– Lab2 –
Systemprogrammering i Windows –
eller hur man får en planet att snurra

1 Introduktion

Syftet med denna laboration är dels att Ni ska få prova på att använda de systemtjänster som är vanliga i operativsystem (OS), men framförallt att få en känsla för pseudo-parallellism, ömsesidig uteslutning och client-server begreppet. I laborationen kommer operativsystemet Windows att användas.

Skillnaden i funktionalitet mellan olika OS är oftast inte så stor, men sättet som man använder dem på kan skilja sig ganska mycket. (Den som känner till lite om systemanropen i Unix kommer nog bli konfunderad till en början.)

Vi hoppas att denna laboration ska ge Er större förståelse för hur tjänsterna i de kommande laborationerna är tänkta att fungera, samt ge konkreta exempel på det material som gåtts igenom på föreläsningarna. Den funktionalitet som kommer att behandlas i denna laboration är:

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

Tid till förfogande

Ni arbetar givetvis i er egen takt, men det rekommenderas att ni blir klara med denna uppgift under labtillfälle tre ("Planetlab II").

Räkna aldrig att med att hinna göra klart laborationerna under schemalagd labtid!

Allmänt

Under kursen gång kommer det att läggas upp tips, rättelser, och förklaringar till vanligt förekommande problem på kursens hemsida. Titta därför alltid på kursens hemsida först innan ni frågar.

2 Uppgift

I denna laboration ska ni skapa en simuleringsmiljö för ett planetsystem. Detta system ska implementeras enligt client-server modellen, där servern ska erbjuda följande tjänst till dess klienter:

createPlanet(name, mass, position, velocity, life)

där *name* är namnet på den planet som ska skapas, *mass* planetens massa, *position* dess start position, *velocity* är planetens ursprungshastighet uttryckt som en vektor, och *life* är planetens livslängd uttryckt i tidsenheter. (För att inte göra saken alltför komplex kommer vi att arbeta i 2-D, således blir det "x och y" koordinat för en planets position och dess hastighet.)

Servern ska ha en databas (länkad lista) med *aktiva* planeter ($life > 0$), och för varje planet ska servern skapa en tråd. Denna tråd har till uppgift att beräkna planetens nästa position genom

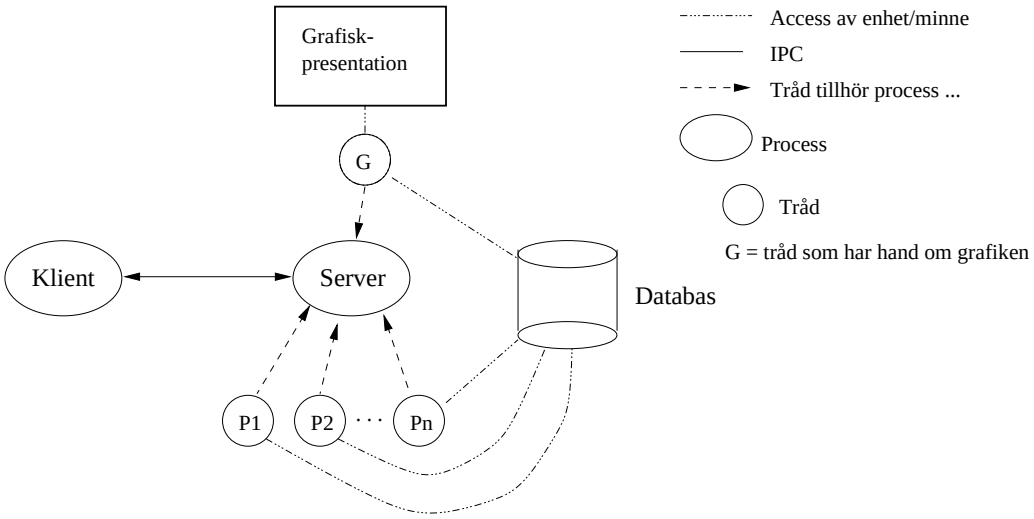

Figure 1: Kopplingen mellan de olika delarna i laborationen

att konsultera serverns databas. (Planetens nästa position beror på dess hastighet samt övriga planeters positioner och massor.) Efter det att planeten har beräknat sin nya position minskas livslängden med ett, vartefter planeten uppdaterar serverns databas med dess nya position, livslängd etc. Figur 1 visar hur de olika delarna i laborationen är tänkta att vara sammankopplade.

