Betriebssysteme und Rechnernetze

Prof. Dr. Christian Baun

Portfolioprüfung – Werkstück A – Alternative 1

SS2022

Frankfurt University of Applied Sciences

Gruppenarbeit von Ayoub Madani, Benedikt-Josip Kovac, Nelli Margaryan, Theresa Alten, Younes Rifi Kafiouni

Dieses Dokument beinhaltet die Dokumentation eines Gruppenprojektes im Modul Betriebssysteme und Rechnernetze. Das Projekt besteht außerdem noch aus verschiedenen Quellcodes (in C geschrieben), sowie einer PowerPoint Präsentation. Die Dokumentation enthält eine Zusammenfassung der Aufgabenstellung, zudem eine Beschreibung der unterschiedlichen Herangehensweisen und Lösungsansätze zum erarbeiteten Code.

Inhaltsverzeichnis:

1. Einleitung

Die Aufgabenstellung der Alternative 1 des Werkstück A fordert ein Echtzeitsystem das Zufallszahlen bildet, aus diesen Mittelwert und Summe berechnet, die errechneten Werte anschließend in der Shell ausgibt und zusätzlich eine lokale Datei, mit den Zufallszahlen als Inhalt, erstellt. Anschließend folgen weitere Anforderungen an das System, die in den nächsten Zeilen näher beschrieben werden.

Das entwickelte und implementierte Echtzeitsystem besteht aus vier Prozessen. Der Prozess *Conv* erzeugt Zufallszahlen. Der Prozess *Log* liest die Messwerte von *Conv* aus und schreibt sie in eine lokale Datei. Prozess *Stat* liest die Messwerte von *Conv* aus und berechnet statistische Daten wie Mittelwert und Summe daraus. Der Prozess *Report* greift auf die Ergebnisse von *Stat* zu und gibt die statistischen Daten in der Shell aus. Die Zufallszahlen aus *Conv* haben keine fest definierte Anzahl. Es können bis zu zehn Zahlen erzeugt werden. Die Zufallszahlen werden in einem Array gespeichert (array[10]). Der mögliche Wertebereich der gebildeten Zufallszahlen liegt zwischen 0 und 10.

Dieses System inklusive Datenaustausch soll mit vier verschiedenen Implementierungsvarianten in der Sprache C programmiert werden: Pipes, Message Queues, Shared Memory mit Semaphoren und mit Sockets.

Darüber hinaus sollen die Prozesse als parallele Endlosprozesse laufen und mit der Tastenkombination Ctrl + C abgebrochen werden können. Zusätzlich sollen die Message Queues, Shared Memory-Bereiche und Semaphoren mit dem Kommando ipcs überwacht werden. Entwickelt wurde der Code über das Linux Betriebssystem.

Die vier Implementierungsvarianten wurden von je ein bis zwei Gruppenmitgliedern bearbeitet. Folglich wurden Pipes von Nelli Margaran, Message Queues von Benedikt-Josip Kovac, Shared Memory von Younes Rifi Kafiouni und Ayoub Madani und Sockets von Theresa Alten behandelt.

2. Erstellen von Prozessen

Eine Möglichkeit die Prozesse zu erstellen ist der Systemaufruf fork(). Bei fork() erzeugt der aktuelle Prozess eine Kopie von sich selbst. Der Kindprozess übernimmt die Daten vom Elternprozess und erhält vom Betriebssystem eine eigene Prozess-ID (PID). Um vier Prozesse zu erzeugen, wurde drei Mal fork() aufgerufen. Anhand des Rückgabewertes lässt sich prüfen, ob man sich im Kind- oder im Elternprozess befindet. Wenn der Rückgabewert gleich 0 ist, befindet man sich im Kindprozess, wenn der Rückgabewert größer als 0 ist, im Elternprozess.

