Filosofer

Programmering II - Elixir

Kenan Dizdarevic

21 Februari, 2023

Inledning

Vi skall i denna uppgift implementera beteendet av fem filosofer som sitter runt ett matbord. Problemet är att alla filosofer måste få något att äta när de vill.

Filosoferna har en matpinne till vänster om sig. Detta innebär att det finns matpinnar mellan varje filosof och att det totalt finns fem matpinnar. På bordet är en skål med nudlar utställd. Problemet är att alla filosofer inte kan äta samtidigt, eftersom det endast finns fem matpinnar.

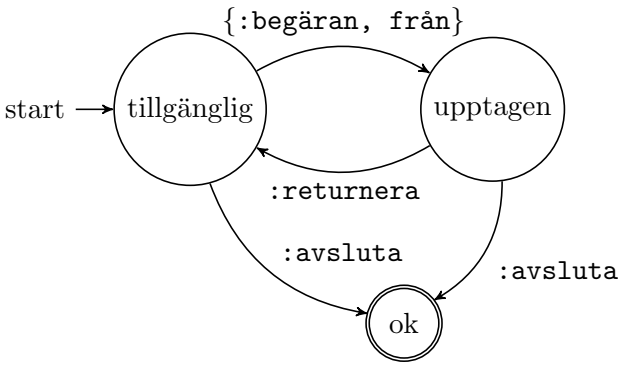
Filosoferna sitter och tänker för det mesta men när de bestämmer sig för att äta kommer de att ta upp en matpinne från vänster och en matpinne från höger. Det tar lite tid för filosofen att äta men när den är klar lägger den tillbaka matpinnarna på deras ursprungsplats.

För att lösa problemet när flera filosofer vill äta samtidigt kommer vi använda oss av *flertrådad programmering*. Det innebär att flera olika beräkningar kan ske samtidigt. De har även möjlighet att kommunicera med varandra för att överföra relevant information.

Implementering

Matpinne

Vi skall representera positionen av en matpinne som en process. Matpinnen har två möjliga tillstånd, antingen är den tillgänglig eller upptagen. Matpinnarna börjar alltid med tillståndet ”tillgänglig”, sedan väntar de på en begäran. När processen tar emot begäran skall den returnera att den har beviljat den samt uppdatera matpinnens tillstånd som upptagen. Om matpinnen returneras från tillståndet där den är upptagen blir den tillgänglig igen. Tillståndsdiagramet för matpinnarna presenteras i figuren nedan.



Figur 1: Tillståndsdiagram för matpinnarna

Denna process skall implementeras i modulen `Chopstick` med en funktion `start/0` som använder sig av `spawn_link/1` för att starta processen. Koden som startar processen ser ut som följande.

```
stick = spawn_link(fn -> available() end)
{:stick, stick}
```

Vi startar en process för en matpinne och låter den börja i tillståndet tillgänglig. Sedan returnerar vi en tupel med matpinnens processidentifierare.

I Elixir har vi möjligheten att skicka meddelanden till processer med funktionen `send/2` och ta emot dem med `receive/1`. Båda dessa funktioner skall nyttjas i `available/0`.

```
def available() do
    receive do
        {:request, from} ->
            send(from, :granted)
            gone()
        :quit ->
            :ok
    end
end
```

När processen tar emot tupeln `:request, from` som är en begäran att ta en matpinne utför vi mönstermatchning med tupeln. Om mönstermatchningen är lyckad skall vi bevilja filosofen att ta matpinnen med `:granted` och uppdatera matpinnens tillstånd till upptagen.

Funktionen `gone/0` utgår ifrån samma principer. Vi tar emot ett meddelande, om meddelandet innehåller `:return` skall vi uppdatera matpinnens tillstånd till tillgänglig. Annars skall vi avsluta på samma vis som vi gör i `available/0`.

