2. Test aus Systemnahe Programmierung 2003-06-18

KNr. MNr. Zuname, Vorname

Ges.)(60) 1.)(30) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Shared Memory and Semaphore

Auf einem Shared Memory soll ein Datenarray verwaltet werden, dessen Elemente zum Speichern von Knoten einer (ungeordneten) einfach verketteten Liste verwendet werden. Die nicht belegten Arrayelemente werden ebenfalls in einer einfach verketteten Liste (Free List) verwaltet.

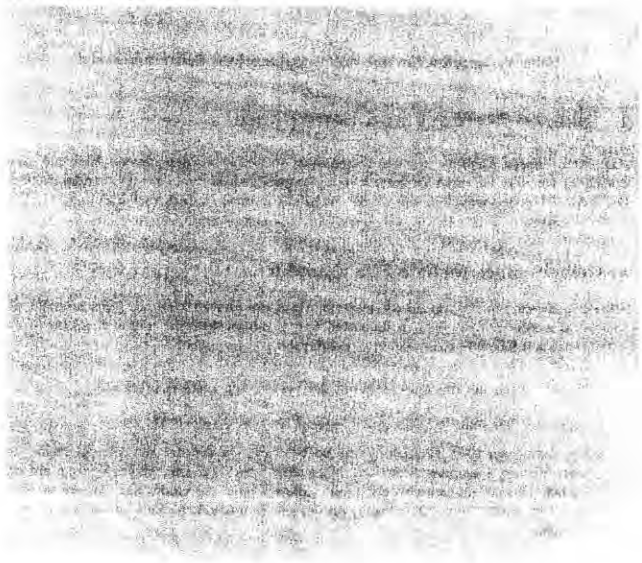
a) Gemeinsames Include-File shared.h

Schreiben Sie eine Definition für die Datenstruktur des Shared Memory (einschließlich der Definitionen aller notwendigen Konstanten und Datenstrukturen). Diese Struktur (siehe Abbildung) soll folgende Komponenten beinhalten:

- Einen Verweis auf das erste Element der verketteten Liste (`ListRoot`).
- Einen Verweis auf das erste Element der Free List (`FreeListRoot`).
- Ein Array von SIZE Elementen zur Speicherung der Knoten der Listen. Dabei hat jeder Knoten folgendes Aussehen: einen Key vom Typ `int`, einen Verweis auf das nächste Listenelement (wird auch zum Kennzeichnen des Endes der Liste verwendet) und Nutzdaten vom Typ `T_Data`, um dessen Definition Sie Sich nicht zu kümmern brauchen.

List Root		
Free List Root		
Key	Data	Next


```
/* ------------------ shared.h ------------------- */

#define SIZE 20     /* max. Anzahl von Listenelementen am Shared Memory */

typedef int  T_Data;   /* Typ fuer Nutzdaten */

/* Shared Memory Strukturen */
```

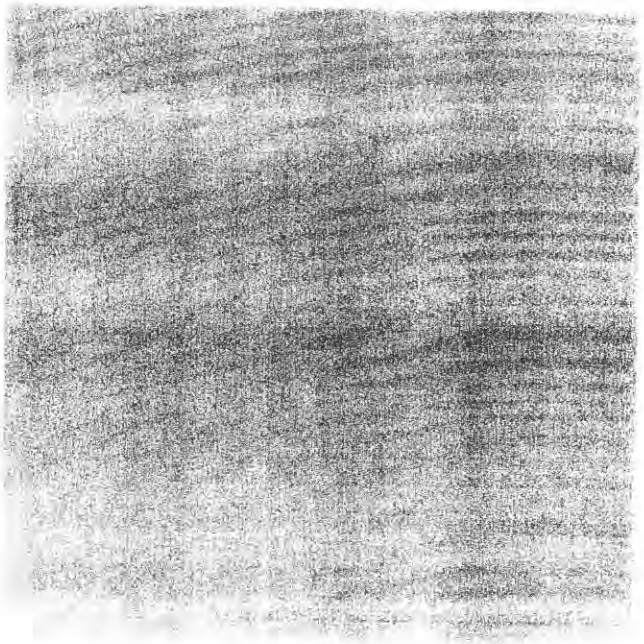


b) Includes und globale Variablen

In den Teilaufgaben (c) und (d) sollen Sie Funktionen schreiben, die exklusive Operationen auf dem Shared Memory ausführen. Die Funktionen sind so zu implementieren, dass jede in einem eigenen Prozesskontext ausgeführt werden kann. Die Funktion *init()* legt

das Shared Memory an und initialisiert es. Die Funktion *insert()* fügt ein neues Element in die Liste am Shared Memory ein. Es kann angenommen werden, dass *init()* aufgerufen wird, bevor der erste Aufruf von *insert()* erfolgt. Gleichzeitige Aufrufe von *insert()* müssen jedoch von Ihnen synchronisiert werden. Zur Fehlerbehandlung verwenden Sie die Funktion `error_exit(char *s)`, die die Fehlermeldung s ausgibt und anschließend eine Programmtermination bewirkt.

Geben Sie hier *#include*-Files und globale Definitionen an, die für die Programmierung der Funktionen *init()* und *insert()* notwendig sind.

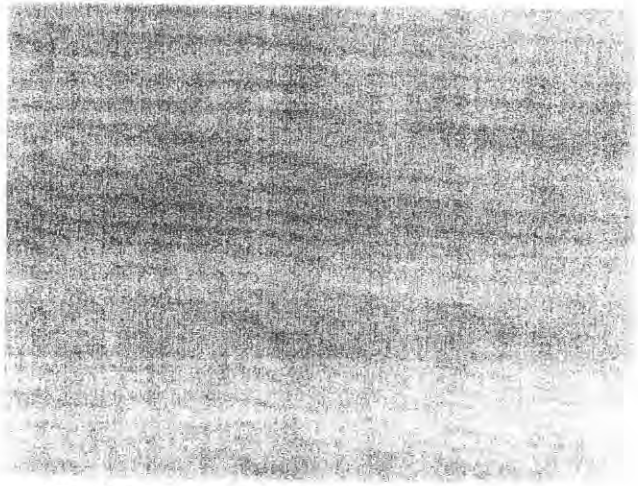


c) Funktion init()

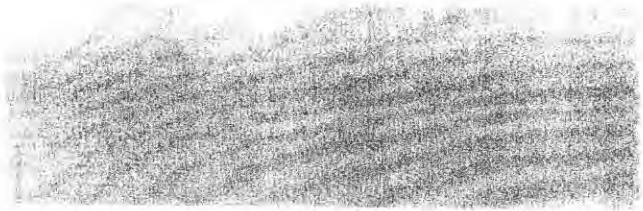
Schreiben Sie eine Funktion *init()* zum Anlegen und Initialisieren des Shared Memory sowie sonst notwendiger Synchronisationskonstrukte.

```
static void init()

{
   /* lokale Variablen */
```

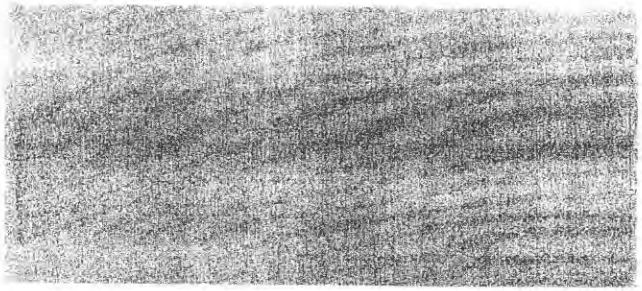
```
   /* Initialisierungen */
```



```
   /* Initialisierung der Daten des Shared Memory */
```



```
    /* Aufraeumen */
```



```
}
```

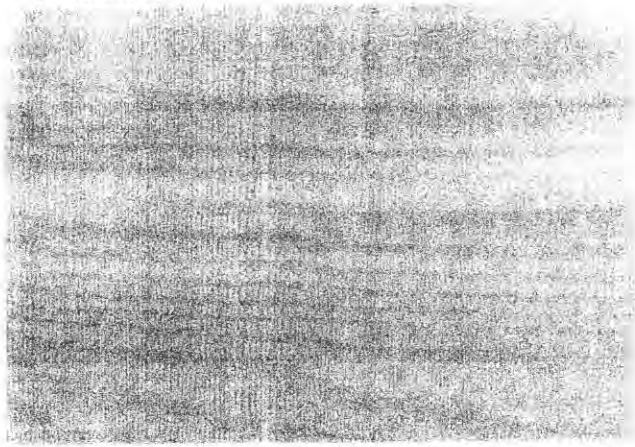
d) Funktion insert()

Schreiben Sie die Funktion *insert()*, die einen neuen Knoten an vorderster Stelle in der Liste auf dem Shared Memory einträgt. Sie können annehmen, dass beim Aufruf von *insert()* Knoten in der Free List vorhanden sind. Der Funktion *insert()* werden Schlüssel und Nutzdaten als Parameter übergeben. Der Code der Funktion hat für die korrekte Synchronisation bei Mehrfachaufrufen zu sorgen. Beachten Sie, dass die Funktion in einem eigenen Prozesskontext laufen können soll.

```
static void insert(const int key, const T_Data data)

{

    /* lokale Variablen */
```



```
    /* Initialisierungen */
```



```
    /* Einfuegen des Knotens in die Liste*/
```

2 Fork, Exec und Unnamed Pipes (30)

Schreiben Sie ein Program `try`, welches mit zwei Kommandos als Parameter aufgerufen wird:

```
try  "cmd1 [args ...]"  "cmd2 [args ...]"
```

Das Programm `try` soll das erste Kommando ausführen und alle `stderr`-Ausgaben dieses Kommandos auf `stdin` des zweiten Kommandos umleiten. Sie können davon ausgehen, dass die beiden Kommandos, wie im Beispiel angegeben, als *String* übergeben werden, d.h. jedes Kommando wird gemeinsam mit all seinen Parametern als **ein** String an das Program `try` übergeben. Beispiel für einen Aufruf von `try`:

```
./try  "ls yy.txt"  "sed -e 's/^/Error(try): /' | cat >> errlog.txt"
```

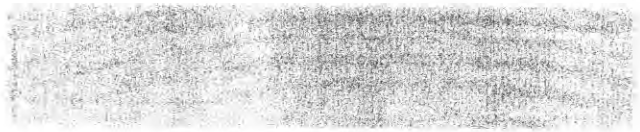
Im fehlerfreien Fall soll das Programm `try` mit dem Wert `EXIT_SUCCESS` beendet und alle Ressourcen freigegeben werden. Überprüfen Sie, ob genau zwei Kommandoaufrufe übergeben werden (ansonsten die Funktion `usage()` aufrufen).

Erzeugen Sie für die Ausführung von jedem Kommando *einen eigenen* Kindprozess. Verwenden Sie zum Lösen von Punkt a) dieses Beispiels die Funktionen `create_redirect_process()` bzw. `double_wait()`, sowie die Funktionen `BailOut()` and `usage()`. Das Freigeben der Ressourcen im Fehlerfall passiert durch die Funktion `BailOut()`.

- Die Funktion `void BailOut(const char *szMessage)` gibt eine Fehlermeldung, welche den String `szMessage` enthält, aus und terminiert das Programm mit Exit-Status `EXIT_FAILURE` nach Freigabe aller Ressourcen (zB: Pipes, Kindprozesse). Diese Funktion ist *nicht* zu implementieren!
- Die Funktion `void usage(void)` gibt eine entsprechende Usage-Meldung aus und terminiert das Programm mit Exit-Code `EXIT_FAILURE`. Diese Funktion ist *nicht* zu implementieren!
- Die Funktion `pid_t create_redirect_process(const char *szCmd, int anPipe[2], int nPipeChannel, int nFileDes)` startet den in `szCmd` übergebenen Kommandoaufruf als Kindprozess, wobei dessen Filedeskriptor `nFileDes` auf den Kanal `anPipe[nPipeChannel]` der Pipe umgeleitet wird und das verbleibende Ende der Pipe geschlossen wird. Der Rückgabewert ist die Prozess-ID des Kindprozesses (`-1` im Fehlerfall). Zur Bestimmung des Filedeskriptors steht die Funktion `int fileno(FILE *stream)` zur Verfügung. Diese Funktion ist in Punkt b) des Beispiels zu implementieren.
- Die Funktion `int double_wait(pid_t pid1, pid_t pid2)` wartet auf das Terminieren der beiden Kindprozesse `pid1` und `pid2`. Es wird nur auf diejenigen `pid_t`-Werte gewartet, die ungleich `-1` sind. Der Rückgabewert ist im fehlerfreien Fall 0, andernfalls `-1`. Diese Funktion ist in Punkt c) des Beispiels zu implementieren.

