GSS-Übungsblatt 3

Chamier, Eickhoff, Gäde, Hölzen, Jarsembinski · SoSe 2016

1 Speicherverwaltung

(a) Eine Seite setzt sich, wie in der Tabelle zu c, aus 4-Bit Seitennummer, 3-Bit Seitenrahmennummer und 1-Bit für Gültigkeit zusammen. Eine Seite ist also 8-Bit groß.
Die Adressen aus Teilaufgabe c) verwenden einen Offset von 12 Bit. Eine Kachel ist mit einem Offset von 12 Bit und einer Wortlänge von 1 Byte (8 Bit) $2^{12} \div 8 = 512Bit = 64Byte$ groß. Eine Seite ist genau so groß.

(c) i) 001 1111 1110 1000 = 0x1FE8

ii) >Page Fault<

iii) 000 0100 0111 0000 = 0x0470

iv) 101 0001 0000 0001 = 0x5101

(d) Pro kleine Seitengröße:

i) Mehr Kapazität für multiple Prozesse

ii) Effizientere Speichernutzung

Pro große Seitengröße:

i) Weniger Aufwand bei Adressberechnung

(e) Je kleiner die Seitentabellengröße ist, desto weniger Seiten können referenziert werden und desto größer sind die Seiten und Kacheln. Damit ist dann also der interne Speicher weniger fragmentiert.
Eine gute Seitengröße für die durchschnittliche Prozessgröße $p = 4MiB$ ist, durch Annäherung ermittelt, $1,6MiB$. Mit dieser Seitengröße belegt der durchschnittliche Prozess 3 Seiten, wobei die letzte Seite halb leer bleibt. Kleinere Prozesse haben noch einen gewissen Spielraum nach unten und größere Prozesse müssen einfach mehr Seiten verwenden.

GSS-Übungsblatt 3

Chamier, Eickhoff, Gäde, Hölzen, Jarsembinski · SoSe 2016

2 Seitenersetzungsalgorithmen

a)

NRU

t	1	2	3	4	5	6	7	8	9	10	11	12
Angeforderte Seite	1	2	3	4	2	1	2	5	6	2	6	3
Zugriffsart	r	w	w	r	r	r	r	w	r	w	w	r
Seitenalarm	j	j	j	j	n	j	n	j	j	n	n	J
Seiten Im Speicher	1	1	1	4	4	1	1	5	5	5	5	5
		2	2	2	2	2	2	2	6	6	6	6
			3	3	3	3	3	3	2	2	2	3

SCA

t	1	2	3	4	5	6	7	8	9	10	11	12
Angeforderte Seite	1	2	3	4	2	1	2	5	6	2	6	3
Zugriffsart	r	w	w	r	r	r	r	w	r	w	w	r
Seitenalarm	j	j	j	j	n	j	j	j	j	n	n	j
Seiten Im Speicher	1	1	1	2	2	2	4	1	1	2	2	3
		2	2	3	3	4	1	2	5	5	5	6
			3	4	4	1	2	5	6	6	6	2

Queue:	t	Elemente
	1	1R
	2	1R:2R
	3	1R:2R:3R
	4	1R:2R:3R → 2R:3R:1 → 3R:1:2 → 1:2:3 → 2:3:4R
	5	2R:3:4R
	6	2R:3:4R → 3:4R:2 → 4R:2:1R
	7	4R:2R:1R
	8	4R:2R:1R → 2R:1R:4 → 1R:4:2 → 4:2:1 → 2:1:5R
	9	2:1:5R → 1:5R:6R
	10	1:5R:6R → 5R:6R:2R
	11	5R:6R:2R
	12	5R:6R:2R → 6R:2R:5 → 2R:5:6 → 5:6:2 → 6:2:3R

R = Reference-Bit

3 Synchronisation

a)

Listing 1: Semaphore.py

```
class Semaphore():
   def __init__(self):
      self.W = 1 # Lock can be acquired
      self.NumberOfActiveReaders = 0 # No Active Readers
      self.Mutex = 1 # Changing NumberOfActiveReaders allowed

   def P(self, caller):
      type = caller.__class__.__name__

      if type == "Reader":
         if self.Mutex > 0
            if self.NumberOfActiveReaders == 0:
               self.W = 0
            self.Mutex = 0
            self.NumberOfActiveReaders+=1
            self.Mutex = 1
            return true
         else:
            return False

      elif type == "Writer":
         if self.W > 0:
            self.W = 0
            return True
         else:
            return False

      else:
         return False

   def V(self, caller):
      type = caller.__class__.__name__

      if type == "Reader":
         if self.Mutex > 0
            self.Mutex = 0
            self.NumberOfActiveReaders-=1
            if self.NumberOfActiveReaders == 0:
               self.W = 1
            self.Mutex = 1
            return true
         else:
            return False

      elif type == "Writer":
         self.W = 0

```

```
48          else:
49              return False
50
51
52  class Reader():
53      # Static Variable semaphore
54      semaphore = []
55
56      def __init__(self, s):
57          self.semaphore = s
58
59      def processReader(self):
60          # Try to acquire Lock
61          if self.semaphore.P(self):
62              self.readData()
63              self.semaphore.V(self)
64          else:
65          # Retry to acquire Lock
66              self.processReader()
67
68      def readData(self):
69          # Insert Read Data
70
71
72  class Writer():
73      # Static Variable semaphore
74      semaphore = []
75
76      def __init__(self, s):
77          self.semaphore = s
78
79      def processWriter(self):
80          # Try to acquire Lock
81          if self.semaphore.P(self):
82              self.writeData()
83              self.semaphore.V(self)
84          else:
85          # Retry to acquire Lock
86              self.processWriter();
87
88      def readData(self):
89          # Insert Write Data
```

b) In obiger Implementation werden Prozesse, die schreibend auf die Daten zugreifen wollen, benachteiligt, da ”Reader” unbegrenzt auf die Daten zugreifen können. Es wäre also möglich, dass der schreibende Prozess niemals Zugriff bekommt, da immer neue ”Reader” auf die Daten zugreifen. Um dies zu verhindern lässt sich z.B. eine Queue einbauen, die Prozesse nur nach und nach Zugriff auf die Daten erlaubt.