Vi vill hålla processernas inre delar dolda från användarna av modulen. Därför skall vi implementera ett gränssnitt så att användaren inte behöver känna till strukturen av meddelandena. Vi skall implementera funktionerna `request/1` som skickar en begäran, `return/1` som returnerar en matpinne och `terminate/1` som avslutar en process. De två sistnämnda funktionerna är enkla att implementera, vi använder oss av `send/2` och skickar med processidentifieraren och meddelandet som antingen är `:return` eller `:quit`. Funktionen `request/1` använder sig av samma princip som de tidigare funktionerna, den använder sig av `send/2` på följande vis:

```
send(stick, {:request, self()})
```

Nuvarande processen skickar ett meddelande till en process med matpinnens processidentifierare. Meddelandet som skickas består av en tupel som innehåller atomen `:request` och processidentifieraren för den nuvarande processen. När vi har skickat meddelandet väntar vi på att ta emot `:granted` för att sedan returnera `:ok`.

Filosofen

En filosof har tre möjliga tillstånd, antingen drömmer filosofen, eller så väntar filosofen på att få en matpinne eller så äter filosofen. I modulen `Philosopher` skall vi implementera funktionen `start/6` som startar en process för en filosof. Funktionen kommer ta emot följande argument:

- `hunger`: hur många gånger filosofen måste äta tills den returnerar `:done` till kontrollprocessen.
- `strength`: filosofens uthållighet, tappar en poäng för varje gång som filosofen inte orkar vänta.
- `right & left`: processidentifierare för matpinnarna som filosofen har åtkomst till.
- `name`: filosofens namn.
- `ctrl`: processidentifierare för kontrollprocessen som skall informeras när filosofen är mätt eller död.

Det enda `start/6` gör är att den startar en ny process som har funktionen `dreaming/6` som ingångspunkt. Vi behöver således implementera funktionerna `dreaming/6`, `eating/6` och `waiting/6` som behandlar filosofernas tillstånd.

`dreaming/6` har tre fall där det första behandlar de filosofer som är mätta. Den returnerar endast `:done` till kontrollprocessen. Det andra fallet behandlar filosoferna som inte har någon styrka kvar, den skickar också `:done` till kontrollprocessen men filosoferna betraktas i detta fall som döda.

Detta fall kommer vi endast kunna använda senare i uppgiften. Det sista fallet behandlar filosoferna som fortfarande deltar i middagen.

```
def dreaming(hunger, strength, left, right, name, ctrl) do
    IO.puts("#{name} is dreaming!")
    sleep(100)
    IO.puts("#{name} is awake!")
    waiting(hunger, strength, left, right, name, ctrl)
end
```

Det enda vi gör är att vi inför en fördröjning vilket beskriver hur länge filosofen skall drömma. Sedan anropar vi `waiting/6`.

Funktionen `waiting/6` skall skicka begäran om matpinnarna med `request/1` som vi tidigare implementerade. Koden för denna funktion ser ut som följande:

```
def waiting(hunger, strength, left, right, name, ctrl) do
    IO.puts("#{name} is waiting for chopsticks!")
    case Chopstick.request(left) do
      :ok ->
        IO.puts("#{name} recieved one chopstick!")
        case Chopstick.request(right) do
          :ok ->
            IO.puts("#{name} is now able to eat!")
            eating(hunger, strength, left, right, name, ctrl)
        end
    end
end
```

I detta fall kan `request/1` endast returnera `:ok`. När filosofen erhåller båda matpinnarna kan den äta, vi anropar således `eating/6`.

Funktionen `eating/6` har precis som `dreaming/6` en artificiell fördröjning som beskriver tiden det tar att äta en portion. Filosofen returnerar även båda matpinnarna med funktionen `return/1` som vi tidigare implementerade. Slutligen anropas `dreaming/6` men där parametern hunger har minskat.

