Seminararbeit

Concurrent C Programming

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering

Autor: Simon Marcin

Datum: 17.06.2014

Dozent: Nico Schottelius

Schule: ZHAW Zürich

1. Einleitung

1.1 Ausgangslage

Im Rahmen des Kurses „Betriebssysteme“ wird eine Seminararbeit erwartet. Die Aufgabenstellung zur Seminararbeit wurde im Kickoff Meeting zusammen mit dem Dozenten besprochen. Das Wissen aus dem Kurs „Betriebssysteme“ soll in dieser Seminararbeit praktisch dargelegt werden. Das Programm muss in C programmiert werden und sollte „forks“ sowie „shm - shared memory“ verwenden. Das Protokoll für den ausgewählten File Server wurde vom Dozenten bereitgestellt.[1]

1.2 Zielsetzung & Abgrenzung

Das Ziel dieser Arbeit ist es, die Kenntnisse der C Programmierung zu festigen und die Theorie über die parallele Abarbeitung aus dem Unterricht anzuwenden. Das fertige Programm muss verschiedene Anforderungen, sowie das vordefinierte Protokoll erfüllen. Der File Server darf nur im Memory laufen, es darf nicht auf das Filesystem zurückgegriffen werden. Da parallel in einem geteilten Memory Bereich gearbeitet wird, müssen die Abfragen koordiniert und kontrolliert werden.

1.3 Vorgehen

Es wurden zunächst die Anforderungen aufgenommen und interpretiert. Anhand dieser Informationen wurde ein grobes Design entworfen, welches dann implementiert wurde. Schlussendlich wurde das Programm getestet und ein Fazit gezogen.

[1] https://github.com/telmich/zhaw_seminar_concurrent_c_programming/blob/master/protokoll/fileserver

2. Anleitung

Dieses Kapitel beschreibt den Umgang mit dem erarbeiteten Programm.

Kompilieren

Um das Tool ausführen zu können, muss es zunächst kompiliert werden. Anhand den Anforderungen geschieht dies mittels make, welches das Binary „run“ erstellt.

```
$ make
```

Ausführen

Das „run“ Binary kann nun ausgeführt werden. Dies startet den File Server. Dieser zeigt seinen jeweiligen Status immer in der Konsole an. Die jeweiligen Operationen werden mit der Prozess-ID ausgegeben.

```
$ ./run
[8392] - START:  Fileserver is ready.
```

Automatisierte Tests

Die automatisierten Tests können durch die Ausführung des „test“ Scripts durchgeführt werden. Dies ist ein Bash Script, welches mittels echo und netcat Verbindungen zum Server aufbaut und verschiedene Abfragen durchführt.

```
$ ./test
------ Create file1: ------
FILECREATED

_______________________________
------ Create file2: ------
FILECREATED

_______________________________
...
```

Manuelle Tests

Manuelle Test werden ebenfalls mittels echo und netcat entsprechende dem Protokoll ausgeführt. Folgende Operationen sind möglich.

- LIST\n
- CREATE FILENAME LENGTH\n
  CONTENT
- READ FILENAME\n
- UPDATE FILENAME LENGTH\n
  CONTENT
- DELETE FILENAME\n

Darauf können beliebige Kombinationen erstellt werden. Damit die verwendeten Semaphoren getestet werden können, müssen die Tests überschneidend parallel ausgeführt werden. Anhand der Prozess-ID kann nachverfolgt werden, wie die geschützten Operationen nur nacheinander und nicht miteinander ausgeführt werden.

Nachfolgende Beispiele zeigen die manuellen Testaufrufe auf.

```
$ echo -e "CREATE file1 5\ntest" | netcat $IP $PORT 2>&1
FILECREATED

$ echo "LIST" | netcat $IP $PORT 2>&1
ACK 1
file1

$ echo "READ file1" | netcat $IP $PORT 2>&1
FILECONTENT file1 5
Test

$ echo -e "UPDATE file1 13\nUPDATED:test" | netcat $IP $PORT 2>&1
UPDATED

$ echo -e "DELETE file1" | netcat $IP $PORT 2>&1
DELETED
```

Anpassungen

Es wurden Konstanten für die Anzahl Dateien, die Länge des Dateinamens, die Länge des Dateiinhaltes und des verwendeten Ports erstellt. Diese können in der Datei „server.c“ falls nötig angepasst werden. Danach muss das Programm neu kompiliert werden.

```
1  #define NUMBER_OF_FILES 10
2  #define MAX_FILE_LENGHT 20
3  #define MAX_CONTENT_LENGHT 256
4  #define PORT 8080
```

Die automatischen Tests laufen Standardmässig lokal. Falls die IP Adresse oder der Port geändert werden muss, so kann dies in der „test“ Datei vorgenommen werden.

```
1  PORT="8080"
2  IP="127.0.0.1"
```

3. Umsetzung

3.1 Weg & Lösungen

Zunächst wurde anhand der Anforderungen ein Grobdesign entworfen. Dieses Design sieht folgende Dateistruktur im Memory vor. Die angegeben Werte in eckigen Klammern sind die dafür reservieren Speicherblöcke oder die maximale Anzahl der Dateien.