Det finns inget strikt tidskrav på att varje planet/tråd arbetar helt i fas med de övriga, d.v.s. det behövs ingen synkroniseringen mellan olika planeter/trådar. Snarare är kravet att någon sådan synkronisering inte får finnas! (När tråden/planeten beräknat en ny position och uppdaterat databasen, ska tråden sova en viss stund.)

Här nedan följer en härledning av vad som bestämmer planeters positioner. Denna härledning är inte något ni måste kunna, och de som vill kan hoppa direkt fram till resultatet.

Härledning: De fysikaliska lagar som bestämmer planeters positioner och hastigheter är (enligt den Newtonska-modellen):

$$F_s = G\frac{m_1 m_2}{r^2} \qquad F = m \cdot a$$

$$v = \int a(t)\ dt \qquad s = \int v(t)\ dt$$

$$G = 6.67259 \times 10^{-11} N \cdot m^2/kg^2$$

där s betecknar sträcka, v hastighet, och a acceleration.

Exempel: I figur 2 visas hur gravitationskrafterna påverkar två planeter. Med hjälp av formlerna ovan går det att härleda vilken acceleration detta kommer att medföra på en planet. Om planetens massa, position och hastighet är given kan även planetens nya position härledas. Vi utför detta för planet 1 (P1 i figur 2):

$$F_s = G\frac{m_1 m_2}{r^2} \qquad F_s = m_1 \cdot a_1$$

$$m_1 \cdot a_1 = G\frac{m_1 m_2}{r^2} \Rightarrow a_1 = G\frac{m_2}{r^2} \tag{1}$$

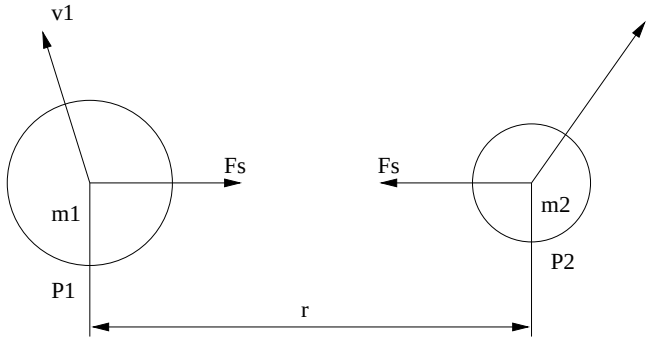

Figure 2: Attraktionskraften mellan två himlakroppar

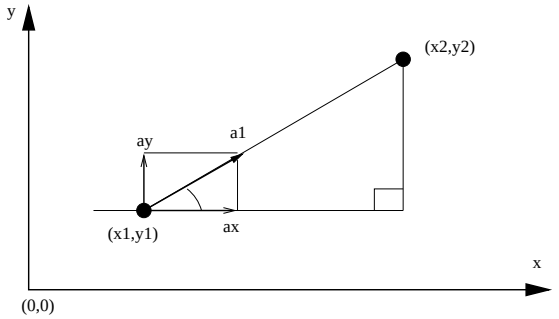

Figure 3: Uppdelning av kraftkomposanten F_s

där r beräknas enligt följande om de två planeterna har positionerna (x_1, y_1) respektive (x_2, y_2), se figur 3.

$$r = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

Dessa lagar är utformade i det endimensionella planet. Vi måste därför hitta ett sätt att överföra vårt tvådimensionella problem till en form så att de endimensionella lagarna kan användas.

