FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Betriebssysteme und Rechnernetze

WS 2022/23

Portfolioprüfung – Werkstück A – Alternative 1

Echtzeitsystem

Projektleiter:

Diego Gutiérrez 1387049

Jabrail Shurayev 1343575

Mohammed Matran 1402704

Omar Abd Alwahed 1407819

Dozenten:

Christian Baun

Henry Cocos

Abgabe:

25.06.2023

Dokumentversion

Version	Datum	Autor	GitHub Repository
1.0	25.06.2023	SmartDevGroup	https://github.com/MohammedMatran/BSRN.git

Inhaltsverzeichnis

1.Projektbeschreibung

Dieses Projekt stellt ein fortschrittliches Echtzeitsystem dar in der Programmiersprache C, bestehend aus vier permanent arbeitenden Prozessen, die in einem koordinierten Rhythmus miteinander kommunizieren. Verschiedene Interprozesskommunikationsmechanismen, wie Pipes, Message Queues, gemeinsam genutzter Speicher mit Semaphoren und Sockets, gewährleisten eine nahtlose Interaktion zwischen den Prozessen. Sie erzeugen imitierte Messdaten, dokumentieren diese, führen entsprechende Analysen durch und generieren aussagekräftige Berichte.

Die Prozesse sind streng synchronisiert, sodass gewährleistet ist, dass Daten korrekt erst geschrieben, dann gelesen und letztendlich analysiert werden, bevor sie zur Meldung gebracht werden. Das System setzt Mechanismen ein, um die Prozesse simultan durch den Systemaufruf "fork" zu starten. Ebenso ist es mit einer Funktion zur Behandlung von SIGINT-Signalen ausgestattet, die eine geregelte Beendigung sicherstellt und dafür sorgt, dass alle von den Prozessen genutzten Ressourcen freigegeben werden.

Zur Überwachung des Systemzustandes und zur Nachverfolgung der Ressourcennutzung werden Überwachungstools wie "top", "ps", "pstree" und "ipcs" eingesetzt. Diese Tools ermöglichen eine lückenlose Kontrolle und Optimierung der Systemleistung.

1.1 Prozessen

Der Prozess "Conv" erzeugt Zufallszahlen, um Messwerte eines Analog-Digital-Wandlers (A/D) zu simulieren. Der "Log"-Prozess schreibt diese Messwerte in eine lokale Datei, während der "Stat"-Prozess den Mittelwert und die Summe dieser Werte berechnet. Schließlich ruft der Prozess "Report" die statistischen Daten ab und zeigt sie in der Konsole an.

```c
// Random number generator
int conv()
{
    int random = rand() % 100000 + 1;
    return random;
}

// Open the text file and write the values in it
void writeToLog(int value)
{
    FILE *file = fopen("werte.txt", "a");
    fprintf(file, "%d\n", value);
    fclose(file);
}
```

Abbildung 1: Conv und Log Funktionen

```c
// Calculate sum and average of the generated values
double stat(int value)
{
    x++;
    sum += value;
    double average = (double)sum / x;
    return average;
}

// Prints sum and average in the console
void report(double value)
{
    printf("Sum: %d\n", sum);
    printf("Average: %.2f\n", value);
}
```

Abbildung 2: Stat und Report Funktionen

1.2 Terminate Funktion

Im gegebenen Programm agiert die Funktion 'terminate' als Signalhandler für das SIGINT-Signal. Normalerweise wird das SIGINT-Signal an einen Prozess gesendet, wenn der Benutzer die Tastenkombination Strg+C drückt. Dieses Signal dient dazu, einen Prozess zu unterbrechen. In Unix-ähnlichen Betriebssystemen führt der Empfang dieses Signals standardmäßig zur Beendigung des Prozesses. Im Kontext unseres Programms ist die Funktion

'terminate' so konfiguriert, dass sie aufgerufen wird, wenn der Benutzer Strg+C eingibt, da sie mit dem SIGINT-Signal verknüpft ist.Im Kontext dieses Programms wird bei der Eingabe eines Strg+C-Befehls durch den Benutzer die Funktion terminate aufgerufen, da diese Funktion mit dem SIGINT-Signal verknüpft ist.

