Lehrstuhl für Informatik VI **16. Oktober 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney

1. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 2. Vorlesungswoche (23.-27.10.2000) in den Übungsgruppen

---

Allgemeine Hinweise

Zur Vorlesung gibt es eine Website, die immer die aktuellen Termine und ggf. wichtige Ankündigungen enthält. Dort werden auch die Übungszettel und sonstige Dokumente, die zur Lösung der Aufgaben benötigt werden, abgelegt.
Sie erreichen die Website unter: `http://www-i6.informatik.rwth-aachen.de/HTML/Lehre/SysPro/`
In den ersten Übungen werden Sie mit wichtigen Konzepten der Programmiersprache C vertraut gemacht. Es wird davon ausgegangen, daß Sie bereits grundlegende Kenntnisse vom Aufbau eines Programms in einer modularen Programmiersprache besitzen.
Die in den einzelnen Aufgaben vorgestellten Sprachkonzepte werden zunächst jeweils kurz erläutert. Zum tieferen Verständnis einzelner Funktionsaufrufe sollten Sie die UNIX Manual Pages (`man <topic>` am Kommandoprompt eingeben) oder die angegebene Literatur konsultieren.

Hinweise zu Programmieraufgaben

Im Rahmen der Übungen werden verschiedene Programmieraufgaben in der Programmiersprache C gestellt. Geben Sie bei solchen Aufgaben den Quelltext des Programms und ein Testprotokoll ab. Sie müssen den Quelltext ordentlich kommentieren und darauf achten, daß Ihr Quelltext strukturiert und modular aufgebaut ist. Die Verständlichkeit und Übersichtlichkeit des Quelltextes fliessen mit in die Bewertung der Aufgabe ein. Schicken Sie bei Programmieraufgaben die Ergebnisse per e-Mail an den Betreuer Ihrer Übungsgruppe (`syspro1..syspro6@i6.informatik.rwth-aachen.de`).
Sie können Programmieraufgaben sowohl im Rechnerpool der Informatik als auch bei Ihnen zu Hause bearbeiten. Im Rechnerpool sind zu folgenden Zeiten Rechner reserviert: 4U13 (blauer Raum) und 4U18 (roter Raum) reserviert: Dienstag 9-17 Uhr, Mittwoch 9-13 Uhr und Donnerstag 9-12 Uhr.

Übungen

Die Übungen sollten in Kleingruppen von zwei oder drei Studenten bearbeitet werden. Die Einteilung in Übungsgruppen erfolgt in der ersten Vorlesung. Falls Sie an dieser Vorlesung nicht teilnehmen konnten, aber dennoch an den Übungen teilnehmen möchten, melden Sie sich bitte per E-Mail (`syspro@i6.informatik.rwth-aachen.de`) bei uns. Wir werden dann – insofern noch Plätze frei sind – eine Verteilung durchführen.

Wie bekomme ich den Schein?

Den Schein zur Vorlesung bekommt man durch Erreichen von mindestens 50% der erreichbaren Übungspunkte (DPO 89: zusätzlich mündliche Prüfing). Die Aufgaben werden jeweils in der Vorlesung am Dienstag ausgegeben und müssen eine Woche später in der Übung abgegeben werden. In der darauf folgenden Woche werden die Aufgaben in der Übung am Mittwoch/Donnerstag in den Übungsgruppen vorgerechnet.

Aufgabe 1:

Machen Sie sich mit Unix vertraut. Alle Programmieraufgaben müssen normalerweise in C oder C++ und unter Unix entwickelt werden. Mögliche Informationsquellen hierzu sind:

- Skript des Rechnerbetriebs Informatik: *Einführung in Unix*
  `http://www.informatik.rwth-aachen.de/Pool/doku.verkauf.html`
- Handbuch des RRZN: *Unix - Eine Einführung in die Benutzung*
- das Unix-Kommando `man`

Insbesondere sollten Sie sich mit den folgenden Begriffen und Kommandos vertraut machen: Unix-Dateisystem, Shell, Pipes, Makefile, `ls`, `mkdir`, `rmdir`, `cd`, `pwd`, `mv`, `cp`, `rm`, `mail`, `emacs`, `gdb`, `gcc`, `g++`.

Aufgabe 2:

Frischen Sie Ihre C- bzw. C++-Kenntnisse auf. Mögliche Informationsquellen hierzu sind:

- Handbuch des RRZN: *Die Programmiersprache C. Ein Nachschlagewerk.*
- Brian W. Kernighan, Dennis M. Ritchie: The C Programming Language (Second Edition). Prentice Hall, Englewood Cliffs, 1988. (oder deutsche Übersetzung).
- Bjarne Stroustroup: The C++ Programming Language, 3rd Edition, Addison-Wesley, Reading, 1997. (oder deutsche Übersetzung).

Aufgabe 3 (4 Punkte):

Die erste Teilaufgabe befaßt sich mit den verschiedenen Zahlentypen.

Die Programmiersprache C bietet Zahlentypen mit unterschiedlicher Größe, Wertebereich und Präzision an, z.B.:

	Typ	Größe	Wertebereich	Beispiele
Ganzzahlig	(signed) char	8 Bit	-128 .. 127	39, -12, 'a'
	unsigned char	8 Bit	0 .. 255	
	(signed) short	16 Bit	-32768 .. 32767	
	unsigned short	16 Bit	0 .. 65535	
	(signed) int	32 Bit	-2147483648 .. 2147483647	
	unsigned int	32 Bit	0 .. 4294967295	
Fließkomma	float	32 Bit	-1×10^{38} .. $1 * 10^{38}$	-2.0, -1.3e-12
	double	64 Bit	-1.8×10^{308} .. $1.8 * 10^{308}$	

Default-Wert bei ganzzahligen Typen ist signed.

In C gibt es eine automatische (implizite) Typkonversion. Man kann also z.B. folgende Zuweisung machen:

```
int i=3;
float f=2.5;
...
f = i;
```

Implizite Typkonversionen stellen eine erhebliche Fehlerquelle dar. Man sollte daher die Konvertierung explizit durch sog. *casting* anweisen:

```
f = (float)i;
```

Zu beachten ist, daß innerhalb von arithmetischen Ausdrücken auf der rechten Zuweisungsseite ebenfalls Typkonversionen auftauchen können.

a) Übersetzen Sie den Sourcecode `u1_3.c` (verfügbar auf der Systemprogrammierungs-Website) und erzeugen Sie das Programm `u1_3`. Überprüfen Sie die Umrechnung von Grad Celsius in Fahrenheit nach der genäherten Formel

$$t[F] = 5/9 * t[C] + 32$$

und beseitigen Sie die fatalen Programmierfehler. Erklären Sie jeweils warum das Programm fehlerhaft ist.

b) Erweitern Sie das korrigierte Programm dahingehend, daß eine Celsius-Fahrenheit-Umrechnungstabelle ausgegeben wird. Es sollen die Temperaturen $-30, -20, -10, \ldots, 100$ Grad Celsius umgerechnet und tabellarisch ausgegeben werden.

Aufgabe 4 (6 Punkte):

Die zweite Aufgabe behandelt Felder. Felder sind Listen mit fester Länge, die Objekte eines bestimmten Typs beinhalten. So ist

```
int a[10];
```

ein Feld (Vektor, Liste, Array, ...) von 10 Elementen, die jeweils vom Typ `int` sind. Ein String ist ein Array mit Elementen vom Typ `char`. Der einzige Unterschied liegt darin, daß Strings mit dem Zeichen \0 enden (Strings sind *null-terminiert*). Ein String vom Typ

```
char s[10];
```

kann also neun Zeichen und das terminierende \0 aufnehmen. Strings können in der Variablendeklaration initialisiert werden, z.B.:

```
char s[10]="hallowelt";
```

Man kann direkt auf die Elemente eines Arrays zugreifen, indem man ihren Index angibt. Zum Beispiel:

```
a[2]=13;
```

Dabei ist zu beachten, daß das erste Element eines Feldes den Index 0 hat!

a) Schreiben Sie ein Programm, welches die Eingabe von bis zu 100 Zahlen erwartet und diese Zahlen dann sortiert auf dem Bildschirm ausgibt. Verwenden Sie das Sortierverfahren *Bubblesort*.

b) Analysieren Sie, ob folgende Arrayzugriffe korrekt sind. Geben Sie gegebenenfalls an auf welches Arrayelement zugegriffen wird.

```
int a[10],b[10];
a[10]=0;
a[2.5]=3;
a[1,2,3]=4;
a[(a-b)/2]=5;
```

Aufgabe 5 (5 Punkte):

Ziel der 3. Aufgabe ist es, daß Sie sich mit dem Gültigkeitsbereich von Variablen, der Parameterübergabe bei Funktionsaufrufen und dem Modulkonzept von C++ vertraut machen.

Grundsätzlich unterscheidet man zwischen globalen und lokalen Variablen. Globale Variablen sind im gesamten Modul gültig, in dem sie deklariert werden. Lokale Variablen können hingegen nur in der Funktion benutzt werden, in der sie deklariert wurden. Globale und lokale Variablen können gleiche Namen haben. Bei Namenskonflikten hat die lokale Variable Vorrang.

Die Parameterübergabe an Funktionen erfolgt nach dem *call-by-value*-Prinzip. Im Funktionskopf werden Variablen deklariert. Diese erhalten bei Ausführung jeweils den Wert zugewiesen, der beim Funktionsaufruf angegeben wurde. Die Variablen können dann zwar im Körper der Funktion modifiziert werden, jedoch wird der neue Wert nicht an das rufende Programm zurückgegeben. Die Parameterrückgabe muß entweder explizit über den Ergebniswert der Funktion (mittels return) oder impliziert über Speicheradressen erfolgen. Im zweiten Fall wird nicht der Wert einer Variable, sondern die Adresse ihrer Speicherstelle an die Funktion übergeben. Alle Änderungen, die an der adressierten Speicherstelle vorgenommen werden, sind dann automatisch auch in der rufenden Funktion sichtbar.

Unter Modularisierung versteht man die Unterteilung eines Programmes in mehrere Teile (Module). Diese werden dann zu einem Programm gebunden (gelinkt). Damit Funktionen eines Moduls in einem anderen bekannt sind müssen sogenannte Prototypen angegeben werden. Ein Funktionsprototyp beinahltet dabei die Information über Parameter und Rückgabewert der Funktion. Der Prototyp von `strcpy` sieht beispielsweise folgendermassen aus:

```
int strcmp(const char *s1, const char *s2);
```

Typischerweise faßt man die Prototypen für ein Modul in einer modulspezifischen Headerdatei zusammen.

a) Erzeugen Sie das Programm `u1_5a`. Starten Sie es und erklären Sie die Ausgaben. Wieso haben die Variablen `a` und `b` zu unterschiedlichen Zeitpunkten verschiedene Werte? Welchen Rückgabewert hat die Funktion `diff()`?

b) Schreiben Sie ein Programm, welches aus den drei Modulen `main.c`, `eingabe.c` und `ausgabe.c` besteht. Die Funktionalität des Programmes soll darin bestehen einen Text einzulesen (bis zu 1000 Zeichen) und in umgekehrter Reihenfolge wieder auszugeben. Aus dem Text `abc def` soll also `fed cba` erzeugt werden. Dabei soll im Modul `eingabe.c` die Eingabe und im Modul `ausgabe.c` die Ausgabe realisiert werden.

   Vermeiden Sie globale Variablen. Nutzen Sie Funktionsprototypen.

Aufgabe 6 (5 Punkte):

Die letzte Aufgabe soll Sie mit dem Dateikonzept unter C bekannt machen.

Es gibt verschiedene C-Funktionen, mit denen Sie auf Dateien zugreifen können. Zwei häufig verwendete Funktionen sind `fopen()` zum Öffnen und `fclose()` zum Schließen einer Datei. Nachdem eine Datei erfolgreich geöffnet worden ist, können Sie Daten aus der Datei lesen bzw. in die Datei schreiben. Das Dateihandle ist solange gültig, bis Sie die Datei wieder schließen.

a) Schreiben Sie ein Programm `split`, welches eine Datei in Gruppen von jeweils n Zeichen aufsplittet. Für jede n-Zeichen-Sequenz soll eine eigene Datei erstellt werden. Das Programm soll zwei Kommandozeilenparameter entgegennehmen: den Dateinamen der zu splittenden Datei und n.

b) Welche 3 Grundtypen gibt es, eine Datei mit `fopen()` zu öffnen? Worin unterscheiden sich diese Typen? Modifizieren Sie den Quelltext der vorigen Aufgabe derart, daß die Ausgabefiles (so vorhanden) nicht überschrieben wird, sondern daß die Daten an das Dateiende angehängt werden.

Lehrstuhl für Informatik VI **23. Oktober 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

2. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 3. Vorlesungswoche (2.11.2000) in den Übungsgruppen
Die Mittwochs-Übungsgruppen entfallen wegen Feiertag. Bitte geben Sie Ihre Lösungen am Donnerstag in den Übungen Ihrer Übungsgruppenleiter ab.

In der zweiten Übung zur Vorlesung Systemprogrammierung soll das Konzept der Pointer in C und die dynamische Speicherverwaltung im Mittelpunkt stehen.

Wichtig beim beim Umgang mit Pointern sind:

- Der Adress–Operator & mit dem auf die Speicheradresse einer Variablen zugegriffen wird.
- Der Inhalts–Operator * mit dem auf den Inhalt zugegriffen wird.

Beispiel:

```
int x = 1, y = 2;    /* x und y sind vom Typ integer   */
int *ip;             /* ip ist ein Pointer auf integer */
...
ip = &x;             /* ip zeigt nun auf die Adresse von x         */
y = *ip;             /* y wird der Inhalt von ip zugewiesen: 1     */
*ip = 0;             /* dem Inhalt von ip also x wird 0 zugewiesen */
```

Eine Array Variable ist auch ein Zeiger (auf das erste Element):

```
int number[100], *num_pointer;

num_pointer = &number[0];  /* Zeigt auf das erste Element.         */
num_pointer = number;      /* Dieser Ausdruck ist aequivalent dazu */
```

Im Gegensatz dazu währe `int *number[100]` ein Feld von 10 Zeigern auf integer-Zahlen.

Bei Arrays wird der zur Kompilezeit festgelegte Speicherplatz automatisch angefordert. Besser ist es jedoch nach tatsächlichem Bedarf Speicher zur Laufzeit anzufordern. Dazu dient die Funktion `malloc()`. Im Beispiel wird Speicher für N integer-Zahlen angefordert:

```
int *number;
...
number = malloc(N * sizeof(int));
```

Mit `sizeof(int)` wird hier sichergestellt, daß systemunabhängig die richtige Bytezahl angefordert wird. Datentypen können auf verschiedenen Rechnerarchitekturen unterschiedliche Größe haben, statt `sizeof(int)` genau 4 Byte anzufordern kann Fehler verursachen.

