Praktikumsprotokoll Mikrorechentechnik II – „Simulation und Optimierung von Fertigungsprozessen“

Gruppe 50:
Marcel Junige, Fabian Kurz
Martin Laabs, Lars Lindenmüller

30. Mai 2005

Inhaltsverzeichnis

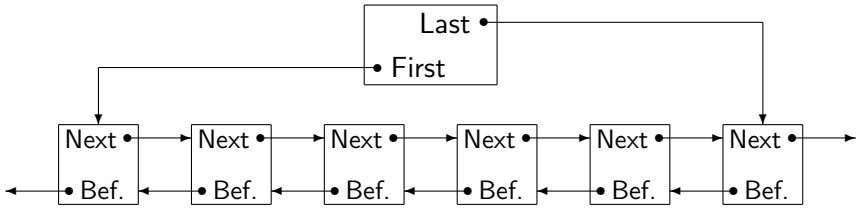

Abbildung 1: Doppelt verkettete Liste

1 Aufgabenstellung

Aufgabe des Praktikums war die Implementierung von Methoden zur Manipulation doppelt verketteter Listen im Rahmen des Simulationssystems ROSI.

ROSI wurde entwickelt um ereignisdiskrete Prozesse (insbesondere Fertigungsprozesse) zu simulieren und optimieren. Ein solches Simulationsmodell wird aus verschiedenen Bausteienen (Maschine, Warteschlange, Jobs, Steuerelementen) zusammengesetzt.

Bei der Optimierung durch stochastische Suchverfahren ist es nötig, die Reihenfolge der Jobs in einer Warteschlange verändern zu können. Abbildung 1 zeigt ein Modell einer solchen Warteschlange. Es handelt sich um eine doppelt verkettete Liste von Jobs: jeder Job hat einen Zeiger auf seinen Vorgänger und Nachfolger. Ein First- und Last-Zeiger verweisen auf das erste bzw. letzte Element.

Es waren die folgenden Methoden zu implementieren:

JOB CQueue::GetList (long n) Gibt einen Zeiger auf den n-ten Job der Warteschlangenliste zurück. Bei einem Wert von $n < 1$ wird `NULL` zurückgegeben.

int CQueue::ExchJobs (JOB A, JOB B) Vertauscht die Jobs `A` und `B`, die jeweils mit einem Zeiger referenziert werden.

int CQueue::ExchJobs (long Ax, long Bx) Dito, jedoch wird anstelle der Zeiger auf die zu vertauschenden Jobs die Jobnummer in der Liste angegeben.

2 Entwurf und Implementierung

2.1 CQueue::GetList

`CQueue::GetList` Gibt einen Zeiger auf den n-ten Job der Warteschlangenliste zurück. Bei einem Wert von $n < 1$ wird `NULL` zurückgegeben.

Die Implementierung von `GetList` erfolgt intuitiv: Es wird überprüft, ob der Wert `No` positiv ist, wenn dies nicht der Fall ist, wird `NULL` zurückgegeben. Ein Hilfszeiger `jp` wird vereinbart. Dieser wird mittels der Methode `GetJob` auf den ersten Job gesetzt.

In einer `for`-Schleife wird der Zeiger nun `No` mal auf den nächsten Job gesetzt. Falls kein nachfolgender Job exisitert, wird ebenfalls `NULL` zurückgegeben.

Wenn die `for`-Schleife ordnungsgemäß beendet wurde, zeigt der Hilfzeiger `jp` auf das `No`-te Element und kann zurückgegeben werden.

Quelltext

```
/*************************************************************************
   Es wird der No-te Job in der Warteschlange ausgegeben,
   ohne ihn aus der Warteschlange zu entfernen. Die Zaehlung
   beginnt mit 1. No = 0 -> NULL, No > Length -> NULL
*************************************************************************/
```

```
JOB  CQueue::GetList (long No) {
    JOB jp;                          // Hilfszeiger auf aktuellen Job
    long i;                          // Zaehlervariable

    if (No < 1) {                    // Ueberpruefe auf guelltigen Wert
        return NULL;                 // Es gibt keine neg. Elemente
    }

    jp=FirstJob();

    for(i=0;i<=No-1;i++) {           // Erster Job No = 0
        jp=jp->GetNext();            // durch Liste durchhangeln
        if (jp==NULL) {              // Job existiert nicht
            return NULL;             // NULL zurueckgeben
        }
    }

    // Wenn das Programm bis hier gekommen ist, zeigt jp auf den No-ten
    // Job. Dieser wird zurueckgegeben.

    return jp;
}
```

2.2 CQueue::ExchJobs (1)

`CQueue::ExchJobs` Vertauscht die Jobs `A` und `B`, die jeweils mit einem Zeiger referenziert werden.