File Name [MAX_FILE_LENGHT]	**File Inhalt [MAX_CONTENT_LENGHT]**	**File Länge [Integer]**	**Inhalt Länge [Integer]**
File1	Test-Inhalt	5	12
File2	Bla	5	4
…			
File[NUMBER_OF_FILES]			

Aus dieser Definition entstand folgende Struktur.

```
1  struct file_list{
2   char file_name[NUMBER_OF_FILES][MAX_FILE_LENGHT];
3   char file_content[NUMBER_OF_FILES][MAX_CONTENT_LENGHT];
4   int file_lenght[NUMBER_OF_FILES];
5   int content_lenght[NUMBER_OF_FILES];
6  };
```

Die parallele Verarbeitung wurde nach einem klassischen Server/Client Design implementiert. Pro Client-Verbindung wird der laufende Serverprozess mittels fork() geklont. Dieser Child-Prozess übernimmt dann die Abarbeitung der gesendeten Daten und antwortet auch dem Client.

Damit die verschiedenen Client Prozesse an den gleichen Daten arbeiten können, wird die oben erwähnte Datenstruktur in ein shared memory gelegt. Der nachfolgende Codeausschnitt zeigt dies.

```
1  //Create shared memory
2   shmid = shmget(mykey, sizeof(struct file_list), 0666 | IPC_CREAT);
3   if (shmid == -1) {
4    fprintf(stderr, "[%d] - ERROR:  shmget failed.\n",getpid());
5    exit(EXIT_FAILURE);
6   }
7   shared_memory = shmat(shmid, (void *)0, 0);
8   if (shared_memory == (void *)-1) {
9    fprintf(stderr, "[%d] - ERROR:  shmat failed.\n",getpid());
10   exit(EXIT_FAILURE);
11  }
12  //Put struct in shared memory an memset all entries to '\0'
13  shared_list = (struct file_list *)shared_memory;
14  for(x = 0; x < NUMBER_OF_FILES; x ++) {
15   memset(&shared_list->file_name[x][0], '\0', sizeof(shared_list->file_name[x]));
16   memset(&shared_list->file_content[x][0], '\0', sizeof(shared_list->file_content[x]));
17  }
```

Nun wurden alle im Protokoll definierten Operationen (siehe Kapitel 2) umgesetzt. Der Fileserver war zu diesem Zeitpunkt einsetzbar. Alle Operationen sowie die Rückgabe an den Client wurden in eigenen Funktionen implementiert.

Damit der File Server auch bei mehreren Clientanfragen gleichzeitig funktioniert, mussten die geteilten Ressourcen geschützt werden. Ansonsten konnten sich schreibenden Operationen in die Quere kommen und so zu ziemlich unerwarteten Ausgaben führen. Die Absicherung wurde mittels Semaphoren erstellt. Es wurde ein Semaphoren-Set erstellt, welches genau für jede Datei eine Semaphore bereitstellt. Die Zuordnung zwischen Datei und Semaphore geschieht anhand des Indexes innerhalb der Dateistruktur. Diese Beziehung ist nachfolgend aufgezeichnet.

Semaphoren	**File Name**	**File Inhalt**	**File Länge**	**Inhalt Länge**
0	File1 [Index 0]			
1	File2 [Index 1]			
2	File2 [Index 2]			
…				