Da der Hauptspeicher eines Computer begrenzt ist, sollte man den reservierten Speicher mittels `free()` wieder freigeben, sobald er nicht mehr benötigt wird.

```
free(number);
```

Aufgabe 1 (10 Punkte):

Pointer sind gut zum Aufbau komplexer Datenstrukturen geeignet. Ein Beispiel dafür sind *lineare Listen*. Eine lineare Liste ist eine Menge von Objekten gleichen Typs, die über Zeiger linear miteinander verkettet sind. Im Beispiel wird eine Liste verwaltet, in der Studenten mit ihrem Namen und der Matrikelnummer erfaßt sind. Die Liste ist nach der Matrikelnummer aufsteigend sortiert.

Folgende Zugriffsfunktionen stehen für die lineare Liste zur Verfügung:

`is_empty()` testet, ob die Liste leer ist

`enqueue()` fügt einen neuen Studenten in die Liste ein

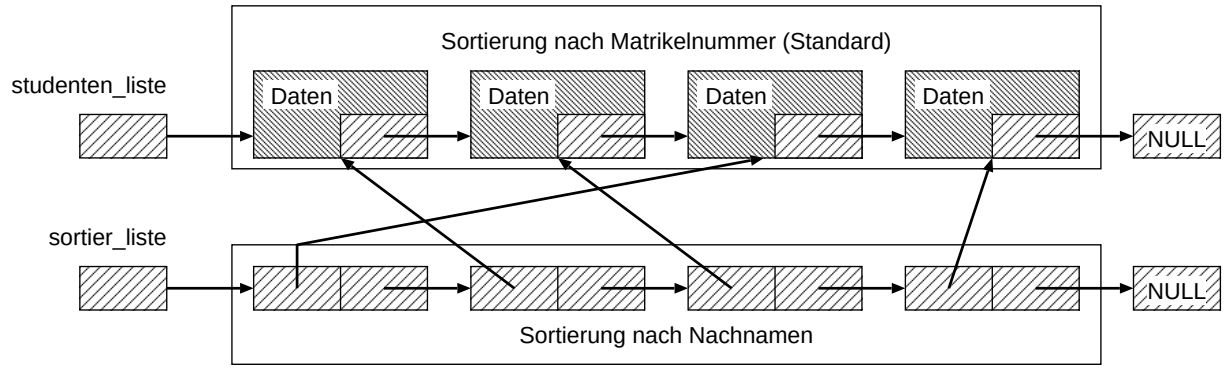
`dequeue()` löscht einen Studenten aus der Liste

`get_student()` liest die Daten zu einem Studenten aus der Liste

a) Laden Sie den Quelltext des Beispiels von der i6–Webseite herunter. Die Files liegen diesmal nicht einzeln, sondern in einem komprimierten Archiv `u2.tar.Z` vor. Nachdem Sie das Archiv heruntergeladen haben, können Sie es mit `uncompress u2.tar.Z` dekomprimieren und die einzelnen Files mit `tar -xvf u2.tar` entpacken. Ermitteln Sie die Funktion der einzelnen Optionen von `tar` in den UNIX Manual-Pages.
Studieren Sie den Quellcode, compilieren Sie das Programm und testen Sie es.

b) Welchen Vorteil hat die dynamische Erfassung der Studentendaten in einer linearen Liste gegenüber der statischen Erfassung in einem Array?

c) Es soll ermöglicht werden, die Liste statt der Matrikelnummer nach dem Nachnamen zu sortieren und auszugeben. Schreiben Sie hierzu eine Routine, die eine weitere lineare Liste `sortier_liste` erzeugt, wobei jedes Element dieser Liste keinen eigenen Eintrag des Datums enthält sondern lediglich auf die ursprünglichen Elemente der Datenbank verweist (siehe Skizze).



Was hat diese Form der Referenzierung für Vorteile gegenüber einer Kopie der Liste mit anderer Sortierung?

d) Implementieren Sie Routinen, um die im Speicher befindliche Datenbank permanent auf das Filesystemin eine Datei zu schreiben bzw. von dort in den Hauptspeicher zurückzuladen.

Aufgabe 2 (10 Punkte):

Mittels einfach verketteter Listen ist es möglich einen *Stack* (Stapel) zu implementieren. Neue Listenelemente werden immer "oben" auf dem Stapel abgelegt. Gelesen werden kann ebenfalls nur das oberste Element. Ein Stack arbeitet also nach dem LIFO-Prinzip (*last in, first out*).

a) Implementieren Sie einen generischen Stack mit den folgenden Funktionen:

`int is_empty():` testet, ob der Stack leer ist

`void push(void *elem):` legt ein Element auf den Stack ab

`void *pop();` entfernt das oberste Element vom Stack und gibt es zurück

Halten Sie sich an die vorgegebenen Funktionsdefinitionen. Der Typ `void *` wird verwendet, um einen allgemeinen Zeigertyp zu deklarieren.

Wenn `double`-Zahlen auf dem Stack gespeichert werden soll, dann soll `push` ein Pointer auf diese Zahl übergeben werden und `pop` liefert einen Pointer auf diese Zahl zurück.

b) Schreiben Sie ein Programm, welches mathematische Ausdrücke in Umgekehrt Polnischer Notation (UPN) bearbeiten kann.

Die UPN ist eine Postfixnotation von mathematischen Ausdrücken. Im Gegensatz zur herkömmlichen Schreibweise (Infixnotation) wird dabei der Operator (etwa "+") hinter seinen Operanden eingegeben (statt in die Mitte). Beispielsweise entspricht dem (Infix-)Ausdruck `(3+4)*5` der UPN-Ausdruck `3 4 + 5 *`.

Die Auswertung solcher Ausdrücke durch die Nutzung von Stacks ist sehr einfach zu programmieren. Der UPN-Ausdruck muß nur von links nach rechts abgearbeitet werden. Ein Operand wird auf den Stack g*epush*t. Eine 2-stellige Operation nimmt sich dann die obersten zwei Zahlen vom Stack, führt die Operation aus und schreibt das Ergebnis auf einen Stack. Am Ende liegt dann das Gesamtergebnis auf dem Stack und kann ausgegeben werden.

Berücksichtigen Sie bei Ihrer Implementierung die Operationen `+`, `-`, `*`, `/`. Nutzen Sie den generischen Stack aus Aufgabe 2b.

Lehrstuhl für Informatik VI **30. Oktober 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney

3. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 4. Vorlesungswoche (08.–09.11.2000) in den Übungsgruppen

---

Die dritte Übung soll die vorangehenden Übungen zum Thema Programmieren in C vertiefen. Mit der letzten Aufgabe sollen Sie an wichtige UNIX Tools herangeführt werden.

Aufgabe 1 (4 Punkte):

Die erste Aufgabe behandelt die Prioritäten von Operatoren und komplizierte Vereinbarungen in C.

a) In einem Kopierprogramm soll mit folgender while Schleife eine Tastatureingabe eingelesen und kopiert werden:

```
while ( c = getchar () != EOF )}
  putchar (c);
```

Warum zeigt dieser Ausdruck nicht das gewünschte Verhalten? Wie sollten Klammern gesetzt werden?

b) Warum ist der folgende Ausdruck problematisch?

```
printf ("%d %d\n",++n,power(2,n));
```

c) Beschreiben Sie in Worten was in welcher Reihenfolge in den Zeilen 1 bis 5 passiert.

```
struct {
  int len;
  char *str;
} *p

++p -> len;     /* 1 */
p++ -> len;     /* 2 */
*p -> str++;    /* 3 */
(*p -> str)++;  /* 4 */
*p++ -> str;    /* 5 */
```

d) Worin besteht der Unterschied zwischen `f` und `pf` bzw. zwischen `ap` und `pa`?

```
int *f();       /* 1 */
int (*pf)()     /* 2 */
int *ap[100];   /* 3 */
int (*pa)[100]; /* 4 */
```

e) Bonusaufgabe (2 Punkte): Was bedeutet der Ausdruck `float (*(*y[7])())[12];` in Worten? Begründen Sie Ihre Antwort!

Aufgabe 2 (10 Punkte):

Diese Aufgabe vertieft das in der letzten Übung vorgestellte Konzept von Pointern in C und befaßt sich mit der Rekursion.

Die rekursive Programmierung ist ein sehr mächtiges Programmierkonzept. Vor allem rekursiv formulierte Aufgabenstellungen und Datenstrukturen lassen sich damit sehr elegant und kompakt bearbeiten. Die Grundidee der rekursiven Programmierung besteht darin, daß sich eine Funktion selber aufruft.

Rekursive Funktionen haben i.A. folgenden Aufbau: Zunächst wird eine Abbruchbedingung getestet. Sollte diese Bedingung erfüllt sein, wird die Funktion verlassen. Die Abbruchbedingung ist besonders wichtig, weil ein rekursives Programm sonst unweigerlich in einer Endlosschleife landet und in kurzer Zeit wegen erschöpfter Ressourcen abbricht.
Dann werden bestimmte Berechnungen ausgeführt, wobei sich die rekursive Funktion zur Bestimmung von Teilergebnissen selbst aufruft.

In der letzten Übung war eine einfach verkettete lineare Liste von Studenten zu implementieren. Die Liste war nach der Matrikelnummer der Studenten sortiert. Wenn man in einer solchen Liste einen Studenten sucht, der bereits eingetragen ist, dann hat man eine lineare Komplexität: Im Mittel müssen bei n Studenten $\frac{n}{2}$ Listeneinträge geprüft werden, bis der Student gefunden ist. Ist er nicht in der Liste enthalten, müssen sogar alle n Einträge überprüft werden.

Der Suchaufwand wird deutlich kleiner, wenn man die Studenten nicht in einer linearen Liste sondern in einem binären Baum organisiert. Der Suchaufwand ist dann nur noch von der Ordnung $O(\log_2 n)$. Der Baum ist so sortiert, daß alle Sohnknoten links eines jeden Knotens kleiner (im Sinne des Sortierkriteriums) und alle Sohnknoten rechts größer sind.

Zur Umstrukturierung der Datenbank muß zunächst die Struktur `stud_type` modifiziert werden:

```
typedef struct tmp {                    /* Struktur des Datensatzes:    */
  int           matnum;                 /* Matrikelnummer, Vor-         */
  char          vorname[20];            /* und Nachname sind Eintraege. */
  char          nachname[20];           /* Die Datenbank ist eine       */
  struct tmp    *left,*right;           /* binaerer Baum.               */
  } stud_type;
```

Anstatt des Verweises auf einen Nachfolger gibt es nun Verweise auf einen linken und einen rechten Sohnknoten.

a) Implementieren Sie in Anlehnung an die Funktionen zur Verwaltung linearer Listen folgende Routinen:

   `int insert_node(stud_type **node, int matnum, char vorname[20], char nachname[20])`

   fügt einen neuen Knoten in den Baum ein,

   `int delete_node(stud_type **node, int matnum)`

   löscht einen Knoten aus dem Baum,

   `int get_student(stud_type node, int matnum, char vorname[20], char nachname[20])`

   liefert die zu einem Studenten gespeicherten Informationen.

   Die Implementierung der einzelnen Funktionen soll rekursiv erfolgen. Zurückgegeben wird, ob der Funktionsaufruf erfolgreich war.
   Die Suchfunktion `get_student` könnte z.B. wie folgt aussehen (Pseudocode):

```
int get_student(aktueller_Knoten) {
  if aktueller_Knoten==NULL                    /* Abbruchbedingung        */
    return Fehlermeldung('Student nicht gefunden');
  if aktueller_Knoten==gesuchter_Knoten        /* Knoten gefunden         */
    return Information(aktueller_Knoten);
  if aktueller_Knoten>gesuchter_Knoten         /* rekursiver Aufruf von   */
    return get_student(linker_Sohnknoten);     /* get_student mit linkem  */
  else                                         /* oder                    */
    return get_student(rechter_Sohnknoten);    /* rechtem Sohnknoten      */
}
```

Der Aufruf erfolgt mit Angabe des Wurzelknotens.

b) Es kann bewiesen werden, daß sich jeder rekursive Algorithmus in einen iterativen Algorithmus umwandeln läßt und umgekehrt. Belegen Sie das am praktischen Beispiel, indem Sie die Funktion `get_student` so umschreiben, daß Sie iterativ arbeitet.

c) Unter bestimmten Umständen ist der Baum, den Sie mit der Funktion `insert_node` erstellen, sehr ineffektiv ('entartet'). Die Suche nach einem Knoten kann dann wieder lineare Komplexität annehmen. Wodurch wird diese Entartung verursacht?

Schreiben Sie eine Funktion, die einen gegebenen binären Baum balanciert. Ein Baum gilt dann als balanciert, wenn für jeden Knoten im Baum gilt, daß sich die maximale Zahl der Nachfolgegenerationen im linken und im rechten Ast maximal um eins unterschiedet.

Hinweis: Sie können den Baum z.B. balancieren, indem Sie den ursprünglichen Baum in eine Liste umwandeln und dann einen neuen Baum generieren.

Aufgabe 3 (6 Punkte):

Machen sie sich mit den Möglichkeiten von UNIX vertraut. UNIX bietet eine Vielzahl kleiner Utility-Programme an, die zur Manipulation von Textfiles geeignet sind (z.B. `awk`, `grep`, `cut`, `paste`, `sort`, `uniq`, `diff`, `tr`, `wc`, `cat`, ...). Viele Aufgaben, zu denen man bei anderen Betriebssystemen komplette Programme schreiben müßte, lassen sich unter UNIX geschickte Verknüpfung dieser Tools über Pipes direkt auf der Kommandozeile einer Shell lösen.

Lösen Sie die Aufgaben unter Verwendung der genannten UNIX–Tools. Geben Sie neben der Lösung die verwendete Befehlsfolge an.

a) Auf der i6–Webseite finden Sie einen Ausschnitt aus 'In 80 Tagen um die Welt' (`80days.txt`). Laden Sie das File herunter. Entfernen Sie die Zeichen `" ? . ! : ; , + & ‘` ersetzen Sie die Zeichen Apostroph `’` und Bindestrich `-` durch Leerzeichen und entfernen Sie alle Leerzeilen.

b) Wieviele Zeilen und Wörter hat der Text nun? Wieviele Zeilen enthalten die Zeichenfolge “the” nicht (dabei nicht nach Groß– und Kleinschreibung unterscheiden) ?

c) Formatieren sie den Text so um, daß jedes Wort getrennt in einer neuen Zeilen steht. Ermitteln Sie, wieviel unterschiedliche Wörter der so vorverarbeitete Text enthält und wie häufig diese sind. Welches sind die 10 häufigsten Wörter?

d) Wie groß ist die durchschnittliche Wortlänge?

e) Wie häufig kommen die einzelnen Buchstaben vor (auch hier nicht nach Groß– und Kleinschreibung unterscheiden)? Ermitteln sie die häufigsten Buchstabenpaare und –tripel (ohne Leerzeichen) im Text.

Anmerkung: Solche Buchstabenstatistiken werden von Algorithmen benutzt die beim Eintippen von SMS–Nachrichten den wahrscheinlichsten Folgebuchstaben vorschlagen. In der automatischen Spracherkennung und –übersetzung werden Statistiken auf Wortebene sog. Sprachmodelle (“language models”) benutzt, um die Wahrscheinlichkeit von hypothetisierten Wortfolgen zu errechnen.