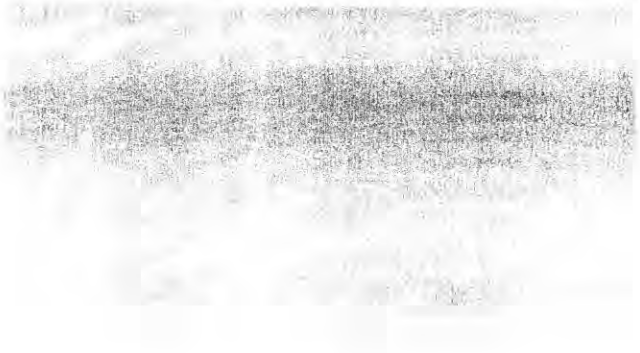
a) Ergänzen Sie das Programmgerüst von `try` (14)

```
/* ********************************************************** includes ***/
        /* includefiles muessen nicht angegeben werden */
/* ********************************************************** globals ***/
```



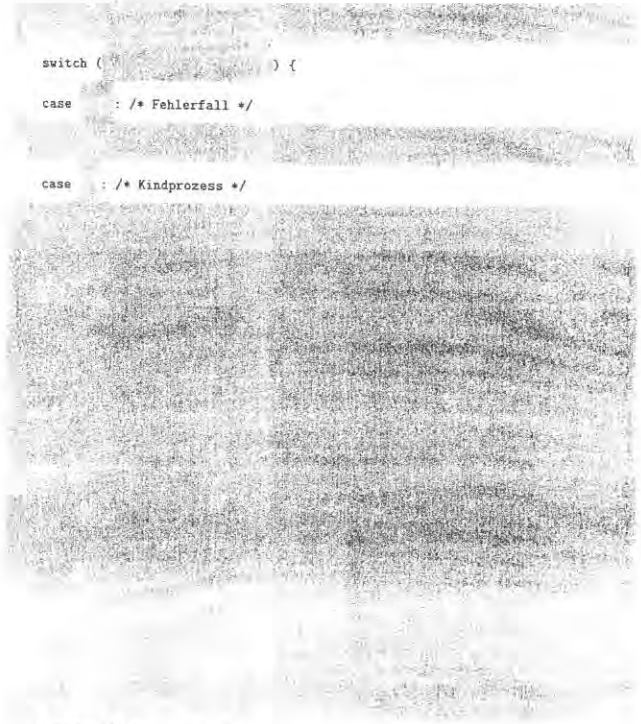
```
/* ******************************************************* prototypes ***/
void BailOut(const char *szMessage);
void usage(void);
/* ******************************************************** functions ***/
int main(int argc, char** argv)
{
    /* Test auf korrekte Parameteranzahl */
```

```
    /* Unnamed Pipe anlegen, Prozesse erzeugen (create_redirect_process) */
```



b) Ergänzen Sie die Funktion `pid_t create_redirect_process()` (8)

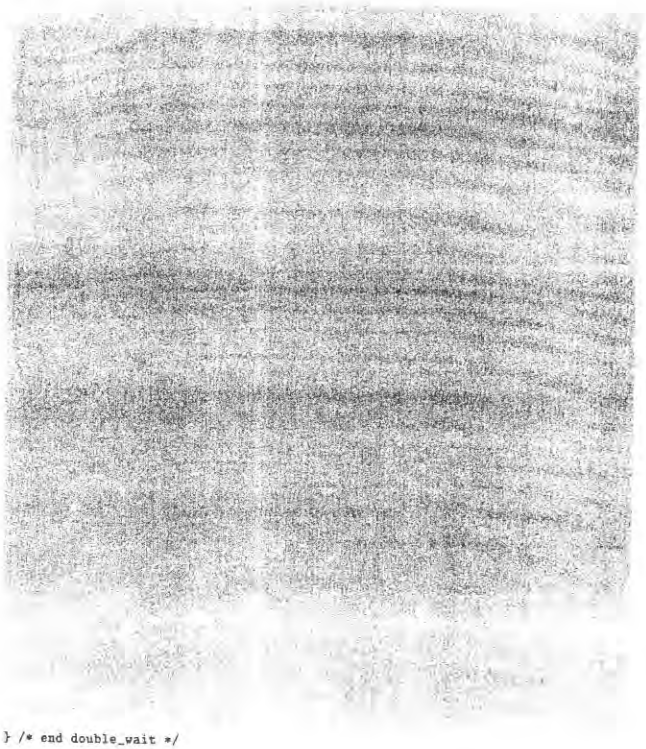
```
pid_t create_redirect_process(const char *szCmd, int anPipe[2],
                              int nPipeChannel, int nFileDes)
{
```



```
    switch (                    ) {
    case     : /* Fehlerfall */

    case     : /* Kindprozess */

    default: /* Elternprozess */
```


c) Ergänzen Sie die Funktion `double_wait()` (8)

```
int  double_wait(pid_t pid1, pid_t pid2)
{
```



```
} /* end double_wait */
```

2. Nachtragstest aus Systemprogrammierung 2001-06-12

KNr. MNr. Zuname, Vorname

Ges. (60) 1.)(30) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Fork, Exec und Unnamed Pipes (30)

Schreiben Sie ein Programm sanbox, das mit dem Namen eines auszuführenden Programmes und optionalen Parametern aufgerufen wird:

SYNOPSIS:

```
sandbox prog [args ...]
```

Das Programm *sandbox* führt das Programm *prog* mit den angegebenen Argumenten aus und verarbeitet dabei alle Fehlerausgaben, die *prog* während der Ausführung auf *stderr* schreibt.

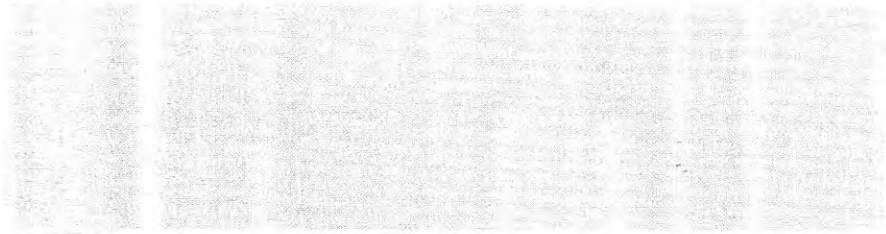
Realisieren Sie *sandbox* unter Verwendung von *fork*, *exec* und einer Unnamed Pipe unter Beachtung folgender Punkte:

- Prüfen Sie ob mindestens ein Argument angegeben wurde.
- Erzeugen Sie eine Unnamed Pipe.
- Generieren Sie einen Kindprozess und lenken Sie die Fehlerausgabe von *prog* auf diese Pipe um und lesen Sie die Fehlermeldungen im Vaterprozess zeilenweise von dieser Pipe. Sie können dabei annehmen, dass keine Fehlermeldung inklusive Newline mehr als STRLEN Zeichen enthält.
- Der Vaterprozess verarbeitet die Fehlermeldungen von *prog*, indem er für jede Fehlermeldung die Funktion `(void) process_message(char *text)` aufruft und die Fehlermeldung als Argument übergibt. Die Funktion *process_message* brauchen Sie nicht zu implementieren.
- Der Vaterprozess muss vor seiner Beendigung auf die Termination des Kindprozesses warten.

- Verwenden Sie zur Fehlerbehandlung die Funktion `(void) error_exit(char *text)` die einen Fehlertext auf *stderr* ausgibt und anschließend mit *exit()* terminiert, bzw. die Funktion `usage(void)`, die eine Usage-Meldung ausgibt und anschließend mit *exit()* terminiert.

Tragen Sie Ihre Lösung in das auf den folgenden Seiten gegebene Programmgerüst ein. Includefiles und die Definitionen der beiden Funktionen zur Fehlerbehandlung brauchen Sie nicht anzugeben.

```
/* ********************************************************* includes ***/
        /* includefiles muessen nicht angegeben werden */
/* ********************************************************* defines ***/

#define STRLEN 80

/* ********************************************************* globals ***/
```



```
/* ********************************************** prototypes ***/
void error_exit(const char *szMessage);
void process_message(const char *szMessage);
void usage(void);
/* ********************************************** functions ***/

int main (int argc, char **argv)
{
```


```
switch (                    )
{
        /* fork failed */
    error_exit("cannot fork");

        /* child process */
```

```
        /* parent process */
```



```
        /* wait for termination of child process */
```

```
        return 0;
    } /* end switch */
} /* end main */
```

2 Shared Memory (30)

Implementieren sie zwei Prozesse *Schreiber* und *Leser*. Der Prozess Leser soll hierbei vom Prozess Schreiber Integer-Zufallszahlen über einen Ringpuffer erhalten und diese auf `stdout` ausgeben.

Ein Ringpuffer ist ein Speicherbereich bestehend aus einer vorgegebenen Anzahl von Speicherzellen, wobei diese Speicherzellen "möglichst parallel" von einem Schreibprozess beschrieben und von einem Leseprozess ausgelesen werden können. Dabei darf der Schreibprozess solange Datenelemente sequentiell in den Speicherbereich schreiben, bis alle Speicherzellen besetzt sind. Dass heißt, der **Schreibprozess wird nur dann blockiert**, wenn alle Speicherzellen belegt sind. Speicherzellen werden wiederum frei, indem der Leseprozess die besetzten Speicherzellen sequentiell ausliest. Der **Leseprozess wird nur dann blockiert**, wenn keine neuen Datenelemente in dem Speicherbereich zur Verfügung stehen. Bei Programmstart beginnen Schreib- und Lesevorgang am Anfang des Speicherbereichs (d.h. mit der ersten Speicherzelle). Wird das Ende des Speicherbereichs erreicht, werden Schreib- und Lesevorgang wiederum am Anfang des Speicherbereichs fortgesetzt.

Der Ringpuffer soll mithilfe eines Shared Memory realisiert werden, das ein Integerarray der Größe `MAX_BUF` beinhaltet. Der Zugriff auf den Ringpuffer soll mit Sequencer und/oder Eventcounts synchronisiert werden.

Ergänzen Sie die angegebenen Programmfragmente. Beachten Sie dabei bitte:

- Es gibt genau zwei Prozsse, einen Leser und einen Schreiber, die auf den Ringpuffer gemeinsam zugreifen.
- Legen Sie nur soviele Sequencer und/oder Eventcounts an, wie Sie **unbedingt** zur Synchronisation benötigen. Beachten Sie dabei jedoch, dass Sie den Zugriff auf den Ringpuffer nicht unnötig einschränken (z.B. entspricht ein abwechselndes Schreiben und Lesen einer Zahl nicht der Funktionalität eines Ringpuffers)!
- Sorgen Sie dafür, dass das Anlegen des Shared Memory und der Synchronisationskonstrukte derart erfolgt, dass die Prozesse Schreiber und Leser in beliebiger Reihenfolge gestartet werden können.
- Eine Argumentbehandlung ist ausnahmsweise **nicht** erforderlich.
- Verwenden Sie zum Erzeugen der Zufallszahlen die Funktion `int random(void)`.
- Verwenden Sie für Fehlerausgaben die Funktion (muss **nicht** selbst implementiert werden!) `void error_exit(const char * szMsg)`. Diese Funktion gibt die als Parameter übergebene Fehlermeldung (`szMsg`) entsprechend den Übungsrichtlinien aus, entfernt alle angelegten Resourcen und beendet den Prozess (mit einem negativen Returnwert).