Jedoch konnten trotz dieser Sperrung auf Dateiebene weiterhin nicht gewollte Resultate mit der Create-Operation erstellt werden. So konnten zum Beispiel Dateien mit demselben Namen erstellt werden. Auf die zweite Datei konnte dann nicht mehr zugegriffen werden, erst wieder als die erste gelöscht wurde. Um dies zu verhindern wurde eine weitere Semaphore erstellt. Diese schützt den gesamten Create-Prozess. Der nachfolgende Codeausschnitt zeigt die Erstellung der beiden Semaphoren.

```
1  //Create Semaphore
2   sem_id_create = semget(IPC_PRIVATE,1,IPC_CREAT | 0660);
3   if (sem_id_create == -1) {
4     fprintf(stderr, "[%d] - ERROR:  Create Semaphore failed.\n",getpid());
5     exit(EXIT_FAILURE);
6   }
7   if (sem_Init(sem_id_create, 1, 1) == -1) {
8     fprintf(stderr, "[%d] - ERROR:  Set Create Semaphore value to 1 failed.\n",getpid());
9     exit(EXIT_FAILURE);
10  }
11
12  //Files Semaphore
13  sem_id_files = semget(IPC_PRIVATE,NUMBER_OF_FILES,IPC_CREAT | 0660);
14  if (sem_id_files == -1) {
15    fprintf(stderr, "[%d] - ERROR:  Files Semaphore.\n",getpid());
16    exit(EXIT_FAILURE);
17  }
18  if (sem_Init(sem_id_files, 1, NUMBER_OF_FILES) == -1) {
19    fprintf(stderr, "[%d] - ERROR:  Set Files Semaphore value to 1 failed.\n",getpid());
20    exit(EXIT_FAILURE);
21  }
```

Um mit den Semaphoren zu kommunizieren wurden folgende drei Funktionen erstellt.

```
1  int sem_Init(int sem_id, int val, int count);
2  int sem_P(int sem_id, int count);
3  int sem_V(int sem_id, int count);
```

Sem_Init: Initialisieren der Semaphoren
sem_P: Sperren der gewünschten Semaphore anhand der ID und dem Index (count).
sem_V: Freigeben der gewünschten Semaphore

3.2 Probleme

Erfahrung in der C Programmierung

Dies war eines der grössten Probleme des Autors. Die fehlende Erfahrung kostete viel Zeit bei der Implementierung. Triviale Operationen und Funktionen funktionierten Anfangs nie auf Anhieb. Eines der schlimmsten Ereignisse war der „Segmentation Fault“. Diese Fehlermeldung kam immer wieder auf, weil mit Pointern falsch umgegangen wurde und brauchte viel Zeit bis das Problem gelöst war.

Cleanup

Da die Semaphoren und das shared Memory auch nach dem Schliessen des Programms noch vorhanden sind, wurde das Memory der virtuellen Maschine immer mehr aufgebraucht. Mittels „ipcs“ wurden dann sehr viele alte Semaphoren und shared Memory Einträge gesehen. Dies wurde mit einer einfachen Cleanup Funktion gelöst, welche beim „Beenden“ des Programmes mit CTRL&C aufgerufen wird. Ebenfalls wurde so der Socket auf dem Server korrekt abgebaut und es musste nicht gewartet werden, bis das Betriebssystem dies nachträglich aufgeräumt hat.

```
1  //Cleanup on CTRL+C
2   signal(SIGINT, cleanup);
3
4  void cleanup(){
5
6   //Delete Semaphores
7   semctl(sem_id_create,0,IPC_RMID,0);
8   semctl(sem_id_files,NUMBER_OF_FILES,IPC_RMID,0);
9
10  //Delete shared memory
11  shmctl(shmid,IPC_RMID,NULL);
12
13  //Byebye
14  fprintf(stderr, "[%d] - CLEAN: Byebye.\n",getpid());
15
16  //Close socket
17  close(sockfd);
18
19  exit(0);
20
21 }
```

Parallele Abarbeitung

Die Problematiken mit der parallelen Abarbeitung waren ziemlich klein, respektive es verhielt sich so wie gedacht. Da bereits durch den theoretischen Unterricht viel Wissen rund um dieses thema vermittelt und die meisten Fehler angesprochen wurden, gab es nicht viele Fehlerfälle. Dazu beigetragen haben auch die vielen Beispiele aus dem Internet, welche das Verständnis weiter förderten.

4. Fazit

Der Aufwand für diese Arbeit wurde ein wenig unterschätzt, dies hat den einfachen Grund, dass die Erfahrung in der C Programmierung fehlte. Es wurde sehr viel Zeit für triviale Dinge verbraucht und die C spezifischen Eigenheiten mussten zuerst „schmerzhaft“ erfahren werden

Der Aufwand hat sich jedoch gelohnt. Das Verständnis der hardware-nahen Programmierung wurde massiv verbessert und das erarbeitete Ergebnis ist zufriedenstellend und arbeitet wie vorgegeben nach dem Protokoll. Die theoretisch gelernten Grundlagen für parallele Ausführungen konnten gut in die Praxis umgesetzt werden. Die einzelnen Prozesse arbeiten im gleichen Memory Teil und schützen sich voreinander mittels Semaphoren.