Script **generated by TTT**

Title: Grundlagen_Betriebssysteme (16.11.2015)

Date: Mon Nov 16 13:47:30 CET 2015

Duration: 88:23 min

Pages: 29

Semaphore

Semaphore wurden 1968 von Dijkstra eingeführt. Ein ***Semaphor*** (Signalmast) ist eine ganzzahlige Koordinierungsvariable s, auf der nur die drei vordefinierten Operationen (Methoden) zulässig sind:

Initialisierung,

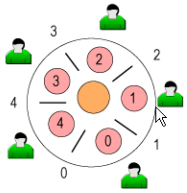
Prolog P (kommt von protekt),

Epilog V (kommt von vrej).

Generated by Targeteam

Beispiel Philosophenproblem

Zu den klassischen Synchronisationsproblemen zählt das Problem der ***speisenden Philosophen*** ("Dining Philosophers"). In einem Elfenbeinturm leben fünf Philosophen. Der Tageszyklus eines jeden Philosophen besteht abwechselnd aus Essen und Denken. Die fünf Philosophen essen an einem runden Tisch, auf dem in der Mitte eine Schüssel voller Reis steht. Jeder Philosoph hat seinen festen Platz an dem Tisch und zwischen zwei Plätzen liegt genau ein Stäbchen. Das Problem der Philosophen besteht nun darin, dass der Reis nur mit genau zwei Stäbchen zu essen sind. Darüber hinaus darf jeder Philosoph nur das direkt rechts und das direkt links neben ihm liegende Stäbchen zum Essen benutzen. Das bedeutet, dass zwei benachbarte Philosophen nicht gleichzeitig essen können.



Realisierung mit Semaphoren

Generated by Targeteam

Variante 1

Für eine Lösung des Philosophenproblems seien die folgenden 5 Semaphore definiert: stab_0, stab_1, ..., stab_4, wobei jedes der 5 Semaphore mit 1 initialisiert ist. Jeder Philosoph j, mit j ∈ {0,...,4}, führe den folgenden Algorithmus aus:

```
philosoph_j:
    while (true) {
        Denken;
        stab_i.P;                    mit i = j
        stab_i.P;                    mit i = j + 1 mod 5
        Essen
        stab_i.V;                    mit i = j
        stab_i.V;                    mit i = j + 1 mod 5
    }
```

Generated by Targeteam

Semaphore

Semaphore wurden 1968 von Dijkstra eingeführt. Ein ***Semaphor*** (Signalmast) ist eine ganzzahlige Koordinierungsvariable s, auf der nur die drei vordefinierten Operationen (Methoden) zulässig sind:

Initialisierung,

Prolog P (kommt von protekt),

Epilog V (kommt von vrej).

Synchronisierung von Java Threads

Java unterstützt synchronisierte Methoden.

```
public synchronized void methodname(...) { ... }
```

Eine synchronisierte Methode kann nur exklusiv von einem Java Thread betreten werden.

Beispiel TakeANumber Class

Java Monitor

Ein Monitor ist ein Java-Objekt, das synchronisierte Methoden enthält.

Ein Monitor stellt sicher, dass nur ein Thread zur Zeit in einer der synchronisierten Methoden sein kann. Bei Aufruf einer synchronisierte Methode wird das Objekt gesperrt.

Während das Objekt gesperrt ist, können keine anderen synchronisierten Methoden des Objekts aufgerufen werden.

Kritische Abschnitte können in Java als Objekte mit den zugehörigen synchronisierten Methoden spezifiziert werden.

Beispiel TakeANumber Class

Nur ein Thread kann zu einem Zeitpunkt eine Nummer ziehen

```
class TakeANumber {
     private int next = 0; //next place in line
     public synchronized int nextNumber() {
         next = next + 1; return next;
     } //nextNumber
} //TakeANumber

public class Customer extends Thread {
     private static int number = 1000; //initial customer ID
     private int id;
     private TakeANumber takeANumber;
     public Customer(TakeANumber gadget) {
         id = ++number;
         takeANumber = gadget;
     } //Customer constructor
     public void run() {
         try {
             sleep( (int)(Math.random() * 1000) );
             System.out.println("Customer "+id+" takes ticket " + takeANumber.nextNumber());
         } catch ......
     } //run
} //Customer

public class Bakery {
     public static void main(String args[]) {
```

