INTERPROZESSKOMMUNIKATION

Andreas Mieke & Rafael Lazenhofer

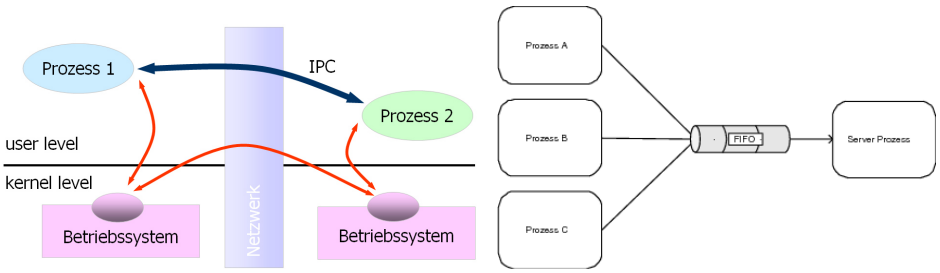
4. DEZEMBER 2017

HTBL HOLLABRUNN
5BHEL

Inhaltsverzeichnis

1.Aufgabenstellung

Systemprogrammierung Beispiel 15:



- **Angabe:**
  Es ist Herbst und Bauer Sepp möchte sein Fallobst zu Most verarbeiten. Einige Nachbarskinder und seine eigenen Kinder sammeln Äpfel und bringen sie zu ihm in die Scheune. Sobald die Kinder 100kg Äpfel gesammelt haben, kann Bauer Sepp mit dem Mosten beginnen.
  Technischer Hintergrund:
  In diesem Beispiel stellt der Bauer Sepp und seine Mostmaschine einen Server dar (Apfelverwertung). Die Kinder entsprechen Clients, die einen Dienst in Anspruch nehmen (Abnahme der Äpfel). Server und Clients sind als selbständige und unabhängige Programme implementiert. Die Abgabe der Äpfel repräsentiert die Kommunikation von Client zu Server, die Antwort von Bauer Sepp entspricht der Kommunikation vom Server zum Client. Der Server (Bauer Sepp) terminiert, wenn er von den Kindern mehr als 100kg Äpfel erhalten hat. Pro Aufruf bringen die Kinder zwischen 5kg und 10kg Äpfel (Zufallsgenerator). Die Ausgabe des Servers sieht wie folgt aus (der Name des Clients wird ausgegeben):

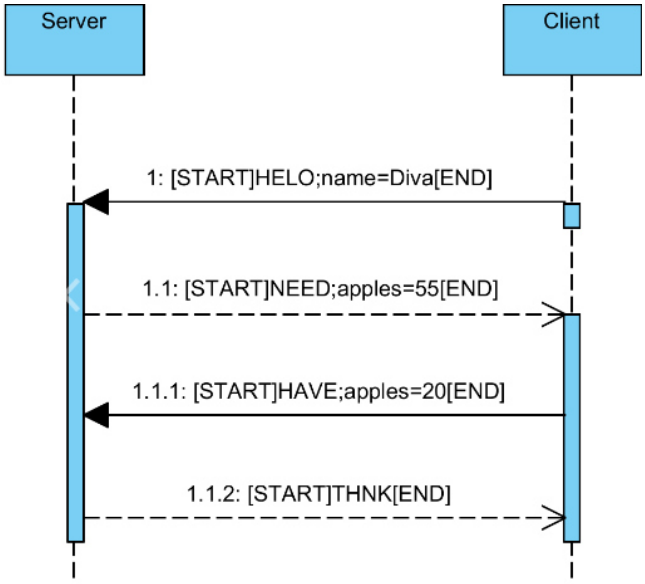
```
Bauer Sepp: heute ist Mosttag ...sammelt bitte Aepfel

Bauer Sepp: habe 6 kg Aepfel von diva erhalten
Bauer Sepp: jetzt sind es insgesamt 6 kg Aepfel

Bauer Sepp: habe 8 kg Aepfel von diva erhalten
Bauer Sepp: jetzt sind es insgesamt 14 kg Aepfel

....
```

  Pro Aufruf gibt der "Kinderprozess" folgende Meldung aus (Aufforderung mehr Äpfel zu sammeln):
  `Kinder: Vater Sepp benoetigt noch 94 kg Aepfel bis er mosten kann`

- **Abgabe:**
  Abzugeben sind ein dokumentierter SourceCode und ein Funktionsnachweis in Form von ScreenShots, Unix-Befehlen, Programmausgaben, …

2. Realisierung

Die Aufgabenstellung aus Punkt 1. wurde wie im Folgenden Diagramm dargestellt realisiert:



[START] = "$ID:" wobei ID bei 0 beginnt und bei jeder Message des Clients inkrementiert wird, der Server nutzt diese für die Antworten.
[END] = "#\r\n" wird verwendet um die Messages eindeutig voneinander unterscheiden zu können.

