Systemprogrammierung 3 (sysprog-3)

Einleitung

Diese Aufgabe ist ähnlich zu den Aufgaben aus `sysprog-1` bzw. `sysprog-2` und soll den richtigen Umgang mit Syscalls vermitteln — mit dem Ziel, ein einfaches Kommunikationsprotokoll für Client und Server zu implementieren. Für nähere/einführende Details zu Syscalls kann die Aufgabe `sysprog-1` verwendet werden.

Protokoll

Für das Protokoll wird Ihnen hier eine Header-Datei zur Verfügung gestellt. Außerdem können Sie mit diesem Wireshark-Dissector aufgezeichneten Netzwerkverkehr von Server und Client (in Form einer `pcap`-Datei, z.B. mit `tcpdump -w capture.pcap` gefolgt von `CTRL + C` erstellt) und darin enthaltene relevante Protokoll-Felder analysieren. Einen Beispiel-Dump für das Protokoll finden Sie hier.

Das Protokoll soll auf IPv4 (*network layer*) und dem verbindungsbasierten TCP (*transport layer*) aufbauen, unter Verwendung des Ports `1234`.

Die versendeten Daten bestehen bei Protokollnachrichten immer aus Bytes des `struct message_unit` gefolgt von einem optionalen `struct challenge_unit`. Wie die Structs definiert sind, und welche Bedeutungen die jeweiligen Felder haben, können Sie in proto.h nachlesen. Da alle Member 32-Bit-Typen sind, sollte Padding kein Problem darstellen.

Die Kommunikation erfolgt in drei Schritten:

- Nach dem Verbindungsaufbau sendet der Server ein "Challenge-Paket" an den verbundenen Client. Dabei muss `msg_type` in `message_unit` dem definierten Wert `CHALLENGE` entsprechen — `server_info` wird in der Nachricht nicht beachtet, und soll auf `0` gesetzt sein. Die Werte in `challenge_unit` geben nun die vom Client zu lösende "Challenge" an. Es wird eine Berechnung gefordert, deren Ergebnis vom Server zuerst auf `0` gesetzt wird — abhängig vom `op`-Wert in `challenge_unit` und angewendet auf die Operanden `lhs` bzw. `rhs`.
- Der Client empfängt das "Challenge-Paket" des Servers, führt die geforderte mathematische Operation durch (beachten Sie: für `unsigned`-Zahlen, welche wir verwenden, sind alle geforderten Operationen *mit Ausnahme der Shift-Operationen* wohldefiniert! Überläufe in der Theorie sind dabei also egal), und schreibt die Antwort in das `answer`-Feld — alle restlichen Felder müssen unberührt bleiben. Für die Shift-Operationen (`LEFT_SHIFT` und `RIGHT_SHIFT`) muss `rhs` *modulo der Anzahl an Bits* in `lhs` genommen werden, um laut C- bzw. C++-Standards kein undefiniertes Verhalten zu verursachen. Das architekturspezifische Verhalten von `lhs <shift_op> rhs` auf dem Abgabe-Server würde genau `lhs <shift_op> (rhs % (CHAR_BIT * sizeof(lhs)))`, also `lhs <shift_op> (rhs % 32)` entsprechen, und wird auch so erwartet. Es wird also eine nahezu identische Nachricht an den Server zurückgeschickt, mit dem einzigen Unterschied im `answer`-Feld.
- Der Server empfängt das "Lösungs-Paket" des Clients, überprüft die Lösung, und sendet eine Antwortnachricht an den Client, welche **nur** aus einer `message_unit` besteht, nun mit Member `msg_type` als `SERVER_INFO` und Member `server_info` dem Ergebnis entsprechend (`CORRECT` oder `WRONG`).

Damit ist die Kommunikation wieder zuende, und die Verbindung soll von beiden Seiten wieder geschlossen werden. Schickt der Client ein falsches Ergebnis (mit Antwort `WRONG`), so hat er sich nicht protokollgemäß verhalten.

Der Client muss sein Ergebnis innerhalb von `50 ms` nach Erhalt der Challenge an den Server übermittelt haben, um überhaupt von einer Antwort ausgehen zu können.

Relevante Manpages zu Sockets und den relevanten Protokollen aus Sektionen 2 bzw. 7:

- `socket(2)`
- `socket(7)`
- `address_families(7)`
- `ip(7)`
- `tcp(7)`
- `setsockopt(2)`
- `bind(2)`
- `listen(2)`
- `accept(2)`

- `connect(2)`
- `send(2)`
- `recv(2)`
- `shutdown(2)`
- `close(2)`

Abgabe

Im anschließenden Kapitel finden sich 2 Programmieraufgaben. Die Abgabemodalitäten sind identisch zu `sysprog-1` und `sysprog-2`. Das Makefile kann hier heruntergeladen werden. Die verwendete Rust libc-Library wurde neu kompiliert und kann hier heruntergeladen werden.

Client

- Abzugebendes Programm: `client.cpp`
- Abzugebende Antwortdatei (siehe "Fragen"): client.txt
- Erreichbare Punkte: **10**
- Erreichbare Bonuspunkte bei Abgabe in Assembly: **5**

Ziel dieser Aufgabe ist es, durch Socket-Programmierung einen funktionierenden Client zu schreiben, welcher *ohne* Commandline-Argumente versucht, eine Verbindung zu einem Server mit IP:Port `127.0.0.1:1234` laut Protokoll herzustellen, und die Challenge immer korrekt und protokollgemäß löst.

