Übungsblatt 13

Dr. Matthias Frank, Dr. Matthias Wübbeling

Ausgabe Mittwoch, 17. Januar 2024
Abgabe bis
Freitag, 26. Januar 2024, 23:59 Uhr
Vorführung vom 29. Januar bis zum 2. Februar 2024

Alle Programme müssen unter **Ubuntu 22.04** kompilierbar bzw. lauffähig und ausreichend kommentiert und mit **Makefile** (C, Assembler) versehen sein, um Punkte zu erhalten. Als Compiler sollen **clang** (C) und **nasm** (Assembler) verwendet werden. Die Lösungen sind bei Ihrem Tutor während Ihrer Übungsgruppen vorzuführen. Alle Gruppenmitglieder sollten die Abgabe erklären können. Die Abgabe erfolgt mittels Ihres Git-Repositories und der vorgegebenen Ordnerstruktur.

Die Punkte der Aufgaben sind relevant für die Zulassung. Die Punkte der Bonusaufgaben werden auf Ihren Punktestand addiert, werden aber nicht auf die für die Zulassung benötigten Punkte addiert.

Abgabestruktur: Jede Abgabe, die die folgende Struktur nicht umsetzt, wird **NICHT** bepunktet bzw. mit 0 Punkten bewertet

- die zu korrigierenden Lösungen müssen bis zur Deadline auf dem `master`-Branch liegen. Lösungen auf anderen Branches werden nicht gewertet
- alle Lösungen müssen in der vorgegebenen Ordnerstruktur (**blattXX/aufgabeYY**) abgelegt werden, wobei `XX` und `YY` durch die jeweiligen Nummern des Zettels und der Aufgaben ersetzt werden sollen. Der Name soll exakt nur aus diesen Zeichen bestehen und achtet auf Kleinschreibung
- sämtliche Aufgaben, mit Ausnahme der theoretischen Aufgaben, die eine PDF erfordern, benötigen zwingend ein **Makefile**. Abgaben ohne dieses werden nicht gewertet

Hinweis:	Manche Browser sind nicht dazu in der Lage die in die PDFs eingebetteten Dateien anzuzeigen. Mittels eines geeigneten Readers, wie dem Adobe Reader, lassen sich diese jedoch anzeigen und verwenden.

Hinweis:	Zum Testen der Netzwerkprogramme besteht auch die Möglichkeit sich per ssh auf den Rechnern der Universität einzuwählen

Aufgabe 1. Iterativer TCP-Server (4 Punkte Server + 4 Punkte Client)

In dieser Aufgabe soll in C ein iterativer TCP-Server implementiert werden, der nur eine Verbindung zu einer Zeit bearbeiten kann. Verbindungen werden also nacheinander (iterativ) abgearbeitet. Verbindet sich ein zweiter Client, so wird dessen Verbindung so lange zurückgestellt, bis die erste Verbindung vollständig bearbeitet wurde.

Implementieren Sie sowohl einen Bank-Server als auch einen Bank-Client. Der Server erhält per Kommandozeilenparameter den Port auf dem er horchen soll. Verwenden Sie dabei für die Umwandlung von `argv[1]` von `char*` nach `long` die Funktion `strtol`. Anschließend müssen Sie die Zahl korrekt casten; achten Sie hierbei darauf, dass die Zahl im Wertebereich für Portnummern liegt. Dieser Port soll für TCP-Verbindungsaufbauwünsche bereitstehen.

Der Server verwaltet genau ein Bankkonto, auf das mehrere Clients nacheinander zugreifen können. Das Konto wird durch einen Integer-Wert repräsentiert und existiert nur solange der Server läuft, das heißt der Wert des Kontos muss *nicht* in einer Datei gespeichert werden. Beim Start des Servers wird es mit 0 initialisiert. Wird der Client beendet soll der Server den Verbindungs-Socket schließen und für neue Verbindungen bereit stehen.

Achtung: `accept` erzeugt für die Verbindung einen neuen Socket, der dann auch per `close` geschlossen werden muss. Auf dem ersten Socket kann dann weiter „gehorcht“ werden. Achten Sie bitte auf Host- und Networkbyteorder.

Der Bank-Client bildet das Gegenstück zum Server. Er bekommt über Kommandozeilenparameter die Zieladresse und den Port. Nach dem erfolgreichen Verbindungsaufbau zum Server wird der aktuelle Kontostand ausgelesen und auf dem Bildschirm dargestellt. Der Nutzer des Clients soll folgende Möglichkeiten haben, auf das Konto zuzugreifen: **Geld einzahlen** / **Geld abheben**

Nach einem Zugriff auf das Konto soll der neue Kontostand vom Server gelesen werden.

Wird der Server vorzeitig beendet, soll dies der Client korrekt abfangen (d.h. `read` gibt 0 zurück). Weiterhin soll auch der Server einen Abbruch des Client abfragen können und dann wieder auf weitere Anfragen reagieren. Im Ergebnis sollen dann mehrere Clients verbunden werden und die Anfragen nacheinander (iterativ und nicht parallel) verarbeitet werden. Bitte testen Sie dies.

Aufgabe 2. Iterativer TCP-Server mit mehreren Prozessen/Threads (8 = 4 + 4 Punkte)

a)

Erweitern Sie den Bank-Server so, dass er mehrere Anfragen parallel bearbeiten kann. In dieser Aufgabe soll zu diesem Zweck für jede Anfrage ein neuer Prozess mit Hilfe von `fork` generiert werden.