Wenn der Server noch Äpfel braucht antwortet er auf die Message 1.1.1 mit 1.1, ansonsten mit 1.1.2

2.Dokomentierten Source Code & Erläuterung

2.2. Client

2.2.1. Erläuterung

Der Client sind die Kinder die dem Bauer Sepp bzw. dem Server die Äpfel bringen. Als Startparameter benötigt der Client die IPv4 Adresse oder den Hostname des Servers und dessen Port. Des weiteren benötigt das Programm auch den Namen des Kindes bzw. des Clients.

2.2.3. Source Code

```
/************************
 Name: Client - Child
 Autor: Rafael Lazenhofer
 Version: 1.0
 Date: 01.12.2017
 ***********************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <time.h>
//Unterprogramm zum Error handling
void error(const char *msg)
{
    perror(msg);
    exit(0);
}
// Messagetyp auswahl
int value(char *str) {
    char *words[]={"HELO","NEED","THNK"}, str2[10];
    int i,j=0;
    bzero(str2,10);
    /*********************************
     Aussortieren für der unwichtigen
     Zeichen
    *********************************/
    for(i=6;i<10;i++)
    {
        if((strcmp(str,";"))==0)
            break;
        str2[j]=str[i];
        j++;
    }
    /************************************
     Wert in Messagetyp in Integer
     umwandeln für switch im Hauptprogramm
    ************************************/
    for (i = 0; i < sizeof words/sizeof words[0]; i++) {
        if ((strcmp(str2, words[i]) == 0)) {
            return i;
        }
    }
}
```

```
int main(int argc, char *argv[])
{
    /*********************
    Varibalen definierten
    **********************/
    int sockfd, portno, n, random_variable, id=1, m;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];
    char mesg[256]="$0001:HELO;name=Diva#\r\n";

    /*********************************
    Abfrage ob alle Argumente beim Start
    mit geben wurden
     wenn diese nicht mit gelfiefert
    werden wird das Programm abgebrochen
    *********************************/
    if (argc < 4) {
       fprintf(stderr,"usage %s hostname port child-name\n", argv[0]);
       exit(0);
    }
    /*************************************************
     atoi() macht aus einem String eine Int Zahl

    (Hier wird der String in dem die Portnr. enthalten ist
    umgewandelt in eine Int Zahl)
     *************************************************/
    portno = atoi(argv[2]);
    /*************************************************************************
     socket()
     AF_INET => IPv4
     SOCK_STEAM => Unterstützt eine zuverlässig Byte-Stream-Kommunikation
     0 => Für das nötige Protokoll
     *************************************************************************/
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    /*********************************
     Überprüfung ob Socket zugänglich
    *********************************/
    if (sockfd < 0)
        error("ERROR opening socket");

    /*********************************
     Überprüfen ob Host zugänglich ist
     z.B.: 127.0.0.1 //localhost
    *********************************/
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    /*************************************
     bzero()
     Setzt alle Elemente auf 0 von serv_addr
     *************************************/
    bzero((char *) &serv_addr, sizeof(serv_addr));

    serv_addr.sin_family = AF_INET; //IPv4
    /*************************************
     bcopy()
     eine Anzahl von Zeichen in einer
     andere Zeichenfolge kopieren
    *************************************/
```

```
    bcopy((char *)server->h_addr,
          (char *)&serv_addr.sin_addr.s_addr,
          server->h_length);

    serv_addr.sin_port = htons(portno);//Port setzen
    /*Zum Server verbinden*/
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr))<0)
        error("ERROR connecting");

/*******************************
     Beginn der Kommunikation
     zwischen Client & Server
*******************************/
    printf("CHILD: %s\n",argv[3]);
    do
    {

     bzero(buffer,256);
    /*********************************************
     Message auswahl
     Aufbau:
     [Befehl-NR][START][NACHRICHT][END]
     [START] . . . . ID wird jedes mal inkrementiert

     1.1 HALO . . . . senden vom Namen z.B. Diva
     1.1.1 HAVE . . . wie viele Aepfel das Kind hat
     *********************************************/
     m=value(mesg);
     switch(m)
     {
         case 0:
                sprintf(buffer, "$%04d:HELO;name=%s#\r\n",id++,argv[3]);
                printf("%s: %s",argv[3],buffer);
         break;
         case 1:
                /*******************************
                Als Zufallsgenertator wird eine
                Zeitabgfrage verwendet
                *******************************/
                srand(time(NULL));
                random_variable = rand()%20;
                sprintf(buffer,
"$%04d:HAVE;apples=%d#\r\n",id++,random_variable);
                printf("%s: %s",argv[3],buffer);
         break;
         case 2:
         printf("Farmer Sepp has enough apples!!!\n");
         exit(0);
         break;

         default:
         puts("Unkown Message!");
         exit(0);
         break;

     }
```

```
// Schreiben
       n = write(sockfd,buffer,strlen(buffer));

/*********************************
     ERROR Handling falls beim Befehl
     write ()
     etwas fehlschlägt
*********************************/
    if (n < 0)
         error("ERROR writing to socket");
    // Lesen
    bzero(mesg,256);
    n = read(sockfd,mesg,255);

/*********************************
 ERROR Handling falls beim Befehl
 read()
 etwas fehlschlägt
*********************************/
    if (n < 0)
         error("ERROR reading from socket");
    printf("Sepp: %s\n",mesg);
    }while(1==1);

      close(sockfd);
    return 0;
}
```