Lehrstuhl für Informatik VI **13. November 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney

4. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 6. Vorlesungswoche (22.–23.11.98) in den Übungsgruppen

---

Aufgabe 1 (3 Punkte):

Viele Zusammenhänge in Betriebssystemen lassen sich durch Schichtenmodelle darstellen. Die vertikalen Schichten sind hierarchisch angeordnet, d.h. Objekte auf einer höheren Schicht greifen auf die Objekte der darunterliegenden Schicht zu, welche wiederum auf die nächsttiefere Schicht zugreifen usw.

a) In einem Computersystem kann man eine grobe Einteilung in folgende Schichten vornehmen: `Betriebssystem`, `Anwendungsprogramme`, `Hardware` und `Nutzer`. Bringen Sie die Schichten in die richtige Reihenfolge und ordnen Sie folgende Begriffe den einzelnen Schichten zu:

   `CPU, Datenbanksystem, Studentin/Student, Scheduler, Transistor, Festplatte, Assembler, Texteditor, Lader, Professorin, Speichermanagement, Dateisystem, Graphikprogramm, Compiler, Speicher, Debugger, Makroprozessor, Bibliotheken, Linker, X-Windows, Textverarbeitungsprogramm, BIOS, Cache, CD-ROM, Sekretärin, Gerätemanagement, Interprozeßkommunikation, Netzwerkunterstützung, WWW-Browser`

b) Eine andere Unterteilung ist die Gliederung in `Hardware` und `Software`. Erklären Sie in einem Satz, welche Aufgabe aus dieser Sicht einem Betriebssystem zukommt.

Aufgabe 2 (4 Punkte):

In der Vorlesung haben Sie gelernt, daß man Prozesse entsprechend ihres Zustandes in verschiedene Klassen einteilen kann:

new: neuer Prozeß wird gestartet

running: Prozeß wird gerade vom Prozessor bearbeitet

ready: Prozeß ist bereit und wartet auf einen freien Prozessor

waiting: Prozeß wartet auf ein eigenes Betriebsmittel

blocked: Prozeß wartet auf ein Betriebsmittel, das von einem anderen Prozeß belegt wird

killed: Prozeß wird abgebrochen

terminated: Prozeß ist beendet

Gegeben seien 4 Prozesse, die zu unterschiedlichen Zeitpunkten gestartet werden und sich in die ready-Warteschlange einreihen. Der Scheduler arbeitet nach dem FCFS-Verfahren. Ein Teil der Prozesse fordert zwischendurch Daten von der Festplatte an. Das Lesen der Daten dauert 5 Zeiteinheiten, wobei der Prozessor jeweils freigegeben wird. Sobald der Prozess seine Daten erhalten hat reiht er sich wieder in die ready-Warteschlange ein. Der Prozeß P2 wird im 6. Arbeitsschritt nach seinem Start abgebrochen (ready und waiting Zeiten zählen nicht).

Prozeß	P1	P2	P3	P4
Startzeitpunkt	0	1	2	3
Dauer (incl. Lesen der Daten)	12	10	7	5
Beginn des Lesens nach n Arbeitsschritten	3	5	0	-

a) Geben Sie zu jedem Zeitpunkt den aktuellen Prozeßzustand der Prozesse P1 bis P4 an.

b) Zu welchen Zeitpunkten befindet sich das System im Zustand des "busy waiting", d.h. ein Prozeß wartet auf ein Betriebsmittel und blockiert derweil die CPU?

c) Nennen Sie ein Beispiel, bei dem der Abbruch eines Prozesses sinnvoll sein kann. Wie kann man unter den Betriebssystem UNIX Prozesse abbrechen?

Aufgabe 3 (7 Punkte):

In der Vorlesung wurden verschiedene Verfahren der Zuteilung von CPU-Zeit zu einzelnen Prozessen in einem Multitasking-Betriebssystem vorgestellt. Arbeitet der Scheduler ineffizient, wird die CPU nicht optimal ausgelastet oder einzelne Prozesse haben eine unverhältnismäßig lange Laufzeit.

In dieser Aufgabe geht es um das Round-Robin Scheduling-Verfahren. Folgende acht Prozesse seien gegeben:

Prozeßnr.	1	2	3	4	5	6	7	8
Rechenzeit	8	11	7	13	2	1	5	10

Der Scheduler soll zunächst ein Zeitquantum von Q = 5 verwalten. Die Umschaltzeiten zwischen den Prozessen werden vernachlässigt.

a) Wenden Sie das Round-Robin-Verfahren auf das obige Beispiel an. Zu welchem Zeitpunkt wird welcher Prozeß fertig?

b) Schreiben sie ein C–Programm das eine Tabelle erzeugt, in der für alle sinnvollen ganzzahligen Zeitquanten Q angegeben wird wann die einzelnen Prozesse (Pi) beendet wurden und wie groß die durchschnittliche Wartezeit war. Beispiel:

```
#    Q   P1    P2    P3    P4    P5    P6    P7    P8  Avg.Time
#-------------------------------------------------------------
     1   44    54    41    57    13     6    32    53    37.50
     .                                                      .
     .                                                      .
     .                                                      .
    13    8    19    26    39    41    42    47    57    34.88
```

Hinweise: Nutzen sie zur Verwaltung der Warteschlange eine lineare Liste wie in der 2. Übung. Die Laufzeiten der einzelnen Prozesse sollten dem Programm als integer Werte auf der Kommandozeile übergeben werden können: `round_robin t1 t2 ... tn` (mit beliebigem n)

c) Sollte das Zeitquantum in unserem Beispiel vergrößert oder verkleinert werden? Diskutieren Sie Vor- und Nachteile die sich ergeben, wenn man das Zeitquantum erhöht oder erniedrigt. Experimentieren Sie mit anderen Beispielwerten für die Rechenzeiten. Für welche Anwendungsgebiete eignen sich längere (kürzere) Zeitquanten besser?

Aufgabe 4 (6 Punkte):

Sie haben in der Vorlesung die nicht-preemptiven Scheduling Strategien FCFS, LCFS und SJF kennengelernt. In der Vorlesung wurden die Strategien jeweils für eine CPU vorgestellt. Sie sind aber leicht so erweiterbar, daß sie ankommende Prozesse auf mehreren CPUs verteilen können.

Betrachten wir als Beispiel die Kassen eines Supermarktes. Jede Kasse entspricht einer CPU und jeder Kunde, der an die Kasse kommt, stellt einen Prozeß dar. Die Zahl der Artikel, die ein Kunde eingekauft hat entspricht der Zeit, die ein Prozeß zur Ausführung benötigt. Wir nehmen dabei an, daß die Zeit zum Registrieren eines Artikels an der Kasse konstant ist und setzen diese mit einer Zeiteinheit an, den eigentlichen Bezahlvorgang vernachlässigen wir.

Da es Kunden im Supermarkt nicht zugemutet werden kann, andere Kunden vorzulassen, kommt üblicherweise die FCFS-Strategie zum Einsatz, d.h. wer zuerst die Kasse erreicht, wird zuerst abkassiert. Da die Kunden eines Supermarktes auf ihren Vorteil bedacht sind, stellen sie sich immer an der Kasse an, an der die Wartezeit am geringsten ist, d.h. an der vor ihnen die wenigsten Artikel registriert werden müssen.

a) Ein Supermarkt habe drei Kassen (K1, K2 und K3). An keiner der Kassen wartet ein Kunde. Nun treffen (fast gleichzeitig) Kunden mit folgenden Artikelanzahlen (in dieser Reihenfolge) ein:

   6, 15, 23, 10, 3, 13, 15, 7, 20, 40, 19, 4, 6, 21 (14 Kunden)

   O.B.d.A. stellt sich der erste Kunde an K1, der zweite an K2, der dritte an K3 an.

   - Dokumentieren Sie die sich ergebenden Warteschlangen an den Kassen K1, K2 und K3.
   - Wieviele Zeiteinheiten muß ein Kunde im Durchschnitt warten, bis er an der Kasse ankommt?

b) Der Leiter des Supermarktes ist mit dieser mittleren Wartezeit für seine Kunden nicht zufrieden und führt eine Kasse ein, an der nur Kunden bedient werden, welche weniger oder gleich 11 Artikel eingekauft haben (K1 wird in diesem Sinn ausgezeichnet). Kunden mit wenigen Artikeln dürfen sich natürlich nach wie vor auch an den anderen Kassen anstellen, sofern sie eine geringere Wartezeit erwarten.

   - Dokumentieren Sie die sich ergebenden Warteschlangen (K1 ist die Kasse für Kunden mit weniger als 12 Artikeln; die ersten Kunden stellen sich an die Kassen K1, K2, K3 - in dieser Reihenfolge).
   - Berechnen Sie die mittlere Wartezeit der Kunden.
   - Was für eine Strategie versucht der Supermarktleiter näherungsweise zu erreichen?
   - Was für einen Nachteil hat die neue Regelung?

c) Der Supermarktleiter ist nicht zufrieden mit der unter b) eingeführten Lösung. Er entscheidet, die Kasse K1 nur noch für Kunden mit 20 oder mehr Artikeln zu erlauben.

   - Dokumentieren Sie die sich ergebenden Warteschlangen (K1 ist die Kasse für Kunden mit mehr als 19 Artikeln; die ersten Kunden stellen sich an die Kassen K2, K3, K1 - in dieser Reihenfolge).
   - Berechnen Sie die mittlere Wartezeit der Kunden.
   - Wird der Supermarktleiter bei dieser Lösung bleiben?

Lehrstuhl für Informatik VI **20. November 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

5. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 7. Vorlesungswoche (29.11./30.11.2000) in den Übungsgruppen

---

Aufgabe 1 (5 Punkte):

Gegeben sei folgendes Scheduling mit vier Prioritätsklassen. Jeder Klasse ist eine eigene Warteschlange und ein eigenes Schedulingverfahren zugeordnet:

Klasse	Schedulingverfahren	Priorität
0	RR mit Quantum 1 Zeiteinheit	höchste
1	RR mit Quantum 4 Zeiteinheiten	
2	RR mit Quantum 16 Zeiteinheiten	
3	FCFS	niedrigste

(RR = Round Robin; FCFS = First Come First Served)

Es wird *Process Aging* verwendet, d.h. wenn das Zeitquantum eines Prozesses abgelaufen ist, wird er an das Ende der Warteschlange mit nächstniedriger Priorität gestellt. Neuen Prozessen ist eine bestimmte Priorität zugeordnet. Sie werden an das Ende der Warteschlange ihrer Prioritätsklasse angehängt. Es wird am Ende jeder Zeiteinheit überprüft welcher Prozess am Anfang der nicht leeren Warteschlange mit der höchsten Priorität steht und dieser dann im nächsten Schritt bearbeitet.
Folgende Prozesse sollen betrachtet werden:

Prozeß	Ankunftszeit	Priorität	Laufzeit
A	0	0	4
B	0	1	7
C	1	0	1
D	2	1	4
E	5	3	20
F	10	1	3
G	13	0	2
H	15	1	3
I	16	1	3

Ein Prozeß, der zu einem Zeitpunkt t ankommt, wird in der $(t+1)$-ten Zeiteinheit berücksichtigt.

a) Geben Sie für die ersten 20 Zeiteinheiten an, welchem Prozeß Rechenzeit zugeteilt wird.

b) Geben Sie für alle Prioritätsklassen an, welche Prozesse sich nach Ablauf der ersten 20 Zeiteinheiten noch in der Warteschlange befinden (in der korrekten Reihenfolge), wieviele Zeiteinheiten jeder einzelne Prozeß noch laufen muß und wieviele Zeiteinheiten von seinem Quantum noch übrig sind.

Aufgabe 2 (4 Punkte):

Beim Probebetrieb eines Computernetzwerkes kommt es gelegentlich zu Übertragungsfehlern. Dabei kann es auftreten, daß die Anzahl gesendeter Pakete und die Anzahl empfangener Pakete pro Sekunde voneinander abweicht. In einem mathematischen Modell werden diese Anzahlen durch einen Zufallsvektor (X, Y) beschrieben:

X: Anzahl gesendeter Pakete (pro Sekunde)
Y: Anzahl empfangener Pakete (pro Sekunde)

Aufgrund bisheriger Analysen konnte die gemeinsame Verteilungsfunktion von (X, Y) bestimmt werden. Jedoch traten bei deren Übermittlung ebenfalls Fehler auf, so daß lediglich folgende Einträge bekannt sind:

X=x \ Y=y	1	2	3	4	$P(X = x)$
1	2/20	?	1/20	0	5/20
2	?	3/20	?	1/20	7/20
3	2/20	1/20	?	?	?
4	?	2/20	1/20	?	4/20
$P(Y = y)$	4/20	?	5/20	?	1

a) Vervollständigen Sie die Tabelle.

b) Berechnen Sie Erwartungswert und Varianz der Zufallsvariablen X und Y.

c) Bestimmen Sie die Verteilungsfunktion der Zufallsvariablen $Z := X - Y$ und berechnen Sie Erwartungswert und Varianz dieser Zufallsvariablen.

Aufgabe 3 (7 Punkte):

Die Poisson-Verteilung (mit Parameter $\lambda = n \cdot p$)

$$P(k) = \frac{\lambda^k}{k!} e^{-\lambda}$$

kann als Approximation der Binomial-Verteilung (mit Parametern n und p)

$$P(k) = \left(\begin{array}{c} n \\ k \end{array} \right) p^k (1-p)^{n-k}$$

angesehen werden wenn n groß und p klein ist.

a) Die Wahrscheinlichkeit, daß ein bestimmtes elektronisches Bauteil einer Fertigung defekt ist betrage $p = 0.02$. Berechnen Sie die Wahrscheinlichkeit dafür, daß unter $n = 100$ zufällig ausgewählten Bauteilen $k = 0, 1, 2$ oder 3 Bauteile defekt sind. Berechnen Sie das Ergebnis einmal mit der Poisson-Verteilung und einmal mit der Binomialverteilung.

b) Berechnen Sie die Wahrscheinlichkeit, daß mehr als 3 Bauteile defekt sind.

c) Vergleichen Sie nun Theorie und Praxis indem Sie eine Simulation durchführen:
Das Programm soll 10^6 Bauteile mit einer Fehlerwahrscheinlichkeit von $p = 0.02$ "hergestellen." Entnehmen Sie aus dieser Menge N mal eine Stichprobe von jeweils 100 zufällig ausgewählten Teilen. Geben sie für $N = 10, 100$ und 1000 an mit welcher relativen Häfigkeit sie $k = 0, \ldots, 10$ fehlerhafte Teile in ihren Stichproben gefunden haben. Vergleichen Sie diese Werte mit der Poisson–Verteilung.

Aufgabe 4 (4 Punkte):

Berechnen Sie einen allgemeinen Ausdruck für den Erwartungswert und die Varianz einer Poisson-verteilten Zufallsvariablen mit Parameter $\lambda = n \cdot p > 0$.