Denna omvandling görs genom att beräkna accelerationen för var och en av de två dimensionerna, a_x och a_y i figur 3.

$$cos(\alpha) = \frac{x_2 - x_1}{r} \qquad a_x = a_1 \frac{x_2 - x_1}{r} \tag{3}$$

$$sin(\alpha) = \frac{y_2 - y_1}{r} \qquad a_y = a_1 \frac{y_2 - y_1}{r} \tag{4}$$

När vi har dessa kan vi behandla varje dimension för sig. Positionerna för varje dimension bestäms med följande formel:

$$s_x = \int v_x(t)\, dt = \int \int a_x(t)\, dt$$

Om vi approximerar integreringen med summation och utför följande beräkning för små värden på dt, får vi:

$$v_x^{new} = v_x^{old} + a_x \cdot dt \tag{5}$$

$$s_x^{new} = s_x^{old} + v_x^{new} \cdot dt \tag{6}$$

där s_x^{new} är planetens nya position, och s_x^{old} dess gamla position. Motsvarande i y-led måste också utföras.

Detta är allt vi behöver! Kom ihåg att desto större värde på dt som väljs desto större fel får man. (I laborationen krävs inte någon enorm precision...)

Resultat: Här nedan följer pseudo-koden för att beräkna en planets position när det finns flera andra himlakroppar i närheten, d.v.s. den kod som planettråden ska köra:

1. $a_{tot_x} = 0$, $a_{tot_y} = 0$ (det totala accelerationsbidraget i varje dimension)
2. För varje planet (γ) i databasen utom den egna (δ):
   (a) Bestäm accelerationbidrag från γ på δ, d.v.s. beräkna (1). (Ekvation (2) behövs för att lösa (1).)
   (b) Beräkna acceleration för varje dimension enligt ekvation (3) och (4).
   (c) Addera resultat från ovan till a_{tot_x} resp. a_{tot_y}.
3. Från ovanstående steg får vi ut totalt accelerationsbidrag i x- och y-led.

   För varje dimension (x och y) beräknas planetens nästa position med hjälp av ekvation (6), och för att lösa den behövs (5). (För att kunna beräkna denna krävs att nuvarande position och hastighet finns lagrade.)
4. Låt tråden sova en stund. Lämplig tid är $10ms$. (OBS! Det finns två olika funktioner för att låta en tråd ”sova”: sleep och Sleep. sleep sover i givet antal sekunder, Sleep sover i angivet antal millisekunder. Använd Sleep! (Sleep tillhör Win32 APIet, det gör inte sleep.))

   Gå till steg 1 och upprepa hela proceduren igen (en evig loop).

□

Det finns två saker som kan göra att en planet tas bort ur systemet:

- planetens livslängd går ut ($life = 0$)
- planetens position är utanför en *bounding box*. Lämpligen sätts bounding box till att ha storleken 800×600.

När en planet tas bort från systemet ska detta meddelas till den klient som skapade planeten, samt orsaken till varför planeten togs bort. Klienten ska sedan presentera denna information på lämpligt sätt. (*Hint:* Klienternas mailslots måste ha unika namn. För att skapa unika klientnamn är funktionen *GetCurrentProcessId eller GetCurrentThreadId* lämplig.)

Servern ska vara så pass generell att den klarar av att hantera flera klienter samtidigt. Servern har även som uppgift att rita ut planeternas positioner i ett fönster med storleken $aaa \times bbb$. För att kunna utföra detta tillhandahåller vi ett antal funktioner (så att ni slipper grotta ner er i hur grafik och fönster hanteras i Windows, detta kommer i lab 3). *Dessutom* ska varje klient kunna ha flera aktiva planeter igång samtidigt hos servern. Detta gör att det finnas ett behov av att låta även klienten ha fler än en tråd. (*Hint:* Fler än två är oftast onödigt.)

