182.023 Systemnahe Programmierung
1. Test

09. April 2003

KNr. | MNr. | Zuname, Vorname

Ges.)(80) | 1.)(25) | 2.)(20) | 3.)(35) | Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Makefile, Unix und C-Fragen (25)

1.1 Makefile und Unix (10)

Sie befinden sich im Verzeichnis `project`. In diesem Verzeichnis liegen die folgenden Files:

- `statistics.c`, `statistics.h`
- `reliability.c`, `reliability.h`
- `analysis.c`, `analysis.h`
- `input.data`

Das Programm `analysis` wird aus den Files `analysis.o`, `statistics.o` und `reliability.o` erzeugt. `analysis` liest Eingabedaten von `stdin` ein und gibt als Ergebnis die zwei Files `plotfile.p` und `result.txt` aus. Das File `plotfile.p` wird von einem Visualisierungstool (bestehend aus einem Server und beliebig vielen Clients) graphisch dargestellt. Der Plotserver und der Plotclient sind im `/usr/bin/plot/` Verzeichnis abgelegt. Der Plotclient (`pl_client`) liest Eingabedaten von `stdin`.

1.1.1 Makefile (5)

Erstellen Sie ein Makefile, welches die im Kommentar stehenden Aufgaben erfüllt. Verwenden Sie die Makefile-Variablen!

```
#####################################################################
# Project:        analysis of field data
# Creation date:  09.04.2003
# Version:        1.0
# Project:        Makefile for project analysis

FLAGS= -migrate -std1 -check -w0 -g
CC= c89

# Fuegen Sie das Target "analysis" hinzu, welches das
```



```
# ausfuehrbare Programm erzeugt.

all: analysis



# Erzeugen Sie das Object-File "analysis.o" aus
# analysis.c und analysis.h



# Erzeugen Sie das Object-File "reliability.o" aus
# reliability.c und reliability.h



# Erzeugen Sie das Object-File "statistics.o" aus
# statistics.c und statistics.h



# Loeschen Sie die erzeugten Object-Files, Ergebnisfiles und
# das ausfuehrbare Programm. Verhindern Sie Fehlermeldungen
# bei nicht existenten Files.

clean:



# End of file
#####################################################################
```

1.1.2 Unix (5)

Alle `#define`-Tags in den Header-Files (*.h) sollen in ein File mit dem Namen `defines.txt` geschrieben werden. Verwenden Sie dazu die Befehle `cat` und `grep`. Es dürfen keine temporären Files benutzt werden (verwenden Sie unnamed pipes)! Annahme: alle `#define`-Tags stehen in einer Zeile.

Starten Sie den Plot-Server als Hintergrundprozess. `/usr/bin/plot/` befindet sich nicht in der `PATH` Umgebungsvariable der Shell.

Visualisieren Sie das Output File mit dem Plotclient.

Ändern Sie die Rechte vom Programm `analysis` derart, dass der Owner alle Rechte (lesen, schreiben, ausführen) hat, und die Gruppe nur lesen und ausführen darf. Alle anderen haben keine Rechte. Geben Sie die Rechte im `absolute mode` an, d.h. die Rechte sollen im oktalen System gesetzt werden.

Sie sind in das Verzeichnis `/home/bob/misc/` gewechselt, in dem sich folgende Dateien befinden: `input.d01`, `inpUt.d02`, `inpuT.d03`, `iNput.d04`, `inpUt.d01`, `iNput.d02`, `Input.d03`, `inpuT.d04`.
Welche Datei(en) wird/werden ausgegeben, wenn Sie den Befehl `ls i?pU?.[a-d][1-4]` ausführen?

1.2 C-Fragen (15)

1.2.1 Bitfields (2)

Deklarieren Sie das Bitfield mit dem Namen `SensorMode`, welches Einstellungen für einen Sensor speichert. Es sollen nur vorzeichenlose Variablen vom Typ `int` verwendet werden. Das Bitfield hat folgende Variablen (Anzahl der Bits in Klammer): type(3), service_mode(1), checked(1), confidence(4), version(4), mode(3).

1.2.2 Strukturierte Datentypen (5)

Deklarieren Sie eine Struktur `TSensor`, welche folgende Elemente enthält: eine Zeichenkette `name` mit `MAXLENGTH` echten Zeichen, ein Bitfeld `smode` vom Typ `SensorMode` (siehe vorheriges Beispiel), einen Sensorwert `measurement` vom Typ vorzeichenloser Integer, einen Schwellwert `threshold` vom Typ Integer und einen Pointer vom Typ `TSensor` auf ein Reservesensor `backup`. Deklarieren Sie weiters eine Variable `mysensor` vom Typ `TSensor`. Es darf kein `typdef` verwendet werden!

1.2.3 Nachvollziehen der Programmsemantik (8)

```
#define MAX 5
#define INIT_VECTOR {12,34,89,4,1}

int main() {
   int i;
   int vector[MAX] = INIT_VECTOR;
   int *p[MAX];

   for(i=0; i<(MAX-2); i++) {
      p[i] = &vector[i];
   }
   p[3] = &vector[4];
   p[4] = &vector[3];
   vector[1] = vector[0] + *p[2];
   vector[2] = (*p[2])++;
   vector[4] = vector[0] * *p[4];
   for(i=0; i<MAX; i++) {
     printf("vector[%d]: %d \n",i, vector[i]);
     printf("p[%d]: %d \n", i, *p[i]);
   }
   return 0;
}
```

Was gibt das Programm (`printf()`) auf die Konsole aus?

2 Argumentbehandlung (20)

In diesem Beispiel soll die Argumentbehandlung für das Programm `simple_grep` durchgeführt werden. `simple_grep` ist eine vereinfachte Variante des Unix-Tools `grep`. Es gibt jene Zeilen von Files aus, die einem spezifizierten Muster entsprechen.

Die Aufrufsyntax lautet:

```
simple_grep [-c|-l|-q] [-pparagraph_separator] -e pattern file [file .. ]
```

Implementieren Sie die Argumentbehandlung von `simple_grep` nach UNIX-Konvention. Die Argumente/Optionen sind in der nachfolgenden Tabelle spezifiziert:

Argument	Beschreibung
`-c`	Das Programm zeigt die Anzahl der übereinstimmenden Zeilen an.
`-l`	Das Programm zeigt den Namen jeder Datei mit übereinstimmenden Zeilen an.
`-q`	Alle Ausgaben mit Ausnahme von Fehlermeldungen werden unterdrückt.
`-p` *paragraph_separator*	Es wird der gesamte Absatz bei einer Übereinstimmung angezeigt. Das Optionsargument *paragraph_separator* ist ein String und dient zur Bestimmung der Grenzen von Absätzen. Als Vorgabewert wird eine leere Zeile für *paragraph_separator* verwendet.
`-e` *pattern*	Das Optionsargument definiert das zu suchende Pattern.

Beachten Sie dabei folgende Hinweise:

- Die Optionen `-c`, `-l`, und `-q` schließen sich wechselweise aus.
- Die Optionen `-e` ist obligatorisch. Fehlt sie, so stellt dies eine Verletzung der Aufrufsyntax dar.
- Nach erfolgreicher Argumentbehandlung ist die Funktion `search` aufzurufen:
  `void search(int mode,char *sep,char *pattern,char **file,int nr_files)`
  Der Funktionsparameter mode spezifiziert die Art der Ausgabe. Bei Angabe der Option `-c` ist `mode` auf den Wert `MODE_C` zu setzen. Bei `-l` muss `mode` den Wert `MODE_L` erhalten und bei `-q` den Wert `MODE_Q`. Der Parameter `sep` legt den Absatz-Separator fest (Option `-p`). Der Parameter `pattern` zeigt auf den zu suchenden Musterstring. Im Paramter `file` wird ein Array von Dateinamen übergeben. Die Anzahl der Dateinamen in diesem Array ist in `nr_files` abgelegt.
- Das mehrfache Auftreten einer Option stellt eine Verletzung der Aufrufsyntax dar.
- Geben Sie bei der Verletzung der Aufrufsyntax eine Usage-Meldung auf `stderr` aus!

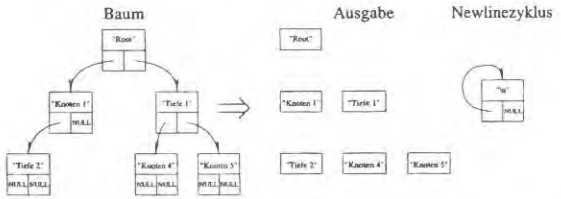
```
/* Anzeigemodi */
#define MODE_DEFAULT    0
#define MODE_C 1
#define MODE_L 2
#define MODE_Q 3

#include ... /* headerfiles muessen nicht angegeben werden */


int main (                                  ) {
```



```
search(

        );
}
```

3 Message Queue (35)

Ein binärer Baum besteht aus Knoten. Jeder Knoten in einem binären Baum besitzt ein Info-Feld und 2 Zeiger auf Nachfolgeknoten. Hat ein Knoten keine zwei Nachfolger, so werden Zeiger, welche nicht auf gültige Nachfolger weisen, mit `NULL` belegt.



Die Knoten eines binären Baumes sollen auf einem Terminal ausgegeben werden. Die Ausgabe soll folgende Form haben: in der $k$ten Zeile werden alle Knoten der Tiefe $k-1$ aufgelistet. Für nicht vorhandene Knoten wird kein Platz freigelassen.

Ein kreativer Programmierer hat sich dazu folgenden Algorithmus ausgedacht:

1. Eine Message Queue wird angelegt.
2. Weiters wird ein „Newlinezyklus" erstellt. Dieser enthält einen einzigen Knoten vom gleichen Typ wie der Baum, dessen Info-Feld nur das Newlinezeichen ('\n') enthält und nur sich selbst als einzigen Nachfolger besitzt.
3. Der Pointer auf den Wurzelknoten wird in die Queue gestellt.
4. Dann wird der Pointer auf den Newlinezyklus in die Queue gestellt.
5. Für die Abarbeitung wird eine Nachricht (ein Pointer) aus der Queue gelesen.
6. Das Infofeld des Knotens wird auf dem Terminal ausgeben.
7. Dann werden die Pointer auf alle Nachfolger in die Queue gestellt.
8. Solange zwei aufeinanderfolgende Pointer in der Queue nicht gleich sind, ist bei Punkt 5 fortzufahren. Anmerkung: Genau dann, wenn in der Queue ein „Newlinepointer" einem „Newlinepointer" nachfolgt, ist der Baum vollständig ausgegeben.

Sie sollten nun die Prozedur `void BaumAusgabe(Knoten_t *)` in C implementieren. Dabei dürfen Sie eine `void BailOut(char *)` Prozedur verwenden, welche eine Fehlermeldung ausgibt, verwendete Resourcen freigibt und terminiert. Hinweis: Stellen Sie zu Punkt 7 folgende Überlegungen an:

- Was hat mit `NULL`-Zeiger zu geschehen.
- Muss die Reihenfolge der Nachfolger beachtet werden?

3.1 Include File

```
/* includefiles müssen nicht angegeben werden */
/* defines */
```

```
/* typdefs */
typedef struct Knoten{                /* Struktur für einen Knoten */
  char szInfo[8];                     /* die Knoten-Info           */
  struct Knoten *pkLeft,*pkRight;     /* und die beiden Nachfolger */
}Knoten_t;
```

3.2 Algorithmusimplementation

```
#include<Baum.h>
/* globale Variablen */
```

```
void AllocateResources(void){
```

```
}
void FreeResources(void){
```

```
}
void BaumAusgabe(Knoten_t                    ){
```

}

Was ist von diesem Versuch zu halten? Ist diese Lösung genial oder kann es Probleme geben? Für welche Aufgaben wurde die Messagequeue entwickelt? Ist sie auch für solche Anwendungen das richtige Werkzeug?

b) Hauptprogramm (15)

Schreiben Sie ein entsprechendes Hauptprogramm, welches als Argument die aktuelle Systempriorität übergeben bekommt. Zur Argumentbehandlung steht Ihnen die Funktion `ProcessArguments(int argc, char **argv, long *nPriority)` zur Verfügung, welche in Ihrem letzten Parameter (einem Variablenparameter) die Systempriorität als `long` Wert zurückliefert.

Die Terminierung des Run-Servers soll mittels Signalen ermöglicht werden. Hierzu steht Ihnen die Signalbehandlungsfunktion `void SignalHandler(int nSignal)` zur Verfügung, welche für die Signale `SIGTERM`, `SIGINT` und `SIGQUIT` zu installieren ist.

Legen Sie eine Message Queue mit den Permissions `PERM` und dem Key `KEY` an, wobei darauf zu achten ist, dass ein *mehrfacher Start* des Run-Servers verhindert werden soll.

Verwenden Sie nun die von Ihnen geschriebene Funktion `ProcessQueue()` zur Bearbeitung der Start-Anforderungen.

Für den Fall, dass beim Anlegen der Message Queue oder bei der Bearbeitung der Start-Anforderungen ein Fehler auftritt, soll der Run-Server durch den Aufruf der Funktion `BailOut(const char *szMessage)` beendet werden. `BailOut()` entfernt alle angelegten Resourcen und gibt eine Fehlermeldung entsprechend den Übungsrichtlinien aus.

```
const char *szCommand = "<not yet set>";                    /* Programmname */
static int nMessageQueueID = -1;                            /* Message Queue ID */

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


    szCommand = argv[0];                                    /* Programmname speichern */

    ProcessArguments(argc, argv,                            );

    /* Signalbehandlungsroutinen installieren ... */
```



```
    /* Message Queue anlegen ... */

    {
        BailOut("Can't create message queue!");
    }

    /* ProcessQueue() aufrufen ... */

    {
        BailOut("Can't process message queue!");
    }

    return 0;
}
```

2 Argumentbehandlung (35)

Es soll ein Programm zur formatierten Ausgabe von Textfiles programmiert werden. Als Argumente werden die auszugebenden Files angegeben (**mind. ein** File muss angegeben werden!). Diese Files werden sequentiell formatiert ausgegeben. Optional kann nach jedem File ein Seitenumbruch als Trennung eingefügt werden. Weiters kann optional für die Seiten ein Kopfzeilentext angegeben werden, wobei auch angegeben werden kann, wie viele Zeilen unter einer Kopfzeile vom Text leer sein müssen.
Dazu sollen folgende Optionen nach UNIX-Konvention implementiert werden:

-p	aktiviert Seitenumbruch nach jedem File
-h *Text*	aktiviert einen Kopfzeilentext (max. 80 Zeichen)
-d *Anzahl_Leerzeilen*	legt die Leerzeilen unter einer Kopfzeile fest (Default=0), nur in Verbindung mit der Option -h zulässig

Die Anzahl der Leerzeilen muss ein ganzzahliger Wert im Bereich von 0 bis 25 sein und darf nur dann angegeben werden, wenn auch eine Kopfzeile angegeben wurde.

