ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 3

Christoph van Heteren-Frese (Matr.-Nr.: 4465677),
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 3. Mai 2013

Aufgabe 1

Eine binäre Semaphore kann wie ein Schloss eingesetzt werden. Die Spezifikation einer binärer Semaphore ist in [1, S. 54] und die Implementierung von `Lock` und `Unlock` in [1, S. 55] zu finden. Initialer Wert für `s: true` (bzw. 1).

Aufruf	Auswirkung
P1: s.P	`s = false`; P1 betritt kritischen Bereich
P2: s.P	`s = false`; P2 muss warten: *aktiv* → *blockiert*
P3: s.P	`s = false`; P3 muss warten: *aktiv* → *blockiert*
P1: s.V	P1 verlässt kritischen Bereich; `s = true`; P3: *blockiert* → *bereit*; P3 betritt kritischen Bereich; `s = false`
P4: s.P	`s = false`; P4 muss warten: *aktiv* → *blockiert*
P3: s.V	P3 verlässt kritischen Bereich; `s = true`; P2: *blockiert* → *bereit*; P2 betritt kritischen Bereich; `s = false`
P5: s.P	`s = false`; P5: *aktiv* → *blockiert*

Tabelle 1: Protokoll der Aufrufe

Erläuterung: Die Tatsache, das P3 in diesem Beispiel als zweiter Prozess den kritischen Bereich betritt, ergibt sich durch die vorgegebene Reihenfolge der Aufrufe, da P3 der nächste Prozess ist, der eine V-Operation ausführt und „[...] Prozesse P- und V-Operationen stets paarig – in dieser Reihenfolge – ausführen [müssen][vgl. 1, S. 55].“

Grundsätzlich gilt: „[...] ein Aufruf von s.V() realisiert den Übergang von *blockiert* → *bereit* für einen der auf s blockierten Prozesse [...][vgl. 1, S. 55].“ Welcher Prozess das ist, hängt von der Prozessverwaltung des Betriebssystems ab.

Aufgabe 2

a)

Eher nein. Ein Prozess wartet immer nur auf einen Semaphore, welcher aber selbst wieder auf einen anderen warten kann. Faktisch wartet der letzte Prozess dann zwar auf zwei Prozesse, algorithmisch gesehen ist er aber nur von einem Semaphore abhängig und wartet auch nur auf diesen. Vergleichbar ist dies mit einem Zug der vor einem Semaphore wartet. Unabhängig davon ob nur ein anderer Zug oder mehrere für das “Halt”-Signal

zuständig sind, sieht der wartende Zug nur ein Signal und kann nicht erkennen welche oder wie viele Züge dafür verantwortlich sind.

b)

Ob ein Prozess, der auf einen Semaphore wartet rechenbereit sein kann, hängt davon ob, ob das Betriebssystem Semaphore unterstützt oder nicht. Wenn das Betriebssystem Semaphore unterstützt, so setzt es den Zustand des wartenden Prozesses auf "blockiert" und dieser ist nicht "bereit". Wenn diese Unterstützung nicht gegeben ist, so glaubt das Betriebssystem wegen fehlender Informationen dass dieser Prozess rechenbereit ist, auch wenn höhere Schichten die Ausführung blockieren.

c)

Für eine Verallgemeinerung des Semaphorkonzeptes spricht, dass die wartenden Prozesse dann die Information erhalten könnten auf wie viele Semaphore sie warten und in Folge dessen Hardwareoptimierungen beim Füllen der Pipeline darauf Rücksicht nehmen könnten. Dagegen spricht allerdings die zunehmende Komplexität der Semaphore. Schon der informale Beweis der Funktionalität von binären Semaphoren ist extrem aufwändig. Ein korrekter und hinreichender Beweis von allgemeinen Semaphore ist entsprechend noch aufwändiger und schwieriger.

