Prof. Dr. Th. Letschert
FB MNI

21. April 2002

Übung 3 - Betriebssysteme I

Aufgabe 1

1. `open` und `sqrt` sind Bibliotheksfunktionen. Die eine verzweigt mit einem Systemaufruf ins Betriebssystem die andere nicht. Welche? Warum?

2. Was versteht man unter einem nebenläufigen Programm?

3. Kann ein Programm mit mehreren Threads auf einer Maschine mit nur einer CPU ausgeführt werden? Wenn nein: warum nicht, wenn ja: macht das Sinn? Wenn ja welchen?

4. Erläutern Sie welchen Vorteil eine Implementierung von Threads auf Sprachebene bringt. Erläutern Sie dazu, wie ein Thread als Sprach–Konstrukt (z.B. von C++) aussehen könnte und vergleichen Sie dessen Nutzung mit der PThread–Bibliothek.

5. Pipes verbinden verwandte Prozesse. Wieso ist das ein Indiz dafür, dass sie zur Realisation nebenläufiger Programme gedacht waren?

6. Wie könnte man experimentell herausfinden, wie groß der Pufferbereich ist, der zu einer Pipe gehört? Führen Sie ein entsprechendes Experiment durch!

7. Was passiert mit einem Prozess, der aus einer Pipe lesen will, die keine Daten enthält?

8. Was passiert mit einem Prozess, der aus einer Pipe lesen will, die auf der anderen Seite vom Schreiber–Prozess geschlossen wurde?

9. Wie sieht eine Pipe–Implementierung aus: Ist es ein Prozess, eine Datenstruktur, eine Funktion, eine Klasse, ein Objekt, ...?

10. Geben Sie ein Beispiel an für eine durch Konkurrenz und Kooperation notwendige Prozessysnchronisation.

11. Was ist ein *kritischer Abschnitt*? Geben Sie ein Beispiel an!

12. Was ist eine *atomare Aktion*? Erläutern Sie, warum kritische Abschnitte als Definition von Atomarität zu verstehen sind.

13. Erläutern Sie, warum die durch kritische Abschnitte definierte Atomarität nicht “absolut” sondern “relativ” ist.

14. Zwei nebenläufige Prozesse einer Anwendung müssen in ihren Aktionen synchronisiert werden. Der eine darf beispielsweise nicht auf eine Datenstruktur zugreifen, während sie von dem anderen manipuliert wird. Wo befindet sich der Code, der diese Synchronisation erzwingt? In einem in beiden Prozesse, in beiden Prozessen, an anderer Stelle im Anwendungscode, im Code des Betriebssystems, ...? Welcher Prozess führt den entsprechenden Code aus? Einer der beiden Anwendungsprozesse, beide Anwendungsprozesse, ein Synchronisationsprozess im Betriebssystem? Erläutern Sie!

15. Zwei als Prozesse aktive Anwendungen wollen gleichzeitig eine Hardware–Ressource nutzen, beispielsweise die Festplatte. Wo befindet sich der Code, der die notwendige Synchronisation erzwingt? Welcher Prozess führt den entsprechenden Code aus? Erläutern Sie!

16. Was versteht man unter *Bedingungssynchronisatiom*? Geben sie ein Beispiel an.

17. Was passiert, wenn zwei Prozesse gleichzeitig auf eine Speicherstelle zugreifen? Kann das überhaupt passieren?

18. Was versteht man unter *Verschränkung*? Warum und in wie weit kann man Parallelität als Verschränkung verstehen?

19. Warum heißt der Mutex "Mutex"? Wo könnte der Name herkommen?

20. Kritische Abschnitte werden in Java dadurch ausgedrückt, das man eine Funktion oder Methode als `synchronized` deklariert. Erläutern Sie wie dieses `synchronized` mit Mutexen realisiert werden kann.

21. Ist es denkbar, dass ein Prozess sich zu einem Zeitpunkt in zwei kritischen Bereichen aufhält? Geben Sie eventuell ein Beispiel an.

22. Was ist ein *Monitor*? Geben Sie ein Beispiel an.

23. Warum enthält die POSIX–Thread–Bibliothek kein Monitorkonstrukt?

24. Erläutern Sie das Prinzip der aktiven und der passiven Objekte. Warum ist es aus Gründen der Softwaretechnik sinnvoll, dass passive Objekte Synchronisationsmechanismen enthalten und nicht die aktiven Objekte?