Beispiel TakeANumber Class

```
     private int id;
     private TakeANumber takeANumber;
     public Customer(TakeANumber gadget) {
         id = ++number;
         takeANumber = gadget;
     } //Customer constructor
     public void run() {
         try {
             sleep( (int)(Math.random() * 1000) );
             System.out.println("Customer "+id+" takes ticket " + takeANumber.nextNumber());
         } catch ......
     } //run
} //Customer

public class Bakery {
     public static void main(String args[]) {
         System.out.println("Starting Customer threads");
         TakeANumber numberGadget = new TakeANumber();
         .........
         for (int k = 0; k < 5; k++) {
             Customer customer = new Customer(numberGadget);
             customer.start();
         }
     } // main
} //Bakery
```

Zuordnung zum "Nummer ziehen"

Synchronisierung von Java Threads

Java unterstützt synchronisierte Methoden.

```
public synchronized void methodname(...) { ... }
```

Eine synchronisierte Methode kann nur exklusiv von einem Java Thread betreten werden.

Beispiel TakeANumber Class

Java Monitor

Ein Monitor ist ein Java-Objekt, das synchronisierte Methoden enthält.

Ein Monitor stellt sicher, dass nur ein Thread zur Zeit in einer der synchronisierten Methoden sein kann. Bei Aufruf einer synchronisierte Methode wird das Objekt gesperrt.

Während das Objekt gesperrt ist, können keine anderen synchronisierten Methoden des Objekts aufgerufen werden.

Kritische Abschnitte können in Java als Objekte mit den zugehörigen synchronisierten Methoden spezifiziert werden.

Synchronisation

Eine wichtige Systemeigenschaft betrifft die Synchronisation paralleler Ereignisse. Mehrere Prozesse konkurrieren um eine gemeinsame Ressource (CPU), oder greifen auf gemeinsame Daten zu.

Beispiele

Die beiden Beispiele basieren auf der speicherbasierten Prozessinteraktion, d.h. Prozesse (oder auch Threads) interagieren über gemeinsam zugreifbare Speicherzellen.

Beispiel: gemeinsame Daten

Erzeuger-Verbraucher-Problem

Definition: Wechselseitiger Ausschluss

Modellierung

Synchronisierungskonzepte

Semaphore

Synchronisierung von Java Threads

Verklemmungen

Verklemmung = Prozesse warten wechselseitig auf das Eintreten von Bedingungen; gemeinsame Verwendung von Ressourcen (CPU, Speicherzellen, EA-Geräte, Dateien) kann zu Verklemmungen führen.

Allgemeines

Belegungs-Anforderungsgraph

Die Zuteilung/Belegung und Anforderung von Ressourcen kann man sich an einem Graphen, dem Belegungs-Anforderungsgraph, veranschaulichen. Die Knoten sind die Prozesse und Ressourcen, die Kanten spiegeln Belegungen und Anforderungen wider.

Beispiel

Verklemmungs-Ignorierung

Vogel-Strauß-Politik und hoffen, dass nichts passiert.

Verklemmungs-Erkennung

Verklemmungs-Verhinderung

Verklemmungs-Vermeidung

Vergleich der Ansätze

Allgemeines

Es lässt sich zeigen, dass die folgenden Bedingungen ***notwendig und hinreichend*** dafür sind, dass eine Verklemmung auftreten kann.

1. Die gemeinsam benutzbaren Ressourcen können nicht parallel genutzt werden, d.h. sie sind nur ***exklusiv*** benutzbar.
2. Die zugeteilten/belegten Ressourcen können ***nicht entzogen*** werden, d.h. die Nutzung ist nicht unterbrechbar.
3. Prozesse ***belegen*** die schon zugeteilten Ressourcen auch dann, wenn sie auf die Zuteilung weiterer Ressourcen warten, d.h. wenn sie weitere Ressourcen ***anfordern*** .
4. Es gibt eine ***zyklische Kette*** von Prozessen, von denen jeder mindestens eine Ressource belegt, die der nächste Prozess in der Kette benötigt, d.h. zirkuläre Wartebedingung.

Verklemmungen

Verklemmung = Prozesse warten wechselseitig auf das Eintreten von Bedingungen; gemeinsame Verwendung von Ressourcen (CPU, Speicherzellen, EA-Geräte, Dateien) kann zu Verklemmungen führen.

Verklemmungs-Erkennung

In der Praxis häufig angewendete Strategie: Verklemmungen in Kauf nehmen, sie erkennen und beseitigen.

Erkennungs-Algorithmus

Ansatz 1: Suche nach Zyklen im Belegungs/Anforderungsgraph.

Ansatz 2: Prüfen, ob es eine Reihenfolge für die Ausführung der Prozesse gibt, so dass alle Prozesse terminieren können.