Bei fehlerhafter Angabe von Argumenten soll das Programm mit einer *Usage*-Meldung terminieren und den Rückgabewert EXIT_FAILURE liefern.
Bei allgemeinen Fehlern verwenden Sie die Funktion void error_exit(char *szMessage); zum Programmausstieg. Bei korrekter Angabe soll das Programm die Textformatierung wie angegeben durchführen und den Rückgabewert EXIT_SUCCESS liefern.

Folgende Funktionen stehen zur Textformatierung zur Verfügung:

void clear_page(void); ...
fügt einen Seitenvorschub ein.
void format_text(char *szFileName); ...
gibt das Textfile *szFileName* formatiert aus.
void set_header(int nDist, char *szText); ...
setzt die Kopfzeile einer Seite auf *szText* mit *nDist* Leerzeilen für den Text.

Vervollständigen Sie das vorgegebene Programmgerüst, sodass es besagte Funktionalität erfüllt.

```
/* ********************************************************** includes ***/
      Includes muessen nicht angegeben werden!!
/* ********************************************************** defines ***/
#define MAX_LINE 25
#define MAX_COL 80
/* ********************************************************** globals ***/
const char *szCommand = "<not yet set>";
/* ********************************************************** prototypes ***/
void clear_page (void);
void format_text (char *szFileName);
void set_header (int nDist, char *szText);
```



```
/* ********************************************************** functions ***/

void usage(void)          /* Ausgabe der Usage-Meldung und Programmausstieg */
{
   (void) fprintf(




      );

                                ;  /* Programm beenden und Rueckgabewert */
}                              /* f. fehlerhafte Ausfuehrung zurueckliefern */
/* ********************************************************** main program ***/

int main(                                )
{                                          /* *** Variablendeklarationen *** */




   szCommand =                ;     /* Globalen Zeiger auf Programmnamen setzen */

   while ((       = getopt(                                  )) !=       )

   {

      switch (   )
```

1. Test aus Systemprogrammierung 2000-04-05

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(35) 3.)(35) Zusatzblätt

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Makefile und C-Fragen (30)

a) Pointer und Arrays (10)

Betrachten Sie folgendes Programmstück:

```
int anVector[] = {1, 2, 3, 4, 5};
int * pnElement = &anVector[3];

*pnElement = 10;
*anVector += 2
pnElement[1] = 15;
pnElement -= 2;
*pnElement = 42;
```

Welche Werte haben die einzelnen Elemente des Arrays `anVector` nach Exekution des obigen Programmstücks?

```
anVector[0]=

anVector[1]=

anVector[2]=

anVector[3]=

anVector[4]=
```


b) Makefile (10)

Der Quellcode zum Programm `strings` ist auf zwei verschieden Module (`functions.c` und `main.c`) verteilt. Hierbei stellt `functions.c` Funktionen zur Verfügung, die von `main.c` benützt werden. Aus diesem Grund inkludiert `main.c` das zu `functions.c` gehörige Headerfile `functions.h`.

Auch `functions.c` inkludiert `functions.h` damit bei der Übersetzung von `functions.c` die Konsistenz zwischen `functions.c` und `functions.h` geprüft wird.

Vervollständigen Sie das folgende Makefile damit aus den beiden Modulen (`functions.c` und `main.c`) das ausführbare Programm `strings` erzeugt wird. – Regeln zum Entfernen der erzeugten Files brauchen nicht angegeben werden (d.h., ein eigenes `clean` Target ist **nicht** nötig).

```
##
## Makefile
##


all : strings


strings :

    c89 -migrate -std1 -check -w0 -o


functions.o :

    c89 -migrate -std1 -check -w0 -c


main.o :

    c89 -migrate -std1 -check -w0 -c


##
## = eof ==========================================================================
##
```

Wie müssen Sie Wertparameter an eine Funktion übergeben ?

- ○ Der Wert der Variablen wird als Wertparameter übergeben
- ○ Die Adresse der Variablen wird als Wertparameter übergeben.

Strings und Arrays

Gegeben ist folgender Abschnitt aus einem C-Programm:

```
char  line[10];

/* Eingabe */
if (fgets(line, sizeof(line), stdin) == NULL) {
  /* Fehlerbehandlung */
}
if (fgets(line, sizeof(line), stdin) == NULL) {
  /* Fehlerbehandlung */
}
```

Geben Sie für **jede** Speicherstelle des Arrays *line* den Wert an, der nach Ablauf des Abschnittes *Eingabe* da[illegible] gespeichert ist unter der Annahme, daß folgende zwei Zeilen (bestehend aus je einem Wort) eingegeben werden

```
Convert
URL
```

Gehen Sie davon aus, daß beim Einlesen kein Fehler erfolgt. Wählen Sie die Form **line[C] : *Wert*** usw.. Sollte ein Wert unbestimmt sein, so verwenden Sie bitte den Wortlaut *unbestimmt*.

line[0]:

line[1]:

line[2]:

line[3]:

line[4]:

line[5]:

line[6]:

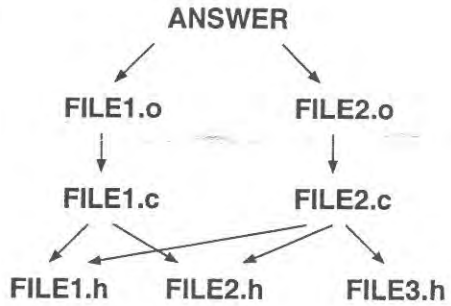
line[7]:

line[8]:

line[9]:

Makefile (10)

Erstellen Sie ein Makefile nach den Übungsrichtlinien, das die in der Abbildung dargestellten Abhängigkeiten auflöst und das File *ANSWER* erzeugt. Verwenden Sie zum Compilieren und Linken die entsprechenden Anweisungen aus der Übung. Das Makefile soll außerdem eine *Clean* Regel enthalten, die alle generierten Dateien löscht. Verwenden Sie als TAB Zeichen das Zeichen ↪!



1. Nachtragstest aus Systemprogrammierung 2001-06-12

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(35) 3.)(35) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Message Queues (30)

Ein bestehendes Multiuser System soll um einen Mechanismus erweitert werden, der es erlaubt, die Exekution von Prozessen abhängig von deren Priorität zu unterbinden bzw. zuzulassen.

Hierzu werden einem Server Prozess (dem sogenannten *Run-Server*) *Start-Anforderungen* über eine Message-Queue geschickt, die den *Namen des auszuführenden Programms* und dessen *Priorität* enthalten. Die Aufgabe des Run-Servers besteht nun darin, alle jene Start-Anforderungen zu behandeln (d.h. die jeweiligen Programme zu starten), deren Priorität *größer oder gleich* der aktuellen Systempriorität ist, und zwar geordnet nach Prioritäten (d.h. eine Start-Anforderung mit einer niedrigen Priorität wird erst *nach* einer Start-Anforderung mit einer höheren Priorität behandelt).

Alle Prioritäten liegen hierbei im Intervall [1...MIN_PRIORITY], wobei MIN_PRIORITY eine ganze Zahl größer 1 ist und die niedrigste Priorität darstellt.

a) Funktion ProcessQueue() (15)

Schreiben Sie eine Funktion ProcessQueue(), welche die ID einer Message Queue und die aktuelle Systempriorität als Parameter erhält. Lesen Sie in dieser Funktion alle jenen Start-Anforderungen von der Message Queue, deren Priorität größer oder gleich der Systempriorität ist. Verarbeiten Sie diese Anforderungen, indem Sie die entsprechenden Programme exekutieren. Eine Start-Anforderung hat hierbei das folgende Format:

```
typedef struct
{
    long nPriority;                    /* Prioritaet */
    char szCommand[PATH_MAX];          /* Programmname */
} request_t;
```


Verwenden Sie zum Starten des Programms die Bibliotheksfunktion system(3). Diese Funktion liefert im Fehlerfall einen Wert kleiner als 0 oder den Wert 127 zurück.

Führen Sie insofern eine Fehlerbehandlung durch, dass die Funktion ProcessQueue() im Fehlerfall den Wert 0 zurückliefert (es soll **keine Fehlermeldung** von der Funktion ausgegeben werden).

```
int ProcessQueue(int nMessageQueueID, long nPriority)
{
    /* lokale Variablen deklarieren ... */

    /* Nachrichten empfangen ... */

    /* Kommando ausfuehren ... */

    /* Fehler ? */
}
```

c) Parameter (10)

Betrachten Sie folgende Funktion:

```
void zoppel(int nFoo, int * nBar, char * szString)
{
    nFoo = 42;
    nBar = &nFoo;
    (void) strcpy(szString, "Hello!");
}
```

Im Hauptprogramm wird nun die Funktion `zoppel()` in folgender Weise aufgerufen:

```
int nFile = 8;
int nZahl = 12;
char szProg[] = "Zoppelprogramm";

zoppel(nFile, &nZahl, szProg);
```

Welche Ausgabe erzeugt nun (also nach dem obigen Aufruf der Funktion `zoppel()`) folgende Anweisung?

```
(void) printf("nFile=%d\nnZahl=%d\nszProg=%s\n", nFile, nZahl, szProg);
```

nFile=

nZahl=

szProg=

2 Argumentbehandlung (35)

Es soll ein Programm zur formatierten Ausgabe von Textfiles programmiert werden. Als Argumente werden die auszugebenden Files angegeben (**mind. ein** File muss angegeben werden!). Diese Files werden sequentiell formatiert ausgegeben. Optional kann nach jeder Seite ein Seitenumbruch als Trennung eingefügt werden. Weiters kann optional für die Seiten ein Kopfzeilentext angegeben werden, wobei auch angegeben werden kann, wie viele Zeilen unter einer Kopfzeile vom Text leer sein müssen.
Dazu sollen folgende Optionen nach UNIX-Konvention implementiert werden:

`-p`	aktiviert Seitenumbruch nach jedem File
`-h` *Text*	aktiviert einen Kopfzeilentext (max. **80** Zeichen)
`-d` *Anzahl_Leerzeilen*	legt die Leerzeilen unter einer Kopfzeile fest (Default=0), nur in Verbindung mit der Option -h zulässig

Die Anzahl der Leerzeilen muss ein ganzzahliger Wert im Bereich von **0** bis **25** sein und darf nur dann angegeben werden, wenn auch eine Kopfzeile angegeben wurde.

Bei fehlerhafter Angabe von Argumenten soll das Programm mit einer *Usage*-Meldung terminieren und den Rückgabewert `EXIT_FAILURE` liefern.
Bei allgemeinen Fehlern verwenden Sie die Funktion `void error_exit(char* szMessage);` zum Programmausstieg. Bei korrekter Angabe soll das Programm die Textformatierung wie angegeben durchführen und den Rückgabewert `EXIT_SUCCESS` liefern.

Folgende Funktionen stehen zur Textformatierung zur Verfügung:

`void clear_page(void);` ...
fügt einen Seitenvorschub ein.
`void format_text(char *szFileName);` ...
gibt das Textfile *szFileName* formatiert aus.
`void set_header(int nDist, char *szText);` ...
setzt die Kopfzeile einer Seite auf *szText* mit *nDist* Leerzeilen vor dem Text.

Vervollständigen Sie das vorgegebene Programmgerüst, sodass es besagte Funktionalität erfüllt.

```
/* ********************************************************** includes ***/
      Includes muessen nicht angegeben werden!!
/* ************************************************************ defines ***/
#define MAX_LINE 25
#define MAX_COL 80
/* ************************************************************ globals ***/
const char *szCommand = "<not yet set>";
/* ********************************************************** prototypes ***/
void clear_page (void);
void format_text (char *szFileName);
void set_header (int nDist, char *szText);
```

```
                /* Vaterprozess */

    } /* end switch */
} /* end main */
```


1. Übungstest aus Systemprogrammierung 1998-04-01

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(70) Zusatzblätter:

Ich bin damit einverstanden, daß das Ergebnis der Prüfung zusammen mit meiner Matrikelnummer ausgehängt wird.

Unterschrift

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Makefile und C-Fragen (30)

a) Makefile (10)

Im aktuellen Verzeichnis befinden sich die Quelldateien (inklusive zugehörigem Makefile) für ein C-Programm. Die Eingabe von `ls -al` liefert folgende folgende Ausgabe (beachten Sie bitte die Modifikationszeiten).

```
-rw-r--r--   1 tom      inst         448 Mar 12 08:42 Makefile
-rwxr-xr-x   1 tom      inst       24576 Mar 12 08:43 fancy
-rw-r--r--   1 tom      inst         134 Mar 12 08:42 fancy.c
-rw-r--r--   1 tom      inst          19 Mar 12 08:42 fancy.h
-rw-r--r--   1 tom      inst        1040 Mar 12 08:43 fancy.o
-rw-r--r--   1 tom      inst          75 Mar 12 08:42 strange.c
-rw-r--r--   1 tom      inst          39 Mar 12 08:42 strange.h
-rw-r--r--   1 tom      inst        1032 Mar 12 08:43 strange.o
-rw-r--r--   1 tom      inst          51 Mar 12 08:42 weird.c
-rw-r--r--   1 tom      inst          39 Mar 12 08:46 weird.h
-rw-r--r--   1 tom      inst         704 Mar 12 08:43 weird.o
```

Durch die Eingabe von `more Makefile` stellen Sie fest, daß die Datei `Makefile` folgendes Aussehen hat:

```
#
# Makefile
#

all : fancy

fancy : fancy.o strange.o weird.o
        c89 -migrate -std1 -check -w0 -o fancy fancy.o strange.o weird.o

fancy.o : fancy.c fancy.h strange.h Makefile
        c89 -migrate -std1 -check -w0 -c fancy.c

strange.o : strange.c strange.h weird.h Makefile
        c89 -migrate -std1 -check -w0 -c strange.c

weird.o : weird.c weird.h Makefile
        c89 -migrate -std1 -check -w0 -c weird.c

clean :
        rm -f fancy.o strange.o weird.o

clobber : clean
        rm -f fancy
```

Welche Kommandos werden führt `make` bei der Eingabe von `make fancy.o` aus?

