Sovellusohjelmointi Assignment 2 oppimispäiväkirja, Santeri Kokkonen 907679, 19.12.2023

Aluksi hieman ohjelmani toiminnasta.
Ohjelma toimii aivan kuten se on kuvattu tehtävänannossa, ja normaalissa ohjeiden mukaisessa käytössä se ei ole kertaakaan kun olen itse sitä käyttänyt törmännyt mihinkään virheeseen. Toteutin pthread kirjastolla ohjelman säikeistyksen.

Tehtävänannossa jäi ehkä hieman epäselväksi, että kuinka rinnakkaista ohjelman toiminta tulee olla. Tällä viittaan siihen, että tehtävänannossa puhutaan “Individual R/W locks”. Tämä oli mielestäni tulkinnanvarainen, ja päädyin siihen lopputulokseen, että yksittäinen MUTEX lukko riittäisi. Eli minulla ohjelmani toimii niin, että aina yksi käyttäjä kerrallaan saa käsitellä tietokantaa, vaikka serveri palvelee useampaa asiakasta samaan aikaan eri säikeissä.

Tein tämän näin siksi, koska itse ohjelma ei missään vaiheessa ota omaan muistiin tietokantaa, vaan tietokanta on olemassa koko ajan erillisessä tiedostossa, joka kirjoitetaan uudestaan aina, kun jonkin tilin balanssiin tehdään muutos.

Etsiessäni tietoa netistä tiedoston uudelleen kirjoittaminen kerrottiin olevan simppelein ja fiksuin tapa toteuttaa tälläinen text tiedosto tietokanta. Kääntöpuolena tässä on nyt kuitenkin se, että serveri ei varsinaisesti muokkaa tietokantaa rinnaikkaisissa säikeissä, mutta se on täysin resistantti datan korruptoinnille.

Ymmärrän, että tämän olisi voinut toteuttaa enemmän rinnakkaisena niin, että serverin ollessa päällä se lukisi omaan muistiin tietokannan, jossa se voisi antaa jokaiselle tililllle oman mutex lukon, jolloin eri säikeet voisivat tehdä muutoksia eri tileihin rinnakkain serverin muistissa, ja serveri voisi aina väli ajoin ja sen sulkeutuessa kirjoittaa muistista tietokannan uudestaan. Periaatteessa tein hyvin pitkälle tällaisen version, mutta totesin, että alkuperäinen toteutus on ihan tarpeeksi hyvä ja nopea.

Pohdin, oman toteutukseni mahdollisia huonoja puolia. Niitä tulee mieleen muutama: 1. Jos käyttäjä säikeitä on hurja määrä, niin voi tämä sarjaan suoritettava tapa hidastua liikaa, 2. Jos pankkiin rekistyröityneiden tilien määrä on suuri niin aina tietokannan uudelleen kirjoittaminen jokaisen yksittäisen muutoksen jälkeen voi alkaa olla liian hidas ja 3. Näiden kahden tapauksen yhteinen skenaario, eli tilanne jossa on suuri määrä sen hetkisiä käyttäjiä ja rekistyröityneitä tilejä. Kuitenkin tämän tehtävän yhteydessä, kun tilien ja käyttäjien määrä oli kohtuullinen, tästä ei ollut haittaa testipenkissä niin kelpuutin alkuperäisen toteutuksen.

Toinen mainitsemisen arvoinen seikka, jos käyttäjä yhdistyy palveluun, joutuu jonoon ja hän jonossa ollessaan antaa käskyjä niin, heti hänen päästessä palveltavaksi kaikki nämä annetut käskyt toteutetaan suoraan putkeen.

Myös tässä syntyy ohjelman ainoa löytämäni bugi. Jos käyttäjä jonossa ollessaan terminoi client sovelluksen, niin heti kun hän olisi pääsemässä palveltavaksi serveri kaatuu. Tähän ei myöskään auta se, että jos hän avaa uudestaan client:in ja yrittää yhdistää uudelleen deskiin tai, että hänen jälkeen on muita jonossa. Tällöin kuitenkaan mitään hänen tai muiden jonossa olleiden käskyjä ei toteuteta.

Arvioin, että tämä johtuu socketeista ja eri “read/write” käskyistä, jolloin serveri varmaan yrittää lukea jotain mitä ei ole olemassa ja crashaa. Ajatuksen tasolla en millään tahtonut keksiä mitään yksinkertaista ratkaisua tähän ongelmaan johtuen socketeista. Periaattessa ohjelman toiminta ei vaarannu tästä seikasta olettaen, että kaikki käyttäjät odottavat rauhassa vuoroaan ja eivät poisto palvelusta ennen kun heitä on palveltu. Oikeassa sovelluksessa/nettipankissa tämä kuuluisi ottaa huomioon totta kai.

Tehtävän/ohjelmoinnin vaiheista
Tehtävä oli omalta osaltani suhteellisen suoraviivainen ja sujuva. Harmi kyllä ennätin aloittamaan suorituksen vasta joulukuussa johtuen kertausharjoituksista. Sitten kun vihdoin sain aloitettua prosessi oli aika ongelmaton.