Gemeinsames Headerfile

```
/* Includes brauchen nicht angegeben werden */
/* Defines */
#define MAX_BUF 5
```

```
/* Shared Memory Struktur */
```

Schreiber

```
/* Includes brauchen nicht angegeben werden */
/* Globale Variablen */
const char *szCommand = "<not yet set>";
```

```
int main(int argc, char **argv) {

    /*** Variablen deklarieren ***/


    szCommand = argv[0];
```

```
    /*** Anlegen und Attachen des Shared Memories ***/
```

```
    /*** Anlegen der Sequencer und/oder Eventcounts ***/
```

```
while( 1 ) {
    /*** Synchronisiertes Schreiben eines neuen
         Zufallswerts in den Ringpuffer (Endlosschleife) ***/
```



```
}
```

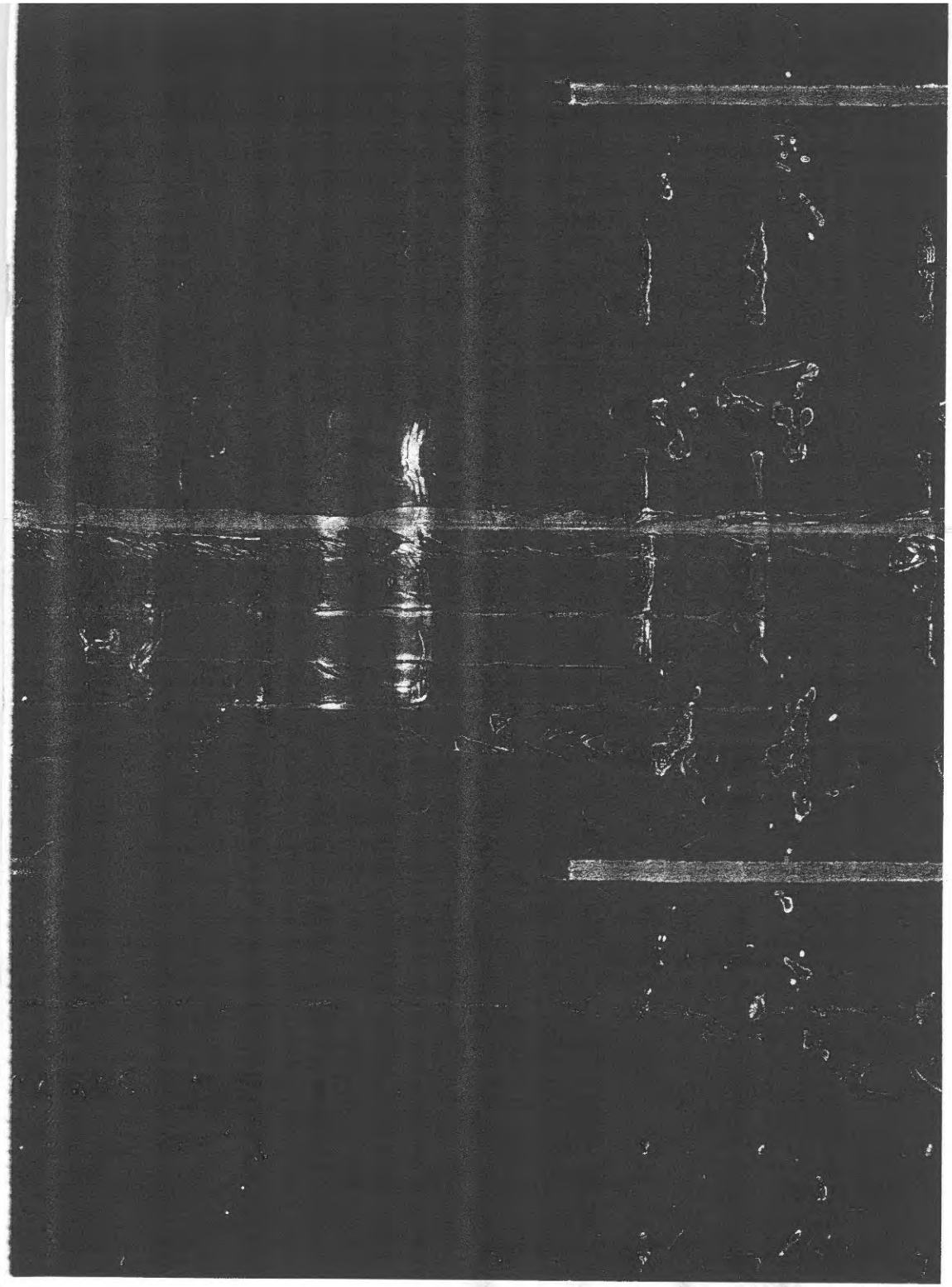
Leser

```
/* Includes brauchen nicht angegeben werden */
/* Globale Variablen */
const char *szCommand = "<not yet set>";
```

```
int main(int argc, char **argv) {
    /*** Variablen deklarieren ***/
```

```
    szCommand = argv[0];
    /*** Anlegen und Attachen des Shared Memories ***/
```

```
    /*** Anlegen der Sequencer und/oder Eventcounts ***/
```



```
while( 1 ) {
    /*** Synchronisiertes Lesen der naechsten Zahl aus dem
         Ringpuffer und Ausgabe auf stdout (Endlosschleife) ***/

}
```

2. Test aus Systemprogrammierung 2001-06-06

KNr. MNr. Zuname, Vorname

Ges.)(60) 1.)(30) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Explizite Synchronisation und Shared Memory (30)

Ein Programmpaket zum Testen von Schedules besteht aus einem Programm *Generator*, das zu untersuchende Schedules generiert und auf ein Shared Memory schreibt, und einem Programm *Tester*, das ein Schedule vom Shared Memory liest und auf seine Durchführbarkeit untersucht. *Tester*-Prozesse werden wiederholt gestartet und dürfen parallel laufen.

a) Headerfile (4)

Ergänzen Sie das Headerfile `schedule.h`, das vom *Generator* und dem *Tester* inkludiert wird und die Definitionen für gemeinsame Datenstrukturen und Synchronisation enthält. Definieren Sie alle nötigen Schlüssel zum Anlegen des Shared Memories und der/des Semaphor(e/s).

Geben Sie weiters die Definition der Datenstrukturen für das Shared Memory an. Definieren Sie zuerst den Typ `T_task`, der zwei Integerwerte `P` und `C` (Periode und Computation Time von Tasks) enthält. Definieren Sie unter Verwendung von T_task die Shared Memory Struktur `T_SharedMem`. `T_SharedMem` enthält zwei Integerwerte `nr` und `anz` (Nummer des Schedules und Anzahl der enthaltenen Tasks), sowie ein Array `tasks` der Länge `MAX_TASKS` mit Elementen vom Typ `T_task`.

```
/* ------------------- schedule.h ------------------- */

/* includes (nicht noetig anzugeben!) */

#define MAX_TASKS 10

/* Defines fuer Shared Memory */
```



```
/* Defines fuer das/die Semaphor(e) */
```

```
/* Shared Memory Strukturen */

/* Strukturtyp T_task */
```

```
/* Typ T_SharedMem */
```

b) Generator (13)

Realisieren Sie den *Generator*. Das Programm verwendet die Funktion `get_problem()` (Deklaration siehe Listing) zum Generieren eines neuen zu untersuchenden Scheduling-Datensatzes und schreibt diesen auf das Shared Memory. Der Vereinfachung halber wird bei `get_problem()` auf Fehlerbehandlung verzichtet.

1. Erzeugen Sie das/die benötigte(n) Semaphor(e).
2. Legen Sie den Shared Memory-Bereich an. Geben Sie eine Fehlermeldung aus, wenn das Shared Memory bereits existiert.
3. Stellen Sie mit Hilfe der Synchronisationskonstrukte sicher, dass jeder Datensatz von genau einem Tester gelesen wird, bevor ein neuer Datensatz auf das Shared Memory geschrieben wird.
4. Verwenden Sie die Semaphoroperationen der *sem182*-Library.

5. Behandeln Sie Fehler durch Aufruf von `error_exit()`. Die Funktion gibt die übergebene Fehlermeldung aus, löscht alle Ressourcen und beendet dann das Programm.

```
/* includes (nicht noetig anzugeben!) */

#include "schedule.h"

 /* globale Variablendeklarationen */

extern void error_exit(char *error_message);

extern void get_problem(T_SharedMem *MemPtr);

int main(int argc, char *argv[])
{
 /* Variablendeklarationen */

  /* init Semaphor(e) */

  /* Anlegen und Einhaengen des Shared Memories */
```



```
  /* Erzeugen der zu testenden Schedules und Uebergabe ueber Shared Mem. */

  for(;;){

  }
}
```

c) Tester (13)

Ergänzen Sie das Programmskelett für den *Tester*. Das Programm wartet auf einen verfügbaren Schedule-Eintrag, liest diesen aus dem Shared Memory, testet ihn mit Hilfe der Funktion `is_schedulable()` und gibt eine Meldung "is schedulable" oder "is not schedulable" auf dem Bildschirm aus. Anschließend gibt das Programm die Ressourcen frei und

terminiert. Gehen Sie wie folgt vor:

1. Verbinden Sie das Programm mit den Mechanismen zur Interprozesskommunikation.
2. Lesen Sie vom Shared Memory und testen Sie einen Schedule-Eintrag mit der Funktion `is_schedulable()`. Die Funktion `is_schedulable()` liefert den Wert `1` für durchführbare und den Wert `0` für undurchführbare Schedules. Beachten Sie, dass die Berechnung der Funktion `is_schedulable()` lange dauert und daher *außerhalb* des kritischen Abschnitts erfolgen soll. Achten Sie dabei auf die Erhaltung der Datenkonsistenz.
3. Geben Sie die Nummer des Schedules sowie eine entsprechende Meldung, "is schedulable" bzw. "is not schedulable" aus. Im Anschluss sind die Ressourcen freizugeben und das Programm zu beenden.
4. Verwenden Sie wie beim Generator die Funktion `error_exit()` zur Fehlerbehandlung und Ressourcenfreigabe.

```
/  includes (nicht noetig anzugeben!) */
   clude "schedule.h"
  /* globale Variablendeklarationen */
```

```
extern void error_exit(char *error_message);
extern int is_schedulable(T_SharedMem schedule);
int main(int argc, char **argv){
  /* Variablemdeklarationen */

  /* Holen und Einhaengen des Shared Memories */
```



```
/* Holen des/der Semaphor(e)*/
```

```
/* Lesen vom Shared Memory und Schedule testen */
```

```
/* Ausgabe und Cleanup */

}
```


2 Fork & Exec (30)

Implementieren Sie ein Programm *verbinde* welches zwei weitere Programme startet und die Ausgabe des ersten Programmes als Eingabe an das zweite Programm leitet. Das Programm *verbinde* hat zwei Parameter, wobei jeder Parameter den Namen eines Programmes mit seinen Argumenten enthält. In *verbinde* werden die beiden Programme gestartet, wobei vorher der Ausgabestream (`stdout`) des ersten Programmes an den Eingabestream (`stdin`) des zweiten Programmes umgeleitet wird. Im Fehlerfall soll *verbinde* durch die Funktion `error_exit()` terminieren. Vorher sollen aber noch alle verwendeten Recourcen frei gegeben werden. Im Fehlerfreien Fall soll das Programm mit dem Rückgabewert `0` beendet werden.

Ein Aufruf von `verbinde "cat READ.ME" "more"` bewirkt das Gleiche, wie die Eingabe von `cat READ.ME | more` in einer Shell (die Anführungszeichen bewirken, dass »cat READ.ME« als ein Parameter »argv[1]« übergeben wird). Sie können annehmen, dass ein Benutzer von *verbinde* die Parammeter auch wirklich mit Hochkommata klammert.

Verwenden Sie zum Lösen von Punkt a) dieses Beispiels die Funktionen `spawn()` und `clean()` sowie die Prozeduren `error_exit()` bzw. `usage )`.

- Die Funktion `spawn()` ist folgendermaßen deklariert:

  ```
  pid_t spawn(const char *pProg, const FILE *pRedirect, int anPipe[]);
  ```

  Diese Funktion erzeugt einen Kindprozess und startet das Program `pProg` in einer Shell. Die Funktion verbindet zuvor den über den Parameter `pRedirect` angegebenen Eingabe-, bzw. Ausgabestream (`stdin` bzw. `stdou`) des Kindprozesses mit dem passenden Ende der Pipe `anPipe` und schließt die vom Kindprozess nicht benötigten Filedeskriptoren. Tritt ein Fehler vor/bei der Aufspaltung mit `fork` auf, liefert die Funktion `spawn()` den Wert `-1` zurück. Wenn der Kindprozess fehlerfrei erzeugt wurde soll `spawn` die PID des erzeugten Kindprozesses zurückliefern.

- Die Funktion `clean()` hat folgende Eigenschaften:

  ```
  int clean(pid_t pid1, pid_t pid2);
  ```

  Diese Funktion wartet auf die Terminierung zweier Prozesse mit der Prozess-ID `pid1` bzw. `pid2`. Soll nur auf die Terminierung eines Prozesses gewartet werden, so ist für den jeweils anderen Parameter anstelle einer gültigen Prozess-ID der Wert `-1` anzugegeben. Im Fehlerfall, bzw. falls ein Kind `EXIT_FAILURE` zurückliefert, gibt die Funktion den Wert `-1` zurück, ansonsten den Wert `0`.

- Die Deklaration der Routine `error_exit` ist:

  ```
  void error_exit(char *pStr);
  ```

  Sie gibt eine Fehlermeldung mit Programmnamen und dem String `pStr` am Bildschirm aus und terminiert mit Fehlercode.