25. Mutexe sind ein Mittel, um gegenseitigen Ausschluss zu realisieren. Sie benötigen aber selbst zu ihrer Implementierung gegenseitigen Ausschluss. Wie wird dieser Zirkel durchbrochen?

26. Auf einem Einprozessorsystem kann eine Mutex–Operation durch Abschalten der Unterbrechungen atomar gemacht werden. Das Abschalten der Unterbrechungen gilt allgemein als zu "radikal". Warum ist es zu radikal, was bedeutet hier "radikal"?

27. Jeder Bedingungsvariablen ist intern eine Warteschlange zugeordnet. Was ist der Inhalt dieser Warteschlange?

28. Warum hat die Warteoperation bei einer Bedingungsvariablen einen Mutex als Parameter?

29. Nehmen Sie an, eine gemeinsame Datenstruktur wird mit einem Mutex der PThread–Bibliothek vor gleichzeitigem Zugriff geschützt. Ein Thread hat den Mutex belegt. Gibt es für einen zweiten (von einem böswilligen oder unfähigen Programmierer definierten) Thread eine Möglichkeit trotzdem auf die Daten zuzugreifen?

30. Semaphore können mit Mutexen und Bedingungsvariablen emuliert werden. Definieren Sie eine Klasse `Semaphor` mit den Semaphoroperationen, die nur Mutexe und Bedingungsvariablen als Synchronisationsmittel benutzt.

Aufgabe 2

Ein Programm bestehe aus drei Prozessen P1, P2 und P3. Die Prozesse wiederum bestehen aus den atomaren Aktionen A, B, ... I:

```
P_1                P_2                P_3
do {               do {               do {
  A;                 D;                 G;
  B;                 E;                 H;
  C;                 F;                 I;
} while(1);        } while(1);        } while(1);
```

1. Was sind die möglichen Verschränkungen bei der Ausführung des Programms?
2. Angenommen A;B; und E;F; bilden einen kritischen Abschnitt. Welche Verschränkungen sind damit nicht erlaubt?
3. Geben Sie ein praktisches Beispiel für A;B; und E;F; an, das dazu führt, dass sie einen kritischen Abschnitt bilden.
4. Handelt es sich hier um eine Konkurrenzsituation oder um eine Kooperation der Prozesse.
5. Erläutern Sie wie und mit Hilfe welcher Synchronisationsmittel A;B; und E;F; als kritischer Abschnitt realisiert werden können.
6. Angenommen A;B; und E;F; sowie E;F; und G;H; bilden jeweils einen kritischen Abschnitt. Welche Verschränkungen sind damit nicht erlaubt?
7. Wie kann dies mit Hilfe welcher Synchronisationsmittel realisiert werden?

Aufgabe 3

1. Erstellen Sie ein Verzeichnis mit Umschlagklassen für POSIX–Mutexe, –Bedingungsvariabeln, Semaphore und Threads (H– und C–Dateien).
2. Schreiben Sie eine Make–Datei, die diese Klassen übersetzt und den erzeugten Code in eine statische Objektbibliothek einträgt.
3. Schreiben Sie zum Test Ihrer Definitionen drei nebenläufige Programme (mit Threads) die diese Bibliothek (und die entsprechenden Include–Dateien) benutzen:
   (a) Ein Programm zur parallelen Matrixmultiplikation, das zwei kleine Matrizen multipliziert,
   (b) ein Produzenten–Konsumenten–Programm mit beschränktem Puffer, der Semaphore benutzt und
   (c) eines, das einen mit Bedingungsvariablen implementierten beschränkten Puffer benutzt.
4. Schreiben Sie eine Make–Datei, die diese Testprogramme übersetzt und gegen Ihre Bibliothek mit der Thread–Klasse und den Synchronisationskonstrukten bindet. Testen Sie Ihre Programme.

