Aufgabe 2: (60 Punkte)

a) Schreiben Sie ein Programm `msgd` (Message-Daemon), das einen Mitteilungstext bei Anfragen über eine Netzverbindung ausgibt. Der Text wird dem Programm über eine Konfigurationsdatei zur Verfügung gestellt.

Dem Programm `msgd` kann als Parameter der Name der Konfigurationsdatei übergeben werden. Bei Aufruf ohne Parameter wird die Datei `msgd.conf` verwendet.

Das Programm soll folgendermaßen arbeiten:

- `msgd` startet zuerst einen Sohnprozess, der die eigentliche Arbeit im Hintergrund übernimmt (deshalb auch Daemon-Prozess genannt), gibt dessen Prozess-Id aus ("`msgd startet, PID=...`") und terminiert dann sofort wieder. Die eigentliche Arbeit wird in der Funktion `void msgserver()` erledigt.
- Funktion `msgserver`: der Text aus der Konfigurationsdatei wird unter Verwendung der Funktion `char *getconf(int *size)` eingelesen. Anschliessend werden TCP-Verbindungen von beliebigen IP-Adressen auf Port 9999 erwartet.
- Es wird jeweils eine Verbindung angenommen, der konfigurierte Text auf die Verbindung ausgegeben und anschliessend die Verbindung beendet. Es sollen bis zu 20 Verbindungsanforderungen warten dürfen.
- Funktion `getconf`: Die Funktion ermittelt die aktuelle Größe der Konfigurationsdatei, allokiert den dafür benötigten Speicher, liest die komplette Konfigurationsdatei in diesen Speicherbereich ein und gibt einen Zeiger auf den Speicherbereich als Ergebnis zurück. Im Fehlerfall wird NULL geliefert. Im Parameter `size` wird die Länge des Textes zurückgegeben.
- Änderungen der Konfigurationsdatei können dem Daemon-Prozess durch ein SIGHUP-Signal mitgeteilt werden. Der Empfang des Signals soll bewirken, dass ab der nächsten TCP-Verbindungsanfrage der neue Text ausgegeben wird. Die Textausgabe auf einer gerade bestehenden Verbindung darf durch die Neukonfiguration nicht zerstört werden. Es muss entweder noch der alte oder schon der neue Text ausgegeben werden.
  Bedenken Sie, dass ein neuer Text eine andere Länge haben kann. Eine Neukonfiguration des Textes soll während der Laufzeit des Daemon-Prozesses beliebig oft möglich sein.
- Sie können davon ausgehen, dass der Name der Konfigurationsdatei nicht länger als 255 Zeichen ist. Die Länge der Datei selbst kann beliebig sein.

Auf den folgenden Seiten finden Sie ein Gerüst für das beschriebene Programm. In den Kommentaren sind nur die wesentlichen Aufgaben der einzelnen, zu ergänzenden Programmteile beschrieben, um Ihnen eine gewisse Leitlinie zu geben. Es ist überall sehr großzügig Platz gelassen, damit Sie auch weitere notwendige Programmanweisungen entsprechend Ihrer Programmierung einfügen können.

Einige wichtige Manual-Seiten liegen bei - es kann aber durchaus sein, dass Sie bei Ihrer Lösung nicht alle diese Funktionen oder ggf. auch weitere Funktionen benötigen.


```
/* includes */
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/stat.h>


/* Funktionsdeklarationen, globale Variablen */
void sighup_handler();
void msgserver();
char *getconf(int *);
static char cfile[256];
static int reload_msg = 1;
```

```
/* Funktion main */
int main (int argc, char **argv)
/* Funktionsdeklaration: 1/2A, return/exit aus main 1/2A */
{
  /* Argumente prüfen */
    if (argc == 1) {            /* 3 P */
        strcpy(cfile, "msgd.conf");
    } else if (argc == 2) {
        strcpy(cfile, argv[1]);
    } else {
        fprintf(stderr, "usage: msgd [cfile]\n");
        exit(EXIT_FAILURE);
    }


    /* Serverprozess starten */
    {
      pid_t pid;

      pid = fork();          /* fork und switch/if: 3 P */
      if (pid == -1 ) {             /* Fehlerbeh.: 1 P */
          perror("fork msgd-server");
          exit(EXIT_FAILURE);
      } else if ( pid == 0 ) {
          msgserver();   /* Funktionsaufr. im Sohn: 1 P */
      } else {                 /* Ausgabe im Vater: 1 P */
          printf("msgd startet: PID = %d\n", pid);
      }
      return 0; /* 0,5A bei main */


    }
} /* Ende Funktion main */
```

