Mittuniversitetet
MID SWEDEN UNIVERSITY

Laboration 4 - Trådar
DT011G Operativsystem introduktionskurs

Jimmy Åhlander*

9 mars 2021

Innehåll

1 Introduktion

Det finns flera anledningar till att arbeta med flera trådar i processer. Bland annat kan generellt prestandan förbättras genom att nyttja multiprocessorarkitekturer och responsen i användargränssnitt hållas låg genom att köra belastande operationer och gränssnittet på separata trådar.

*jimmy.ahlander@miun.se. Avdelningen för informationssystem och -teknologi (IST)

Att arbeta med trådar kan dock leda till synkroniseringsproblem. Ett klassiskt sådant problem, *the readers-writers problem* utforskas i den här laborationen.

2 Mål

Efter genomförd laboration ska du förstå grundläggande synkronisering, den underliggande problematiken för ett givet synkroniseringsproblem och kunna implementera trådsäkerhet.

Följande lärandemål är kopplat till laborationen. Du ska kunna:

- redogöra för de vanligaste problemen med resursfördelning och synkronisering samt känna till vanliga lösningar på dessa problem.

3 Teori

Trådar i Python

I teorin följer en rad fördelar med flertrådade processer. I praktiken är det fullt beroende på vilket stöd som finns för trådar, dels på kärnnivå och dels på användarnivå. Det finns många olika Pythontolkar, men de facto standard är tolken CPython. I CPython kopplas användartrådar till kärntrådar enligt en-till-en-principen. Schemaläggningen av trådarna lämnas därefter till operativsystemet. CPython implementerar dock ett *global interpreter lock* (GIL) vilket förhindrar nyttjandet av flera kärntrådar för de flesta vanliga operationerna. Detta begränsar effektivt exekveringen av den huvudsakliga koden till en kärna och eliminerar potentiella prestandavinster, men övriga fördelar kvarstår mestadels. Overheaden för GIL-hanteringen blir också stor vilket gör att två trådar som delar en uppgift kommer att utföra den noterbart långsammare än om en enskild tråd utfört uppgiften själv [1]. I python.orgs wiki finns en djupare diskussion om GIL och andra Pythontolkar [2].

Att GIL existerar förhindrar dock inte synkroniseringsproblem i egenhändigt författad kod som nyttjar trådmodulen **threading**. Enskilda opcodes eller bytecodes skyddas av GIL men även så simpel högnivåkod som `i += 1` består av ett flertal opcodes och kan därför bli desynkroniserad.

The readers-writers problem

Det finns ett flertal klassiska synkroniseringsproblem så som *the producer-consumer problem*, *the readers-writers problem* och *the dining philosophers problem*. Alla har olika förutsättningar där flera aktörer arbetar mot gemensamma resurser vilket kan leda till en variation av problem; generellt är problemen racetillstånd och deadlocks. En aktör kan exempelvis vara en process eller en tråd. I den här laborationen är aktörerna trådar.

I the readers-writers problem finns en gemensam resurs där en grupp aktörer kallade läsare vill läsa från resursen och en annan grupp aktörer kallade skrivare vill skriva till resursen. Det grundläggande synkroniseringsproblemet som måste lösas är:

1. Ömsesidig uteslutning *(mutual exclusion)* för resursen, där:
    - endast en skrivare får skriva åt gången, och
    - ingen skrivare får skriva medan en läsare läser.

Ett vanligt mutex-lås som appliceras lika för alla aktörer kan lösa detta men kommer leda till att endast en aktör kan gå in i sin kritiska sektion åt gången. Detta är ett problem då vi normalt vill att flera läsare ska kunna läsa samtidigt. Nästa problem som måste lösas är därför följande:

2. Flera läsare ska kunna läsa resursen samtidigt.

I en lösning där flera läsare kan läsa samtidigt och låsa ut skrivarna finns alltid en risk för svält hos skrivarna. Ju fler läsare och ju mer tid de spenderar i sin kritiska sektion desto mer osannolikt blir det att skrivarna hittar en öppning där samtliga läsare befinner sig utanför sin kritiska sektion. Det sista problemet som måste lösas är därför följande:

3. Så snart en skrivare vill skriva så får inte nya läsare gå in i sin kritiska sektion.[1]

Med andra ord prioriteras skrivare över läsare. I regel är detta att föredra över motsatsen då en resurs, exempelvis en variabels värde, är inaktuellt för en läsare tills dess att skrivaren uppdaterat den. Läsarna läser alltså ett inaktuellt värde medan skrivaren väntar.

Det är möjligt att gå ännu djupare i laborationen och readers-writers problem genom att använda synkroniserade köer så att läsare och skrivare blir likvärdigt prioriterade utan att svält uppstår för endera. Det är dock inte aktuellt i denna laboration.

4 Genomförande

Din uppgift är att skapa ett program som löser de tre problemen listade i Teori med hjälp av trådar i Python. Din lösning måste också leva upp till kraven som ställs på alla lösningar för kritiska sektionen [3, s. 256].

Innan du påbörjar laborationen måste du läsa sektionen Examination för viktig information om vad du ska leverera.

Arbeta med trådsäkra strukturer där det behövs och skydda alla icke-trådsäkra

[1] Läsare som redan exekverar i sin kritiska sektion behöver emellertid inte spärras (avbrytas).

strukturer med synkroniseringsmekanismer som lås eller semaforer. Använd modulen threading [4] för aktörerna: läsarna och skrivarna.