Aufgabe 4

Schreiben Sie einen Monitor `Bruecke` zur Überwachung des Zutritts zu einer Brücke. Die Brücke kann nur ein maximales Gewicht `maxLast` tragen und Ihr Monitor hat die Aufgabe, darüber zu wachen, dass diese Traglast nicht überschritten wird. Ihr Monitor soll folgende Struktur haben:

```
class Bruecke {
public:
   Bruecke () { .....}
   void betrete (unsigned int last) { ... blockiert bis die Bruecke
                                          die Last tragen kann ... }
   void verlasse (unsigned int last) { ... }
private :
   static unsigned int maxLast; // maximale Belastung der Bruecke
.... weitere Komponenten nach Bedarf ...
};
```

Fahrzeuge werden durch Threads dargestellt. Ein Fahrzeug mit dem Gewicht `gewicht`, das eine Brücke `b` betreten will, ruft `b.betrete(gewicht);` auf. Beim Verlassen der Brücke wird `b.verlasse` aufgerufen. Sie können davon ausgehen, dass die Methoden korrekt verwendet werden. Benutzen Sie in Ihrer Lösung Bedingungsvariablen und Mutexe.

Aufgabe 5, Bonusaufgabe 3, Termin 3. Juni 2002

Das Problem des schlafenden Frisörs (*"The Sleeping Barber Problem"*) Ein Frisörladen besteht aus einem Frisörstuhl und einem Warteraum (unbeschränkter Kapazität). Im Laden arbeitet ein Frisör. Der Frisör bedient Kunden. Sind keine Kunden da, schläft der Frisör so lange bis einer eintrifft. Kunden betreten den Laden und werden bedient, oder warten, solange der Frisör beschäftigt ist, schlafend bis sie an der Reihe sind und der Frisör sie weckt. Die Bedienung eines Kunden beginnt damit, dass dieser im Stuhl Platz nimmt, der Frisör wartet mit der Bedienung solange, bis der Kunde sitzt. Dann beginnt die Bedienung und der Kunde wartet schlafend bis der Frisör fertig ist. Ist der Frisör fertig, zahlt der Kunde und entfernt sich. Der Frisör wartet, bis der Kunde den Laden verlassen hat und bedient den nächsten Kunden.

Modell Das Problem kann mit Prozessen/Threads und einem Monitor modelliert werden. Der Laden ist ein Monitor der Klasse `FrisoerSalon`, Kunden und Frisör sind Prozesse/Threads der Klassen `Frisoer` bzw. `Kunde`. Der Monitor übernimmt die Synchronisation des Frisörs und seiner Kunden. Frisör und Kunden interagieren nur über den Monitor. Ein Kunde, der Bedienung wünscht, ruft die Monitor–Methode `bedien`. Der Frisör ruft die beiden Methoden `naechsterKunde` und `fertig`. Mit `naechsterKunde` wird der nächste Kunden–Auftrag abgeholt und mit `fertig` wird signalisiert, dass dessen Bearbeitung beendet ist. (Für eine entsprechende Lösung des Problems mit Hilfe von Bedienungsvariablen siehe unten.)

Modellierte Problemstellung Das Problem des schlafenden Frisörs ist nicht etwa bekannt, weil Frisöre und ihre Kunden eine maschinelle Synchronisation nötig hätten, sondern weil es ein wichtiges Problem bei der Synchronisation eines Bedienprozesses (*Server*) und seiner Kunden (*Clients*) modelliert (*Client–Server Synchronisation*). Der Bedienprozess kann zu einer Zeit nur einen Kunden bedienen. Die Bedienung beginnt damit, dass der Kunde die Parameter seines Wunsches an den Monitor übergibt. Der Bedienprozess (Server) wartet bis der Kunde seine Auftragsdaten abgegeben hat, holt sie sich ab und beginnt mit der Bearbeitung seines Auftrags. Der Kunde wartet bis der Auftrag abgearbeitet ist und holt sich das Ergebnis ab. Der Bedienprozess wartet bis der Kunde sein Ergebnis abgeholt hat und bearbeitet dann den nächsten Auftrag. Insgesamt haben wir folgende Synchronisationspunkte:

- Kunden müssen warten bis sie dran sind.

- Der Bediener wartet bis die Kunden ihren Auftrag abgegeben haben. Im Modell Frisörsalon: Der Frisör wartet bis die Kunden im Sessel sitzen.
- Die Kunden warten bis die Bedienung abgeschlossen ist.
- Der Bediener wartet bis die Kunden ihr Ergebnis abgeholt haben. Im Modell Frisörsalon: Der Frisör wartet bis der fertig bediente Kunde den Sessel verlassen hat.