Zu Beginn des Quellcodes wurde der Prozess *Conv* erzeugt. Mit dem Systemaufruf fork() wurde aus *Conv* ein Kindprozess erzeugt. Der Kindprozess heißt *Stat*. Mit einer If-Anweisung wurde geprüft, wie groß der Rückgabewert ist. Wenn der Rückgabewert gleich Null ist, befindet man sich im *Stat*-Prozess, wenn der Rückgabewert ungleich Null ist, befinden wir uns immer noch im *Conv*-Prozess und erzeugen einen weiteren Kindprozess. Der neue Kindprozess heißt *Log*. Anschließend wird aus *Stat* noch ein Kindprozess erzeugt. Der Kindprozess von *Stat* heißt *Report*.

3. Pipes

Damit das Programm einen Datentransfer realisieren kann, gibt es anonyme Pipes als eine Möglichkeit für Interprozesskommunikation. Die anonyme Pipe ist ein Kommunikationskanal zwischen zwei Prozessen. Da das Programm aus vier Prozessen besteht, wurden dementsprechend drei Pipes angelegt: *firstpipe[2], secondpipe[2]* und *thirdpipe[2]. Firstpipe[2]* ermöglicht die Kommunikation zwischen *Conv* und *Stat*, secondpipe[2] ermöglicht sie zwischen *Conv* und *Log* und thirdpipe[2] zwischen *Stat* und *Report*.

Der Systemaufruf pipe() erhält vom Betriebssystem zwei Zugriffskennungen. Die erstellten Pipes, sind einfache Arrays, die aus zwei Elementen bestehen. An der nullten Stelle liest (read())Pipe die Daten, an der ersten Stelle schreibt (write()) die Daten und somit ermöglicht einen Datenstrom zwischen zwei Prozessen. Beim

Lesen auf Pipes muss die Schreibseite geschlossen sein (close(firstpipe[1])). Nachdem die Daten gelesen werden, schließt man auch die Leseseite (close(firstpipe[0])). Mit dem selbem Prinzip geht man auch beim Schreiben auf Pipes vor. Bei der Erzeugung von Kindprozessen mit fork() erben die Kindprozesse auch den Zugriff auf die Zugriffskennungen. *(Quelle: Prof. Dr. Christian Baun – 6. Foliensatz Betriebssysteme und Rechnernetze – FRA-UAS – SS2022)*

Anonyme Pipes wurden verwendet, weil die erstellten Prozesse eng miteinander verwandt sind. Da immer eine unterschiedliche Anzahl von Zufallszahlen erzeugt wird, reicht die Übertragung von dem Array alleine nicht aus: (write(firstpipe[1], array, sizeof(int) * anzahlZahlen). Die Anzahl der Zufallszahlen muss auch auf Pipe geschrieben und im Kindprozess gelesen werden: write(firstpipe[1], &anzahlZahlen, sizeof(int). Die Summe und der Mittelwert sind einfache Zahlenwerte und werden sehr einfach auf Pipes geschrieben und im Kindprozess gelesen: read(thirdpipe[0], &summe, sizeof(summe); read(thirdpipe[0], &mittelwert, sizeof(float).

Am Anfang des Quellcodes zu Pipes sind noch zwei verschiedene Methoden, welche die Prozesse abbrechen lassen, sobald man Strg + C gedrückt hat.

4. Message Queues

In Message Queues werden Nachrichten empfangen und gesendet. Diese Nachrichten werden oftmals nicht sofort abgerufen und werden daher in eine Warteschlange verschickt beziehungsweise dort gespeichert. Aufgrund dessen können Prozesse unabhängig, also asynchron, miteinander kommunizieren. Der Grundaufbau einer Message Queue sind verkettete Listen, die Nachrichten beinhalten. In diesem Programm kommunizieren die Prozesse anhand einer Message Queue und der *System V* Nachrichtenwarteschlange. Es werden jedem Prozess Nachrichtentypen vergeben, die sie Empfangen oder Senden dürfen.