Welche Kommandos werden führt `make` bei der Eingabe von `make` aus (bedenken Sie, daß Sie vorher `make fancy.o` getippt haben) ?

Welche Kommandos werden führt `make` bei der Eingabe von `make clobber` aus?

b) Pointer, Arrays, Strings (10)

Betrachten Sie folgendes Programmstück:

```
char szString[11];
char * pChar;

pChar = &szString[3];
(void) strcpy(szString, "Ein String");
*pChar = '\0';
```

Welche Ausgabe auf den Terminal erzeugen die folgenden Funktionsaufrufe:

```
(void) fprintf(stdout, "%s\n", pChar + 1);
(void) fflush(stdout);


(void) fprintf(stdout, "%s\n", &szString[4]);
(void) fflush(stdout);


(void) fprintf(stdout, "%s\n", szString);
(void) fflush(stdout);
```

c) Parameter (10)

Wie sieht der Prototyp (= Funktionskopf) einer Funktion `convert` aus, die als ersten Parameter einen *Werteparameter* `nBase` vom Typ `int`, als zweiten Parameter einen *Variablenparameter* `nValue` vom Type `long` und als dritten Parameter einen *Variablenparameter* `szString` vom Typ `const char *` erhält. Die Funktion soll *keinen* Rückgabewert liefern.

Betrachten sie nun folgende Deklarationen:

```
long nBase = 10;
int nValue = 42;
const char * szString = "Ein konstanter String!";
```

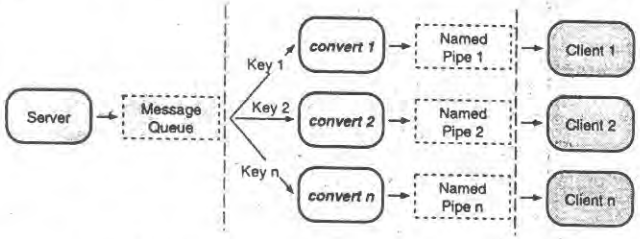
Wie sieht ein Aufruf der Funktion `convert` mit `nBase` als erstem, `nValue` als zweitem und `szString` als drittem Parameter aus?

2 Implizite Synchronisation, Argumente (70)

Gegeben sei ein Mail-System bestehend aus einem Server und einer Anzahl von Clients. Leider wurden Server und Clients von unterschiedlichen Herstellern programmiert und sind nicht kompatibel:

- Der Server empfängt die Nachrichten und stellt diese in eine *Message Queue*. Als Key wird die User-ID des Empfängers verwendet.
- Die Clients erwarten, daß ihnen die Nachricht in einer *Named Pipe* geliefert werden, wobei der Name der Pipe der User-ID des Empfängers entspricht.

Da die Client- und Serverprogramme nicht verändert werden können, muß ein Programm *convert* geschrieben werden, das die Nachrichten von der Message Queue liest und auf eine Named Pipe überträgt. Das Programm *convert* wird einmal für jeden Client gestartet (siehe Bild).



Das Programm *convert* erlaubt die Verwendung von zwei "Sicherheitsklassen" für Named Pipes. Named Pipes der Sicherheitsklasse 'l' (low security) können grundsätzlich von allen Benutzern ausgelesen werden. Named Pipes der Sicherheitsklasse 'h' (high security) dürfen nur von den Mitgliedern der User Gruppe des Empfängers der Nachricht gelesen werden.

a) Message Queues, Named Pipes (40)

Implementieren Sie das Programm *convert*, das die Nachrichten für einen Empfänger aus der Message Queue ausliest und in eine Named Pipe für diesen Empfänger schreibt. Berücksichtigen Sie dabei folgende Punkte:

- Das Programm *convert* bezieht wesentliche Information über seine Argumente. Zur Argumentbehandlung steht die Funktion *int processarg(int argc, char **argv, char *fname, long *key, char *security)* zur Verfügung. Die Funktion wertet die Parameter aus und liefert in der Variablen *fname* den Namen der Named Pipe, in der Variablen *key* den Schlüssel für die aus der Message Queue zu lesenden Nachrichten und in *security* die Sicherheitsklasse ('l' bzw. 'h') für die Named Pipe. Der Returnwert der Funktion *processarg* ist im fehlerfreien Fall 1, im Fehlerfall 0.
- Die Message Queue wird vom Server erzeugt. Beachten Sie, daß der Konverter keine Message Queue selbstständig erzeugen darf, sondern mit einer Fehlermeldung abbrechen muß, wenn die Queue nicht existiert.

- Die Named Pipe mit dem Namen des Benutzers ist von *convert* anzulegen. Berücksichtigen Sie dabei die Unterscheidung der zwei Sicherheitsklassen.
- Die Nachrichten haben eine maximale Länge von 160 lesbaren Zeichen ('A' - 'Z', 'a' - 'z', '0' - '9', ' ' und '.').
- Trennen Sie in der Named Pipe einzelne Nachrichten durch ein Newline Zeichen.
- Zur Ausgabe von Fehlermeldungen und zum Terminieren des Programms verwenden Sie die Funktion *void error_exit(char *msg)*.

Vervollständigen Sie das folgende Programmgerüst!

```
/**                                                    **
 ** convert.c                                          **
 **                                                    **/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <limits.h>

/** Message Queue Key ist bekannt                      **/

#define MQ_KEY 1821

/**                                                    **
 ** Message Queue Permission definieren                **
 **                                                    **/



/**                                                    **
 ** Messagetyp fuer Queue definieren!                  **
 **                                                    **/
```

```
static char *cmnd;

/**                                                    **
 ** Prototyp fuer Funktion "processarg" deklarieren!   **
 **                                                    **/



/** Funktion error_exit im Fehlerfall aufrufen         **/

void error_exit (char *msg)
{
   (void) fprintf(stderr, "%s: %s: %s\n", cmnd, msg, strerror(errno));
   exit(EXIT_FAILURE);
}

int main(int argc, char **argv)
{
   /**                                                 **
    ** Ergaenzen Sie die benoetigten Variablen        **
    ** Achten Sie darauf, den richtigen Typ zu waehlen! **
    **                                                 **/

   char fname[PATH_MAX];       /** File Name der Named Pipe **/



   cmnd = argv[0];

   /**                                                 **
    ** Argumente durch Aufruf von "processarg" verarbeiten. **
    **                                                 **/
```



```
   /**                                                 **
    ** Resourcen anfordern. Message Queue existiert schon! **
    **                                                 **/



   while (1) {
      /**                                              **
       **  Daten von Message Queue in Named Pipe umleiten. **
       **                                              **/



   }
   return EXIT_SUCCESS;                    /** not reached **/
}
```

b) Argumentbehandlung (30)

Programmieren Sie die Funktion *int processarg(int argc, char **argv, char *fname, long *key, char *security)*, die vom Programm *convert* zur Durchführung der Argumentbehandlung aufgerufen wird. Berücksichtigen Sie dabei die folgende Aufrufsyntax von *convert*.

```
convert [-s <securityclass>] <receiver_id>
```

Dabei ist folgende Funktionalität erwünscht:

- Die Anzahl und das Format der Argumente sind zu überprüfen. Verwenden Sie die Funktion *getopt(3)*.
- Mit der Option *-s* kann dem Programm explizit die gewünschte Sicherheitsklasse übergeben werden. Gültige Werte sind 'l' und 'h'. Wird eine gültige Sicherheitsklasse angegeben, so wird diese von *processarg* im Paramter *security* retourniert. Wird beim Aufruf von *convert* keine Sicherheitsklasse angegeben, so wird als Default der Wert 'l' zurückgegeben.
- Im Argument `<receiver_id>` wird die User-ID Nummer des Empfängers an das Programm *convert* übergeben. Diese User-ID wird von *processarg* sowohl als String (Parameter *fname*), als auch als Integerwert (Parameter *key*) zurückgeliefert.
- Der Returnwert der Funktion *processarg* ist im fehlerfreien Fall 1, im Fehlerfall 0.

```
int processarg(int argc, char **argv, char *fname, long *key, char *security)
{
```



```
}
```

1. Übungsstest aus Systemprogrammierung

1997-011-01

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(35) 3.)(35) Zusatzblätter:

Ich bin damit einverstanden, daß das Ergebnis der Prüfung zusammen mit meiner Matrikelnummer ausgehängt wird.

Unterschrift

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 C-Fragen und Makefile (30)

Funktionsaufrufe

Gegeben ist eine Struktur, die folgendermaßen deklariert ist:

```
typedef struct {
   int flag;
   int count;
   char data[100];
} SHM_TYPE;
```

Die Funktion *shmread* liest Daten aus einem Shared Memory in eine Variable vom Typ *SHM_TYPE*. Wie muß die Variable an die Funktion *shmread* übergeben werden ?

☐ Wertparameter

☐ Variablenparameter

Geben Sie die Deklaration für die Funktion *shmread* an. Die Funktion *shmread* liefert einen Integerwert zurück.

Deklarieren Sie eine Variable vom Typ *SHM_TYPE* und rufen Sie damit die Funktion *shmread* auf.

Wie müssen Sie Variablenparameter an eine Funktion übergeben ?

- ☐ Der Wert der Variablen wird als Wertparameter übergeben
- ☐ Die Adresse der Variablen wird als Wertparameter übergeben.

Wann müssen Sie sich in einer Funktion um Dereferenzierung einer Variablen kümmern?

- ☐ Wenn die Variable als Wertparameter übergeben wurde.
- ☐ Wenn die Variable als Variablenparameter übergeben wurde.

Wie müssen Sie Wertparameter an eine Funktion übergeben ?

- ☐ Der Wert der Variablen wird als Wertparameter übergeben
- ☐ Die Adresse der Variablen wird als Wertparameter übergeben.

Strings und Arrays

Gegeben ist folgender Abschnitt aus einem C-Programm:

```
#include <stdio.h>

int main(int argc, char **argv)
{
   char  line[10];

   /* Eingabe */

   if (fgets(line, sizeof(line), stdin) == NULL) {
     /* Fehlerbehandlung */
   }

   if (fgets(line, sizeof(line), stdin) == NULL) {
     /* Fehlerbehandlung */
   }

   /* Ausgabe */
```


Geben Sie für **jede** Speicherstelle des Arrays *line* den Wert an, der nach Ablauf des Abschnittes *Eingabe* darin gespeichert ist unter der Annahme, daß folgende zwei Zeilen (bestehend aus je einem Wort) eingegeben werden:

```
Windows
NT
```

Gehen Sie davon aus, daß beim Einlesen kein Fehler erfolgt. Wählen Sie die Form **line[0]** : *Wert* usw.. Sollte ein Wert unbestimmt sein, so verwenden Sie bitte den Wortlaut *unbestimmt.*

Makefile

Ergänzen Sie im folgenden Makefile die Anweisungen, die zum Compilieren und Linken notwendig sind. Verwenden Sie hierzu die entsprechenden Anweisungen aus der Übung. Das Makefile soll außerdem eine *Clean* Regel enthalten, die alle generierten Dateien löscht. Verwenden Sie als TAB Zeichen das Zeichen ↪!

```
all: test


clean:


test: file1.o file2.o


file1.o: file1.c file1.h file3.h


file2.o: file2.c file1.h file2.h
```

2 Argumentbehandlung und Files (35)

Implementieren Sie das Kommando *my_digest* zur Ausgabe der ersten Zeichen jeder Zeile eines Files:

```
my_digest [-n number] [-t] file ...
```

Das Kommando muß mit mindestens einem Filenamen als Argument aufgerufen werden. Es liest die als Parameter angegebenen Files und gibt die ersten Zeichen jeder Zeile der Reihe nach auf *stdout* aus. Dabei kann abhängig von den angegebenen Optionen jeder Zeile ein Tabulator vorangestellt werden. Wird keine Option angegeben, werden die ersten 5 Zeichen jeder Zeile ausgegeben.

`-n number` Wird das Programm mit dieser Option aufgerufen, werden die ersten *number* Zeichen jeder Zeile ausgegeben. Diese Option darf höchstens einmal angegeben werden.

`-t` Auch diese Option darf höchstens einmal angegeben werden und bewirkt, daß jeder Zeile bei der Ausgabe ein Tabulator vorangestellt wird.

Achten Sie bei der Lösung der Aufgabe auf die richtige Behandlung von Argumenten und Files, sowie auf die korrekte Fehlerbehandlung (alle Fehler sollen zur Termination des Programms führen). Ihr Programm soll keine künstlichen Beschränkungen aufweisen. Ein Tip: Lesen Sie die Files zeichenweise mit Hilfe der Funktion *fgetc()* ein.

