4. Übung: Kopieren einer Datei zwischen zwei Threads eines Prozesses über einen Ringpuffer **Echtzeitsysteme**

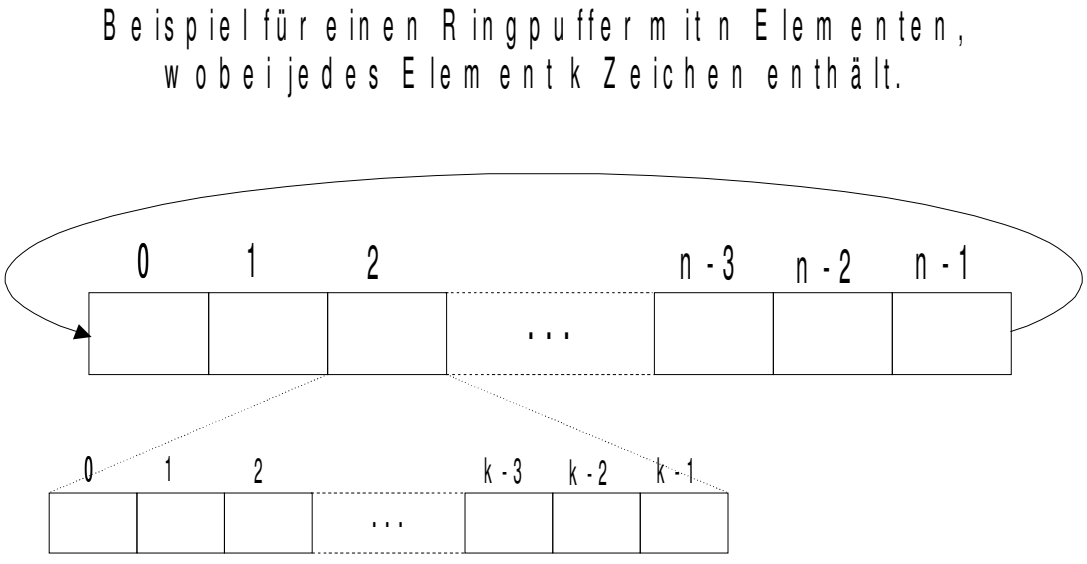
Abgabe: 12. Dezember 2011 für die Gruppe 1
19. Dezember 2011 für die Gruppe 2

Realisieren Sie das Kopieren einer !beliebigen! Ursprungs-Datei in eine Ziel-Datei dadurch, dass

- ein Thread eines Prozesses (Erzeuger) aus der Ursprungs-Datei jeweils Blöcke der Länge MAXLEN liest und diese in das nächste freie Element des Ringpuffers schreibt sowie
- ein zweiter Thread desselben Prozesses (Verbraucher) die Daten aus diesem Ringpuffer liest und sie in die Ziel-Datei schreibt.

Verwenden Sie keine vorab bestimmte Information über die Dateilänge (sie könnte sich ja während des Kopiervorganges noch ändern)!

Die Parameter des Ringpuffers (Anzahl der Pufferelemente und deren einheitliche Länge) sollen als Konstante definiert sein. Verwenden Sie zur Verwaltung zählende Semaphore. Denken Sie an das Problem des letzten aus der Datei gelesenen Datensatzes.



Lagern Sie die Zugriffe auf die Ursprungs- und Ziel-Datei in ein separates Quelltext-Modul (z.B. `fileio.c`) aus und gestalten Sie die davon zur Verfügung gestellten Funktionen so einfach wie möglich.

Mögliche Funktionen des Moduls könnten sein:

```
err = myOpen(quelle, ziel);
n   = myRead(buffer, len);.
n   = myWrite(buffer, len);
myClose();
```

Die „Dienstleistungen“ dieses Moduls stellen Sie über eine Header-Datei (z.B. `fileio.h`) zur Verfügung. Diese Datei soll auch "`#define`'s" für Fehlerrückgabewerte enthalten. Beispiele für sinnvolle "`#define`'s" findet man z.B. in der man-page für `open` (Anzeigen mit `man 2 open`, s. Abschnitt FEHLER, bzw. ERRORS).

Das Modul `fileio.c` soll nur die Dateizugriffe (`open`, `close`, `read`, `write` oder `fopen`, …) enthalten und so vom Mechanismus der Interprozesskommunikation getrennt sein. So können Sie dieses Modul auch für die nächsten Aufgaben unverändert einsetzen. Sinnvoll ist es z.B. auch, dem Nutzer dieses Moduls nur Aufrufe wie z.B. `myOpen(…)`,`myRead(…)`,`myWrite(…)` und `myClose(…)` zur Verfügung zu stellen und die File-Deskriptoren der Quell- und Ziel-Datei im Modul zu kapseln.

