# Dokumentation von Werkstück A

Alternative 1

Prüfer: Prof. Dr. Christian Baun
Frankfurt University of Applied Sciences
(1971-2014: Fachhochschule Frankfurt am Main) Nibelungenplatz 1
60318 Frankfurt am Main

**Victor Gandsha: 1393033**
**Sebastian Maas: 1393729**
**Michael Zitz: 1392924**
**Maurice Wendel: 1405756**
**Tim Arenz**

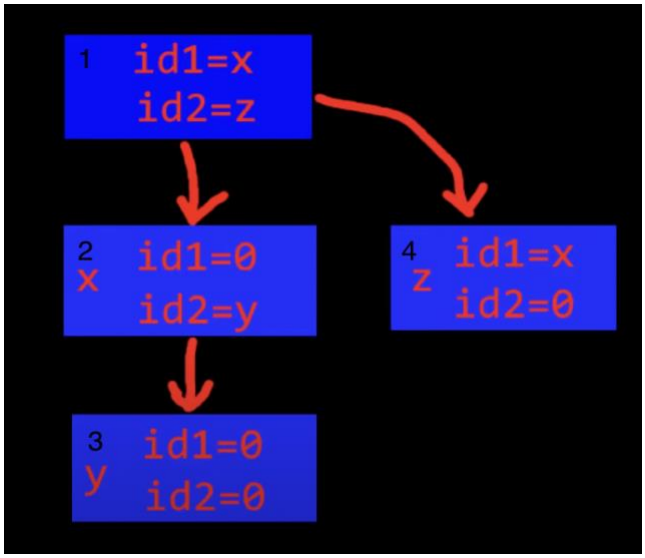
Abgabe: 27.06.2022

Diese Dokumentation beschreibt das Implementieren eines Echtzeitsystems, das aus vier Prozessen besteht. Diese sollen über vier Verschiedene Varianten miteinander kommunizieren und sich synchronisieren.

# 1 Prozess Erstellung und Identifizierung

Das erste Problem, was zu lösen galt, war zu verstehen was genau ein Prozess ist und wie man diesen respektiv mehrere erzeugt.  Die Lösung ist der Befehl fork();, der einen Kindprozess erzeugt, welcher identisch zum Elternprozess ist.

Da wir vier Prozesse benötigen, war das einmalige Aufrufen von fork(); nicht ausreichend um die erforderliche Anzahl zu erreichen, daher führen wir den Befehl direkt noch einmal aus. Somit haben wir vier Prozesse, auf die es zuzugreifen gilt.




Jeder Prozess erhält vom System eine ID, nachdem ein Prozess geforkt wurde, erhält der Kindprozess die ID 0. Bei dem ersten Anwenden des Befehls erhält der Prozess 1 die ID x und der Prozess 2 (der Kindprozess) die ID 0. Aufgrund der Tatsache das der Befehl zweimal ausgeführt wird, hat jeder Prozess zwei IDs. Da Prozess 3 und 4 beziehungsweise y und z zu diesem Zeitpunkt noch nicht existieren, besitzen sie als ID1 die gleiche ID wie ihr Elternprozess. Nachdem der Befehl nun ein zweites Mal ausgeführt wird, werden der Prozess 1 und 2 geforkt. Dadurch bekommen diese als Elternprozess eine beliebige zweite ID vom System zugewiesen und deren KIndprozesse, die Prozesse 3 und 4 die ID 0. In der oben dargestellten Grafik wird dies verdeutlicht. Durch „int id2 = fork();“ und „int id1 = fork();“ können die IDs der einzelnen Prozesse im Programm identifiziert werden.

Durch eine verschachtelte If-Anweisung in der jeweils id1 und id2 auf 0 geprüft werden, kann auf die vier Prozesse zugegriffen werden.

# 2 Grundstrukturen der Prozesse

Die Grundstruktur beinhaltet nicht die vier Implementierungsvarianten, sondern nur die Prozessaufgaben.