- Die Deklaration der Routine `usage()` ist:

  ```
  void usage(void);
  ```

Sie erklärt, wie ein richtiger Aufruf aussehen muss und terminiert mit Fehlercode.

a) Ergänzen Sie das Programmgerüst von verbinde (15)

```
int main(int argc, char *argv[])
{ /* Defines und Includes nicht notwendig!                    *
   * Variablen deklarieren und, falls notwendig, initialisieren */
```

```
  /* Testen ob wirklich 2 Argumente vorhanden sind. Sonst -> usage() */
```

```
  /* Unnamed Pipe anlegen und Fehlerbehandlung */
```

```
  /* Beide Programme jeweils durch Aufruf von spawn() starten *
   * Fehlerabfragen nicht vergessen!                          */
```



```
  /* Im Vaterprozess nicht benoetigte Filedeskriptoren schliessen */
```

```
  /* Warten auf die Terminierung der Kindprozesse */
```

```
  return SUCCESS;
}
```

b) Ergänzen Sie die Funktion spawn() (15)

Beachten Sie bitte zum Lösen dieser Aufgabe den folgenden Hinweis:

- Um herauszufinden, ob der Stream `stdin` oder `stdout` mit der Pipe verbunden werden sollte, verwenden Sie die Funktion `int fileno(FILE *stream)`. Diese Funktion liefert Ihnen den Filedeskriptor zum jeweiligen Stream `stream`, d.h. im Fall von `stdin` den Wert 0 und im Fall von `stdout` den Wert 1. Sie können annehmen, dass für den Streamparameter niemals ein anderer Parameter als `stdin` bzw. `stdout` angegeben wird.
- Die Funktion `spawn()` soll -1 zurückgeben, wenn ein Fehler im Vaterprozess aufgetreten ist. Tritt der Fehler im Kindprozess auf soll mit `exit(EXIT_FAILURE)` terminiert werden.

Implementieren Sie nun die Funktion `spawn()`.

```
pid_t spawn(const char *pProg,const FILE *pRedirect,int anPipe[])
{
  /* Variable(n) deklarieren */
  :

  /* Kindprozess starten und PID ermitteln */
  switch(          ){
    /* Fehlerfall */
      :

    /* Kindprozess */
      :

      /* Umlenken von stdin bzw. stdout an das jeweilige Pipeende   *
       * Fehlerabfrage nicht vergessen (Fehlercode an Vaterprozess) */
```



```
      /* Im Kindprozess nicht benoetigte Filedeskriptoren schliessen */

      /* In einer Shell das geforderte Programm starten */

      /* Fehlercode an Vaterprozess zurueckliefern */

    /* Vaterprozess */
      :

    break;
  }
  /*** Prozess ID des Kindprozesses zurueckliefern ***/

}
```

2. Test aus Systemprogrammierung 2001-01-17

KNr. MNr. Zuname, Vorname

Ges.)(60) 1.)(35) 2.)(25) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Explizite Synchronisation und Shared Memory (35)

A, B und C seien Matrizen. Es soll auf einem Computer das Matrizenprodukt $C := A \cdot B$ errechnet werden.

$$\begin{pmatrix} c_{0,0} & c_{0,1} & \dots & c_{0,(l-1)} \\ c_{1,0} & c_{1,1} & \dots & c_{1,(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(n-1),0} & c_{(n-1),1} & \dots & c_{(n-1),(l-1)} \end{pmatrix} :=$$

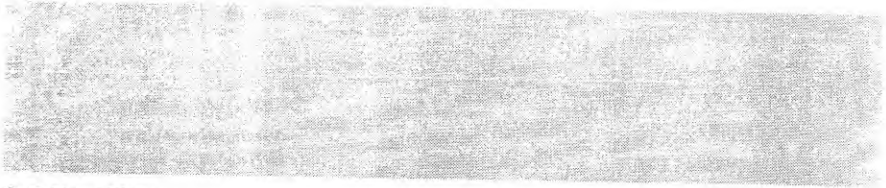
$$:= \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,(m-1)} \\ a_{1,0} & a_{1,1} & \dots & a_{1,(m-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1),0} & a_{(n-1),1} & \dots & a_{(n-1),(m-1)} \end{pmatrix} \cdot \begin{pmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,(l-1)} \\ b_{1,0} & b_{1,1} & \dots & b_{1,(l-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(m-1),0} & b_{(m-1),1} & \dots & b_{(m-1),(l-1)} \end{pmatrix}$$

$$\begin{aligned} c_{0,0} &:= a_{0,0}b_{0,0} + a_{0,1}b_{1,0} + \ldots + a_{0,(m-1)}b_{(m-1),0} \\ c_{0,1} &:= a_{0,0}b_{0,1} + a_{0,1}b_{1,1} + \ldots + a_{0,(m-1)}b_{(m-1),1} \\ &\vdots \\ c_{(n-1),(l-1)} &:= a_{(n-1),0}b_{0,(l-1)} + a_{(n-1),1}b_{1,(l-1)} + \ldots + a_{(n-1),(m-1)}b_{(m-1),(l-1)} \end{aligned}$$

Weil das eine aufwendige Berechung ist, sollen mehrere Prozesse (auf mehreren Prozessoren laufend) zusammen das Ergebnis ermitteln (siehe Skizze). **Achtung:** Nur die Programmteile in den grau unterlegten Kartuschen müssen ausgeführt werden.

```
/* Weitere defines*/
```

```
/* Shared Memory Struktur */

typedef struct   {
```



```
} T_SharedMem;
```

b) Elternprozeß (16)

Implementieren Sie die Routine
`T_Matrix *init_sharedmem(int n, int m, int l, T_Matrix *A, T_Matrix *B)`
des Elternprozesses, der die shared Variablen anlegt und richtig initialisiert.

Diese Routine liefert einen Zeiger auf die Ergebnismatrix C (deren Elemente noch nicht berechnet sind). Im Fehlerfall wird einfach ein Nullzeiger zurueckgegeben (kein freigeben der bereits allocierten Resourcen; keine Fehlermeldung).

Vaterprozess

1. Erzeugen Sie das/die benötigte(n) Semaphor(e).
2. Legen Sie den Shared Memory-Bereich an.
3. Initialisieren Sie wenn notwendig die Variablen.

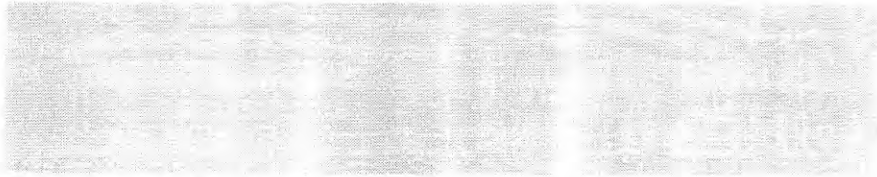
```
/* includes (nicht noetig anzugeben!) */
#include "matrix.h"
/* Globals */
```

```
T_Matrix *init_sharedmem(int n, int m, int l, T_Matrix *A, T_Matrix *B){
```

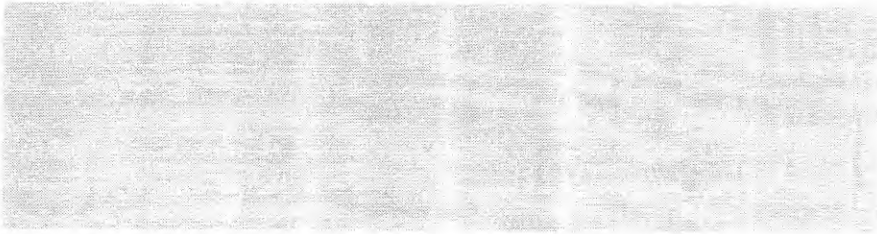


```
/* Variablendeklarationen */
```

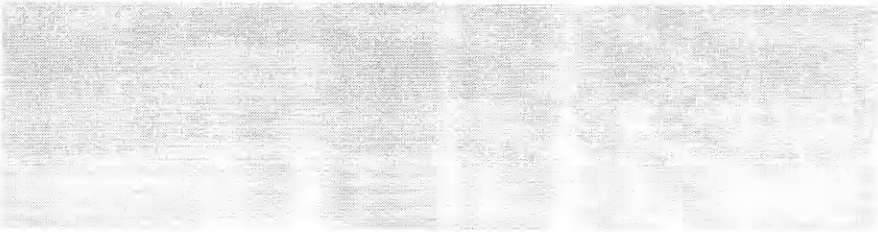
```
/* init Semaphor(e) */
```



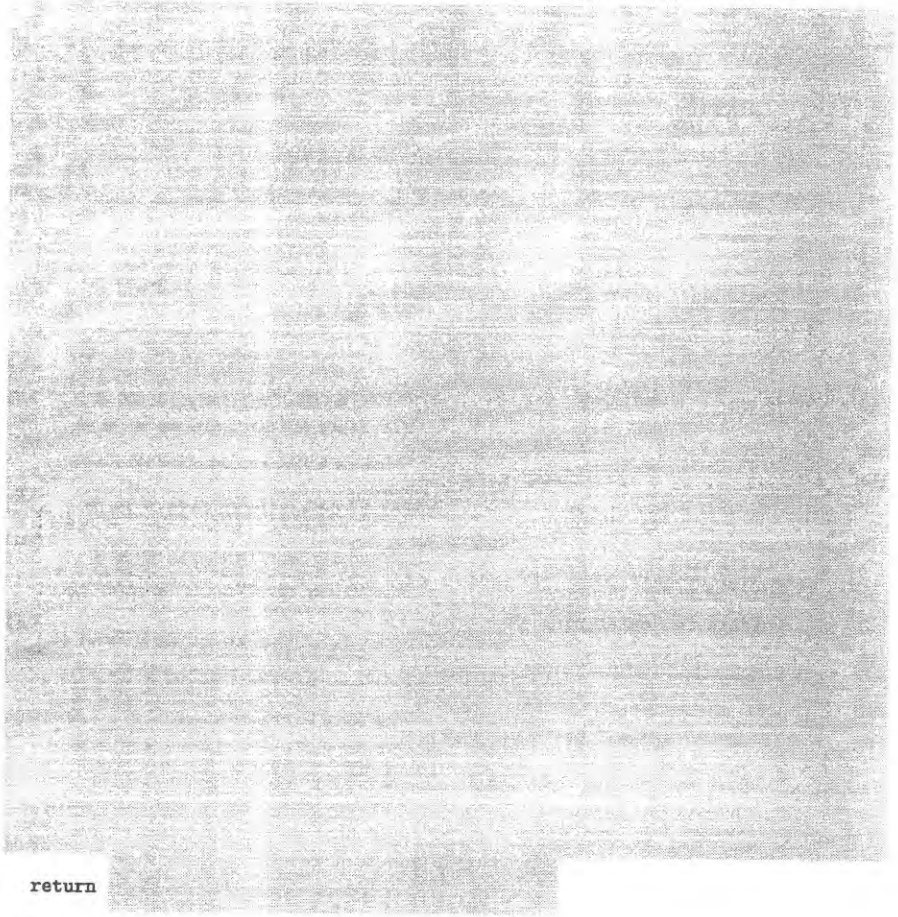
```
/* Anlegen des Shared Memories */
```



```
/* Einhaengen des Shared Memories */
```



```
/* Kopieren & init. der Uebergabewerte in den shared memory Bereich */
```



```
  return
}
```

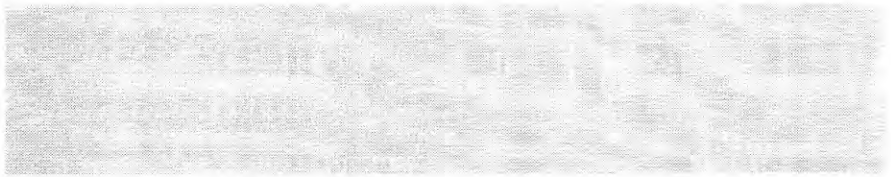
c) Kindprozess (16)

Implementieren Sie den Kindprozess: Tritt ein Fehler auf, ist das Programm mit der Routine `error_exit("<message>");` zu beenden, welche alle allocierten Resourcen wieder freigibt, eine Fehlermeldung ausgibt und mit einem Fehlercode terminiert, ansonsten ist der Rückgabewert 0. Die Routine `error_exit("<message>");` muss nicht implementiert werden.