Abbildung 2 verdeutlicht die dabei zu tätigen Zeigeroperationen: Die Vorgänger und Nachfolger von `A` und `B` werden in den Hilfzeigern `An`, `Ap`, `Bn` und `Bp` gespeichert. Mittels der Methoden `PutBefore` und `PutNext` werden sodann die Elemente `A` und `B` neu eingereiht. Dies geschieht in beide Richtungen: Sowohl `A` und `B` als auch ihre Vorgänger und Nachfolger müssen neu verknüpft werden.

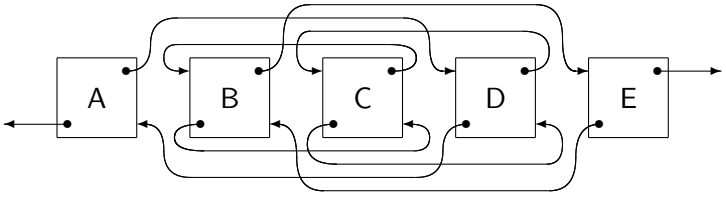

Abbildung 2: Vertauschung der Elemente `B` und `D` in einer doppelt verketteten Liste

Abbildung 2 stellt den einfachsten Fall dar, in dem zwei nicht benachbarte Listenelemente miteinander vertauscht werden. Es gibt jedoch folgende Sonderfälle zu beachten:

Element mit sich selbst vertauschen: In diesem Falle ist nichts zu machen.

Benachbarte Elemente vertauschen: Hier (Beispiel `A` vor `B`) ist zu beachten, dass der Vorgängerzeiger des Elements `B` auf `A` zeigt, bzw. der Nachfolgezeiger des Elements `A` auf das Element `B` zeigt. Würde man nun den gewöhnlichen Weg einschlagen, den ehemaligen Vorgänger von `B` (welcher `A` ist) als neuen Vorgänger von `A` zu vereinbaren, ist `A` sein eigener Vorgänger und die Listenstruktur ist zerstört. Dieses wird verhindert indem mal den Nachfolger von `A` auf `A`, den Vorgänger von `B` auf `B` setzt. Analog bei `B` vor `A`.

Eines/beide Elemente am Anfang oder Ende der Liste: In diesem Fall gibt es Pointer ins Leere, die nicht als Vorgänger oder Nachfolger eines Elements eingesetzt werden dürfen. Anstelle dessen muss der First- bzw. Last-Pointer neu gesetzt werden.

Quelltext

```
/*********************************************************************
   Es wird der Job A der Warteschlange mit Job B vertauscht.
*********************************************************************/
int CQueue::ExchJobs (JOB A, JOB B) {
    JOB An,Ap,Bn,Bp;                    // Nachfolger bzw. Vorgaenger von A,B

    if(A==B) {                          // Job A identisch Job B
        return ROSI_NOP;                // Nichts zu tun
    }
    if((A==NULL) || (B==NULL)) {        // Einer der Jobs ist NULL
        return ROSI_ERROR;              // Error!
    }

    An=A->GetNext();                    // Nachfolger von A
    Ap=A->GetBefore();                  // Vorgaenger von A
    Bn=B->GetNext();                    // Nachfolger von B
    Bp=B->GetBefore();                  // Vorgaenger von B

    // Bei benachbarten Jobs aufpassen, dass Nachfolger/Vorgaenger kein
    // Zeiger auf sich selbst wird!

    if (B->GetNext() == A) {
        Ap = A;
        Bn = B;
    }
    else if (A->GetNext() == B) {
        An = A;
        Bp = B;
    }


    // Pointer-Vertauschung

    A->PutBefore(Bp);
    A->PutNext(Bn);
    B->PutBefore(Ap);
    B->PutNext(An);

    // Bei Verknuepfungen von An, Ap, Bn, Bp ggf. First- und Last-Zeiger
    // Anpassen!

    if (Bn == NULL) {                   // B letztes Element
        Last = A;                       // A wird letztes Element
    }
    else {                              // B nicht letztes Element
```

```
        Bn->PutBefore(A);              // A vor Bn einreihen
    }
    if (Bp == NULL) {                  // B erstes Element
        First = A;                     // A wird erstes Element
    }
    else {                             // B nicht erstes Element
        Bp->PutNext(A);                // A nach Bp einreihen
    }

    if (An == NULL) {                  // Analog zur B
        Last = B;
    }
    else {
        An->PutBefore(B);
    }
    if (Ap == NULL) {
        First = B;
    }
    else {
        Ap->PutBefore(B);
    }

    return ROSI_OK;
}
```

2.3 CQueue::ExchJobs (2)

Die Methode `ExchJobs` wird zusätzlich noch in einer zweiten Variante implementiert, in der sich die Eingabeparameter im Typ unterschieden. Während die obige Variante Eingabeparameter vom Typ `JOB` erwartet, soll bei dieser Variante die *Jobnummer* angegeben werden können.