Anhand der vorab definierten Variablenart *„message“,* wird die Struktur der Warteschlange, Nachrichtenart und Nachrichtengröße vorgegeben. Hier werden unsere Sende- und Empfangswarteschlange gespeichert.

```
message sendBuffer;    // Empfangspuffer
message receiveBuffer; // Sendepuffer
```

Message Queues enthalten Informationen über den Nachrichteninhalt, ihren Nachrichtentyp, Größe und Speicherort. Um eine Message Queue beziehungsweise Warteschlange zu erzeugen oder einen Zugriff auf bestehende Queues zu gewähren, wird die Methode *„msgget()“* aufgerufen.

```
codeMessageGet = msgget(messageKey, IPC_CREAT | 0600); // IPC CREAT Neuerzeugung der Message Queue/ Warteschlange
if (codeMessageGet < 0)
{
    printf("Wartrschlange wurde nicht erstellt!\n");
    perror("messageGet");
    exit(1);
}
else
{
    printf("Warteschlange %i inklusive ID %i ist bereit\n", messageKey, codeMessageGet);
}
```

Bevor man eine Nachricht sendet oder empfängt wird der Pfad angegeben, indem man den Nachrichtenwarteschlangen die zugelassenen Nachrichtentypen deklariert. In diesem Programm schickt der Prozess Conv seine Nachrichten an *Stat* und *Log* und erhält von niemanden eine Nachricht. Des Weiteren schickt *Stat* nur an *Report* eine Nachricht. *Log* und *Report* erhalten nur Nachrichten, sie versenden keine.

```
sendBuffer.messageType = 3;
receiveBuffer.messageType = 2;
```

Daraufhin wird die Nachricht des Prozesses reingeschrieben.

```
strncpy(sendBuffer.messageSize, "Stat: PID: %i mittelwert %i summe %d", getpid() && mittelwert && summe); // Nachricht in Sendepuffer schreiben
```

Mit dem Methodenaufruf *„msgsnd()“* wird eine Nachricht in eine Queue reingeschrieben und durch *„msgrcv()“* aus der Queue gelesen beziehungsweise entfernt.

```
(msgsnd(codeMessageGet, &sendBuffer, sizeof(sendBuffer.messageSize), IPC_NOWAIT) == -1)
```

```
msgrcv(codeMessageGet, &receiveBuffer, sizeof(receiveBuffer.messageSize), receiveBuffer.messageType, MSG_NOERROR | IPC_NOWAIT);
```

Abschließend wird eine Message Queue anhand von dieser Methode *„msgctl()“* gelöscht oder gesteuert.

```
codeMessageCTL = msgctl(codeMessageGet, IPC_RMID, 0);  //Warteschlange wird geloescht
if (codeMessageCTL < 0)
{
    printf("Warteschlange wurde nicht geloescht!");
    perror("messageCTL");
    exit(1);
}
else
{
    printf("%i mit ID %i wurde geloescht \n", messageKey, codeMessageGet);
}
```

5. Shared Memory mit Semaphoren

„Shared Memory ist ein vom Kernel verwalteter Speicherbereich, der von mehreren Prozessen gelesen und beschrieben werden kann. Es muss dabei lediglich beachtet werden, dass die einzelnen Prozesse untereinander abgeglichen, also synchronisiert werden müssen.“[1] Es gibt drei Arten von Shared Memory: einmal die Shared Memory in der Interprozesskommunikation, der Shared Memory in Mehrprozessorsystemen und Shared Memory bei Grafikkarten.

