

2. Programmierertest

23.Jänner 2024

Testumgebung

Im Verzeichnis `~/exam` finden Sie die C-Quelldateien, in welchen Sie Ihre Lösung implementieren sollen. Sie können jederzeit die Original-Dateien mit dem Shellkommando

\$ fetch

wiederherstellen. Änderungen, die Sie an den Dateien vorgenommen haben, werden gesichert (`<Datei>→<Datei>~1~`).

In dem Verzeichnis befindet sich auch ein Makefile. Um Ihre Lösung zu kompilieren, verwenden Sie den Befehl **make**. Testen und debuggen Sie Ihr Programm wie gewohnt. Auch **gdb** steht Ihnen zur Verfügung. Übrig gebliebene Ressourcen (wie Semaphoren oder Shared Memory) können Sie mit dem Kommando **cleanup** entfernen.

Um Ihre Lösung zu prüfen, führen Sie

\$ deliver

aus. Sie können **deliver** beliebig oft ausführen. Dabei wird Ihnen auch angezeigt, wie viele Punkte Sie derzeit auf Ihre Lösung erhalten würden.

Wenn Sie mit der Bewertung zufrieden sind, dann melden Sie sich bei der Aufsicht, die das Ergebnis entgegennimmt und Sie ausloggen wird.

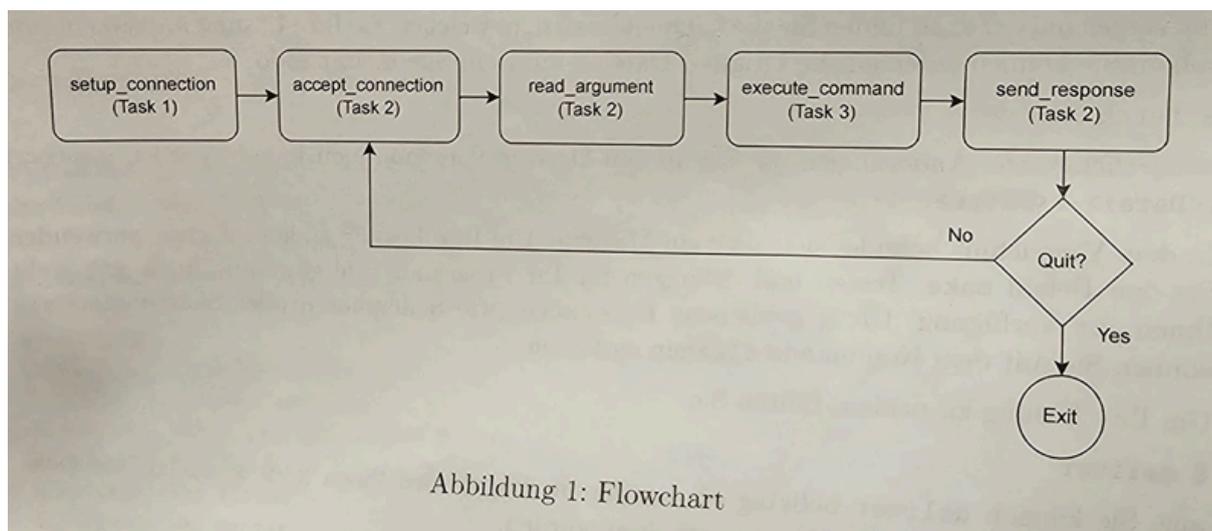
Der Test besteht aus mehreren Aufgaben, die Sie **unabhängig voneinander** und in **beliebiger Reihenfolge** implementieren können. Für jede Aufgabe steht Ihnen eine DEMO-Funktion zur Verfügung, die Sie in der **main()**-Funktion einkommentieren können, wenn Sie eine Aufgabe überspringen möchten.