## 2.1 Conv

```
while (1)
{

    time_t t;
    srand(time(&t));
    int zufallszahl = rand() % 101;

    sleep(1);
}
```

Durch den Befehl rand() kann eine Zufallszahl erzeugt werden, jedoch ist diese bei jedem neu Start identisch. Um dieses Problem zu umgehen, wird die aktuelle Uhrzeit durch time_t festgestellt und als Startpunkt festgelegt. Durch Modulo 101 ist die Zufallszahl im Bereich 0 – 100. Mit sleep(1); lässt man den Prozess für eine Sekunde nach dem Schleifendurchlauf pausieren, damit das Programm nicht zu schnell durchgeführt wird.

## 2.2 Log

```
FILE *ed;
ed = fopen("ZufallszahlenMessageQueues", "a");
if (ed == 0)
{
    printf("Datei kann nicht geöffnet werden! \n");
}
else
{
    fprintf(ed, "%d\n", Zahl);
}
fclose(ed);
```

Je nach Implementierungsvariante wird die Zufallszahl an die Variable y übergeben. Mit FILE *ed wird ein Pointer erstellt, welcher einen Ort im Hauptspeicher referenziert und dessen Speicheradresse zwischenspeichert, um dort das File zu speichern. In der nächsten Zeile wird mit fopen das File mit dem Namen „Zufallszahlen“ geöffnet. Mit „a“ wird der Modus festgelegt, in dem das File geöffnet wird, das „a“ steht hierbei für „append“ und legt fest, das neue Eingaben an das Ende vom File angehängt werden. Um eventuelle Fehler

zu vermeiden, wird geprüft, ob das File geöffnet wurde, ist dies der Fall, wird mit fprintf die Zufallszahl hineingeschrieben, ansonsten wird eine Fehlermeldung ausgegeben. Am Ende muss das File wieder mit fclose(); geschlossen werden, da sonst beim nächsten Durchlauf die neue Zufallszahl nicht hineingeschrieben werden kann.

## 2.3 Stat

```
while (1)
{

    int zVonConv;

    Summe += zVonConv;
    Mittelwert = Summe / Schleifenzahl;

    Schleifenzahl++;
}
```

Die Zufallszahl wird wieder zu Beginn, je nach Implementierungsvariante, der Variable zVonConv übergeben. Die Variablen Summe, Mittelwert und Schleifenanzahl wurden bereits außerhalb des while-Schleife und der If-Anweisung definiert. Mittelwert und Summe haben dabei den Wert 0 und Schleifenanzahl den Wert 1.
Die Summe wird durch die Formel Summe = Summe + zVonConv und der Mittelwert durch Summe / Schleifenanzahl, wobei dem Wert von Schleifenanzahl nach jedem Durchlauf durch ++ um eins erhöht wird.

## 2.4 Report

```
while (1)
{

    int SummeAusS;
    int MittelwertAusS;

    printf("Akutelle Summe: %d\n", SummeAusS);
    printf("Aktueller Mittelwert: %d\n", MittelwertAusS);
}
```

Die Summe und der Mittelwert werden aus Stat den Variablen SummeAusS und MittelwertAusS übergeben und daraufhin mit printf ausgegeben.

## 3 Pipes

Die erste Variante, die wir programmieren, sind die Pipes. Diese können einerseits zum Speichern der Zufallszahl, der Summe und des Mittelwerts und andererseits zur Synchronisation verwendet werden. Eine Pipe besitzt nur zwei Enden, eines zum Schreiben und eines zum Lesen. Es kann erst gelesen werden, wenn etwas geschrieben worden ist, so wird sichergestellt, dass die Synchronisationsbedingungen erfüllt werden.

```
int pip1[2];
if (pipe(pip1) == -1)
{
    printf("ERROR with Pipe 1\n");
}
```

Auf die in dem Bild gezeigte Weise, werden zu Beginn des Programmes vier Pipes erstellt. Davon ist jeweils eine von Conv zu Stat und von Conv zu Log, von Stat zu Report sind es zwei, da eine für die Summe ist und eine für den Mittelwert. Sollte das Erstellen einer Pipe nicht funktioniert haben, wird eine Fehlermeldung ausgegeben. Zu Beginn eines Prozesses müssen vor der Unendlichkeitsschleife alle nicht benötigten Pipe-Enden und übrigen Pipes geschlossen werden.

```
close(pipS[0]);
close(pipM[0]);
close(pipS[1]);
close(pipM[1]);
close(pip2[0]);
close(pip2[1]);

close(pip1[1]);
```

Hier wird das Schließen der Pipes vor dem Prozess Log als Beispiel gezeigt. Es werden alle Pipe-Enden geschlossen bis auf das Lese-Ende der Pipe zwischen Conv und Log, damit die Zufallszahl in das File geschrieben werden kann.

