Obligatorisk uppgift 3: mdu

Namn: **Abdiaziz Ibrahim Adam**
CS-användare: **dv23aam**
Kurs: **Systemnära programmering**

November 30, 2024

1 Resonemang om trådsäkerhet

I detta projekt har trådsäkerhet varit en avgörande aspekt, särskilt då programmet använder sig av flera trådar för att parallellisera arbetet med katalogtraversering. För att uppnå trådsäkerhet i programmet har följande åtgärder vidtagits:

- **Mutexlåsning av kritiska sektioner:** Delade resurser, såsom uppgiftsstacken och lagringen av katalogstorlekar, skyddas av mutexlås. Mutexar används här för att säkerställa att endast en tråd åt gången kan komma åt kritiska data, vilket effektivt förhindrar race conditions. Varje mutex används specifikt för att låsa den del av data som är relevant för aktuell operation och frigörs omedelbart efter att tråden har slutfört sin uppgift, vilket minimerar potentiella lås- och prestandaproblem.

- **Villkorsvariabler för effektiv synkronisering:** För att undvika busy-waiting när uppgiftsstacken är tom används villkorsvariabler (`pthread_cond_t`). När en tråd finner stacken tom väntar den istället på en signal om nya uppgifter i stacken, vilket görs genom att tråden sätts i ett vilande tillstånd istället för att konstant kontrollera stacken. När nya uppgifter läggs till i stacken signaleras de väntande trådarna via villkorsvariabeln, vilket effektivt väcker upp trådarna utan onödig CPU-användning. Denna lösning minimerar resursförbrukning och effektiviserar hanteringen av trådar vid varierande arbetsbelastning.

- **Avslutning av trådar och samordning av arbete:** För att hantera trådarnas avslutning och säkerställa att alla uppgifter bearbetas korrekt används en central struktur, `ThreadPool`. Här spåras varje tråds status och arbetsuppgifter synkroniseras med hjälp av ett räkningssystem för väntande trådar. Om alla trådar är i vänteläge och uppgiftsstacken är tom signaleras en arbetsnedläggning via villkorsvariabler. Detta säkerställer att programmet avslutas korrekt utan att lämna kvarhängande trådar eller osynkroniserad data.

- **Säkerhetsåtgärder vid felhantering:** Eftersom många trådsäkerhetsmekanismer, såsom mutex- och villkorsvariabler, kan returnera felstatus används en central funktion, `handle_error`, som hanterar och loggar fel innan programmet avslutas. Detta ger en robust hantering av potentiella problem, såsom låsfel, som annars kan skapa inkonsistens och påverka programstabiliteten.

Ett annat viktigt område har varit att förhindra deadlocks. Deadlocks kan uppstå om flera trådar väntar på resurser som hålls av varandra, vilket leder till att ingen av dem kan fortsätta sitt arbete. I detta program undviks deadlocks genom att varje mutex används för att låsa specifika resurser i en tydlig och konsekvent ordning. Dessutom ser implementationen av villkorsvariabler till att trådar inte låser resurser medan de väntar på signaler, vilket ytterligare minskar risken för deadlocks.

2 Analys av prestandan

För att analysera programmets prestanda testades `mdu` på datorn Itchy med katalogen `/pkg/`. Analysen genomfördes genom att variera antalet trådar från 1 till 100. Itchy, som

har 64 kärnor och stöd för 64 trådar, ger en bra grund för att utvärdera hur effektivt programmet utnyttjar parallellism. Körtiderna mättes för varje trådkonfiguration, och resultaten visualiserades i en graf som visas i figur 1. På x-axeln representeras antalet trådar och på y-axeln den faktiska körtiden i sekunder.