Hinweis: Verwenden Sie bei der Fehlerbehandlung die Funktionen *usage()* und *error_exit()*.

```
/**
 **  my_digest.c
 **/

/* Zu inkludierende Header-Files muessen NICHT angegeben werden! */

/* Bitte ergaenzen */

#define DEFAULT_NUM



static char *cmnd;

/* Die Funktionen usage() und error_exit() */

/* Schreibt den usage string auf stderr und beendet Programm */
extern void usage ( void );

/* Schreibt den error string auf stderr und beendet mit spezifiziertem error
```



```
code */
extern void error_exit ( char * , int );


int main(int argc, char **argv)
{
    /* Ergaenzen Sie hier die noetigen Deklarationen und Initialisierungen */




    /* Ergaenzen Sie hier die Argumentbehandlung. Verwenden Sie getopt()! */
```

```
/* Ergaenzen Sie hier die Bearbeitung der Files */
```

```
}
```


3 Implizite Synchronisation (35)

In einer Datenerfassungsstelle gibt es N Clients und neun (1..9) Server. Bei den Clients werden der Nachname, der Vorname und die Postleitzahl einer zu erfassenden Person eingegeben. Die Clients stellen die Daten dann so in eine Message Queue, daß die Server die Datensätze gezielt auslesen können. Gezielt heißt, daß es für jeden Postleitzahlbereich (von 1000 bis 1999, von 2000 bis 2999, ...) einen Server gibt. Ein Server soll daher nur die für seinen Postleitzahlbereich relevanten Datensätze aus der Message Queue lesen und diesen an 'sein' Datenfile anhängen.

Bei Kommentaren oder Punkten in den Programmgerüsten sollten Sie das Programm ergänzen!

Für alle Clients und Server gibt es ein gemeinsames Headerfile `cs.h`, in dem Sie benötigte Konstanten noch ergänzen sollen. Gehen Sie davon aus, daß keine Eingabe länger als `MAXTEXT` ist!

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

#define MAXTEXT 255

#define KEY ......

#define PERM ......

/* Message fuer Datensatz definieren */
typedef struct MESSAGE
{




} msgtype;

static char *cmnd;
volatile int msgid = -1;
```

Ergänzen Sie das Programmgerüst des Clients so, daß obige Funktionalität erfüllt wird. Zum Einlesen von Nachname, Vorname und Postleitzahl rufen Sie die Funktion

```
int daten_einlesen(msgtype *msg)
```

auf. welche in die übergebene Struktur Nachname und Vorname als Charakterstring und Postleitzahl als Integer einliest. Returnwert ist 1 im fehlerfreien Fall, -1 wenn ein Fehler aufgetreten ist.

```
#include "cs.h"

int main(int argc, char **argv)
{
   struct MESSAGE msg;
/* Deklarieren Sie hier weitere benoetigte Variablen */



   cmnd = argv[0];
   if (argc != 1)
   {
      (void) fprintf(stderr, "Usage: %s\n", cmnd);
      exit(EXIT_FAILURE);
   } /* if */

/* msgid bestimmen - die Message Queue muss schon existieren ! */

   if (
   {
      (void) fprintf(stderr, "%s: Can't get Message Queue - %s\n",
                     cmnd, strerror(errno));
      exit(EXIT_FAILURE);
   }

   while(1)
   {

/* Nachname, Vorname und Postleitzahl einlesen */









/* Postleitzahlbereich feststellen */
```



```
/* Message in die Queue stellen */

      if (
      {
         (void) fprintf(stderr, "%s: Can't send message - %s\n",
                        cmnd, strerror(errno));
         exit(EXIT_FAILURE);
      } /* if */
   } /* while (1) */
} /* main */
```

Das folgende Programmgerüst stellt den Server für alle Wiener Postleitzahlen dar (1000 $\leq$ Postleitzahl $<$ 2000). D.h. der Server soll alle für ihn relevanten Datensätze aus der Message Queue lesen und an ein File (`wiener.txt`) anhängen. Falls beim Lesen aus der Message Queue oder beim Schreiben auf das File ein Fehler auftritt, soll der Server ordnungsgemäß terminieren!

```
#include "cs.h"

int main(int argc, char **argv)
{
   struct MESSAGE msg;
   FILE *fp;

   cmnd = argv[0];
   if (argc != 1)
   {
      (void) fprintf(stderr, "Usage: %s\n", cmnd);
      exit(EXIT_FAILURE);
   } /* if */

/* msgid bestimmen oder Message Queue anlegen, falls sie noch nicht existiert */

   if (
   {
      (void) fprintf(stderr, "%s: Can't create Message Queue - %s\n",
                     cmnd, strerror(errno));
      exit(EXIT_FAILURE);
   } /* if */
```

```
   if ((fp = fopen ("wiener.txt", "a")) == NULL)
   {
      (void) fprintf(stderr, "%s: Can't open file wiener.txt - %s\n",
                     cmnd, strerror(errno));
      exit(EXIT_FAILURE);
   } /* if */

   while(1)
   {

/* Datensatz aus Message Queue auslesen */









/* Daten an File wiener.txt anhaengen */









   } /* while (1) */
} /* main */
```

1 Übungstest aus Systemprogrammierung 1997-05-07

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(35) 3.)(35) Zusatzblätter:

Ich bin damit einverstanden, daß das Ergebnis der Prüfung zusammen mit meiner Matrikelnummer ausgehängt wird.

Unterschrift

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 C-Fragen und Makefile (30)

Funktionsaufrufe (10)

Gegeben ist die folgende Deklaration:

```
struct shm {
   int flag;
   int count;
   char data[99];
} shmvar;
```

Geben Sie die Deklaration für eine Funktion mit dem Namen *shmwrite* an, die als Werteparameter die Struktur *shm* erhält und einen Integerwert zurückliefert.

Rufen Sie die Funktion mit der Variablen *shmvar* auf. Der Rückgabewert soll in der Integervariablen *nReturn* gespeichert werden.

Geben Sie die Deklaration für eine Funktion mit dem Namen *shmwrite* an, die als Variablenparameter die Struktur *shm* erhält und einen Integerwert zurückliefert.

Rufen Sie die Funktion mit der Variablen *shmvar* auf. Der Rückgabewert soll in der Integervariablen *nReturn* gespeichert werden.

Strings und Arrays (10)

Gegeben ist folgender Abschnitt aus einem C-Programm:

```
  :
  :
FILE *fp;
char  line[12];
  :
  :
if ((fp=fopen("info.txt", "r")) == NULL) {
   (void) fprintf(stderr, "Cannot open info.txt: %s\n",
   strerror(errno));
 }

while (fgets(line, sizeof(line),fp) != NULL);
  :
  :
```

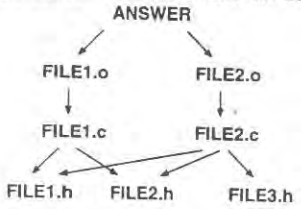
Die Datei *info.txt* besteht aus folgenden Zeilen, es stehen keine Leerzeichen in der Datei:

```
Sysprog
Uebung
Test
```

Geben Sie für jede Speicherstelle des Arrays *line* den Wert an, der nach Beendigung der while-Schleife gespeichert ist. Wählen Sie hierzu die Form **line[0] :** *Wert* usw.. Sollte ein Wert unbestimmt sein, schreiben Sie den Wortlaut *unbestimmt* hin. Gehen Sie davon aus, daß beim Einlesen kein Fehler auftritt.

Makefile (10)

Erstellen Sie ein Makefile, das die in der Abbildung dargestellten Abhängigkeiten auflöst und das File *ANSWER* erzeugt. Verwenden Sie zum Compilieren und Linken die entsprechenden Anweisungen aus der Übung. Das Makefile soll außerdem eine *Clean* Regel enthalten, die alle generierten Dateien löscht. Verwenden Sie als TAB Zeichen das Zeichen ↪!



2 getopt() und Files (35)

Programmieren Sie das Kommando *column* zum Ausgeben von Spalten aus Files:

```
column [-m VonSpalte] [-n BisSpalte] [-t]* file ...
```

Das Kommando muß mit mindestens einem Filenamen als Argument aufgerufen werden. Es liest die als Parameter angegebenen Files und gibt die angegebenen Spalten aus den Files auf *Stdout* aus. Die Spaltenpositionen werden mit den Parametern `-m` (Start der Spalte) und `-n` (Ende der Spalte) definiert. Der Defaultwert für `-m` ist 0 und für `-n` ist 25.

Vor dem Spaltentext können beliebig viele Tabulatoren ausgegeben werden. Die Anzahl von `-t` beim Aufruf bestimmt die Anzahl der Tabulatorzeichen `'\t'`, die vor dem Spaltentext ausgegeben werden.

`-m VonSpalte` Der Begin der Spalte wird auf den Wert *VonSpalte* gesetzt.

`-n BisSpalte` Das Ende der Spalte wird auf den Wert *BisSpalte* gesetzt.

`-t` Diese Option darf beliebig oft angegeben werden und bewirkt, daß jedem Spaltentext genau die Anzahl an Tabulatorzeichen `'\t'` vorangestellt werden, wie diese Option angegeben wurde.

Achten Sie bei der Lösung der Aufgabe auf die richtige Behandlung von Argumenten und Files, sowie auf die korrekte Fehlerbehandlung (alle Fehler sollen zur Termination des Programms führen). Ihr Programm soll keine künstlichen Beschränkungen aufweisen.

Hinweis: Verwenden Sie bei der Fehlerbehandlung die vordefinierten Funktionen *usage()* und *error_exit()*. Sie brauchen im Fehlerfall geöffnete Files nicht schließen.

```
/*
 * column.c
 *
 */


static char *cmnd;

/*
 * Die Funktionen void usage( void ) und void error_exit( char * ) koennen
 * Sie als gegeben betrachten
 *
 */

int main ( int argc, char **argv )
{
    /* Ergaenzen Sie hier die noetigen Deklarationen und Initialisierungen */
```

```
int     nStart= 0;
int     nEnd= 25;



/* Ergaenzen Sie hier die Argumentbehandlung */

while ( ( c = getopt (                                    ) ) != EOF )
{
    switch ( c )
    {









    }
}

/* Sind die Optionen fehlerhaft ? */
```



```
/* Ergaenzen Sie hier die Bearbeitung der Files */


while (                    )
{
    char      * pActFileName=                    ;/* !!! Zuweisung ergaenzen */
    FILE      * pFile;
              nC;                                  /* verwendet bei fgetc */
/* !!! Datentyp ausfuellen */
    int       nPos= 0;


    if ( ( pFile = fopen ( pActFileName, "r" ) ) == NULL )
    {
        error_exit("can't open file");
    }


    /* bis an das Ende der Datei */

    while ( ( nC= fgetc ( pFile)) != EOF )
    {
        /* Zeilenende feststellen */

        if ( nC == '\n' )
        {
            nPos= 0;

            /* schreibe Zeilenende Zeichen */









            continue;
        }


        if ( nPos >= nStart && nPos < nEnd )
```

```
        {
            /* Behandlung der Tabulatoren */

            /* Zeichen ausgeben */

        }
        nPos++;
    }

    /* Fehlerbehandlung stdout */

    /* schliessen des gelesenen Files */
    (void)fclose (pFile);

    /* Behandlung des naechsten Files vorbereiten */

  }
  exit(0);
}
```


3 Named Pipes (35)

Schreiben Sie ein Programm *min_und_max* zur Verdichtung von Meßdaten.

```
SYNOPSIS
        min_und_max <pipe-name>
```

Das Programm *min_und_max* liest über die Named Pipe *pipe-name* Meßwerte von Meßstationen in der Form

```
min: <minval> max: <maxval>
```

wobei jeder solche Meßwertstring maximal 20 Zeichen lang ist und *<minval>* bzw. *<maxval>* für die minimalen bzw. maximalen Meßwerte einzelner Meßstationen aus dem Integer-Wertebereich {`MINVAL` ... `MAXVAL`} stehen. Das Programm *min_und_max* läuft endlos und hängt jeweils nach hundert gelesenen Paaren von Meßdaten den kleinsten Minimalwert und den größten Maximalwert der letzten hundert Wertepaare an das File mit dem Namen *ergebnisse* an.

Schreiben Sie ein Programmstück, das die beschriebene Funktionalität erfüllt. Verwenden Sie die Funktion *sscanf()* zum Aufspalten der Meßwertstrings. Nehmen Sie dabei an, daß die Strings syntaktisch korrekt sind. Behandeln Sie Fehler beim Programmablauf mit der Funktion *void error_exit(char *err_msg)*, die den als Parameter übergebenen String ausgibt und anschließend das Programm beendet. Files und Pipes brauchen Sie im Fehlerfall nicht zu schließen. Includefiles brauchen Sie nicht anzugeben.

