Del 1: Synkronisering

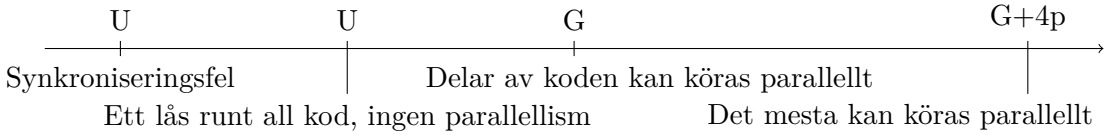
Bedömning

För att få godkänt på tentan krävs att den här uppgiften är godkänd. Uppgiften kan dessutom ge totalt 6 bonuspoäng.

För att uppgiften ska bedömas som godkänd ska koden fungera enligt beskrivningen nedan och enligt beskriven i kommentarerna i den givna koden. Busy-wait ska undvikas, och olägliga trådbyten ska inte kunna orsaka fel givet att koden används enligt specifikationen, oberoende av hur osannolikt detta kan vara. Om inte annat anges ska alla funktioner kunna anropas från flera trådar samtidigt utan att det leder till problem, oavsett om detta görs i exempelprogrammet eller inte. Om du inser att du har glömt något kan du fortfarande få G på uppgiften genom att i del 2 beskriva varför det är fel och beskriva hur du skulle lösa problemet.

För upp till 4 bonuspoäng ska det vara möjligt för kod som arbetar på orelaterad data att köras parallellt i så stor mån som möjligt. Detta innebär att kritiska sektioner ska vara så korta som möjligt, och att lås i den mån det är möjligt är placerade tillsammans med den data de skyddar. Alltså ska exempelvis kod som hanterar orelaterade datastrukturer kunna köras parallellt, och i den mån det är möjligt ska även operationer på samma datastruktur kunna köras parallellt. Detta gäller även om exempelprogrammet inte instansierar flera datastrukturer, eller kör funktioner parallellt från flera trådar.

Bedömningen kan sammanfattas i diagrammet nedan:



Utöver detta är en del av koden och uppgiften märkt med `Bonusuppgift`. Denna del behöver **inte** behandlas för G, men kan ge upp till 2 bonuspoäng. För full poäng vill vi även här se maximal teoretisk parallellism.

Notera: all kod under kommentaren *Huvudprogram* är inte en del av uppgiften och kommer inte att rättas, så lägg ingen synkronisering där. Huvudprogrammet finns för att programmet ska gå att köra, och för att visa hur resterande kod kan användas. Det är *inte* byggt för att illustrera alla potentiella problem i den givna koden. Du får gärna modifiera huvudprogrammet för att testa din lösning om du vill, det är dock inget krav.

Notera: I denna uppgift ska du använda lås, semaforer, och/eller condition variables. Atomiska operationer används i en senare uppgift.

Inlämning

Modifiera koden i filen `invoice.c` och lämna in din lösning i Lisam senast klockan 9:30.

Ange följande text i textfältet där du lämnar in uppgiften för att visa att du har läst och förstått reglerna i regeldokumentet:

Jag har läst och förstått tentans regler, och jag lovar att jag har följt dem under tentan.

Uppgift

Företaget Bricksbyggen (som bygger och underhåller LEGO-hus) har problem med sitt fakturasystem. Sedan årsskiftet har de erbjudit kunder att få sin faktura skickad via e-post i stället för vanligt brev. Dock så verkar det som att några fakturor försvinner ibland. Det är mycket problematiskt eftersom varken Bricksbyggen eller kunden noterar detta förrän nästa månad då kunden (förhoppningsvis) får en ny faktura med påminnelseavgift, vilket såklart gör kunden något upprörd.

Systemet som hanterar utskicken är byggt utefter antagandet att det tar tid att skicka e-postmeddelanden (det måste prata med en server som kanske är upptagen och som kanske är långt borta). Därför läggs alla utgående fakturor på kö, så att systemet kan skicka ut dem i lugn och ro utan att personalen ska behöva vänta på systemet hela tiden.

Systemet gör dock sitt bästa för att skicka ut fakturorna så snabbt det kan. Personen som byggde systemet insåg att det är problematiskt att försöka skicka flera fakturor till e-postadresser på samma server. För att göra det skulle man behöva skapa flera anslutningar till samma server samtidigt, något som felaktigt kan tolkas som att Bricksbyggen försöker skicka spam. Däremot är det inga problem att skicka meddelanden till e-postadresser på olika servrar samtidigt!

För att kunna skicka ut meddelanden till olika servrar samtidigt på det viset finns det inte bara *en* kö i systemet, utan en kö för *varje* mottagarserver. När en faktura ska skickas ut kommer systemet alltså undersöka vilken server som ska ta emot meddelandet (det som står efter `@`), och lägga det i motsvarande kö. Exempelvis läggs fakturor till `a@liu.se` och `b@liu.se` i samma kö, medan fakturor till `c@svt.se` och `d@svd.se` i två separata köer. Detta implementeras i funktionen `send` (som är en del av bonusuppgiften).

Varje kö representeras som en instans av typen `connection`, som utöver kön innehåller anslutningen till servern (i den givna koden skriver vi till en fil i stället för att skicka e-postmeddelanden). När en `connection` skapas, startas också en tråd som ansvarar för att skicka ut de fakturor som läggs till i kön. Typen har tre publika funktioner:

`connection_create` Skapar en instans av `connection`. Startar även en tillhörande tråd.

`connection_send` Lägger till en faktura på kön. Tråden kommer i sinom tid att skicka fakturan. Om kön råkar vara full ska funktionen vänta på att det finns plats i kön. Implementationen garaneterar inte i vilken ordning fakturorna skickas ut.

`connection_destroy` Stänger anslutningen och frigör minne. Implementationen ser till att vänta på att alla meddelanden som lagts till med `connection_send` faktiskt har skickats innan tråden stängs av och minnet frigörs.

Filen `invoice.c` innehåller Bricksbyggens implementation av datatypen `connection`. Hjälp dem att synkronisera sin implementation så att fakturor inte längre försvinner. Du har också hört rykten om att systemet använder mycket CPU-resurser trots att det inte gör något, vilket direkt får dig att tänka på *busy-wait*. Du tänker att du lika gärna kan lösa det problemet när du löser resterande synkroniseringsproblem.

Bonusuppgift (2p): Ibland verkar det som att systemet skapar flera anslutningar till samma server, vilket gör att fakturorna ibland hamnar i spamfiltret. Du tror att problemet ligger i `send`-funktionen. Lös synkroniseringsproblemen där.

Du kan kompilera koden med kommandot `make invoice` i mappen där du har extraherat tentafilerna. Kompileringen ska fungera på det Linux-system du har använt för att göra labbserien. Din kod behöver *inte* kompilera för att ge poäng, så länge vi förstår vad du menar.