Wenn die Funktion 'terminate' aufgerufen wird, passieren zwei Dinge. Erstens gibt die Funktion eine Nachricht auf der Standardausgabe aus, normalerweise dem Terminal. Diese Nachricht informiert den Benutzer, dass er die Tastenkombination Strg+C gedrückt hat und das Programm daher vom Benutzer beendet wird. Zweitens wird der Befehl 'exit(0)' ausgeführt, um den Prozess zu beenden. Die '0' in diesem Befehl signalisiert, dass der Prozess erfolgreich und ohne Fehler beendet wurde.

```c
// Terminate the program when the user clicks Ctrl+C
void terminate()
{
    printf("\nCtrl+C pressed. Program terminated by user.\n");
    exit(0);
}
```

Abbildung 3: Terminate Funktion

```c
signal(SIGINT, terminate); // Set singnal handler for Ctrl+C
```

Abbildung 4: Signal handler code line in Main

Zusammengefasst ermöglicht die Funktion 'terminate' dem Programm, auf die Eingabe von Strg+C durch den Benutzer zu reagieren. Sie gibt dem Programm die Möglichkeit, sich ordnungsgemäß zu beenden und eine Meldung auszugeben, die den Benutzer über die Beendigung informiert.2.

2. Sockets

2.1 Server – Client Funktion

Die Funktion server_client_func() implementiert die Server-Seite einer Client-Server-Kommunikation unter Verwendung von Sockets in einem TCP/IP-Netzwerk. Hier ist die detaillierte Aufschlüsselung der Funktion:

Zunächst definiert sie Server- und Client-Deskriptoren, Strukturen für die Server- und Client-Adressen und die Port-Nummer für die Kommunikation. Außerdem wird eine Variable für die Länge der Client-Adresse eingerichtet.

Die Funktion beginnt mit der Erstellung eines Server-Sockets. Als Nächstes definiert sie die Parameter für die Serveradresse, indem sie die IP-Adressfamilie (AF_INET für IPv4), die Portnummer (mit der Funktion htons in die Netzwerk-Byte-Reihenfolge konvertiert) und die spezifische IP-Adresse (auf INADDR_ANY gesetzt, um den Socket an alle verfügbaren Schnittstellen zu binden) angibt. Die Funktion bindet dann den Server-Socket an die gerade angegebene Adresse. Nachdem der Socket erfolgreich gebunden wurde, beginnt der Server mit einer Warteschlangengröße von 5 auf Client-Verbindungen zu warten. Der Server nimmt dann eine Client-Verbindung an. Die Funktion accept() blockiert, bis ein Client eine Verbindung

zum Server herstellt. Wenn bei der Annahme der Client-Verbindung ein Fehler auftritt, gibt sie eine Fehlermeldung aus und kehrt zurück.

```c
void server_client_func()
{
    int server_socket, client_socket;
    struct sockaddr_in server_address, client_address;
    int port_number = 12345;
    socklen_t client_address_len;

    // Create server socket and check for error
    server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1)
    {
        printf("Error creating server socket.\n");
        return;
    }
    // Define server address parameters
    server_address.sin_family = AF_INET;
    server_address.sin_port = htons(port_number);
    server_address.sin_addr.s_addr = INADDR_ANY;
```

Abbildung 5: Server Client Funktion

```c
while (1)
{
    // Read from client socket
    int value;
    if (read(client_socket, &value, sizeof(int)) < 0)
    {
        printf("Error reading from client socket.\n");
        break;
    }

    writeToLog(value);            // Write recieved value to log
    double average = stat(value); // Calculate the average value

    // Write the average value back to the client socket
    if (write(client_socket, &average, sizeof(double)) < 0)
    {
        printf("Error writing to client socket.\n");
        break;
    }

    printf("Received average value: %.2f\n", average); // Print the value
}

close(client_socket); // Close connection after communication is done
```

Abbildung 6: Endlosschleife Server Client Funktion

Nachdem die Verbindung hergestellt wurde, tritt der Server in eine Endlosschleife ein, in der er einen Integer-Wert aus dem Client-Socket liest, den empfangenen Wert mit der Funktion writeToLog() in ein Protokoll schreibt, den Durchschnitt der empfangenen Werte mit der Funktion stat() berechnet und diesen Durchschnitt zurück in den Client-Socket schreibt. Wenn beim Lesen vom oder Schreiben in den Client-Socket ein Fehler auftritt, wird eine Fehlermeldung ausgegeben und die Schleife abgebrochen. Nach jeder Interaktion wird der empfangene Durchschnittswert auf der Konsole ausgegeben.

