

Musterlösungen zu den Hausaufgaben auf
Blatt 8
der Übungen zur Vorlesung
“Grundlagen Betriebssysteme und Systemsoftware

G.Groh, 10.12.2007

Aufgabe 1.1 Lösungsvorschlag

Gegebener Code:

Aufgabe 1.1 Lösungsvorschlag

```
1: class Calc {
2:     Vector startWerte, puffer1, puffer2, ergebnis;
3:     Init init;
4:     Sum sum;
5:     Mult mult1, mult2;
6:
7:     public static void main( String[] args) {
8:         Calc _this = new Calc();
9:         _this.calc();
10: }
11:
12: private void calc() {
13:     startWerte = new Vector();    // Alle Puffer leer initialisieren
14:     puffer1 = new Vector();
15:     puffer2 = new Vector();
16:     ergebnis = new Vector();
17:
18:     init = new Init(startWerte); // Initialisierungsobjekt erstellen
19:     init.calc();                // Initialisierungsobjekt startet Berechnung
20:     mult1 = new Mult(startWerte, puffer1, 0); // Multiplikationsobjekt erstellen
21:     mult1.calc();               // Multiplikationsobjekt startet Berechnung
22:     mult2 = new Mult(startWerte, puffer2, 2); // Multiplikationsobjekt erstellen
23:     mult2.calc();              // Multiplikationsobjekt startet Berechnung
24:     sum = new Sum(puffer1, puffer2, ergebnis); // Summenobjekt erstellen
25:     sum.calc();                // Summenobjekt startet Berechnung
26: }}
27:
```

Aufgabe 1.1 Lösungsvorschlag

```
29:
30: class Init {
31:     Vector startWerte;
32:
33:     public Init( Vector _startWerte ) {
34:         startWerte = _startWerte;    // Übernehmen des zu füllenden Puffers
35:     }
36:
37:     public void calc() { // Alle 160000 möglichen Eingabewerte werden erstellt
38:         int[] wert; // Eine Parameterkombination wird in einen int[4] Array abgelegt
39:             // Dieser Array kommt dann in den Puffer 'startWerte'
40:         for (int a=1; a<=20; a++)
41:             for (int b=1; b<=20; b++)
42:                 for (int c=1; c<=20; c++)
43:                     for (int d=1; d<=20; d++) {
44:                         wert = new int[4];
45:                         wert[0]=a; wert[1]=b; wert[2]=c; wert[3]=d;
46:                         startWerte.add( wert );
47:                     }
48:     }}
49:
50:
```

Aufgabe 1.1 Lösungsvorschlag

```
51: class Mult {
52:     Vector startWert, puffer;
53:     int index;
54:
55:     public Mult( Vector _startWert, Vector _puffer, int _index ) {
56:         startWert = _startWert; // Übernehmen der Pufferobjekte
57:         puffer = _puffer;
58:         index = _index;         // Eintrag an index und an index+1 wird multiplizie
59:     }
60:
61:     public void calc() {
62:         int i=0, erg;
63:         int[] werte;
64:
65:         while (i < 160000 ) { // Solange nicht alle Berechnungen durchgeführt sind
66:             werte = (int[])startWert.elementAt(i); // Holen der Parameterkombination
67:             erg = werte[index] * werte[index+1]; // Berechnen der Multiplikation
68:             puffer.addElement( new Integer(erg) ); // Ergebnis in den Puffer eintragen
69:             i ++;
70:         }
71:     }}
72:
```

Aufgabe 1.1 Lösungsvorschlag

```
75: class Sum {
76:     Vector puffer1, puffer2, ergebnis;
77:
78:     public Sum( Vector _puffer1, Vector _puffer2, Vector _ergebnis ) {
79:         puffer1 = _puffer1;    // Übernehmen der Pufferobjekte
80:         puffer2 = _puffer2;
81:         ergebnis = _ergebnis;
82:     }
83:
84:     public void calc() {
85:         int i=0;
86:         Integer sum;
87:
88:         while ( i < 160000 ) {    // Solange nicht alle Berechnungen durchgeführt si:
89:             sum = new Integer( ((Integer)puffer1.elementAt(i)).intValue()    // Summ
90:                 + ((Integer)puffer2.elementAt(i)).intValue()); // berechne:
91:             ergebnis.add ( sum ); // Ergebnis in Puffer schreiben
92:             i ++;
93:         }
94:     }}
```

Änderungen:

Aufgabe 1.1 Lösungsvorschlag

Variante 1: "extends Thread"

```
(Z30): class Init extends Thread {
```

```
(Z51): class Mult extends Thread {
```

```
(Z75): class Sum extends Thread {
```

```
(Z19): init.start();
```

```
(Z21): mult1.start();
```

```
(Z23): mult2.start();
```

```
(Z25): sum.start();
```

Aufgabe 1.1 Lösungsvorschlag

Variante 2: "implements Runnable"

```
(Z30): class Init implements Runnable {  
(Z51): class Mult implements Runnable {  
(Z75): class Sum implements Runnable {
```

```
(statt Z19):  
Thread thread = new Thread( init, "Init" );  
thread.start();
```

```
(statt Z21):  
thread = new Thread( mult1, "Mult1" );  
thread.start();
```

```
(statt Z23):  
thread = new Thread( mult2, "Mult2" );  
thread.start();
```

```
(statt Z25):  
thread = new Thread( sum, "Sum" );  
thread.start();
```

Aufgabe 1.1 Lösungsvorschlag

Bei beiden Varianten: Umbenennen der `calc()`-Methoden:

```
(Z37) public void run() {  
(Z61) public void run() {  
(Z84) public void run() {
```

Jetzt sind vier Threads eingerichtet und gestartet. In der `run()` Methode gehört jetzt noch überprüft, ob im Puffer neue Werte vorhanden sind oder ob der Prozessor freigegeben werden soll. Für `init` ist nichts mehr zu tun, da `init` auf nichts warten muss und `add()` schon `synchronized` ist (vgl. Angabe).

