Kurs: DVA248 – Datorsystem

Version: 0.2, uppdaterad 2021-04-22

Utvecklad för Python av: Dag Nyström, dag.nystrom@mdh.se

Lab 2: Trådar och synkronisering, fortsättning

Introduktion

Syftet med den här laborationen är att du ska förstå hur pseudoparallellism kan uppnås med multitrådade processer samt hur trådar kan synkroniseras.

Genomförande och redovisning

Laborationen utförs gemensamt av labgrupperna.

Den färdiga koden skickas in via CANVAS efter redovisning för labassistent.

NOTERA: Räkna INTE med att ni kommer att hinna bli klara under labbtillfällena, de flesta behöver lägga tid utanför schemalagd tid.

Uppgift 1 – Think productively….

a)

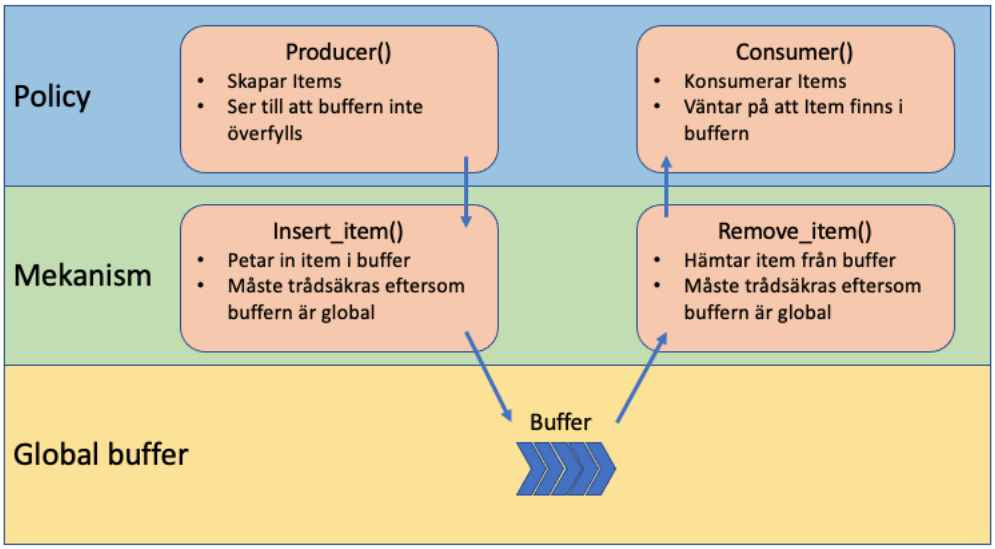
I denna uppgift ska vi titta på ”The Producer Consumer Problem”. Upplägget är enkelt.

Det finns ett antal producenter som producerar *items*.
Vidare finns det ett antal konsumenter som konsumerar dessa *items*.

Producenterna lägger upp sina producerade items i en global buffer.
Konsumenterna hämtar items att konsumera från samma globala buffer.

Bufferns storlek är begränsad. I denna lab är bufferten 5 item stor.

Öppna filen Lab2.py och sätt er in i den för att förstå vad den gör.



Figur 1 Struktur på funktionerna

I Figur 1 kan ni se den övergripande strukturen av filen. Som ni ser är den indelad i tre nivåer där den globala buffern är allra längst ner.

På nivån ovanför finns de mekanismer som krävs för att lägga till och ta bort data ur buffern. Dessa måste vara trådsäkrade eftersom de arbetar med en delad resurs. Den här bufferten är generisk och innehåller ingen som helst logik annat än att den petar in och petar ut items. Den kommer att returnera error om man försöker hämta från en tom buffer eller överfylla den.

På översta nivån har ni själva policyn, dvs två funktioner som producerar items och konsumerar items. Dessa utgår ifrån att det finns en fullt fungerande buffert under implementerar själv policyn att man måste vänta om bufferten är tom eller full.

Slutligen finns en `main`-funktion som har som uppgift att skapa ett antal producenttrådar och ett antal konsumenttrådar (se `numProd` och `numCons`). Själva skapandet är inte implementerat men detta kan ni ju från Lab 1. Main lever sedan i 10 sekunder innan programmet terminerar.

Implementera färdigt main-funktionen. Kom ihåg att skapa deamon-trådar, annars kommer inte programmet att terminera korrekt.

b)

Kör programmet (med en konsument och en producent) och ta en titt på utdatat. Verkar det funka, eller får ni några felmeddelanden??

Troligen ser det ganska bra ut, men ni kan få slumpvis klagomål om att bufferten är överfull eller tom. Vi lämnar det problemet därhän nu och fokuserar på att göra mekanismerna trådsäkra. Detta gör ni på ett liknande sätt som i lab1.

Implementera trådsäkring i insert_item och remove_item.

c)

Öka antalet producenter till 10 och antalet konsumenter till 3 och kör programmet igen. Vad händer? Verkar det funka, eller får ni felmeddelanden.

Troligtvis fick ni en del error om överfull buffert.

