Kurs: DVA248 – Datorsystem

Version: 1.1, uppdaterad 2021-05-06

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

Planetlab – eller hur man får en planet att snurra

Introduktion

Syftet med den här laborationen är att du ska, med hjälp av systemtjänster få en känsla för pseudo-parallellism, ömsesidig uteslutning och client/server arkitekturen. Ni kommer att skapa en applikation där flera processer samverkar. En server, som hanterar ett universum, skall kunna ha flera olika klienter som var och en kan utnyttja serverns service som är:

createPlanet(namn, massa, position, hastighet, liv)

där en planet skickas från en klient till servern för att ritas ut i universum. Servern kommer sedan att räkna på planeternas banor och hur de påverkar varandra och rita ut detta. Om planeter krockar, åker utanför universum eller dör av ålder så kommer servern att skicka meddelande tillbaks till respektive klient och informera om detta. För att undvika att detta blir alltför komplext så kommer universum att modelleras i 2D.

Genomförande och redovisning

Laborationen utförs gemensamt av labgrupperna.

Den färdiga koden skickas in via CANVAS och redovisas därefter för labassistent.

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

Några ord om Python innan vi sätter igång

Syftet med den här laborationen är egentligen att lära sig att använda operativsystemsstöd för

- Skapande av processer och trådar
- Pseudo-parallellism
- Kommunikation mellan processer (IPC)
- Delade resurser

I de flesta programmeringsspråk så brukar man direkt använda sig av operativsystemets tjänster genom att man importerar en operativsystemsspecifik modul till sitt program och får då tillgång till just det aktuella operativsystemets tjänster.

Python är egentligen ett scriptspråk, alternativt ett högnivåspråk, som är tänkt att vara operativsystemsoberoende. För Python är dessutom innebörden av högnivåspråk en produkt med ”extra allt” (eng. all bells and whistles), vilket naturligtvis gör det bekvämt och effektivt att koda i men tyvärr abstraherar sig rätt långt ifrån de tjänster som tillhandahålls av operativsystemet (så som det presenteras på kursens föreläsningar).

Detta får egentligen tre effekter på den här laborationen:

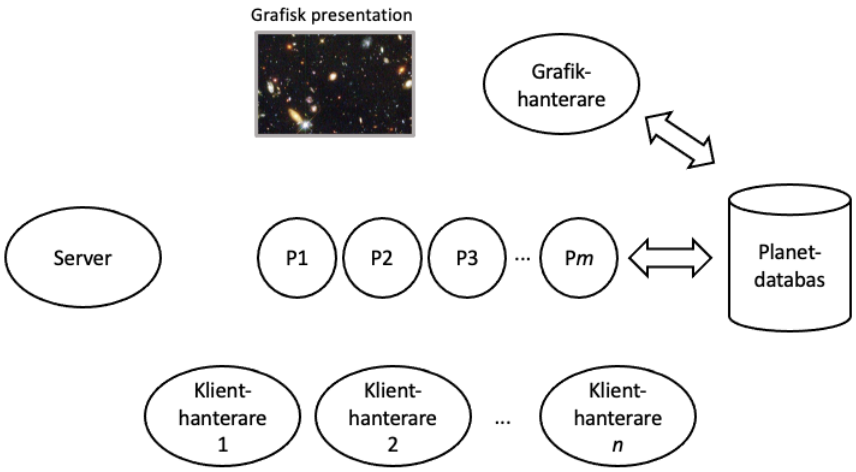
1. Vid skapande av trådar, behöver vi skala ner en del av de funktioner man får på köpet för att emulera operativsystemets beteende.
2. Vid skapande av listor och köer så tillhandahåller Python viss trådsäkring, vi väljer i den här laborationen att helt bortse från det och ni måste själva skapa en ny trådsäkring för er planetlista.
3. Vid kommunikation mellan processer, ger Python en krångligare lösning än många underliggande operativsystemsspecifika mekanismer. I denna laboration använder vi sockets för att kommunicera. För att hjälpa er förstå socket-programmering så finns det ett videoklipp på Canvas som förklarar hur det fungerar och vilka krångligheter som finns.

Min förhoppning är dock att ni, ”trots Python” ska få samma kunskap som om ni kodat detta direkt mot ett operativsystem.

Nu kör vi!!

Servern

Servern kretsar kring en databas som innehåller samtliga aktiva planeter, se Figur 1. En aktiv planet är en planet som (i) har koordinater inom universums kanter, samt (ii) fortfarande har liv kvar. De aktiva planeterna ritas ut i ett fönster av en *grafikhanterare* som är en tråd som med jämna intervall springer igenom hela databasen och ritar ut planeterna med det medföljande grafikpaketet du hittar i filen *space.py* i labbfilerna.



Figur 1: Serverns arkitektur

I andra änden av servern har vi en *servertråd* som ligger och lyssnar efter nya klienter som vill ansluta. När en ny klient ansluter så accepterar servern klienten och en *klienthanterare* startar i en egen tråd. Klienthanteraren ligger och lyssnar på sin tilldelade klient efter nya planeter från klienten. När en klient skickar en ny planet skapar klienthanteraren en planettråd *P* som periodiskt uppdaterar planetens position samt minskar planetens liv. Kod för att beräkna positionen finns i labskelettet. Om planettråden detekterar att planetens koordinater är utanför universums kanter, eller dess liv är slut, tar tråden bort planeten från databasen och meddelar den klient som skapat planeten att den antingen är out of bounds eller har dött av ålder. Servertråden, klienthanterarna samt planettrådarna kommunicerar med klienterna med hjälp av det kommunikationspaket i filen *cscomm.py* som ni utvecklar i uppgift 3 nedan.

Notera att det inte finns någon strikt tidsmässig synkronisering mellan olika planeter samt mellan planeter och grafikhanteraren utan planettrådarna uppdaterar sin planet och sover sedan en liten stund. Däremot måste databasen skyddas mot att bli korrupt när flera trådar accessar den samtidigt.

Klienten

Klienten skall vara ett terminalprogram (textbaserat program) som tar planetdata som input från användaren och fyller i ett *planet* objekt som ni finner i filen *space.py* och skickar över denna till servern. Därefter skapar klienten en tråd som ligger och lyssnar på meddelande om planetens död eller om den blivit out of bounds. Användare skall kunna mata in flera planeter från samma klient. Det skall vara möjligt att ha flera klienter uppkopplade mot samma server.

Uppgift 1 – Bekanta er med serverns filer

Börja med att köra, och bekanta er med filen *server.py*. När ni kör filen, ska ett fönster öppnas och röda prickar börja ritas ut i ett slumpmässigt mönster.

I main-funktionen så skapas en tom rymd, `space`, varefter en tråd som slumpvis ritar planeter i rymden startas. Notera flaggan `daemon=True` som ser till att tråden beter sig som en vanlig ”operativsystemstråd”, dvs när main-tråden terminerar så dör tråden.

Slutligen körs fönstrets hanteringsloop som bryts när fönstret stängs.

I filen finns även klassen `universe` som hanterar listan av planeter samt beräkar positioner för planeter.

När ni känner att ni förstår vad som händer i servern så gå vidare till nästa uppgift

Uppgift 2 – Skapa kommunikationsstubbar[1] i klient och server

Öppna ytterligare en ny editor och skapa en ny fil *client.py* i samma katalog.

I klienten lägger ni nu till raderna:

```
from planet import planet
from cscomm import clientInitSocket,clientRecvString,clientSendPlanet

p=planet(”Sun”,300,300,0,0,10e8,10e8)
s=clientInitSocket()
clientSendPlanet(s,p)
s.close()
```

Ignorera att ni får en varning och att koden inte går att köra.

Nu har klienten en kommunikationsstubbe för att kunna koppla upp sig mot servern med en socket och skicka en planet ”Sun” till servern.

Motsvarande i servern lägger ni till (före skapandet av paintertråden) raderna:

```
serverSocket=serverInitSocket()
clientSocket=serverWaitForNewClient(serverSocket)
p=serverRecvPlanet(clientSocket)
```

Denna stubbe skapar en socket för servern att lyssna på och sedan, när klienten kopplat upp sig, skapar den en klientsocket för att kommunicera med klienten. Slutligen tar den emot en planet *p*.

Nu är klienten och servern redo att koppla upp sig och skicka en planet… om nu bara själva kommunikationspaketet vore implementerat.

[1] En stubbe är en testimplementation för att testa något.

Uppgift 3 - Kommunikationspaket med sockets *cscomm.py*

När olika trådar kommunicerar inom en process så delar de ju samma minnesrymd, vilket gör kommunikation enklare. När olika processer, med egna adressrymder, kommunicerar med varandra så blir det krångligare och man måste ta hjälp av det underliggande operativsystemet. Sockets är en sådan mekanism som hanterar kommunikation mellan olika processer. Sockets är en kommunikation på relativt låg nivå där en hel del hantering och kontroller måste ske av den som implementerar.

För att enkelt ge er en introduktion till hur sockets fungerar i Python så finns det en länk till en kort film upplagd på Canvas. Den är ca 10 minuter lång och jag rekommenderar att ni tittar igenom den innan ni går vidare.

Filen *cscomm.py* använder modulerna `socket` och `pickle` för att kunna skapa client/server uppkopplingar samt för att överföra planeter och text via sockets.

Det finns mycket resurser online med information om hur sockets implementeras i python.

Implementera funktionerna i *cscomm.py* och verifiera att de fungerar genom att först starta servern, och sedan starta klienten.

När ni startar servern skall inget ritas ut i universum. Sedan när klienten startas skall variabeln *p* i servern innehålla en planet och servern skall återgå till att slumpvis rita ut punkter. Ni kan även prova att lägga till att servern skickar tillbaks en sträng till klienten. När allt detta fungerar så borde kommunikationen vara klar.

Planetstrukturen hittar ni i filen *planet.py*.

Uppgift 4 – Ta emot flera planeter från olika klienter

Nu är det dags att sätta ihop detta till ett fungerande system, dvs ta bort stubbarna och implementera på riktigt.

Utöka funktionaliteten med en servertråd som kontinuerligt ligger och lyssnar efter nya klienter som vill koppla upp sig (på serverSocket).

När en ny klient anropar skall servertråden skapa en klienthanterare för denna klient, dvs en ny tråd som kontinuerligt ligger och väntar på att klienten ska skicka planeter via clientsock.

Dessa planeter ska läggas in i planetdatabasen (universum) samt få var sin planettråd som kontinuerligt uppdaterar positionen för planeten med *calculate_planet_pos()*.

NOT1: Lägg in en kort sleep i klienten mellan att ni skickar planet 1 och planet 2. Skickar ni båda direkt efter varandra finns en risk att servern tar emot båda planeterna i ett meddelande (se förklaring i filmklippet ovan). Som en extrauppgift kan ni implementera stöd i kommunikationspaketet att detektera om flera planeter finns i kö.

Testa att skapa följande planeter:

Namn	Massa	Position x	Position y	Fart x	Fart y	Liv
Sun	10^8	300	300	0	0	10^8
Earth	1000	200	300	0	0.008	10^8

I detta universum skall jorden röra sig i en ellips runt solen.

Uppgift 5 – Trådsäkert universum

Ert universum används nu av flera trådar (klienthanterare, planettrådar samt grafikhanteraren). Se till att alla accesser synkroniseras och se till att de kritiska sektionerna blir så små som möjligt.

Uppgift 6 – Skicka dödsmeddelanden

Utöka funktionaliteten i planettråden så att den detekterar om planeten har dött av ålder eller om den är utanför rymdfönstret. Ta i så fall bort planeten och skicka meddelande till klienten enligt följande_

”Planet *xxx* har dött av ålder” alternativt ”Planet *yyy* lämnade det kända universum” skickas från servern till rätt klient.

Utöka klienten så att användaren kan mata in hur många planeter som helst och så att dödsmeddelanden skrivs ut (bara i klienten som skapade planeten).

Kontrollera att det är möjligt att ha flera klienter uppkopplade och att rätt klient får dödsmeddelanden.

NOTERA: Socketen `csock` i planet bara används av servern och skall INTE skickas över mellan server och klient. Använd istället den clientSock ni får i servern när en ny klient kopplat upp sig.

Ni kan testa ert program enligt följande:

1. Starta en server samt två klienter (K1 och K2)
2. Låt K1 skicka följande 2 planeter (samma som ovan)

Namn	Massa	Position x	Position y	Fart x	Fart y	Liv
Sun	10^8	300	300	0	0	10^8
Earth	1000	200	300	0	0.008	10^8

3. Låt K1 skicka följande planet:

Namn	Massa	Position x	Position y	Fart x	Fart y	Liv
Comet	1000	500	300	0.1	0	10^8

   När kometen lämnar universum skall K1 få dödsmeddelande
4. Låt K2 skicka följande planet

Namn	Massa	Position x	Position y	Fart x	Fart y	Liv
Oops	10^9	500	400	0	0	100

5. Nu skall bara K2 få dödsmeddelandet

Fungerar det?? Grattis, ni har då skapat ert universum.