Lehrstuhl für Informatik VI **28. November 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

6. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 8. Vorlesungswoche (12.12./13.12.2000) in den Übungsgruppen

---

Aufgabe 1 (4 Punkte):

Ein Rechner bedient n Terminals. Jedes Terminal schickt Aufträge an den Rechner ab. Die Zeiten zwischen aufeinanderfolgenden Aufträgen pro Terminal seien exponentialverteilt mit dem Mittelwert 30 sec. Der Bedienzeitbedarf der Aufträge ist exponentialverteilt mit einem Mittelwert von 200 msec.

a) Wie steigen Warteschlangenlänge und Verweilzeit der Aufträge in Abhängigkeit von der Zahl der Terminals?

b) Die mittlere Verweilzeit soll 3 Sekunden nicht übersteigen. Wieviele Terminals können unter diesen Voraussetzungen maximal angeschlossen werden?

c) Wie groß ist die Auslastung des Systems unter diesen Annahmen?

Aufgabe 2 (4 Punkte):

Ein Bankautomat bedient pro Stunde 120 Kunden. Die mittlere Verweilzeit (Warte- und Bedienzeit) beträgt 3 Minuten. Wieviele Kunden können maximal pro Stunde bedient werden (unter M/M/1-Annahmen)? Wieviele Personen befinden sich im Mittel am Bankautomaten?

Aufgabe 3 (6 Punkte):

Ein Postamt besitze mehrere Schalter. Alle Kunden stellen sich in einer Schlange an. Sobald ein Schalter frei wird geht der erste wartende Kunde an diesen Schalter. Die Ankunftsrate sei exponentialverteilt und betrage 2 Kunden pro Minute. Die durchschnittliche Bedienzeit (an einem Schalter) betrage 3 Minuten.

a) Klassifizieren Sie das System in Kendall-Notation. Erstellen Sie ein Zustandsdiagramm.

b) Bestimmen Sie die Zahl der Schalter, die benötigt wird, damit das System einen stabilen Zustand erreichen kann.

Aufgabe 4 (6 Punkte):

Gegeben sei ein erweitertes M/M/1-Warteschlangensystem. Im Gegensatz zu einem normalen M/M/1-System werden hier jedoch Kunden mit der Wahrscheinlichkeit $1/(i+1)$ angenommen, wobei i die Anzahl der im System befindlichen Prozesse darstellen soll, und mit einer Wahrscheinlichkeit von $i/(i+1)$ abgeweisen, ohne abgearbeitet zu werden.

a) Berechnen Sie die Wahrscheinlichkeit, daß sich i Kunden im System befinden. Interessanterweise ergibt sich hierbei eine Verteilung, die Sie kennen. Welche ist das?

b) In einem normalen M/M/1-System ist es erforderlich, daß gilt: $0 \leq \frac{\lambda}{\mu} < 1$. Wie verhält es sich in dem erweiterten M/M/1-System?

c) Berechnen Sie den Erwartungswert der Zahl der Kunden im System und der Zahl der wartenden Kunden.

Lehrstuhl für Informatik VI **4. Dezember 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

7. Übung zur Vorlesung "Systemprogrammierung"

Wichtiger Hinweis:
Die Aufgaben der 6. Übung müssen erst am 13./14. 12. 2000 abgegeben werden!
Diese Woche (6./7.) werden in den Übungsgruppen die folgenden Aufgaben durchgerechnet.

Aufgabe 1:

Der Begriff "Ergodizität" spielt im Zusammenhang mit stochastischen Prozessen eine wichtige Rolle.

a) Wann bezeichnet man ein System als ergodisch? Beschreiben sie das anschaulich was die verschiedenen Mittelungen, die in diesem Zusammenhang auftreten, bedeuten.

b) Kann ein nicht stationäres System ergodisch sein? Begründen Sie ihre Antwort.

Aufgabe 2:

Gegeben sei ein Warteschlangensystem bei dem ein Server (mit unbegrenzter Warteschlangen–Kapazität) Jobs bearbeitet. Die Ankunftszeiten der Jobs seien Poisson verteilt. Die Bedienzeiten seien exponentialverteilt.

a) Geben Sie in Kendall–Notation an um was für ein System es sich handelt.

b) Wie sieht das Zustandsdiagramm dieses Systems aus?

c) Formulieren sie die Bilanzgleichungen und beschreiben sie in Worten was die einzelnen Terme anschaulich bedeuten. Kennen Sie Beispiele für ähnliche Bilanzgleichungen in anderen Bereichen (E–Technik, Physik)?

d) Berechnen Sie die Auslastung des Systems. Wann gibt es ein stationäres Gleichgewicht?

e) Berechnen sie die Erwartungswerte für: Die Zahl der Jobs im System, die Zahl der Jobs in der Warteschlange, die Wartezeit in der Queue, die Verweilzeit.

Aufgabe 3:

Beim System aus Aufgabe 2 werde nun die Kapazität der Warteschlange auf k begrenzt.

a) Geben Sie in Kendall–Notation an um was für ein System es sich handelt.

b) Wie sieht das Zustandsdiagramm dieses Systems aus?

c) Berechnen Sie die Auslastung des Systems. Wann kann es ein stationäres Gleichgewicht geben?

d) Mit welcher Rate werden Kunden abgewiesen?

d) Wie groß ist der Durchsatz des Systems?

Lehrstuhl für Informatik VI **11. Dezember 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

8. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 9. Vorlesungswoche (20.12./21.12.2000) am Lehrstuhl

Aufgabe 1 (7 Punkte):

Im Call-Center des Weihnachtsmannes, Sektion Nordrhein-Westfalen, sitzen fünf eifrige Wichtel, die sich ein Telefon teilen müssen. Die Wichtel haben die Aufgabe Kinder anzurufen und ihre Weihnachtswunschzettel zu erfragen. Ein Anruf dauert im Mittel 10 Minuten. Nach jedem Anruf machen die Wichtel eine Zigarettenpause, deren Länge als exponentialverteilt anzusehen ist. Im Mittel versucht ein Wichtel an einem 8-Stunden-Arbeitstag 8 Telefongespräche zu führen. Wenn ein Wichtel anrufen möchte und in diesem Moment das Telefon besetzt ist, dann wartet er solange bis das Telefon frei wird. Man kann davon ausgehen, daß die Gesprächsdauer exponentialverteilt ist.

Nach einigen Wochen beschweren sich die Wichtel beim Weihnachtsmann, daß ein Telefon in ihrem Büro zu wenig ist. Sie sollen nun herausfinden, ob die Arbeitsbedingungen wirklich unzumutbar sind.

a) Um was für ein Bediensystem (Kendall-Notation) handelt es sich?

b) Wie hoch ist die Wahrscheinlichkeit, daß ein Wichtel zum Telefon greifen will, welches jedoch gerade von einem Kollegen benutzt wird?

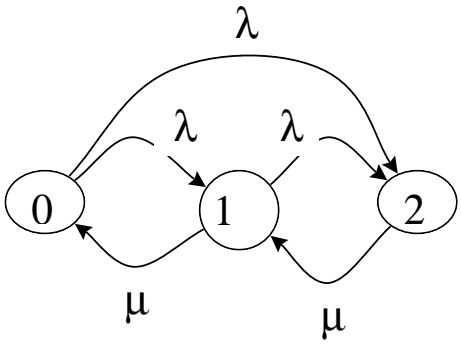
c) Berechnen Sie, wie lange ein Wichtel während seiner Arbeitszeit raucht.

d) Wie hoch ist die Wahrscheinlichkeit, daß das Telefon eine Stunde lang unbenutzt ist?

e) Der Weihnachtsmann erklärt sich bereit ein weiteres Telefon zu installieren. Wie sieht nun die Wahrscheinlichkeit aus, daß ein Wichtel telefonieren will, aber beide Telefone besetzt sind?

Aufgabe 2 (5 Punkte):

Gegeben sei ein Markoffprozess mit folgendem Zustandsdiagramm:



a) Berechnen Sie die stationäre Verteilung des Markoffprozesses in Abhängigkeit von $\rho = \lambda/\mu$.

b) Für welche Werte von λ und μ ergibt sich eine gleichverteilte stationäre Verteilung (also alle 3 Zustände haben gleiche Zustandswahrscheinlichkeit).

Aufgabe 3 (4 + 1 Punkte):

Schreiben Sie ein Programm zur Simulation eines M/M/1 Systems (vgl. Übung 7 Aufgabe 2). Es soll:

a) zu jedem Zeitpunkt die Zahl der Jobs im System ausgeben

b) und am Ende die gemessenen Werte $E\{N\}$ und $Var\{N\}$ mit den theoretischen vergleichen.

c) Führen Sie verschiedene Testläufe mit unterschiedlichen Simulationsdauern, Ankunfts– und Bedienraten durch.

Hinweis: Achen Sie darauf, daß die Ankunfts– und Bedienraten im Verhältniss zu den diskreten Zeitschritten Ihres Programms geeignet gewählt werden.
Bonuspunkt: Führen Sie die Berechnung des Mittelwerts und der Varianz in einer einzigen Schleife über die Zeit durch.

Lehrstuhl für Informatik VI **18. Dezember 2000**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

9. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 10. Vorlesungswoche (10.01./11.01.2001) in den Übungsgruppen

---

Aufgabe 1 (5 Punkte):

Sollen verschiedene Prozesse in einem parallelen Prozeßsystem miteinander kooperieren, muß die Synchronisation der beteiligten Partner sichergestellt werden. Ein einfaches Beispiel dafür haben Sie in Form des *Erzeuger-Verbraucher-Problems* bereits kennengelernt.

a) Nennen Sie 3 praktische Beispiele für das Erzeuger-Verbraucher-Problem in einem Rechnersystem.

b) Als eine einfache Lösung zur Synchronisation eines Erzeugers und eines Verbrauchers wurde der Ringpuffer eingeführt, der über die Variablen `in` und `out` gesteuert wird. Erklären Sie, wieso beim Übergang auf diese zwei Variablen keine Konsistenzprobleme mehr auftreten können. Gilt das auch, wenn mehr als ein Erzeuger oder Verbraucher im System aktiv sind? Begründen Sie Ihre Aussage.

c) Gegeben sei ein Datenbanksystem mit einem Puffer für eingehende Anfragen, der Platz für 100 Nachrichten hat. Sowohl die Klienten als auch der Datenbankserver sind Poisson-Prozesse: Im Mittel werden 5 Anfragen pro Sekunde in den Puffer gestellt und 10 Anfragen pro Sekunde vom Server abgearbeitet. Wie oft kommt es nach Neustart des Datenbankservers und seines Puffers im Mittel vor, daß ein Klient seine Anfrage wiederholen muß, weil der Puffer gerade voll ist?

Aufgabe 2 (5 Punkte):

Im Zentrum der in der Systemprogrammierung behandelten Probleme steht der Prozeß. Sie haben bereits kennengelernt, nach welchen Verfahren den einzelnen Prozessen CPU-Zeit zugeteilt werden kann. Derzeit wird die Synchronisation von nebenläufigen Prozessen behandelt. Die Programme, die Sie bisher in den Übungen geschrieben haben, beinhalteten jedoch alle nur genau einen Prozeß.

In dieser Aufgabe steht daher die Generierung neuer Prozesse im Mittelpunkt. Das Prozeßsystem unter UNIX gleicht in gewissem Sinne dem Filesystem: Prozesse bilden eine hierarchische Baumstruktur, an deren Wurzel der `root`-Prozeß steht, der zusammen mit dem UNIX-System gestartet wird. Jeder Prozeß kann mehrere Kindprozesse erzeugen, jeder Kindprozeß hat hingegen genau einen Vaterprozeß.

Die Identifikation der Prozesse erfolgt über eine Nummer, die Prozeß-ID (PID). Welche Prozesse gerade laufen und welche PID sie haben, können Sie sich dem Kommando `ps` (process status) ermitteln.

Die Erzeugung eines neuen Prozesses geschieht unter UNIX mit dem Systemruf `fork()`. Das Programm startet dabei einen zweiten Prozeß (einen Child-Prozeß). Der Sohnprozeß erhält eine Kopie der Systemumgebung des Vaterprozesses, d.h. er arbeitet auch auf demselben Programmcode. Die Unterscheidung zwischen Vater und Sohn erfolgt über den Rückgabewert von `fork()`: Während der Vater die PID seines Sohnes zurückgegeben bekommt, ist der Rückgabewert für den Sohn Null. Wenn also Vater- und Sohnprozeß unterschiedliche Aufgaben ausführen sollen, kann man im Quelltext anhand des Rückgabewertes in die entsprechenden Funktionen verzweigen. Benötigt ein Prozeß seine eigene Prozeßnummer, kann er sie vom System mit `getpid()` erfragen.

Mit der Funktion `wait()` kann der Vaterprozeß so lange schlafen gelegt werden, bis der Sohnprozeß terminiert. Ebenfalls in Zusammenhang mit `fork()` wird häufig der Systemruf `exec()` verwendet, mit dem der Kindprozeß ein neues Programmfile lädt und ausführt.

a) Auf der i6 Webseite finden Sie das Programm `u9_a2.c`. Laden Sie sich das Programm herunter, compilieren und starten Sie es. Das Programm beinhaltet eine Endlosschleife. Bestimmen Sie die PID der beiden Prozesse, die gestartet wurden. Informieren Sie sich über die Aufgabe und Parameter des UNIX-Kommandos `kill` und beenden Sie damit die Programmausführung.

b) Modifizieren Sie das Programm derart, daß zwei Sohnprozesse gestartet werden. Der zweite Sohnprozeß soll eine eigene Funktion erhalten, die (analog zu den bestehenden) das Zeichen 'C' ausgibt.

c) Schreiben Sie ein Programm, welches Zeichen von der Standardeingabe (`stdin`) liest und die Häufigkeit der einzelnen Buchstaben protokolliert. Wenden Sie es auf den von Ihnen modifizierten Buchstabengenerator an und überprüfen Sie, ob alle Prozesse etwa dieselbe Prozessorzeit erhalten.

d) Finden Sie heraus, mit welchem Kommando man unter UNIX die Prozeßpriorität verändern kann. Vermindern Sie die Priorität des Prozesses, der 'B' ausgibt, um die Hälfte des maximal möglichen Wertes. Wie verändert sich das Verhältnis zwischen den Häufigkeiten der ausgegebenen Buchstaben?

Aufgabe 3 (3 Punkte):

Der wechselseitige Ausschluß ist ein Grundproblem bei der Synchronisation nebenläufiger Prozesse. Es muß sichergestellt werden, daß ein Prozeß in seinem kritischen Bereich nicht von anderen Prozessen unterbrochen werden kann, weil sonst das Ergebnis der Operation nicht mehr determiniert ist.

Das Problem des wechselseitigen Auschlusses ist genau dann korrekt gelöst, wenn die drei Bedingungen

- mutual exclusion
- progress requirement und
- bounded waiting

erfüllt sind.

