Del 1: Synkronisering

Bedömning

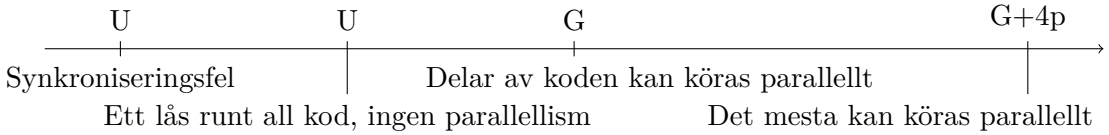
För att få godkänt på tentan krävs att den här uppgiften blir godkänd. Uppgiften kan dessutom ge totalt 4 bonuspoäng.

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 problem, oavsett om detta görs i exempelprogrammet eller inte.

Notera: Om du inser att du har glömt något i din inlämning av del 1 så kan du fortfarande få G på uppgiften genom att beskriva vad som är fel och hur du skulle lösa problemet i del 2.

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. Inser du under del 2 att din lösning kan förbättras så kan du där poängtera det för att få extrapoängen även där.

Bedömningen kan sammanfattas i diagrammet nedan:



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 `server.c` och lämna in din lösning i Lisam senast klockan 09:30.

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

Din kompis driver en hemsida där du ibland publicerar snygga och effektiva lösningar på synkroniseringsproblem. Din kompis hörde nyligen av sig och berättade om alla säkerhetshål som hittas i mjukvara hela tiden, och ni kom fram till att den bästa lösningen är att skriva en egen webbserver (självklart har ni ju inte buggar i er kod, det känns ju onödigt). Självklart vill din kompis använda trådar för att kunna utnyttja alla kärnor i sin fina server.

För att effektivt använda trådar i webbservern har ni kommit fram till att ni vill skapa ett antal trådar (4 till att börja med), och sedan fördela arbetet mellan dem allt eftersom servern får saker att göra. Ni kallar detta för en trådpool. Fördelen med den här lösningen är att ni vet att ni inte kommer att råka överbelasta servern genom att skapa för många trådar, eftersom ni alltid har ett visst antal trådar igång.

Din kompis har börjat implementera webbservern, men har stött på trådningsproblem i sin prototyp och ber därför dig om hjälp. Den nuvarande implementationen finns i filen `server.c`. I filen finns följande funktioner:

- `process_request`: Det är här ni kommer att implementera er webbsida framöver. Den tar in en fråga från en användares webbläsare (i form av en sträng) och ska sedan procudera ett svar. Just nu skriver den bara ut ett meddelande i terminalen, men det duger än så länge. Målet är att kunna anropa `process_request` ifrån flera trådar samtidigt på ett säkert sätt.
- `pool_create`: Startar det givna antalet trådar och kopplar dem till en instans av `struct thread_pool`. Trådarna väntar sedan på att få arbete från `pool_handle_request`.
- `pool_handle_request`: Be någon tråd att hantera en fråga (en sträng). Funktionen väntar på att någon av trådarna i trådpoolen är lediga och ber den tråden att hantera frågan. Funktionen ska inte vänta på att frågan har hanterats färdigt. Ni vill att den här funktionen ska kunna anropas ifrån flera trådar samtidigt, för samma trådpool-objekt.
- `pool_destroy`: Vänta på att alla trådar har arbetat klart, och stäng sedan av alla trådar och frigör minne. Vi antar att den här funktionen inte körs samtidigt som någon annan tråd håller på att köra `pool_handle_request`.
- `worker_main`: Funktionen som nya trådar kör. Väntar på att få arbete och utför sedan arbetet.

Efter lite testande så har ni problemen nedan i koden. Din kompis ber såklart dig att lösa problemen eftersom du ofta publicerar lösningar på trådningsproblem på hemsidan. Du kan titta i filen `wrap/synch.h` för att se vilka synkroniseringsprimitiver som finns (det är samma som i Pintos).

1. Det verkar som att programmet använder mycket CPU, även om ingen av trådarna har fått några frågor att hantera (dvs. inga anrop till `pool_handle_request`).
2. Om två trådar anropar `pool_handle_request` samtidigt så verkar det som att en av frågorna ibland "glöms bort".
3. Ibland så kraschar koden precis efter att `pool_destroy` har anropats.

Du kan kompilera koden med kommandot `make server` i mappen där du har extraherat tentafilerna, och sedan köra koden med `./server`. Detta bör 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.