Den som vill får givetvis implementera sina egna grafikfunktioner. För laborationen räcker det att representera en planet som en pixel i presentationsfönstret. Helst ska de olika planeterna ha olika färger när de presenteras. (Det är bra om planeterna lämnar ”spår” efter sig, så att man ser hur planetbanan ser ut. Detta är det sätt vi föreslår till presentationen, men det finns inget som hindrar andra lösningar.)

3 Labmiljö

Laborationen ska utföras i Windows och förslagsvis används Visual C++ som utvecklingsmiljö.

ifrån för att skapa klienten och servern i laborationen. Klienten är en ”vanlig” applikation som ska köras i MS-DOS prompt (se till att ni väljer rätt vid kompileringen), medans servern använder sig av ett fönster för att presentera grafik. Koden finns även tillgänglig i maskinläsbar form på kurshemsidan. Det första ni bör göra är att **provköra denna kod**, och lista ut varför programmen beter sig som de gör. (Starta först servern och sedan klienten.) När ni väl förstår hur dessa program fungerar kan ni börja ändra i denna kod för att skapa lösningen på labben. Ni kommer märka att det mesta av koden kan vara kvar. (En normal lösning på denna laboration kräver att ni lägger till ungefär 200 rader kod till labskelettet.)

OBS! Filerna som ni hämtat måste läggas in er Visual C++ Workspace (Solution) i två nya projekt.

OBS! 2 För att ni skall kunna köra koden ni hämtat måste ni inkludera era wrappers som ni gjorde i laboration 2 i projektet. Får ni inte under några omständigheter programmet att fungera kan ni börja fundera på om det möjligtvis finns något fel i era wrappers.

4 Debuggning

I denna laboration kommer ni oftast inte att kunna använda debuggern för att leta efter fel, eftersom stödet för debuggning av multitrådade applikationer är begränsat. Istället rekommenderar vi att ni gör spårutskrifter med hjälp av funktionen *MessageBox* i servern. En av fördelarna med denna funktion är att den stoppar den tråd som anropat funktionen tills det att meddelandefönstret stängs. Exempel på hur funktionen används finns i Appendix.

Funktionen *MessageBox* tar bland annat en parameter med fönstertitel och en sträng som ska visas i fönstret. För att skapa strängar med värden på variabler används lämpligen funktionen *sprintf*.

Exempel:

```
int i = 267;
char str[64];

sprintf(str, "Värdet på i är %d", i);
```

5 Testning

Det kan vara svårt att veta vad som är lämpliga testvärden för denna laboration. Därför tillhandahåller vi ett exempel som placerar en planet i en cirkulär bana runt en annan. Sätt *dt* till 10.

	m	x	y	v_x	v_y
P_1	10^8	300	300	0	0
P_2	1000	200	300	0	0.008

Om Ert program fungerar som det ska kommer planeten P_2 cirkulera runt P_1. P_1 kommer nästan stå helt still. Efter detta test är det fritt fram att skapa de mest verkliga och overkliga planetbanor!

Skulle det vara så att matematiken i denna uppgift ställer till stora problem för er. Be då att labassistenten hjälper till och förklarar. Det är inte meningen att ni ska slösa en massa tid på

detta, men lite tid kommer det alltid att ta...

6 Redovisning

Laborationen redovisas med en demonstration av Ert program för labassistenten under labtid. Vid denna redovisning skall **designen** finnas till hands, **färdigskriven**. Ur designen skall det klart och tydligt vilka trådar/processer som finns i systemet, vilka datastrukturer dessa kommer åt, hur dessa operationer synkroniseras, samt hur kommunikationen går till mellan trådar/processer.

Efter godkänd demonstration skall **koden** och **designen** laddas upp i WebCT för granskning. Givetvis ska koden vara kommenterad, indenterad, samt väl strukturerad för att godkänt skall ges.