In der Vorlesung haben Sie zum Algorithmus von Petersen drei Ansätze kennengelert, die nicht funktionieren:

1.
```
P0: repeat                              P1: repeat
        while (turn!=0) do noop;                while (turn!=1) do noop;
        critical_section(P0);                   critical_section(P1);
        turn:=1;                                turn:=0;
        remainder_section(P0);                  remainder_section(P1);
    until FALSE;                            until FALSE;
```

2.
```
P0: flag[0]:=FALSE;                     P1: flag[1]:=FALSE;
    repeat                                  repeat
        while (flag[1]) do noop;                while (flag[0]) do noop;
        flag[0]:=TRUE;                          flag[1]:=TRUE;
        critical_section(P0);                   critical_section(P1);
        flag[0]:=FALSE;                         flag[1]:=FALSE;
        remainder_section(P0);                  remainder_section(P1);
    until FALSE;                            until FALSE;
```

3.
```
P0: flag[0]:=FALSE;                     P1: flag[1]:=FALSE;
    repeat                                  repeat
        flag[0]:=TRUE;                          flag[1]:=TRUE;
        while (flag[1]) do noop;                while (flag[0]) do noop;
        critical_section(P0);                   critical_section(P1);
        flag[0]:=FALSE;                         flag[1]:=FALSE;
        remainder_section(P0);                  remainder_section(P1);
    until FALSE;                            until FALSE;
```

a) Zeigen Sie für jedes der 3 Verfahren an einem konkreten Beispiel, daß eine der Bedingungen für den wechselseitigen Ausschluß verletzt wird.

Aufgabe 4 (5 Punkte):

Betrachten wir nun ein Beispiel aus der Praxis: Eine Landstraße verbindet die Orte A und B, auf halbem Weg befindet sich eine einspurige Brücke.

a) Sie sollen eine Verkehrsregelung für die Brücke schaffen und ziehen folgende Lösungen in Betracht:

   - first come, first served: Sie regulieren den Verkehr überhaupt nicht sondern gehen davon aus, daß derjenige zuerst fährt, der zuerst ankommt.
   - Sie sperren die Brücke wegen Baufälligkeit.
   - Sie bauen an jeder Seite eine Ampel auf, die im Minutentakt umschaltet.
   - Sie legen eine Vorrangrichtung fest und stellen die entsprechenden Verkehrsschilder auf.

   Der Einfachheit halber wird angenommen, daß die Gelbphase der Ampel so lange dauert, daß ein Auto die gerade befahrene Brücke noch überqueren kann, bevor der Gegenverkehr kommt. Welche der vier Lösungen realisieren den wechselseitigen Ausschluß? Beachten Sie dabei, daß ein Konflikt nur dann auftritt, wenn zwei Autos aus entgegengesetzten Richtungen die Brücke gleichzeitig befahren. Begründen Sie Ihre Aussagen.

b) Sie haben sich für die Ampeln entschieden. Ihnen gefällt jedoch nicht, daß sie auch bei geringem Verkehr stur nach dem festen Zeittakt geschaltet sind und die Autos oftmals vor der leeren Brücke warten müssen. Sie rüsten die Ampeln daher mit Sensoren für ankommende Fahrzeuge aus. Geben Sie einen Algorithmus (Pseudocode) für die Ampelschaltung mit Sensoren an, der einen wechselseitigen Ausschluß garantiert. Er sollte sowohl für wenig als auch für viel Verkehr gut funktionieren (guter Durchsatz bei viel Verkehr, geringe Wartezeit bei wenig Verkehr). Die oben gemachte Annahme zu der langen Gelbphase ist weiterhin gültig.

c) Leider stellt sich heraus, daß die Sensoren störanfällig und damit unbrauchbar sind, so daß sie doch mit einem festen Zeittakt arbeiten müssen. Nun wollen Sie die Schaltung wenigstens so einstellen, daß die mittlere Wartezeit für alle Kraftfahrer minimal wird. Im Laufe der Voruntersuchung haben Sie festgestellt, daß vormittags doppelt so viele Autos von A nach B fahren als umgekehrt. Am Nachmittag kehrt sich der Trend um. In welchem Verhältnis sollten die Zeittakte der Grünphasen für die Richtungen A→B und B→A vormittags und nachmittags stehen?

Frohe Weihnachten und einen guten Rutsch ins neue Jahr!

Lehrstuhl für Informatik VI **8. Januar 2001**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

10. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 11. Vorlesungswoche (17.01./18.01.2001) in den Übungsgruppen

---

Aufgabe 1 (4 Punkte)

Der Bäckerei-Algorithmus ist als Lösung des wechselseitigen Ausschlußproblems für n Prozesse bekannt. Der Algorithmus für einen Prozeß P_i ist durch folgendes Programstück im Pseudo-Code gegeben:

```
repeat
   choosing[i] := true;
   number[i] := max(number[0,...,n-1])+1;
   choosing[i] := false;
   for j:=0 to n-1 do {
      while (choosing[j]) do noop;
      while (number[j]!=0 and
             ((number[j]<number[i]) or
              (number[j]=number[i] and j<i))) do noop;
   }
   critical_section(Pi);
   number[i] := 0;
   remainder_section(Pi);
until false;
```

a) Wozu sind die Felder `choosing` und `number` vorhanden?

b) Zeigen Sie: Wenn P_i im kritischen Bereich ist und für P_k $(k \neq i)$ der Wert `number[`k`]` $\neq 0$ ist, dann gilt `number[`i`]` $<$ `number[`k`]` oder `number[`i`]` $=$ `number[`k`]` und $i < k$.

c) Sind die Bedingungen *Bounded Waiting* und *Progress* beim Bäckerei-Algorithmus erfüllt? Begründen Sie Ihre Antwort und geben Sie eine obere Schranke für das *Bounded Waiting* an!

Aufgabe 2 (5 Punkte)

Als Leiter der EDV-Abteilung von Apag haben Sie die Aufgabe, ein Konzept zur Verwaltung der in einem Parkhaus zur Verfügung stehenden Parkplätze zu erstellen. Um niemand in das Parkhaus einzulassen, für den kein Parkplatz mehr vorhanden ist, wollen Sie sich über das geniale Konzept der Semaphoren informieren.
Ihre Recherchen ergeben, daß eine Semaphore in etwa einer gekapselten Integer-Variablen entspricht, auf die Sie mittels der drei Operationen `init`, `wait` und `signal` zugreifen können.

a) Recherchieren Sie weiter! Stellen Sie dar, was mit den drei Operationen auf der Integer-Variablen gemacht wird und inwiefern Semaphoren für unser Problem geeignet sind.

b) Werden Sie konkret: Geben Sie Algorithmen an (Pseudocode), mit denen ein Parkhaus mit 423 Plätzen verwaltet wird. Das Parkhaus wird morgens geöffnet und abends, nachdem das letzte Auto weggefahren ist, wieder geschlossen.

c) Momentan fahren Autos, für die kein Parkplatz vorhanden ist, so lange um den Block, bis ein Parkplatz frei wird. Um das zu verhindern schaffen Sie einen Wartebereich vor dem Eingang. Erweitern Sie Ihre Algorithmen dementsprechend.

d) Der Wartebereich kann höchstens 13 Autos aufnehmen. Weitere Fahrzeuge bilden einen Rückstau auf der Straße. Installieren Sie eine Rot-Grün-Ampel, die den Wartebereich vor Überlauf schützt. Realisieren Sie dieses Konzept in ihren Algorithmen. Geben Sie zusätzlich eine Ampelsteuerung an.

Aufgabe 3 (6 Punkte):

In dieser Aufgabe sollen Sie das Erzeuger/Verbraucher-Problem einmal praktisch unter C/UNIX bearbeiten. Auf der Vorlesungs-Homepage finden Sie dazu das Programm `ProdCons.c`. Es handelt sich um eine Implementierung des Erzeuger/Verbraucher-Problems mit zwei Prozessen. Die Kommunikation der Prozesse läuft über einen gemeinsam genutzten Speicherbereich (*shared memory*), welcher vom Betriebssystem angefordert wird. Der Verbraucherprozeß gibt die Anzahl der verbrauchten Elemente aus, so daß leicht zu überprüfen ist, ob alles richtig funktioniert.

a) So ein Pech! Beim Kopieren des Quelltextes auf den Server ist leider der Code zum Lesen und Schreiben von Elementen verlorengegangen. Implementieren Sie diese Funktionen neu!

b) Der Verbraucherprozess wird mit `fork()` erzeugt, er kennt also zu diesem Zeitpunkt dieselben Variablen wie der Erzeugerprozess. Wieso wurde im Programm die komplizierte Kommunikation über shared memory gewählt? Kann der gemeinsam genutzte Ringpuffer nicht normal mit `malloc()` angefordert werden? Begründen Sie Ihre Aussage.

c) Erzeugen Sie innerhalb des Programms einen weiteren Verbraucherprozeß. Rufen Sie für diesen die Verbraucherfunktion mit dem Parameter 2 auf. Funktioniert das Programm noch einwandfrei?

d) Seien Sie ein Speicherfetischist und führen Sie wie in der Vorlesung skizziert die Variable `count` ein, so daß wirklich alle `BUFSIZE` Elemente des Ringpuffers genutzt werden können.

Aufgabe 4 (6 Punkte):

In dieser Aufgabe sollen verschiedene Kommunikationsverfahren zwischen Prozessen praktisch unter UNIX implementiert werden. Für die direkte Kommunikation zwischen Prozessen können *Pipes* genutzt werden, zur indirekten Kommunikation gibt es das *Message*-Konzept aus dem IPC-Paket.

Pipes haben Sie bereits bei der Verknüpfung von UNIX-Kommandos am Systemprompt der Shell kennengelernt. Dabei wird die Standardausgabe des einen mit der Standardeingabe des nächsten Prozesses verknüpft, so daß die Prozesse praktisch 'in Reihe' verschaltet sind. So ähnlich funktionieren auch Pipes, die Sie in C benutzen können: Ein Prozeß schreibt Daten in die Pipe hinein, ein anderer liest die Daten am anderen Ende der Pipe aus. Zum Lesen und Schreiben einer Pipe werden dieselben Systemaufrufe `read()` und `write()` wie für Dateien verwendet.

In UNIX unterscheidet man zwischen benannten und unbenannten Pipes. Eine unbenannte Pipe wird durch den Systemruf `pipe()` erzeugt. Sie kann nur durch verwandte Prozesse (also den Prozeß, der den `pipe()`-Systemruf ausgeführt hat, sowie seine Nachkommen) benutzt werden. Benannte Pipes hingegen basieren auf einer permanenten Struktur im Filesystem (ähnlich einer Datei), so daß beliebige Prozesse miteinander kommunizieren können.

Den Messages liegt ein höhersprachliches Programmierkonzept zugrunde. Zunächst richtet sich ein Prozeß mit `msgget()` eine Nachrichtenqueue ein bzw. öffnet die Verbindung zu einer bereits bestehenden Queue. Die Parameter dieser Queue (z.B. Zugriffsrechte) können mit `msgctl()` eingestellt werden. Mit `msgrcv()` empfängt man Nachrichten aus einer Queue ('Mailbox') und mit `msgsnd()` kann man Nachrichten verschicken.

a) Auf der Vorlesungs-Homepage finden Sie das Programm `u10_1.c`. In ihm kommunizieren ein Erzeuger und ein Verbraucher über eine unbenannte Pipe. Der Erzeuger liest Eingaben von der Tastatur, der Verbraucher gibt sie auf dem Bildschirm aus. Studieren Sie den Quelltext, übersetzen Sie das Programm und testen Sie es.

   Schreiben Sie das Programm so um, daß der Erzeuger- und Verbraucherprozeß nicht mehr notwendigerweise verwandt sein müssen. Das Programm soll dazu als Kommandozeilenparameter seine Funktion (Erzeuger oder Verbraucher) und den Namen der Pipe übergeben bekommen. Testen Sie das Programm, indem Sie die Eingabe eines Terminals (`xterm`) auf einem anderen Terminal ausgeben.

   Hinweis: Das Programm `u10_3.c` demonstriert Ihnen die Parameterübergabe an ein C-Programm.

b) Können sich Erzeuger und Verbraucher auf zwei verschiedenen Netzwerkrechnern befinden, die dasselbe Filesystem nutzen? Warum (nicht)?

c) Wie kann erreicht werden, daß die Leseoperation aus der Pipe nichtblockierend ist? Modifizieren Sie den Verbraucher derart, daß er nach einer Leerlaufzeit von fünf Sekunden anmahnt, daß Sie nicht an der Tastatur einschlafen sollen.

d) Ebenfalls auf dem Server finden Sie das Programm `u10_2.c`. Hier werden die Tastatureingaben mittels indirekter Kommunikation weitergegeben. Programmieren Sie den zugehörigen Verbraucher, der die ankommenden Nachrichten auf dem Terminal ausgibt.

e) In der aktuellen Implementierung bleibt die Nachrichtenqueue auch dann bestehen, wenn alle Erzeuger und Verbraucher terminiert sind. Damit werden unnötig Ressourcen belegt, die nur begrenzt im System vorhanden sind. Informieren Sie sich über die Parameter des Systemrufs `msgctl()` und sorgen Sie dafür, daß die Queue nach Beendigung der Programme freigegeben wird.

Lehrstuhl für Informatik VI **15. Januar 2001**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

11. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 12. Vorlesungswoche (24.01./25.01.2001) in den Übungsgruppen

---

Aufgabe 1 (4 Punkte):

Eine Variante einfacher, hardwaremäßig nicht teilbarer Operationen sind Decrement-and-Test (`dectest(x)`) und Test-and-Increment (`testinc(x)`). Beide Operationen eignen sich für Variablen, auf die verschiedene nebenläufige Prozesse konkurrierend zugreifen wollen.

```
dectest(x)
  atomar {
    x--;
    if(x>0)
      return 1;
    else {
      if (x<0)
        return -1;
      else
        return 0;
    }
  }
```

```
testinc(x)
  atomar {
    if (x>0)
      return 1;
    else {
      if (x<0)
        return -1;
     else
        return 0;
    }
    x++;
  }
```

a) Korrigieren Sie den Fehler in `testinc(x)`.

b) Unternehmen Sie einen Lösungsversuch für den wechselseitigen Ausschluß mit Hilfe von `dectest(x)` und `testinc(x)`. Stellen Sie an Beispielen dar, in welchen Situationen Probleme auftreten könnten.

c) Realisieren Sie die Funktionen `signal()` und `wait()` von Semaphoren (mit Warteschlangen) mit Hilfe dieser unteilbaren Operationen.

Aufgabe 3 (4 Punkte):

Im Reitstall des Pony–Express eine Grippewelle ausgebrochen und hat alle verfügbaren Pferde bis auf eines (namens Rosinante) außer Gefecht gesetzt. Um Streit unter den m Boten zu verhindern, wurde die zuständige EDV-Abteilung eingeschaltet.

Die EDV-Abteilung entscheidet sich für folgendes Vorgehen: Rosinante bekommt einen eigenen Stall mit einem Vorraum, in dem n Stühle stehen. Wenn kein Bote da ist, um mit Rosinante auszureiten, so legt sie sich (verständlicherweise) schlafen. Ein Bote, der seine Lieferung antreten will, betritt den Vorraum und schaut sich nach einem leeren Stuhl um. Findet er einen, so nimmt er auf diesem Platz und wartet, bis er an der Reihe ist, reitet mit Rosinante los und kommt erst nach Erledigung aller seiner Termine zurück (d.h. jeder Bote führt nur einen Ritt durch). Findet der Bote keinen freien Stuhl, so verläßt er den Vorraum wieder und geht zu Fuß.