Die beiden unteren Bilder zeigen, wie mit read und write in eine Pipe geschrieben werden kann. Es muss das richtige Ende, die Variable die eingelesen oder in die gelesen werden soll und der Typ der Variable angegeben werden.

```
int y;
read(pip1[0], &y, sizeof(int));
```

```
write(pip1[1], &zufallszahl, sizeof(int));
```

# 4 Message Queues

Als nächstes haben wir die Message Queues programmiert. Message Queues funktionieren ähnlich wie Pipes und haben auch einen Sender und Empfänger. Schreibt der Sender eine Nachricht wird sie in der Warteschlange zwischenspeichert. Sobald der Empfänger die Nachricht aus der Warteschlange liest, wird sie gelöscht.

```
typedef struct msgbuf
{
    int mtype;
    int Zahl;
} msg;
```

Zu Beginn des Programms wurde eine Nachrichten Struktur definiert. Als Nachrichtentyp wurde ein Integer genommen (int mytype;) und die Zahl selbst wird in dem Integer (int Zahl;) gespeichert. Zudem wird noch ein Sende- und Empfangspuffer angelegt, indem die Zahl gespeichert werden kann.

```
msg sendebuffer, recievebuffer;
```

Als nächstes müssen die verschiedenen Message Queues erstellt werden. Das geschieht mit der Funktion (msgget();). In die Klammern definiert man einen Key, um die Message Queue leichter identifizieren zu können und durch IPC_CREATE wird die Message Queue erstellt.

```
rc_msgget1 = msgget(mq_key1, IPC_CREAT);
rc_msgget2 = msgget(mq_key2, IPC_CREAT);
rc_msgget3 = msgget(mq_key3, IPC_CREAT);
rc_msgget4 = msgget(mq_key3, IPC_CREAT);
```

```
sendebuffer.Zahl = zufallszahl;
msgsnd(rc_msgget1, &sendebuffer, sizeof(sendebuffer.Zahl), 0);
```

Verschickt werden Nachrichten mit der Funktion (msgsnd();). Die Zahl selbst wird vorher in den Sendepuffer gespeichert. In den Klammern wird an erster Stelle auf die gewünschte Message Queue verwiesen. An zweiter Stelle ist die Nachricht selbst und an dritter Stelle die Größe der Nachricht. An letzter Stelle könnte man der Nachricht noch eine Priorität oder andere Eigenschaften zuweisen, dass ist hier jedoch nicht nötig.

```
msgrcv(rc_msgget1, &recievebuffer, sizeof(recievebuffer.Zahl), 0, 0);
int Zahl = recievebuffer.Zahl;
```

Empfangen werden Nachrichten mit der Funktion (msgrcv();. Der Aufbau ist ziemlich der gleiche wie beim Senden. Erst wird auf die gewünschte Message Queue verwiesen, die Nachricht im Empfangspuffer empfangen, die Größe

festgelegt und mögliche Prioritäten berücksichtigt. Die Zahl kann man nach dem Empfangen in eine Integer Variable speichern, um sie weiterzuverwenden.

```
while (msgrcv < 0)
{
    wait(NULL);
}
```

Um die Synchronisationsbedingungen zu erfüllen wurde eine while-Funktion implementiert, die mit der Funktion wait(NULL); auf das Ankommen einer Nachricht wartet.

## 5 Sockets

Sockets sind ebenfalls ein Weg der Interprozesskommunikation, wobei diese die Einzige ist, die auch über Lokale Grenzen hinaus geht.
Bei Sockets hat man verschieden Implementierungsmöglichkeiten, da wären z.B. Die TCP/IP Methode oder mit UDP-Sockets, beide funktionieren lokal oder mit einem Server, zudem gibt es noch eine Methode die über den Befehl socketpair(); ausgeführt wird. Wir haben uns für socketpair entschieden, da dieser Befehl sehr gut geeignet ist, für die Interprozesskommunikation. Durch das Protokoll laufen die Paare aber auch auf Basis von TCP.