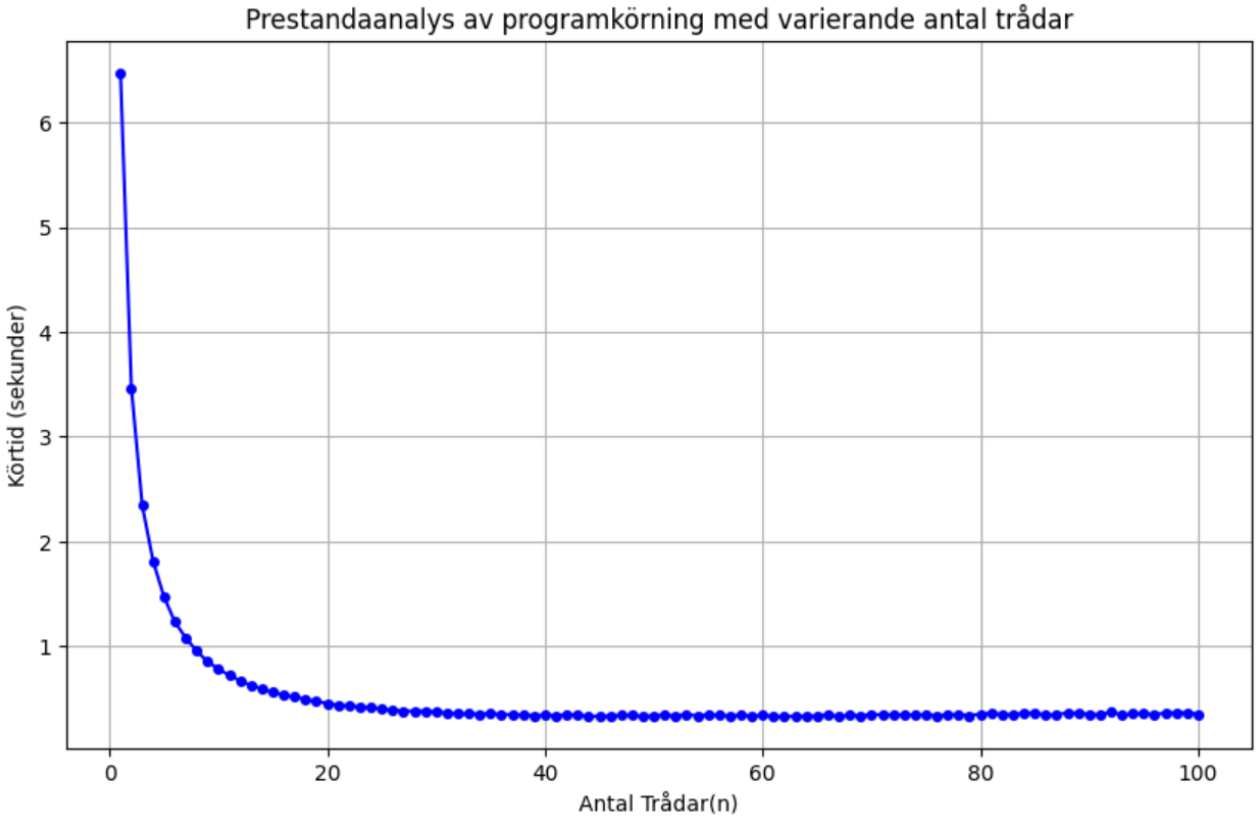

Figure 1: Körtid för `mdu` med varierande antal trådar.

Observationer från grafen:

- **Effektiv förbättring upp till 20 trådar:** Programmet visar en tydlig och markant förbättring av körtiden när antalet trådar ökar från 1 till 20. Vid användning av en enda tråd är körtiden 6.467 sekunder. När två trådar används halveras nästan körtiden till 3.462 sekunder, vilket visar en omedelbar fördel av parallell bearbetning. Med fyra trådar sjunker körtiden ytterligare till 2.343 sekunder, och vid åtta trådar når programmet 0.955 sekunder, en betydande minskning jämfört med färre trådar. Vid 16 trådar reduceras körtiden till 0.535 sekunder, vilket motsvarar en förbättring på över 90% jämfört med en enda tråd. Den största förbättringen observeras dock vid 20 trådar, där programmet når sin bästa prestanda med en körtid på endast 0.456 sekunder. Detta innebär en total tidsreduktion på över 92%, vilket kan tillskrivas programmets förmåga att bearbeta flera kataloger parallellt och utnyttja Itchys 64 kärnor effektivt
- **Avtagande förbättring efter 20 trådar:** Programmet visar en tydlig prestandaförbättring när antalet trådar ökar upp till cirka 20. Detta beror på att Itchy, med sina 64

kärnor, effektivt kan hantera flera trådar parallellt utan resurskonflikter. Efter 20 trådar börjar dock prestandavinsten avta, vilket kan tillskrivas den ökade extraarbetet (overhead) som krävs för att hantera fler trådar. Detta extraarbete inkluderar synkronisering av trådar, kontextväxling (där processorn växlar mellan olika trådar) och väntetid för åtkomst till delade resurser som hanteras av mutexar och villkorsvariabler. När detta extraarbete ökar, spenderar programmet mer tid på att koordinera trådarna än att bearbeta uppgifterna, vilket leder till minskad effektivitet. Detta fenomen syns tydligt i prestandagrafen, där körtiden inte förbättras nämnvärt trots att fler trådar används efter 20.

Sammanfattningsvis observerades den mest effektiva körningen vid **20 trådar**, där körtiden uppmättes till **0.456 sekunder**. Detta representerar en avsevärd förbättring jämfört med körning med en enda tråd, där körtiden var **6.467 sekunder**. Vid 20 trådar kan programmet parallellisera arbetet effektivt och utnyttja Itchys flertrådiga arkitektur optimalt.

Förbättringen i procent kan beräknas enligt följande:

$$Förbättring(\%) = \left(\frac{\text{Tid vid 1 tråd} - \text{Tid vid 20 trådar}}{\text{Tid vid 1 tråd}} \right) \times 100$$

Med värdena:

$$Förbättring(\%) = \left(\frac{6.467 - 0.456}{6.467} \right) \times 100 \approx 92.95\%$$

Detta visar att körtiden minskade med över **92.95%** vid 20 trådar, vilket bekräftar att denna konfiguration ger den bästa balansen mellan parallellisering och synkronisering.

3 Diskussion och reflektion

Under arbetet med denna uppgift stötte jag på flera utmaningar som krävde en djupare förståelse för parallell programmering och trådsäkerhet.

En av de största utmaningarna var att implementera trådsäkerhet, särskilt genom korrekt användning av **mutexar** och **villkorsvariabler**. Vid åtkomst till delade strukturer, som uppgiftsstacken och katalogstorleksdata, användes mutexar för att förhindra race conditions. Det var avgörande att planera mutexanvändningen noggrant för att undvika onödig låsning, vilket annars kunde leda till prestandaförluster. Att skapa en effektiv och balanserad hantering av trådar var en central del av utvecklingsarbetet.

Ett annat problem var att vissa trådar överbelastades medan andra förblev inaktiva. För att lösa detta implementerade jag en gemensam uppgiftsstack, vilket jämnade ut arbetsfördelningen mellan trådarna och förbättrade programmets effektivitet.

Överlag var uppgiften både utmanande och lärorik. Jag uppskattade möjligheten att fördjupa min kunskap inom parallell programmering och trådsäkerhet. Att designa och implementera en lösning där flera trådar samarbetade effektivt för att bearbeta stora mängder data gav mig en större förståelse för systemnära programmering och hur parallellisering kan användas för att optimera prestanda. Jag är nöjd med resultatet och känner att jag har utvecklats som programmerare genom att arbeta med denna uppgift.