Dieser Vorschlag muß nun noch formalisiert werden. Die EDV-Abteilung hat in der Kürze der Zeit leider nur den Algorithmus für Rosinante erstellt, welcher auf Semaphoren mit assoziierten Warteschlangen beruht:

```
repeat
        begin
        signal(horsefree);
        wait(wakeup);
        Ausritt;
        end;
until false;
```

a) Wie lautet der zugehörige Algorithmus für die Boten? Wie sind die Semaphoren zu initialisieren?

b) Lösen Sie das Problem durch Verwendung bedingter kritischer Regionen!

c) Unter welchem Namen ist das Problem in der Literatur bekannt?

Aufgabe 3 (5 + 1 Punkte):

Die folgende Aufgabe ist (in unausgeschmückter Form `:-)` unter dem Namen *5-Philosophen-Problem (Dining-Philosophers-Problem)* bekannt.

Fünf großartige Philosophen treffen sich, um ihren wichtigsten Tätigkeiten nachzugehen: Essen und denken. Da Philosophen etwas durcheinander sind, vergißt jeder eines seiner zwei Eß-Stäbchen zu Hause. Die Küche bringt zwar jedem Philosophen einen ständig vollen Teller Reis, davon satt wird ein Philosoph aber nur, wenn er zwei Stäbchen zur Verfügung hat. So einigen sich die Philosophen darauf, daß jeder sein Stäbchen auf seine rechte Seite legt. Da sie an einem runden Tisch sitzen hat so jeder von ihnen zwei Stäbchen neben sich zu liegen.

Es wird folgende Regel ausgemacht:

I. Ein Philosoph, der Hunger hat, nimmt sich erst sein linkes Stäbchen, dann sein rechtes und ißt, bis er satt ist. Danach legt er beide Stäbchen wieder hin und denkt, bis er wieder hungrig wird.

Da ein Streit über die Stäbchen ausbricht, wird außerdem folgende Regel eingeführt:

II. Ein einmal aufgenommenes Stäbchen gibt man nicht mehr her, bevor man gegessen hat.

a) Nach ca. 10 Wochen sind alle Philosophen verhungert. Warum?

b) Stellen Sie den Zustand der Philosophen vor dem Verhungern als Graphen dar. Die Philosophen und die Stäbchen seien hierbei zwei verschiedene Arten von Knoten. Der Besitz eines Stäbchen kann als durchgezogener Pfeil (vom Stäbchen aus) und eine Anfrage als gestrichelter Pfeil vom Philosophen zum Stäbchen dargestellt werden.

c) Die nächste Philosophengeneration denkt über eine weitere Regel nach, um sich vor dem Verhungern zu schützen. Demnach muß sich ein eßwilliger Philosoph einen lustigen Hut aufsetzen, bevor er anfangen darf, nach Stäbchen zu grabschen. Das Privileg, einen lustigen Hut zu tragen, hat man jedoch nur, wenn keiner der Nachbarn einen solchen trägt. Schreiben Sie unter Beachtung der drei Regeln einen Algorithmus (Pseudocode) für einen der Philosophen auf.

d) Zwei der Philosophen sind der Ansicht "Ein leerer Magen denkt schlecht". Warum droht der Philosoph zwischen den beiden zu verhungern?

e) Die Hüte werden daraufhin wieder abgeschafft. Der verhungerte Philosoph wird durch einen neuen ersetzt, der jedoch als Rechtshänder darauf besteht, sich zuerst das rechte Stäbchen zu nehmen. Prüfen Sie anhand eines neuen Graphen (s. Teil b), ob sich die Situation jetzt bessert.

Bounuspunkt: Der Rechtshänder war altehrwürdig und 101 Jahre alt – Zeit für seinen Vorruhestand. Ein neuer Philosoph tritt an seine Stelle. Im Gegensatz zu seinen Kollegen verlässt er den Tisch nach dem Essen zu einem kurzen Verdauungsspaziergang, was seinen Mitstreitern für ein paar Happen reicht. Außerdem ist er körperlich so gut gebaut, daß er seine Nachbarn zur Freigabe ihrer Stäbchen zwingen kann, bevor er selbst verhungert. Werden in dieser Fünferrunde alle satt?

Aufgabe 4 (5 Punkte):

Auf der i6–WebSeite finden Sie das Programm `u11.c`. Es handelt sich um eine Implementierung des Erzeuger-Verbraucher-Problems (entsprechend der Aufgabe 3 aus der 10. Übung) mit zwei Verbrauchern. Die Synchronisation der Verbraucherprozesse erfolgt über UNIX-Semaphoren.

a) Informieren Sie sich über die Handhabung von Semaphoren unter UNIX! Lesen Sie dazu die man-Pages von `semget()`, `semctl()` und `semop()` und studieren Sie den gegebenen Quelltext. Testen Sie das Programm!

b) Erweitern Sie das Programm um einen weiteren Erzeugerprozeß! Stellen Sie bei Ihrer Lösung durch eine zweite Semaphore sicher, daß alles richtig funktioniert.

c) Implementieren Sie eine Simulation des *5-Philosophen-Problems* mit der in Aufgabe 3 diskutierten "Rechtshänderlösung" unter Verwendung von UNIX-Semaphoren! Die Zeiten, die ein Philosoph ißt oder denkt, sollen Zufallszahlen sein, die Sie mit der Funktion `rand()` (siehe man-Pages) generieren. Mit `srand()` können Sie den Startwert des Zufallszahlengenerators setzen und damit während der Programmentwicklung beeinflussen, ob immer dieselbe oder eine neue Sequenz von Zufallszahlen erzeugt wird.

   Ein Ergebnisprotokoll (Ausgaben des Programms) soll zeigen, daß ihre Lösung einwandfrei funktioniert!

Lehrstuhl für Informatik VI **23. Januar 2001**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

12. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 13. Vorlesungswoche (31.01./01.02.2001) in den Übungsgruppen

---

Aufgabe 1 (4 Punkte):

Der Hauptspeicher eines Computers stellt eine wertvolle Ressource dar. Jedes Programm sollte deswegen immer nur soviel davon belegen, wie es zum momentanen Zeitpunkt benötigt.

Gehen wir davon aus, daß m Speicherbereiche gleicher Größe vorliegen und a verschiedene Programme gestartet werden sollen, die jeweils $\min_i$ Speicherbereiche minimal und $\max_i$ Speicherbereiche maximal ($i = 1 \ldots a$) benötigen. Der Speicherbedarf jedes einzelnen Programms variiert während seiner Laufzeit innerhalb dieser Grenzen, sein Bedarf zum Zeitpunkt T sei durch $\mathrm{now}_i(T)$ gegeben. I_t ist die Indexmenge aller zum Zeitpunkt t laufenden Programme. Voraussetzung sei, daß $\forall i$ gilt: $\max_i < m$.

a) Zeigen Sie an einem Beispiel ($m = 10$; $a = 4$), daß es trotz der Voraussetzung bei schlechtem Vorgehen zu einem Deadlock kommen kann. D.h. kein im Speicher befindliches Programm kann weitermachen, da ihm für den nächsten Schritt nicht genügend Speicher zur Verfügung gestellt werden kann.

b) $\sum_{i \in I_t} \mathrm{now}_i(T) < m$ stellt eine erste Betriebsmittelbeschränkung dar. Suchen Sie nach Ungleichungen (auch mit Quantoren), die erfüllt sein müssen, damit das System nicht in einen Deadlock gerät.

c) Geben Sie nun einen Algorithmus an, der zu startende bzw. weiterzuführende Programme so auswählt, daß der Fortschritt des Systems stets gewährleistet ist.

d) Begründen Sie, daß ihr Algorithmus funktioniert.

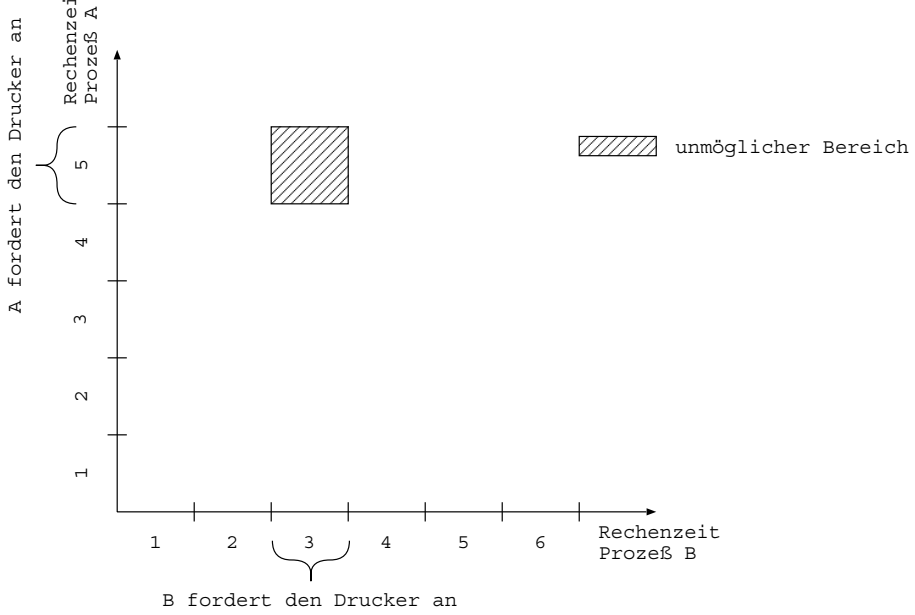
Aufgabe 2 (4 Punkte):

In dieser Aufgabe sollen Sie sich mit konkurrierenden Zugriffen auf exklusive Betriebsmittel und der Vermeidung von Deadlocks auseinandersetzen.

Wenn zwei Prozesse während ihrer Laufzeit verschiedene Betriebsmittel exklusiv benutzen wollen, könnten Sie das vorher dem System zur Synchronisation mitteilen. Das Betriebssystem kann dann, falls es überhaupt möglich ist, ihren Ablauf so koordinieren, daß ein erfolgreiches Beenden beider Prozesse gesichert ist.

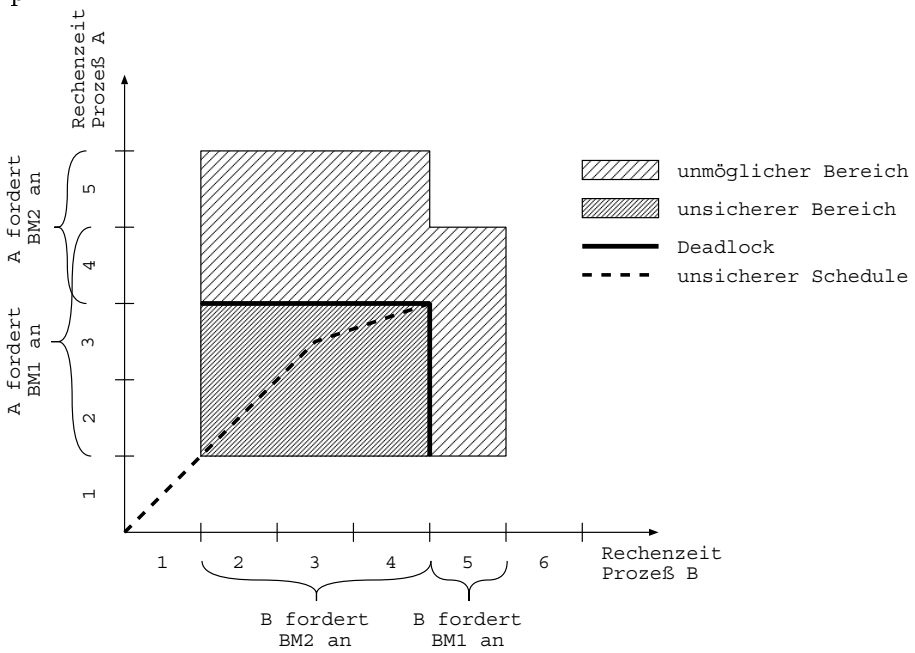
Zur Veranschaulichung des Ablaufs bedient man sich sogenannter *Prozeßfortschrittsdiagramme*. Die Achsen markieren dabei die Laufzeiten der beiden Prozesse, die um Betriebsmittel konkurrieren. Schraffiert werden die Zeiträume, in denen beide Prozesse ein exklusives Betriebsmittel gleichzeitig beanspruchen.

Beispiel: Wenn Prozeß A den Drucker zum Zeitpunkt 5 und Prozeß B zum Zeitpunkt 3 benötigt, ergibt sich folgendes Bild:



Ein *Schedule* gibt an, wie die Prozesse innerhalb ihrer Rechenzeit fortschreiten. Er kann als ein Pfad durch das Diagramm dargestellt werden. Schraffierte Flächen heißen *unmöglicher Bereich*, weil sie von keinem Schedule erreicht werden können.

Die Aufgabe, schraffierte Bereiche zu umgehen, stellt mit den bisherigen Strategien kein Problem dar. Deadlocks treten jedoch auf, wenn ein Schedule in einen *unsicheren Bereich* führt. Das bedeutet, daß in Kürze keiner der Prozesse mehr weiter kann, ohne daß ein schraffierter Bereich betreten wird. Hierzu ein weiteres Beispiel:



Haben die Prozesse A und B jeweils drei Zeittakte gearbeitet, läuft das System zwangsweise in einen Deadlock. Nach vier Zeittakten geht nichts mehr, weil jeder auf ein Betriebsmittel wartet, das gerade vom anderen Prozeß belegt wird.

a) Wieviele Betriebsmittel und Prozesse müssen an einer Deadlocksituation beteiligt sein? Welche Voraussetzung sind außerdem nötig?

b) Gegeben seien die Prozesse A und B, die jeweils zehn Zeittakte arbeiten, und fünf exklusiv zu nutzende Betriebsmittel BM1 bis BM5. Tragen Sie die Situation in ein Prozeßfortschrittsdiagramm ein.

	BM1	BM2	BM3	BM4	BM5
Prozeß A	2-3	3-5	5-8	2-6	9
Prozeß B	3-5	1-3	7-8	9-10	2-3

Hinweis: Der erste Tabelleneintrag 2-3 heißt, daß der Prozeß A das Betriebsmittel BM1 in der 2. und 3. Zeiteinheit benötigt.

c) Entscheiden Sie, welchen Status (Deadlock, unmöglich, sicher, unsicher) folgende Zeitpunkte haben:

$$(2,2); (7,4); (5,6); (5,3); (10,10); (7,9)$$

d) Geben Sie einen Algorithmus an (verbal), nach dem man unsichere Bereiche markieren kann.

e) Schreiben Sie einen sicheren Schedule für die beiden Prozesse auf und zeichnen Sie ihn in das Diagramm ein.