Beachten Sie folgende Bewertungskriterien:

- Ihr Programm muss ohne Fehler kompilieren, sonst erhalten Sie keine Punkte.
- Compiler Warnings (das Programm wird mit `-Wall` übersetzt) führen zu Punktabzügen.
- Bei Segmentation Faults (Speicherzugriffsverletzung) erhalten Sie für die entsprechende Aufgabe keine Punkte.

Einleitung

Ihre Aufgabe ist es, ein Server-Programm zu schreiben, welches auf Anfrage ein Kommando ausführt. Dazu wird ein passiver Socket angelegt, an dem der Server auf Verbindungen wartet. Kommt eine Verbindung zustande, dann wartet der Server auf eine Anfrage. Die Anfrage enthält ein Argument, das vom Server an ein Kommando übergeben wird, welches lokal ausgeführt wird. Bei Erhalt einer Anfrage erzeugt der Server einen Kindprozess, welcher anschließend das Kommando mit dem erhaltenen Argument ausführt. Danach sendet der Server die Ausgabe des Kommandos an den Client zurück. Abbildung 1 zeigt, wie die zu implementierenden Funktionen in der **main**-Methode in **server.c** aufgerufen werden.



Synopsis

Das Server-Programm **server** kann wie folgt aufgerufen werden (die Argumentbehandlung für den Server ist bereits vorgegeben und muss von Ihnen nicht mehr implementiert werden):

./server [-p PORT] COMMAND

Dabei entspricht die Option **PORT** dem gewünschten Port, an dem der Server auf Verbindungen warten soll, und das positionelle Argument **COMMAND** ist das Kommando, das bei einer Anfrage ausgeführt werden soll.

Bitte beachten Sie die Hinweise zum Testen Ihrer Implementierung auf Seite 5

1. Verbindungsaufbau

Vervollständigen Sie die Funktion **int setup_connection(const char *port_str)** in **server.c** mit dem Verbindungsaufbau.

Diese Funktion bekommt den Port übergeben, auf dem der Server lauschen soll (der Port wird als C-String übergeben, welcher unverändert an die Funktion **getaddrinfo(3)** weitergereicht werden kann).

Erstellen Sie ein passiver Socket der Familie **AF_INET** und vom Typ **SOCK_STREAM** und binden dieses an den übergebenen Port. An diesem Port wird auf Verbindungen von Clients gewartet. Der File Descriptor dieses Sockets wird schließlich von der Funktion zurückgegeben.

Achten Sie darauf, dass Sie auch tatsächlich den File Descriptor des Sockets zurückgeben und nicht den File Descriptor einer Verbindung an diesem Socket (die Funktion **accept(2)** soll erst im nächsten Task aufgerufen werden)!

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code **EXIT_FAILURE** beendet werden. Sie können dazu die Funktion **error_exit(char *msg)** verwenden.

Hinweise:

- Verwenden Sie die Funktion **getaddrinfo(3)** sowie die System-Calls **socket(2)**, **bind(2)** und **listen(2)** und den Server zu konfigurieren.

2. Anfrage Abarbeiten

Vervollständigen Sie folgende Funktionen in **server.c**:

- **FILE* accept_connection(int sockfd)**
- **void read_argument(FILE* socket_fp, char* argument)**
- **void send_response(FILE* socket_fp, FILE* command_fp)**

In der Funktion **accept_connection** warten Sie auf den Verbindungsaufbau eines Clients mit dem zuvor erzeugten Socket, dessen File Descriptor im Argument **sockfd** übergeben wird. Öffnen Sie danach ein FILE-Objekt für diese Verbindung und retournieren Sie die Referenz darauf.

Die Funktion **read_argument** bekommt diese FILE-Referenz und einen String als Argumente übergeben. Lesen Sie die Anfrage des Client (die Argumente für das auszuführende Programm) von **socket_fp** und schreiben Sie diese dann in **argument**. Der Client überträgt seine Anfrage und schließt danach sein Schreib-Ende der Verbindung, lesen Sie daher solange von der Verbindung, bis beim Lesen ein End-of-File (kurz EOF) auftritt. Ihr Programm muss Anfragen mit bis zu **MAX_ARGUMENT_LEN** Zeichen lesen können (siehe **server.h** für die Definition dieses Macros).