Nu är det dags att addressera policyn. Ni vill ju att producenten ska vänta med att skicka iväg om bufferten är full och ligga och vänta på item om bufferten är tom. Kontroll på detta finns ju inte i er buffert utan detta är något som ni måste implementera i consumer och producer

Dags att läsa lite i boken….. Det finns en enkel och elegant lösning på detta som använder semaforer. Leta rätt på ”The Bounded-Buffer Problem” som också kallas ”The Producer Consumer Problem” och läs på hur man löser detta.

I Python finns `threading.Semaphore` som fungerar exakt som Lock fast är räknande.

Implementera ett fungerande producer consumerprogram.

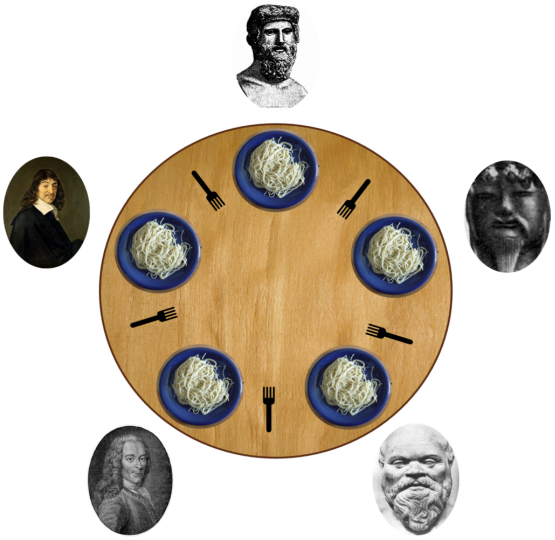
Glöm inte att prova ert program med olika antal producenter och konsumenter, prova till exempel med 3 producenter och 10 konsumenter….. fungerar det fortfarande??

(PS: Det finns faktiskt en till ”bugg” i programmet som egentligen skulle behöva synkroniseras…. Kan ni hitta det?? Ni behöver inte lösa problemet)

Uppgift 2 – … and eat some pasta!

The dining philosophers problem är ett klassiskt datavetenskapligt problem som avhandlar synkronisering av parallella trådar. I korthet går detta ”mycket trovärdiga och realistiska” problem ut på följande:

Kring ett runt bord sitter **fem** filosofer, se Figur 2.
Framför varje filosof finns en tallrik med pasta. (dvs 5 tallrikar)
Mellan varje tallrik finns **en** gaffel. (dvs det finns bara 5 gafflar)



Figur 2 Bordet med gafflar och pasta samt fem kända, men i dagsläget inte helt pigga filosofer.

Dessa fem filosofer kan bara göra två saker; (i) Tänka djupsinniga tankar och (ii) äta pasta. Dock kan de ju naturligtvis inte göra dessa båda saker samtidigt. (För vem kan ju det egentligen??).

Eftersom pasta är omständlig mat att äta så krävs två gafflar.

En enskilds filosofs liv kan med andra ord elegant sammanfattas med följande pseudokod:

```
Philosopher(i):
      While(true):
            Think(10ms)
            Pickup fork i and fork (i+1)%5
            Print(Filosof i har nu plockat upp sina gafflar)
            Eat
            Print(Filosof i är nu mätt)
            Place fork i and fork (i+1)%5
            Print(Filosof i har nu lagt tillbaks sina gafflar)
```

Det är värt att nämna att det alltså blir en sådan snurra (tråd) för varje filosof.

a)

Fundera igenom följande: Det finns 5 trådar som kör ovanstående kod, vilka potentiella problem som kräver någon form av synkronisering kan ni se?

Som ni säkert inser är ju gafflarna delade resurser som antingen ligger på bordet eller är tagen av en filosof. Och delade resurser behöver vi ju på något sätt hantera, eller hur?

Implementera ett program med 5 filosofer med Locks för de olika gafflarna Observera att ert program ska tillåta att så många filosofer som möjligt har möjlighet att äta pasta samtidigt.

b)

Fungerade programmet i a)?? Är ni säkra??

Betänk följande scenario: Alla filosofer råkar bli hungriga samtidigt och plockar upp gaffeln på sin vänstra sida. Nu är alla gafflar tagna men ingen filosof har 2 gafflar och kan börja äta. Katastrofen är ett faktum, filosoferna svälter ihjäl och världen kommer att bli en annorlunda plats.

Detta fenomen kallas för *baklås* (eng. *deadlock*) och är ett reellt problem man måste tänka på när man programmerar med kritiska sektioner och delade resurser. Fler än ett system har havererat p.g.a. baklås genom tiderna…. Det finns många sätt att lösa detta problem. Läs i boken och googla runt på nätet.

Implementera, om nödvändigt, om programmet så att baklås undviks.

Bra jobbat. Dags att redovisa labben!!!
Glöm inte att berätta för labassen hur fantastiska dessa datalogiska problem är. Och att ni naturligtvis tänker spendera hela helgen till att grunna på ett annat realistiskt problem: The sleeping barber problem.