Dies ist einfach zu implementieren, da die dazu benötigten Methoden mit `GetList` und `ExchJobs` schon zur Verfügung stehen.

Die Behandlung ungültiger Eingabeparameter wird ebenfalls durch die vorhandenen Methoden uebernommen, so dass sich die Methode als Dreizeiler implementieren lässt:

Quelltext

```
/*********************************************************************
   Es wird der Ax-te Job der Warteschlange mit dem Bx-ten Job getauscht.
*********************************************************************/
int CQueue::ExchJobs (long Ax, long Bx) {
    return ExchJobs(GetList(Ax),GetList(Bx));
}
```

3 Diskussion

Die zu implementierenden Methoden funktionierten auf Anhieb wie gefordert. Dies konnte schnell durch Vertauschungsoperationen in einer Liste im Konsolenmodus von ROSI nachgewiesen werden. Dabei wurden auch die oben beschriebenen Sonderfälle berücksichtigt.

Es wurden im Folgenden zwei verschiedene Modelle von Produktionsstrategien mit ROSI simuliert und durch stochastische Methoden optimiert:

3.1 Flowshop

Im *Flowshop*-Modell durchläuft das zu fertigende Produkt eine bestimmte Anzahl von Bearbeitungsstationen, wobei die Reihenfolge dieser Stationen für alle Aufträge gleich ist. Die Stationen werden in ROSI als `Maschinen` bezeichnet. Zu finden ist die optimale Anordnung der Bearbeitungsschritte, bei der die Fertigungszeit minimal wird.

Die Auslastung der einzelnen Maschinen und Warteschlangen wird von ROSI in einem Gant-Diagramm dargestellt. Abbildung 3 zeigt das Gant-Diagramm eines Produktionsablaufes vor der Optimierung. In diesem Zustand, die neun Jobs sind der Reihe nach $(1 \ldots 9)$ angeordnet, benötigt die Produktion 2:23h.

Durch zufälliges Verändern der Jobreihenfolge per Hand (Reihenfolge 6, 9, 5, 4, 3, 1, 7, 2, 8) ergab sich eine leicht verbesserte Zeit von 2:15h.

Abbildung 4 stellt den Zustand nach einer Optimierung durch 500 zufällige Vertauschungen dar. Die günstigste sich ergebene Zeit von 1:59h trat bei der Jobreihenfolge 9, 8, 1, 5, 6, 7, 2, 3, 4 auf.

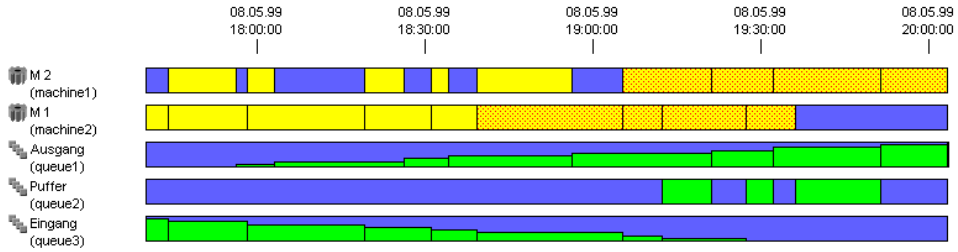

Abbildung 3: Flowshop-Modell, nicht optimiert

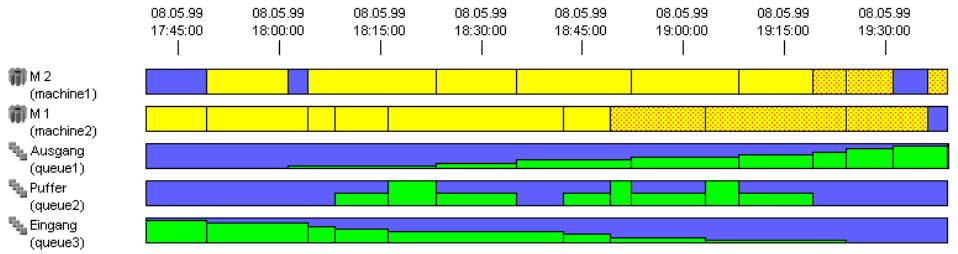

Abbildung 4: Flowshop-Modell, Optimierung mit 500 Schritten

Offensichtlich wurde durch diese Umverteilung der Aufträge die Auslastung der Maschinen verbessert. Im Gant-Diagramm ist zu sehen, dass die Leerlaufzeit der Maschinen (blau) nach der Optimierung deutlich geringer geworden ist.