Sobald die Schleife unterbrochen ist, wird der Client-Socket geschlossen und damit die Kommunikation beendet. Wenn weitere Clients kommunizieren wollen, müssen sie eine neue Verbindung aufbauen. Diese Funktion kapselt die Kernfunktionalitäten eines TCP/IP-Servers, einschließlich Socket-Erstellung, Bindung, Abhören, Annahme von Client-Verbindungen und Durchführung von Lese- und Schreiboperationen.

2.2 Main – Sockets

Die Hauptfunktion in diesem Programm beginnt damit, dass der Zufallszahlengenerator mit der aktuellen Zeit gefüttert wird. Dies geschieht, damit die Aufrufe von rand() bei jedem Durchlauf des Programms eine andere Folge von Zufallszahlen erzeugen. Danach wird der Systemaufruf fork() verwendet, um einen neuen Prozess zu erzeugen. Der neue Prozess ist eine exakte Kopie des aktuellen Prozesses, hat aber eine andere Prozess-ID. Fork() gibt die Prozess-ID des Kindprozesses an den Elternprozess zurück und 0 an den Kindprozess. Wenn das Programm als Kindprozess läuft (d.h. fork gibt 0 zurück), ruft es die Funktion server_client_func auf, die für die Handhabung von Server- und Client-Funktionen gedacht ist, wie zuvor definiert. Läuft das Programm dagegen als Elternprozess (d. h., fork() hat eine positive Zahl zurückgegeben), öffnet es eine Datei namens "werte.txt" im Schreibmodus. Wenn die Datei aus irgendeinem Grund nicht geöffnet werden kann, gibt es eine Fehlermeldung aus und beendet sich mit einem Rückgabewert von 1.

Unter der Annahme, dass die Datei erfolgreich geöffnet wurde, tritt der Elternprozess in eine Endlosschleife ein, in der er durch Aufruf der Funktion conv eine Zufallszahl erzeugt, diese Zahl in die Datei schreibt, die Datei leert, um sicherzustellen, dass die Zahl sofort auf die Festplatte geschrieben wird, eine Sekunde lang pausiert, den Durchschnitt der erzeugten Zahlen mit der Funktion stat berechnet und den berechneten Durchschnitt durch Aufruf der Funktion report meldet.

2.3 Herausforderungen und Lösungen

Die erste große Herausforderung war die Synchronisierung der Eltern- und Kindprozesse, um die korrekte Abfolge der Operationen zu gewährleisten. Da der untergeordnete Prozess die Server-Client-Interaktion abwickelte und der übergeordnete Prozess die Zufallszahlen generierte, sie in eine Datei schrieb und den Durchschnitt berechnete, war die Synchronisierung dieser Aktivitäten entscheidend. Durch die Verwendung der Funktion fork() konnte ein separater Prozess für die Server-Client-Funktionalität erstellt werden. Dadurch konnte der Elternprozess weiterhin Zufallszahlen erzeugen und in eine Datei schreiben, während der Kindprozess die Server-Client-Operationen unabhängig verwaltete. Die sleep(1)-Funktion wurde verwendet, um sicherzustellen, dass der Elternprozess den Kindprozess nicht überlastet.

Ein weiteres Problem, das gelöst werden musste, war die Gewährleistung einer genauen und sofortigen Aktualisierung der Datei nach dem Schreibvorgang. Jede Verzögerung in diesem Prozess hätte zu inkonsistenten oder falschen Daten führen können. Dieses Problem wurde durch die Verwendung des Befehls fflush(file) unmittelbar nach dem Schreiben in die Datei behoben. Mit diesem Befehl wurde sichergestellt, dass die Ausgabe sofort in die Datei geschrieben wurde, so dass die Server-Client-Prozesse mit den aktuellsten Daten versorgt wurden.

3. Pipes

3.1 Pipe – Funktion

Die pipeFunc-Methode ist eine spezielle Methode, die dazu dient, Daten zwischen verschiedenen Prozessen mittels Pipes zu übertragen. Sie wird mit 'void' initialisiert, da sie keinen Rückgabewert liefert. Zu Beginn der Methode, genauer gesagt in den Zeilen 41-45, werden zwei Arrays namens conv_to_log und log_to_stat initialisiert. Diese Arrays haben eine Größe von zwei und dienen dazu, Pipe-Daten beim Lesen und Schreiben von Informationen zu speichern. Die Pipes werden mittels des Befehls pipe(value) erstellt und in Array [0] (zum Lesen von Daten) und Array [1] (zum Speichern von Daten) initialisiert. Die PID-Initialisierung in den Zeilen 48-50 dient dazu, den Variablen convID, logID und statID universelle ID-Codes zuzuweisen. Diese IDs werden anschließend im Code verwendet, um einzelne Prozesse zu unterscheiden und zu steuern

```
39  void pipeFunc() { //Function
40
41      int conv_to_log[2]; // C
42      pipe(conv_to_log);//Crea
43
44      int log_to_stat[2]; // C
45      pipe(log_to_stat); //Cre
46
47
48      pid_t convID; // Generat
49      pid_t logID;
50      pid_t statID;
51
```