```
}
                ;           /* Programm beenden, Rueckgabewert */
}
                /* f. korrekte Ausfuehrung zurueckliefern */
```


3 C-Fragen (35)

Funktionsaufrufe

Gegeben ist eine Struktur, die folgendermaßen deklariert ist:

```
typedef struct {
   int flag;
   int count;
   char data[100];
} SHM_TYPE;
```

Die Funktion *shmread* liest Daten aus einem Shared Memory in eine Variable vom Typ *SHM_ TYPE*. Wie muß die Variable an die Funktion *shmread* übergeben werden ?

- ○ Wertparameter
- ○ Variablenparameter

Geben Sie die Deklaration für die Funktion *shmread* an. Die Funktion *shmread* liefert einen Integerwert zurück

Deklarieren Sie eine Variable vom Typ *SHM_ TYPE* und rufen Sie damit die Funktion *shmread* auf.

Wie müssen Sie Variablenparameter an eine Funktion übergeben ?

- ○ Der Wert der Variablen wird als Wertparameter übergeben
- ○ Die Adresse der Variablen wird als Wertparameter übergeben.

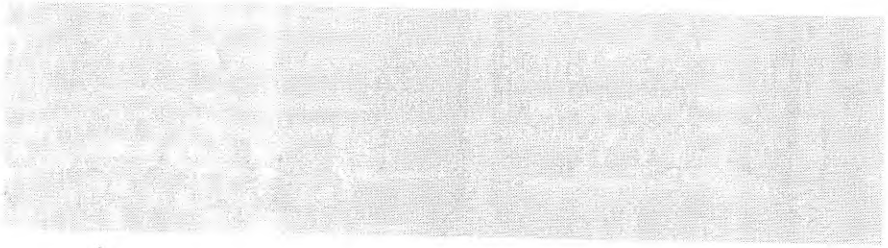
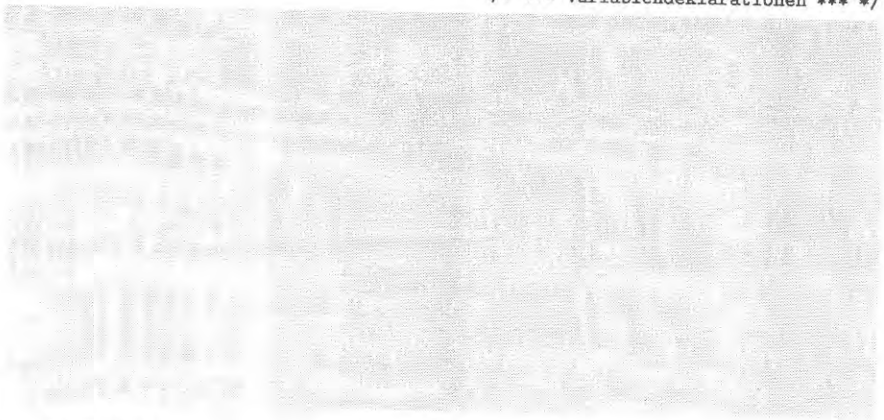
Wann müssen Sie sich in einer Funktion um Dereferenzierung einer Variablen kümmern?

- ○ Wenn die Variable als Wertparameter übergeben wurde.
- ○ Wenn die Variable als Variablenparameter übergeben wurde.

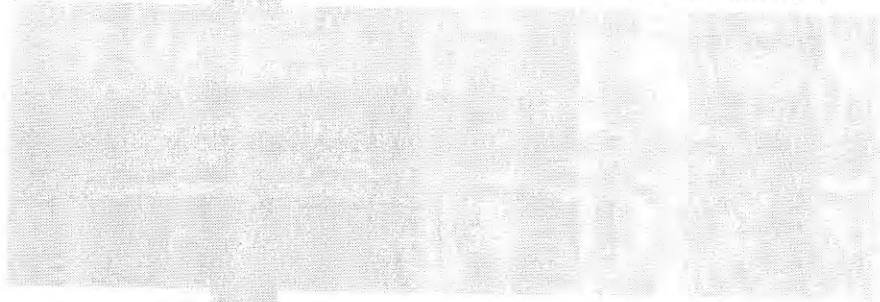
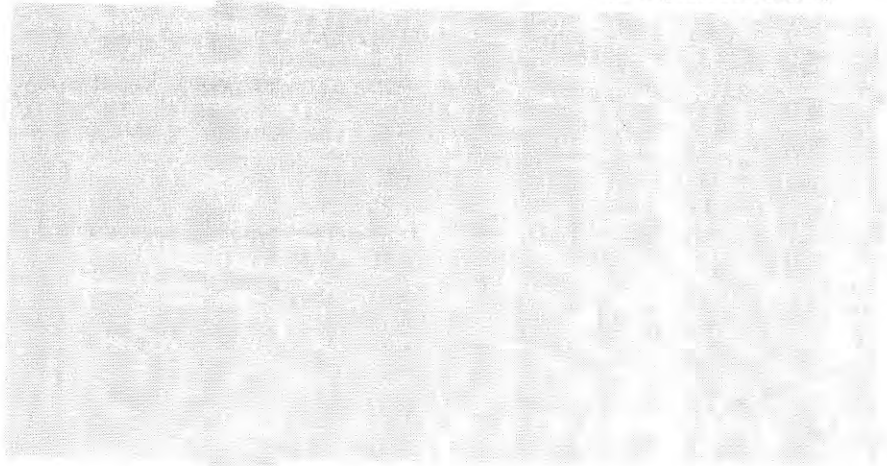
```
{
        case      :                          /* Seitenumbruch */

        case      :                          /* Kopfzeilen-Text */

        case      :                          /* Leerzeilen nach Kopfzeile */
```



```
        case      :          /* falsches Argument wurde gefunden */

    default:
        assert(0);
    break;
    }                                        /* Ende von switch(.. */
}                                            /* Ende von while (.. */
                                  /* *** Fehlerbehandlung *** */

if (                    )                    /* Kopfzeile setzen? */
    set_header (                    );
for (                                        ) {
    format_text(             );

    if (                    )                /* neue Seite beginnen? */
        clear_page();
```

```
/* ************************************************************ functions ***/

void usage(void)            /* Ausgabe der Usage-Meldung und Programmausstieg */
{
   (void) fprintf(



        );


; /* Programm beenden und Rueckgabewert */
}                                   /* f. fehlerhafte Ausfuehrung zurueckliefern */
/* ************************************************************ main program ***/

int main(  )
{                                           /* *** Variablendeklarationen *** */



   szCommand =              ;      /* Globalen Zeiger auf Programmnamen setzen */

   while ((       = getopt(                              )) !=       )
   {

       switch (   )
```



```
{

        case      :                                    /* Seitenumbruch */



        case      :                                    /* Kopfzeilen-Text */



        case      :                                    /* Leerzeilen nach Kopfzeile */

:
```

```
        default:
            assert(0);
        break;
    }                                          /* Ende von switch(.. */
}                                              /* Ende von while (.. */
                                             /* *** Fehlerbehandlung *** */
```

```
if (                    )                      /* Kopfzeile setzen? */

    set_header (                    );

for (                                        ) {

    format_text(                );


    if (                    )                  /* neue Seite beginnen? */

        clear_page();

}

                            ;                  /* Programm beenden, Rueckgabewert */
}                          /* f. korrekte Ausfuehrung zurueckliefern */
```


3 Message Queues (35)

Gegeben sei ein einfaches Mail-System bestehend aus einem Serverprozess *m_server*, der eine Message Queue verwaltet, und einer Anzahl von Clientprozessen *m_client*, die dem Senden und Empfangen von Nachrichten dienen. Der Client *m_client* kann hierzu auf **zwei** verschiedene Arten aufgerufen werden:

- `m_client <receiver_id> <message>`
  Es wird die Message `<message>` mit dem Messagetyp `<receiver_id>` in der mit `MSERVER_KEY` festgelegten Message Queue abgelegt. Die `<receiver_id>` entspricht der User ID des Empfängers. Anschließend terminiert der Client (*Sendemodus*).
- `m_client`
  Alle für den entsprechenden Benutzer bestimmten Nachrichten werden aus der Message Queue ausgelesen und auf *stdout* ausgegeben. Danach terminiert der Client (*Empfangsmodus*). Ist die Queue leer, terminiert der Client ebenfalls.

Die Struktur einer Nachricht für das Mail-System ist im Include-File (siehe nächste Seite) bereits definiert.

Implementieren Sie die Programme *m_server* und *m_client* unter Berücksichtigung folgender Punkte:

- Die Message Queue wird vom Server *m_server* erzeugt. Sollte die Queue schon existieren ist eine Fehlermeldung auszugeben. Ein Client *m_client* darf keine Message Queues selbständig anlegen, sondern soll mit einer Fehlermeldung abbrechen, wenn die benötigte Message Queue nicht existiert. Das Löschen (bzw. Entfernen) der Message Queue soll nicht implementiert werden.
- Die Ausgabe der Nachrichten soll im Format `<sender_id> <message>` erfolgen.
- Zur Ausgabe von Fehlermeldungen und zum Beenden des Programms im Fehlerfall steht Ihnen die Funktion `void BailOut(const char *)` zur Verfügung.
- Die Argumentbehandlung für den Client ist im gegebenen Programmgerüst bereits weitgehend vorgegeben. Die Verwendung von getopt ist hier nicht mehr erforderlich! Ergänzen Sie den Programmcode, so dass sichergestellt ist, dass
  - die evtl. angegebene receiver_id auch wirklich **ganzzahlig** und **größer als 0** ist
  - die evtl. angegebene short_message auf keinen Fall länger als die definierte Konstante `MAX_MSGLEN` ist

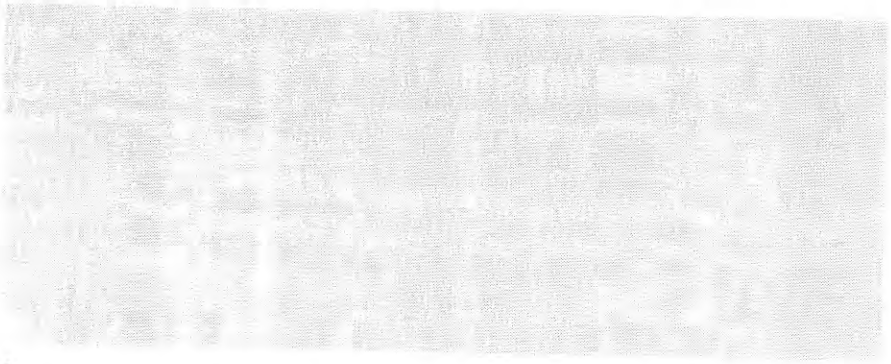
  und rufen Sie die bereits vordefinierte Funktion `void usage(void)` auf, wenn einer dieser Punkte nicht erfüllt wird.
- Die User-ID des *aktuellen Benutzers* lässt sich mit einem Aufruf der bereits vordefinierten Funktion `long getuid(void)` ermitteln.
- Beide Programme greifen auf das folgende Include-File zu:

```
#define MSERVER_KEY   88776655
#define MSERVER_PERM  0644
#define MAX_MSGLEN 50

typedef struct msg_s {
    long receiver_id;              /* User ID Empfaenger */
    long sender_id;                /* User ID Sender     */
    char text[MAX_MSGLEN+1];       /* Nachricht          */
} msg_t;
```

a) Server (10)

Vervollständigen Sie folgendes Programmgerüst!

```
/** m_server.c **/
/* Zu inkludierende Header-Files muessen NICHT angegeben werden! */

static char *cmnd;
int main (int argc, char **argv)
{
   cmnd = argv[0];
```



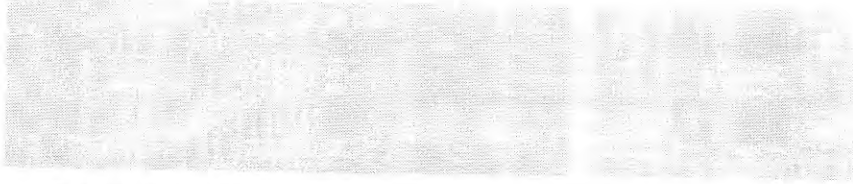
```
}
```

b) Client (25)

Vervollständigen Sie folgendes Programmgerüst!

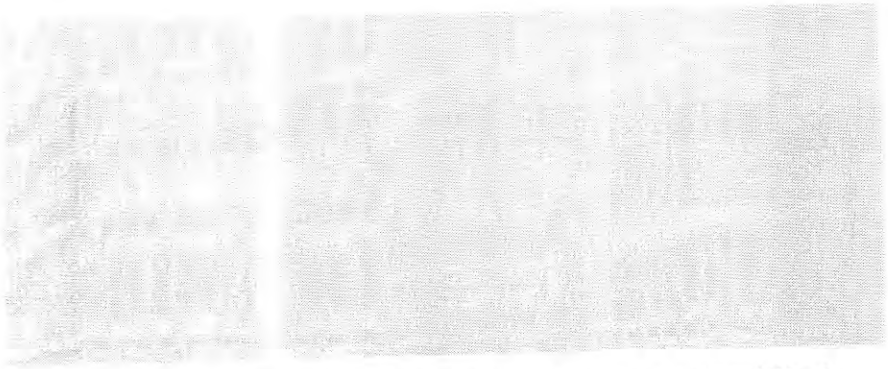
```
/** m_client.c **/
/* Zu inkludierende Header-Files muessen NICHT angegeben werden! */

static char *cmnd;
int main (int argc, char **argv)
{
   /* Zusaetzlich benoetigte Variable! */
```


```
   cmnd = argv[0];
   /* Zugriff auf Message Queue sichern! */
```

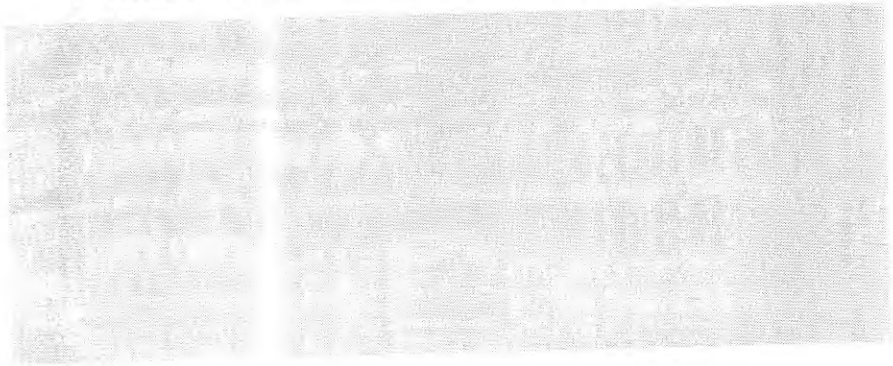


```
   if (argc == 1) {
      /* Programmcode fuer den Empfangsmodus! */
```

```
   }
   else if (argc == 3) {
      /* Programmcode fuer den Sendemodus!                                  */
      /* Ueberpruefen Sie zunaechst, ob sich die angegebene receiver_id     */
      /* (argv[1]) in eine nicht negative, ganze Zahl konvertieren laesst! */
```



```
    /* Stellen Sie sicher, dass die angegebene short_message (argv[2]) */
    /* nicht laenger als die Konstante MAX_MSGLEN ist, und dass alle   */
    /* Felder der Nachricht mit den entsprechenden Werten belegt sind! */
```



