

Übung zur Vorlesung "Grundlagen Betriebssysteme und Systemsoftware"

(Prof. Dr. J. Schlichter, WS 2008 / 2009)

Übungsleitung: Dr. Georg Groh (grohg@in.tum.de)

Tutoren: Dipl. Inform. Vivian Prinz (prinzv@in.tum.de), Dr. Nils Kammenhuber (kammenhuber@net.in.tum.de), Dipl. Inform. Robert Schmohl (schmohl@in.tum.de), Dipl. Inform. Dipl. Geogr. Jan Herrmann (hermanj@in.tum.de), Dipl. Inform. Robert Eigner, David Brodski (brodski@in.tum.de), Yang Guo (yang.guo@gmx.de), Jan Finis (finis@in.tum.de), Martin Levihn (levihn@in.tum.de)

<http://www11.in.tum.de/Veranstaltungen/GrundlagenBetriebssystemeundSystemsoftware0809>

<http://www11.in.tum.de/Veranstaltungen/GrundlagenBetriebssystemeundSystemsoftware0809/uebung>

Blatt 5

- Abgabe: bis 24.11.2008 12:00 Uhr per E-Mail an den Tutor der eigenen Gruppe. Die Mail soll einen Zip-Ordner als attachment haben, der für jede Hausaufgabe einen Unterordner enthält, in dem die Lösung als .txt-File(s), als .c-File(s) o.ä. enthalten ist.
- Musterlösungen Hausaufgaben: ab 24.11.2008 12:00 Uhr auf der Übungswebseite zum Download.
- Musterlösungen Tutoraufgaben: ab 01.12.2008 12:00 Uhr auf der Übungswebseite zum Download.

Stoff

Es wird empfohlen folgende Literatur durchzuarbeiten:

- Skript Kapitel 3.4 (Thread Konzept) und 3.5 (Synchronisation)
- Tanenbaum "Modern Operating Systems" (Engl. Ausgabe von 2001) Kapitel 2.1 (Processes), 2.2 (Threads), 2.3 (Interprocess Communication) (Hier wird Synchronisation ausführlich behandelt) und 2.4 (Classical IPC Problems)

Das Kapitel 2 ist eines der empfehlenswertesten und wichtigsten Kapitel im Tanenbaum. Die Lektüre soll den Stoff aus dem Skript noch weiter vertiefen und enthält auch noch einige weitere Beispiele.

1 Hausaufgabe (Erzeuger - Verbraucher Problem)

Lernziele

Aktives und Passives Warten, Erzeuger - Verbraucher Problem: Lösung mit Semaphoren. Vertiefung des Umgangs mit C

Aufgabe

Sie haben in der Vorlesung das Erzeuger - Verbraucher Problem kennengelernt. Folgendes C-Codefragment zeigt einen Versuch, das Problem durch passives Warten zu lösen:

```
#define N 100                                /* Anzahl Plätze im Puffer */
int count = 0;                               /* Anzahl der Elemente im Puffer */

void producer(void)
{
    int item;

    while( TRUE ) {                          /* Endlosschleife */
        item = produce_item();               /* erzeuge nächstes Element */
        if( count == N ) sleep();           /* wenn Puffer voll, schlafe */
        insert_item(item);                  /* schreibe Element in Puffer */
        count = count + 1;                  /* erhöhe Anzahl der Elemente */
        if( count == 1 ) wakeup(consumer); /* war der Puffer leer? */
    }
}

void consumer(void)
{
    int item;

    while( TRUE ) {                          /* Endlosschleife */
        if( count == 0 ) sleep();           /* wenn Puffer leer, schlafe */
        item = remove_item();               /* hole Element aus Puffer */
        count = count - 1;                  /* erniedrige Anzahl der Elemente */
        if( count == N-1 ) wakeup(producer);/* war der Puffer voll? */
        consume_item(item);
    }
}
```

1.1 Teilaufgabe

Erklären Sie den Ablauf des Programms.