Abbildungen 7 : Pipe Funktion

```
52      convID = fork(); // Separation of convID into two parallel
53      if (convID == 0) { //To check if the code is in a child pr
54          // Conv Prozessing
55          close(conv_to_log[0]); //Closing the conv to log chan
56          close(log_to_stat[0]); //Closing the log_to_stat chan
57          close(log_to_stat[1]);
58
59          while (1) { //Loop for writing the values from the co
60              int random = conv();
61              write(conv_to_log[1], &random, sizeof(int)); //Wri
62              sleep(1); //Pausing the loop, so that it wouldnot
63          }
64
65          close(conv_to_log[1]); //Closing the Pipe after code
66      } else if (convID > 0) { // To check if the code is in a p
67          logID = fork(); //Separation of logID after convID int
68          if (logID == 0) { //To check if the code is in a child
69              // Log Prozessing
70              close(conv_to_log[1]);//Closing the conv_to_log ch
71              close(log_to_stat[0]);//Closing the log_to_stat ch
72
73              while (1) { //Loop for reading the Data from conv_
74                  int value;//Inizialisation the value that wil
75                  read(conv_to_log[0], &value, sizeof(int)); //R
76                  writeToLog(value);  //Log function with value
77                  write(log_to_stat[1], &value, sizeof(int));//W
78              }
79
```

Abbildung 8 : Pipe Funktion fork & Schleifen

Ab Zeile 52 beginnen die endlosen parallelen Prozesse. Zunächst wird der convID durch die fork()-Funktion in zwei parallele Prozesse aufgeteilt, wobei der Elternprozess einen Wert von 1 oder höher und der Kindprozess einen Wert von 0 hat. In den Zeilen 53 und 66 werden if-Anweisungen verwendet, um zu überprüfen, in welchem Prozess sich das Programm (basierend auf dem convID-Wert) gerade befindet.

Wenn sich das Programm im Kindprozess befindet, wird die Datenübertragung über die Pipes bei log_to_stat und die Lese-Funktion bei conv_to_log vollständig geschlossen, um unnötige Datenübertragungen zu vermeiden. Dann wird in Zeile 59 eine while(1)-Schleife erstellt, die durch die Einstellung des Wertes '1' endlos wird. Innerhalb der Schleife wird dem Wert 'random' eine Zahlenausgabe von 'conv' zugewiesen, die dann in Zeile 61 verwendet wird, um den 'conv'-Wert in der 'conv_to_log'-Pipe zu speichern und weitere Datenübertragungen durchführen zu können.

In Zeile 65, wenn das Programm abgebrochen wird, wird 'conv_to_log' auch vollständig geschlossen. Wenn sich das Programm (basierend auf dem convID-Wert) im Elternprozess befindet, wird der logID auch durch die fork()-Funktion in zwei parallele Prozesse aufgeteilt.

Es ist wichtig zu beachten, dass dieser Prozess nach der Aufteilung von "convID" stattfindet, was sicherstellt, dass wir immer die korrekte Reihenfolge der Prozesse haben. In Zeile 68 prüft das Programm (basierend auf dem logID-Wert), ob es sich im Kindprozess befindet. Falls dies der Fall ist, werden die Schreibfunktionen von conv_to_log Pipe und die Lesefunktionen von log_to_stat Pipe geschlossen, um unnötigen Datenaustausch zu vermeiden. In Zeile 73 wird erneut eine while(1)-Schleife geöffnet, in der ein Wert initialisiert wird. Dieser Wert dient dazu, den gelesenen Wert aus der conv_to_log Pipe zu speichern und anschließend in der writeToLog-Methode zu verwenden. In Zeile 77 wird der Wert mittels der write-Funktion in der "log_to_stat" Pipe gespeichert. Wenn der Code abbricht, werden in den Zeilen 80-81 die conv_to_log und log_to_stat Pipes vollständig geschlossen. Bitte geben Sie eine detaillierte Erklärung des Prozesses, der nach der Aufteilung von "convID" stattfindet, einschließlich des Zwecks der while(1)-Schleife, der Rolle der Wertvariable und der Funktion der conv_to_log und log_to_stat Pipes. Zusätzlich erklären Sie bitte, wie das Programm feststellt, ob es sich im Kindprozess befindet und warum die Pipes in diesem Fall geschlossen werden.