Den delade resursen som ska implementeras är en textsträng. Två olika skrivare försöker kontinuerligt skriva till textsträngen. Den ena skrivaren skriver en datumstämpel hela vägen ned till sekunder. Den andra skrivaren skriver samma sak, fast skriver strängen baklänges. Åtminstone tre identiska läsare försöker samtidigt kontinuerligt läsa textsträngen och skriva ut den till standardutströmmen. Se även Tabell 1 för en sammanfattning.

Tabell 1: En sammanfattning över de aktörer som ska implementeras.

Aktör	**Antal**	**Uppgift**
Läsare	3+	Läser och skriver ut textsträngen till standardutströmmen.
Skrivare	1	Skriver en datumstämpel, inklusive sekunder till textsträngen.
Skrivare	1	Skriver en omvänd datumstämpel, inklusive sekunder till textsträngen.

Försök att adressera problemen en efter en. Börja med det första problemet och jobba dig uppåt i problemkedjan, även om det så kräver att du skriver om tidigare kod. Att försöka lösa samtliga krav i ett svep är en stor utmaning.

Under tiden du arbetar med lösningen rekommenderas utskrifter vid relevanta delar av koden för både läsare och skrivare för att få en bättre uppfattning av när trådarna exempelvis går in i och ut ur sin kritiska sektion. I den färdiga lösningen ska emellertid endast lästrådarna skriva något till utströmmen. Du kan även ha nytta av modulen **time** för att fördröja trådarnas exekvering. Detta är användbart för att enklare kunna tolka utskrifterna och för att konstruera scenarion som utmanar din lösning.[2] Din lösning får dock inte förlita sig på fördröjningar för att adressera problemen — på samma sätt som trafikljusen i en korsning inte kan visa grönt ljus i alla riktningar och förlita sig på att det aldrig kommer in flera bilar i korsningen samtidigt.

Ett annat praktiskt tips är att du under testfasen sätter någon begränsning till hur många gånger trådarna utför sina uppgifter. Det kan nämligen vara rätt svårt att avbryta exekveringen när den väl börjat då endast huvudtråden svarar på signalen `SIGINT [Ctrl + C]` samtidigt som den troligtvis är blockerad i väntan på att övriga trådar avslutat sin exekvering. Alternativt så kan du starta dina trådar som daemons (bakgrundstrådar). Då räcker det att huvudtråden avbryts för att hela processen ska avslutas.

För praktiska synkroniseringsexempel rekommenderas Pythonfilerna som finns tillgängliga på lärplattformen.

[2] Windows är i många fall slug i sin schemaläggning av trådarna vilket kan leda till att inga problem uppenbarar sig, att alla lästrådar får köra före skrivtrådarna, eller liknande och varför lite handpåläggning därför krävs för att faktiskt utmana lösningen.

5 Examination

The readers-writers problem är ett klassiskt problem vilket innebär att det finns motsvarande klassiska lösningar på problemet. Kom ihåg att alltid ange dina källor — i kommentarer i koden vid behov. Du kommer under examinationen att få individuellt anpassade frågor om synkroniseringsproblemen och hur du adresserat dem samt om synkroniseringsmekanismerna och hur du implementerat dem. Att presentera en fullt funktionell lösning utan vidare detaljförståelse för lösningen eller teoriförståelse för problematiken kommer att bemötas med kompletteringskrav, eller underkänt betyg i de fall där plagiat är uppenbart. Du förväntas inte återuppfinna hjulet — men du förväntas förstå hur hjulet fungerar.

Redovisa laborationen under ett av laborationstillfällena som ges under kursens gång genom att presentera funktionaliteten och besvara frågor relaterade till ämnet. När du fått klartecken från labbhandledaren att redovisningen är godkänd kan du lämna in i inlämningslådan på lärplattformen där en slutgiltlig bedömning utförs av examinator.

Din inlämning ska bestå av en eller flera Pythonfiler. I de fall där du arbetar med flera filer bör du döpa huvudfilen till `main.py`. Varje pythonfil ska innehålla en header med åtminstone ditt namn och aktuellt datum.

5.1 Bedömning och återkoppling

Uppgiften bedöms med betygen *Godkänd (G)*, *Komplettering (Fx)*, och *Underkänd (U)*. Bedömningen baseras i huvudsak på huruvida fullständig funktionalitet uppnåtts och om du under den muntliga delen av examinationen kunnat förklara innebörden av din kod och de andra punkterna lyfta under Examination, samt om de formella kraven efterlevts.

Återkoppling erhåller du i första hand muntligt under redovisningen. Normalt lämnas ingen skriftlig återkoppling vid godkänt resultat för denna uppgift.

Referenser

[1] V. Tomar, "Python threading and its caveats," jan 2011. [Online]. Available: http://opensourceforu.com/2011/01/python-threading-and-its-caveats/

[2] Python wiki contributors, "GlobalInterpreterLock - Python Wiki." [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock

[3] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Hoboken, N.J.: John Wiley & Sons Inc, 2013, international Student Version.

[4] Python Software Foundation, “threading — higher-level threading interface.” [Online]. Available: https://docs.python.org/2/library/threading.html