Aufgabe 3 (12 Punkte):

In den letzten Übungen haben Sie verschiedene Möglichkeiten der Prozeßkommunikation und -synchronisation in C/UNIX kennengelernt. Eine praktische Anwendung stellen Datenbanksysteme dar.

Ihre Aufgabe soll es nun sein, ein rudimentäres Datenbanksystem zu implementieren. Die von unserer Seite gemachten Vorgaben betreffen lediglich die Funktionalität der Datenbank, alle Aspekte der Implementierung können Sie selbst entscheiden.

Die Datenbank soll eine Liste von Studenten verwalten, die zu einem bestimmten Zeitpunkt in einem lokalen Rechnernetz eingeloggt sind. Folgende Funktionen sind zu implementieren:

`int login (char *user, char *machine) :` schickt eine Nachricht an die Datenbank, daß sich der Student `user` am Rechner `machine` eingeloggt hat. Der Rückgabewert gibt an, ob der Funktionsaufruf erfolgreich war.

`int logout (char *user, char *machine) :` schickt eine Nachricht an die Datenbank, daß sich der Student `user` vom Rechner `machine` ausgeloggt hat. Rückgabewert wie bei `login()`.

`int who_uses (char **user, char *machine) :` erfragt von der Datenbank, wieviele und welche Studenten zum aktuellen Zeitpunkt am Rechner `machine` arbeiten. Rückgabewerte sind die Anzahl und eine Liste von Strings (`**user`).

`int what_uses (char *user, char **machine) :` erfragt von der Datenbank, an wievielen und welchen Rechnern der Student `user` zum aktuellen Zeitpunkt eingeloggt ist. Rückgabewert analog zu `who_uses()`.

`int free_machines (char **machine) :` erfragt eine Liste aller Rechner, an denen keiner eingeloggt ist.

Die Datenbank soll von einem Serverprogramm verwaltet werden. Das Programm initialisiert die leere Datenbank, liest eine Liste `rechner.txt` von Rechnernamen im lokalen Netz ein und wartet dann in einer Endlosschleife auf Anfragen.
Bei jeder neuen Anfrage startet der Server einen Sohnprozeß, der die Kommunikation mit dem Klienten übernimmt und die gewünschte Funktion ausführt. Damit ist gewährleistet, daß der Server während der Bearbeitung einer Anfrage bereits die nächste entgegennehmen kann. Fehlerhafte Anfragen (z.B. nach einem Rechner, der nicht existiert) werden auf dem Bildschirm des Servers mit einer Warnung und dem Rückgabewert `-1` quittiert.

Zu jeder der fünf Funktionen gibt es ein eigenes Klientenprogramm, dem die Parameter in der Kommandozeile übergeben werden und das dann den Kontakt zum Server aufnimmt. Z.B. sollte der Aufruf der Login-Funktion vom Studenten *Meier* am Rechner *wasserstoff* am Kommandoprompt

`wasserstoff# login meier wasserstoff`

lauten. Die Ergebnisse der Abfragen sollen auf dem Bildschirm ausgegeben werden. Danach wird das Klientenprogramm beendet.

a) Wählen Sie eine geeignete Art der Kommunikation zwischen den Klienten und dem Server sowie einen Mechanismus zur Sicherung der Konsistenz des Datenbestandes. Legen Sie die folgenden vier Regeln zugrunde:

   - In dem kritischen Bereich dürfen sich entweder nur ein Schreibprozeß oder beliebig viele Leseprozesse aufhalten.
   - Solange sich mindestens ein Leseprozeß in dem kritischen Bereich befindet, dürfen beliebig viele weitere Leseprozesse den Bereich betreten.
   - Ein Schreibprozeß darf erst dann in den kritischen Bereich, wenn sich kein Leseprozeß darin befindet.
   - Ein auf Einlaß wartender Schreibprozeß darf die ihm folgenden Leseprozesse nicht am Eintreten hindern, solange sich noch mindestens ein lesender Prozeß im kritischen Bereich befindet.

b) Welche fehlerhaften Anfragen kann es geben? Entwerfen Sie die Struktur der Klientenfunktionen und des Servers in Pseudocode. Kommentieren Sie Ihren Entwurf.

c) Implementieren Sie die Funktionen in C und testen Sie die Programme.

Lehrstuhl für Informatik VI **29. Januar 2001**
Rheinisch–Westfälische Technische Hochschule Aachen **F. Hilger, F.J. Och**
Prof. Dr.–Ing. H. Ney {och,hilger}@i6.informatik.rwth-aachen.de

13. Übung zur Vorlesung "Systemprogrammierung"

Abgabe: 14. Vorlesungswoche (07./08.02.2001) in den Übungsgruppen

---

Aufgabe 1 (5 Punkte):

Gegeben seien drei Prozesse P1, P2 und P3, die drei exklusive Betriebsmittel BM1, BM2 und BM3 benutzen wollen. Es existieren 9 Exemplare von BM1, 5 von BM2 und 6 von BM3: Available $= (9, 5, 6)$. Weiterhin gilt:

- $\text{Max}(1) = (6, 3, 3)$, $\text{Allocation}(1) = (0, 2, 1)$
- $\text{Max}(2) = (4, 3, 5)$, $\text{Allocation}(2) = (1, 0, 1)$
- $\text{Max}(3) = (5, 1, 1)$, $\text{Allocation}(3) = (3, 1, 1)$

a) Prüfen Sie mit dem in der Vorlesung vorgestellten *Banker's Algorithmus* zur Deadlockvermeidung, ob sich das System in einem sicheren Zustand befindet!

b) Welche der folgenden Zuteilungen führen vom obigen Zustand ausgehend in einen sicheren Zustand, d.h. welche Zuteilung darf erlaubt werden?

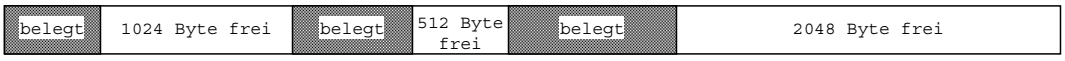
1. $\text{Request}(2) = (0, 2, 0)$
2. $\text{Request}(2) = (0, 0, 2)$
3. $\text{Request}(3) = (0, 0, 1)$
4. $\text{Request}(3) = (0, 1, 0)$

c) Der Banker's Algorithmus arbeitet konservativ. Er ist so ausgelegt, daß ein Deadlock auch im ungünstigsten Fall vermieden werden kann. Geben Sie ein Beispiel für einen Schedule an, der zwar ohne Deadlock auskommt, vom Banker's Algorithmus jedoch nicht als sicher eingestuft wird.

d) Mit welchen zusätzlichen Informationen könnten Sie für einen "offensiveren" Algorithmus entwerfen? Skizzieren Sie den Ablauf Ihres Verfahrens.

Aufgabe 2 (5 Punkte):

Gegeben sei folgende Belegung eines Speichers:



Weiterhin seien vier Belegungensmethoden für Speicherplatzanforderungen gegeben:

First-Fit	Belege den ersten freien Speicherbereich, der groß genug ist, die Anforderung zu erfüllen.
Rotating-First-Fit	Wie First-Fit, jedoch wird von der Position der vorherigen Platzierung ausgehend ein passender Bereich gesucht. Wird das Ende des Speichers erreicht, so wird die Suche am Anfang des Speichers fortgesetzt (maximal bis zur Position der vorherigen Platzierung). Bei der ersten Anforderung beginnt die Suche am Anfang des Speichers.
Best-Fit	Belege den kleinsten freien Speicherbereich, in den die Anforderung paßt.
Worst-Fit	Belege den größten freien Speicherbereich, in den die Anforderung paßt.

a) Wie sieht der obige Speicher aus, wenn nacheinander vier Anforderungen der Größe 384 Byte, 640 Byte, 512 Byte und 2048 Byte ankommen? Notieren Sie für jede Strategie die freien Speicherbereiche nach jeder Anforderung (z.B. (1024, 512, 2048) für die obige Belegung), und geben Sie an, für welche Methoden die Anforderungen erfüllt werden können!

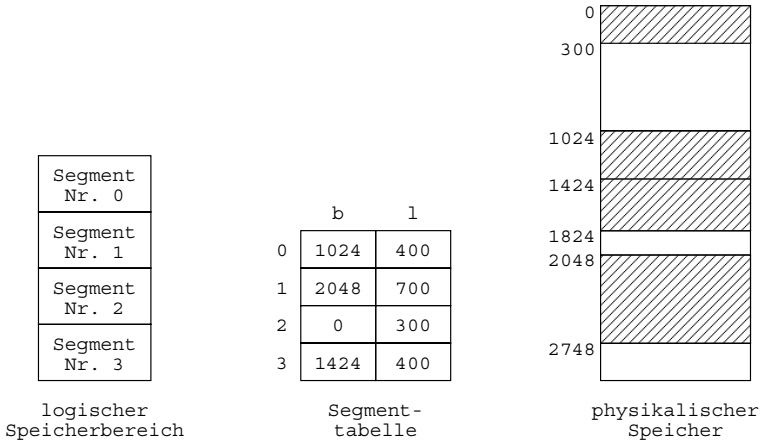
   Beachten Sie, daß die Daten jeweils linksbündig in einer Lücke abgelegt und einmal belegte Speicherbereiche nicht wieder freigegeben werden!

b) Sie dürfen nun die Reihenfolge der freien Speicherbereiche und die der Anforderungen vertauschen (aber keine Speicherbereiche zusammenfassen). Geben Sie für jede Strategie eine Speicherbelegung und die zugehörigen Anforderungen an, die für jeweils nur diese Methode erfolgreich arbeitet und für die übrigen Methoden nicht alle Anforderungen erfüllen kann! Geben Sie auch die Speicherbelegungen nach Abarbeitung der einzelnen Anforderungen an!

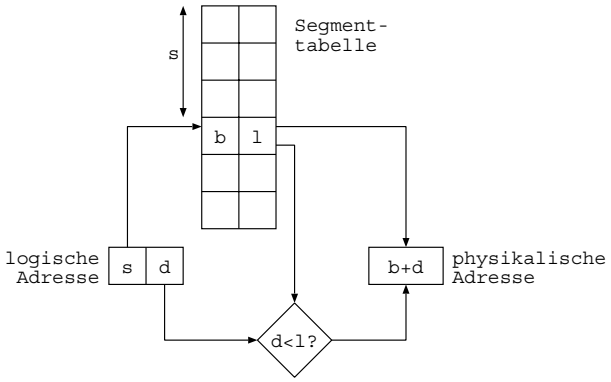
Aufgabe 3 (4 Punkte):

Ein großer Vorteil der virtuellen Speicherverwaltung ist, daß der Programmierer sich nicht um den physikalischen Adreßraum kümmern muß, sondern mit logischen Adressen arbeiten kann. Spätestens bei der Ausführung eines Programmes müssen diese logischen Adressen aber in physikalische Adressen umgewandelt werden.

Das Umsetzungsprinzip beim *Segmenting* funktioniert wie folgt: Jedem Programm ist eine Segmenttabelle zugeordnet, in der für jedes logische Segment die physikalische Anfangsadresse `b` und die Größe des Segments `l` eingetragen ist, die das Programmsegment im Speicher einnimmt.



Eine logische Adresse besteht nun aus der Segmentnummer `s`, mit der aus der Segmenttabelle die entsprechende Anfangsadresse `b` bestimmt wird, und einem Offset `d`, der angibt, an welcher Stelle in Relation zur Anfangsadresse sich die Adresse, auf die man zugreifen will, befindet. Dabei wird geprüft, ob der Offset kleiner als die Segmentlänge `l` ist, da anderenfalls eine ungültige Adresse angesprochen wird. Die physikalische Adresse ergibt sich also zu `b+d`.



Für die Speicherverwaltung nach dem Segmentierungsverfahren ist für ein Programm folgende Segmenttabelle gegeben (Länge in Speicherworten):

Segment	Basis	Länge
0	1410	365
1	400	70
2	500	120
3	630	515
4	1145	150

a) Wieviele Speicherworte stehen dem Programm im physikalischen Speicher zur Verfügung?

b) Welches ist die kleinste und welches die größte für das Programm verfügbare physikalische Adresse?

c) Berechnen Sie zu den folgenden physikalischen Adressen jeweils die logischen Adressen:

1. 762
2. 1145
3. 1146
4. 485

Aufgabe 4 (6 Punkte):

Im Mittelpunkt der Vorlesung Systemprogrammierung stehen derzeit Algorithmen der Speicherzuweisung an Prozesse. Aus diesem Grund wollen wir noch einmal genauer auf die Speicherverwaltung unter UNIX eingehen.

In einer früheren Übung haben Sie bereits gelernt, daß man Variablen sowohl direkt von einem bestimmten Typ als auch indirekt als Pointer auf diesen Typ deklarieren kann.

```
int i;
int *j;
```

Will man mit der Integervariable `*j` arbeiten, muß man ihr zunächst mit `malloc()` einen Speicherplatz zuweisen:

```
j=malloc(sizeof(int));
```

Während sich `i` auf dem Stack befindet, wird der Platz für `j` auf dem Heap reserviert.

Wird die Variable `j` nicht mehr benötigt, kann der Speicherplatz mit

```
free(j);
```

wieder freigegeben werden.

Den Speicherbedarf eines Prozesses kann man mit verschiedenen UNIX-Kommandos ermitteln. `ps` (process status) liefert neben einer Vielzahl weiterer Werte Angaben über den aktuellen Speicherbedarf. `top` gibt einen Echtzeit-Überblick über alle aktiven Prozesse im System. Außerdem wird (bei einigen UNIX-Implementationen) angegen, wieviel Speicher im System insgesamt vorhanden ist und wieviel davon gerade genutzt wird.

a) Welche Funktion hat `sizeof()`? Wieso wird im obigen Beispiel nicht Speicher mit `j=malloc(4);` angefordert? Worin besteht der Unterschied zwischen `malloc()` und `calloc()`? Welche C-Funktion müssen Sie nach `malloc()` wie aufrufen, um dieselbe Funktionalität wie `calloc()` zu erreichen?

b) Mit welcher Option muß `ps` gestartet werden, damit der Speicherbedarf von Prozessen ausgegeben wird? Ermitteln Sie, welcher Prozeß auf Ihrem Rechner gerade am meisten Speicherplatz belegt. Versuchen Sie zu bestimmen, um was für einen Prozeß es sich handelt.

c) Schreiben Sie ein einfaches Programm, das im Sekundentakt 500 Byte Speicher anfordert. Starten Sie das Programm und überprüfen Sie den Speicherplatzverbrauch zur Laufzeit. Was stellen Sie fest? Erklären Sie ihre Beobachtung.

d) Ermitteln Sie, wieviel physischen Speicher der Rechner besitzt, an dem Sie arbeiten. Versuchen Sie einen Speicherbereich zu allozieren, der größer ist. Testen Sie anhand des Rückgabewertes von `malloc()`, ob die Speicheranforderung erfolgreich war. Wie kann es möglich sein, daß ein Prozeß mehr Speicherplatz zugeteilt bekommt, als der Rechner physisch besitzt?

e) Bestimmen Sie durch Probieren näherungsweise das Maximum an Speicher, das Sie mit `malloc()` anfordern können. Fordern Sie einen Speicherbereich dieser Größe an. Lesen Sie nun die erste Millionen Speicherstellen und eine Millionen zufällig mit `rand()` gewählter Speicherstellen aus. Beachten Sie dabei, daß Sie nicht die von Ihnen angeforderten Speichergrenzen übersteigen. Können Sie einen Geschwindigkeitsunterschied feststellen?