Vorteile: Es ist leicht verständlich, es muss manuell kein Port angegeben werden.

Nachteil: socketpair ist eine Funktion, die nur auf Unix-basierten Betriebssystemen läuft.

```
if (socketpair(AF_UNIX,SOCK_STREAM,0, sv1)==-1){
        perror("socketpair 1 ");
        exit(1);
    }
```

Bei Ausführung von Socketpair(); werden zwei miteinander verbundene, unbenannte Sockets erstellt. AF_UNIX ist in dem Fall die Domain und ermöglicht die lokale Kommunikation, SOCK_STREAM ist der type und bietet sequenzierte, zuverlässige in beide Richtungen verwendbare Byte-Streams an, welche dann hier im Beispiel in sv1 gespeichert werden.
Die 0 legt ein Protokoll fest, dass sog. Default Protokoll dieses verwendet die Werte und Spezifikationen des types und der Domain.

Wenn das Erstellen funktioniert hat, wird der Wert 0 zurückgegeben, falls -1 zurückgegeben wird ist ein Fehler aufgetreten und es würde dann ein Error

erfolgen. Den Code, wie im Screenshot gezeigt, wiederholt man dann viermal, da 4 socket paare gebildet werden müssen.

```
close(sv1[0]);
close(sv2[0]);
close(sv3[0]);
close(sv3[1]);
close(sv4[0]);
close(sv4[1]);
```

Ähnlich wie bei den Pipes muss man alle NICHT benutzen Socket-Deskriptoren schließen, um Probleme und weitere Unannehmlichkeiten zu verhindern.

```
write(sv1[1],&buf, 1);
write(sv2[1],&buf, 1);
```

Um die Zahl von einem zum anderen Prozess zu schicken, muss man write(); öffnen, danach weist man einen Socket-Deskriptor zu.
In den Eckenklammern muss man dann einen Wert entweder 0 oder 1 eintragen, jedoch anders als bei den Pipes ist die 0 nicht nur zum Lesen und die 1 zum Schreiben der Informationen da, sondern ein Prozess empfängt und sendet nur mit einer Zahl an den anderen.
Nachdem man den Deskriptor geöffnet hat, wird die Variable, die die Zahl beinhaltet, in die Klammer geschrieben und zuletzt die Größe.

```
read(sv3[0], &SummeAusS, 1);
```

Um die Zahl dann auslesen zu können benutzt man den Befehl read();.
Letztendlich ist es der gleiche Aufbau wie bei write();, nur das man hierbei jetzt eine Variable angibt in welche die Zahl gespeichert werden soll und nun könnte man z.B. ausgeben.

## 6 Shared Memory mit Semaphoren

Über Shared Memory können Prozesse auf ein gemeinsames Speichersegment zugreifen und Daten dort speichern oder lesen. Mit den Semaphoren können diese Prozesse gesteuert werden, also wann welcher abläuft.

```
int ConvLogKey = 12345893;
int ConvLog_shmget;
int ConvLog_shmdt;
int ConvLog_sprintf;
int ConvLog_printf;
int ConvLog_shmctl;
char *ConvLogpointer;


ConvLog_shmget = shmget(ConvLogKey,
                        MAXMEMSIZE,
                        IPC_CREAT | 0600);


// Gemeinsames Speichersegment anhängen
ConvLogpointer = shmat(ConvLog_shmget, 0, 0);
```

In diesem Bild wird dargestellt, wie ein gemeinsames Speichersegment erzeugt und angehängt wird. Ersteres passiert durch den Befehl shmget(); . Diesem werden der Key, die maximale Größe, ein Kommando, welches in diesem Falle IPC_Create ist, da dieses ein neues Segment erzeugt und die Zugriffsrechte übergeben. Mit 0600 dürfen die Prozesse schreiben und lesen. Zweiteres passiert durch den Befehl shmat(); , wobei

diesem die ID des Segments übergeben werden und dieser sie einem Pointer zuweist, durch welchen auf die Daten zugegriffen werden kann.
Der Vorgang des Erzeugens und Anhängens wird viermal ausgeführt, also einmal jeweils zwischen Conv und Log, und Conv und Stat zum Übergeben der Zufallszahl und zweimal zwischen Stat und Report für die Summe und den Mittelwert.

```
ConvStat_sprintf = sprintf(ConvStatpointer, "%d", zufallszahl);
```