Dödläge

Ett *dödläge* kan uppstå när två eller fler processer väntar på varandra och ingen av dem kan fortsätta utan att den andra fortsätter. Detta leder till att programmet ”låser” sig. Vi lyckades hamna i dödläge när vi exekverade vårt program. Det som händer är att alla filosofer tar matpinnen som finns till vänster om dem. Detta innebär att alla väntar på den högra pinnen men ingen kan äta, eftersom alla pinnar är upptagna. Vi skall nu lösa problemet med dödläget som uppstår.

Det enda vi egentligen behöver göra är att uppdatera funktionen `request/1` till `request/2`. Det andra argumentet kommer vara en tidsgräns för hur

länge en filosof kan vänta på en matpinne. Den nya implementeringen av `request/2` ser ut som följande:

```
def request({:stick, stick}, timeout) when is_number(timeout) do
    send(stick, {:request, self()})
    receive do
      :granted ->
        :ok
    after timeout ->
      :no
    end
end
```

Vi skickar en förfrågan om matpinnen och väntar på meddelandet. Om vi inte får svar innan tidsgränsen är uppnådd returnerar vi endast `:no`.

För att detta skall ha någon funktion måste vi uppdatera `waiting/6` och anpassa den efter `request/2`. För att anpassa funktionen måste vi lägga till de fall där vi får `:no` vid en förfrågan. I det fall där vi har erhållit den vänstra matpinnen och skickar en förfågan om den högra matpinnen skall vi lämna tillbaka den vänstra matpinnen. Detta presenteras i koden nedan:

```
:no ->
    IO.puts("#{name} has aborted wait for chopstick, strength is #{strength}!")
    Chopstick.return(left)
    dreaming(hunger, strength - 1, left, right, name, ctrl)
```

Slutligen anropar vi `dreaming/6` och sänker filosofens styrka. Nu har vi även möjlighet att nyttja det andra fallet av `dreaming/6` där styrkan är 0.

Ett problem som uppstår med denna implementering är att de två sista filosoferna som är kvar kan hamna i ett dödläge. Detta sker eftersom filosofen hinner börja drömma innan den har fått begäran bekräftad.

Asynkron förfrågan

Filosoferna skickar alltid en begäran till den vänstra matpinnen först. Om begäran blir accepterad skickar vi en förfrågan om den högra matpinnen. Vi skall istället skicka en förfrågan till båda matpinnarna.

Vi implementerar funktionen `async/2` som skickar en förfrågan om en matpinne. Argumenten till denna funktion är tupeln som består av processen som är associerad med atomen `:stick`. Det andra argumentet är en referens som möjliggör spårning och matchning av begäran och svar, det är meddelandet som skickas som refereras. Koden för `async/2` ser ut som följande:

```
def async({:stick, stick}, ref) do
    send(stick, {:request, ref, self()})
end
```

Ett meddelande skickas till processen, meddelandet som skickas består av en tupel som har tre element. Det första elementet är `:request` som indikerar att meddelandet är en begäran. Det andra elementet `ref` är en referens som genereras av avsändarprocessen och skickas med för att spåra och matcha begäran och svaret. Det sista elementet `self()` skickar med den processidentiferaren för den nuvarande processen så att mottagarprocessen kan svara till rätt process.

Funktionen `granted/2` används för att vänta på ett svar från en annan process. Funktionen tar två argument en referens och en tidsgräns för hur länge den skall vänta på ett svar. Funktionen använder sig av receivekonstruktionen för att ta emot meddelanden. Den första matchningen som sker är följande:

```
{:granted, ^ref} ->
    :ok
```

Funktionen matchar tupeln som innehåller `:granted` och värdet på referensen som skall matchas exakt. Om mönstermatchningen lyckas skall funktionen returnera `:ok`.

Den andra mönstermatchningen som sker är följande:

```
{:granted, _ref} ->
   granted(ref, timeout)
```

Denna mönstermatchning innebär att en annan process har beviljat en begäran som matchar referensen men inte nödvändigtvis den ursprungliga begäran som funktionen väntar på. Därför sker ett rekursivt anrop på `granted/2` som fortsätter vänta på att den korrekta referensen returneras.