Aufgabenstellung Das durch das Problem des schlafenden Frisörs dargestellte Synchronisationsproblem entspricht der Synchronisation des Zugriffs auf eine Festplatte. Die Kunden sind die Prozesse, die lesend oder schreibend auf die Platte zugreifen. Die Platte wird durch eine Prozess verwaltet. Zwischen Kunden–Prozessen (Klasse `DiskUser` und dem Prozess Plattencontroller (Klasse `Disk`) synchronisiert ein Monitor (Klasse `DiskMonitor`) die Interaktion. Kunden rufen die Methode `DiskMonitor::useDisk(Zugriffsart, BlockNr, Blockadresse)`, wenn sie auf die Platte zugreifen wollen. Der Plattencontroller ruft `DiskMonitor::getNextRequest(Zugriffsart, ZylinderNr, SektorNr, Sektoradresse)` um den nächsten Auftrag abzuholen und `DiskMonitor::finishedTransfer(Blockadresse)` um das Ergebnis abzuliefern.

1. Realisieren Sie eine Lösung des schlafenden Frisörs.
2. Wenden Sie das Synchronisationsmuster des schlafenden Frisörs auf die Synchronisation eines Plattencontrollers und seiner Benutzer an. Modellieren Sie die Platte inklusive Controller als Thread der Klasse `Disk` und die Benutzerprozesse als Threads der Klasse `DiskUser`. Die Synchronisation erfolgt nach dem Muster des schlafenden Frisörs durch einen zwischengeschalteten Monitor `DiskMonitor`, der dem Frisörsalon entspricht.

Aufgabe 6, Bonusaufgabe 4 (Sonderbonus) , Termin 3. Juni 2002

Statt die Plattenzugriffe strikt sequentiell in der Reihenfolge der Anfragen abzuarbeiten, ist es besser sie so zu bearbeiten, dass der Weg des Lese–/Schreibkopfs minimiert wird. Ein übliches Verfahren ist der sogenannte Fahrstuhlalgorithmus (auch “SCAN–Algorithmus” genannt). Nach ihm werden erst alle Anfragen in einer Richtung abgearbeitet, dann wird die Richtung gewechselt und alle Anfragen in der anderen Richtung abgearbeitet. Erweitern Sie den `DiskMonitor` derart, dass er Zugriffe nach dem Fahrstuhlalgorithmus abarbeitet.

Hinweis Lassen Sie Prozesse mit einem neuen Auftrag in zwei Warteschlangen warten: eine für den Durchlauf in der aktuellen Richtung (der Auftrag liegt in der aktuelle Richtung) und eine für den Durchlauf in der nächsten Richtung (der Auftrag liegt hinter der aktuellen Position). Die Platte arbeitet die Warteschlange für die aktuelle Richtung ab und wechselt dann Richtung und Warteschlange.