1. Der Parent-Prozess nimmt die Anfrage entgegen und ruft anschließend `fork` auf (vgl. Kapitel 2 der Vorlesung). Für ihn ist die Bearbeitung damit beendet — er kann weitere Anfragen auf gleiche Weise entgegennehmen.
2. Der neu generierte Child-Prozess bearbeitet die Anfrage wie in der Ein-Prozess-Implementierung. Sobald die Bearbeitung abgeschlossen ist, beendet sich der Prozess selbstständig.
3. Achten Sie darauf, dass durch die Bearbeitung der Anfragen nicht dauerhaft Zombie-Prozesse entstehen. Gleichzeitig soll der Parent-Prozess durch das Cleanup nicht blockiert werden.
4. Der Parent-Prozess sollte die Anzahl der laufenden Kindprozesse mitzählen. Hierfür sind `signal` und `waitpid` hilfreich.

b)

Ändern Sie Ihre Bank-Server-Implementierung so ab, dass für die Bearbeitung jeder Anfrage nicht mehr ein separater Prozess, sondern jeweils ein neuer Thread generiert wird. Der Thread soll die Bearbeitung der Anfrage selbstständig übernehmen und die Antwort direkt an den anfragenden Client zurückschicken. Ihre Implementierung soll folgende Anforderungen erfüllen:

- Ein Dispatcher-Thread nimmt die Anfragen von Clients entgegen und erstellt für jede Anfrage einen Worker Thread, der die Bearbeitung der Anfrage übernimmt.
- Nachdem der Worker Thread die Bearbeitung der Anfrage beendet hat, terminiert er sich selbstständig. Hinweis: Es dürfte hilfreich sein, die Worker Threads detached zu betreiben.

Aufgabe 3. Prozessunterbrechungen messen mit pselect (4 Punkte)

Hinweis: `select` für I/O-Multiplexing wird erst in der Vorlesung am 23.01.2024 besprochen.

Ähnlich wie schon in der vorherigen Aufgabe *Prozessunterbrechungen messen* sollen Sie ein C-Programm schreiben, das in einer Schleife eine gegebene Zeitspanne wartet und die tatsächlich gewartete Zeit misst. Dabei sollen Sie dieses Mal allerdings mithilfe von `pselect` warten. Nutzen Sie hierfür den Timeout-Parameter.

Wie bisher soll die zu wartende Zeitspanne und die Zahl der Iterationen über einen Kommandozeilenparameter in Nanosekunden übergeben werden. Der Aufruf, wenn das Programm 100 ns warten und 10 000 Iterationen laufen soll, soll aussehen wie

```
./timing 100 10000
```

.

Schreiben Sie in jedem Durchlauf die tatsächlich gemessene Zeitdifferenz in Nanosekunden in eine neue Zeile einer Datei, die sie mit dem Namen **wartezeit_iterationen.txt** abspeichern; im obigen Beispiel soll sie also **100_10000.txt** heißen.

Vergleichen Sie Ihre Ergebnisse mit den Ergebnissen von vorher. Beschreiben Sie kurz Ihre Erwartungen und ob sich diese bestätigt haben. Was ist der Unterschied zwischen dem hier verwendeten `pselect` und dem in der Vorlesung vorgestellten `select`?

Bonusaufgabe IV. Serverbasierter TCP-Chat mit select-loop (4 Punkte)

Hinweis: `select` für I/O-Multiplexing wird erst in der Vorlesung am 23.01.2024 besprochen.

Auf den vorausgegangenen Zetteln haben Sie einen Chat via UDP realisiert. Nun soll die Kommunikation mittels TCP durchgeführt werden. TCP unterstützt ausschließlich Punkt zu Punkt-Kommunikation („Unicast“). Um trotzdem eine Kommunikation zwischen mehr als zwei Teilnehmern zu ermöglichen, melden sich mehrere Chat-Clients an einem Chat-Server an, der für die Weiterleitung der einzelnen Nachrichten an alle Teilnehmer des Chats sorgt.

Der Server, der prinzipiell eine beliebige Anzahl an Clients unterstützen soll, soll eine Select-Loop benutzen (vgl. Vorlesungsfolien), um auf allen Verbindungen zu den einzelnen Clients gleichzeitig empfangsbereit zu sein. Empfängt der Server eine Textnachricht von einem Client, so leitet er die Nachricht unverändert an alle Clients (auch den Sender) weiter. Stellen Sie dabei sicher, dass sich während des Chats beliebige Clients vom Server an- und abmelden können.

Neben dem Server sollen Sie auch noch einen dazugehörigen Chat-Client implementieren, der `select` verwendet, um zwischen Tastatureingaben und vom Server weitergeleiteten Nachrichten zu wechseln. Der Client stellt der eigentlichen Nachricht jeweils den Nickname des Nutzers (per Kommandozeile zu übergeben) und die eigene IP-Adresse voran. Die eigene IP-Adresse können Sie mit Hilfe von `getsockname` auf dem verbundenen Socket ermitteln.

Testen Sie Ihren Client wenn möglich auch mit anderen Implementierungen von Kommilitonen. Verwenden Sie daher für den Server einheitlich die Portnummer 4711.