockieren können. Wir wollen natürlich nicht den Philosophen wecken, wenn bereits nur einer beider Sticks frei ist. Daher ist `ptherad_cond_wait` und `pthread_cond_signal` korrekt.
Macros erleichteren den Zugriff auf die benachbarten Philosophen und die relevanten Sticks in dieser Datei.
Das `disp_philo_states` prüft zunächst auf benachbarte essende Philsophen, benachrichtigt in dem Fall und gibt alle Philsophen-States aus.
In der main.h Datei stehen die Konstanten für die diversen Zustände, sowie Macros für den/das jeweils linke(n)/rechte(n) Phislophen/Stick und einige sonstige Konstanten wie die Anzahl der Philosophen(`NPHILO`) und `das ASCII_NUM_OFFSET`. Außerdem werden hier den Prozeduren Doxygen-Kommentare angefügt.

Kommentare:

- Wir haben den gcc-Compiler im makefile auch den Befehl `-std=c99` gegeben, damit wir Iterationsvariablen direkt im Kopf der For-Schleifen definieren können und diese daher lokal halten können. Das Default-Verhalten lässt diese (für uns als AI-ler angewohnte) Deklaration nicht zu.
- Obwohl die Entwicklung auf XCode auf Mac comfortabler als auf der VM ist, ist es nicht klug auf dem Mac zu entwickeln, da viele Sachen auf der VM nicht mehr funktionierten. Nächstes mal, werden wir direkt auf openSuse in der VM schreiben und auf der VM schrittweise testend entwickeln.
- Nicht nur haben wir viel über Threads und Synchronisierung in C verstanden, sondern auch über die Organisation einer größeren Gruppe – und dass größere Gruppe schwerer zu koordinieren sind und nicht unbedingt (für diesen Aufgabenkontext) einen Nutzen gebracht haben.

- Wir haben auch festgestellt, dass das Git-Protokoll zumsammen mit Github sehr nützlich als Versionsmanagement für die Gruppenarbeit und die Verteilung war.
- Bezüglich Doxygen haben wir festgestellt, dass es ein comfortables und leistungsstarkes Tool zum automatischen Generieren von HTML-Dokumentationen ist.
- Doxygen unterstützt nur C Code, aber keine Präprozessor-Anweisungen.