Prof. Dr. M. Jäger
FB MNI

21. Juni 2014

Hausübung 2 - Betriebssysteme I

Bearbeitungszeit: Upload auf saturn.mni.thm.de bis 17.7.2014.

Präsentation der Lösung: In den Übungsstunden bis 17.7.2014

Aufgabenstellung

Implementieren Sie in C (oder C++) den Zugriff auf eine Datei als verteilte Anwendung mit Client-Server-Architektur:

- Der Server verwaltet eine(!) Datei, die in seinem lokalen Dateisystem gespeichert ist. Er führt im Auftrag seiner Clients Lese- und Schreiboperationen auf der Datei durch. Außerdem hat ein Client die Möglichkeit, die Datei für konkurrierende Zugriffe zu sperren. Der Server merkt sich, welcher Client die Datei sperrt.
- Clients sind beliebige Prozesse auf beliebigen Rechnern im Netz, die via Nachrichtenaustausch mit dem Server konkurrierend auf die Datei zugreifen. Der Testclient ist ein Kommandointerpretierer, der Dateikommandos des Benutzers in Aufträge für den Server transformiert und die Serverantworten dem Benutzer anzeigt.

Kommunikation

Der Nachrichtenaustausch basiert auf einem von Ihnen zu implementierenden Netzwerkprotokoll der Anwendungsschicht. Dieses Protokoll entspricht dem Request/Response-Muster und ist weiter unten spezifiziert. Als Transportstack wird TCP üver IPv4 verwendet.

Jeder Client hat eine eindeutige Client-Identifikation CID (vom Typ `cid_t`) aus zwei ganzen Zahlen:

- IPv4-Adresse (Nummer) der Netzwerkschnittstelle des Client-Rechners (4 Byte)
- Prozess-Identifikation des Client-Prozesses (2 Byte)

Diese CID wird bei jeder Auftragsnachricht an den Server mitgeschickt. Der Server nutzt die CID, um den Besitzer einer Dateisperre zu identifizieren.

Der Server

Mehrere Client-Aufträge werden nebenläufig bearbeitet: Nach Empfang einer Auftragsnachricht wird die Bearbeitung an einen neuen Server-Subprozess delegiert. Die Kommunikation soll mit Stream-Sockets basierend auf IPv4 und TCP implementiert werden. Auf der Serverseite benötigen Sie die Socket-Funktionen *socket*, *bind*, *listen*, und *accept*. Für den Nachrichtaustausch können Sie *send* und *recv* verwenden.

Der Server soll alle seine Netzschnittstellen bedienen. Die TCP-Portnummer des Servers wird dem Client und dem Server selbst über die Umgebungsvariable SERVERPORT mitgeteilt. Die Internetadresse des Servers wird dem Client über die Umgebungsvariable FILESERVER mitgeteilt und kann als Nummer (z.B. 127.0.0.1) oder Name (z.B. saturn.mni.thm.de) angegeben werden.

Die Dateioperationen des Servers

Der Server implementiert für seine Datei mehrere Dienste gemäß folgeder API:

```
int file_reset (cid_t cid);
          // alle Daten loeschen
          //            Resultat: 0 bei Erfolg, sonst -1
int file_write (cid_t cid, unsigned distanz, unsigned anzahl, char *daten);
          // schreiben, distanz ist Byte-Distanz zum Dateianfang
          //            Resultat: 0 bei Erfolg, sonst -1
int file_read  (cid_t cid, unsigned distanz, unsigned anzahl, char *daten);
          // lesen,     distanz ist Byte-Distanz zum Dateianfang
          //            Resultat: Anzahl gelesener Bytes bei Erfolg, sonst -1
int file_lock  (cid_t cid);
          // ggf. blockierendes sperren, Resultat: 0 bei Erfolg, sonst -1
int file_trylock  (cid_t cid);
          // nicht blockierendes sperren, Resultat: 0 bei Erfolg, sonst -1
int file_unlock(cid_t cid);
          // entsperren, Resultat: 0 bei Erfolg, sonst -1
int file_locked(cid_t *cid);
          // Sperre pruefen,
          // Resultat: 0, falls keine Sperre, sonst -1
          // falls eine Sperre existiert, enthält cid die ID des sperrenden Client
```

Die Datei-API ist in Form eines C-Moduls mit den Dateien `file.c` und `file.h` zu implementieren.

Dateisperren

Jeder Client kann eine Dateisperre setzen, die ihm solange exklusiven Dateizugriff ermöglicht, bis er die Sperre mit *file_unlock* wieder aufhebt. Zum Setzen der Sperre gibt es eine nicht blockierende Operation *file_trylock* und eine ggf. blockierende Operation *file_lock*. Wenn die Datei nicht gesperrt ist, setzt der aufrufende Client bei beiden Operationen die Sperre. Wenn die Datei durch einen anderen Client gesperrt ist, kehrt *file_trylock* sofort erfolglos zurück, während *file_lock* solange blockiert, bis die Dateisperre erfolgreich gesetzt werden kann.