3.2 Main – Pipes

Der Pipes Code wird in eigenem Main ausgeführt, in Zeile 113 wird erst die Funktion verwendet, die den Zufallsgenerator mit einem Startwert installiert, der von der aktuellen Systemzeit abgeleitet wird. Dadurch wird sichergestellt, dass bei der Verwendung von Funktionen wie rand() zufällige Werte erzeugt werden. Dann in Zeile 115 wird das Signal SIGINT behandelt, das gesendet wird, wenn der Benutzer die Tastenkombination Ctrl+C drückt, um das Programm zu beenden. Die Funktion terminate wird als Signal-Handler für SIGINT registriert. Wenn das Signal empfangen wird, wird die Funktion terminate aufgerufen. Schließlich in Zeile 116 wird der pipeFunc() Methode aufgerufen.

```
110         }
111     }
112     int main() {
113         srand(time(NULL));
114
115         signal(SIGINT, terminate);
116         pipeFunc();
117
118         return 0;
119     }
120
```

Abbildung 9: Main Pipes

3.3 Herausforderungen und Lösungen

Während der Implementierung des Programms sind wir auf bestimmte Probleme gestoßen, von denen einige gelöst wurden und andere nicht. Das erste Problem war die Implementierung der Prozesse in der richtigen Reihenfolge, die anschließend mit Hilfe von ähnlich Prinzip wie Vererbung und if Anweisung gelöst wurde. Wir hatten auch gewisse Probleme mit dem Schreiben von Zahlen in ein Textdokument, dem Berechnen und Anzeigen von Daten mithilfe von Pipes. Dieses Problem wurde gelöst, indem unnötige Prozesse geschlossen und mit PID gesteuert wurden. Leider konnten wir nicht herausfinden, wie wir Prozesse mit den Befehlen pstree, ps und top steuern können. Die Teams arbeiteten, aber immer wieder kam etwas Unklares heraus. Generell hat sich der Code für Pipes als recht logisch und leicht verständlich herausgestellt, wenn man etwas länger über die Implementierung nachdenkt.

4. Message Queues

Dieser Abschnitt des Dokuments gibt einen detaillierten Überblick und Erklärungen zum Consumer-Producer-Message-Queue-System. Das System nutzt Nachrichtenwarteschlangen, um die Kommunikation zwischen einem Verbraucherprozess (Kind) und einem Produzentenprozess (Elternteil) zu ermöglichen. Es wird mit Hilfe von POSIX IPC (Interprozesskommunikation) Mechanismen implementiert. Der Verbraucherprozess empfängt Nachrichten, berechnet den Durchschnitt der empfangenen Werte und erstellt einen Bericht. Gleichzeitig generiert der Produzentenprozess Zufallszahlen, verpackt diese in Nachrichten und sendet sie an den Verbraucherprozess.

4.1 Consumer – Funktion

Die Funktion **consumer()** des Codes stellt den Consumer-Prozess dar. Sie beginnt mit der Generierung eines Schlüssels mit der Funktion **fork()** aus den Headern **<sys/types.h>** und **<sys/ipc.h>**. Dieser Schlüssel wird für die Erstellung einer Nachrichtenwarteschlange mit der Funktion **msgget()** aus dem **<sys/msg.h>**-Header verwendet. Innerhalb einer Endlosschleife empfängt die Funktion **consumer()** mit Hilfe der **Funktion msgrcv()** Nachrichten vom Producer-Prozess. Die Funktion extrahiert den Wert aus der empfangenen Nachricht und protokolliert ihn durch Aufruf der Funktion **log_value()**. Anschließend berechnet sie den Mittelwert der empfangenen Werte mit Hilfe der Funktion **stat()**. Um Informationen über die aktuelle Gesamtsumme und den Mittelwert zu erhalten, ruft er die Funktion **report()** auf. Wenn der Consumer-Prozess beendet ist, verwendet er die Funktion **msgctl()**, um die Nachrichtenwarteschlange aus dem System zu entfernen. Dies gewährleistet eine ordnungsgemäße Bereinigung und gibt Systemressourcen frei.