Kindprozess

1. Ermitteln, welches Element $c_{i,j}$ noch von keinem anderen Prozeß berechnet wird.
2. j um eins erhöhen, wenn $j \geq l$ ist: $j = 0$ setzen und i um eins erhöhen. Stellen Sie sicher, dass während des Schreibens im Shared Memory kein anderer Prozess auf die gleiche Speicherzelle zugreift.
3. $c_{i,j}$ berechnen: $c_{i,j} = \sum_{k=0}^{m-1} a_{i,k} \cdot b_{k,j}$ Und in C abspeichern.
4. Wenn noch weitere Elemente von C berechnet werden müssen, wieder bei Punkt 1 beginnen.

```
/* includes (nicht noetig anzugeben!) */
#include "matrix.h"
/* Globals */
```

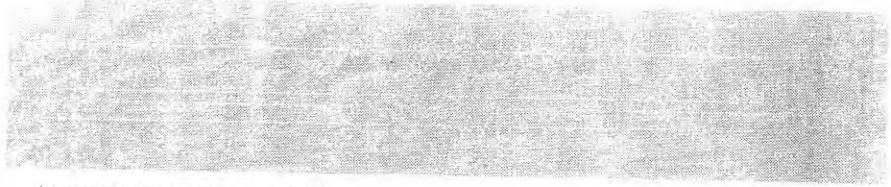


```
int main(int argc, char **argv){
  /* Variablendeklarationen */
```

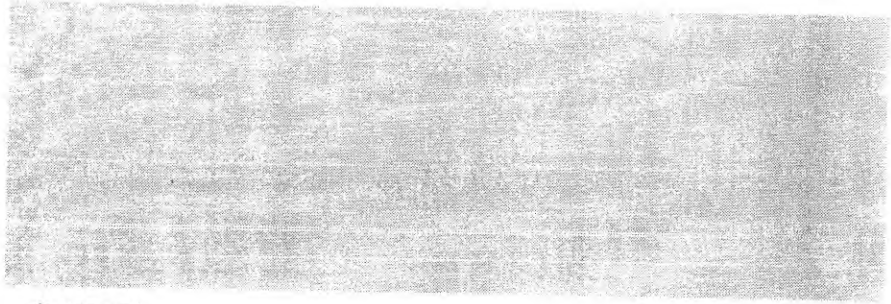
```
  /* holen des Shared Memories */
```

```
  /* Einhaengen des Shared Memories */
```

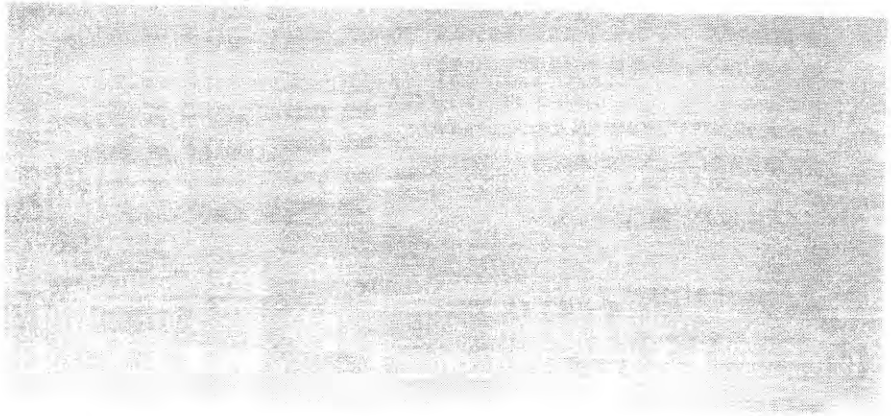
```
/* holen des/der Semaphor(e)*/
```



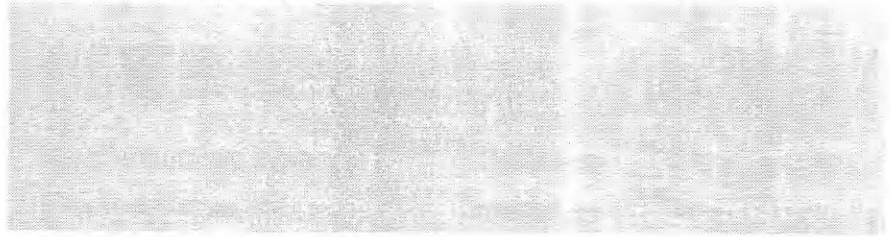
```
/* holen der Indizes i,j */
```



```
/* loop */
while(                    ){
  /* increment i & j */
```




```
  /* berechnen von c[i,j] */
```



```
  /* holen der Indizes i,j */
```

```
  }
  /* terminieren */
  return 0;
}
```

2 Fork, Exec und Unnamed Pipes (25)

Schreiben Sie ein Programm `sandbox`, das mit dem Namen eines auszuführenden Programmes und optionalen Parametern aufgerufen wird:

```
sandbox prog [args ...]
```

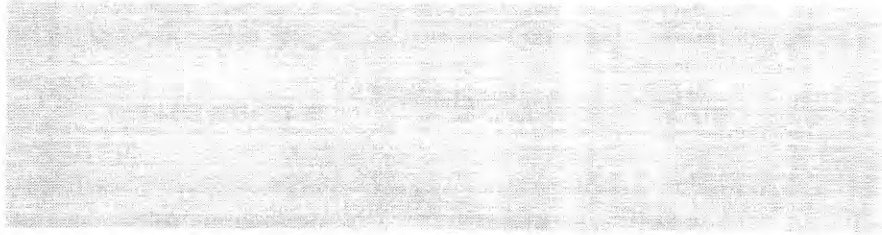
Das Programm `sandbox` führt das Programm `prog` mit den angegebenen Argumenten aus und verarbeitet dabei alle Fehlerausgaben, die `prog` während der Ausführung auf die Standardfehlerausgabe (`stderr`) schreibt.

Realisieren Sie `sandbox` unter Verwendung von fork, exec und einer Unnamed Pipe unter Beachtung folgender Punkte:

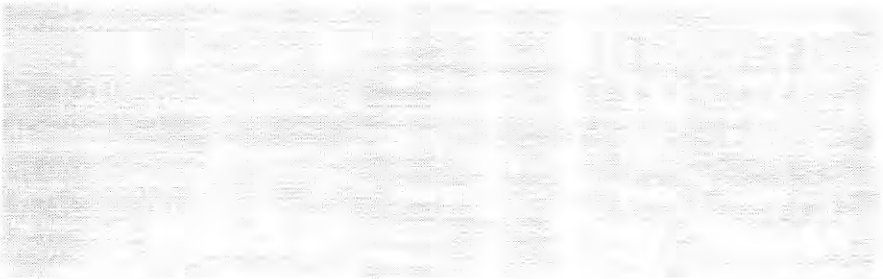
- Prüfen Sie, ob mindestens ein Argument zu `sandbox` angegeben wurde. Eine Argumentbehandlung mit `getopt()` muss nicht implementiert werden.
- Erzeugen Sie die Unnamed Pipe.
- Generieren Sie einen Kindprozess, lenken Sie die Fehlerausgabe von `prog` auf diese Pipe um und lesen Sie die Fehlermeldungen im Vaterprozess zeilenweise von dieser Pipe.
- Der Vaterprozess liest die Fehlermeldungen von *prog* aus der Pipe. Sollte eine Fehlermeldung mehr als `STRLEN` Zeichen enthalten, wird diese ignoriert. Ansonsten wird die Fehlermeldung mit der Funktion `(void) process_message(char *text)` verarbeitet. Die Funktion `process_message()` brauchen Sie nicht zu implementieren.
- Der Vaterprozess muss vor seiner Beendigung auf die Termination des Kindprozesses warten.
- Verwenden Sie zur Fehlerbehandlung die Funktion `(void) error_exit(char *text)`, die einen Fehlertext auf `stderr` ausgibt, bzw. die Funktion `(void) usage(void)`, die eine Usage-Meldung ausgibt. Beide Funktionen löschen die angelegten Ressourcen und terminieren mit `exit()`.

Tragen Sie Ihre Lösung in das auf den folgenden Seiten gegebene Programmgerüst ein.

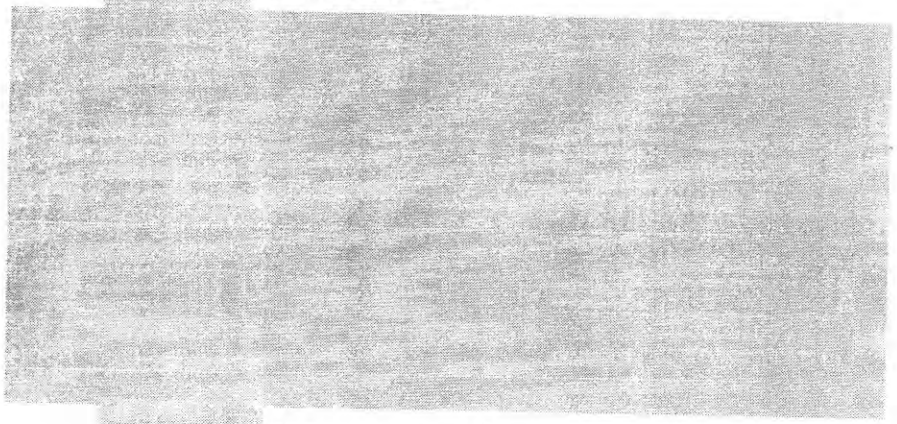

```
/* ********************************************************** includes ***/

        /* includefiles muessen nicht angegeben werden */

/* ********************************************************** defines ***/

#define STRLEN 80

/* ********************************************************** globals ***/
```



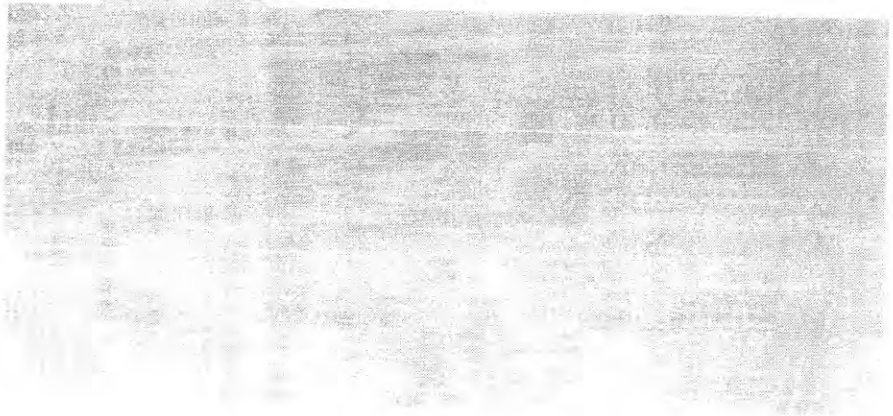
```
/* ********************************************************** prototypes ***/

void error_exit(const char *szMessage);

void process_message(const char *szMessage);

void usage(void);

/* ********************************************************** functions ***/

int main (int argc, char **argv)

{
```

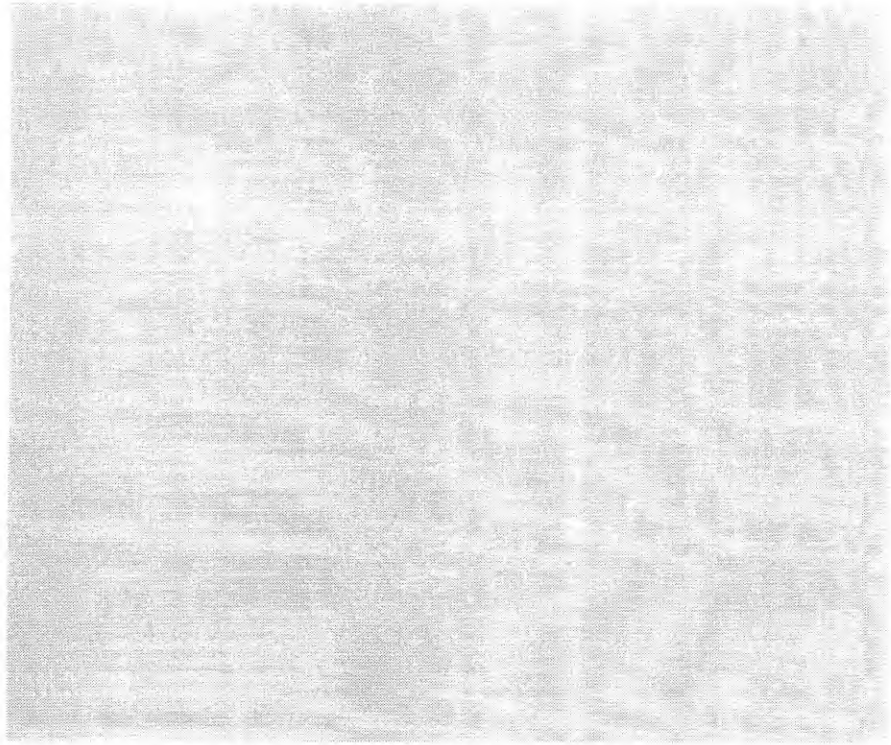


```
switch (                    )
{
        /* fork failed */
    error_exit("cannot fork");

        /* child process */
```