Versucht ein Benutzer auf eine von einem anderen gesperrte Datei zuzugreifen, liefert die Zugriffsfunktion -1 als Ergebnis. Ausnahme: die Abfrage der Sperre mit *file_locked* ist für alle immer möglich. Die CID (*cid*-Argument in den Operationen) dient zum Überprüfen der Zugriffsberechtigung.

Das Client-Server Kommunikationsprotokoll

Der Server gewährt externen Auftraggeber-Prozessen (Clients) Zugriff auf die von ihm verwaltete Datei. Er benutzt dazu ein **Netzwerkprotokoll**. Das Protokoll definiert eine Reihe von Nachrichtentypen und den Ablauf der Kommunikation.

Der Ablauf entspricht dem Request/Response-Muster: Der Server initialisiert einen Kommunikationsendpunkt zur Entgegennahme von Auftragsnachrichten und wartet auf Aufträge. Ein Client schickt eine Auftragsnachricht, der Server empfängt diese und schickt eine Antwortnachricht. Der Client empfängt die Antwortnachricht.

Jede Nachricht eines Client steht für einen Dateizugriff. Der Server analysiert die Nachricht und setzt sie dann um in einen entsprechenden Aufruf seiner internen Dateischnittstelle. Er schickt das

Ergebnis des Dateizugriffs als Antwort-Nachricht an den Auftraggeber.

Die Nachrichtentypen sind in der nachfolgenden Tabelle definiert.

Typ	Absender	Semantik	Format
w	Client	Write-Operation	'w' CID OFFSET NBYTES DATA
r	Client	Read-Operation	'r' CID OFFSET NBYTES
d	Server	Daten als Read-Antwort	'd' NBYTES DATA
l	Client	Lock-Aufruf	'l' CID
t	Client	Trylock-Aufruf	't' CID
u	Client	Unlock-Aufruf	'u' CID
?	Client	Locked-Aufruf	'?' CID
!	Server	Locked-Antwort	'!' CID (oder 8 Null-Bytes)
o	Server	Operation erfolgreich	'o'
L	Server	Fehler: Datei gesperrt	'L'
e	Server	sonstiger Fehler	'e'

Das Nachrichtenformat in der Tabelle beschreibt die Elemente, aus denen die Nachricht besteht. Unter den Elementen sind binär codierte Zahlen. Die Zahlen sollen in Netzwerk-Bytereihenfolge (Big Endian) codiert werden. Der Typ ist ein Byte in ASCII-Codierung, CID besteht aus der Prozessnummer des Client (2 Byte), und der IP-Nummer des vom Client benutzten Netzwerkinterfaces (4 Byte). OFFSET ist 4 Byte lang und NBYTES 2 Byte. DATA ist eine beliebige Bytefolge.

Eine 'o'-Nachricht („O.K.“) wird vom Server bei erfolgreich ausgeführter *file_write*-, *file_lock*- , *file_trylock*- oder *file_unlock*-Operation zurückgesendet, bei erfolgreicher *file_read*-Operation wird nur eine 'd'-Nachricht gesendet.

Die Clients

Es gibt beliebig viele Client-Prozesse im System. Diese kommunizieren mit dem Server, um die Datei zu lesen bzw. zu modifizieren.

Zum Test der Serverfunktion ist ein Programm *client* zu implementieren, das in einer Interaktionsschleife Dateibearbeitungs-Kommandos von der Standardeingabe einliest, diese als Aufträge an den Server weiterleitet und die Antworten des Servers anzeigt. Als Daten werden dabei nur Zeichenketten aus lesbaren Zeichen verwendet.

Client-Kommandos, Parameter	Dateioperation
C	Reset-Aufruf
W *Offset String*	Write(ID, Offset, Länge(String), String)
R *Offset NBytes*	Daten, die Read(ID, *Offset*, *NBytes*) liefert, ausgeben
L	Lock-Aufruf
T	trylock-Aufruf
U	Unlock-Aufruf
?	Locked-Aufruf – Sperr-Information ausgeben
Q	Quit - Client-Programm beenden

Eine Beispiel-Sitzung:

```
$ client
> C
o.k.
> W 0 "HALLI "
o.k.
> W 7 "HALLO"
o.k.
> R 3 6
LI HALLO
> ?
0
> Q
bye
$
```

Wenn eine Operation nicht erfolgreich ist, soll das Client-Programm eine Fehlermeldung ausgeben.

Makefile

Für den Erzeugungsprozess von Client und Server ist ein Makefile zu erstellen, das die Compiler- und Linkeraufrufe steuert. Als Ziele sind zu implementieren: *client*, *server*, *clean*, *dist-clean* und *all*.

Upload

Die Lösung ist genau wie bei Hausübung 1 in ein Tar-Archiv zu verpacken und auf den Rechner saturn.mni.thm.de in das Verzeichnis `/home/hg52/bs/ha2/moss` hochzuladen.