Hier die Shared Memory in der Interprozesskommunikation behandelt. „Hier nutzen zwei oder mehrere Prozesse einen bestimmten Teil des Hintergrundspeichers (RAM) gemeinsam. Für alle beteiligten Prozesse liegt dieser gemeinsam genutzte Speicherbereich in deren Adressraum und kann mit normalen Speicherzugriffsoperationen ausgelesen und verändert werden.“[2]

Der erste Schritt ist die Erstellung/Erzeugung eines oder mehreren Segmentes/n. Durch die Funktion

```
returncode_shmget1 = shmget(shared_memory_key1, MAXMEMSIZE, IPC_CREAT
| 0600);
```

[1] https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap09-006.htm, Autor: Jürgen Wolf
[2] https://de.wikipedia.org/wiki/Shared_Memory

```c
// Gemeinsames Speichersegment erzeugen
returncode_shmget1 = shmget(shared_memory_key1, MAXMEMSIZE, IPC_CREAT | 0600);

if (returncode_shmget1 < 0)
{
    printf("Das Segment konnte nicht erstellt werden.\n");
    perror("shmget");
    exit(1);
}
else
{
    printf("Das 1.Segment wurde erstellt.\n");
}
```

wird das Segment erstellt. Nach dem Erstellen eines Segmentes müssen die Shared Memory Segmente angehängt werden. Die Funktion dafür lautet:

```c
sharedmempointer1 = (char *)shmat(returncode_shmget1, 0, 0);.
```

```c
// Gemeinsames Speichersegment anhängen
sharedmempointer1 = (char *)shmat(returncode_shmget1, 0, 0);
if (sharedmempointer1 == (char *)-1)
{
    printf("Das Segment konnte nicht angehängt werden.\n");
    perror("shmat");
    exit(1);
}
else
{
    printf("Das 1.Segment wurde angehängt.\n");
}
```

Damit wird der Adressraum angehängt. Diese Schritte werden so lange wiederholt, bis die gewünschte Anzahl der Segmente erstellt worden ist. Nach dem Erzeugen und dem Anhängen eines Segments werden die Prozesse erzeugt. Näheres dazu in Punkt 2.

Zur guter Letzt müssen die Segmente gelöst und gelöscht werden. Durch das `shmdt(sharedmempointer1);` wird das Shared Memory gelöst und durch das `shmctl(returncode_shmget1, IPC_RMID, 0);` gelöscht.

Damit es nicht vorkommt, dass auf mehr als einen Prozess aus diesem Segment zugegriffen wird, verwendet man häufig die Semaphoren. „Wenn sich ein Prozess in seinem kritischen Abschnitt befindet, sollte kein anderer Prozess ständig nachfragen, ob er in den kritischen Abschnitt eintreten darf“. „Vielmehr sollte es möglich sein, dass ein Prozess, der eintreten möchte, deaktiviert und wieder aktiviert/aufgeweckt wird, wenn der derzeitige aktive Prozess seinen kritischen Abschnitt verlässt.“ [3]

Als Allererstes wird ein Semaphor mit `sem_t semaphore;` erstellt und mit

[3] Abgewandelt von: https://www.karteikarte.com/card/1120795/was-ist-ein-semaphor

`sem_init(&semaphore, 0, 1);` initialisiert. Außerdem gibt das `sem_init` an, dass der Semaphor mit den Prozess arbeiten soll.

```
// Semaphore
// erstellung Semaphor
sem_t semaphore;
// Initialisierung des Prozess
sem_init(&semaphore, 0, 1);
```

Ist dies erledigt, kommt an jeden Anfang eines Prozesses die Funktion `sem_wait(&semaphore);`.

```
//Semaphore sperren
sem_wait(&semaphore);
```

Diese sperrt die Semaphore. Zusätzlich wird am Ende eines jeden Prozesses die Semaphore wieder entsperrt. Das wird mithilfe der Funktion `sem_post(&semaphore);` realisiert.

```
//Semaphore entsperren
sem_post(&semaphore);
```

Wenn folglich der Wert kleiner 0 wird, wird ein anderer Prozess, der durch `sem_wait()` wartet „freigegeben“ und die aktuelle Semaphore gesperrt. Alle anderen Prozesse, die jetzt ebenfalls in diesen kritischen Bereich kommen, müssen nun warten, bis die Variable wieder einen positiven Wert bekommt (bspw. 1). Am Ende eines Programms müssen die Semaphoren entfernt und geschlossen werden. Dies wird durch die Funktionen `sem_unlink("Semaphore");`

```
//Semaphore entfernen----
sem_unlink("Semaphore");
```