1.2 Teilaufgabe

Kann es bei dem Ablauf zu Problemen kommen? Zu welchen?

1.3 Teilaufgabe

Verändern Sie den Code so, dass die genannten Probleme nicht mehr auftreten können. Verwenden Sie dazu Semaphoren.

- Notieren Sie eine geeignete Definition eines Typs *semaphore*.
- Deklarieren Sie Semaphoren, die Sie zur Lösung der Probleme einsetzen wollen.
- Gehen Sie davon aus, dass Sie zur Belegung bzw. Freigabe auf unteilbare Funktionen *p* und *v* zugreifen können, denen Sie die Adresse des jeweiligen Semaphors übergeben.

2 Aufgabe (Modellierung mit Semaphoren) (Klausuraufgabe aus dem letzten Jahr)

Ein Möbelhändler hat eine Lagerhalle für Schränke und Couches. Beliefert wird er von einem Fabrikanten *F*, der bei jeder Lieferung einen Schrank und eine Couch bringt. Ein Ausfahrer *S* bringt je Fahrt einen Schrank zum Kunden, ein Ausfahrer *C* je Fahrt eine Couch. Zum Be- und Entladen muss eine Laderampe benutzt werden. Für sich alleine betrachtet laufen die beteiligten Prozesse folgendermaßen ab:

```
Process S          Process C          Process F
{                  {                  {
  while (TRUE)    while (TRUE)    while (TRUE)
  {              {              {
    <zur Rampe fahren>;    <zur Rampe fahren>;    <zur Rampe fahren>;
    <1 Schrank aufladen>;    <1 Couch aufladen>;    <1 Schrank entladen>;
    <Rampe verlassen>;    <Rampe verlassen>;    <1 Couch entladen>;
  }              }              <Rampe verlassen>;
}                }                }

```

Synchronisieren Sie diese drei Prozesse, indem Sie in Pseudocode-Notation in geeigneter Weise Semaphore deklarieren und Semaphor-Operationen einfügen.

Die Pseudocode-Operation zum Deklarieren eines Semaphors mit Namen *name* mit Startwert *n* lautet:

```
name(n);
```

Die Pseudocode-Operationen zum Aufrufen von *P* und *V* auf der Semaphore mit Namen *name* lauten:

```
P(name)
```

```
V(name)
```

Sperrphasen sind möglichst kurz zu halten und folgende Bedingungen müssen erfüllt sein:

- Zur Laderampe kann jeweils nur ein Prozess fahren.
- Der Abholer S darf nur zur Rampe fahren, wenn noch ein Schrank im Lager ist. Analoges gilt für den Abholer C.
- Die Lagerhalle hat beschränkte Kapazität, sie kann höchstens 20 Möbelstücke aufnehmen.
- Der Fabrikant darf nur zur Rampe fahren, wenn er seine Lieferung vollständig abladen kann.
- Zu Beginn sei das Lager leer und die Rampe frei.

2.1 Teilaufgabe

Listen Sie alle benötigten Semaphoren auf und erklären Sie ihren Zweck bzw ihre Pragmatik

2.2 Teilaufgabe

Fügen Sie in den /zu dem oben aufgelisteten Pseudocode des Ablaufs der beteiligten Prozesse an geeigneter Stelle entsprechende Aufrufe zum Deklarieren der Semaphoren und von P und V ein!

3 Hausaufgabe (Wechselseitiger Ausschluss)

Lernziele

Vertiefung des Verständnisses der Probleme in Bezug auf den wechselseitigen Ausschluss durch Veranschaulichung derselben. Vertiefung der Erfahrung mit C.