```
    /* Implementieren Sie ab hier das Senden der Nachricht! */
```

```
  }
  else {
    usage ();
  }
  return EXIT_SUCCESS;
}
```

1. Test aus Systemprogrammierung 1999-11-10

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(35) 3.)(35) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Message Queues (30)

Ein bestehendes Multiuser System soll um einen Mechanismus erweitert werden, der es erlaubt, die Exekution von Prozessen abhängig von deren Priorität zu unterbinden bzw. zuzulassen.

Hierzu werden einem Server Prozess (dem sogenannten *Run-Server*) *Start-Anforderungen* über eine **Message-Queue** geschickt, die den *Namen des auszuführenden Programms* und dessen *Priorität* enthalten. Die Aufgabe des Run-Servers besteht nun darin, alle jene Start-Anforderungen zu behandeln (d.h. die jeweiligen Programme zu starten), deren Priorität *größer oder gleich* der aktuellen Systempriorität ist, und zwar geordnet nach Prioritäten (d.h. eine Start-Anforderung mit einer niedrigen Priorität wird erst *nach* einer Start-Anforderung mit einer höheren Priorität behandelt).

Alle Prioritäten liegen hierbei im Intervall [1...`MIN_PRIORITY`], wobei `MIN_PRIORITY` eine ganze Zahl größer 1 ist und die niedrigste Priorität darstellt.

a) Funktion `ProcessQueue()` (15)

Schreiben Sie eine Funktion `ProcessQueue()`, welche die ID einer Message Queue und die aktuelle Systempriorität als Parameter erhält. Lesen Sie in dieser Funktion alle jenen Start-Anforderungen von der Message Queue, deren Priorität größer oder gleich der Systempriorität ist. Verarbeiten Sie diese Anforderungen, indem Sie die entsprechenden Programme exekutieren. Eine Start-Anforderung hat hierbei das folgende Format:

```
typedef struct
{
    long nPriority;                          /* Prioritaet */
    char szCommand[PATH_MAX];                /* Programmname */
} request_t;
```


b) Hauptprogramm (15)

Schreiben Sie ein entsprechendes Hauptprogramm, welches als Argument die aktuelle Systempriorität übergeben bekommt. Zur Argumentbehandlung steht Ihnen die Funktion `ProcessArguments(int argc, char **argv, long *nPriority)` zur Verfügung, welche in Ihrem letzten Parameter (**einem Variablenparameter**) die Systempriorität als `long` Wert zurückliefert.

Die Terminierung des Run-Servers soll mittels Signalen ermöglicht werden. Hierzu steht Ihnen die Signalbehandlungsfunktion `void SignalHandler(int nSignal)` zur Verfügung, welche für die Signale `SIGTERM`, `SIGINT` und `SIGQUIT` zu installieren ist.

Legen Sie eine Message Queue mit den Permissions `PERM` und dem Key `KEY` an, wobei darauf zu achten ist, dass ein *mehrfacher Start* des Run-Servers verhindert werden soll.

Verwenden Sie nun die von Ihnen geschriebene Funktion `ProcessQueue()` zur Bearbeitung der Start-Anforderungen.

Für den Fall, dass beim Anlegen der Message Queue oder bei der Bearbeitung der Start-Anforderungen ein Fehler auftritt, soll der Run-Server durch den Aufruf der Funktion `BailOut(const char *szMessage)` beendet werden. `BailOut()` entfernt alle angelegten Resourcen und gibt eine Fehlermeldung entsprechend den Übungsrichtlinien aus.

```
const char *szCommand = "<not yet set>";                /* Programmname */

static int nMessageQueueID = -1;                        /* Message Queue ID */


int main(int argc, char **argv)

{

    /* lokale Variablen deklarieren ... */

    szCommand = argv[0];                                /* Programmname speichern */

    ProcessArguments(argc, argv,                        );

    /* Signalbehandlungsroutinen installieren ... */
```

2 Makefile und C-Fragen (35)

a) Pointer und Arrays (10)

Gegeben sei folgender Programmcode:

```
int b[4] = {5,4,3};
int i, *p;

p = b + 1;

p[0] = 0;
p[1] = 1;

*p = 3;

b[3] = 5;
```

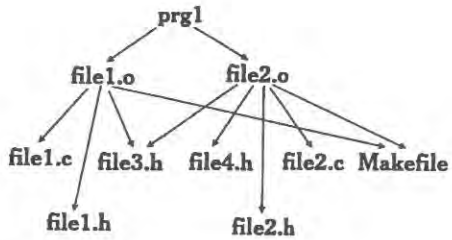
Das Array `b[]` hat _____ Elemente. (1)

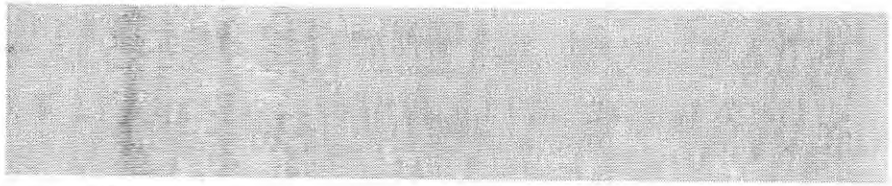
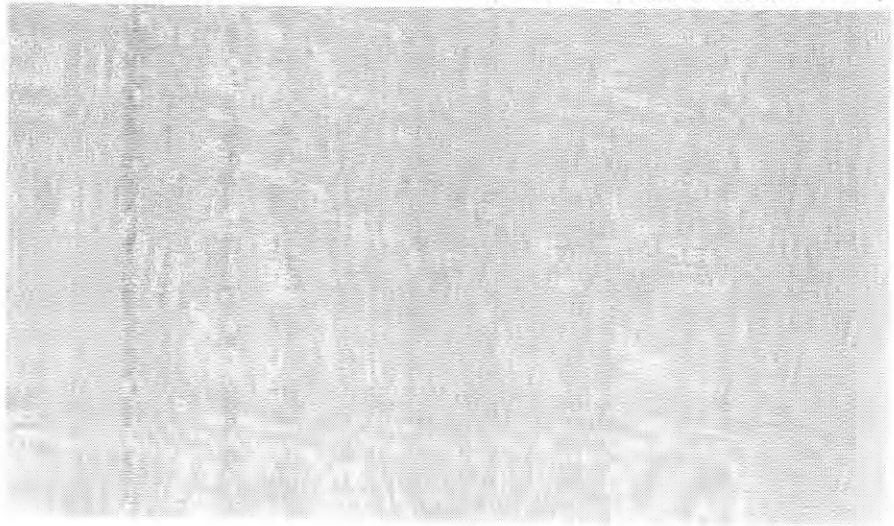
Der gültige Bereich des Index `i` für `b[i]` liegt im Intervall von _____ bis _____. (1)

Bestimmen Sie den Inhalt des Arrays `b[]` nach Ausführung dieses Programmstückes. Geben Sie für jeden gültigen Indexwert `i` den Arrayinhalt in der Form `b[i] = ...` an. (8)

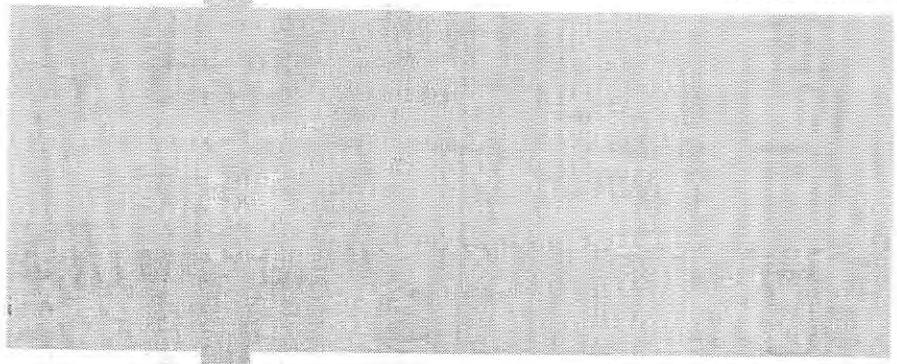
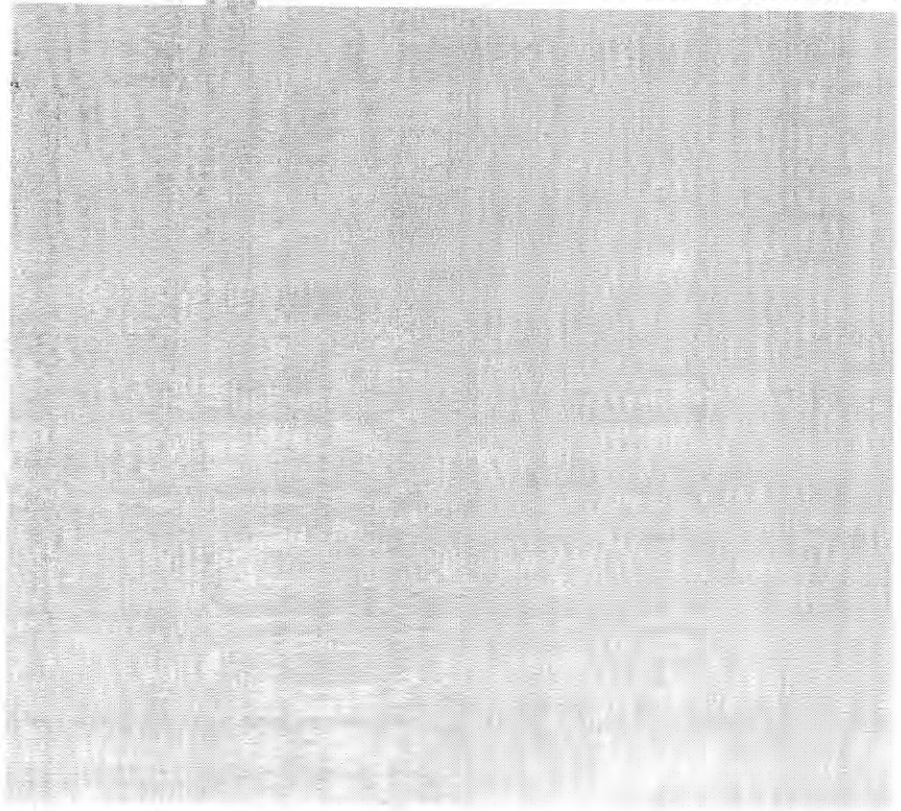
c) Makefile (10)

Erstellen Sie ein Makefile, das die dargestellten Abhängigkeiten auflöst und das File *Prg1* erzeugt. Verwenden Sie zum Kompilieren und Linken die entsprechenden Anweisungen aus der Laborübung. Das Makefile soll außerdem eine *clean*-Regel enthalten, die alle generierten Dateien löscht. Die erste Regel im Makefile muß *all* sein. Rufen Sie den Compiler mit den Optionen auf, die auch in der Laborübung gefordert werden (siehe Skriptum). Verwenden Sie als TAB-Zeichen das Zeichen ↪!



```
/* ********************************************************* prototypes ***/
void Kopiere(char *Master);
void Kopiere2(char *Master);
void MacheDeckblatt(int FC);
/* ********************************************************* functions ***/

void Usage(void)            /* Ausgabe der Usage-Meldung und Programmausstieg */
{
    (void) fprintf(

        );

                            ;     /* Programm beenden, Rueckgabewert */
}                           /* f. fehlerhafte Ausfuehrung zurueckliefern */
/* ********************************************************* main program ***/
int main(  )
{                                   /* *** Variablendeklarationen *** */

```



```
        case  :                    /* Deckblatt */

        case  :                    /* Farbe des Deckblattes */

```

1. Test aus Systemprogrammierung 2000-11-08

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(70) 2.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

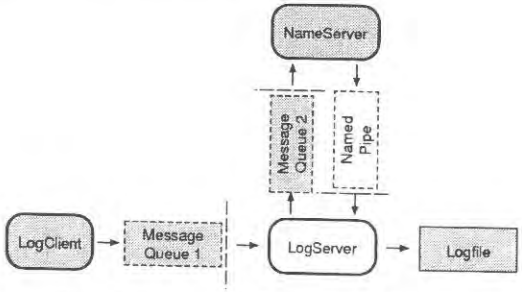
1 Implizite Synchronisation, Argumente (70)

Es ist zu beachten, dass dieses Beispiel **zwei Teile** enthält, die **unabhängig** voneinander gelöst werden können.

Es soll ein LogServer für ein zentrales Loggingsystem geschrieben werden. Dieser LogServer erhält von den LogClients Fehlercodes und übersetzt diese mittels des NameServers in einen Fehlertext und gibt diesen dann auf ein File aus:

- Ein LogClient wird mit einer Fehlerpriorität und -nummer aufgerufen und stellt diese in die *Message Queue 1* für den LogServer.
- Der NameServer bekommt über die *Message Queue 2* die Fehlerpriorität und -nummer sowie den Namen einer *Named Pipe*, auf die er den entsprechenden Fehlertext schreibt.

Der LogServer liest die Daten von der *Message Queue 1* des LogClients, schickt diese gemeinsam mit einem Pipe-Namen an die *Message Queue 2* des NameServers und liest den konvertierten Fehlertext über eine *Named Pipe* aus und schreibt ihn auf ein File (siehe Bild).



Der LogServer verlangt beim Starten als Argument die Angabe einer Mindestpriorität, wobei dann nur Anforderungen mit gleicher oder höherer Priorität verarbeitet werden.