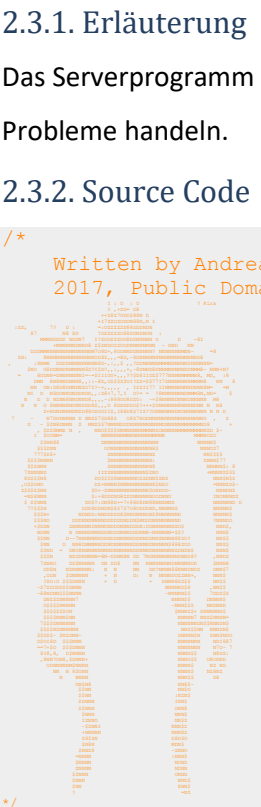
2.3. Server

2.3.1. Erläuterung

Das Serverprogramm stellt den Bauer Sepp dar. Er kann mehrere Clients mittels Threads ohne Probleme handeln.

2.3.2. Source Code

```
/*
    Written by Andreas Mieke          ,          L, do you know
    2017, Public Domain                          Gods of death love apples?
```



```
*/

/*
    Here begins the funny configuration stuff! Actually you just have to
set the
    amount of apples Ryuk ? ugh, I mean Sepp ? will get, the port he will
listen
    to and other stuff once I come up with it.
*/

#define APPLES_WANTED   100             // How many apples does Ryuk want?
#define PORT            8666            // On which port should he listen?
#define NUM_THREADS     10              // How many clients at one time?
#define RYUK_NAME       "Ryuk"          // If someone wants to call me
Sepp,
                                        // change this line -.-
#define BUFFER_SIZE     2048            // String buffer for receiving

/*
    Thank you, human, for the configuration. Now the real magic begins!
*/

#define BACKLOG         10

#include <stdio.h>
#include <string.h>                     // strlen
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>                  // inet_addr
#include <unistd.h>                     // write
#include <signal.h>                     // signal

#include <pthread.h>                    // Threading

#include <assert.h>                     // assert
```

```c
long apples_left    =   APPLES_WANTED;  // Holds the number of apples to
collect
int socket_fd;                          // Socket file descriptor
pthread_mutex_t apple_lock;             // Mutex for the apples_left
counter

void *handle_socket(void *l_socket_fd);
void handle_int(int dummy);

int main(int argc, char const *argv[])
{
    int new_socket_fd, c, *new_sock;
    struct sockaddr_in server, client;
    char *msg;

    // SIGINT handler registration for graceful termination
    signal(SIGINT, handle_int);

    // Init the mutex for the apple_left counter
    if (pthread_mutex_init(&apple_lock, NULL))
    {
        perror("Could not create mutex");
        return 1;
    }

    // Create the server socket
    socket_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd == -1)
    {
        perror("Could not create socket");
        return 1;
    }

    // Address and port
    server.sin_family       = AF_INET;
    server.sin_addr.s_addr  = INADDR_ANY;
    server.sin_port         = htons(PORT);

    // Bind the socket
    if (bind(socket_fd, (struct sockaddr *)&server, sizeof(server)) < 0)
    {
        perror("Could not bind to socket");
        return 1;
    }
    puts("Bound to socket!");

    // Start listening on the bound socket
    listen(socket_fd, BACKLOG);

    puts("Waiting for connections...");
    puts(RYUK_NAME ": Please bring me some apples!");
    c = sizeof(struct sockaddr_in);
    // Wait for new connections and accept them
    while    ((new_socket_fd = accept(    socket_fd,
                                          (struct sockaddr *)&client,
                                          (socklen_t *)&c)
            ))
    {
        puts("Connection accepted!");

        pthread_t apple_thread;
```

```
        // Allocate memory for the thread argument
        new_sock = malloc(sizeof(int));

        // Set the argument for the thread to the socket file descriptor
        // of the newly accepted connection, so the thread can communicate
        // with the client that was just accepted
        *new_sock = new_socket_fd;

        // Create the thread for the accepted client. This thread will
handle all
        // further communication with this client.
        if (pthread_create( &apple_thread,
                            NULL,