Vi har även implementerat en timeout med `after` som returnerar `:no` om tidsgränsen överskrids.

Servitör

En annan strategi för att lösa problemet med dödläge är att använda sig av en servitör. Servitören reglerar hur många filosofer som kan äta äta samtidigt. En möjlig lösning är att servitören håller koll på alla matpinnarna och låter exempelvis endast två filosofer äta åt gången. Servitören kan implementeras som en separat process som har en intern räknare över hur många filosofer som äter och tillåter filosoferna endast plocka upp matpinnarna om räknaren är under det maximala antalet filosofer som kan äta.

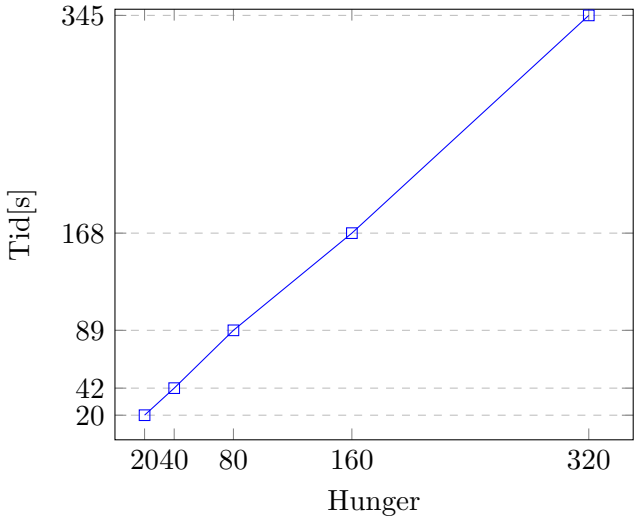
För att undvika att filosofer tappar all styrka och dör kan vi studera andra system som använder sig av flertrådad programmering. Ett sådant exempel är ett operativsystem som hela tiden schemalägger processer och undviker dödläge. Operativsystem kan drabbas av resurssvält vilket innebär att det sker en beräkning där en process ständigt nekas nödvändiga resurser. Detta kan liknas vid filosofen som inte får äta för att den inte har tillgång till

matpinnarna. En algoritm som löser dessa problemen i ett operativsystem är *Round-robin scheduling*, som bygger på *round-robin* principen. Denna princip innebär att resurserna delas jämnt mellan alla processer som väntar på dem. Varje process tilldelas en tid för att använda resursen, när denna tid är över går resurserna vidare till nästa process.

Fördelen med denna princip är att den är enkel att implementera samt att den inte kräver detaljerad information om processernas behov. Principen är dessutom rättvis, eftersom alla processer behandlas lika. Om vi hade implementerat detta med våra filosofer hade programmet varit rättvist, vilket det inte är nu.

Prestandamätning

Vi skall nu utföra prestandamätningar på den asynkrona versionen av vårt program. Vi kommer att öka hungern på filosoferna där vi mäter hur lång tid middagen tar. Mätningen kommer inte att ta i beaktning hur många filosofer som ger upp under middagen.



Figur 2: Middag med hungriga filosofer

Sambandet mellan hunger och tid är linjärt. Det uppstod även några fall där alla filosoferna lyckades dö vilket inte presenteras i grafen ovan. Detta bekräftar vårt antagande om att vårt system inte är rättvist samt att det går att förbättra.

Slutsats

Slutligen ser vi att flertrådad programmering är mycket kraftfullt, framför allt när vi arbetar med distribuerade system. Processer kan köras parallellt

vilket innebär att vi kan nyttja resurserna bättre. Om en process lägger av kommer vårt system att fortsätta köra vidare. Man vill alltid konstruera system på så vis att de klarar av att processer kraschar.

Om vi hade konstruerat våra filosofer som ett sekventiellt program hade det blivit svårläst och komplicerat. Flertrådad programmering förenklar koden och gör det enklare att implementera olika lösningar.