4.2 Producer – Funktion

Die **producer()**-Funktion stellt den Producer-Prozess dar. Sie hat eine ähnliche Struktur wie die Funktion **consumer()**, weist aber einige Unterschiede auf. Die Funktion **producer()** beginnt mit der Erzeugung einer Zufallszahl mit Hilfe der Funktion **conv()**. Diese Zufallszahl wird dann in eine Nachrichtenstruktur eingefügt und der Nachrichtentyp wird festgelegt. Die Nachricht wird mit Hilfe der Funktion **msgsnd()** aus dem **<sys/msg.h>**-Header an den Consumer-Prozess gesendet. Nach dem Senden einer Nachricht wartet die Funktion **producer()** mit der Funktion **sleep()** aus dem **<unistd.h>**-Header eine Sekunde lang. Diese

```c
void producer() {
    key_t key = ftok(".", msg_queue_key);
    int producer_queue = msgget(key, 0666 | IPC_CREAT);
    if (producer_queue == -1) {
        printf("Error creating producer message queue.\n");
        return;
    }
```

Abbildung 10: Producer Funktion

Verzögerung ermöglicht eine kontrollierte Rate der Nachrichtenproduktion. Wenn der Producer-Prozess beendet ist, verwendet er die Funktion **msgctl()**, um die Nachrichten-Warteschlange aus dem System zu entfernen, um eine ordnungsgemäße Bereinigung sicherzustellen und Systemressourcen freizugeben.

4.3 Main – Message Queues

Die Funktion **main()** dient als Einstiegspunkt. Sie beginnt mit der Initialisierung des Zufallszahlengenerators mit **srand()** aus dem **<time.h>**-Header. Der Seed für den Zufallszahlengenerator wird aus der aktuellen Zeit gewonnen, so dass bei jedem Programmdurchlauf eine andere Folge von Zufallszahlen entsteht. Als nächstes registriert die Funktion **main()** das SIGINT-Signal mit **signal()** aus dem **<signal.h>**-Header. Die Funktion stellt den Signalhandler so ein, dass er die Funktion **terminate()** aufruft, wenn das SIGINT-Signal, das normalerweise durch das Drücken von Strg+C ausgelöst wird, empfangen wird. Dadurch kann das Programm ordnungsgemäß beendet werden. Die Funktion **main()** forkt dann einen Kindprozess, um die Consumer-Funktionalität zu erstellen. Wenn die Abspaltung erfolgreich ist, führt der Kindprozess die Funktion **consumer()** aus, während der Elternprozess die Funktion **producer()** ausführt. Diese Trennung ermöglicht die gleichzeitige Ausführung der Consumer- und Producer-Prozesse. Schlägt der Fork fehl, wird eine Fehlermeldung ausgegeben, die auf den Fehler hinweist. Abschließend gibt die Funktion **main()** 0 zurück, um die erfolgreiche Ausführung des Programms anzuzeigen.

4.4 Herausforderungen und Lösungen

Während wir das Nachrichtenwarteschlangensystem implementierten, stießen wir auf Schwierigkeiten bei der Funktion "msgrcv()", die für den Empfang von Nachrichten zuständig ist. Dabei traten verschiedene Fehler auf, wie eine leere Warteschlange, ungültige Nachrichtentypen oder Probleme mit den Funktionsparametern. Aufgrund dieser Probleme konnte der Consumer-Prozess die erwarteten Nachrichten vom Producer-Prozess nicht empfangen, was den reibungslosen Informationsfluss unterbrach. Der fehlgeschlagene Empfang der Nachrichten stellte eine große Herausforderung dar, da er die Datenübertragung zwischen den Prozessen beeinträchtigte und die gesamte Funktionalität des Systems beeinflusste.

Um die Empfangsfehler im Nachrichtenwarteschlangensystem zu beheben, haben wir mehrere Lösungen umgesetzt. Zunächst haben wir eine Fehlerbehandlung implementiert, indem wir den Rückgabewert von "msgrcv()" überprüft und spezifische Fehlerbedingungen behandelt haben. Des Weiteren haben wir eine Wartezeit für leere Warteschlangen eingeführt, um Nachrichten zu empfangen, sobald sie verfügbar sind. Außerdem haben wir eine gründliche Validierung der Funktionsparameter durchgeführt, um Empfangsfehler aufgrund von Parameterproblemen zu vermeiden. Durch die Implementierung dieser Lösungen konnten wir die Probleme effektiv beheben. Der Consumer-Prozess konnte die erwarteten Nachrichten nun problemlos vom Producer-Prozess empfangen und die Systemfunktionalität wurde wiederhergestellt.