1A

3A

6P

A: 4
P: 6


```
/* msgserver-Funktion */
void msgserver()
{
    /* Variablendefinitionen */


    char *message;       /* Text der auszugebenden Meldung */
    int msg_size;        /* Laenge der Meldung */
    int sock_accept;     /* Socketdeskriptor der angenommenen Verb. */
    int sock_listen;
    struct sockaddr_in sa_listen;
    struct sockaddr_in sa_accept;


    /* Socket oeffnen, binden, usw. */
           /* sock_listen richtig def. 0,5 S */
     if ( ( sock_listen =                        /* 1S */
              socket(PF_INET, SOCK_STREAM, 0) ) < 0 ) {
          perror("socket");            /* 0,5 S */
          exit(1);
      }


    /* struct richtig def.:                          1S */
    sa_listen.sin_addr.s_addr = INADDR_ANY;      /* 0,5S */
    sa_listen.sin_family = AF_INET;              /* 0,5S */
    sa_listen.sin_port = htons(9999);            /* 1S */
                       /* davon 0,5P fuer das htons */
    if ( bind(sock_listen, (struct sockaddr *) &sa_listen,
          sizeof sa_listen) < 0 ) {              /* 1,5S */
        perror("bind");                          /* 0,5S */
        exit(1);
    }


    listen(sock_listen, 20);
```

2S

5S

1S

S:8

```
/* Signal-Handler für SIGHUP einrichten */
{
struct sigaction act;              /* 1I */
act.sa_handler=sigchld_handler;    /* 1I */
sigemptyset(&act.sa_mask);         /* 1I */
act.sa_flags = 0;                  /* 1I */
sigaction(SIGHUP, &act, NULL);     /* 2I */
}
/* Deklaration von getconf vorne: 0,5G
    initialer Aufruf von getconf: 1G
    Fehlerbhandlung des Aufrufs: 0,5G
    while-Schleife: 1G
    getriggerter Aufruf ohne Konflikt mit Signalhdlr.: 3G
    (entweder einfach über globales flag (Def!) oder
    aufruf aus Signalhandler und bei Ausgabe Signale
    richtig maskiert
  -2P wenn config-file immer gelesen wird
*/

/* Server-Schleife */
while (1) {

    if (reload_msg) {
        /* get message: Aufruf + Parameter*/
        if ( (message = getconf(&msg_size)) == NULL ) {
            fprintf(stderr, "getconf failed\n");
            exit(EXIT_FAILURE);
        }
        reload_msg = 0;
    }
```

6I

g 6G

I: 6
G: 6


```
    /* Verbindung annehmen */
    {
    int addr_size = sizeof sa_accept;
        /* korrekter Ptr. im accept: 0,5S */
    if ( ( sock_accept = accept(sock_listen,     /* 1S */
                (struct sockaddr *) &sa_accept, /*+Def 1S */
                &addr_size) ) < 0 )
        /* NULL als sockaddr und addr_size auch ok! */
     {
        if (errno == EINTR) {                    /* 1S */
                continue;
        }
        perror("accept");                       /* 0,5S */
        exit(EXIT_FAILURE);
    }
    }

    /* Text ausgeben */
    {

      FILE *conn;
      int i;

      if ((conn = fdopen(sock_accept, "w")) == NULL ) {
        perror("fdopen socket");
        exit(EXIT_FAILURE);
      }
      for(i=0; i<msg_size; i++) putc(message[i], conn);

      fclose(conn); /* alternativ fflush und close */

    }
  } /* Ende Server-Schleife */
} /* Ende msgserver-Funktion */
```

s 4S

s 2S

S:6

```
/* Konfigurationsdatei einlesen: Funktion getconf */
char *getconf(int *s)
{
  /* Variablendefinitionen */
    static char *buffer = NULL;
    struct stat statbuf;
    int i;
    FILE *cf;
  /* Groesse der Datei ermitteln */
    if (stat(cfile, &statbuf) < 0 ) { /* Def. 1G, Aufr. 2G */
        perror("stat cfile");         /* Fehler 1G */
        return NULL;
    }
  /* Speicher allokieren */
    if ( (buffer = realloc(buffer, statbuf.st_size)) == NULL ) {
        perror("realloc"); /* Def. 0,5G, free 0,5G, malloc 2G */
        return NULL;                  /* Fehler 1G */
    }
  /* Datei einlesen */
    if ((cf=fopen(cfile, "r")) == NULL ) {         /* 1G */
        perror("fopen cfile");            /* Fehler abf.1G */
        return NULL;
    }       /* Def. buffer 0,5G */
    if (fread(buffer, 1, statbuf.st_size, cf) !=
                statbuf.st_size) {   /* korrekt einlesen 2G */
        fprintf(stderr, "getconf: file too short???\n");
        return NULL;             /* fehlerhaftes Lesen: 0,5G */
    }
    fclose(cf); /* 1G - in Musterlsg. vergessen => Pkt.
        pauschal bei Addition vorne draufgerechnet */
  /* Ergebnis liefern */
    *s = statbuf.st_size;
    return buffer ;
} /* Ende Funktion getconf */
```

4G

4G

5G

+1G

2G

G:15 +1


```
  /* signal-handler für SIGHUP */
  /* Deklaration vorne + hier 1I */
  void  sighup_handler() {
      reload_msg = 1;
  }
  /* triggern oder direkter Aufruf von getconf: 1I */
```

i 2I

b) Schreiben Sie ein Makefile zum Erzeugen des `msgd`-Programms.