Aufgabe 3

Gegenseitiger Ausschluss bedeutet (nach Definition von Maurer) dass "sich zu jeder Zeit höchstens ein Prozess im kritischen Abschnitt befindet". Der Algorithmus von Barz sichert den gegenseitigen Ausschluss nur dann, wenn der Parameter "n" (bzw. die Variable "val") mit 1 initalisiert wird.
Nachfolgend wird daher gezeigt, dass der Algorithmus von Barz sicher stellt, dass sich immer nur maximal n Prozesse im kritischen Abschnitt befinden. In der Initalisierungs-Funktion wird der Wert von "val" auf n gesetzt und danach überprüft ob dieser Wert gleich 0 ist. Wenn dem so ist, so darf nie ein Prozess den kritischen Abschnitt betreten und wird deshalb im Anschluss mit cs.Lock gesperrt und ist nicht mehr erreichbar.
Sobald ein Prozess den kritischen Abschnitt betreten möchte, ruft dieser die Funktion P() auf. Hierbei wird dann zuerst der kritische Abschnitt vorerst gesperrt (cs.lock) und dann wird auch der Zugriff auf die Variable "val" (mutex.lock) gesperrt. Danach wird der Wert von "val" um eins verringert, um zählen zu können, wie viele Prozesse noch in den kritischen Abschnitt dürfen. Falls noch "Platz" für weitere Prozesse ist (also val größer 0), wird der Eintritt der anderen Prozesse gewährt, in dem der kritsche Abschnitt wieder entsperrt wird (cs.Unlock). Anschließend wird der Zugriff auf die Variable "val" wieder entsperrt (mutex.Unlock). Durch das Sperren der Variable "val" wird sicher gestellt, dass keine falschen Informationen in dieser gespeichert sein können.
Wenn ein Prozess nun den kritischen Abschnitt verlassen möchte, ruft dieser die Funktion V() auf. Auch hier wird erst wieder der Zugriff auf die Variable "val" gesperrt, so dass nur

diese Funktion den Parameter ändern kann. Da dieser Prozess jetzt für anderen Prozess "Platz macht", wird der Wert von "val" um eins erhöht (x.val++). Sofern der kritische Abschnitt vorher gesperrt war, weil die maximale Anzahl der Prozesse im kritischen Abschnitt erreicht wurde, wird dieser jetzt entsperrt (if x.val == 1 { x.cs.unlock()}). Ist die maximale Anzahl (n) der Prozesse im kritischen Abschnitt noch nicht erreicht worden, so wurde der k.A. durch die Funktion P() auch noch nicht gesperrt. Anschließend wird wieder der Zugriff auf die Variable "val" zugelassen, in dem mutex.unlock() ausgeführt wird.
Da mit "mutex.lock()" und "mutex.unlock()" immer sicher gestellt wird, dass die Variable "val" nicht irrtürmlich geändert wird, ist die Anzahl der freien Plätze im kritischen Abschnitt jederzeit korrekt. Auf dieser Basis wird schließlich der kritische Abschnitt gesperrt und entsperrt. Sofern n mit 1 initalisiert wird und V() nur dann verwendet wird, wenn vorher P() verwendet wurde, sichert der Algorithmus von Barz den gegenseitigen Ausschluss.

Aufgabe 4

Nachfolgend soll die Korrektheit des Staffelstab-Algorithmus begründet werden. Es wird davon ausgegangen, das der Algorithmus genau dann korrekt arbeitet, wenn er „Maßnahmen zur Vermeidung von Datenverlusten und -inkonsistenzen [...], die durch den nebenläufigen Zugriff mehrerer Prozesse auf gemeinsamen Daten entstehen können [...]" umsetzt, also eine Sperrsynchronisation[1] implementiert [vgl. 1, S. 19]. Dazu gehört insbesondere, die folgenden Eigenschaften: *Gegenseitiger Ausschluss*, *Verklemmungsfreiheit*, *Behinderungsfreiheit* und *Fairness*. Darüber hinaus muss er natürlich auch seinen Bestimmungszweck erfüllen. Die nachfolgenden Ausführungen beziehen sich auf den Algorithmus, der in [1] auf Seite 80 abgedruckt ist.

Jede Prozessklasse besitzt ein eigenes Semaphor `s[k]`, das auf den speziellen und den gemeinsamen Teil des Protokolls aufgeteilt ist. Dabei gilt die Invariante `s[0] + ... + s[n-1] <= 1`, sodass höchstens eins der beteiligten Semaphore *frei* ist [vgl. 1, S. 81]. Da jede mögliche Anweisungsfolge mit `e.P()` beginnt und mit einer V-Operation endet, ist der *gegenseitige Ausschluss* gewährleistet. Des Weitern wird bei jedem Austritt einer der auf ein s[k] blockierten Prozesse deblockiert, sofern es einen gibt. Somit sind die Protokolle *verklemmungsfrei*. Will ein Prozess den kritischen Abschnitt betreten, wir ihm unverzüglich der Eintritt gewährt, sofern sich kein Prozess im kritischen Bereich befindet. Somit ist der Algorithmus auch *Behinderungsfrei*. Die *Fairness* des Algorithmus ergibt sich aus seiner Struktur.

Literatur

[1] Christian Maurer. *Nichtsequentille Programmierung mit Go 1 Kompakt*. Springer Vieweg, 2012. ISBN 978-3642299681.

[1] Hier: Synchronisation mit allgemeinen Wartebedingungen