Frisörsalon

```
// ------------------------------------------------------------
#include "../include/PThread.h"
#include "../include/Mutex.h"
#include "../include/Cond.h"

#include <unistd.h> // sleep
#include <stdlib.h> // rand

#include <iostream>
```

```
#include <string>

using namespace std;
using PThreadNS::Cond;
using PThreadNS::Mutex;
using PThreadNS::PThread;


class FrisoerSalon {
 public:

  FrisoerSalon () :
    fBereit(false), kSitzt(false), fFertig(false), kWeg(false),
    fBereitWS(m),   kSitztWS(m),   fFertigWS(m),   kWegWS(m)      {}


  // Aufruf durch Kunden der rasiert werden will
  //
  void bedien (string p_kName, string p_kAuftag, string & p_ergebnis) {
    m.lock();
    while ( !fBereit ) fBereitWS.wait();     // warte bis Frisoer frei
    fBereit = false;

    // Kunde setzt sich hin (Auftragsdaten uebergeben)
    l_kName    = p_kName;
    l_kAuftrag = p_kAuftag;
    kSitzt = true;
    kSitztWS.signal();

    while ( !fFertig ) fFertigWS.wait();     // Kunde wartet bis Frisoer fertig
    fFertig = false;

    // Kunde zahlt und geht (Ergebnis zurueck transferieren)
    p_ergebnis = l_kName + " : " + l_ergebnis;
    kWeg = true;                             // Kunde ist weg,
    kWegWS.signal();                         // Frisoer kann weitermachen
    m.unlock();
  }

  // Aufruf durch Frisoer: Der Naechste Bitte
  //
  void naechsterKunde (string & p_Name, string & p_kAuftag) {
    m.lock();
    fBereit = true;
    fBereitWS.signal();                      // der naechste bitte!
    while ( !kSitzt ) kSitztWS.wait();       // warte bis Kunde sitzt
    kSitzt = false;
    p_Name = l_kName;                       // Auftragsdaten holen
    p_kAuftag = l_kAuftrag;
    m.unlock();
  }

  // Aufruf durch Frisoer: Bin fertig, zahl und geh
  //
  void fertig (string p_ergebnis) {
    m.lock();
```

```
    l_ergebnis = p_ergebnis;
    fFertig = true;
    fFertigWS.signal();                    // Bin fertig: steh auf und bezahl!
    while ( !kWeg ) kWegWS.wait();         // warte bis Kunde weg ist
    kWeg = false;
    m.unlock();
  }

 private:
  bool        fBereit;     // Frisoer zum Bedienen bereit
  bool        kSitzt;      // Kunde sitzt noch unbedient auf dem Stuhl
  bool        fFertig;     // Frisoer hat Kunden fertig bedient
  bool        kWeg;        // Kunde hat gezahlt und ist gegangen

  string      l_kName;     // Name des Kunden auf dem Stuhl
  string      l_kAuftrag;  // Kundenauftrag
  string      l_ergebnis;  // Ergebnis der Frisoerarbeit

  Mutex       m;

  Cond        fBereitWS;   // WS der Kunden die auf Bedienung warten
  Cond        kSitztWS;    // WS in der der Frisoer wartet bis Kunde sitzt
  Cond        fFertigWS;   // WS der Kunden die auf das Ende der Bedienung warten
  Cond        kWegWS;      // WS in der der Frisoer darauf wartet, dass Kunde geht
};

// Die Klasse der Frisoere
//
class Frisoer : public  PThread {
 public:
  Frisoer (FrisoerSalon & p_fSalon) : fSalon(p_fSalon) {}

  void run () {
    string kName, kAuftrag;
    while ( true ) {
      cout << "Frisoer wartet auf Kundschaft" << endl;
      fSalon.naechsterKunde( kName, kAuftrag );
      // Bediene Kunden
      cout << "Frisoer beginnt mit Kunde " << kName << " (" << kAuftrag << ")" << endl
      int sleeptime = (3*rand())%5; sleep (sleeptime);
      cout << "Frisoer fertig mit Kunde " << kName << endl;
      // Kunde fertig bedient
      fSalon.fertig( kAuftrag + "_ist_erledigt" );
    }
  }
 private:
  FrisoerSalon &    fSalon; // mein Arbeitsplatz
};


// Die Klasse der Kunden
//
class Kunde : public PThread {
 public:
  Kunde (string p_name, string p_auftrag, FrisoerSalon & p_fSalon) :
name(p_name), auftrag(p_auftrag), fSalon(p_fSalon) {}
```

```
  void run () {
    string ergebnis;
    while ( true ) {
      int sleeptime = (name.length() * rand())%10; sleep (sleeptime);
      cout << "Kunde " << name << " will " << auftrag << endl;
      fSalon.bedien( name, auftrag, ergebnis );
      cout << "Kunde " << name << " fertig, Ergebnis: " << ergebnis << endl;
    }
  }

 private:
  string            name;    // mein Name
  string            auftrag; // mein Auftrag
  FrisoerSalon &    fSalon;  // mein Arbeitsplatz
};


FrisoerSalon leCoiffeur;

Kunde meier  ("Meier",   "Rasur",          leCoiffeur),
      schulze("Schulze", "Haarschnitt",    leCoiffeur),
      hinz   ("Hinz",    "Rasur",          leCoiffeur),
      kunz   ("Kunz",    "Glatzenpolitur", leCoiffeur);

Frisoer luigi(leCoiffeur);

int main () {
  luigi.start();
  meier.start();
  schulze.start();
  hinz.start();
  kunz.start();

  luigi.join();
}
```