3.1 Teilaufgabe

Der wechselseitige Ausschluss muss nicht nur für nebenläufige Prozesse, sondern auch für nebenläufige Threads (eines Prozesses) gewährleistet werden. Es existieren analoge Probleme und Lösungen. Die in der Datei *mutual_exclusion.c* enthaltene Funktion *critical_region* implementiert eine angebliche Lösung des wechselseitigen Ausschlusses der kritischen Phasen zweier Threads. Erweitern Sie die Datei *mutual_exclusion.c* um eine main-Funktion, die zwei Threads erzeugt. Beide sollen die Funktion *critical_region* aufrufen und dabei die Indizes 0 beziehungsweise 1 übergeben. Verwenden Sie die *pthread* Bibliothek und modifizieren Sie gegebenenfalls die Funktion *critical_region*, so dass sie von den Threads aufgerufen werden kann.

3.2 Teilaufgabe

Betrachten Sie die angebliche Lösung des wechselseitigen Ausschlusses, die innerhalb der Funktion *critical_region* implementiert ist! Zeigen Sie durch Beschreibung eines Beispielablaufs, dass

es theoretisch eine Möglichkeit gibt, in der beide Threads in den kritischen Bereich eintreten können! Gehen Sie davon aus, dass beide Threads nebenläufig ausgeführt werden.

Tipp für "Ausprobierer": mit `sleep` können Sie Ausführungswechsel herbeiführen. Ein Ausführungswechsel an einer bestimmten Stelle (wo?) führt dazu, dass beide Prozesse in den kritischen Bereich eintreten können. Ein weiterer führt dazu, dass der Wert der globalen Variable `x` nicht korrekt berechnet wird.

3.3 Teilaufgabe

Durch eine Änderung der Anweisungsreihenfolge in der Funktion `critical_region` kann das Betreten des kritischen Bereichs wechselseitig ausgeschlossen werden. Modifizieren Sie die Datei `mutual_exclusion.c` entsprechend. Zu welchem Problem kann es jetzt kommen?

Tipp: Problem ist wie in Teilaufgabe 3.2 ausprobierbar.

Abgabe

Die modifizierte Version der Datei `mutual_exclusion.c` sowie eine `.doc`, `.pdf` oder `.txt`-Datei mit dem Beispielablauf und den Begriffs- und Problemerkklärungen.

4 Tutoraufgabe (Quizfragen Tanenbaum / Skript zum Thread-Konzept)

Lernziele

Erarbeitung/Wiederholung von Grundlagen des Thread-Konzepts und der Synchronisation von Threads/Prozessen.

4.1 Teilaufgabe

Erläutern Sie den Unterschied zwischen Thread und Prozess. Gehen Sie dabei auf deren Beziehung zueinander ein. Nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden (User-)Thread spezifisch sind und nennen Sie 2 Beispiele für Informationen bzw. Datenstruktur-Instanzen die für jeden Prozess spezifisch sind (die also (User-)Threads gemeinsam nutzen müssen)!

4.2 Teilaufgabe

Erläutern Sie kurz, warum der Aufwand für das Erzeugen eines (User-)Threads i.A. geringer ist als für das Erzeugen eines Prozesses!

4.3 Teilaufgabe

Nennen Sie vier Gründe für die Einführung von Threads.

4.4 Teilaufgabe

Wie können sich verschiedene, nebenläufige Threads gegenseitig negativ beeinflussen?

4.5 Teilaufgabe

Warum darf der spezielle Leerlauf-Thread niemals den Zustand "Warten" annehmen und wie kann dies erreicht werden?

4.6 Teilaufgabe

In einem Rechensystem konkurrieren parallele Aktivitäten um wiederholt exklusiv nutzbare Ressourcen, wie beispielsweise CPU oder Drucker. Zudem können parallele Aktivitäten auch kooperieren, indem sie Daten über gemeinsam benutzte exklusive Objekte austauschen oder sich Nachrichten zusenden. In all diesen Fällen haben wir das Problem, den wechselseitigen Ausschluss zu gewährleisten, d.h. sicherzustellen, dass nur höchstens ein Prozess zu einem gegebenen Zeitpunkt eine exklusiv benutzbare Ressource belegt.

Nennen Sie in diesem Zusammenhang die Bedeutungen folgender Begriffe: kritischer Bereich, race condition, aktives Warten, passives Warten.

