Betriebssysteme **WS 2007/2008**
Dr. Henrik Brosenne Georg-August Universität Göttingen
Institut für Informatik

Übung 5

Aufgabe 1

1. Implementiere ein *C*-Programm, das zwei Threads startet, zwischen denen eine Race-Condition auftritt. Die beiden Threads sollen auf eine gemeinsame globale Variable `in` zugreifen, indem sie diese in eine für jeden Thread jeweils lokale Variable `next_free_slot` kopieren, `next_free_slot` um 1 erhöhen und anschließend diesen Wert nach `in` zurückschreiben. Das Hauptprogramm soll `in` mit 0 initialisieren. Die beiden Threads sollen die Variable `in` jeweils `COUNT_MAX` mal erhöhen und sich danach beenden. Das Hauptprogramm soll per `pthread_join` auf die Terminierung der beiden Threads warten und abschließend den Wert von `in` auf der Standardausgabe ausgeben.

   Hinweise:

   - Zwecks Auffrischung der Kenntnisse zur Thread-Programmierung kann Aufgabe 2 von Übung 2 und die zugehörige Musterlösung verwendet werden. Da der Rückgabewert der Threads nicht relevant ist, kann als zweiter Parameter von `pthread_join` auch `NULL` verwendet werden.
   - `COUNT_MAX` ist eine selbst zu definierende Konstante. Da `COUNT_MAX` sehr groß werden kann, soll für alle relevanten Variablen der Typ `unsigned long int` verwendet werden.
   - Variablen, die von mehreren Threads gemeinsam genutzt werden, müssen als `volatile` deklariert werden, z.B.: `volatile long int in`.

     Ansonsten kann es nämlich passieren, dass der Compiler Maschinencode generiert, der den Inhalt der Variablen `in` aus Geschwindigkeitsgründen in einem CPU-Register statt im Hauptspeicher ablegt. Da jeder Thread seinen eigenen Registersatz hat, der bei einem Kontextwechsel umgeschaltet wird, würden dann die Threads nicht mehr auf eine gemeinsame Variable zugreifen.

     Ebenso sorgt `volatile` dafür, dass einige weitere Codeoptimierungen unterbleiben. Z.B. könnte es sonst passieren, dass der Compiler bei folgendem Quelltext `turn = 1; if (turn == 1) { ... }` erst gar keinen Code für die `if`-Abfrage generiert, sondern dafür sorgt, dass der then-Zweig immer ausgeführt wird, weil der Compiler davon ausgeht, dass `turn == 1` immer erfüllt ist, weil ja unmittelbar vorher `turn` auf `1` gesetzt wurde.

2. Lasse das Programm laufen. Stimmt der ausgegebene Wert von `in` mit dem doppelten Wert von `COUNT_MAX` überein? Wie würden sich Race-Conditions bemerkbar machen?

3. Experimentiere, welche Größenordnung `COUNT_MAX` haben sollte, damit auf dem verwendeten Rechner mit hoher Wahrscheinlichkeit eine Race-Condition auftritt.

4. Die Race-Condition kann auch provoziert werden, indem innerhalb des kritischen Bereichs die Kontrolle von einem Thread an den anderen Thread abgegeben wird. An welcher Stelle in der Lösung muss hierzu der Betriebssystemaufruf `sched_yield` eingefügt werden? Welche Größenordnung reicht nun für `COUNT_MAX` aus, damit auf dem von Dir verwendeten Rechner mit hoher Wahrscheinlichkeit eine Race-Condition auftritt?

Aufgabe 2

1. Realisiere für den kritischen Bereich aus Aufgabe 1 den wechselseitigen Ausschluss, durch Verwendung des Peterson-Algorithmus. Baue hierzu auf der in Aufgabenteil 1.4 erstellten Variante auf und verwende für `COUNT_MAX` ebenfalls den dort ermittelten Wert.

   Hinweis: In der Vorlesung ist für den Peterson-Algorithmus Pseudo-Code angegeben, weshalb die Definitionen von `boolean` und `flag` nicht 100 % der *C*-Syntax entsprechen!

2. Lasse das Programm laufen. Sorge ggf. durch Erhöhen von `COUNT_MAX` dafür, dass während das Programm läuft, auf der Kommandozeile das Programm `top` gestartet werden kann. Zu wie viel Prozent lastet der Prozess bzw. die Threads die CPU aus?

3. Die Zeilen der Implementierung des Peterson-Algorithmus unterscheiden sich für die beiden Threads nur darin, dass `0` und `1` vertauscht sind. Modifiziere daher den in der Vorlesung vorgestellten Peterson-Algorithmus, so dass er auf zwei *C*-Funktionen `void enter_criticalregion(int process)` und `void leave_criticalregion(int process)` aufgeteilt werden kann und für beide Threads ohne Modifikation verwendbar ist. Die beiden Funktionen sollen jeweils vor bzw. nach dem kritischen Bereich aufgerufen werden. Für den Parameter `process` soll `0` oder `1` übergeben werden, je nachdem, ob der Aufruf aus dem ersten oder dem zweiten Thread bzw. Prozess erfolgt.

4. Erläutere kurz, warum der Peterson-Algorithmus auch funktioniert, wenn bei der Zuweisung an die Variable `turn` bzw. beim Vergleich mit der Variable `turn` statt der Nummer des jeweils anderen Prozesses die Nummer des eigenen Prozesses verwendet wird.