```
/* Messagetyp fuer Message Queue 1 (mit LogClient) definieren.          */
```

```
/* Messagetyp fuer Message Queue 2 (mit Nameserver) definieren.         */
```

```
const char *szCommand = "<not yet set>";                  /* command name */
```

```
void Usage(void);                    /* braucht nicht implementiert werden */
/* Prototyp fuer Funktion "ProcessArguments" deklarieren!               */
```

```
int main(int argc, char **argv)
{
   /* Ergaenzen Sie die Deklaration der benoetigten Variablen           */
```

```
/* Oeffnen der Message Queue 2 (fuer NameServer)                    */
```

```
/* Erzeugen der Named Pipe                                          */
```

```
/* Message Queue bearbeiten                                         */
while (msgrcv(
```

```
      ) != -1)
{
```

```
/* Message an NameServer senden                                     */
```


b) Argumentbehandlung (30)

Programmieren Sie die Funktion *void ProcessArguments(int nArgs, char **apcArgs, long *pnPriority, char **pszFileName)*, die vom Programm *logserver* zur Durchführung der Argumentbehandlung aufgerufen wird. Die Eingabeparameter *nArgs* und *apcArgs* sind der Argumentzähler und -buffer des Systems. Der Buffer für den Ausgangsparameter *pszFileName* wird nicht innerhalb dieser Funktion reserviert, sondern wird als bereits reserviert angenommen. Berücksichtigen Sie dabei die folgende Aufrufsyntax von *logserver*:

```
logserver -p <priority> [<log_file>]
```

Dabei ist folgende Funktionalität erwünscht:

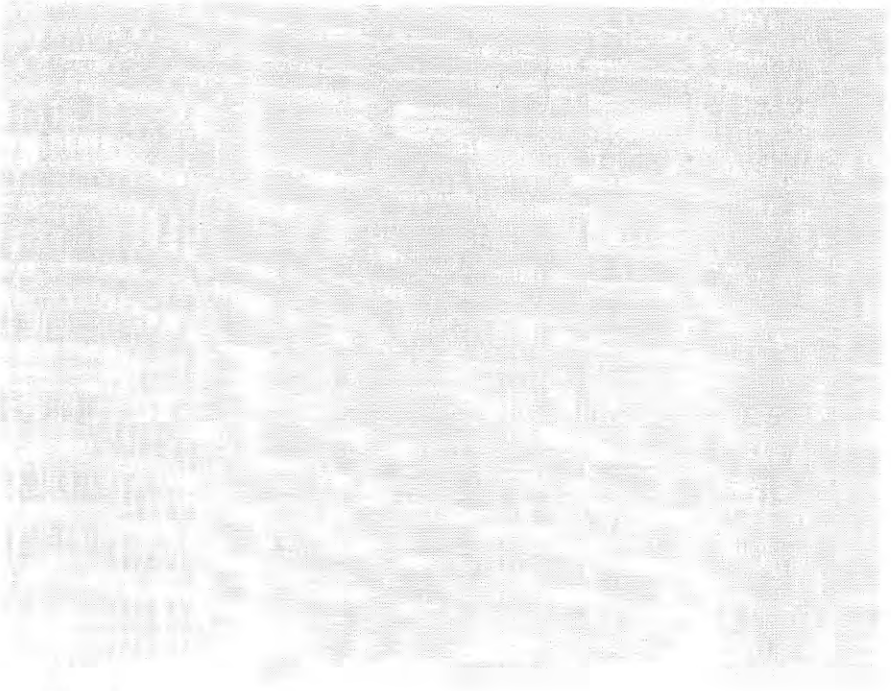
- Die Anzahl und das Format der Argumente sind zu überprüfen. Verwenden Sie dazu die Funktion *getopt(3)*.
- Mit der Option *-p* muss dem Programm explizit die gewünschte Prioritätsgrenze (in *pnPriority*) im Dezimalformat übergeben werden. Gültige Werte sind Zahlen von 1 bis `MAX_PRIORITY`, wobei 1 die höchste Priorität darstellt.
- Im optionalen Argument `<log_file>` wird der Filename des Logfiles angegeben (in *pszFileName*), wobei als Defaultwert "-" (standard output) zu setzen ist.
- Bei ungültigen Argumenten ist die Funktion *void Usage(void)* aufzurufen.
- Die Funktion *ProcessArguments* verwendet keinen Returnwert, aber im Falle eines Fehlers muss die Funktion *void Usage(void)* aufgerufen werden.

Die Funktion *void Usage(void)* gibt eine Meldung mit der entsprechenden Aufrufsyntax aus und terminiert den Prozess. *void Usage(void)* braucht **nicht** implementiert werden.

```
#define MAX_PRIORITY 10                  /*    max priority of messages */
#define MAX_NUMBER   1000                /* max error number of message */

#define MAX_NAME     NUMBER              /*     max length of pipe name, */

void ProcessArguments(int nArgs, char **apcArgs, long *pnPriority,
                      char **pszFileName)
{
```

```
            case    :
```

```
            default:
                assert(0);
            break;
        }                                   /* end of switch */
    }                                       /* end of while */
```



```
    return;
}
```


Das vom Speicherbedarf her kleinste Array, mit dem die folgenden 3 Kommandos fehlerfrei ausgeführt werden können, ist gesucht.

```
char help[    ];

strcpy(help,"Sys");
test(5,strcat(help,"prog!");
```

b) Makefile (10)

Gegeben ist das Makefile:

```
all : test

test : prog.o func.o Makefile
    c89 -o test prog.o func.o

func.o : func.c func.h Makefile
    c89 -c func.c

prog.o : prog.c func.h Makefile
    c89 -c prog.c

clean:
    rm -f *.o test
```

Welche Befehle werden ausgeführt, wenn gerade ein

`make clean`

ausgeführt wurde, die Dateien `Makefile`, `func.c`, `prog.c` und `func.h` vorhanden sind und nun folgende Eingabe gemacht wird:

KNr. MNr. Zuname, Vorname

G~ (100) 1.)(30) 2.)(40) 3.)(30) Zusatzblätter:

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Makefile und C-Fragen (30)

a) Strings, Parameter (18)

Deklarieren Sie ein Array `szName` von char, und initialisieren Sie dieses mit Ihrer Matrikelnummer.

Deklarieren Sie einen String `szCopy`, und stellen Sie anschließend durch Aufruf der Funktion `char *strcpy(char*, const char*)` eine Kopie des Arrays szName her. Der String `szCopy` soll *nicht länger* als unbedingt notwendig sein.

Ergänzen Sie die Funktion `strlen`. Der Funktion wird ein Zeiger auf einen String als Wertparameter übergeben, als Funktionswert liefert die Funktion die Länge des Strings.

```
unsigned long strlen (                    )
{




}
```


Ergänzen Sie im folgenden Makefile die Anweisungen, die zum Compilieren und Linken notwendig sind.

Das Programm `test` besteht aus den beiden Modulen `file1.c` und `file2.c`. Sowohl das Modul `file1.c` als auch das Modul `file2.c` inkludieren die Headerdatei `common.h`.

Das Makefile soll eine *Clean* Regel enthalten, die alle generierten Dateien löscht.

Beachten Sie, daß ein kompletter Compilier- und Linkvorgang auch dann ausgelöst werden soll, wenn das *Makefile* geändert wird.

Verwenden Sie hierzu die entsprechenden Anweisungen aus der Übung. Verwenden Sie als TAB Zeichen das Zeichen ↪!

```
all:


test:
        c89 -g -o


file1.o:
        c89 -g -c -std1 -migrate -check -w0


file2.o:
        c89 -g -c -std1 -migrate -check -w0
```

2 Message Queues und Argumentbehandlung (40)

Ein Server gibt den Dateninhalt von Messages von einem oder mehreren Clients nach Message Typen geordnet auf die Standard Ausgabe (`stdout`) aus. Sind mehrere Daten vorhanden, so werden Nachrichten von höherprioren Clients zuerst ausgegeben.

Hochpriore Clients sind Clients mit nummerisch kleinerer Prioritätszahl (ein Client mit `priority 1` hat die höchste Priorität und verschickt Nachrichten mit einem Message Typ von 1, ein Client mit `priority 2` hat die zweihöchste Priorität und verschickt Nachrichten mit einem Message Typ von 2, usw.).

Der Server wird von einem Benutzer mit der Option `-t` und einer nachfolgenden Nummer aufgerufen. Dieser sollte folgendermaßen möglich sein:

```
server [-t priority]
```

Ein Server sollte Nachrichten mit einem Message Typ, der kleiner oder gleich der angegebenen Priorität ist, akzeptieren. D.h. der Server gibt nur die Nachrichten von Clients mit einer "gewissen" Wichtigkeit aus. Nachrichten mit einem Message Typ höher als in der angegebenen Option sollten vom Server nicht beachtet werden.

Ihre Aufgabe ist es, den vorgegebenen Code unter Beachtung der Anleitungen zu ergänzen.

Anleitungen

Allgemeine Anleitungen

- Alle für dieses Beispiel relevanten Strukturdefinitionen und Konstanten müssen im Header File `msg_queue.h` definiert werden.
- Verwenden Sie die Funktion `error_exit` zur Fehlerausgabe. Diese müssen Sie *nicht* implementieren.
- Der Server legt eine Message Queue mit dem *Key* `MSG_QUEUE_KEY` und den *Permissions* `MSG_QUEUE_PERM` an. Falls die Message Queue bereits existiert, soll der Server mit einer Fehlermeldung terminieren. *Achtung*: Wenn beim Anlegen der Queue ein Fehler auftritt, so unterscheiden Sie bitte den Fall, dass die Queue schon existiert von allen anderen Fehlermeldungungen. Geben Sie für die beiden Fälle *unterschiedliche Fehlermeldungen* aus.
- Löschen Sie die Message Queue im Server im Fehlerfall, wenn Sie bereits angelegt wurde.
- Beim Lesen aus der Queue darf *nicht* mittels *Busy Waiting* auf Nachrichten gewartet werden.
- Die Message sollte einen String enthalten. Die *Länge* des Strings sollte maximal `MAX_DATA` Zeichen betragen. Die Konstante `MAX_DATA` ist bereits definiert.
- Sie müssen die ordnungsgemäße Beendigung des Servers nicht programmieren.

- Die Option `-t` und die dazugehörige Priorität darf *maximal* einmal vorkommen. Wenn keine Option angegeben ist, sollte der Server Nachrichten mit einem Typ von 1 bis `MIN_PRIORITIY` empfangen und ausgeben. Die Konstante `MIN_PRIORITIY` ist schon definiert.
- Die Ausgabe von Nachrichten von Clients mit gleicher Priorität sollte nach dem First-Come-First-Serve-Prinzip in der Queue erfolgen.

Vervollständigen Sie folgende Programmfragmente:

Include File

```
/* You do not have to include header files                          */
#include ....

/* constants                                                        */

#define MAX_DATA        100             /* maximum length of data    */
#define MSG_QUEUE_KEY   9999999999      /* key of message queue      */

/*
 * insert the right permissions here
 */
#define MSG_QUEUE_PERM                  /* message queue permissions */

#define MIN_PRIORITY       10           /* default value for the     *
                                         * maximum number of priorities */

char *cmnd = 0;                         /* char pointer that points to *
                                         * the name of the program   */

/*
 * Insert the definition of the message structure here
 */
```

```
/*
 * function error_exit terminates the program with a message to stderr
 * but does *not* delete the message queue
 * You do *not* have to program this function!
 */
extern void error_exit(int error_code, const char *ptext);
```

Server

```
/* Server                                                          */
#include "msg_queue.h"

/* function usage prints to the stderr and exists                  */
void usage(void) {
        (void)fprintf(stderr, "usage: %s \n", cmnd);
        exit(1);
}

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

        int c;                  /* used for getopt                 */

        /*
         * initialize the variable priority here; it is used to hold the
         * minimal acceptable priority of a client
         * The server should print messages with a message type of up to
         * the content of the variable priority.
         */

        long int priority =             ;

        /*
         * define all your variables at this place
         */
```



```
cmnd = argv[0];         /* command line name                       */

/*
 * Insert missing arguments, restrict the number options as well*/
 * as arguments
 */

/* Use getopt here for handling of the option                      */

while (( c = getopt(argc, argv,         )) !=           )
{
        switch( c )
        {
                case    :       /* option                          */

                        errno = 0;
```

```
                        priority = strtol (optarg, &endptr, 0);

                        if ( (errno != 0 ) ||
                             (optarg == endptr) ||
                             (*endptr != '\0') ||
                             (priority > MIN_PRIORITY) ||
                             (priority <= 0))
                        { /* Error handling                        */
                                usage();
                        }
                        /*
                         * priority contains the minimal
                         * acceptable priority as long integer
                         * now
                         */
                        break;
```

```
            case '?':       /* wrong argument                */
                    error_exit(6, "getopt: wrong argument");
                    break;
            default:        /* unattainable                  */
                    assert(0);

        } /* switch */

} /* while */

/* check number of arguments                                 */
```

```
/*
 * insert the missing parts for creating the message queue
 * create message queue
 * print one error message if the queue already exists and
 * a different one for all other errors
 */
```

```
while(1)
{
```



```
/*
 * insert the missing parts for reading from a message
 * queue
 * read from message queue here
 */


while(
{

        /* this is an error                                  */
        if (errno != EINTR)
        {

        } /* if */
} /* while */

/*
 * insert the name of your string variable
 * print message to stdout
 */
if (printf("%s\n",              ) < 0) {

        error_exit(6, "error occured while printing "
                "to the standard output");
}
} /* while */
}
```

3 Implizite Synchronisation mit Named Pipes (30)

Beantworten Sie die nachfolgenden Fragen zu den Programmen *Schreiber* und *Leser*. Gehen Sie dabei davon aus, dass vor dem Start der Programme die Named Pipe noch nicht vorhanden ist.

Schreiber

```
1:  /* diverse includes */
2:  FILE *fp=(FILE *)0;
3:  int main(int argc, char *argv[])
4:  {
5:      if (mkfifo(FIFO_NAME, PERM) == -1)
6:      {
7:          (void)fprintf(stderr, "%s: mkfifo failed: %s\n",
8:              argv[0], strerror(errno));
9:          return(1);
10:     }
11:     if ((fp = fopen(FIFO_NAME, "w")) == NULL)
12:     {
13:         (void)fprintf(stderr, "%s: fopen failed: %s\n",
14:               argv[0], strerror(errno));
15:         return(1);
16:     }
17:
18:     if (fprintf(fp, "Das ist die Mitteilung!\n") == EOF)
19:     {
20:         (void)fprintf(stderr, "%s: fprintf failed: %s\n",
21:               argv[0], strerror(errno));
22:         return(1);
23:     }
24:     if (fclose(fp) == EOF)
25:     {
26:         (void)fprintf(stderr, "%s: fclose failed: %s\n",
27:               argv[0], strerror(errno));
28:         return(1);
29:     }
30:     if (remove(FIFO_NAME) != 0)
31:     {
32:         (void)fprintf(stderr, "%s: remove failed: %s\n",
33:               argv[0], strerror(errno));
34:         return(1);
35:     }
36:     return 0;
37: }
```


Leser

