Dokumentation

IPC – Ampelanlage mit

- Message Queues -

Schule: bbs | me
Otto-Brenner-Schule
Lavesallee 14
30169 Hannover

Projektberater: Herr Dorau

Ersteller: xxx
xxx

Inhaltsverzeichnis

1 Einleitung

Die Ampelsteuerung Hannovers soll zukünftig über Interprocess Communication mittels Message Queues realisiert werden. Hierzu werden Testprogramme (zwei unabhängige Prozesse) entwickelt, die Daten austauschen sollen. Der Clientprozess erstellt die Ampelphasen nach StVO und der Serverprozess gibt die Ampelphasen aus.

Diese Abhandlung beinhaltet die Grundlagen der Interprozesskommunikation und einige Techniken diese zu realisieren. Hierbei wird der Schwerpunkt auf Message Queues gelegt und die Ansätze des zugehörigen Sourcecodes erläutert.

Das System ist Linux-basiert.

Aufgabenstellung

IPC (Interprocess Communication)

- Systemprogrammierung in C
- Implementierung einer Ampelsteuerung
- Client: Ampelautomat nach StVO
- Server: Folgt Ampel vom Client via "Message Queues"
- Ausgabe der Ampelphasen des Client und Server am Bildschirm
- Dokumentation
- Demo

1.1 Versicherung über die eigenständige Anfertigung

Hiermit versichern wir, dass wir die Projektarbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und die Stellen der Arbeit, die im Wortlaut oder im wesentlichen Inhalt anderen Werken entnommen wurden, gekennzeichnet und im Literaturverzeichnis angegeben haben.

Hannover, 29.05.2017 .....

(Unterschrift)

.....

(Unterschrift)

2 Lastenheft

bbs me Otto-Brenner-Schule

Lastenheft

Projekt

Interprocess Communication

Version 1.0

Autor des Dokuments:			
Dateiname:	Lastenheft_FSET14_Nikic_Matthies		
Seitenanzahl:	4	*Die IPC-Profis*	Vertraulich!

Abbildung 1 Lastenheft Seite 1

	Interprocess Communication Lastenheft	bbs me Otto-Brenner-Schule

Inhaltsverzeichnis

Abbildung 2 Lastenheft Seite 2

	Interprocess Communication Lastenheft	bbs me

1 Einleitung

Die Ampelanlagen in Hannover sollen in Zukunft über IPC gesteuert werden.
Testweise soll eine Ampel auf das neue System umgebaut werden.

2 Allgemeines

2.1 Zweck und Ziel dieses Dokuments

Dieses Dokument beinhaltet die Kundenwünsche bezüglich der Anforderungen der Programmierung.

2.2 Verteiler und Freigabe

Rolle / Rollen	Name	Telefon	E-Mail	Bemerkungen
Projektleiter				
Auftraggeber				

3 Rahmenbedingungen

3.1 Gegebenheiten

Eine Fußgängerampel in der Marienstraße steht für den Umbau zur Verfügung.

3.2 Ziele und Nutzen des Anwenders

Die Ampeln sollen nach einem festen Muster automatisch geschaltet werden. Die Ampelphasen werden über einen Bildschirm ausgegeben.

3.3 Anforderung

3.3.1 Allgemein

Die Systemprogrammierung ist in C. Die Ampeln kommunizieren über einen zentralen Computer.

3.3.2 Client

Die Ampeln werden nach StvO geschaltet.

3.3.3 Server

Der zentrale Computer erhält über Message Queues die Ampelphasen des Clients.

3.3.4 Visualisierung

Ein Bildschirm ist an dem Server angeschlossen. Die Ampelphasen des Clients werden über diesen Bildschirm ausgegeben.

Seite 3 von 4

Abbildung 3 Lastenheft Seite 3

	Interprocess Communication Lastenheft	bbs me Otto-Brenner-Schule

4 Freigabe / Genehmigung

Die Genehmigung dieses Dokuments erfolgt durch Unterschrift des Auftraggebers und des Projektleiters. Zu erbringende Leistungen werden aus diesem Dokument entnommen.

Datum:	
Unterschrift Auftraggeber:	
Unterschrift Projektleiter:	

Seite 4 von 4

Abbildung 4 Lastenheft Seite 4

3 Erwartungshaltung bezüglich der Aufgabenstellung

Zu erwarten ist, dass der Ampel-Anlage gemäß Lasten- und Pflichtenheft ausgelegt ist.

- Die Ampelanlage ist nur auf einem lokalen System und nicht über das Netzwerk lauffähig.
- Die Ampelanlage setzt das Betriebssystem Linux voraus.
- Da der Ampel-Server keinen eigenen Ampelzyklus eigenständig ausführt, wird der Server zu jeder Zeit mit dem Client synchronisiert. Damit sind Zeitverzögerungen minimiert.
- Die Ampelanlage arbeitet nach der aktuellen StVO.
- Die Kommunikation zwischen Client und Server wird mit Message Queues realisiert, die eine Priorisierung der Botschaften zulassen. In dieser Ampelanlage werden keine Priorisierungen benötigt und auch nicht implementiert.
- Die Ampelanlage dient nur zu Lehrzwecken und ist als Softwaremodell in der Programmiersprache C umgesetzt. Eine Überführung in die reale Welt als eigenständige Anlage ist nicht vorgesehen.

4 Theoretische Grundlagen der Aufgabenstellung

4.1 Interprocess Communication

Linux zählt zu den Multitasking-Systemen, was bedeutet, dass mehrere Prozesse neben einander laufen, ohne dass sie kaum von ihrer gegenseitigen Existenz wissen. Greift hierbei ein Prozess auf den Speicherbereich eines anderen Prozesses zu und verändert diesen, kann es zu Programmfehlern oder Programmabstürzen kommen (s. Abb. 5).

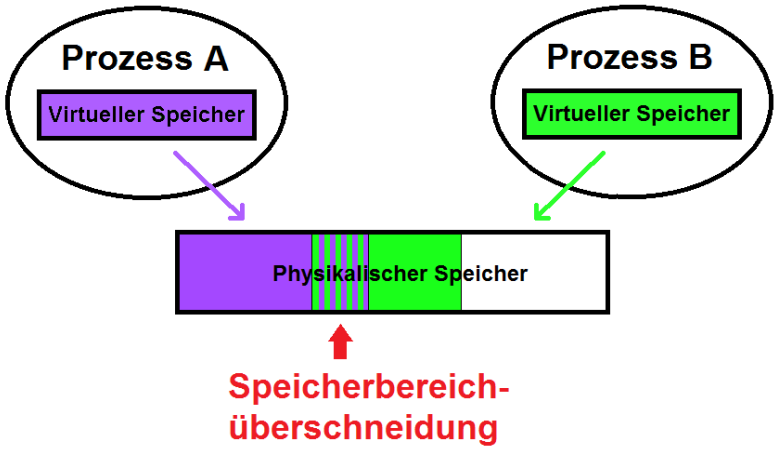

Abbildung 5 Speicherbereichsüberschneidung von zwei Prozessen

Damit die Stabilität der Prozesse und des Gesamtsystems nicht beeinträchtigt wird, muss der Speicher verwaltet werden, damit es zu keinen unerwünschten Speicherübergriffen kommt. Im Betriebssystem übernimmt diese Aufgabe des Speicherschutz die Speicherverwaltungseinheit, kurz MMU (Memory Management Unit) (s. Abb.6).

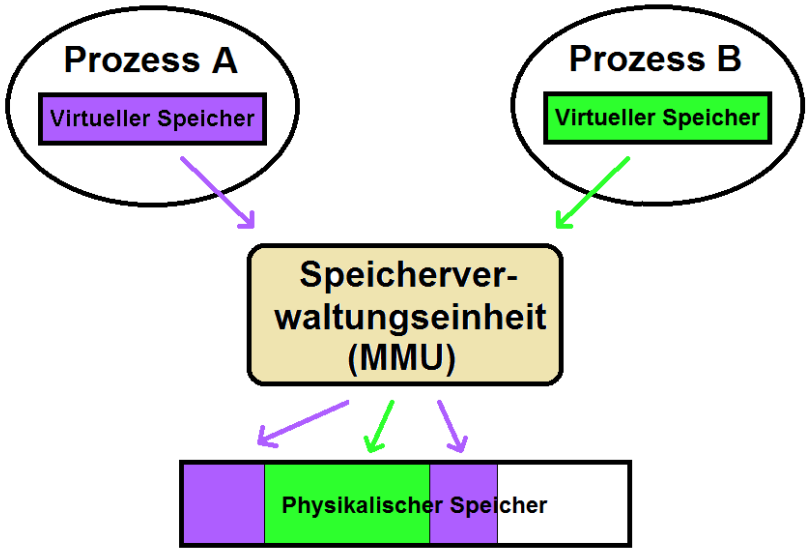

Abbildung 6 Speicherverwaltungseinheit (MMU)

Ist jedoch ein Datenaustausch und somit ein gemeinsam genutzter Speicherbereich, zwischen zwei Prozessen gewollt, muss auf die Interprozesskommunikation zurückgegriffen werden. Für den Datenaustausch sind grundlegend zwei Methoden möglich. Entweder wird ein gemeinsamer Speicherbereich oder ein gemeinsamer Datenkanal verwendet. Bei dem gemeinsamen Speicherbereich schreibt Prozess A seine Daten in den Speicher, währenddessen der Speicher für den anderen Prozess B gesperrt ist. Erst wenn das Schreiben beendet ist, kann der Prozess B auf den Speicher zugreifen (s. Abb.7).

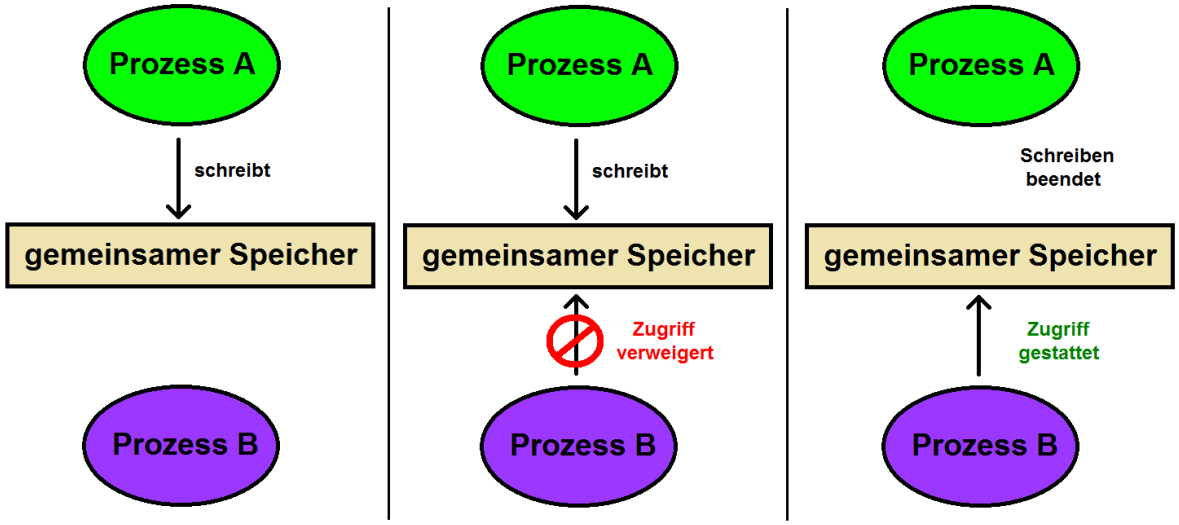

Abbildung 7 Prinzip des gemeinsamen Speichers

Bei dem gemeinsamen Datenkanal werden die Informationen direkt über einen Pufferspeicher von Prozess A an Prozess B übermittelt (s. Abb.8) (vgl. Spinczik, 2007; vgl. Koslowski 2005).

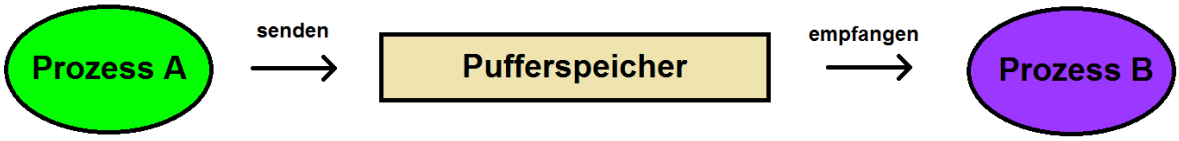

Abbildung 8 Prinzip des gemeinsamen Datenkanals

Techniken für die Interprozesskommunikation sind u.a.:

- Shared Memory
- Pipes
- Streams
- Sockets
- Message Queues

Im weiteren Verlauf werden einige Techniken Beispielhaft betrachtet.

4.1.1 Shared Memory

Bei dieser Interprozesskommunikationstechnik können mehrere Prozesse auf einen gemeinsamen Speicherbereich zugreifen. Dazu muss in den Prozessen der Adressraum des Speichers eingefügt werden, damit sie wissen, um welchen Speicherbereich es sich handelt. Des Weiteren ist darauf zu achten, dass den einzelnen Prozessen die Schreib- und Leserechte zugewiesen werden. Zwingend erforderlich ist eine gegenseitige Verriegelung/ Sperrung der Prozesse. Zwar können mehrere Prozesse

gleichzeitig lesend auf den Speicherbereich zugreifen, aber es darf immer nur ein Prozess zur selben Zeit schreibend zugreifen, damit es zu keinen Fehlern kommt.

4.1.2 Pipes

Man unterscheidet zwei Arten von Pipes, die namenlosen und die benannten Pipes. Für die namenlosen Pipes gelten zwei grundsätzliche Eigenschaften:

- Sie kann ausschließlich zwischen verwandten Prozessen existieren
- Sie hat nur eine Flussrichtung (Halbduplex)

Indem ein Vaterprozess eine Pipe erstellt und dann einen Sohnprozess erzeugt, vererbt er diesem auch die Eigenschaften der Pipe und die beiden Prozess können miteinander kommunizieren. Hierbei ist jedoch die unidirektionale Kommunikation zu beachten. Der eine Prozess kann nur lesen und der der andere nur schreiben. Will man eine bidirektionale Kommunikation aufbauen, benötigt man auch eine zweite Pipe (s. Abb.9) (vgl. Wolf. 2006).

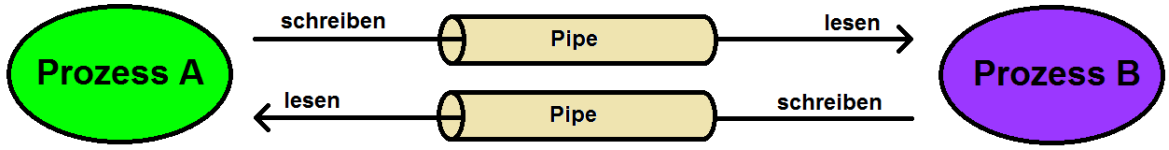

Abbildung 9 Namenlose Pipe

Möchte man nun aber zwischen zwei fremden Prozessen kommunizieren, benötigt man benannte Pipes, auch FIFO (first in, first out) genannt. Wird so eine benannte Pipe erzeugt, wird diese als Datei implementiert, welche im Dateisystem versteckt ist und somit nicht sichtbar. Auch FIFOS sind unidirektional und es muss definiert werden, wer Lesezugriff und wer Schreibzugriff besitzt. Klassisches Beispiel ist eine Server-Client-Architektur mit Drucker. Prozess A und Prozess B wollen etwas ausdrucken. Der Druckauftrag, der zuerst in die Pipe geschrieben wird, wird auch als erstes ausgelesen. Der Druckauftrag des anderen Prozesses muss dann warten (s. Abb.10) (vgl. Koslowski, 2005).

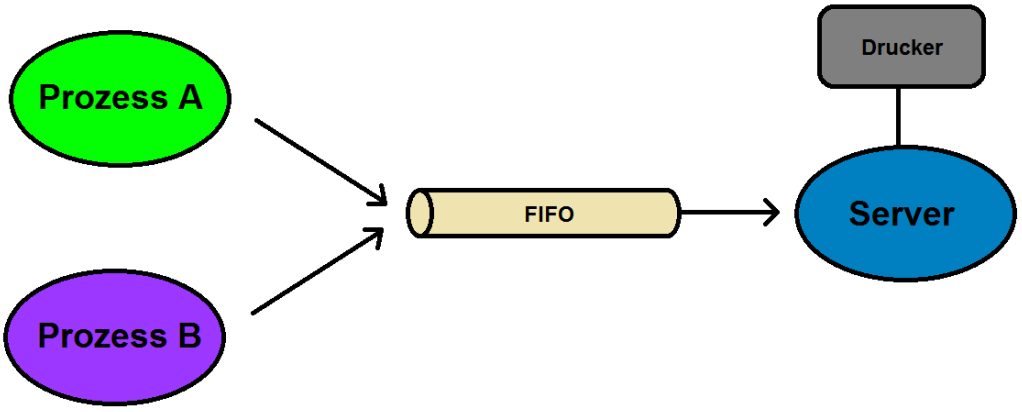

Abbildung 10 Benannte Pipe

4.1.3 Message Queues

Mit dieser Technik können Daten zeitunabhängig zwischen zwei oder mehreren Prozessen ausgetauscht werden. Beim Senden von Nachrichten werden diese aus dem Benutzerdatenbereich in den Betriebssystemdatenbereich kopiert. Beim Empfangen wird dann wiederum die Nachricht aus dem Betriebssystemdatenbereich in den Benutzerdatenbereich verschoben. Dieser Zwischenspeicher im Betriebssystemdatenbereich wird Message Queue genannt oder zu Deutsch Nachrichtenwarteschlange.

Bei der Nachrichtenverarbeitung gilt grundsätzlich das FIFO-Prinzip – first in, first out. Nachrichten, die zuerst abgelegt werden, werden auch als erstes wieder abgeholt. (s. Abb.11).

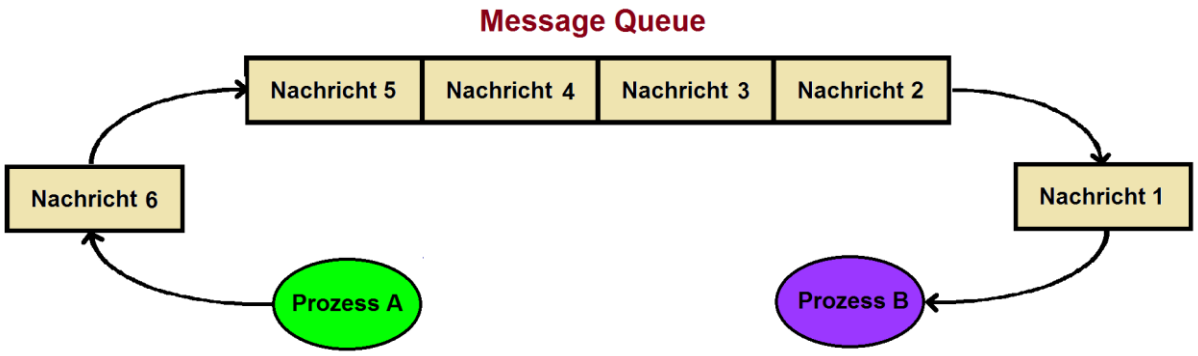

Abbildung 11 Message Queue

Ist eine abweichende Reihenfolge erforderlich, zum Beispiel ist eine Nachricht wichtiger als die andere, können die einzelnen Nachrichten mit Prioritäten versehen werden. Der Empfänger kann dann gezielt nach einer Priorität suchen. Die vorderste Nachricht mit der gesuchten Priorität wird dann aus der Message Queue abgeholt (s. Abb.12) (vgl. Groß, 2005).

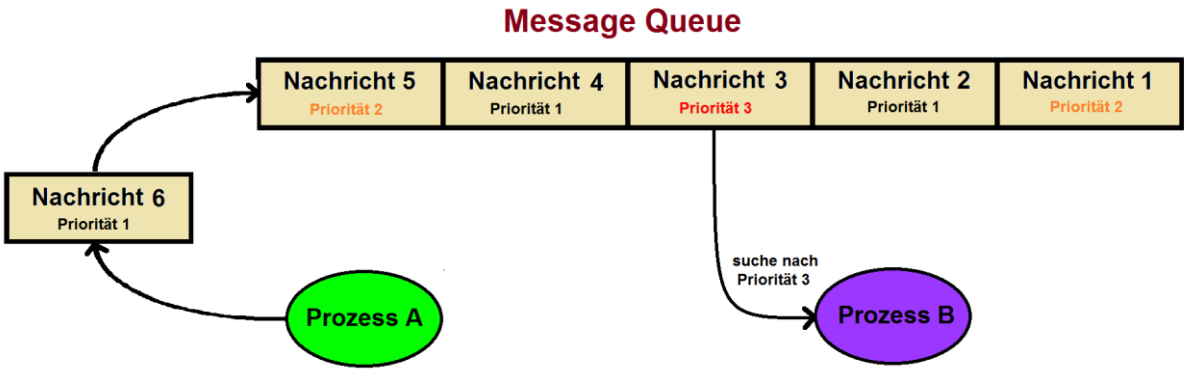

Abbildung 12 Message Queue – Prioritäten

5 Durchführung und Lösung der Aufgabestellung

5.1 Prinzipielle Funktion der Ampelanlage

Die prinzipielle Funktion der Ampelanlage richtet sich nach dem Server-Client-Modell. Der Client ist die Fußgänger- oder Fahrradampel mit einem Umschalttaster und der Server die Fahrzeugampel. Die Ampelphase der Ampelanlage ist in folgender Tabelle zusammengefasst:

Phase	Client	Server
1	Rot	Grün
2	Rot-Gelb	Gelb
3	Grün	Rot

Der Programmablaufplan (PAP) ist nachfolgend dargestellt und beschrieben.

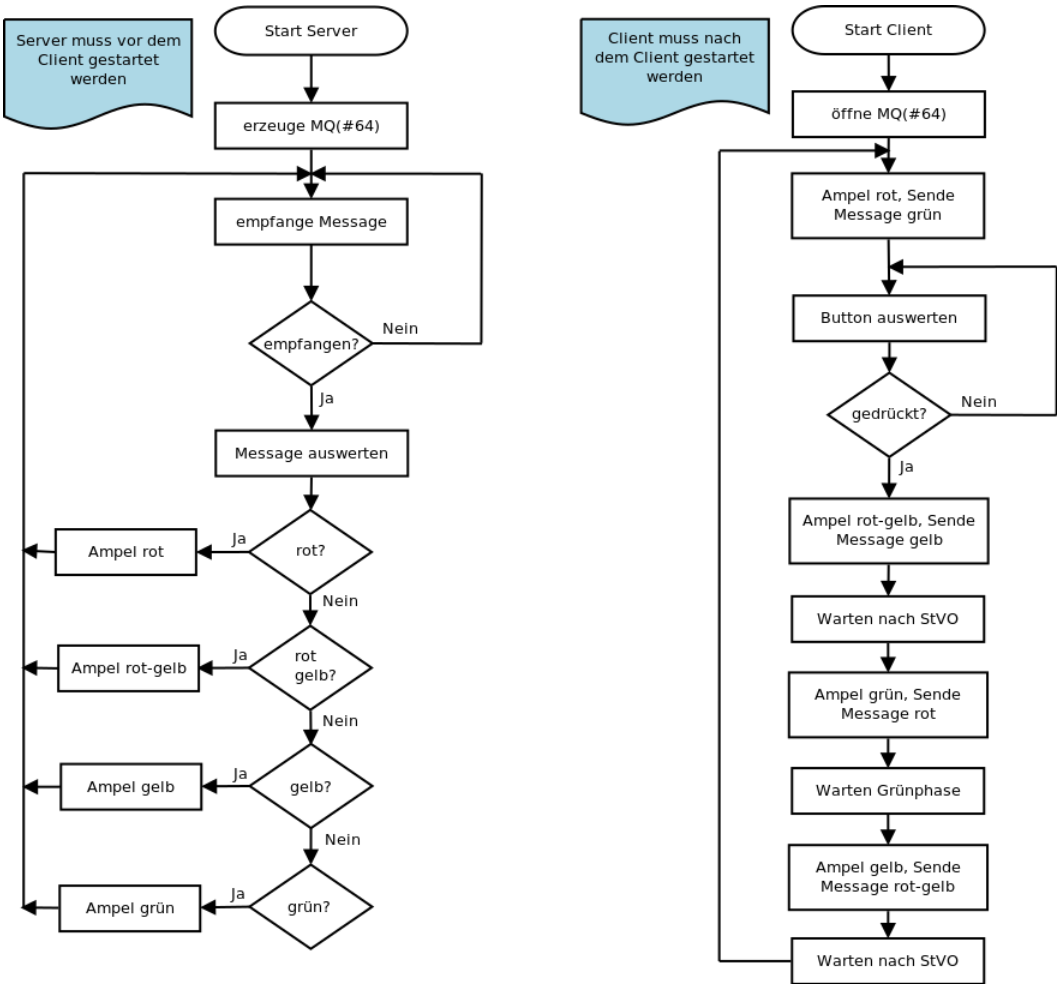

Abbildung 13: PAP Client-Server der Ampelanlage

5.1.1 Ampel-Server

Der Ampel-Server muss als erstes gestartet werden. Zuerst wird eine Message Queue erstellt, die dauernd abgefragt wird, ob eine Message vom Ampel-Client empfangen wurde. Wenn ja, wird die Message ausgewertet und interpretiert, um die Fahrzeugampel entsprechend der Ampelphasen zu schalten. Der Ampel-Server ist nur in der Lage, Messages vom Ampel-Client zu empfangen.

5.1.2 Ampel-Client

Der Ampel-Client muss nach dem Ampel-Server gestartet. Die vom Server bereits erstellte Message Queue wird geöffnet. Gemäß einer Startbedingung werden die Ampelphasen auf Server- und Clientseite eingestellt. Ein Start-Button sorgt dafür, dass der Ampelphasenzyklus auf Server- und Clientseite entsprechend der StVO geschaltet wird. Der Ampel-Client ist für die eigene und auch serverseitige Ampelphase verantwortlich.

5.2 Softwareschnittstelle

Um das im PAP erstellte Softwaremodell in die Realität umzusetzen, wird auf die Softwareschnittstelle der Message Queues zurückgegriffen. Es sind drei Betriebssystemfunktionen vorhanden, um mit einer Message Queue zu arbeiten.

5.2.1 Funktion msgget()

Um eine Message Queue zu erzeugen oder zu öffnen, muss man die Funktion msgget() verwenden. Der Rückgabewert der Funktion ist die ID der Message Queue, welche man zum Senden und Empfangen von Nachrichten benötigen.

Beispiel Server: msgid=msgget(key,0644|IPC_CREAT)

Beispiel Client: msgid=msgget(key,0644)

Der Parameter key ist eine Nummer, die benötigt wird, damit die erstellte Message Queue eindeutig im System identifiziert werden kann. Somit müssen Sender und Empfänger die gleiche Nummer benutzen. Diese kann willkürlich festgelegt werden.

Ein weiterer wichtiger Punkt ist es die Zugriffsrechte des Message Queues zu setzen. Übliche Varianten sind 0644 (rw-r—r--) und 0666 (rw-rw-rw-).

Mit dem Flag IPC_CREAT wird die ID generiert.

5.2.2 Funktion msgsnd()

Zum Versenden von Nachrichten an eine Message Queue wird die Funktion msgsnd() verwendet. Voraussetzung hierfür ist eine Struktur die den Typen der Nachricht und den Text der Nachricht enthält.

Beispiel Server: struct msgbuf

```
{long mtype;
 char msgtxt[200];
}
```

Die Variable mtype ist von Bedeutung, wenn man den Nachrichten verschiedene Prioritäten zuweisen möchte. In der Variablen msgtxt befindet sich der Text der Nachricht.

Beispiel Server:
```
struct msgbuf msg
msgsnd(msgid,&msg,sizeof(msg),0)
```

Beim Versenden von Nachrichten benötigt man die ID der MessageQueue, die Adresse der Nachricht, die Größe der Nachricht und die Festlegung der Zugriffsrechte (in diesem Beispiel werden keine Zugriffsrechte zugewiesen; Flag = 0).

5.2.3 Funktion msgrcv()

Zum Empfangen von Nachrichten wird die Funktion msgrcv() verwendet. Hierbei gelten dieselben Vorrausetzungen wie beim msgsnd().

Beispiel Client:
```
struct msgbuf msg
msgrcv(msgid,&msg,sizeof(msg),1,0)
```

6 Fazit und Schlussfolgerung

Für die Kommunikation zwischen zwei Prozessen ist die Interprozesskommunikation notwendig. Eine Message Queue ist dafür eine Technik, diese Kommunikation zu ermöglichen. Ein Vorteil ist, dass die Kommunikation dabei zeitunabhängig erfolgen kann. Das heißt, die Message bleibt so lange in der Warteschlange, bis sie vom Empfänger auch verarbeitet wurde. Somit geht keine Nachricht verloren, selbst wenn der Empfänger abgeschaltet wird.

Ein weiterer Vorteil, ist die Vergabe von Prioritäten, mit denen man Einfluss auf die Reihenfolge des Empfangens nehmen kann. In diesem Labor wird davon aber kein Gebrauch gemacht.

Die Ampelanlage mit den Message Queues als IPC ist gemäß dem Pflichtenheft und den Erwartungshaltungen geplant, implementiert und lauffähig.

7 In eigener Sache

Bei der praktischen Umsetzung des Projektes ist folgendes Problem aufgetreten: Obwohl der Server und der Client (nahezu) gleichzeitig die Ausgabe der Ampelphase vornehmen sollten, hat der Server die Ampelphase direkt ausgegeben, der Client tat dies jedoch erst zeitversetzt. In der Praxis kann dies zu kritischen Situationen führen. Durch Nachforschungen stellte sich heraus, dass die Zeitverzögerung auf den Ausgabepuffer zurückzuführen ist. Erst wenn genug Text im Ausgabepuffer steht, wird dieser ausgegeben. Das Warten auf genügend Text, ergab die Zeitverzögerung. Lösung für dieses Problem ist die Anweisung `fflush(stdout);`. Diese veranlasst die sofortige Ausgabe auf den Monitor.

8 Abbildungsverzeichnis

9 Literaturverzeichnis

Koslowski, Denis: Interprozesskommunikation IPC, Friedrich-Alexander Universität Erlangen-Nürnberg, https://www4.cs.fau.de/Lehre/SS05/PS_KVBK/talks/Koslowski_Handout.pdf, 04.07.2005

Groß, Prof. Dr. S.: Parallelverarbeitung, Hochschule Fulda, https://www2005.hs-fulda.de/index.php?id=4454&L=1&F=2, 2005

Spinczyk, Olaf: Betriebssysteme, Rechnernetze und verteilte Systeme 1 (BSRvS1) – Interprozesskommunikation, Technische Universität Dortmund, https://ess.cs.uni-dortmund.de/Teaching/SS2008/BSRvS1/Downloads/07-Interprozesskommunikation-1x2.pdf, 2007

Wolf, Jürgen: Linux-UNIX-Programmierung; 2. Aktualisierte und erweiterte Auflage, Bonn 2006

10 Anhang

10.1 Sourcecode

10.1.1 Sender

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


typedef struct
{
  //Indentität der Message(Typ), zwischen welchen Prozessen?
  long mtype;
  //Messagelänge
  char msgtxt[200];
} MQ;
int main(void)
{
  MQ msg;
  int msgid;

  //msgid wir generiert
  //mit der msgid kann auf die Message in der Queue zugegriffen werden
  //96 ist der key der Message
  (msgid=msgget(96,0644|IPC_CREAT));

  printf("\n Starte Ampel...\n");

  //Typ = 1 dient zur "Absprache/Kommunikation" zwischen den Prozessen
  msg.mtype=1;
  strcpy(msg.msgtxt, "ROT");
  while(1)
{
    strcpy(msg.msgtxt, "\e[1;32m⊗\e[0m\t\t\e[1;31m⊗\e[0m");
    //send(ID,Msg-Adress, Größe, flag)   Flag=Zugriffsrechte (0-> keine
flags
```

```
    (msgsnd(msgid,&msg,sizeof(msg),0));
    printf("\n %s",msg.msgtxt);
    fflush(stdout);

    sleep(10);

    strcpy(msg.msgtxt, "\e[1;33m⊗\e[0m\t\t\e[1;31m⊗\e[0m");
    (msgsnd(msgid,&msg,sizeof(msg),0));
    printf("\n %s",msg.msgtxt);
    fflush(stdout);
    sleep(1);

    strcpy(msg.msgtxt, "\e[1;31m⊗\e[0m\t\t\e[1;32m⊗\e[0m");
    (msgsnd(msgid,&msg,sizeof(msg),0));
    printf("\n %s",msg.msgtxt);
    fflush(stdout);
    sleep(5);

    strcpy(msg.msgtxt, "\e[1;31m⊗\e[0m\e[1;33m⊗\e[0m\t\t\e[1;31m⊗\e[0m");
    (msgsnd(msgid,&msg,sizeof(msg),0));
    printf("\n %s",msg.msgtxt);
    fflush(stdout);
    sleep(1);
  }

  return 0;
}
```

10.1.2 Empfänger

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

struct msgbuf
{
```

```
  long mtype;
  char msgtxt[200];
};
int main(void)
{
  struct msgbuf msg;
  int msgid;

  //96 ist der key der Message
  msgid=msgget(96,0644);

  //Endlosschleife
  while(1)
  {
    //receive(ID,Adresse,Größe, Typ(1),flag (0=kein flag[Zugriffsrechte])
    msgrcv(msgid,&msg,sizeof(msg),1,0);

    printf("%s\n",msg.msgtxt);
  }
  return 0;

}
```