ALP4 SoSe 2013, Di. 16-18

Lösung Übungsblatt 7

Christoph van Heteren-Frese (Matr.-Nr.: 4465677)
Sven Wildermann (Matr.-Nr.: 4567553)
Tutor: Alexander Steen, eingereicht am 7. Juni 2013

Aufgabe 1

a)

Nebenläufigkeit kann eingesetzt werden, wenn Algorithmen nichtlinear rekursiv sind. Ein Beispiel dafür ist Mergesort. Prinzipiell lässt sich mit mehr oder weniger aufwendigen Konstrukten jeder Sortieralgorithmus nebenläufig realisiertn. In vielen Fällen ist dies jedoch nocht sinnvoll (s.u.).

b)

Paralleles Sortieren ist aber nicht immer sinnvoll. Insbesondere auf Einprozessorsystemen ergibt sich nur dann ein Geschwindigkeitsvorteil, wenn verzahnte Aktivitäten unterschiedliche Ressourcen vewenden. Des Weiteren ist jeder Geschwindigkeitsvoteil hinfällig, wenn auf das Ergebnis einer anderen Operation gewartet werden muss, damit mit der Bearbeitung fortgefahren werden kann.

c)

Implementierung von Mergesort:

```
1  package main
2  import "fmt"
3  func merge(a []int, m int) []int {
4    n, b := len(a), make([]int, len(a))
5    for j, i, k := 0, 0, m; j < n; j++ {
6      if i < m && k < n {
7        if a[i] < a[k] {
8          b[j] = a[i]
9          i++
10       } else {
11         b[j] = a[k]
12         k++
13       }
14     } else if i < m {
15       b[j] = a[i]
16       i++
17     } else if k < n {
18       b[j] = a[k]
19       k++
20     }
21   }
22   copy(a, b)
23   return a
```

```
}
func mergesort(a []int, e chan []int) {
  var ausgabe []int
  if len(a) > 1 {
    m := len(a) / 2
    c, d := make(chan []int), make(chan []int)
    go mergesort(a[:m], c)
    go mergesort(a[m:], d)
    <-c
    <-d
    ausgabe = merge(a, m)
  }
  e <- ausgabe
}
func main() {
  done := make(chan []int)
  numbers := []int{4, 3, 7, 4, 2, 5, 2, 7, 9, 1}
  go mergesort(numbers, done)
  ausgabe := <-done
  fmt.Print(ausgabe)
}
```

Aufgabe 2

```
package main
import "fmt"

const N = 3
type vector [N]int
type matrix [N]vector

func scalarproduct(v, w vector, p *int, d chan bool) {
  for j := 0; j < N; j++ {
    *p += v[j] * w[j]
  }
}
func column(a matrix, k int) (s vector) {
  for j := 0; j < N; j++ {
    s[j] = a[j][k]
  }
  return s
}
func product(a, b matrix) (p matrix) {
  done := make(chan bool)
  for i := 0; i < N; i++ {
    for k := 0; k < N; k++ {
      go scalarproduct(a[i], column(b, k), &p[i][k], done)
    }
  }
  for j := 0; j < N*N; j++ {
    <-done
  }
  return p
}
func toString(a matrix) (s string) {
  i := a[1][1]
  return "Test"
}
func main() {
```

```
36    a := matrix{vector{1, 2, 9}, vector{4, 5, 6}, vector{7, 8, 9}}
37    b := matrix{vector{9, 8, 7}, vector{6, 5, 4}, vector{3, 2, 1}}
38    c := product(a, b)
39    fmt.Printf("Hello\n" + c)
40  }
```

Aufgabe 3

Es wurden mehrfach die Anzahl der Prozesse und der Wert der Konstanten N geändert. Nachfolgend der Quellcode mit unterschiedlichen Verzögerungszeiten (durch Parametriesierung der Prozessnummer p):

```
1   package main
2
3   import (
4     "fmt"
5     "math/rand"
6     "time"
7   )
8
9   var (
10    Counter int
11    done    chan bool
12  )
13
14  func v(p int64) { time.Sleep(time.Duration(rand.Int63n(p*1e5))) }
15
16  func inc(n *int,p  int64) {
17    Accu := *n // "LDA n"
18    v(p)
19    Accu++ // "INA"
20    v(p)
21    *n = Accu //"STA n"
22    fmt.Println(*n)
23    v(p)
24  }
25
26  func count(p int64) {
27    const N = 5
28    for n := 0; n < N; n++ {
29      inc(&Counter,p)
30    }
31    done <- true
32  }
33
34  func main() {
35    Counter = 0
36    done = make(chan bool)
37    go count(800)
38    go count(50)
39    go count(1)
40    go count(200)
41    go count(30)
42    <-done
43    <-done
44    <-done
45    <-done
46    <-done
47    fmt.Printf("Zaehler = %d\n", Counter)
48  }
```

Aufgabe 4

a)

Die im Buch erwähnten Antinome beschreiben die Tatsache, dass der Scheduler selbst ein Prozess ist. Es wird die Frage aufgeworfen, ob sich der Scheduler selbst verwaltet. Somit müsste er sich auch selbst unterbrechen. Die Frage wäre dann, wie ein präemptiver Scheduler 'zurückfindet'.

Dies kann durch einen Timer-Interrupt mit hoher Priorität (Prioritätsmanagement des Prozessors, nicht Prozessprioritäten im BS) geschehen, der den Scheduler-Prozess aufruft. Das bedeutet, dass die Fähigkeit des Scheduler-Prozesses andere Prozesse verdrängen zu können bereits auf 'Hardware-Ebene' implementiert ist. Darüber hinaus kann festgelegt werden, dass bei präemptiven Scheduling zwischen zwei Zeitscheiben der Scheduling-Prozess selbst mit einer sehr hohen Priorität aufgerufen wird. Der Scheduler-Prozess ist also dementsprechend ein besonderer Prozess, der sich von 'normalen' Prozessen unterscheidet.

b)

Eine mögliche Strategie für das Verhalten eines Schedulers stellt das Zeitscheibenverfahren dar. Ziel ist eine gleichmäßige Verteilung der Prozessorkapazität und der Wartezeit auf die Prozesse. Dazu werden alle Prozesse ihrer Ankunftsreihenfolge nach bearbeitet. Nach Ablauf einer bestimmten Frist wird auf den nächsten Prozess umgeschaltet. Die Wahl des Zeitintervalls stellt ein Optimierungsproblem dar: Für kleine Bearbeitungszeiten wächst der Aufwand fur das häufige Umschalten, während große Zeiteinheiten einer 'First-come-first-serve' Strategie realisieren.

Dementsprechend werden ankommende Prozesse in einer Warteschlane eingereiht. Der Scheduler nimmt den fordersten Prozess, führt ihn für eine bestimmte Zeit aus und fügt ihn anschließend ans Ende der Schlange wieder an, falls er noch nicht abgearbeitet sein sollte. Eine Schleife sorgt dafür, dass sich dieser Ablauf ständig wiederholt.