Die Anfrage wird anschließend von der Funktion **execute_command()** weiterverarbeitet, welche eine Referenz auf ein FILE-Objekt retourniert, von dem die Ausgabe des ausgeführten Kommandos gelesen werden kann. (siehe Task 3; wenn Sie diese Funktion noch nicht implementiert haben, können Sie zum Testen stattdessen **DEMO_execute_command()** verwenden).

Implementieren Sie nun in der Funktion **send_response** die Übertragung der Ergebnisse an den Client. Die Argumente **socket_fp** und **command_fp** sind die FILE-Referenzen auf die Client-Verbindung sowie die Ausgabe des ausgeführten Befehls. Lesen Sie solange von **command_fp**, bis Sie das Ende der Ausgabe erreichen (bis zum End-of-File, kurz EOF) und übertragen Sie diese Daten auch tatsächlich gesendet werden. Für den Fall, dass **execute_command()** fehlschlägt wird anstelle einer gültigen Referenz der Wert NULL zurückgegeben. Schreiben Sie in diesem Fall den String **COMMAND_FAILED** in die Verbindung. Schließlich werden sowohl die Verbindung als auch das von **execute_command()** zurückgegebene FILE-Objekt geschlossen. Achten Sie aber darauf, dass Sie nicht den Socket des Servers schließen, dieser wird weiter verwendet!

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code **EXIT_FAILURE** beendet werden. Sie können dazu die Funktion **error_exit("my err msg")** verwenden.

Hinweise:

- Beachten Sie, dass ein C-String mit einem 0-Byte terminiert wird und somit ein Byte mehr an Speicher benötigt, als er Zeichen enthält.
- Nützliche Funktionen: **accept(3)**, **fdopen(3)**, **fgetc(3)**, **fgets(3)**, **fputs(3)**, **fflush(3)**, **fclose(3)**

3. Kommando Ausführen

Vervollständigen Sie die Funktion **execute_command** in **server.c**, um darin ein Kommando in einem Kindprozess auszuführen.

Erstellen Sie zunächst eine Pipe, um die Standardausgabe (**stdout**) des Kindprozesses lesen zu können (eine weitere Pipe für **stdin** ist **nicht** erforderlich). Anschließend starten Sie einen Kindprozess (mittels **fork(2)**) und leiten unter Verwendung von **dup2(2)** dessen Standardausgabe auf das Schreib-Ende der zuvor erzeugten Pipe um.

Im Kindprozess starten Sie anschließend das Programm, welches mittels des Arguments **command** spezifiziert wird, und übergeben diesem das Argument **argument** als einziges Argument. Verwenden Sie eine geeignete Funktion aus der **exec(3)**-Familie, um dieses Programm im Kindprozess auszuführen.

Im Elternprozess warten Sie zunächst auf das Beenden des Kindprozesses. Danach wird ein FILE-Objekt für das Lese-Ende der Pipe erzeugt, aus dem in weiterer Folge die Ausgabe des Kindprozesses gelesen werden kann. Dieses FILE-Objekt wird schließlich von der Funktion retourniert. Achtung, das Lesen der Ausgabe selbst erfolgt erst zu einem späteren Zeitpunkt, in dieser Funktion wird nur das FILE-Objekt erzeugt!

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code **EXIT_FAILURE** beendet werden. Sie können dazu die Funktion **error_exit("my err msg")** verwenden. Falls Kindprozess mit einem Fehler terminiert, retournieren Sie NULL.

Hinweise:

- Beachten Sie, dass zusätzlich zu den Argumenten einem Prozess auch immer der Programmname als implizites erstes Argument (mit Index 0) übergeben wird.
- Nützliche Funktionen: **pipe(2)**, **fork(2)**, **dup2(2)**, **exec(3)**, **wait(2)**