Vorgehen für Ansatz 2

1. Starte mit Prozessmenge P, die alle Prozesse enthält,
2. suche Prozess p aus P, dessen zusätzliche Anforderungen im aktuellen Zustand erfüllbar sind,
3. falls gefunden, simuliere, dass p seine belegten Ressourcen wieder freigibt,
4. entferne p aus P und gehe zu 2
5. falls kein Prozess mehr in P, dann terminiert Suche: keine Verklemmung,
6. falls $P \neq \emptyset$ und in Schritt 2 kein Prozess mehr gefunden wird, dessen Anforderungen erfüllbar sind, dann terminiert die Suche; P enthält die Menge der verklemmten Prozesse.

Auflösung einer Verklemmung in der Regel durch Abbruch einzelner Prozesse.

Verklemmungen

Verklemmung = Prozesse warten wechselseitig auf das Eintreten von Bedingungen; gemeinsame Verwendung von Ressourcen (CPU, Speicherzellen, EA-Geräte, Dateien) kann zu Verklemmungen führen.

Allgemeines

Belegungs-Anforderungsgraph

Die Zuteilung/Belegung und Anforderung von Ressourcen kann man sich an einem Graphen, dem Belegungs-Anforderungsgraph, veranschaulichen. Die Knoten sind die Prozesse und Ressourcen, die Kanten spiegeln Belegungen und Anforderungen wider.

Beispiel

Verklemmungs-Ignorierung

Vogel-Strauß-Politik und hoffen, dass nichts passiert.

Verklemmungs-Erkennung

Verklemmungs-Verhinderung

Verklemmungs-Vermeidung

Vergleich der Ansätze

Generated by Targeteam

Verklemmungs-Verhinderung

Die Verhinderungssverfahren beruhen darauf, dass man durch die Festlegung von Regeln dafür sorgt, dass mindestens eine der für das Auftreten von Deadlocks notwendigen Bedingungen nicht erfüllt ist.

Festgelegte lineare Reihenfolge

Bedingung "Zyklus tritt auf" in Belegungs-/ Anforderungsgraph darf nicht erfüllt werden. Dazu wird eine lineare Ordnung über den Ressourcen definiert: R1 < R2 < ... Rm.

Die Prozesse dürfen dann Ressourcen nur gemäß dieser Ordnung anfordern,

d.h. ein Prozess, der Ressource Ri belegt, darf nur Ressourcen Rj anfordern, für die gilt: Rj > Ri.

Andere Möglichkeiten sind:

a) Zuteilung aller benötigten Ressourcen zu einem Zeitpunkt.

b) zwangsweiser Entzug aller belegter Ressourcen, falls eine Ressourcen-Anforderung nicht erfüllt werden kann.

c) Spooling: nur der Spooler Prozess hat als einziger die Ressource zugeteilt; Zugriffe anderer Prozesse gehen über diesen Prozess. Beispiel ist das Spooling von Druckaufträgen.

Verklemmungs-Vermeidung

Die Vermeidungsverfahren basieren auf der Idee,

die zukünftigen Betriebsmittelanforderungen von Prozessen zu analysieren (bzw. diese geeignet abzuschätzen) und

solche Zustände zu verbieten (sie also zu verhindern), die zu Verklemmungen führen könnten.

Ein Beispiel ist der Bankiers-Algorithmus, der 1965 von Dijkstra entwickelt wurde.

Veranschaulichung des Algorithmus

Veranschaulichung des Verfahrens anhand eines Bankenszenarios.

Ausgangspunkt

Idee: Verwaltung von nur einer Ressourcen-Klasse, nämlich den Bankkrediten.

Bankier besitzt festen Geldbetrag und verleiht Geld an seine Kunden.

Alle Kunden sind dem Bankier bekannt, jeder Kunde hat einen eigenen maximalen Kreditrahmen, der kleiner als die zur Verfügung stehende Geldmenge des Bankiers ist.

Bankier hat weniger Geld als die Summe dieser Kreditrahmen.

Kunden können jederzeit Geld in der Höhe ihres jeweiligen Kreditrahmens fordern, müssen aber ggf. in Kauf nehmen, dass der Bankier diese Forderung erst nach endlicher Zeit erfüllt.

Aufgabe des Bankiers

Verleihen des Geldes so, dass jeder Kunde seine Geschäfte in endlicher Zeit durchführen kann und Kunden möglichst parallel bedient werden.

Idee : Reihenfolge für Kreditvergabe finden, so dass selbst bei denkbar ungünstigsten Kreditforderungen die Durchführung aller Geschäfte sichergestellt ist.

ungünstigster Fall : alle Kunden fordern Geld bis zu ihrem jeweiligen max. Kreditrahmen, ohne Kredite zurückzuzahlen.