5.Shared Memory

Dieses Teil des Dokuments beschreibt den Aufbau der realisierten Lösung, die die Kommunikation zwischen Prozessen mithilfe von Shared Memory und Semaphore ermöglicht. Der Code besteht aus einer Eltern-und einer Kindprozessfunktion, die jeweils in einer Endlosschleife ausgeführt werden. Ziel des Codes ist es, Daten zwischen den Prozessen auszutauschen und eine synchronisierte Ausführung zu gewährleiten.

5.1 Struktur

Der Zufallszahlengenerator wird mit srand(time(NULL)) initialisiert, um zufällige Generierung von Zahlen sicherzustellen. Anschließend wird ein Signalhandler für das Signal SIGINT (Strg+C) mit signal(SIGINT,terminate) registriert. Dadurch kann das Strg+C-Ereignis behandelt werden, falls das Programm normal beendet wird.

```
// Create shared memory segment
key_t shm_key = ftok(".", 'x');

int shm_id = shmget(shm_key, SHM_SIZE, IPC_CREAT | 0666);
if (shm_id < 0) {
    printf("Error creating shared memory segment.\n");
    return 1;
}

// Attach shared memory segment
int* shm_ptr = (int*)shmat(shm_id, NULL, 0);
if (shm_ptr == (int*)-1) {
    printf("Error attaching shared memory segment.\n");
    return 1;
}
```

Abbildung 11: Shared memory segment

Für das gemeinsam genutzte Speichersegment wird ein eindeutiger Schlüssel generiert. Die Funktion ftok() benötigt zwei Parameter: den Dateinamen oder Pfad und die Projekt-ID. In diesem Fall wird der aktuelle Verzeichnispfad („.“) als Dateiname verwendet und der Buchstabe „x“ ist die Projekt-ID. Die Kombination dieser beiden Parameter erzeugt einen eindeutigen Schlüssel, der dann für den Zugriff auf die entsprechende Freigabe verwendet wird Speichersegment.

Shared-Memory-Segmente wird mit der Funktion shmget() erstellt, die ein Shared-Memory-Segment mit einer bestimmten Größe und Zugriffsberechtigungen erstellt. Der von shm_id zurückgegebene Wert ist die Kennung des gemeinsam genutzten Speichersegments, das erstellt und für den Zugriff auf das nächste Segment verwendet wurde. Die Funktion shmget() benötigt drei Parameter: den Schlüssel, die Segmentgröße und zusätzliche Flags, die das Verhalten der Funktion definieren. In diesem Fall IPC_CREAT | 0666 gebraucht. Das IPC_CREAT-Flag gibt an, dass ein gemeinsam genutztes Speichersegment erstellt werden soll, wenn es noch nicht vorhanden ist. Wenn ein Segment mit dem angegebenen Schlüssel bereits vorhanden ist, wird einfach die ID des vorhandenen Segments zurückgegeben. Die Autorität 0666 ermöglicht Eigentümern, Gruppen und anderen Benutzern Lese- und Schreibzugriff auf das gemeinsam genutzte Speichersegment. Es prüft auch auf Fehler, um eine erfolgreiche Segmenterstellung sicherzustellen. Das Anhängen eines Shared-Memory-Segments an den Adressraum eines Prozesses erfolgt mit shmat(). Dies ermöglicht den Zugriff auf gemeinsam genutzte Speicherbereiche. Auch hier wird überprüft, ob Fehler vorliegen, um sicherzustellen, dass der Anhang erfolgreich war.

Semaphore wird mit semget() erstellt. Semaphore werden verwendet, um den Zugriff auf den gemeinsamen Speicher zu regeln und Konflikte zwischen Prozessen zu vermeiden. Die von sem_id zurückgegebene Werte sind shm_id ähnlich. Der Zahl 1 bezieht sich aber auf die Anzahl der Semaphore, die erstellt werden müssen.

```
// Create semaphore
key_t sem_key = ftok(".", 's');
int sem_id = semget(sem_key, 1, IPC_CREAT | 0666);
if (sem_id < 0) {
    printf("Error creating semaphore.\n");
    return 1;
}

// Initialize semaphore value to 1
semctl(sem_id, 0, SETVAL, 1);
```

Abbildung 12: Semaphore

Ebenso wird auf Fehler geprüft, um sicherzustellen, dass das Semaphor erfolgreich erstellt wurde. Die Initialisierung eines Semaphors mit dem Wert 1 wird durch semctl() durchgeführt. Dies definiert den Anfangszustand des Semaphors, indem zunächst der Zugriff auf den gemeinsam genutzten Speicher ermöglicht wird. Untergeordnete Prozesse werden mit fork() geforkt. Dadurch wird eine Kopie des aktuellen Prozesses erstellt und die beiden Prozesse können unabhängig voneinander arbeiten. Es prüft auf Fehler, um sicherzustellen, dass der Fork-Vorgang erfolgreich war.