Aufgabe 1.1 Lösungsvorschlag

Variante 1: yield()

zwischen Z65, Z66:

```
    if ( puffer.size() <= i ) {  
        yield();  
    } else { ...
```

zwischen Z69, Z70:

```
    }
```

zwischen Z88, Z89:

```
    if ( puffer1.size() <= i || puffer2.size() <= i ) {  
        yield();  
    } else { ...
```

zwischen Z92, Z93:

```
    }
```

Aufgabe 1.1 Lösungsvorschlag

Variante 2: `wait()`, `notify()`

zwischen Z65, Z66:

```
if ( startwerte.size() <= i ) {  
    synchronized(startwerte) { try{ wait(); } catch (InterruptedException e) {} }  
} else { ...
```

zwischen Z69, Z70:

```
    synchronized(puffer) { notify(); }  
}
```

zwischen Z88, Z89:

```
if ( puffer1.size() <= i || puffer2.size() <= i ) {  
    synchronized(puffer1) { try{ wait(); } catch (InterruptedException e) {} }  
} else { ...
```

zwischen Z92, Z93:

```
    synchronized(ergebnis) { notify(); }  
}
```

Ist die ganze Methode `run()` `synchronized`, verliert man sehr viel Parallelität.

Aufgabe 1.2 Lösungsvorschlag

Weil immer nur ein Thread von dem Prozessor bearbeitet werden kann, laufen die Threads nicht wirklich parallel ab.

Aufgabe 2.1 Lösungsvorschlag

Ein Prozesskontrollblock besteht u.a. aus folgenden Komponenten:

- Identifikatoren
 - Name des Prozesses
 - Name des Benutzers, für den der Prozess gerade arbeitet
- Stellung in der Prozesshierarchie
 - Name des Vaterprozesses
 - Namen der Sohnprozesse
- Zustandsinformation
 - Rechnerkernzustand
 - Arbeitszustand
 - Alarmzustand
 - Beschreibung zugeordneter Objekte/Betriebsmittel
- Rechte
 - Zugriffsrechte auf Dateien
 - Zugriffsrechte auf Segmente und Seiten
 - Prozesspriorität
- Betriebsmittelkonten
 - noch verfügbare Kontingente
 - Abrechnungsdaten

Aufgabe 2.2 Lösungsvorschlag

Desweiteren wird ein Warteraum für Prozesse benötigt, der die Prozesse hält, die sich im Zustand rechenbereit befinden. Die Operationen auf diesem Warteraum sollen wechselseitig ausgeschlossen ausgeführt werden.

Aufgabe 2.3 Lösungsvorschlag

Der Kontext-Switch wird am Beispiel von UNIX bei Ausführung der Systemoperation *sleep* skizziert:

Ein Kontext-Wechsel ist nur im Kernel Modus möglich. Im Kernel Modus ist der User-Modus Zustand des Prozesses auf dem Kernel Stack, der Bestandteil der User Struktur des Prozesses ist, abgelegt und verfügbar.

Ein Kontext-Wechsel in UNIX besteht darin, die User Strukturen der Prozesse auszuwechseln. Der Prozesskontrollblock ist Bestandteil der User Struktur.

Beim Aufruf der Operation *sleep* ist ein Warteraum, in den der Prozess eingefügt werden soll, anzugeben (z.B. I/O-Warteraum, oder warten auf Terminieren eines Kind-Prozesses). Bei der Ausführung von *sleep* werden vergrößert folgende Schritte durchgeführt:

- Maskieren von Interrupts;
- Lokalisieren der benötigten Warteschlange;
- Neuberechnung der Priorität des Prozesses;
- Einfügen des Prozesses in die Warteschlange;
- Aufruf der Operation *swtch*

Aufgabe 2.3 Lösungsvorschlag

Bei der Ausführung der Operation *switch* wird vom Scheduler der Prozess mit der höchsten Priorität ausgesucht und mit dieser Information wird die Operation *resume* aufgerufen. Mit der Operation *svpctx* wird zunächst der Zustand des noch aktuellen Prozesses aus den Registern in den Prozesskontrollblock des Prozesses gespeichert. Dann wird die Adresse des Prozesskontrollblocks des neu zu bindenden Prozesses geladen. Mit der Operation *ldpctx* wird der Zustand des neuen Prozesskontrollblocks in die Register geladen und der Kontext-Wechsel ist durchgeführt.

Aufgabe 3.1 Lösungsvorschlag

Generally, daemons run in the background doing things like printing and sending email. Since people are not usually sitting on the edge of their chairs waiting for them to finish, they are given low priority, soaking up excess CPU time not needed by interactive processes.

Aufgabe 3.2 Lösungsvorschlag

A PID must be unique. Sooner or later the counter will wrap around and go back to 0. Then it will go upward to, for example, 15. If it just happens that process 15 was started months ago, but is still running, 15 cannot be assigned to a new process. Thus after a proposed PID is chosen using the counter, a search of the process table must be made to see if the PID is already in use.

Aufgabe 3.3 Lösungsvorschlag

When the process exits, the parent will be given the exit status of its child. The PID is needed to be able to identify the parent so the exit status can be transferred to the correct process.

Aufgabe 3.4 Lösungsvorschlag

If all of the `sharing_flags` bits are set, the `clone` call starts a conventional thread. If all the bits are cleared the call is essentially a `fork`.