```
1:  /* diverse includes */
2:  FILE *fp=(FILE *)0;
3:  int main(int argc, char **argv)
4:  {
5:  char text[MAX_LEN];
6:      if ((fp = fopen(FIFO_NAME, "r")) == NULL)
7:      {
8:          (void)fprintf(stderr, "%s: fopen failed: %s\n",
9:                    argv[0], strerror(errno));
10:         return(1);
11:     }
12:     while (fgets(text, sizeof(text), fp) != NULL)
13:     {
14:         if (printf("read from pipe: %s\n", text) < 0)
15:         {
16:             (void)fprintf(stderr, "%s: printf failed: %s\n",
17:                   argv[0], strerror(errno));
18:             return(1);
19:         }
20:     }
21:     if (ferror(fp))
22:     {
23:         (void)fprintf(stderr, "%s: ferror failed: %s\n",
24:                   argv[0], strerror(errno));
25:         return(1);
26:     }
27:     if (fclose(fp) == EOF)
28:     {
29:         (void)fprintf(stderr, "%s: fclose failed: %s\n",
30:                   argv[0], strerror(errno));
31:         return(1);
32:     }
33:     return 0;
34: }
```

a) (7)

Was passiert, wenn nur das Programm Leser gestartet wird? (Bitte richtige Antwort ankreuzen, Zeilennummer angeben und begründen!)

○ Das Programm Leser terminiert in Zeile: ____

○ Das Programm Leser blockiert in Zeile: ____

Begründung:

b) (7)

Was passiert, wenn nur das Programm Schreiber gestartet wird? (Bitte richtige Antwort ankreuzen, Zeilennummer angeben und begründen!)

○ Das Programm Schreiber terminiert in Zeile: ____

○ Das Programm Schreiber blockiert in Zeile: ____

Begründung:

c) (7)

Die Programme Leser und Schreiber werden gestartet. Jedoch wird das Programm Leser durch ein externes Signal (z.B. SIGINT) terminiert, während sich das Programm Schreiber in Zeile 17 seiner Programmabarbeitung befindet. Was passiert? (Bitte richtige Antwort ankreuzen, Zeilennummer angeben und begründen!)

○ Das Programm Schreiber terminiert in Zeile: ____

○ Das Programm Schreiber blockiert in Zeile: ____

Begründung:

d) (9)

Ändern Sie das Programm Schreiber dahingehend, dass im oben beschriebenen Fall c) eine cleanup Routine aufgerufen wird, die die Pipe schließt und entfernt und das Programm mit EXIT_FAILURE beendet. Eine Fehlerbehandlung dieser Operationen muss ausnahmsweise nicht durchgeführt werden.

Implementieren Sie diese cleanup Routine und geben Sie an, wie und wo Sie diese Routine in das Programm Schreiber einhängen, damit sie im Fall c) aufgerufen wird.

```
        cleanup(              )

{
```



```
}
```

```
Folgender Code wird im Programm Schreiber hinzugefuegt,
damit das Programm Cleanup im Fall c) aufgerufen wird:

nach Programmzeile: ____

Code:
```

1. Test aus Systemprogrammierung 1998-11-11

KNr. MNr. Zuname, Vorname

Ges.)(100) 1.)(30) 2.)(70) Zusatzblätter:

Ich bin damit einverstanden, daß das Ergebnis der Prüfung zusammen mit meiner Matrikelnummer ausgehängt wird.

Unterschrift

Bitte verwenden Sie nur dokumentenechtes Schreibmaterial!

1 Makefile und C-Fragen (30)

a) Makefile (10)

Hinweise

- Zur Beantwortung der nachfolgenden Fragen reicht die Angabe der jeweiligen Zeilennummer des Kommandos (bzw. das Wort 'keines', falls kein Kommando ausgeführt wird) im Makefile (Reihenfolge beachten!).
- Alle Fragen beziehen sich auf den am Anfang angegebenen Verzeichniszustand. - In Ausnahmenfällen ist das in der jeweiligen Frage explizit aufgeführt.

Im aktuellen Verzeichnis befinden sich die Quelldateien (inklusive zugehörigem Makefile) für ein C-Programm. Die Eingabe von `ls -al` liefert folgende Ausgabe (beachten Sie bitte die Modifikationszeiten).

```
-rw-r--r--   1 tom      inst         476 Oct 21 14:53 Makefile
-rw-r--r--   1 tom      inst         710 Oct 21 14:53 input.c
-rw-r--r--   1 tom      inst         695 Oct 21 14:53 input.h
-rw-r--r--   1 tom      inst         504 Oct 21 14:53 input.o
-rw-r--r--   1 tom      inst         701 Oct 21 14:53 output.c
-rw-r--r--   1 tom      inst         703 Oct 21 14:53 output.h
-rw-r--r--   1 tom      inst         512 Oct 21 14:53 output.o
-rwxr-xr-x   1 tom      inst       24576 Oct 21 14:55 program
-rw-r--r--   1 tom      inst         864 Oct 21 14:53 program.c
-rw-r--r--   1 tom      inst         698 Oct 21 14:53 program.h
-rw-r--r--   1 tom      inst        1448 Oct 21 14:56 program.o
```


Durch die Eingabe von `more Makefile` stellen Sie fest, daß die Datei `Makefile` folgendes Aussehen (die Zahlen auf der linken Seite geben die jeweilige Zeilennummer an) hat.

```
 1: all : program
 2:
 3: program : program.o input.o output.o
 4:         c89 -migrate -std1 -check -w0 -o program program.o input.o output.o
 5:
 6: program.o : program.c program.h input.h Makefile
 7:         c89 -migrate -std1 -check -w0 -c program.c
 8:
 9: input.o : input.c input.h output.h Makefile
10:         c89 -migrate -std1 -check -w0 -c input.c
11:
12: output.o : output.c output.h Makefile
13:         c89 -migrate -std1 -check -w0 -c output.c
14:
15: mostlyclean :
16:         rm -f program.o input.o output.o
17:
18: clean : mostlyclean
19:         rm -f program
```

Welche(s) Kommando(s) führt `make` bei der Eingabe von `make` aus?

Welche(s) Kommando(s) führt `make` bei der Eingabe von `make clean` aus?

Welche(s) Kommando(s) führt `make` (nachdem sie vorher `make clean` eingegeben haben) bei der Eingabe von `make input.o` aus?

Angenommen Sie legen eine Datei namens `mostlyclean` an, was zu folgendem zusätzlichen Eintrag im aktuellen Verzeichnis führt:

```
-rw-r--r--   1 tom      inst          21 Oct 21 15:20 mostlyclean
```

Welche(s) Kommando(s) führt `make` nun bei der Eingabe von `make mostlyclean` aus?

) Pointer, Arrays, Strings (10)

etrachten Sie folgendes Programmstück:

```
har szString[11];
har * pChar;

har = &szString[3];
oid) strcpy(szString, "Ein String");
```

elche Ausgabe auf dem Bildschirm erzeugen die folgenden Funktionsaufrufe:

```
oid) fprintf(stdout, "%c\n", *(pChar + 1));
oid) fflush(stdout);


oid) fprintf(stdout, "%c\n", szString[4]);
oid) fflush(stdout);


oid) fprintf(stdout, "%s\n", pChar - 3);
oid) fflush(stdout);
```

) Parameter (10)

ie sieht der Prototyp (= Funktionskopf) einer Funktion **readstring** aus, deren erster arameter ein *Werteparameter* **nFile** vom Typ **int** und deren zweiter Parameter ein *riablenparameter* **szString** vom Typ **char *** ist. Die Funktion soll als Rückgabewert nen Wert vom Typ **int** liefern.

etrachten Sie nun folgende Deklarationen:

```
nt nFile = 0;
nt nLength = 0;
har * szString = (char *) 0;
```

ie sieht ein Aufruf der Funktion **readstring** mit **nFile** als erstem und **szString** als weitem Parameter aus, wobei der Rückgabewert in **nLength** gespeichert werden soll?

2 Implizite Synchronisation

Das Programm *anmeld* dient dazu, Studenten bei Lehrveranstaltungen an- bzw. abzumelden. Dabei werden Anmeldedaten vom Benutzer als Argumente an das Programm übergeben. Die Aufrufsyntax lautet

```
anmeld -l <lva_nummer> -m <matrikel_nummer> [-x]
```

Das Programm verlangt, daß die zwei Argumente *lva_nummer* und *matrikel_nummer* genau einmal angegeben werden, wobei die beiden Argumente in beliebiger Reihenfolge angegeben werden und hinter den Optionsflags *l* und *m* stehen. Die Option *x* ist optional, darf jedoch höchstens einmal angegeben werden. Weiters gilt und ist zu überprüfen:

- Eine gültige *lva_nummer* ist eine Zahl im Wertebereich von 100000 bis 999999.
- Eine gültige Matrikelnummer besteht aus 7 Ziffern, also z.B. "9208234".
- Fehlt die Option -x handelt es sich um eine Anmeldung, ist die Option -x angegeben so ist eine Abmeldung erwünscht.

a) Argumentbehandlung (30)

Implementieren Sie die Funktion *int processarg(int argc, char **argv, long *lva_nr, char *mat_nr, char *action)*, die die Argumentbehandlung für das Programm *anmeld* durchführt. Die Funktion wertet die Parameter *argc* und *argv* aus, überprüft die Korrektheit der übergebenen Argumente und liefert in den drei Parametern folgende Resultate:

- *lva_nr*: Nummer der Lehrveranstaltung als Integerzahl
- *mat_nr*: Matrikelnummer als String
- *action*: 'A' zum Anmelden bzw. 'S' zum Stornieren/Abmelden als Charactervariable.

Der Returnwert der Funktion *processarg* ist im fehlerfreien Fall 0, im Fehlerfall -1.
Hinweis: Include-Dateien brauchen Sie nicht anzugeben.

```
int processarg(int argc, char **argv,
               long *lva_nr, char *mat_nr, char *action)
{
```

```
/* Fortsetzung von processarg */
```


b) Message Queues und Named Pipes (40)

Das LVA-Anmeldesystem besteht aus einem Server- und einem Clientprogramm. Das oben genannte Programm *anmeld* bildet den Client. Es liest Anmeldedaten vom Benutzer und schickt diese über eine *Message Queue* an den Server. Der Server erhält die Anmeldedaten über die *Message Queue*. Der Server greift auf eine zentrale Datenbank zu, wo die Anmeldungen überprüft und gespeichert werden. Die Rückmeldung an den jeweiligen Client, ob die Anmeldung erfolgreich war oder nicht, stellt der Server in eine *Named Pipe*.

Message Queue

Die Message Queue wird vom Server erzeugt. Beachten Sie, daß das Clientprogramm keine Message Queue selbstständig erzeugen darf, sondern mit einer Fehlermeldung abbrechen muß, wenn die Queue nicht existiert. Der Server erwartet, daß Nachrichten, die er vom Client empfängt, die folgende *Nachrichtenstruktur* haben:

- Der Message Typ entspricht der *lva_nummer*.
- Es folgt die *matrikel_nummer* als String!
- Es folgt ein Byte, in dem codiert ist, ob es sich um eine Anmeldung 'A' oder eine Abmeldung/Stornierung 'S' (Aufruf mit Option -x) handelt.
- Es folgt ein String der Länge PATH_MAX mit dem Namen der Named Pipe für die Rückmeldung vom Server.

Named Pipe

Das Clientprogramm ermittelt durch Aufruf von *void tmpnam(char *s)* den Namen der Named Pipe, wobei als Parameter *s* ein Zeiger auf einen String der Länge PATH_MAX an die Funktion zu übergeben ist. Anschließend muß die Named Pipe vom Client erzeugt werden.

Der Name der Named Pipe wird dem Server über die Nachricht, die in die Message Queue gestellt wird, bekannt gegeben. Das Clientprogramm erhält vom Server über eine Named Pipe die Rückmeldung, ob die Transaktion (Anmeldung bzw. Abmeldung) erfolgreich war oder nicht. Wenn sie erfolgreich war, wird das Byte 'O' geschickt, im Fehlerfall das Byte 'F'.

Nach Auslesen der Rückmeldung muß das Clientprogramm die Named Pipe entfernen.

Allgemein

- Geben Sie zum Abschluß als Benutzerdialog *Transaktion erfolgreich* bzw. *Transaktion nicht erfolgreich* aus.
- Zur Ausgabe von Fehlermeldungen und zum Terminieren des Programms im Fehlerfall verwenden Sie die Funktion *void error_exit(char *msg)*.

Implementieren Sie das Clientprogramm *anmeld*. Vervollständigen Sie hierzu das folgende Programmgerüst!

```
/** anmeld.c                                          **/

#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <limits.h>

/** Message Queue Key und Permissions sind bekannt     **/

#define MQ_KEY  1821
#define MQ_PERM 0666
#define NP_PERM 0666

/**                                                    **
 ** Messagetyp fuer Queue definieren!                  **
 **                                                    **/
```

```
/**                                                    **
 ** Prototyp fuer Funktion "processarg" deklarieren!   **
 **                                                    **/


/** Funktion error_exit im Fehlerfall aufrufen         **/

static char *cmnd;

void error_exit (char *msg)
{
   (void) fprintf(stderr, "%s: %s: %s\n", cmnd, msg, strerror(errno));
   exit(EXIT_FAILURE);
}
```



```
int main(int argc, char **argv)
{
  /**                                                  **
   ** Ergaenzen Sie die benoetigten Variablen          **
   ** Achten Sie darauf, den richtigen Typ zu waehlen! **
   **                                                  **/
```

```
  cmnd = argv[0];

  /**                                                  **
   ** Argumente durch Aufruf von "processarg" verarbeiten. **
   **                                                  **/
```

```
  /**                                                  **
   ** Name fuer Named Pipe holen und speichern         **
   ** Named Pipe anlegen                               **
   **                                                  **/

  (void) tmpnam(                                );

  if (mkfifo(                                , NP_PERM) == -1)

     error_exit("Cannot create fifo");

  /**                                                  **
   ** Nachricht an Server schreiben.                   **
   ** Message Queue existiert schon!                   **
   **                                                  **/
```