Nach der Fork wird je nach Prozesstyp (Kind oder Eltern) die entsprechende Funktion aufgerufen. Die child_process()-Funktion repräsentiert den Kindprozess und ist für die Generierung von Zufallszahlen, das Schreiben in den Shared Memory, die Verwendung von Semaphore-Operationen zur Regulierung des Zugriffs auf den Speicher und weitere Schritte wie das Schreiben in eine Logdatei und die Berechnung des Mittelwerts zuständig. Der Elternprozess wird durch die parent_process()-Funktion repräsentiert und führt ähnliche Schritte aus wie der Kindprozess, liest jedoch den Wert aus dem Shared Memory, berechnet den Mittelwert und meldet ihn. Beide Funktionen verwenden eine Semaphor-Operation, um den Speicherzugriff zu steuern. Es werden die Semaphore sem_wait und sem_signal verwendet.

Die Operation sem_wait verringert den Semaphor um 1 und die Operation sem_signal erhöht den Wert um 1. Dies steuert den Speicherzugriff und ermöglicht die Fortsetzung von Sperrprozessen, wenn der Semaphor freigegeben wird. Nach Abschluss der Aufgaben in den Funktionen wird der gemeinsame Speicher freigegeben, indem er mit dem Befehl shmdt() getrennt wird. Anschließend wird das Shared-Memory-Segment mit shmctl() und das Semaphor mit semctl() gelöscht. Dieses Dokument enthält detaillierte Erläuterungen zu verschiedenen Aspekten des Codes, einschließlich der Verwendung von Semaphor Operationen zur Regulierung des Zugriffs auf gemeinsam genutzten Speicher und zur Aufhebung der Zuordnung.

6.Main Funktion

Dieser Code dient als Einstiegspunkt für ein Programm, das es dem Benutzer ermöglicht, zwischen vier verschiedenen Methoden der Interprozesskommunikation (IPC) zu wählen. Diese IPC-Methoden sind: Sockets (S), Pipes (P), Message Queues (M) und Shared Memory (SM).

Die Hauptfunktion des Programms zeigt dem Benutzer zunächst eine Meldung an, in der der Zweck des Programms und die verfügbaren Optionen beschrieben werden. Danach wartet es auf die Eingaben des Benutzers. Das Programm verwendet eine Zeichenvariable, choice, um die Eingaben des Benutzers zu erfassen. Je nach Auswahl des Benutzers führt es die entsprechende Methode der IPC aus.

```
Hello, in this code, data will be exchanged using 4 different methods of your choice
Press the button with which you want to exchange data
P - for Pipes, S - for Sockets, M - for Message Queues, SM - for Shared Memory
```

Abbildung 12: Optionen in Terminal

Falls der Benutzer eine Option eingibt, die nicht 'S', 'P', 'M' oder 'SM' ist, gibt das Programm die Fehlermeldung "Ungültige Wahl" aus und gibt 1 zurück, um einen Fehler zu signalisieren.

Wenn die Auswahl des Benutzers gültig war und das entsprechende Programm erfolgreich ausgeführt wurde, gibt die Hauptfunktion am Ende 0 zurück und zeigt damit an, dass das Programm erfolgreich abgeschlossen wurde.

```c
int main()
{
    printf("Hello, in this code, data will be exchanged using 4 different methods of your choice\n");
    printf("Press the button with which you want to exchange data\n");
    printf("P - for Pipes, S - for Sockets, M - for Message Queues, SM - for Shared Memory\n");

    char choice;
    scanf(" %c", &choice);
    int return_value;
    // Execute the desired compiled biary file
    if (choice == 'S')
    {
        return_value = system("./sockets");
        if (return_value == -1)
        {
            printf("Failed to execute ./sockets\n");
        }
    }
    else if (choice == 'P')
    {
        return_value = system("./pipes");
        if (return_value == -1)
        {
            printf("Failed to execute ./pipes\n");
        }
    }
```

Abbildung 14: Main Code