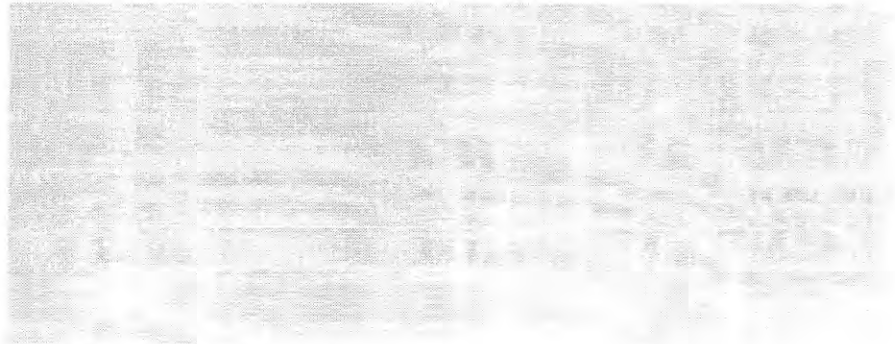


```
        /* parent process */
```





```
        /* wait for termination of child process */
```



```
    } /* end switch */


} /* end main */
```

2. Test aus Systemprogrammierung 2000-06-07

KNr. MNr. Zuname, Vorname

Ges.)(60) 1.)(30) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 `fork()`, `exec()`, `wait()` und Pipes (30)

Schreiben Sie eine Funktion namens `do_work()`, welche den Namen eines Programms über ein Argument namens `szCommand` und einen Filenamen über ein Argument namens `szFileName` übergeben bekommt. Die Funktion `do_work()` soll das Programm `szCommand` so starten, dass das Programm `szCommand` seine Standardeingabe (`stdin`) von einem File namens `szFileName` bezieht.

Für den Fall, dass in der Funktion `do_work()` ein Fehler auftritt, soll die Funktion einen negativen Returnwert zurückliefern. – Im fehlerfreien Fall liefert `do_work()` den Wert `0`.

Gehen Sie hierzu bei der Implementierung der Funktion `do_work()` wie folgt vor:

- Legen Sie eine Pipe an.
- Erzeugen Sie einen Kindprozess, der für die Ausführung des Programms `szCommand` zuständig ist.
- Lenken Sie im Kindprozess die Standardeingabe auf das entsprechende Ende der Pipe um, und schliessen Sie das andere Ende.
- Führen Sie im Kindprozess das Programm `szCommand` aus.
- Schliessen Sie im Vaterprozess das nicht benötigte Ende der Pipe und kopieren Sie den Inhalt des Files `szFileName` in die Pipe, damit es vom Kindprozess gelesen werden kann.
- Schliessen Sie im Vaterprozess nach Beendigung des Kopiervorganges alle noch offenen Pipeenden und Files und warten Sie auf die Terminierung des Kindprozesses. – Der Exitstatus des Kindprozesses wird hierbei **nicht** benötigt.

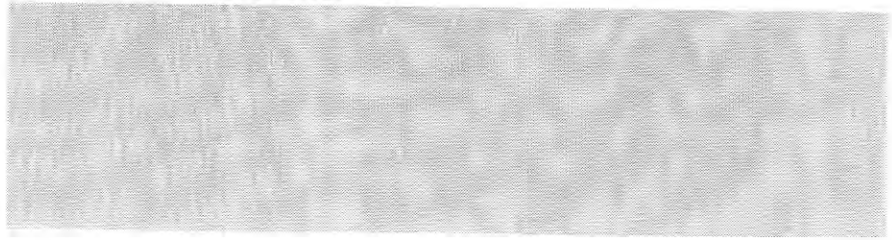
Verwenden Sie für die Implementierung von `do_work()` folgende Funktionen (diese Funktionen müssen **nicht** von Ihnen implementiert werden):

`void error_return(const char * szMsg)` gibt die als Parameter übergebene Fehlermeldung (`szMsg`) entsprechend den Übungsrichtlinien aus, entfernt alle angelegten Resourcen und beendet die aktuelle Funktion (mit einem negativen Returnwert). Für den Fall, dass zum Zeitpunkt des Aufrufs von `error_return()` bereits ein Kindprozess erzeugt wurde, wartet die Funktion `error_return()` auf die Terminierung des selbigen.

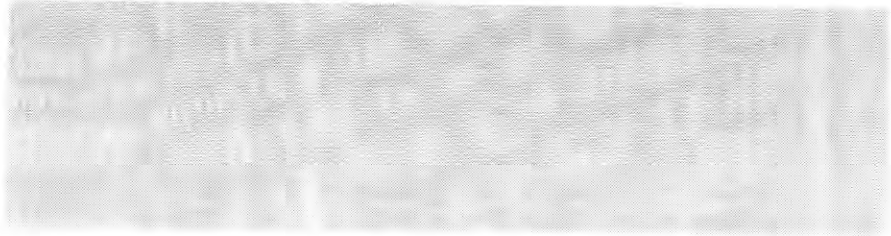
`void error_exit(const char * szMsg)` gibt ebenfalls eine Fehlermeldung (`szMsg`) entsprechend den Übungsrichtlinien aus und beendet den gesamten Prozess (mit einem negativen Exitcode), **ohne** die angelegten Resourcen zu entfernen. – Auf etwaige Kindprozesse wird **nicht** gewartet.

`int copy(FILE * psDst, FILE * psSrc)` kopiert die im Quellfile (`psSrc`) enthaltenen Daten ins Zielfile (`psDst`). Die Funktion liefert im Fehlerfall `EOF` zurück. – Die Funktion `copy()` schliesst weder das Quellfile noch das Zielfile.

```
int do_work(const char * szCommand, const char * szFileName)

{

   /*** Variablen deklarieren ***/
```



```
   /*** Unnamed Pipe anlegen ***/
```

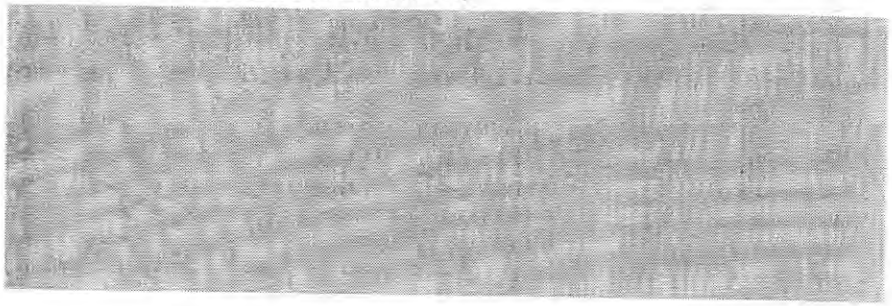


```
/*** Kindprozess erzeugen ***/
[redacted]
{
    [redacted]: /* Fehler */
    [redacted]
    break;

    [redacted]: /* Kind */
    /*** stdin umlenken ***/
    [redacted]
    /*** Pipeende(n) schliessen ***/
    [redacted]
    /*** Programm ausfuehren ***/
    [redacted]
    break;
```

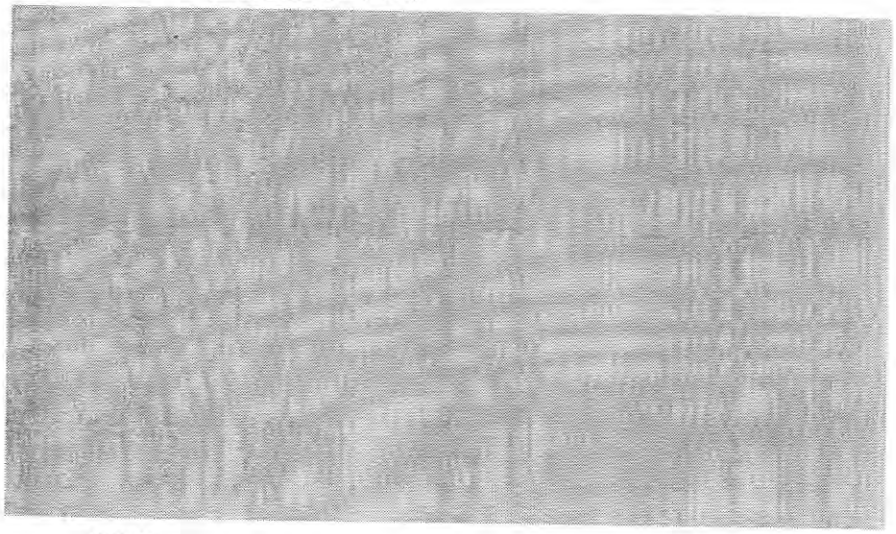


```
    [redacted]: /* Vater */
    /*** Pipeende(n) schliessen ***/
    [redacted]
    /*** Filepointer erzeugen ***/
    [redacted]
    /*** File oeffnen ***/
    [redacted]
    /*** File kopieren ***/
    [redacted]
```

```
/*** Resourcen schliessen ***/
```



```
/*** Warten auf Kind ***/
```



```
    break;
  }

  /*** Erfolgreicher Returnwert ***/
```

```
}
```


2 Ringpuffer (30)

Implementieren sie zwei Prozesse *Schreiber* und *Leser*. Der Prozess Leser soll hierbei vom Prozess Schreiber Integer-Zufallszahlen über einen Ringpuffer erhalten und diese auf `stdout` ausgeben.

Ein Ringpuffer ist ein Speicherbereich bestehend aus einer vorgegebenen Anzahl von Speicherzellen, wobei diese Speicherzellen "möglichst parallel" von einem Schreibprozess beschrieben und von einem Leseprozess ausgelesen werden können. Dabei darf der Schreibprozess solange Datenelemente sequentiell in den Speicherbereich schreiben, bis alle Speicherzellen besetzt sind. Dass heißt, der **Schreibprozess wird nur dann blockiert**, wenn alle Speicherzellen belegt sind. Speicherzellen werden wiederum frei, indem der Leseprozess die besetzten Speicherzellen sequentiell ausliest. Der **Leseprozess wird nur dann blockiert**, wenn keine neuen Datenelemente in dem Speicherbereich zur Verfügung stehen. Bei Programmstart beginnen Schreib- und Lesevorgang am Anfang des Speicherbereichs (d.h. mit der ersten Speicherzelle). Wird das Ende des Speicherbereichs erreicht, werden Schreib- und Lesevorgang wiederum am Anfang des Speicherbereichs fortgesetzt.

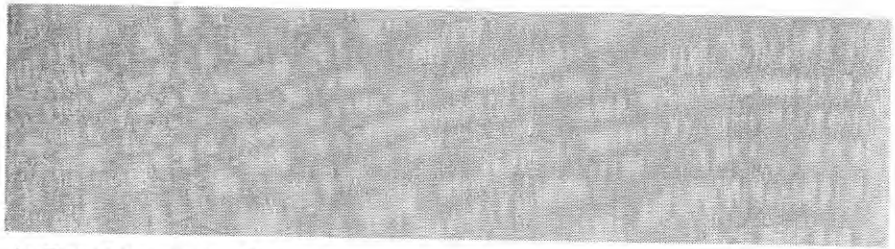
Der Ringpuffer soll mithilfe eines Shared Memory realisiert werden, das ein Integerarray der Größe `MAX_BUF` beinhaltet. Der Zugriff auf den Ringpuffer soll mit Sequencer und/oder Eventcounts synchronisiert werden.

Ergänzen Sie die angegebenen Programmfragmente. Beachten Sie dabei bitte:

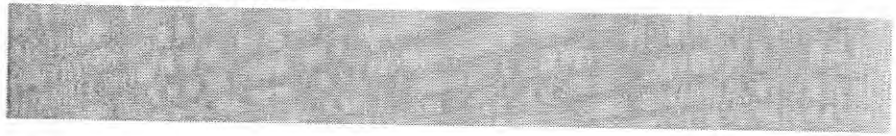
- Es gibt genau zwei Prozsse, einen Leser und einen Schreiber, die auf den Ringpuffer gemeinsam zugreifen.
- Legen Sie nur soviele Sequencer und/oder Eventcounts an, wie Sie **unbedingt** zur Synchronisation benötigen. Beachten Sie dabei jedoch, dass Sie den Zugriff auf den Ringpuffer nicht unnötig einschränken (z.B. entspricht ein abwechselndes Schreiben und Lesen einer Zahl nicht der Funktionalität eines Ringpuffers)!
- Sorgen Sie dafür, dass das Anlegen des Shared Memory und der Synchronisationskonstrukte derart erfolgt, dass die Prozesse Schreiber und Leser in beliebiger Reihenfolge gestartet werden können.
- Eine Argumentbehandlung ist ausnahmsweise **nicht** erforderlich.
- Verwenden Sie zum Erzeugen der Zufallszahlen die Funktion `int random(void)`.
- Verwenden Sie für Fehlerausgaben die Funktion (muss **nicht** selbst implementiert werden!) `void error_exit(const char * szMsg)`. Diese Funktion gibt die als Parameter übergebene Fehlermeldung (`szMsg`) entsprechend den Übungsrichtlinien aus, entfernt alle angelegten Resourcen und beendet den Prozess (mit einem negativen Returnwert).