                            handle_socket,
                            (void *)new_sock)
            < 0)
        {
            perror("Could not create socket thread");
            return 1;
        }
        puts("Handler assigned");
    }

    if (new_socket_fd < 0)
    {
        perror("Could not accept connection");
        return 1;
    }

    // There is no valid condition that should bring us here, so
    // assert that we never reach this point of the program, so if we
    // ever do, we can easier debug it.
    assert(!"End of main should never happen");
}

// This function is used to communicate with a specific client
// that was accepted in our main loop, in a separate thread for
// each client.
void *handle_socket(void *l_socket_fd)
{
    int fd = *(int *)l_socket_fd, read_size, msg_id;
    char msg[BUFFER_SIZE], name[64], *tok;

    // Read data from the client
    while ((read_size = recv(fd, msg, sizeof(msg), 0)) > 0)
    {
        // Parse the message we got from the client
        // Look for the valid start sequence of the clients message
        tok = strchr(msg, '#');
        if (tok == NULL ||
            msg[0] != '$' ||
            msg[5] != ':' ||
            strncmp(tok, "#\r\n", 3) != 0)
        {
            // If there is no valid start sequence, print an error
            // and disconnect the client.
            puts("Wrong message format, disconnecting client!");
            close(fd);
            break;
        }

        // The first 4 bytes after the start sequence ($) hold the id of
        // the message that was sent.
```

```
        char id[5] = {msg[1], msg[2], msg[3], msg[4], 0};
        msg_id = atoi(id);

        // Next check the type of the message and the argument:
        // HELO
        if (strncmp(msg+6, "HELO;name=", 10) == 0)
        {
            tok = strchr(msg, '#');
            *tok = '\0';
            tok = strchr(msg, '=');
            strncpy(name, tok+1, sizeof(name));
        }

        // HAVE
        else if (strncmp(msg+6, "HAVE;apples=", 12) == 0)
        {
            tok = strchr(msg, '#');
            *tok = '\0';
            tok = strchr(msg, '=');
            int got = atoi(tok+1);

            // Lock the mutex for the apple_left counter
            pthread_mutex_lock(&apple_lock);
            // Substract apples we got from the apple_left counter
            // This needs to be done while the lock is hold, to ensure
            // that no other thread decrements the counter at the same
            // time, causing a data race.
            apples_left -= got;
            pthread_mutex_unlock(&apple_lock);
            printf(RYUK_NAME ": Got %dkg apples from %s, now I have
%ldkg.\n",
                                got, name, APPLES_WANTED - apples_left);
        }

        // Check if we still need more apples
        if (apples_left > 0)
        {
            // If yes, ask for them
            sprintf(msg, "$%04d:NEED;apples=%ld#\r\n", msg_id,
apples_left);
            write(fd, msg, strlen(msg));
        }
        else
        {
            // If not, we are done. Thank and disconnect the client.
            sprintf(msg, "$%04d:THNK#\r\n", msg_id);
            write(fd, msg, strlen(msg));
            close(fd);
            puts("Enough apples, disconnecting...");
            break;
        }
        memset(msg, sizeof(msg), 0);
    }
    if (read_size == 0)
    {
        puts("Client disconnected!");
    }
    if (read_size < 0)
    {
        perror("Receive failed");
        close(fd);
    }
```

```
    // Cleanup the socket
    free(l_socket_fd);
    return 0;
}

// SIGINT handler
void handle_int(int dummy)
{
    puts("\rReceived SIGINT, exiting now...");
    close(socket_fd);
    pthread_mutex_destroy(&apple_lock);
    exit(0);
}
```