4.7 Teilaufgabe

Nennen Sie 4 Anforderungen die eine gute Lösung zur Realisierung des wechselseitigen Ausschlusses zwischen Prozessen / Threads erfüllen muss!

4.8 Teilaufgabe

Diskutieren Sie die Vor und Nachteile folgender meist Busy Waiting basierter Strategien zum wechselseitigen Ausschluss:

- Interrupts deaktivieren
- Einfache Lock Variable ohne TSL
- Strict Alteration
- Peterson Lösung
- Lock Variable mit TSL

Ist Busy Waiting (bspw. Lock mit TSL) auch zur Verwirklichung des wechselseitigen Ausschlusses für Threads geeignet?

Abgabe

Die Antworten sollen gemeinsam in den Tutorübungen erarbeitet werden und sollen nicht abgegeben werden.

5 Tutoraufgabe (Definition Semaphore) (TA1: Klausurteilaufgabe aus dem vergangenen Jahr)

Lernziele

Erarbeitung/Wiederholung des wichtigen Konzepts Semaphor

5.1 Aufgabe

Die Funktionen P und V auf Semaphoren `s` (vereinfacht durch `int *` dargestellt) seien durch folgenden C-artigen Pseudocode informell charakterisiert (Es wird als gegeben angenommen, dass die Operationen P und V atomar sind und wechselseitig ausgeschlossen ausgeführt werden.):

```
public void P (int *s) {
    *s = expr1 ;
    if ( expr2 ) { Prozess in die Menge der bezüglich *s
        wartenden Prozesse einreihen }
}
```

```

public void V (int *s) {
    *s = expr3 ;
    if ( expr4 ) { Führe genau einen der bezüglich *s wartenden
        Prozesse in den Zustand Rechenwillig über }
    }

```

5.2 Teilaufgabe

Ersetzen Sie die Platzhalter *expr1*, *expr2*, *expr3* und *expr4* durch geeignete Ausdrücke!

5.3 Teilaufgabe

Erläutern Sie den Unterschied zwischen einer allgemeinen Semaphore und einem Mutex!

Abgabe

Die Antworten sollen gemeinsam in den Tutorübungen erarbeitet werden und sollen nicht abgegeben werden.

6 Tutoraufgabe (Semaphore + Erzeuger-Verbraucher-Problem)

Lernziele

Verdeutlichung des Einsatzes von Semaphoren und damit zusammenhängender Probleme. Vertiefung der Hausaufgabe. Wiederholung und Vertiefung der Erläuterungen im Skript

Aufgabe

Gegeben sei das aus der Vorlesung bekannte Erzeuger / Verbraucher-Problem mit dem Puffer W (Kapazität: n Elemente). Um wechselseitigen Ausschluss zu erreichen, sei folgender Lösungsversuch mit der Semaphore *wa* gegeben (Pseudocode):

Deklaration:

```
wa(1);
```

Erzeuger E:

```

while(true){
    produziere Element;
    wa.P();
    schreibe Element nach W;
    wa.V();
}

```

Verbraucher V:

```

while(true){
    wa.P();
    entnimm Element aus W falls Element vorhanden, sonst warte;
}

```

```
    wa.V();  
    verarbeite Element;  
}
```

6.1 Teilaufgabe

Welches Problem kann dabei auftreten?

6.2 Teilaufgabe

Geben Sie eine verbesserte Version an, in der keine Probleme mehr auftreten, indem Sie zwei weitere Semaphoren geeignet deklarieren und geeignete Aufrufe von P und V einfügen!

6.3 Teilaufgabe

Welche Probleme treten auf, wenn Sie in ihrer verbesserten Lösung die Reihenfolge der P-Operationen für wa, ihrer zusätzlichen Semaphore 1 und ihrer zusätzlichen Semaphore 2 permutieren?

Abgabe

Die Antworten sollen gemeinsam in den Tutorübungen erarbeitet werden und sollen nicht abgegeben werden.