Und

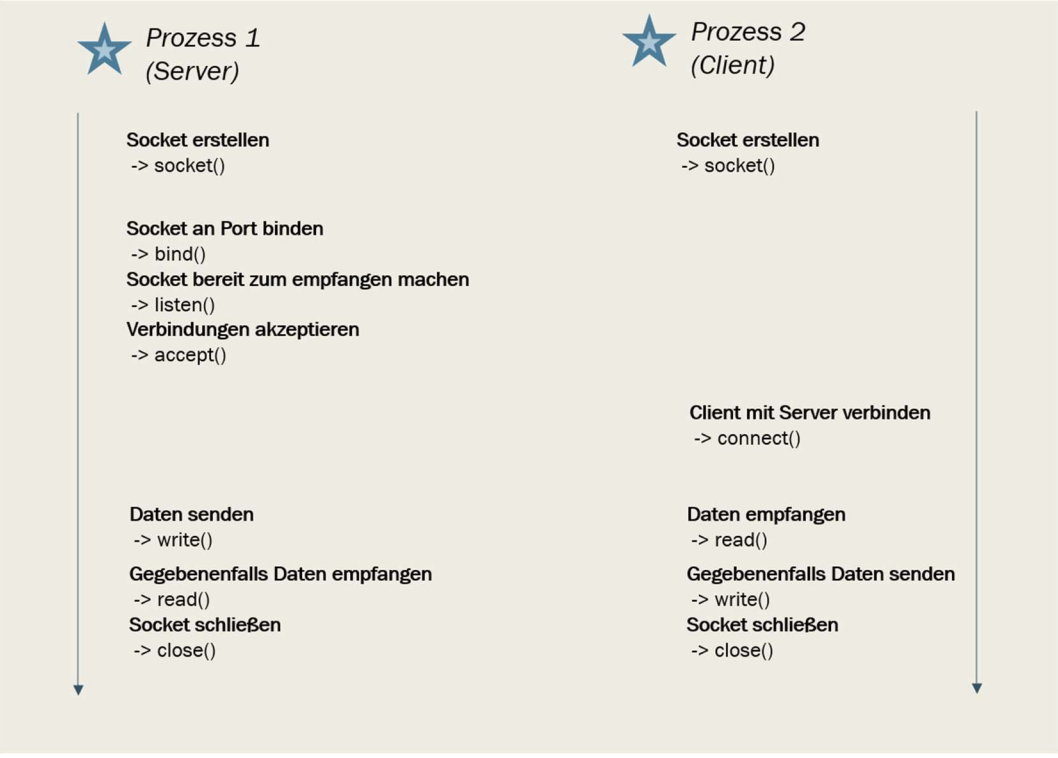
```
sem_close(&semaphore);
```

ermöglicht.

```
sem_close(&semaphore);
```

6. Sockets

Die letzte, in der Aufgabenstellung verlangte, Möglichkeit um Interprozesskommunikation zu ermöglichen ist Sockets. Für diese Art der Kommunikation sind TCP Sockets (verbindungsorientierte Stream Sockets) besser geeignet als die verbindungslosen UDP (Datagram) Sockets. Stream Sockets sind zwar langsamer als Datagram Sockets, zeichnen sich aber wiederum durch eine höhere Verlässlichkeit aus. Im Gegensatz zu den anderen vorgestellten Möglichkeiten, haben Sockets den Vorteil, dass Sie auch über Rechnergrenzen hinaus miteinander kommunizieren können. Dies funktioniert über die Zuweisung eines Portes und einer IP-Adresse. Folgende Grafik zeigt die beiden parallel laufenden Prozesse mit den einzelnen verwendeten Schritten.