3.2 Jobshop

Eine anderes Modell zur Beschreibung von Produktionsabläufen ist der *Jobshop*. Im Gegensatz zum Flowshop gibt es hier keine feste Reihenfolge, in der die Jobs die Maschinen durchlaufen.

Auch für ein solches Modell wurde eine Optimierung durch zufällige Vertauschungen der Jobreihenfolge vorgenommen.

Im Originalzustand betrug die Bearbeitungszeit 1d 9:45h, das entsprechende Gant-Diagramm wird in Abbildung 5 dargestellt. Durch zufälliges Vertauschen per Hand (Reihenfolge 3, 6, 10, 4, 5, 2, 7, 8, 9, 1) ergab sich eine schlechtere Zeit von 1d, 11h.

Erst die Optimierung mit 1000 Schritten brachte eine wesentliche Verbesserung auf 1d, 5:21h, wie in Abbildung 6 dargestellt. Deutlich ist auch hier wieder zu erkennen, dass die Leerlaufzeiten deutlich verringert werden konnten.

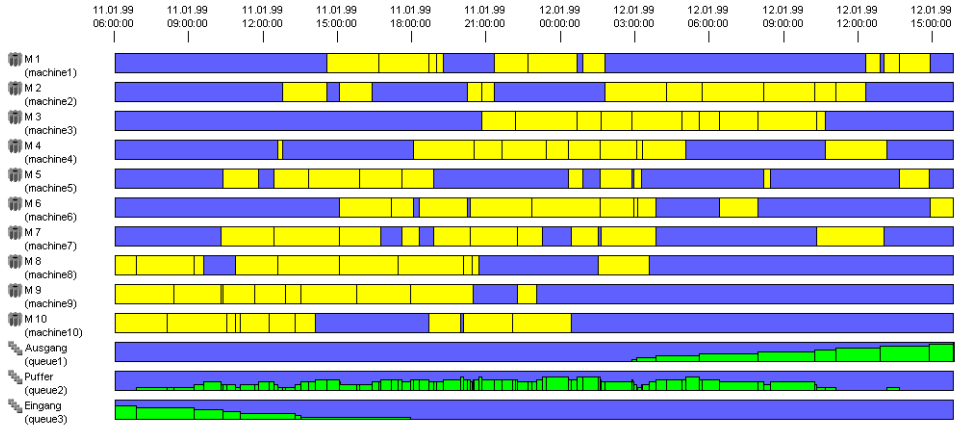

Abbildung 5: Jobshop-Modell, nicht optimiert

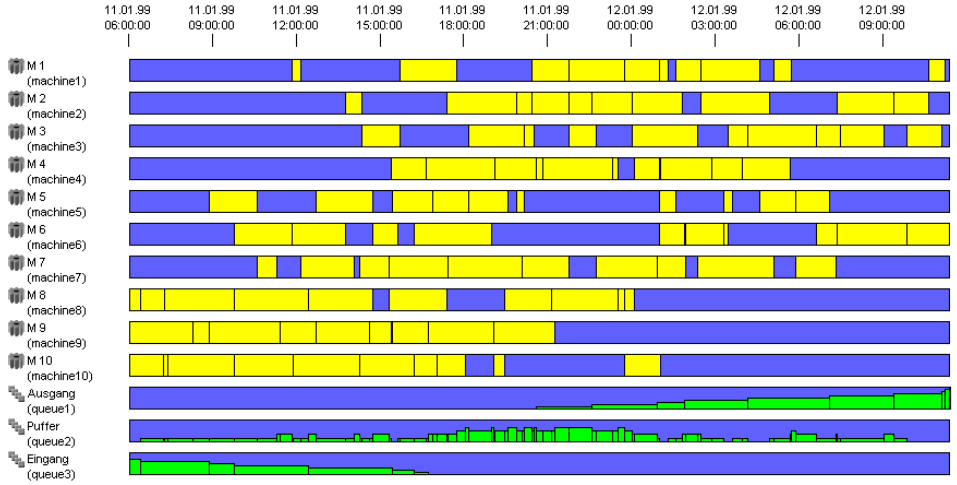

Abbildung 6: Jobshop-Modell, Optimierung mit 1000 Schritten