Beispiel Disk

```
// Disk.h-------------------------------------------------------
#ifndef DISK_H_
#define DISK_H_

#include <PThread.h>

class DiskMonitor;

class Disk : public PThreadNS::PThread {
 public:
  static const unsigned short sectorSize    = 32;  // Sektorgroesse in Bytes
  static const unsigned short sectorCount   = 32;  // Zahl der Sektoren pro Zylinder
  static const unsigned short cylinderCount = 32;  // Zahl der Zylinder pro Disk

  typedef char Sector[sectorSize];
  enum  RequestType { read, write };                        // Zugriffsart
  enum  CylinderNr  { cylNull=0, cylMax = cylinderCount-1 }; // Typ der Zylindernummer
```

```
  enum  SectorNr    { secNull=0, secMax = sectorCount-1 };   // Typ der Sektornummern

  Disk ( DiskMonitor & p_diskMonitor ) :
diskMonitor ( p_diskMonitor ), headPos (static_cast<CylinderNr>(0)) {}

 private:
  void run ();

  typedef Sector  Cylinder[sectorCount];  // Zylinder: Feld von Sektoren
  Cylinder        disk[cylinderCount];    // Platte: Feld von Zylindern

  DiskMonitor &   diskMonitor; // Referenz auf zustaendigen Platten-Controller
  CylinderNr      headPos;     // Position des Lese-/Schreibkopfs

  void moveTo ( CylinderNr );
};

#endif

// Disk.cc-----------------------------------------------------
#include "Disk.h"
#include "DiskMonitor.h"
#include <unistd.h> // sleep
#include <PThread.h>
#include <iostream>

void Disk::run() {
  CylinderNr   cylinderNr;      SectorNr     sectorNr;
  RequestType  t;               Sector *     pSector;
  Sector       _sectorBuffer;
  while (true) {

    // Auftrag holen
    diskMonitor.getNextRequest(  t, cylinderNr, sectorNr, _sectorBuffer );

    moveTo (cylinderNr);
    switch ( t ) {
    case read:
      _sectorBuffer = disk[headPos][sectorNr];
      diskMonitor.finishedTransfer ( &_sectorBuffer );
      break;
    case write:
      disk[headPos][sectorNr] = _sectorBuffer;
      diskMonitor.finishedTransfer ( 0 );
      break;
    default: cerr << "DISK: Illegaler Zugriffstyp\n"; exit(-1);
    }
  }
}

void Disk::moveTo ( CylinderNr c ) {
  // Positionierungszeit in Sekunden
  int sleeptime = headPos > c ? headPos-c : c-headPos;
  sleep (sleeptime);
  headPos = c;
}
```

Beispiel DiskUser

```
// DiskUser.h-----------------------------------------------------
#ifndef DISK_USER_H_
#define DISK_USER_H_

#include <PThread.h>
#include "DiskMonitor.h"

class DiskUser : public PThreadNS::PThread {
 public:
  DiskUser (int, DiskMonitor &);
  void run ();

 private:
  int                       _id;
  DiskMonitor &             _diskMonitor;
};
#endif

// DiskUser.cc----------------------------------------------------
#include "DiskUser.h"

#include "Disk.h"
#include "DiskMonitor.h"
#include <unistd.h> // sleep
#include <PThread.h>
#include <iostream>

DiskUser::DiskUser (int p_id, DiskMonitor & p_diskMonitor ) :
  _id(p_id), _diskMonitor ( p_diskMonitor ) {}


void DiskUser::run() {
  DiskMonitor::Block block = "";
  int i = 0;
  while (true) {
    DiskMonitor::BlockNr nr = static_cast<DiskMonitor::BlockNr>(
        (i+1)*rand() % (Disk::sectorCount * Disk::cylinderCount)
        );
    int z = ((i+13)*rand())%4;
    block[0] = 'A' + ((i + _id) % 26);

    cout << "User " << _id << " START Schreib/Lese Zyklus "
         << " auf Block Nr. " << nr << " Schreibe: " << block[0] << endl;

    _diskMonitor.useDisk (Disk::write, nr, &block);
    sleep(z);
    _diskMonitor.useDisk (Disk::read, nr, &block);

    cout << "User " << _id << " ENDE Zugriff "
         << " auf Block Nr. " << nr << " Gelesen: " << block[0] << endl;
  }
}
```