   Initialisieren Sie vor einem zweiten Testlauf ihren Speicher mit Zufallszahlen. Was ergibt sich jetzt?

   Wiederholen Sie das Experiment, indem Sie diesmal die Speicherstellen nicht lesen sondern dort der Wert 42 eintragen. Gibt es jetzt Unterschiede?

Lehrstuhl für Informatik VI
der RWTH Aachen
Prof. Dr.-Ing. H. Ney

Beispiel-Klausur zu Systemprogrammierung

Hinweis: Fassen Sie sich kurz! Es gibt mehrere Aufgabenteile, in denen Sie Algorithmen erlutern, Begriffe erklren oder Vor- und Nachteile einzelner Verfahren diskutieren sollen. Antworten Sie kurz und prgnant, sonst reicht die Zeit nicht.

Aufgabe S 1: **2+2+1+1 Punkte**

a) Auf die Frage "Welche Komponente eines Rechnersystems sollte die Verwaltung der Resourcen (z.B. Hauptspeicher, Semaphoren, usw.) übernehmen?" bekommt ein Syspro-Übungsleiter drei verschiedene Antworten:

1. Die CPU, denn sie muß Deadlocks vermeiden.
2. Der Anwendungsprogrammierer, denn er braucht die Resourcen erst zu dem Zeitpunkt anzufordern, wenn sie wirklich benötigt werden.
3. Für jede Resource muß es im Betriebssystem eine spezielle Verwaltungseinheit geben, die eingehende Anforderungen der verschiedenen Anwendungsprozesse bearbeitet.

Welche Antwort ist richtig? Widerlegen Sie die beiden anderen Antworten.

b) Unter welchen Bedingungen kann ein Deadlock entstehen? Was ist der Unterschied zwischen "Deadlock Prevention", "Deadlock Avoidance" und "Deadlock Detection"?

c) Benötigt man zur Implementation des Bakery-Algorithmus Semaphoren oder atomare Hardwareoperationen wie `swap` und `test-and-set`? Begründen Sie Ihre Antwort.

d) Was unterscheidet Prozesse in den Zuständen "ready", "running" und "waiting"? Gibt es Prozesse in allen diesen Zuständen, wenn das System in einem Deadlockzustand ist?

Aufgabe S 2: **1+2+2+3 Punkte**

Das Erzeuger-Verbraucher-Problem ist ein Spezialfall des Reader-Writer- Problem. Wir betrachten in dieser Aufgabe eine Nachrichtenqueue, für die es einen Erzeuger und mehrere Verbraucher gibt. Die Queue ist statisch implementiert und hat Platz für 100 Nachrichten:

```
#define MaxMessage 100

typedef struct {
        int     id;
        char    msg[256];
        } Message;

Message MessageBuffer[MaxMessage];
```

a) Nennen Sie einen Vorteil und einen Nachteil der statischen Implementierung einer Queue.

b) Warum muß eine solche Warteschlange synchronisiert werden? Nennen Sie drei verschiedene Fälle, in denen fehlende Synchronisation zu Inkonsistenzen und zum Datenverlust führt.

c) Die Funktion der atomaren Hardwarebefehle `DecTest()` und `TestInc()` kann durch folgenden Pseudocode beschrieben werden:

```
int DecTest (int *i) {
  *i = (*i) - 1;
  if (*i == 0)
    return (0);
    else return (1);
  }
```

```
int TestInc (int *i) {
  if (*i == 0) {
    *i = (*i) + 1;
    return(0);
    }
  else {
    *i = (*i) + 1;
    return(1);
    }
  }
```

Geben Sie die Implementation des Nachrichtenerzeugers und der Verbraucher in Pseudocode an. Verwenden Sie dabei die beiden Hardwarebefehle.

d) Ein höhersprachliches Konzept zur Synchronisation sind Semaphoren. Modifizieren Sie Ihre Lösung, indem Sie nun Semaphoren statt der Hardwarebefehle verwenden. Zeigen Sie kurz, daß Ihre Lösung alle Bedingungen des wechselseitigen Ausschlusses erfüllt.

Aufgabe S 3: **2+1+2+3 Punkte**

An einem Mainframe sind fünf alphanumerische Terminals angeschlossen. Es handelt sich um "intelligente Eingabegeräte", d.h. sie haben Tastatur, Maus und Monitor, können jedoch selber keine Prozesse ausführen. Sie geben ihre Jobs an den Zentralrechner weiter, der zur Kommunikation mit den Terminals eine Warteschlange besitzt. Die Terminals arbeiten synchron, d.h. sie sind blockiert, solange sie auf eine Antwort vom Server warten.
Tagsüber sind alle Terminals ständig besetzt. Das Zeitverhalten jedes Terminals für sich genommen stellt einen Poisson-Prozeß dar: In der Zeit, in der es nicht auf eine Serverantwort wartet, schickt das Terminal im Mittel 4 Anfragen pro Minute an den Zentralrechner. Die Antwortzeiten des Mainframes sind exponentialverteilt. Er benötigt durchschnittlich 4 Sekunden pro Anfrage.

a) Um was für eine Bediensystem handelt es sich (Kendall-Notation)? Zeichnen Sie das Zustandsdiagramm für dieses System und leiten Sie daraus die Bilanzgleichungen für den Gleichgewichtszustand ab.

b) Kann das System unter den genannten Umständen überhaupt in den Gleichgewichtszustand gelangen? Wie groß muß die Warteschlange mindestens sein, damit keine Anfragen abgewiesen werden?

c) Ist der Server überlastet? Berechnen Sie dazu, wieviel Prozent der Anfragen nicht sofort bearbeitet werden können, sondern zunächst in die Warteschlange gelangen.

d) Um mehr Nutzer gleichzeitig arbeiten zu lassen beschließt man, zwei weitere Terminals an den Mainframe anzuschliessen, ohne dessen Rechenkapazität zu erhöhen. Um wieviel Prozent erhöht sich daraufhin die mittlere Antwortzeit auf eine Terminalanfrage?

   Schätzen Sie qualitativ ab, ob die Erweiterung aus der Sicht der Nutzer einen Gewinn (weil sie schneller an ein freies Terminal kommen) oder einen Verlust (weil der Server und damit die Terminals langsamer sind) darstellt?

Aufgabe S 4: **1+2+2+2 Punkte**

Das folgende C-Programm realisiert einen Druckerscheduler und eine Reihe von Anwendungsprozessen. Die Prozesse wollen zu unterschiedlichen Zeiten auf den Drucker zugreifen. Sie signalisieren das dem Scheduler, indem Sie ihr `printer_request_flag` auf `True` setzen. Der Scheduler verwaltet eine Warteschlange, in die alle Requests eingetragen werden, die nicht sofort abgearbeitet werden können. Über die Variable `printer` signalisiert er, welcher Prozeß den Drucker nutzen darf. Hat dieser seinen Druckjob beendet, signalisiert er das dem Scheduler, indem er sein `printer_return_flag` auf `True` setzt.

```
001: #include <stdio.h>
002: #include <stdlib.h>
003: #include <unistd.h>
004: #include <sys/types.h>
005: #include <sys/ipc.h>
006: #include <sys/shm.h>
007:
008: #define NumOfProc 10
009: #define False 0
010: #define True 1
011:
012: typedef struct tmp {
013:   int process_id;
014:   struct tmp *next;
015:   } queue_entry;
016:
017: typedef struct {
018:   queue_entry *first;
019:   queue_entry *last;
020:   int num_entries;
021:   } queue;
022:
023: queue printer_queue={NULL,NULL,0};
024:
025: int *printer;
026: int *printer_request_flag[NumOfProc];
027: int *printer_return_flag[NumOfProc];
028:
029: void printer_scheduler() {
030:   queue_entry *new,*old;
031:   int proc;
032:
033:   do {
034:     for (proc=0;proc<NumOfProc;proc++) {
035:
036:       if (*printer_request_flag[proc]) {
037:         printf("[SCHEDULER] printer request from process %i ...\n",proc);
038:         *printer_request_flag[proc]=False;
039:         if (*printer==-1) {
040:           printf("[SCHEDULER] assign printer to process %i ...\n",proc);
041:           *printer=proc;
042:           }
043:         else {
044:           printf("[SCHEDULER] queue printer request from process %i ...\n",proc);
045:           new=malloc(sizeof(queue_entry));
046:           new->process_id=proc;
047:           new->next=NULL;
048:           if (printer_queue.num_entries==0)
049:             printer_queue.first=new;
050:           else printer_queue.last->next=new;
051:           printer_queue.last=new;
052:           printer_queue.num_entries++;
053:           }
054:         } /* if (*printer_request_flag[proc]) */
055:
056:       if (*printer_return_flag[proc]) {
057:         printf("[SCHEDULER] process %i returns printer ...\n",proc);
```

```
058:         *printer_return_flag[proc]=0;
059:         if (printer_queue.num_entries==0)
060:           *printer=-1;
061:         else {
062:           old=printer_queue.first;
063:           printer_queue.first=printer_queue.first->next;
064:           if (printer_queue.first==NULL) printer_queue.last=NULL;
065:           printer_queue.num_entries--;
066:           printf("[SCHEDULER] assign printer to process %i ...\n",old->process_id);
067:           *printer=old->process_id;
068:           free(old);
069:           }
070:       } /* if (*printer_return_flag[proc]) */
071:     } /* for */
072:   } while(True);
073: } /* printer_scheduler */
074:
075:
076: void process(int process_id) {
077:
078:   srand(process_id);               /* initialize randon number generator */
079:   printf("[PROCESS%02i] process %i started ...\n",process_id,process_id);
080:   do {
081:     sleep(rand()%10);                      /* process does something else */
082:     printf("[PROCESS%02i] request printer ...\n",process_id);
083:     *printer_request_flag[process_id]=True;
084:     while (*printer_request_flag[process_id]==True) sleep(1);
085:     while (*printer!=process_id) sleep(1);
086:     printf("[PROCESS%02i] printing ...\n",process_id);
087:     sleep(rand()%2);                                /* process is printing */
088:     printf("[PROCESS%02i] return printer ...\n",process_id);
089:     *printer_return_flag[process_id]=True;
090:     while (*printer_return_flag[process_id]==True) sleep(1);
091:     } while (True);
092:   } /* process */
093:
094:
095: void main() {
096:
097:   int i,shmid;
098:                                          /* get shared memory segment */
099:   shmid=shmget(IPC_PRIVATE,(2*NumOfProc+1)*sizeof(int),IPC_CREAT|0x1ff);
100:   printer=(int *)shmat(shmid,NULL,0);  /* map variables to shared memory */
101:   *printer=-1;                                    /* and initialize them */
102:   printer_request_flag[0]=(printer+1);
103:   for (i=1;i<NumOfProc;i++)
104:     printer_request_flag[i]=printer_request_flag[i-1]+1;
105:   printer_return_flag[0]=printer_request_flag[NumOfProc-1]+1;
106:   for (i=1;i<NumOfProc;i++)
107:     printer_return_flag[i]=printer_return_flag[i-1]+1;
108:
109:   for (i=0;i<NumOfProc;i++) {
110:     *printer_request_flag[i]=*printer_return_flag[i]=False;
111:     if (fork()!=0) process(i);
112:     }
113:
114:   printer_scheduler();
115:   }
```

a) Welches Schedulingverfahren wird mit dem vorgegebenen Programm realisiert? Wieviel Prozesse sind bei Ablauf des Programms aktiv?

b) Warum sind die Variablen `printer`, `printer_request_flag` und `printer_return_flag` in einem shared-memory-Segment untergebracht? Warum ist ihre Konsistenz sichergestellt, obwohl auf sie von verschiedenen Prozessen gleichzeitig zugegriffen wird?

c) Nennen Sie zwei wesentliche Schwächen der Programms. Wie könnte man diese Schwächen beheben?

d) Modifizieren Sie den Quelltext so, daß der Scheduler nach dem LCFS-Verfahren arbeitet.

Aufgabe S 5: **2+2+2+2 Punkte**

Gegeben sei ein System mit 4 Prozessen $P1$ bis $P4$ und 5 Betriebsmitteln $BM1$ bis $BM5$. Während die Betriebsmittel $BM4$ und $BM5$ von beliebig vielen Prozessen gleichzeitig genutzt werden können, sind die Betriebsmittel $BM1$ bis $BM3$ exklusiv zu nutzen.
Zu Beginn haben die Prozesse jeweils ihre maximalen Anforderungen an den Betriebsmitteln bekanntgegeben:

Max_P1=(5,3,3,1,1)

Max_P2=(4,2,6,0,1)

Max_P3=(4,5,4,1,0)

Max_P4=(3,5,2,0,1)

Zu einem bestimmten Zeitpunkt x der Abarbeitung haben sie folgende Betriebsmittel belegt:

Alloc_P1=(2,2,1,1,0)

Alloc_P2=(0,1,0,0,1)

Alloc_P3=(3,1,3,0,0)

Alloc_P4=(0,2,1,0,0)

Folgende Betriebsmittel sind noch verfügbar:

Avail_Tx(4,3,1,1,1)

a) Von den Betriebsmitteln $BM4$ und $BM5$ gibt es genau eine Instanz im System. Wieviele Instanzen gibt es von den Betriebsmitteln $BM1$, $BM2$ und $BM3$? Überprüfen Sie mit Hilfe des Banker's-Algorithmus, ob der gegebene Zustand sicher ist.

b) Können die folgenden Anforderungen genehmigt werden? Gehen Sie dabei jeweils vom Zustand zum Zeitpunkt x aus. Begründen Sie ihre Aussagen!

   Request_P1=(0,1,0,0,1)
   Request_P2=(0,0,2,0,0)
   Request_P3=(1,0,1,0,0)
   Request_P4=(4,3,1,0,1)

c) Kann das System die folgende Anforderung nach dem Banker's Algorithmus gewähren? Gerät es zwangsläufig in einen Deadlock, falls es die Anforderung genehmigt?

   Request_P1=(2,1,0,0,0)

d) Die Implementation des Banker's Algorithmus erscheint einem Studenten zu schwierig. Er denkt sich daher ein alternatives Verfahren aus:

   Betriebsmittelanforderungen werden nur dann genehmigt, wenn

   - die BM zum Zeitpunkt der Anforderung zur Verfügung stehen und
   - der Prozeß seine maximalen Anforderungen nicht überschreitet.

   Sollten bei einer Anforderung nicht genug Instanzen eines Betriebsmittels vorhanden sein, dann muß der Prozesse alle Instanzen dieses BM, die er bereits angefordert hat, freigeben.

   Ist dieser Algorithmus geeignet, Deadlocks zu vermeiden? Begründen Sie ihre Aussage!