Beispiel

Ausgangspunkt ist die folgende Situation der vier Kunden A, B, C, D (Einheiten jeweils in Tausend Euro):

Kunde	aktueller Kredit	max. Kreditrahmen
A	1	6
B	1	5
C	1	4
D	4	7

Es seien noch 3 Einheiten (Tausend Euro) in der Bank als Kredit verfügbar.

Annahme: Kunde C fordere eine weitere Einheit als Kredit an. Diese Anforderung wird probeweise zugeteilt und mündet nicht in einem Deadlock, da zuerst C (max noch 2 Einheiten bis Kreditrahmen) bedient werden kann.

Wenn C seine Einheiten wieder zurückgezahlt hat, können B oder D und schließlich A bedient werden.

Verklemmungs-Vermeidung

Die Vermeidungsverfahren basieren auf der Idee,

die zukünftigen Betriebsmittelanforderungen von Prozessen zu analysieren (bzw. diese geeignet abzuschätzen) und

solche Zustände zu verbieten (sie also zu verhindern), die zu Verklemmungen führen könnten.

Ein Beispiel ist der Bankiers-Algorithmus, der 1965 von Dijkstra entwickelt wurde.

Veranschaulichung des Algorithmus

Veranschaulichung des Verfahrens anhand eines Bankenszenarios.

Vergleich der Ansätze

Ansatz	Verfahren	Vorteile	Nachteile
Erkennung	periodischer Aufruf des Erkennungs Algorithmus	Prozesserzeugung wird nicht verzögert; erleichtert interaktive Reaktion	Verlust durch Abbruch
Verhinderung	feste Reihefolge bei der Zuteilung	keine Verklemmungsanalyse zur Laufzeit notwendig; Überprüfung während Übersetzung	keine inkrementellen Anfragen für Ressourcen möglich
	Zuteilung aller Ressourcen auf einmal	keine Präemption (Entzug) von Ressourcen notwendig; gut für Prozesse mit einzelner Aktivitätsphase (single burst)	ineffizient; verzögert Prozesserzeugung; Bedarf für Ressourcen muss bekannt sein
Vermeidung	Bankiers-Algorithmus	keine Präemption (Entzug) von Ressourcen notwendig	zukünftiger Bedarf muss bekannt sein; Prozesse können längere Zeit blockiert werden

Parallele Systeme - Modellierung, Strukturen

In einem allgemeinen Rechensystem finden eine Vielzahl paralleler Abläufe statt. Dieser Abschnitt beschäftigt sich mit den Eigenschaften paralleler Systeme.

Prozessverwaltung

Dieser Abschnitt behandelt das Prozesskonzept, Datenstrukturen zur Beschreibung des aktuellen Prozesszustandes sowie Dienste zum Aufruf von Systemfunktionen.

Prozesse repräsentieren eine Aktivität; sie haben ein Programm, Eingaben, Ausgaben und einen Zustand.

Dienste der Prozessverwaltung

Die Prozesse werden durch das Betriebssystem verwaltet.

Auslösende Ereignisse für die Erzeugung eines Prozesses

- Initialisierung des Systems.
- Systemaufruf zum Erzeugen eines Prozesses durch einen anderen Prozess.
- Benutzeranforderung zum Starten eines neuen Prozesses (Start einer Applikation).
- Auslösung eines Stapelauftrags (Batch Job).

Formen der Terminierung von Prozessen

- Normale Beendigung (freiwillig).
- Vorzeitige Beendigung bei einem durch den Prozess selbst erkannten Fehler (freiwillig).
- Vorzeitige Beendigung bei einem katastrophalen Fehler, erkannt durch das BS (unfreiwillig).
- Terminierung durch einen anderen Prozess (unfreiwillig).

Prozess-Auswahl, Strategien zur Prozessorzuteilung: ***Scheduling*** .

Prozessor-Anbindung; ***Dispatching*** .

Prozesskontrollblock

Jeder Prozess muss als eine ***Verwaltungseinheit*** beschrieben sein. Ein Prozess wird durch seinen Prozess-Kontext und dieser durch den Prozesskontrollblock (PCB) beschrieben. Ein ***PCB*** (process control block) enthält i.d.R. folgende Informationen:

- eindeutiger Name, z.B. fortlaufende Nummerierung des Prozesses (z.B. pid in Unix)
- Name des Benutzers, dem der Prozess zugeordnet ist
- der momentane Prozesszustand (wartend, rechnend, rechenwillig, ...)
- falls der Prozess rechnend ist, Angabe der zugeordneten CPU
- falls der Prozess wartend ist, eine Spezifikation des Ereignisses, auf das der Prozess wartet (z.B. Adresse eines Semaphors).
- die Ablaufpriorität des Prozesses
- die Inhalte der programmierbaren Register (die Anzahl ist abhängig von der jeweiligen CPU-Architektur), z.B. Kellerpointer.
- die Inhalte der Register, in denen die Anfangsadresse und Länge der prozessspezifischen Speicherabbildungstabellen enthalten sind (virtuelle Adressierung).
- das Programmstatuswort (PSW). Das PSW enthält weitere Informationen, die die CPU über den Prozess kennt, z.B. Ablaufcodes, Condition Codes
- PCB unter Linux ist durch die Struktur `task_struct` spezifiert; definiert unter `include/linux/sched.h` .

Prozesskonzept

Wir unterscheiden ***Benutzerprozesse*** , die Anwendungsprogrammen in Ausführung entsprechen, und ***Systemprozesse*** , die Programme/Dienste des Betriebssystems durchführen.

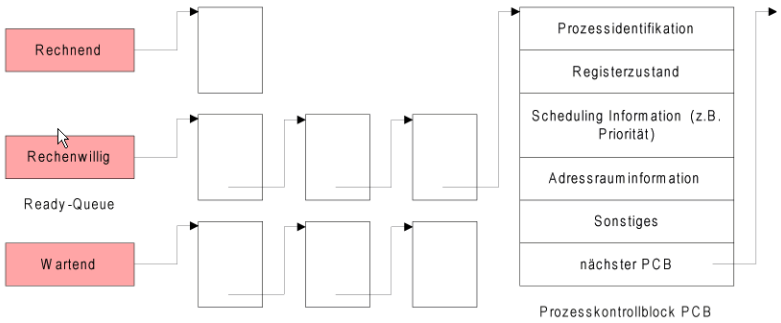
a) Jeder Prozess besitzt einen eigenen Prozessadressraum.

b) Spezielle Systemprozesse sind die ***Dämonen*** (engl. daemon); das sind Hilfsprozesse, die ständig existieren, die meiste Zeit aber passiv sind.

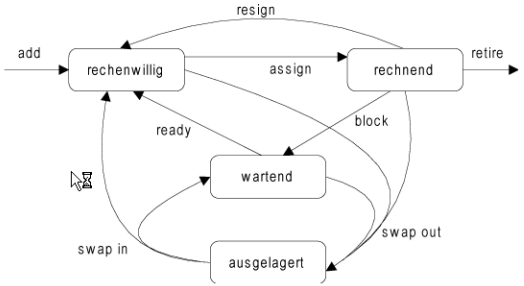
Prozesslisten

Die Prozesse werden in Zustandslisten verwaltet, die als verkettete Liste der PCBs realisiert sind.

für E/A-Geräte (z.B. Platte, Terminal) existiert i.d.R. jeweils eine eigene Warteschlange, die die PCBs der wartenden Prozesse enthält.



Das Prozess-Zustandsmodell unterscheidet neben den bereits vorgestellten Zuständen ***rechenwillig*** , ***rechnend*** , ***wartend*** auch den Zustand ***ausgelagert*** . Letzterer Zustand tritt ein, wenn der Adressraum aufgrund Speichermangels aus dem Arbeitsspeicher auf den Hintergrundspeicher verlagert wird ("swapping").



Zustandsübergänge sind:

`add` : ein neu erzeugter Prozess wird zu der Menge der rechenwilligen Prozesse hinzugefügt;

`assign` : als Folge eines Kontextwechsels wird dem Prozess die CPU zugeordnet;

`block` : aufgrund eines EA-Aufrufs oder einer Synchronisationsoperation wird der Prozess wartend gesetzt;

`ready` : nach Beendigung der angestoßenen Operation wechselt der Prozess in den Zustand rechenwillig; er bewirbt sich erneut um die CPU;

`resign` : dem rechnenden Prozess wird die CPU entzogen; er bewirbt sich anschließend erneut um die CPU;

`retire` : der aktuell rechnende Prozess terminiert;

`swap out` : der Prozess wird auf die Festplatte ausgelagert;

`swap in` : der Prozess wird von der Festplatte in den Arbeitsspeicher geladen.

Wir unterscheiden ***Benutzerprozesse*** , die Anwendungsprogrammen in Ausführung entsprechen, und ***Systemprozesse*** , die Programme/Dienste des Betriebssystems durchführen.

a) Jeder Prozess besitzt einen eigenen Prozessadressraum.

b) Spezielle Systemprozesse sind die ***Dämonen*** (engl. daemon); das sind Hilfsprozesse, die ständig existieren, die meiste Zeit aber passiv sind.