Karkeasti se meni näin. Aluksi opettelin miten socketit toimivat, tässä suurena apuna oli yhden luennon koodit (userver ja uclient), joita käytin pohjana. Seuraavaksi serverin yhdistäminen tietokantaan ja clientin komentojen toteuttaminen siellä. Tässä oli apua annetusta as2_mockup.c:stä. Nämä ensimmäiset vaiheet menivät aika hujauksella.
Sitten oli ehkä projetkin eniten ajatustyötä vaativa osuus eli serverin säikeistäminen. Tämä oli vaikein osuus, koska minun piti opiskella miten ensinnäkin säikeistäminen toimii, mutta eniten siksi koska netistä/luennoista löytyi niin paljon tietoa ja eri esimerkkejä niin asiasta tietämättömänä oli vaikeaa valita juuri niitä osia mitkä oman ohjelman kannalta on tärkeää. Säikeistämisen jälkeen tein R/W lukon, mikä oli hyvin simppeli.
Lopuksi korjasin koodin varoituksista ja muistivuodoista ja kommentoin sen, ja tein lokitiedoston. Olin onnistunut aikaisemmissa vaiheissa hyvin, joten tähän viimeiseen vaiheeseen ei mennyt liikaa resursseja.
Kaiken kaikkiaan prosessi meni hyvin ja pärjäsin projektin kanssa 90% itse. Ainoa asia koodissani, joka ei ole täysin oma ideani on se, että clientit yhdistävät ensin johonkin heidän valmiiksi tietämään sockettiin (main_desk), jossa heidät otetaan vastaan, ja heille välittömästi lähetetään service socket (desk), johon he sitten yhdistävät.

Kohdatuista onglemista ja niiden ratkaisuista
Mitään suuria ongelmia tehtävänteko prosessin aikana ei tullut. Pieniä virheitä muistin ja pointtereiden kanssa tuli tehtyä. Esimerkiksi, kun serveri vastaanottaa käskyjä socketin kautta ne ovat vaihtuvan pituisia ja oman koodini kannalta sen pitää tallentaa käskyt erilliseen string muuttujaan jatkoa varten. Mutta ongelmana oli se, että jos ensin luetaan merkkimäärältään pitkä käsky ja sitten lyhyt, tähän string muuttujaan jäi aikaisemman käskyn merkkejä, jotka voisivat olla ongelma, kun käskyä käsittellään. Ratkaisu tähän oli, se että minulla oli string muuttujan kokoinen 0 alustettu muuttuja, jolla aina resetoin string muuttujan käskyjen välillä.

Toinen esimerkki ongelma oli SIGINT/SIGTERM hallittu käsittely. Tämä oli ongelma sen takia, koska kaikki säikeet aina jäävät odottamaan johonkin koodin “read” kohtaan, jolloin jos vaikka teki jonkin globaalin flagin joka signaloi SIGINT/SIGTERMI:ä niin ohjelman sulkeutuminen jäi jumiin. Myös jos kutsui vain suoraan “exit()” funktiota ei se silloin poikkeaisi ollenkaan ohjelman normaalista toiminnasta SIGINT/SIGTERM:in tullessa.

Hieman etsittyäni löysin ratkaisuksi “pthread_cancel()” ja “pthread_setcancelstate()” funktiot. Tällöin näiden funktioiden avulla pystyin signal handler funktiosta käsin erikseen

lopettamaan kaikki säikeet ja vielä niin, että jos ne olivat juuri tekemässä muutosta tietokantaan niin se toteutettaisiin.

Lisäksi piti hieman kikkailla muistin kanssa juurikin tähän hallittuun alasajoon liittyen. Kun lähdin compilemaan koodiani kaikkien flagien kanssa törmäsin muutamaan “conditional jump or move depends on uninitialised values” varoitukseen tämän takia päätin koodini muutamassa kohdassa dynaamisesti allokoida muistia esim serverin copydata funktiossa. Tämä tarkoitti sitä, että piti olla tarkka että muistaa deallokoida muistit, myös hallitun alasajon tapauksessa.

Testaus
Tehtävän tekemisen aikana koodin testaus pääasiassa tapahtui usean eri wsl terminaalin ja printf() komentojen avulla. Koodin kaikkien toimintojen valmistuttua aloin testaamaan sitä annetulla testipenkillä. Testipenkissä annoin argumenteiksi 1-100 maxuser ja 1-100 numtests. Ohjelmani läpäisi testipenkin ja toimi kaikilla testaamallani asetuksilla ja kerroilla.

Ohjelman edelleen kehittäminen/parannus
Tulee mieleen muutamia asioita, joilla voisi ohjelmaa parantaa.

1. Toteuttaisi ohjelman muistin sisäisen tietokannan, jossa käyttäjät voisivat rinnakkain muokata eri tilejä.
2. Koodin puhtaaksi kirjoittaminen, turhien ja monimutkaisten koodirakenteiden poisto/yksinkertaistaminen
3. Koodin tarkastelu tehokkuuden kannalta ja sen maksimointi
4. Mahdollisesti jonkinlaisen graafisen käyttöliittymän tekeminen client puolelle
5. Lokitiedoston parantelu: aikaleimat, tarkemmat kuvaukset yms.
6. Ohjelman yhdistäminen internettiin niin, että sitä voisi käyttää verkon yli
7. Korjata serverin crashaamis bugi, kun client terminoi oman prosessin jonossa ollessaan.