Gemeinsames Headerfile

```
/* Includes brauchen nicht angegeben werden */
/* Defines */
#define MAX_BUF 5
```

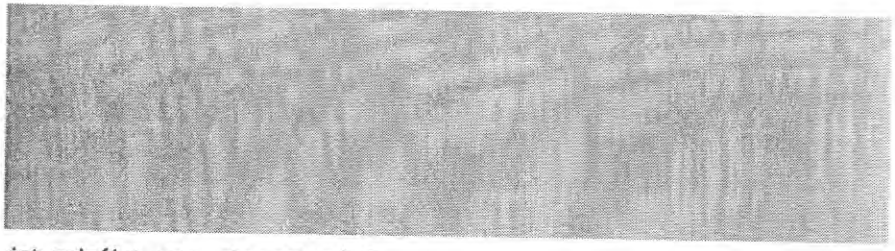


```
/* Shared Memory Struktur */
```

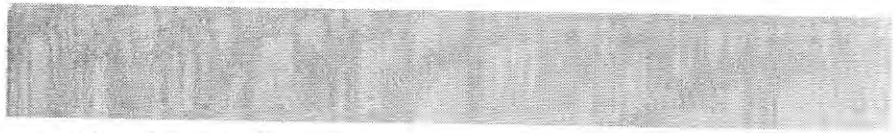


Schreiber

```
/* Includes brauchen nicht angegeben werden */
/* Globale Variablen */
const char *szCommand = "<not yet set>";
```



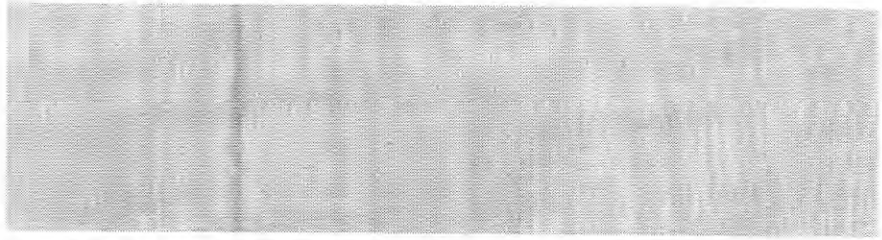
```
int main(int argc, char **argv) {
    /*** Variablen deklarieren ***/
```



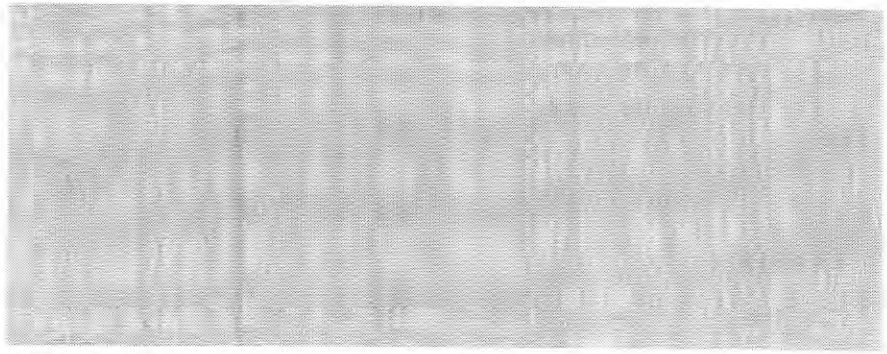
```
    szCommand = argv[0];
```



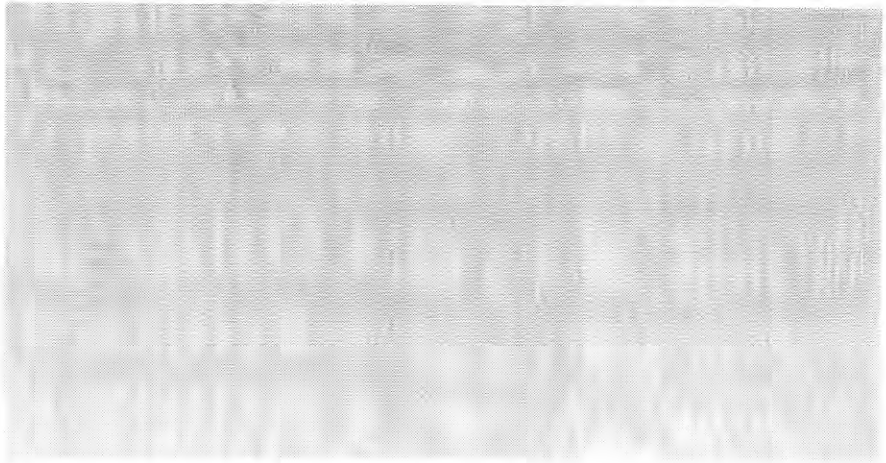
```
    /*** Anlegen und Attachen des Shared Memories ***/
```



```
    /*** Anlegen der Sequencer und/oder Eventcounts ***/
```



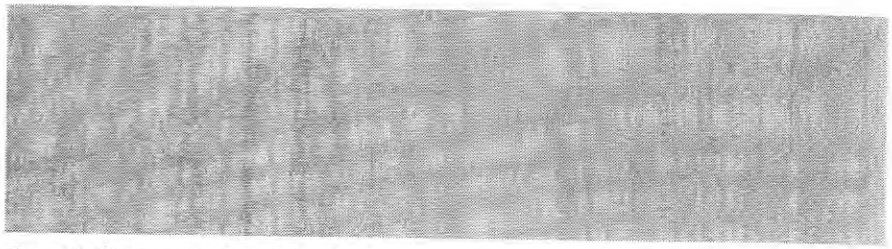
```
while( 1 ) {
    /*** Synchronisiertes Schreiben eines neuen
         Zufallswerts in den Ringpuffer (Endlosschleife) ***/
```



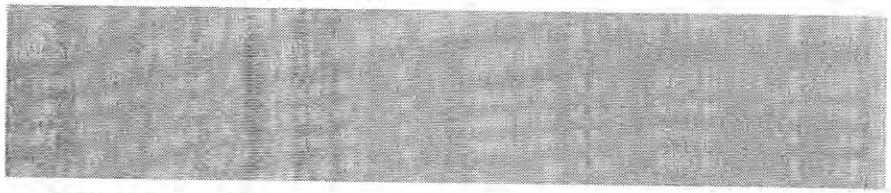
```
}
```

Leser

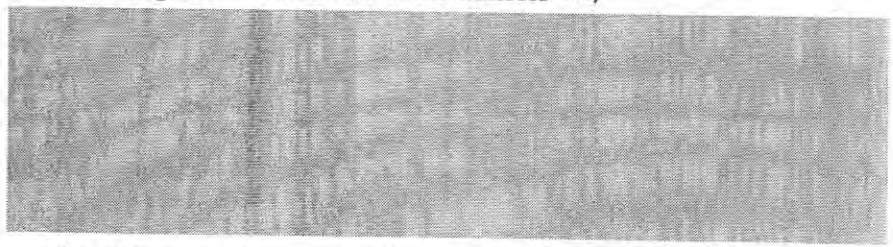
```
/* Includes brauchen nicht angegeben werden */
/* Globale Variablen */
const char *szCommand = "<not yet set>";
```



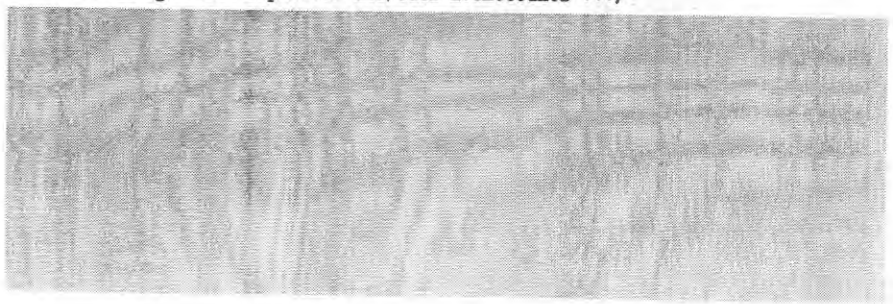
```
int main(int argc, char **argv) {

    /*** Variablen deklarieren ***/
```

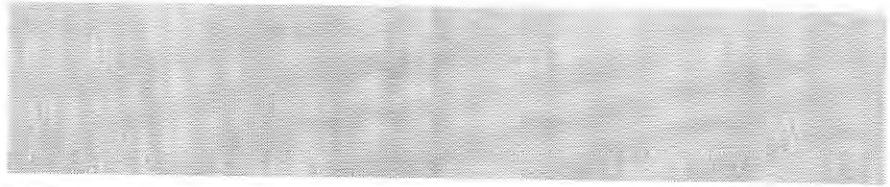


```
    szCommand = argv[0];

    /*** Anlegen und Attachen des Shared Memories ***/
```

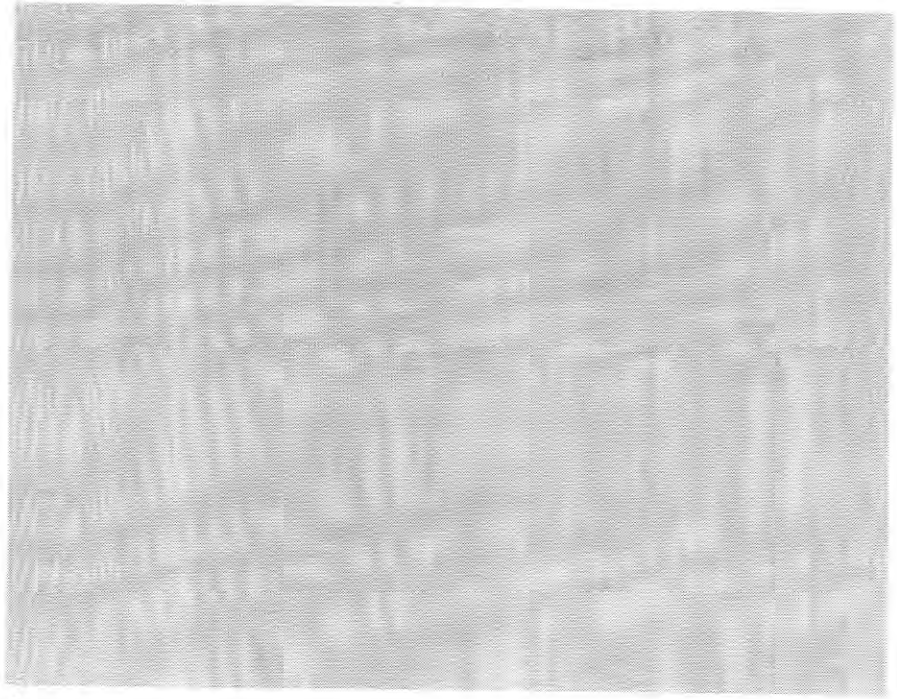


```
    /*** Anlegen der Sequencer und/oder Eventcounts ***/
```





```
while( 1 ) {
    /*** Synchronisiertes Lesen der naechsten Zahl aus dem
         Ringpuffer und Ausgabe auf stdout (Endlosschleife) ***/
```



```
}
```

2. Test aus Systemprogrammierung 2000-01-19

KNr. MNr. Zuname, Vorname

Ges.)(60) 1.)(30) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Explizite Synchronisation und Shared Memory (30)

Gegeben sei ein Spieleprogramm bestehend aus einem Serverprozess und einer **vorgegebenen Anzahl** von Spieleprozessen. Die Spieleprozesse sollen in einer streng zyklischen Reihenfolge aufgerufen werden, um einzelne Spielzüge durchzuführen. Pro Spielzug erfolgt eine Interaktion zwischen dem jeweilig aktiven Spieleprozess und dem Serverprozess über ein Shared Memory. Die streng zyklische Abarbeitungsreihenfolge der Spieleprozesse und die Synchronisation zwischen Spieleprozess und Serverprozess soll mit den Synchronisationskonstrukten Sequencer und/oder Eventcounter realisiert werden.

a) Headerfile (4)

Ergänzen Sie das Headerfile `support.h`, das vom Serverprozess und von den Spieleprozessen inkludiert wird. Definieren Sie dazu alle nötigen Keys zum Anlegen des Shared Memories und der Sequencer und/oder Eventcounter. Beachten Sie, dass nur Keys für **unbedingt nötige** Synchronisationskonstrukte, angegeben werden.