Zuerst wird im Server und im Client ein Socket erstellt. Anschließend wird im Server das Socket an einen Port gebunden. Dieser ist im Code fest vorgegeben, kann aber auch über das Terminal mit angegeben werden. Nachfolgend wird das erstellte Socket empfangsbereit gemacht und akzeptiert Verbindungen in einer Warteschlange. Danach verbindet sich der Client mit dem Server und ist bereit für einen Datenaustausch. Im Code beginnt dieser mit dem Daten senden von *CONV* aus dem Server. Der Client empfängt diese übergebenen Daten und verwendet sie für *LOG, STAT* und *REPORT.* Gegebenenfalls kann nun der Client auch Daten in den Server zurückschicken. Abschließend werden die erstellten Sockets wieder geschlossen.

Für die Erstellung der Endlosprozesse wird eine while()-Schleife verwendet. Diese lässt den eben beschriebenen Ablauf der Prozesse CONV, STAT, LOG und REPORT unendlich oft durchführen, bis die Tastenkombination STRG + C betätigt wird und das Programm abbrechen lässt.

7. Schlussfolgerung und Literaturverzeichnis

In diesem Dokument wurden nun vier verschiedene Umsetzungsarten der Interprozesskommunikation vorgestellt. Jede von ihnen hat Vor- und Nachteile, je nach Einsatzgebiet. In der präsentierten Anwendung wurden alle diese implementierten Möglichkeiten erfolgreich eingesetzt und haben ihre Funktionstüchtigkeit bewiesen.

Literatur Ayoub Madani, Younes Rifi Kafiouni:

26.06.2022 21.00 Uhr https://www.karteikarte.com/card/1120795/was-ist-ein-semaphor

26.06.2022 21.30 Uhr https://openbook.rheinwerkverlag.de/linux_unix_programmierung/Kap09-004.htm#RxxKap09004040002D71F017100

20.06.2022-22.06.2022 und 25.06.2022-26.06.2022 https://man7-org.translate.goog/linux/man-pages

19.06.2022 16.00 – 21.00 Uhr http://www.christianbaun.de/BSRN22/Skript/Listing_9_1_shared_memory_systemv.c

25.06.2022 18.00 Uhr https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap09-006.htm

26.06.2022 20.00 Uhr https://de.wikipedia.org/wiki/Shared_Memory

Literatur Theresa Alten:

06.06.2022 14:00 Uhr http://www.willemer.de/informatik/unix/unprsock.htm

06.06.2022 14:00 Uhr Betriebssysteme und Rechnernetze (SS2022) (christianbaun.de)

06.06.2022 14.00 Uhr Linux Howtos: C/C++ -> Sockets Tutorial

14.06 16:00 Uhr Inter-process communication in Linux: Sockets and signals | Opensource.com

18.06 11:00 Uhr https://www.geeksforgeeks.org/socket-programming-cc/

18.06 11:00 Uhr C Socket Programming for Linux with a Server and Client Example Code (thegeekstuff.com)

18.06 11:00 Uhr
https://de.wikipedia.org/wiki/Socket_(Software)#:~:text=Ein%20Socket%20%28von%20engl.%20Sockel%2C%20Steckverbindung%20oder%20Steckdose%29,verwendet%20Sockets%2C%20um%20Daten%20mit%20anderen%20Programmen%20auszutauschen

Literatur Nelli Margaryan:

20.05 00:20 Uhr https://youtu.be/Mqb2dVRe0uo

20.05 00:20 Uhr https://youtu.be/6u_iPGVkfZ4

20.05 00:20 Uhr https://youtu.be/rE1n-4z_n0Y

http://www.christianbaun.de/BSRN22/Skript/bsrn_SS2022_vorlesung_06_de.pdf

http://www.christianbaun.de/BSRN22/Skript/Listing_9_5_anonyme_pipe.c

Literatur Benedikt-Josip Kovac:

20.06 18:00 Uhr https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap09-005.htm#RxxKap09005040002D81F03B100