Der Client soll in einer Endlosschleife laufend Challenges auf diese Weise lösen, und nur im Fehlerfall von selbst terminieren (keine Verbindung möglich, Challenge laut Server falsch gelöst, Verbindungsverlust, ...).

Sie sollen hier der Einfachheit halber mit *blocking Syscalls/Sockets* arbeiten. Außerdem soll kein Multithreading zur Anwendung kommen (das geht auch aus der Liste der erlauben Syscalls hervor).

Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **socket**
- **connect**
- **write** (für Logging-/Debug-Output)
- **recv/recvfrom**
- **send/sendto**
- **close**
- **exit/exit_group**
- fstat
- newfstatat
- getrandom
- brk

Fragen

- Wie heißt die C-Konstante *für das Protokoll* aus der **Internet Protocol**-Implementierung (vorkommend im IP-Header), die als Default-Wert beim Erstellen eines Sockets angenommen wird mit `domain`-Parameter `AF_INET`, `type`-Parameter `SOCK_STREAM` und `protocol`-Parameter `0`?

Server

- Abzugebendes Programm: `server.cpp`
- Abzugebende Antwortdatei (siehe "Fragen"): server.txt
- Erreichbare Punkte: **10**
- Erreichbare Bonuspunkte bei Abgabe in Assembly: **5**

Ziel dieser Aufgabe ist es, durch Socket-Programmierung einen Server laut Protokoll zu schreiben, welcher *ohne* Commandline-Argumente auf Port `1234` hört (vgl. `bind(2)` bzw. `listen(2)`). Der Server soll Verbindungen von beliebigen IP-Adressen akzeptieren,

und nacheinander abarbeiten (Endlosschleife).

Für die Bewertung ist es wichtig, dass Ihre Server-Implementierung die Werte für `lhs`, `rhs` und `op` (Achtung, nur Enum-Werte `ADD`, `SUB`, `MUL`, `LEFT_SHIFT`, `RIGHT_SHIFT` erlaubt!) **zufällig** mit einer geeigneten Entropie-Quelle (`getrandom(2)` oder äquivalente `libc`-Funktionen) auswählt!

Achten Sie bei den Syscalls vor allem auf die Return-Werte, um Fehlerfälle wie Verbindungsverlust zu behandlen. Außerdem könnten je nach Verbindungsstatus auch Signale, wie z.B. `SIGPIPE` an den Prozess gesendet werden. Es empfiehlt sich daher, sich die Manpages zu den erlaubten Syscalls genau durchzulesen.

Da Sie auch hier ohne Multithreading und mit *blocking Syscalls* arbeiten sollen, sollten Sie z.B. wie im Protokoll beschrieben ein Timeout für `recv`s setzen. Syscalls wie `select(2)` oder `poll(2)` werden Sie hier also auch nicht benötigen, um die Aufgabenstellung zu erfüllen.

Erlaubte Syscalls

Die hervorgehobenen Syscalls sind üblicherweise für die minimale Funktionalität notwendig. Die übrigen Syscalls können vom Compiler generiert werden und sind erlaubt, aber nicht direkt notwendig.

- **socket**
- **setsockopt**
- **bind**
- **listen**
- **accept**
- **write** (für Logging-/Debug-Output)
- **send/sendto**
- **recv/recvfrom**
- **shutdown**
- **close**
- **getrandom**
- **exit/exit_group**
- fstat
- newfstatat
- brk

Fragen

- Wie heißt die C-Konstante für einen `setsockopt(2)`-Syscall, die es erlaubt, eine lokale Adresse für `bind(2)` wiederzuverwenden, solange kein Socket im `LISTENING`-Zustand an die Adresse gebunden ist?

Beispielimplementierungen für Test-Zwecke

Um Sie bei der Implementierung zu unterstützen, stellen Wir Ihnen hier bereits kompilierte Beispielprogramme zur Verfügung:

- example-client
- example-server

Antwort-Datei Templates

Diese Templates können für die Beantwortung der Fragen verwendet werden:

- client.txt
- server.txt

Punkteverteilung

Insgesamt: **20** Punkte

- `client`: **10** Punkte
- `server`: **10** Punkte

Bonuspunkte

Es gibt Bonuspunkte, wenn die abgegebenen Programme in Assembly geschrieben wurden. Insgesamt sind **10** Bonuspunkte erreichbar.

- `client`: **5** Bonuspunkte
- `server`: **5** Bonuspunkte

Mit normalen Punkten und Bonuspunkten sind somit insgesamt **30** Punkte erreichbar.

Deliverables

- File: **client.cpp**
  Source-Code für das Client-Programm. Für Abgabe C/ASM/Rust die Warnung beim Upload ignorieren.
- File: **client.txt**
  Beantwortung der Text-Fragen für das Client-Programm.
- File: **server.cpp**
  Source-Code für das Server-Programm. Für Abgabe C/ASM/Rust die Warnung beim Upload ignorieren.
- File: **server.txt**
  Beantwortung der Text-Fragen für das Server-Programm.