Dessutom ska ni skicka in en **rapport** med svar på följande frågor:

- Hur är det möjligt att köra flera trådar/processer på en processor? Förklara.
- Vad händer om man lägger till ytterligare en tråd i servern, med högre prioritet än övriga trådar, och denna nya tråd kör en oändlig loop utan sleep? Hur påverkas övriga processer i systemet? Varför blir det som det blir?
- I er lösning på denna uppgift finns det kritiska sektioner som ni säkert har skyddat med *Critical Sections*. Vad skulle kunna hända om dessa inte används? Ge en sekvens av händelser som resulterar i ett allvarligt fel.
- I Windows så fördelas CPU-tid på trådbasis. Däremot i UNIX så fördelas CPU-tid först på processbasis och sedan på trådbasis inom varje process. Antag att du inte tror att påståendet för Windows är sant! Hur skulle du kunna verifiera detta? Beskriv hur du skulle kunna göra detta experiment (teoretiskt).

Vi vill även att ni skriver ned problem som ni hade vid genomförandet av denna laboration, och bifogar detta i ovanstående mail. (Denna information kommer att användas för att förbättra laborationen till nästa år.)

7 Tips

- Börja med att reda ut begrepp och se till att ni verkligen förstått vad som skall göras. Bästa sättet att verifiera detta är genom att göra en design och be laborationsassistenten om feedback på denna.
- Gör inte allt på en gång. Ett tips är att skriva en fristående applikation som simulerar planeternas rörelser och skriver ut dessa. Kopiera sedan koden till den ”riktiga” simulatorn, lägg till grafikstöd och testa igen. Därefter kan det vara ide att sikta in sig på kommunikationen.
- Visual C++ spottar ur sig väldigt stora filer under kompilering och länkning. Var uppmärksam på quotan, det är möjligt att kopiera källkoden till den lokala hårddisken och köra därifrån. Glöm inte att rensa efteråt!!! *(Glöm inte bort att kopiera tillbaka filerna till er hemkatalog innan ni rensar.)*

A Systemanrop i Windows

Detta appendix ger kort information om några av de systemanrop som Windows tillhandahåller, genom det så kallade Win32 APIet. Mer detaljerad information om dessa anrop finns tillgängliga på MSDN. Eftersom alla detaljer om dessa systemanrop finns i hjälpen kommer vi här endast kort nämna vissa anrop, och sedan ge ett exempel hur dessa kan användas.

I Win32 används följande funktioner för att hantera ömsesidig uteslutning och semaforer:

1. Ömsesidig uteslutning inom en och samma process: *InitializeCriticalSection*, *EnterCriticalSection*, *LeaveCriticalSection*, *TryEnterCriticalSection*.

   Exempel:

   ```
   Tag ett stycke global variabel, t.ex. 'CRITICAL_SECTION foo;'.
   Initiera sektionen: 'InitializeCriticalSection(&foo);' (görs en gång)
   Gå in i kritisk sektion: 'EnterCriticalSection(&foo);'
   Gå ur kritisk sektion: 'LeaveCriticalSection(&foo);'
   ```

2. Ömsesidig uteslutning mellan processer: *OpenMutex*, *CreateMutex*, *ReleaseMutex*, *WaitForSingleObject*, *CloseHandle*.

3. Semaforer som kan delas mellan processer: *OpenSemaphore*, *CreateSemafore*, *ReleaseSemaphore*, *WaitForSingleObject*, *CloseHandle*.

För att ändra schemaläggningsprioriteten på processer och trådar används: *SetThreadPriority*, *GetThreadPriority*, *SetThreadPriorityBoost*, *GetThreadPriorityBoost*, *GetProcessPriorityBoost*, *SetProcessPriorityBoost*, SetPriorityClass, och GetPriorityClass. (Prioriteten sätts vid skapandet av processer, men kan förändras med dessa funktioner.)

Se MSDN för mer information om systemanrop i Windows!