Durch den hier dargestellten Befehl kann die Zufallszahl in dem Shared Memory Segment gespeichert werden.

```
int zahl = atoi(ConvStatpointer);
```

Mit dem Pointer kann aus dem Segment gelesen werden, da dieser jedoch ein Character (char) ist, wird durch atoi die Zahl zu einem Integer und kann zum Berechnen der Summe und des Mittelwerts benutzt werden. Ansonsten kann sie ohne Umwandlung direkt in das File eingelesen oder in der Konsole ausgegeben werden.

```
int ConvLog_key1 = 123450234;
int ConvLog_key2 = 432134611;
int ConvLog_semget1, ConvLog_semget2, ConvLog_semctl;
int output;

setbuf(stdout, NULL); // Das Puffern Standardausgabe (stdout) unterbinden 15

ConvLog_semget1 = semget(ConvLog_key1, 1, IPC_CREAT | 0600);

ConvLog_semget2 = semget(ConvLog_key2, 1, IPC_CREAT | 0600);
```

Diese Abbildung zeigt, wie ein Semaphor beziehungsweise ein Semaphore Paar erzeugt wird. Dies geht durch das Zuweisen eines Keys, dem Kommando IPC_CREAT und dem Zuweisen der Zugriffsrechte. Ein Paar ist immer zwischen zwei Prozessen. Da es vier Prozesse gibt, gibt es drei Semaphore Paare. Die beiden Semaphore haben abwechselnd den Wert 0 und den Wert 1. Ist der Wert auf 0, muss der Prozess warten bis der anderen diesen auf 1 setzt.

```
// Erste Semaphore der Semaphorgruppe 12345 initial auf Wert 1 setzen
ConvLog_semctl = semctl(ConvLog_semget1, 0, SETVAL, 1);

// Erste Semaphore der Semaphorgruppe 54321 initial auf Wert 0 setzen
ConvLog_semctl = semctl(ConvLog_semget2, 0, SETVAL, 0);
```

Damit dies funktioniert, wird zu Beginn des Programmes der Wert der ersten Semaphore auf 1 gesetzt und der, der anderen auf 0.

Im Anschluss werden die P- und V-Operation definiert. Diese Prüfen den Wert der Semaphore und entweder

```
// P-Operation definieren. Wert der Semaphore um eins dekrementieren
struct sembuf p_operation = {0, -1, 0};

// V-Operation definieren. Wert der Semaphore um eins inkrementieren
struct sembuf v_operation = {0, 1, 0};
```

inkrementieren oder dekrementieren diesen. Die P-Operation steht dabei immer am Anfang des Prozesses, da diese den Prozess nur dann starten lässt, wenn der Wert größer als 0 ist und dekrementiert ihn darauf hin. Am Ende des Prozesses wird dann der Wert der anderen Semaphore inkrementiert, sodass dieser Prozess starten kann und alles wieder von vorne beginnt.

```
semop(ConvLog_semget1, &p_operation, 1);
```

```
semop(ConvLog_semget2, &v_operation, 1);
```

## Fazit

Alles in allem kann gesagt werden, dass Shared Memory am schwierigsten und aufwendigsten zu implementieren war. Im Gegensatz dazu war Sockets sehr modern und einfach zu verstehen. Insgesamt ähneln sich Sockets, Pipes und Message Queues, da sie sich selbst synchronisieren und nicht noch zum Beispiel Semaphore implementiert werden muss.

## Literaturverzeichnis


https://stackoverflow.com/questions/62174130/semaphore-cleanup-in-linux-c
https://stackoverflow.com/questions/24906080/how-to-delete-shared-memory-segment-after-some-program-failed-to-detach-from-it
https://www.delftstack.com/de/howto/c/sigint-in-c/
https://man7.org/linux/man-pages/man1/ipcrm.1.html
https://www.youtube.com/watch?v=cex9XrZCU14&list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3kZUY
https://man7.org/linux/man-pages/man2/socketpair.2.html