4. Funktionsnachweis

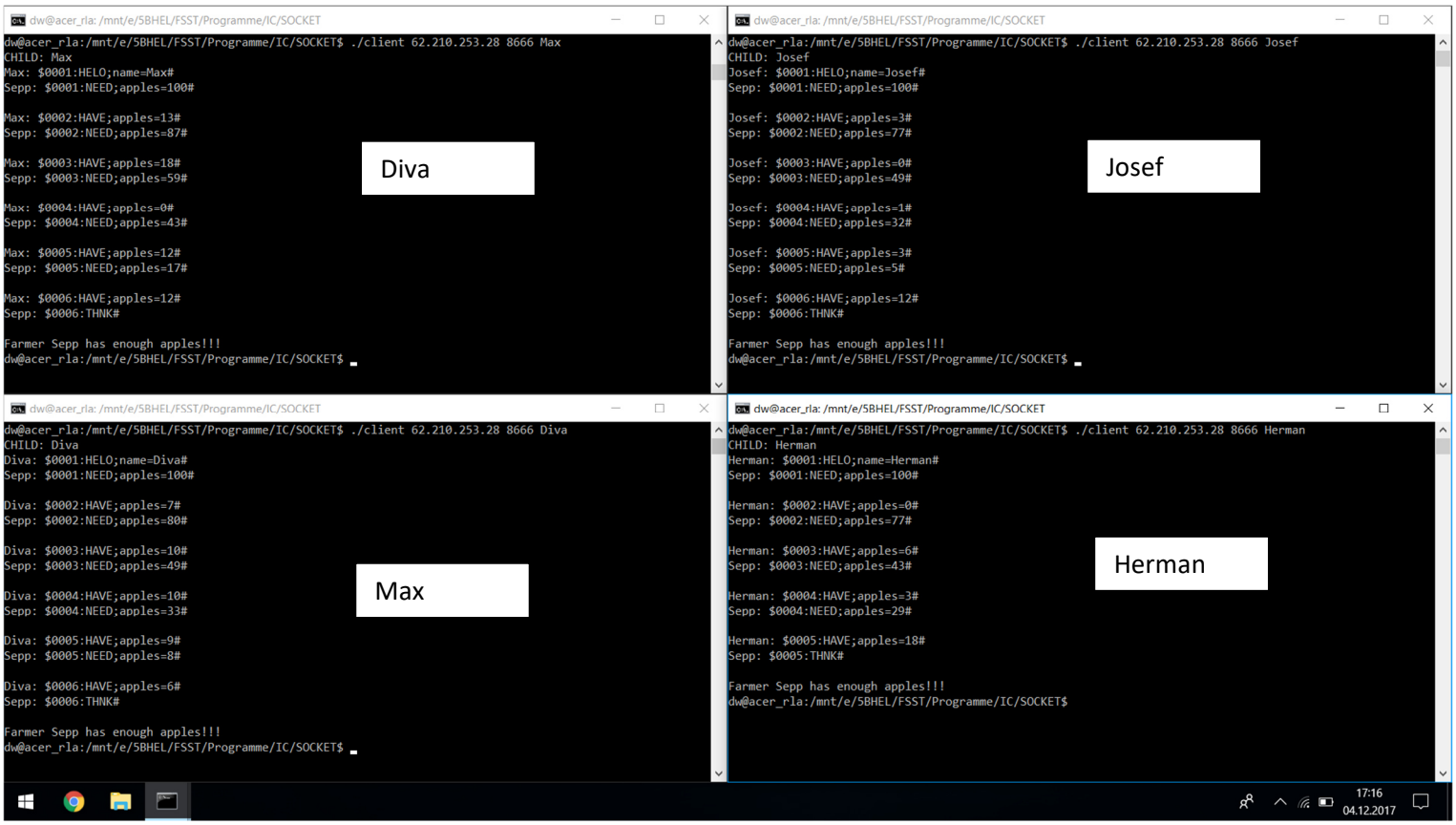
Das Programm wurde auf seine Funktionen geprüft. Alle Ergebnisse werden in den unter punkten beschrieben und mit Screenshots bewiesen.

4.1. Tests

Zu Beginn wurde getestet ob ein Client sich zum Server verbinden kann. Des weiteren ob er Daten vom Server empfangen und zum Server senden kann. Diesen Test hat das Programm bestanden wie im unteren Screenshot zu sehen:

Danach wurde getestet ob der Server gleichzeitigt mehrere Daten von verschiedenen Clients empfangen und senden kann. Diesen Test hat das Serverprogramm einwandfrei bestanden:

Senden von mehrern Clients:



Server Daten: