Frankfurt University of applied Sciences
Fachbereich 02 Informatik und Ingenieurwissenschaften
Veranstaltung: Betriebssysteme und Rechnernetze
Sommersemester 2022

Interprozesskommunikation

Prüfer: Prof. Dr. Christian Baun

Vorgelegt von:

Ariana Rashid Matrikelnr.: 1406292

Donika Osmani Matrikelnr.: 1406331

Ela Küpelikilinc Matrikelnr.: 1179862

Mehmet Kahveci Matrikelnr.: 1400311

Sinesaan Sivakumar Matrikelnr.: 1250367

Inhaltsverzeichnis

Abstract

Die Hauptsächliche Funktion der Interprozesskommunikation ist der Informationsaustausch zwischen mehreren Prozessen eines Systems. Es gibt mehrere Techniken von Prozesskommunikationen. Einige davon sind Pipes, Messages Queue, Sockets und Shared Memorys, auf welche in dieser Dokumentation eingegangen werden.

Ziel dieser Dokumentationsausarbeitung ist die Erläuterung eines Simulators für eine Interprozesskommunikation. Zuerst wird auf die Problemstellung eingegangen und die Funktionsweise näher erläutert. Anschließend wird das Konzept als auch die Implementierung und das Programm erklärt. Abschließend folgt das Fazit.

1. Einleitung

Als Gruppe haben wir uns für das Thema „Interprozesskommunikation“ in der Programmiersprache C entschieden, da das Thema, wie die einzelnen Prozesse untereinander kommunizieren und interagieren, ein grundlegendes Allgemeinwissen für angehende Wirtschaftsinformatiker sein sollte. Ebenso ist C die verbreitetste betriebssystemnahe Programmiersprache. Ziel des Projektes ist es ein Echtzeitsystem zu entwickeln, das aus vier Prozessen besteht. Damit diese vier Prozesse miteinander kommunizieren können benötigen wir ein Mechanismus, die sogenannte Interprozesskommunikation. Bei diesem Datenaustausch betrachten wir vier Möglichkeiten: Pipes, Massage Queues, Shared Memory mit Semaphoren und mit Sockets.

2. Aufgabenstellung

Die Aufgabe besteht darin, eine Interprozesskommunikation bzw. ein Echtzeitsystem zu entwickeln und zu implementieren, dass aus vier Prozessen besteht. Der erste Prozess ***Conv.*** liest Messwerte von A/D (Analog/Digital) Konvertern ein. Der zweite Prozess ***Log.*** liest die Messwerte von ***Conv.*** aus und schreibt diese in eine lokale Datei. Der dritte Prozess ***Stat.*** ist dafür zuständig die Messwerte von ***Conv.*** auszulesen und berechnet statistische Daten (Mittelwert und Summe). Der letzte und vierte Prozess ***Report.*** greift auf die vorherigen Ergebnisse von ***Stat.*** zu und gibt die Daten in eine Shell aus.

Der Datenaustausch zwischen den jeweiligen vier Prozessen soll mit den folgenden Prozesskommunikationen: Pipes, Message Queues, Sockets und Shared Memory mit Semaphoren realisiert werden. Als Ergebnis müssen vier Implementierungsvarianten des Prozessprogramms existieren.

3. Problemstellung

Es konnte festgestellt werden, dass die Befehle, die für die Umsetzung des Projektes notwendig sind, nur mit einem Linux Betriebssystem möglich sind. Diese können nicht auf einem Windows Betriebssystem ausgeführt werden.

4. Umsetzung

Zunächst wurde mit dem vorgegeben Wissen durch die Aufgabenstellung ein Konzept entwickelt, welches die Grundbasis für die Implementierung der einzelnen Prozesskommunikationsmöglichkeiten dargestellt hat.

5. Konzept

Prozesse haben einen bestimmten Zustand und Ressourcen, die vom Betriebssystem verwaltet werden. Grundsätzlich werden auf dem Rechner Prozesse gleichzeitig ausgeführt, bzw. erweckt dies den Eindruck. Aus funktionaler Sicht ermöglicht die Prozessinteraktion die Kommunikation und die Kooperation zwischen den Prozessen. Dabei wird darauf eingegangen, wie einzelne Prozesse miteinander kommunizieren. Hierbei werden wir auf die folgende Kommunikationsmöglichkeiten eingehen und sie implementieren.

Die erste Möglichkeit ist der Prozess mit Pipes (Kommunikationskanäle), hierbei wird zwischen zwei Arten von Pipes unterschieden, einmal die anonymen und die benannten Pipes. Des Weiteren wird im Abschnitt „*Prozess mit Pipe*“ die Kommunikationsart tiefer erläutert.

Die zweite Kommunikationsmöglichkeit nennt sich Message Queues (Nachrichtenwarteschlange), sind verkettete Listen, in die Prozesse Nachrichten ablegen und aus denen sie Nachrichten abholen können.

Ein weiteres Konzept ist Sockets, ein Benutzerprozess kann einen Socket vom Betriebssystem anfordern, und über diesen Daten versenden und empfangen.

Das letzte Konzept nennt sich Shared Memory mit Semaphoren (Gemeinsamer Speicher), die dabei verwendeten gemeinsamen Speichersegmente sind Speicherbereiche, auf die mehrere Prozesse direkt zugreifen können.

7. Implementierung

Im folgenden Abschnitt werden die Programmcodes der vier Prozesskommunikationsmöglichkeiten mit Codeabschnitten verdeutlicht und erläutert. Das Programm wurde als Shell Skript in der Programmiersprache C geschrieben.

7.1 Prozess mit Pipe

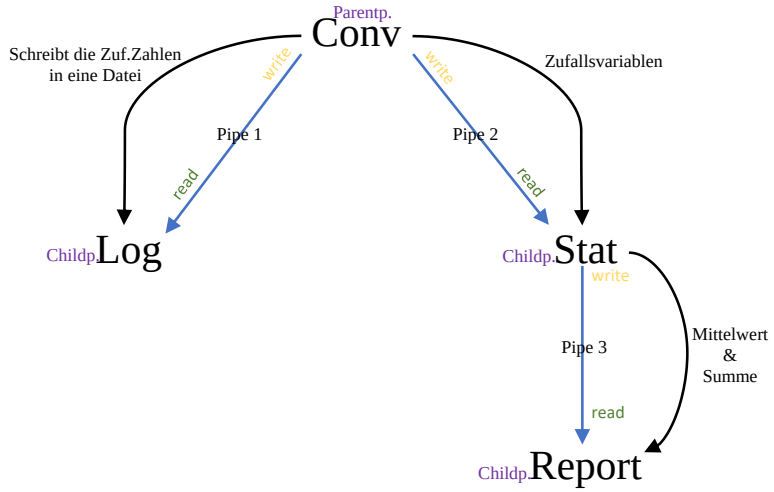

Abbildung 1: Struktur mittels Pipes

In der Interprozesskommunikation mittels Pipes, werden im Elternprozess ***Conv***. zehn zufällige Zahlen generiert, welche anschließend in die zwei Pipes geschrieben werden. Die Pipes bilden jeweils eine Verbindung zwischen dem *Conv- zu Log-* und *Conv- zu Stat-* Prozessen. In dem Prozess ***Log.*** werden die zufälligen Zahlen herausgelesen und anschließend in einer externen lokalen Datei gespeichert. Der darauffolgende Prozess ***Stat***. liest die zufälligen Zahlen ebenfalls heraus. Mithilfe des zufälligen Wertes wird der Mittelwert und die Summe berechnet. Daraufhin werden die Werte in eine weitere Pipe *(Pipe 3)* geschrieben. Im nächsten Prozess werden die Werte von Pipe 3 gelesen und in der Shell ausgegeben. Dieser Ablauf wird in der **Abbildung 1** dargestellt.

```
// Erstellung von 3 Pipes
int pipe1[2];
int pipe2[2];
int pipe3[2];

//-----------------------------------------------Die 1. Pipe -----
if (pipe(pipe1) < 0)
{
  printf("Das Anlegen der Pipe 1 ist fehlgeschlagen. \n");
  exit(1); // Programmabbruch
}
else
{
  // printf("Die Pipe 1 wurde angelegt.\n");
}
```

Abbildung 2: Erstellung von Pipes

Abbildung 2 zeigt einen Code-abschnitt wie eine Pipe erzeugt wird. Anhand einer If – Anweisung wird das Anlegen einer Pipe geprüft. Dieser Vorgang wurde insgesamt drei Mal durchgeführt, somit wurden drei verschiedene Pipes erzeugt.

```
47  //-----------------------------Ein Elternprozess erzeugen von Conv---------
48  int conv = fork();
49  if (conv > 0)
50  {
51    int n = 10;
52    int i;
53    int arr[10];
54    srand(time(NULL));
55
56    // printf("Zufallsvariablen generiert: ");
57    for (i = 0; i < n; i++)
58    {
59      arr[i] = rand() % 51;
60      // printf(" %d", arr[i]);  // Zur Kontrolle Kommentar Zeichen wegmachen
61    }
62
63    // Die Zufallsvariabeln in die Pipe schreiben
64    write(pipe1[1], &n, sizeof(int));
65    write(pipe3[1], &n, sizeof(int));
66
67    // printf("gesendete Anzahl = %d\n", n);
68    write(pipe1[1], arr, sizeof(int) * n);
69    write(pipe3[1], arr, sizeof(int) * n);
70    // printf("Array in die Pipe gesendet\n");
71    close(pipe1[0]);
72    exit(0);
73  }
```

In **Abbildung 3** wird ein Elternprozess ***conv.*** mit dem Befehl `fork()` erstellt. Da werden die zufälligen Zahlen (1-50) mit dem Befehl `rand()` generiert. Wenn der Befehl `srand(time (NULL))` ausgeführt wird, werden neue Zahlen generiert, diese werden in die Pipe 1 und 3 mit dem Befehl `write()` geschrieben, anschließend werden die Pipes mit dem Befehl `close()` geschlossen.

Abbildung 3: Elternprozess Conv.

```
75  //-----------------------Kindprozess von stat----
76  close(pipe1[1]);
77  int stat = fork();
78  if (stat == 0)
79  {
80
81    close(pipe1[1]);
82
83    int arr[10];
84    int i, n;
85    int sum = 0;
86    int mittelwert = 0;
87
88    read(pipe1[0], &n, sizeof(int));
89    read(pipe1[0], arr, sizeof(int) * 10);
90
91    for (i = 0; i < n; i++)
92    {
93      sum += arr[i];
94      mittelwert = sum / n;
95    }
96    // printf("Das Summe ist: %d\n", sum);
97    // printf("Mittelwert ist: %d\n", mittelwert);
98    close(pipe1[0]);
99
100   // mittelwert und Summe in die Pipe schreiben
101   write(pipe2[1], &sum, sizeof(int));
102   write(pipe2[1], &mittelwert, sizeof(int));
103   close(pipe2[1]);
104   exit(0);
105 }
```

In **Abbildung 4** wird als erstes ein Kindprozess ***stat.*** erstellt. In diesem werden die Zufälligen Zahlen aus der Pipe 1 mittels dem Befehl `read()` gelesen. Mit Hilfe einer For-Schleife werden Mittelwert und Summe ausgerechnet. Danach werden die Werte in die Pipe 2 geschrieben und anschließend wieder geschlossen.

Abbildung 4: Kindprozess Stat.

Im Kindprozess werden die Werte aus der Pipe 2 gelesen und mittels eines `printf()` Befehl in der Shell ausgegeben.

Mögliche Ausgabe:

```
Die empfangene Summe: 190
Der empfangene Mittelwert: 19
```

```
107 //-----------------------------Report Childprozess------
108 close(pipe1[1]);
109 int sum;
110 int mittelwert;
111 int report = fork();
112 if (report == 0)
113 {
114   read(pipe2[0], &sum, sizeof(int));
115   read(pipe2[0], &mittelwert, sizeof(int));
116   printf("Die empfangene Summe: %d\n", sum);
117   printf("Der empfangene Mittelwert: %d\n", mittelwert);
118   printf("\n");
119
120   close(pipe2[0]);
121 }
122 //<-------------------------------------------------
```

Abbildung 4: Kindprozess Report

Im ***Log***-Kindprozess werden als erstes die Zahlen gelesen, die im ***Conv***-Elternprozess (**Abbildung 3**) in die Pipe 3 geschrieben wurden. Der folgende Codeabschnitt in **Abbildung 5** zeigt das Schreiben der Zahlen in eine lokale Datei. Dazu wird als erstes der Befehl `fopen()` aufgerufen. Dieser Befehl trägt dazu bei, dass eine Datei namens „Zufällige_Zahlen_Pipes.txt“ geöffnet wird. Als nächstes werden die Zahlen mittels dem Befehl `fprintf()` in die Datei geschrieben.

```
123  //--------------------------------Childprozess von log ---------------------------------------->
124
125  int log = fork();
126  if (log == 0)
127  {
128    int arr[10];
129    int i, n;
130
131    read(pipe3[0], &n, sizeof(int));
132    read(pipe3[0], &arr, sizeof(int) * 10);
133
134    // printf("Die Zahlen in der Datei: %d\n", arr[i]);
135    // printf("\n");
136    close(pipe1[0]);
137
138    //<----------------------------------------------------------------------------------------
139    //-------------------------Die Zahlen in eine Datei schreiben------------------------------>
140    FILE *fp;
141    fp = fopen("Zufällige_Zahlen_Pipes.txt", "w"); // Datei namens Zufällige_Zahlen_Pipes wird geöffnet
142    if (fp == NULL)
143    {
144      printf("Die Datei konnte nicht gefunden werden. \n");
145    }
146    else
147    {
148      // array Liste schreiben
149      for (i = 0; i < n; i++)
150      {
151        fprintf(fp, " %d: %d\n", i + 1, arr[i]);
152      }
153    }
154    fclose(fp);
155  }
```

Abbildung 6: Struktur mittels Pipes

7.2 Prozess mit Message Queues

```
19   // Erstellung von 3 MQ
20   //-----------------------------------------------Die 1.MQ ----------------------------
21   int mq_key = 12341;            // Message Queue Key
22   int rc_msgget;                 // Rückgabewert (return code) von msgget
23   int rc_msgctl;                 // Rückgabewert (return code) von msgctl
24   int rc_msgrcv;                 // Rückgabewert (return code) von msgrcv
25   msg sendbuffer, receivebuffer; // Einen Empfangspuffer und einen Sendepuffer anlegen
26
27   rc_msgget = msgget(mq_key, IPC_CREAT | 0600);
28   if (rc_msgget < 0)
29   {
30     printf("Die Warteschlange konnte nicht erstellt werden.\n");
31     perror("msgget");
32     exit(1);
33   }
34   else
35   {
36     printf("Nachrichtenwarteschlange %i mit ID %i ist bereit.\n",
37                mq_key, rc_msgget);
38   }
39
40   sendbuffer.mtype = 1; // Nachrichtentyp festlegen
```

Abbildung 7: Die Erstellung der ersten Message Queue

In der Code-Zeile 21 wird ein Schlüssel, vom Typ `int`, festgelegt. Mit der Funktion `msgget()` wird eine Message Queue erzeugt. Anhand einer If – Anweisung wird überprüft, ob die Message Queue erstellt wurde.

```
171     // Mittelwert Übergabe
172
173       sendbuffer.mtext[11] = mittelwert;
174     //  printf("Mittelwert ist  ------- %i\n", sendbuffer.mtext[11]);
175       sendbuffer.mtext[14] =sum;
176     //    printf("Summe ist  -------++++++++++++ %d\n", sendbuffer.mtext[14]);
177
178
179     // Mittelwert übergeben----------------------------------------------------
180         if (msgsnd(rc_msgget3,
181                    &sendbuffer,
182                    sizeof(sendbuffer.mtext),
183                    IPC_NOWAIT) == -1)
184       {
185         printf("Das Senden des Mittelwert ist fehlgeschlagen.\n");
186         perror("msgsnd");
187         exit(1);
188       }
```

Abbildung 8: Mittelwerte Übergabe

Die Funktion `msgsnd()` übergibt den Sendepuffer an die Message Queue. Mit der If-Anweisung wird überprüft, ob dieser Vorgang erfolgreich war.

```
280   // Mittelwert herauslesen-------------------------------------------------
281       rc_msgrcv3 = msgrcv(rc_msgget3,
282                           &receivebuffer,
283                           sizeof(receivebuffer.mtext),
284                           receivebuffer.mtype,
285                           MSG_NOERROR | IPC_NOWAIT);
286       if (rc_msgrcv < 0)
287       {
288         printf("Lesen der Nachricht fehlgeschlagen.\n");
289         perror("msgrcv");
290         exit(1);
291       }
292       printf("Empfangene Mittelwert ist: %i\n", receivebuffer.mtext[11]);
293       printf("Empfangene Summe ist %d\n\n", receivebuffer.mtext[14]);
```

Im Quellcode (**Abbildung 9**) werden die Summe und Mittelwert mit Hilfe des Befehls `msgrcv()` gelesen. Danach werden diese Werte mit einem `printf()` Befehl ausgegeben.

Abbildung 9: Mittelwert und Summe lesen

```
203       // Nachrichtenwarteschlange löschen
204       rc_msgctl = msgctl(rc_msgget, IPC_RMID, 0);
205       if (rc_msgctl < 0)
206       {
207         printf("Die Warteschlange konnte nicht gelöscht werden.\n");
208         perror("msgctl");
209
210         exit(1);
211       }
212       else
213       {
214         printf("%i mit ID %i wurde gelöscht.\n\n", mq_key, rc_msgget);
215       }
216
217       exit(1);
218     }
```

Abbildung 10 zeigt den Quellcode für die Löschung der Message Queue. Diese erfolgt über den Befehl `msgctl()` und mit dem Kommando IPC_RMID.

Abbildung 10: Nachrichtenwarteschlange löschen

Das IPC_RMID Kommando löscht die Message Queue ID und die in der Message Queue enthaltenen Daten. Mit der If-Anweisung wird überprüft, ob der Löschvorgang erfolgreich war.

7.3 Prozess mit Sockets

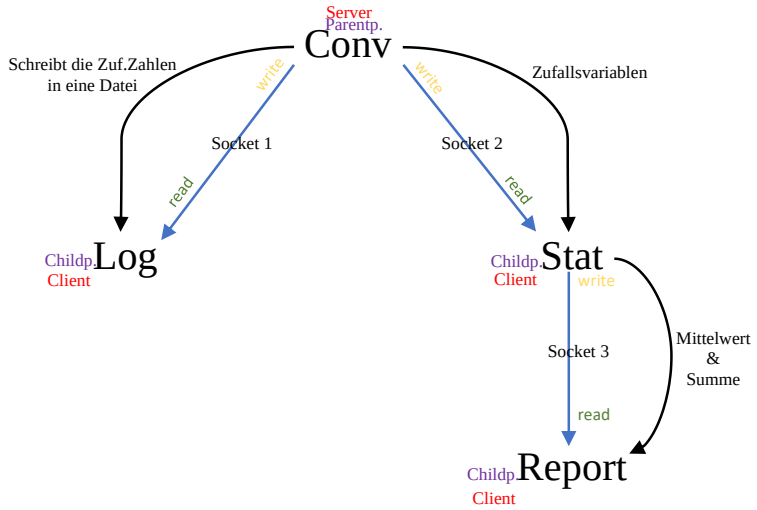

Abbildung 11: Struktur von Socket

In der Interprozesskommunikation mittels Sockets, werden im Elternprozess ***Conv***. ebenfalls zehn zufällige Zahlen generiert, welche anschließend jeweils in die zwei Sockets geschrieben werden. Die Sockets befinden sich in den Clients. Die Sockets bilden jeweils eine Verbindung zwischen dem *Conv- zu Log-*, *Conv- zu Stat-* und *Stat- zu Report* Prozessen. Die Verbindung zwischen dem Server und den Clients ist über die Portnummer gegeben. Der darauffolgende Prozess ***Stat***. liest die zufälligen Zahlen ebenfalls heraus. Mithilfe des zufälligen Wertes wird der Mittelwert und die Summe berechnet. Daraufhin werden die Werte an den nächsten Client übergeben, der wiederum die Werte ausgibt.

```
13  int sd;
14  int neuer_socket;
15  int portnummer;
16  int clientadresselength;
17  struct sockaddr_in adresse;
18
19  // Inhalt des Puffers mit Null-Bytes fÃ¼llen
20  char puffer[1024];
21
22  // die Portnummer
23  portnummer = 1111;
24
25  // Speicherbereich der Struktur sockaddr_in mit Nullen fÃ¼llen
26  memset(&adresse, 0, sizeof(adresse));
27
28  // Socket-Adresse in der Struktur sockaddr_in speichern
29  adresse.sin_family = AF_INET;
30  adresse.sin_addr.s_addr = INADDR_ANY;
31  adresse.sin_port = htons(portnummer);
```

Abbildung 12: Portnummer definieren

In **Abbildung 12** wird die Portnummer definiert und die Sock-Adresse wird in der Struktur gespeichert.

```c
// Neuen Socket erstellen
int sc;
sc = socket(AF_INET, SOCK_STREAM, 0);
if (sc < 0)
{
  printf("Der Socket konnte nicht erzeugt werden.\n");
  exit(1);
}
else
{
  printf("Der Socket wurde erzeugt.\n");
}
if (bind(sc,
         (struct sockaddr *)&adressen,
         sizeof(adressen)) < 0)
{
  printf("Der Port ist nicht verfügbar.\n");
  exit(1);
}
else
{
  printf("Der Socket wurde an den Port gebunden.\n");
}
```

Abbildung 13: Erstellung von Sockets

In **Abbildung 13** ist der Quellcode für die Erstellung eines Sockets zu sehen. Hierzu wird als erstes der Befehl `socket()` aufgerufen. Mit einer If-Anweisung, erfolgt eine Prüfung, ob dieser Socket erstellt wurde. Mit dem Befehl `bind()` wurde der Socket an eine Port gebunden. Dieser Vorgang wird ebenfalls mit einer If-Anweisung geprüft.

```c
// Eine Warteschlange für bis zu 5  Verbindungsanforderungen einrichten
if (listen(sc, 5) == 0)
{
  printf("Warte auf Verbindungsanforderungen.\n");
}
else
{
  printf("Es kam beim listen zu einem Fehler.\n");
  exit(1);
}
clientadresselength2 = sizeof(adressen);
neuer_socket2 = accept(sc,
                       (struct sockaddr *)&adressen,
                       &clientadresselength2);
if (neuer_socket < 0)
{
  printf("Verbindungsanforderung fehlgeschlagen.\n");
  exit(1);
}
else
{
  printf("Verbindung zu einem Client aufgebaut.\n");
}
if (write(neuer_socket2, puffer, sizeof(puffer)) < 0)
{
  printf("Der Schreibzugriff ist fehlgeschlagen.\n");
}
```

Abbildung 14: Quellcode für listen() und accept()

Als nächstes wird eine Warteschlange für bis zu fünf Verbindungsanforderungen eingerichtet. Dies erfolgt mit dem Befehl `listen()`. Im ersten Parameter wird angegeben, welches Socket „belauscht“ werden soll, als zweite Parameter wird die maximale Anzahl von Verbindungen angegeben, die gleichzeitig entgegengenommen werden können. Als nächstes wird der Befehl `accept()` ausgeführt. Im ersten Parameter wird der Socket Name geschrieben. Der zweite Parameter enthält die Adresse und der dritte Paramater enthält die Länge des zweiten Parameters.

Abbildung 15 zeigt den Quellcode für die Schließung eines Sockets. Hierzu wird der `close()` Befehl aufgerufen und als Parameter den Socket Name übergeben.

```c
// Socket schließen
if (close(sc) == 0)
{
  printf("Der Socket wurde geschlossen.\n\n");
}
```

Abbildung 15: Schließung des Sockets

7.4 Prozess Shared Memory mit Semaphoren

```c
int key;
int shmid;
int *shm;

if ((key = ftok("/tmp", 'y')) == -1)// aus anme in ein Schlüssel bei -1 Fehlermeldung
{
  perror("Fehler!!!\n");
  exit(EXIT_FAILURE);
}

shmid = shmget(key, SIZE * sizeof(int), IPC_CREAT | IPC_EXCL | 0600);
if (shmid == -1)
{
  perror("Das Segment konnte nicht erstellt werden.\n");
  exit(EXIT_FAILURE);
}

shm = (int *)shmat(shmid, NULL, 0);
if (shm == (int *)-1)
{
  perror("Das Segment konnte nicht angehängt werden. \n");
  exit(EXIT_FAILURE);
}
```

Abbildung 16: Segment erstellen

Das Programm erzeugt einen gemeinsamen Speichersegment mit der Funktion `shmget()`. Der Parameter *0600* definiert die Zugriffsrechte. Der Systemaufruf `shmat()` sorgt dafür, dass ein Segment an einen Prozess anhängt. Wenn der Rückgabewert der Funktion „–1“ ist, kann der Betriebssystemkern das Segment nicht anlegen.

```c
int key;
int sum = 0;
int shmid;
int *shm;
int n = 10;

key = ftok("/tmp", 'y');

shmid = shmget(key, 0, 0600);
if (shmid == -1)
{
  perror("Das Segment konnte nicht erstellt werden.\n ");
  exit(EXIT_FAILURE);
}
const char sem1_name[] = "Semaphor1";
const char sem2_name[] = "Semaphor2";
int returncode_close, returncode_unlink;
int output;

sem_t *sem1, *sem2;

// Das Puffern Standardausgabe (stdout) unterbinden
setbuf(stdout, NULL);

// Neue benannte Semaphore /mysem1 erstellen die den initialen Wert 1 hat
sem1 = sem_open(sem1_name, O_CREAT, 0600, 1);
if (sem1 == SEM_FAILED)
{
  printf("Die Semaphore konnte nicht erstellt werden.\n");
  perror("sem_open");
  exit(1); // Programmabbruch
}
```

Abbildung 17: Die Erstellung von Semaphoren

Wenn der gemeinsame Speicher (*Shared Memory*) von zwei oder mehreren Prozessen gleichzeitig genutzt wird, muss verhindert werden, dass sie gleichzeitig schreiben oder dass ein Prozess liest, während ein anderer Prozess schreibt. Die erstellten Semaphoren *“Sem1, Sem2”* dienen als Schutz gegen gleichzeitiges Passieren in einem kritischen Bereich. Ist der Wert 0, wird der Prozess blockiert. Ist der Wert > 0, wird er um den Wert 1 erniedrigt.

```c
// Semaphore sem2 schliessen
returncode_close = sem_close(sem2);
if (returncode_close < 0)
{
  printf("%s konnte nicht geschlossen werden.\n", sem2_name);
  exit(1); // Programmabbruch
}
else
{
  printf("%s wurde geschlossen.\n", sem2_name);
}

// Semaphore /mysem1 entfernen
returncode_unlink = sem_unlink(sem1_name);
if (returncode_unlink < 0)
{
  printf("%s konnte nicht entfernt werden.\n", sem1_name);
  exit(1); // Programmabbruch
}
```

In der **Abbildung 18** wird der *"Sem2"* durch den Systemaufruf `sem_close(sem2)` geschlossen, da der Teilprozess geschlossen wurde. Daraufhin wird das Semaphor entfernt, da es nicht mehr benötigt wird.

Abbildung 18 Die Schließung des Semaphors

```c
if (returncode_shmdt < 0)
{
  printf("Das Segment konnte nicht gelöst werden.\n");
  exit(1);
}else{
  printf("Das Segment wurde vom Prozess gelöst.\n");
}
// // Gemeinsames Speichersegment löschen
int returncode_shmctl;
returncode_shmctl = shmctl(shmid, IPC_RMID, 0);
if (returncode_shmctl == -1)
{
  printf("Das Segment konnte nicht gelöscht werden.\n");
  perror("shmctl");
  exit(1);
} else{
  printf("Das Segment wurde gelöscht.\n");
}
exit(0);
```

Abbildung 19: Semaphore entfernen und löschen

Ist ein gemeinsames Speichersegment an keinen Prozess mehr gebunden, wird es nicht automatisch vom Betriebssystem gelöscht, sondern bleibt erhalten, bis die Löschung durch den Systemaufruf `shmctl()` angewiesen wird oder bis zum Neustart des Betriebssystems.

8. Fazit

Bei der Implementierung des Konzeptes der Interprozesskommunikation mit der Programmiersprache C haben sich anfangs viele Schwierigkeiten aufgewiesen. Zudem hat es besonders viel Zeit in Anspruch genommen, das Projekt in der Programmiersprache C umzusetzen, da unter anderem ursprünglich die Umsetzung mit Python geplant war. Allerdings haben wir bemerkt, dass die Umsetzung mit Python anspruchsvoller ist. Die Implementierung des Echtzeitssystems, welches aus vier Prozessen besteht, haben wir zunächst mit einer groben Struktur bildlich dargestellt, welches uns das Verständnis und die Umsetzung besser zum Verständnis gebracht hat. Die Einführung in das Thema hat uns die Literatur „Betriebssysteme Kompakt" von Prof. Dr. Christian Baun vereinfacht und diente praktisch als Leitfaden. Abschließend lässt sich sagen, dass wir durch das Projekt die Kommunikation zwischen den Prozessen näher verstanden haben und diese auch eigenständig in Programmiersprachen darstellen bzw. entwickeln können.