Geben Sie weiters die Definition der Shared Memory Struktur an. Das Shared Memory soll zwei Integerwerte `nReqNumber` und `nOk` beinhalten. Definieren Sie dazu ein Typedef `sharedmem_t`, das diese zwei Variablen in einer Struktur beinhaltet.

```
/* ------------------  support.h ------------------ */

/* includes (nicht noetig anzugeben!) */

#define MAX 3

#define ROUND 5

/* Defines fuer Shared Memory */
```



```
/* Defines fuer Sequencer */

/* Defines fuer Eventcounter */

/* Shared Memory Struktur */
```

b) Spieleprozess (14)

Implementieren Sie die Spieleprozesse in einem Programm. Bei jedem Aufruf dieses Programms liefert die vorgegebene Funktion `long process_args(int argc, char **argv)` eine eindeutige Identitätsnummer, die den Spieleprozess identifiziert. Insgesamt gibt es `MAX` Spieleprozesse, wobei die Identitätsnummern in dem Bereich `[0..(MAX-1)]` liegen. Die Spieleprozesse sollen in der Reihenfolge ihrer Identitätsnummern ausgeführt werden, d.h. der Spieleprozess mit der Identitätsnummer 0 beginnt, danach kommt der mit der Nummer 1 etc. Es sollen die in der Konstanten `ROUND` definierte Anzahl von Runden gespielt werden, wobei pro Runde jeder Spieleprozess genau einmal einen Spielzug durchführen soll. Ein Spielzug beinhaltet folgende Aktionen:

1. Sobald der Spieleprozess an der Reihe ist, schreibt er in das Shared Memory Feld `nReqNumber` den Returnwert der vorgegebenen Funktion `int Number(void)`.
2. Der Spieleprozess wartet bis der Serverprozess das Shared Memory Feld `nReqNumber` ausgelesen hat und in das Shared Memory Feld `nOk` einen Wert geschrieben hat.
3. Der Spieleprozess ruft die Funktion `void CheckOk(int nOk)` auf, wobei das Shared Memory Feld `nOk` als Parameter übergeben wird. Danach signalisiert er dem nächsten Spieleprozess, dass er seinen Spielzug beendet hat.

Sind `ROUND` Runden gespielt, ruft der Spieleprozess die vorgegebene Funktion `Free_Resources()` auf und terminiert.

Ergänzen Sie das vorgegebene Programmgerüst des Spieleprozesses. Beachten Sie dabei folgende Punkte:

- Definieren Sie globale Variablen für den Zugriff auf alle nötigen Sequencer und/oder Eventcounter bzw. auf das Shared Memory.

- Die Funktion `long process_args(int argc, char **argv)` führt eine Argumentbehandlung durch und liefert die Identitätsnummer.

  Diese Funktion braucht **nicht** implementiert zu werden!

- Gehen Sie davon aus, dass die Funktion `void Allocate_Resources()` alle nötigen Ressourcen anlegt und die von Ihnen definierten Variablen für den Zugriff auf die Sequencer und/oder Eventcounter und das Shared Memory initialisiert. Die Funktion `void Free_Resources()` löscht wiederum alle Ressourcen.

  Diese Funktionen brauchen **nicht** implementiert zu werden!

- Verwenden Sie zur Synchronisation die in der Funktion `Allocate_Resources()` angelegten Sequencer und/oder Eventcounter.

- Rufen Sie im Fehlerfall die Funktion `void BailOut(const char *szMessage)` auf. Diese Funktion gibt eine Fehlermeldung aus und löscht auch alle Ressourcen.

  Diese Funktion braucht **nicht** implementiert zu werden!

```
/* includes (nicht noetig anzugeben!) */
#include "support.h"
/* Globals */
const char *szCommand = "<not yet set>";

int main(int argc, char **argv) {

    szCommand = argv[0];

              = processargc(argc,argv);    /* liefert Identitaetsnummer */

    AllocateResources(); /* (braucht nicht implementiert zu werden!) */

    /* Schleife zum Abarbeiten der Spielzuege */
```



```
                                  = Number();

        CheckOk(                          );

    Free_Resources();  /* (braucht nicht implementiert zu werden!) */

    exit(EXIT_SUCCESS);

}
```

c) Serverprozess (12)

Implementieren Sie den Serverprozess. Der Serverprozess behandelt die Spielzüge aller Spieleprozesse. Pro Spielzug sollen folgende Aktionen durchgeführt werden:

1. Der Serverprozess wartet bis ein Spieleprozess in das Shared Memory geschrieben hat. Danach ruft er die Funktion `int CheckCommand(int nReqNumber)` auf. Als Parameter soll der Serverprozess das Shared Memory Feld `nReqNumber` übergeben und den Returnwert in das Shared Memory Feld `nOk` speichern.
2. Der Serverprozess signalisiert dem Spieleprozess, dass er auf das Shared Memory geschrieben hat.

Nachdem alle Spieleprozesse `ROUND` Spielezüge durchgeführt haben, soll der Serverprozess durch den Aufruf der Funktion `Free_Resources()` alle Ressourcen löschen. Achten Sie darauf, dass **keine Ressourcen** mehr von Spieleprozessen verwendet werden!

Ergänzen Sie das vorgegebenen Programmgerüst des Serverprozesses. Beachten Sie dabei folgende Punkte:

- Definieren Sie globale Variablen für den Zugriff auf alle nötigen Sequencer und/oder Eventcounter bzw. das Shared Memory.
- Gehen Sie davon aus, dass die Funktion `void Allocate_Resources()` alle nötigen Ressourcen anlegt und die von Ihnen definierten Variablen für den Zugriff auf die Sequencer und/oder Eventcounter und auf das Shared Memory initialisiert. Die Funktion `void Free_Resources()` löscht wiederum alle Ressourcen.

  Diese Funktionen brauchen **nicht** implementiert zu werden!
- Verwenden Sie zur Synchronisation die in der Funktion `Allocate_Resources()` angelegten Sequencer und/oder Eventcounter.
- Rufen Sie im Fehlerfall die Funktion `void BailOut(const char *szMessage)` auf. Diese Funktion gibt eine Fehlermeldung aus und löscht auch alle Ressourcen.

  Diese Funktion braucht **nicht** implementiert zu werden!

```
/* includes (nicht noetig anzugeben!) */
#include "support.h"
/* Globals */
extern const char *szCommand = "<not yet set>";
```



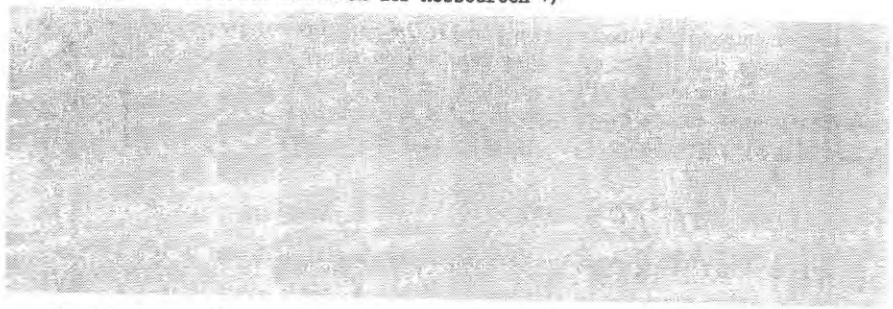
```
int main(int argc, char **argv)
{
```

```
    szCommand = argv[0];
    if (argc != 1) {
        Usage();
    }
    AllocateResources();  /* (braucht nicht implementiert zu werden!) */
```

```
    /* Schleife zum Abarbeiten der Spielzuege */
```

```
                                = CheckCommand(
                                                );
```

```
    /* Synchronisiertes Loeschen der Ressourcen */
```



```
    Free_Resources();  /* (braucht nicht implementiert zu werden!) */

    exit(EXIT_SUCCESS);
}
```


2 Prozesse und Pipes (30)

Implementieren Sie eine Funktion `join()`. Die Funktion hat zwei Parameter, wobei jeder Parameter den Namen eines Programmes samt seiner Argumente enthält. Die Funktion `join()` startet beide Programme, wobei vorher der Ausgabestream `stdout` des ersten Programmes an den Eingabestream `stdin` des zweiten Programmes umgeleitet wird. Im Fehlerfall liefert die Funktion `join()` den Wert `-1` zurück, ansonsten den Wert `0`. Der Funktionsaufruf

```
join("ls -al","grep Dec");
```

bewirkt daher das Gleiche, wie die Eingabe des Shellkommandos

```
ls -al | grep Dec
```

Nehmen Sie zum Lösen von Punkt a) an, dass Ihnen eine Funktion `spawn()` und eine Funktion `join_clean()` zur Verfügung stehen.

- Die Funktion `spawn()` ist folgendermaßen deklariert:

  ```
  pid_t spawn(const char *pProg, const FILE *pRedirect, int anPipe[]);
  ```

  Diese Funktion startet das Program `pProg` als Kindprozess. Die Funktion verbindet zuvor den über den Parameter `pRedirect` angegebenen Eingabe-, bzw. den Ausgabestream (`stdin` bzw. `stdout`) mit dem passenden Ende der Pipe `anPipe` und schließt alle vom Kindprozess nicht benötigten Filedeskriptoren. Im Fehlerfall liefert die Funktion `spawn()` den Wert `-1` zurück, ansonsten die PID des erzeugten Kindprozesses.

- Die Funktion `join_clean()` ist folgendermaßen deklariert:

  ```
  int join_clean(pid_t pid1, pid_t pid2);
  ```

  Diese Funktion wartet auf die Terminierung zweier Prozesse mit der Prozess-ID `pid1`, bzw. `pid2`. Soll nur auf die Terminierung eines Prozesses gewartet werden, so ist für den jeweils anderen Parameter anstelle einer gültigen Prozess-ID der Wert `-1` anzugegeben. Im Fehlerfall, bzw. falls ein Kind `EXIT_FAILURE` zurückliefert, liefert die Funktion den Wert `-1` zurück, ansonsten den Wert `0`.

a) Ergänzen Sie die Funktion `join()` (15)

```
int join(const char *pProg1, const char *pProg2)

{  /*** Variable deklarieren und, falls notwendig, initialisieren ***/
```

```
/*** Unnamed Pipe anlegen und Fehlerbehandlung ***/

/*** Beide Programme jeweils durch Aufruf von spawn() starten ***/
/*** Fehlerabfragen nicht vergessen!                           ***/

/*** Im Vaterprozess nicht benoetigte Filedeskriptoren schliessen ***/

/*** Warten auf die Terminierung der Kindprozesse ***/
/*** Rueckgabe des Funktionswertes                ***/
return join_clean(                          );
}
```


b) Ergänzen Sie die Funktion spawn() (15)

Beachten Sie bitte zum Lösen dieser Aufgabe den folgenden Hinweis:

- Um herauszufinden, ob der Stream `stdin` oder `stdout` mit der Pipe verbunden werden sollte, verwenden Sie die Funktion `int fileno(FILE *stream)`. Diese Funktion liefert Ihnen den Filedeskriptor zum jeweiligen Stream `stream`, d.h. im Fall von `stdin` den Wert `0` und im Fall von `stdout` den Wert `1`. Sie können annehmen, dass für den Streamparameter niemals ein anderer Parameter als `stdin` bzw. `stdout` angegeben wird.

Implementieren Sie nun die Funktion `spawn()`.

```
pid_t spawn(                                        )
{
  /*** Variable deklarieren ***/

  /*** Kindprozess starten und PID ermitteln ***/
  switch(             )
  {
    /*** Fehlerfall ***/
              :

    /*** Kindprozess ***/
              :
      /*** Umlenken von stdin, bzw stdout an das jeweilige Pipeende   ***/
      /*** Fehlerabfrage nicht vergessen (Fehlercode an Vaterprozess) ***/
```

```
        /*** Im Kindprozess nicht benoetigte Filedeskriptoren schliessen ***/

        /*** Neues Programm im laufenden Prozess starten ***/

        /*** Fehlercode an Vaterprozess zurueckliefern ***/

    /*** Vaterprozess ***/
        :
      break;
  }
  /*** Prozess ID des Kindprozesses zurueckliefern ***/
}
```