BERGISCHE UNIVERSITÄT
WUPPERTAL
GAUSS-STRASSE 20
42119 WUPPERTAL
TELEFAX (0202) 439-2901
TELEFON (0202) 439-0

Fachbereich C

MATHEMATIK UND NATURWISSENSCHAFTEN
Mathematik

Prof. Dr. Hans-Jürgen Buhl
Praktische Informatik / Numerik

e-mail: Juergen.Buhl@math.uni-wuppertal.de

Betriebssysteme: Konzepte, Dienste, Schnittstellen (Betriebssysteme und betriebssystemnahe Programmierung)

SS 2005 – Übungsblatt 2

Ausgabe: 25. April 2005

Abgabe: bis spätestens 2. Mai 2005
im Fachschaftsraum Mathematik
oder per email an `c.markmann@uni-wuppertal.de`

Aufgabe 1. *file/record locking*

Geben Sie das folgende C-Programm ein und bringen Sie es zur Ausführung:

```
/* lock.c -- simple example of record locking */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

/* displays the message, and waits for the user to press
   return */
void waitforuser(char * message) {
    char buf[10];

    printf("%s", message);
    fflush(stdout);

    fgets(buf, 9, stdin);
```

```
}

/* Gets a lock of the indicated type on the fd which is passed.
   The type should be either F_UNLCK, F_RDLCK, or F_WRLCK */
void getlock(int fd, int type) {
    struct flock lockinfo;
    char message[80];

    /* we'll lock the entire file */
    lockinfo.l_whence = SEEK_SET;
    lockinfo.l_start = 0;
    lockinfo.l_len = 0;

    /* keep trying until we succeed */
    while (1) {
        lockinfo.l_type = type;
        /* if we get the lock, return immediately */
        if (!fcntl(fd, F_SETLK, &lockinfo)) return;

        /* find out who holds the conflicting lock */
        fcntl(fd, F_GETLK, &lockinfo);

        /* there's a chance the lock was freed between the F_SETLK
           and F_GETLK; make sure there's still a conflict before
           complaining about it */
        if (lockinfo.l_type != F_UNLCK) {
            sprintf(message, "conflict with process %d... press "
                    "<return> to retry:", lockinfo.l_pid);
            waitforuser(message);
        }
    }
}

int main(void) {
    int fd;

    /* set up a file to lock */
    fd = open("testlockfile", O_RDWR | O_CREAT, 0666);
    if (fd < 0) {
        perror("open");
        return 1;
    }

    printf("getting read lock\n");
    getlock(fd, F_RDLCK);
    printf("got read lock\n");

    waitforuser("\npress <return> to continue:");
```

```
    printf("releasing lock\n");
    getlock(fd, F_UNLCK);

    printf("getting write lock\n");
    getlock(fd, F_WRLCK);
    printf("got write lock\n");

    waitforuser("\npress <return> to exit:");

    /* locks are released when the file is closed */

    return 0;
}
```

Versuchen Sie in einem zweiten Terminalfenster das Programm ein zweites Mal zu starten und hier eine Schreibsperrung zu bekommen, während im anderen Prozess noch eine Lesesperrung aktiv ist. Was geschieht?

Erstellen Sie eine 3x3-Tabelle, die den Erfolg/Mißerfolg des Programms darstellt, wenn ein weiterer Prozess einen read-Lock bzw. einen write-Lock besitzt oder wenn kein weiterer Prozess einen Lock angefordert hat.

Wie ist das Programm abzuändern, wenn ein Lock nur auf einen kontinuierlichen Bytebereich einer Datei angelegt werden soll?

Aufgabe 2. *Systemaufrufe eines einfachen C-Programms*

Untersuchen Sie mittels `truss` (oder `strace`) die Systemaufrufe bei Ausführung des Programms `test-read` (Seite 1 der Materialsammlung).

Versuchen Sie in eigenen Worten zu erklären, was da alles vorgeht.

Aufgabe 3. *system()*

In

```
#include        <stdio.h>
#include        <stdlib.h>

int main(){

   if (system("rm x.txt") == 0)
      printf("Datei x.txt wurde gelöscht\n");
   else
      printf("Datei x.txt konnte   n i c h t   gelöscht werden\n");
}
```

wird das UNIX-Shellkommando `rm` innerhalb eines Programms benutzt. Was genau wird beim Programmlauf ausgegeben, wenn eine Datei `x.txt` existiert bzw. nicht existiert?

Suchen Sie im UNIX-Manual nach dem SystemCall zum Löschen von Dateien und erstellen Sie ein analoges Programm, das diesen System-Call direkt benutzt.

Aufgabe 4. *Algorithmen-Zeitmessung*

Das Programm

```
////////////////////////////////
// Datei:   TakeTime.cc
// Version: 1.0
// Zweck:   Zeitmessung von Algorithmenteilen
// Autor:   Hans-Juergen Buhl
// Datum:   20.05.2003
////////////////////////////////


#include        <iostream>
#include        <iomanip>
#include        <cmath>

#include        <time.h>
#include        <sys/times.h>
#include        <sys/types.h>
#include        <unistd.h>


using namespace std;

#ifndef CLK_TCK
        double CLK_TCK = sysconf(_SC_CLK_TCK);
#endif

double get_Time(void){

  double tim = -1.0;

  tms time_buff;

  if (times(&time_buff) < 0) {
        cerr << " times() failed " << endl;
  } else {
        double total_t = time_buff.tms_utime +
                         time_buff.tms_stime +
                         time_buff.tms_cutime +
                         time_buff.tms_cstime ;
        tim = total_t /double(CLK_TCK);
  };
  return tim;
```

```
}

int main()
{
  double StartTime(get_Time());

  // ...
  double f(sin(1.4567));
  for (int i = 0; i<1000000; i++) f = sin(f);
  // ...

  double Time(get_Time()-StartTime);

  cout <<  "Der Algorithmus dauerte "
       <<  Time
       <<  " CPU-Sekunden"
       <<  " ( Auflösung: " << 1000.0 / double(CLK_TCK) << " ms )"
       <<  endl;

// cout << CLK_TCK << endl << CLOCKS_PER_SEC << endl;

}
```

mißt die Ausführungszeit von Codesegmenten. Bringen Sie es auf Ihrem Rechner zum Laufen. Warum werden hier vier verschiedene Zeitanteile aufaddiert? (Welche sind das? Warum will man alle zusammen berücksichtigen?)

Aufgabe 5. *Turn-Around-Time*

Ändern Sie das Beispiel der vorangehenden Aufgabe so ab, dass die Turn-Around-Zeit (was ist das?) des Algorithmenanteils statt der CPU-Zeit gemessen wird. (Hinweis: Benutze den SystemCall `ftime()`)