Ein Aufruf von

```
make msgd
```

soll - falls erforderlich - die aktuelle Version der Quelldatei aus der RCS-Datei "auschecken" und das Programm erzeugen, ein Aufruf von

```
make clean
```

soll das `msgd`-Programm und evtl. bei vorherigen make-Läufen erzeugte `.o`-Dateien entfernen.

```
msgd: msgd.c                    /* Abhängigk.: 1M */
    ${CC} -o msgd msgd.c   /* Kommando: 2M */

msgd.c:                 /* RCS: 2M */
  co msgd.c

clean:                        /* 1M */
    rm -f msgd     /* -0,5 wenn -f fehlt */
```

6M

I: 2
M: 6

Aufgabe 3: (18 Punkte)

a) Skizzieren Sie (graphisch) die Abbildung von einer logischen Adresse in eine physikalische Adresse in einem System mit Segmentierung.

6 Punkte

logische Adresse (Segmentnummer | Offset): 2P

Segmenttabelle - Segmentdescriptor: 1P

Segmentlänge gegen Offset prüfen: 1P, Trap: 0,5P

Segmentbasis aus deskriptor 0,5P

Addition von Offset = phys. Addr: 1P

b) Was muss das Betriebssystem tun, wenn aufgrund von Hauptspeichermangel ein Segment ausgelagert werden soll? Was passiert, wenn der Prozess nach dem Auslagern auf die Daten des Segments zugreift?

4 Punkte

Anwesenheitsbit auf 0: 1P

Platz auf Hintergrundspeicher suchen und

Segment rauskopieren: 1P

alternativ: 0,5P für "Auslagerungsstrategie

Segment-fault: 0,5P

Platz im HSP suchen, einlagern: 0,5P

Anwesenheitsbit auf 1: 0,5P

Befehl wieder aufsetzen: 0,5P

c) Der Inhalt eines Segments soll als Nachricht von einem Prozess an einen anderen mittels copy-on-write-Technik übertragen werden. Was muss das Betriebssystem tun (z.B. was ist in welchen Datenstrukturen zu ändern) wenn:

1. die Nachricht von Prozess 1 abgeschickt wird (send)
2. die Nachricht von Prozess 2 empfangen wird (receive)
3. Prozess 2 nach dem Empfang den Segmentinhalt modifiziert
4. Prozess 2 den Segmentinhalt nur liest, dann terminiert und anschliessend modifiziert Prozess 1 den Segmentinhalt

8 Punkte

1. Segment als read-only markieren
   1P
2. Segmentbasis in Segmentdescriptor on Proz. 2 eintragen
   (einblenden) und als read-only markieren
   2P
3. segment-fault (0,5P),
   Segment wird kopiert(0,5P),
   Segmentbasis der Kopie wird in Segmentdeskriptor von
      Proz. 2 eingetragen (0,5P)
   in den Segmentdeskriptoren beider Prozesse wird das
      Segment als read-write markiert (1P)
   getrappter Befehl von Proz. 2 wird wiederholt. (0,5P)
   Summe von 3: 3P
4. lesen macht nichts, (0,5P)
   beim Terminieren bekommt Proz. 1 das Segment wieder
   read-write markiert (1P)
   bei der anschliessenden Modifikation passiert
   nichts (0,5P)
   Summe von 4: 2P

Aufgabe 4: (12 Punkte)

Skizzieren Sie in einer programmiersprachen-ähnlichen Form die Programmierung der zwei Funktionen *put* und *get* (entspricht *store* und *fetch*), die in der üblichen Art und Weise als Erzeuger und Verbraucher auf einem Puffer fester Größe (Bounded Buffer) operieren.

Koordinieren Sie die Funktionen mit Hilfe von Semaphoren.

Beschreiben Sie kurz die Bedeutung der von Ihnen eingesetzten Semaphore und welche Werte sie initial haben müssen.

Gehen Sie davon aus, dass auch jeweils mehrere Erzeuger und Verbraucher gleichzeitig einen Zugriff versuchen könnten.

store() {		fetch() {	
1	P(freie Plätze)	1	P(belegte Plätze)
0,5	P(lock)	0,5	P(lock)
0,5	speichern	0,5	entnehmen
0,5	V(lock)	0,5	V(lock)
1	V(belegte Plätze)	1	V(freie Plätze)
}		}	
Summe: 3,5		Summe 3,5	

freie Plätze: initial = Puffergröße, enthält die aktuelle Zahl freier Plätze,
blockiert store wenn der Puffer voll ist: 2P

belegte Plätze: initial = 0, enthält aktuelle Zahl belegter Plätze,
blockiert fetch wenn der Puffer leer ist: 2P

lock: zum gegenseitigen Ausschluss mehrere store oder fetch Operationen. Man könnte auch zwei verschiedene locks haben, da store und fetch untereinander nicht kollidieren.: 1P