Im Modul `fileio.c` müssen alle System-Calls auf mögliche Fehler überprüft werden. Als Fehlerbehandlung sollen in diesem Modul keine Ausgaben erfolgen, sondern der Grund des Fehlers in sinnvoller Codierung (s. obige Bemerkung zur header-Datei) als Wert der Funktionen an die Aufrufstelle zurückgegeben werden. Dort hat dann eine entsprechende Fehlerbehandlung zu erfolgen (z. B. Ausgabe einer Fehlermeldung, Programmabbruch bzw. Erfragen von neuen Dateinamen).

Achten Sie darauf, dass keine Bestandteile des Kopiervorganges und keine Mechanismen der Interprozesskommunikation in das Funktionsmodul (`fileio.c`) gelangen.

Die Namen von Quell und Zieldatei (auch eine absolute Pfadangabe soll möglich sein) sollen beim Aufruf des Programms auf der Kommandozeile angebbar sein. Achten Sie auf eine sinnvolle Überprüfung der eingegebenen Dateinamen und die Ausgabe entsprechender Fehlermeldungen.

Erstellen Sie eine zugehörige Datei `Makefile` zur Verwendung mit dem Hilfsprogramm `make`. Dabei sollen unnötige Übersetzungs- und Link-Vorgänge vermieden werden.

Um das korrekte Kopieren zu überprüfen, können Sie das Kommando `diff` verwenden.

An dieser Stelle sei nochmals an die **Anforderungen der anzufertigenden Quelltexte** erinnert, so wie sie im Aufgabenblatt der ersten Aufgabe genannt sind (hier speziell die Funktionsköpfe für die Funktionen in `fileio.c`).

Das nachfolgende Beispiel zeigt die Verwendung von Semaphoren zur Synchronisation zwischen dem main-Thread und einem Kind-Thread. Es stellt ***keinen*** Lösungsvorschlag für diese Aufgabe dar.

```
/*****************************************************************************/
/*        Beispiel fuer den Einsatz von Threads und Semaphoren:              */
/*        Es wird die Zeit für MAL Kontextwechsel zwischen Threads gemessen. */
/*                                                                           */
/* Complilieren und Linken mit:                                              */
/*      gcc -Wall -D_REENTRANT thread_sem.c -lpthread -o thread_sem           */
/*****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/time.h>

#define MAL             10000
#define MIO             1000000
#define INTER_THREAD    0

/* Definition von Semaphoren als !globale! Variablen.                        */

sem_t go_on1, go_on2;



/* Diese Funktion enthaelt den Programmcode, der als Thread ausgefuehrt wird.*/

void myThread(int param)
{
      int i;

      for (i=0; i<MAL; i++) {
            if (sem_wait(&go_on1) < 0) {
                  perror("sem_wait");
                  pthread_exit(0);
            }

            if (sem_post(&go_on2) < 0) {
                  perror ("sem_post");
                  pthread_exit(0);
            }
      }
}



int main()
{
      pthread_t tid;
      long status;
      int i;
      struct timeval start, end;
      struct timezone zone;
```

```
/* sem_int(...) erzeugt den Semaphor go_on1 und initialisiert ihn mit 0.     */

      if (sem_init(&go_on1, INTER_THREAD, 0) < 0) {
            perror("sem_init");
            return(1);
      }

      if (sem_init(&go_on2, INTER_THREAD, 0) < 0) {
            perror("sem_init");
            return(1);
      }

/* pthread_create(...) erzeugt einen neuen Thread, dessen Programmzaehler    */
/* auf die Startadresse der Funktion myThread gesetzt wird. Als Parameter    */
/* kann ein vier-Byte langer Wert uebergeben werden (hier 0 aber sonst     */
/* auch oft die Adresse einer Struktur oder eines Feldes.                    */
/* tid enthaelt nach erfolgreicher Ausfuehrung die Thread ID des erzeugten   */
/* Threads.                                                                  */

      if (pthread_create(&tid, NULL, (void *)myThread, (void *)0) == -1) {
            perror("create");
            return(1);
      }

/* Der Aufruf sem_post(&go_on1) entspricht der V(...)-Funktion und          */
/* korrespondiert zum Aufruf von sem_wait(&go_on1...) in myThread.          */

      gettimeofday(&start, &zone);
      for (i=0; i<MAL; i++) {
            if (sem_post(&go_on1) < 0) {
                  perror ("sem_post");
                  return(1);
            }

            if (sem_wait(&go_on2) < 0) {
                  perror("sem_wait");
                  return(1);
            }
      }
      gettimeofday(&end, &zone);

      printf("Zeit fuer einen Durchlauf: %6.1f us\n",
            ((end.tv_sec - start.tv_sec)*MIO
            + (end.tv_usec - start.tv_usec))/(float)MAL);

/* pthread_join(tid, &status) blockiert solange, bis der Thread mit der     */
/* Thread-ID tid beendet wurde. In status wird der Status der Beendigung    */
/* mitgeteilt.                                                               */

      if (pthread_join(tid, (void **)&status) == -1) {
            perror("join");
            return(1);
      }

      return(0);
}
```