Introduktion

Det första du behöver veta om PintOS är att det är ett riktigt operativsystem, om än med mycket begränsad funktionalitet. Det är inte ett lab-skelett i vanlig mening (även om jag då och då försöker förtydliga kod och kommentarer). Varje rad i PintOS har ett funktionellt syfte även i versionen du startar med, och alla variabler används till något.

Ofta får jag, när jag ber studenter motivera en lösning, kommentaren att “det såg ut som det var förberett för vår kod här”. När jag sedan frågar hur de funktioner studenterna modifierat och de variabler de återanvänt används av PintOS har de ingen aning. De har alltså lagt ned massor av tid och jobb på en lösning som med all sannolikhet saboterar den befintliga koden så att något annat går sönder. Vilket kostar dem ännu mer tid i felsökning. Det går inte att ta en snabb titt på koden och dra slutsatsen “det är nog här vi skall lägga till vår kod”. Ta istället en längre titt på koden, ta reda på hur den fungerar och används, och utnyttja den sedan på rätt sätt till din lösning. Det finns en specifik lösning som det är tänkt att ni skall kunna komma på och implementera inom rimlig tid, men tänk på att det finns många andra lösningar som fungerar, som är rätt, och som alltså godkänns. Smarta genomtänkta lösningar välkomnas. De använder ofta lite mer avancerad kod, men i gengäld mindre mängd kod. Spendera hellre 4h att planera en lösning och 1h att implementera den än tvärtom, det lönar sig.

Meningen med laborationerna är att du skall komplettera PintOS med kod som gör att en uppsättning vanliga systemanrop fungerar, samt korrigera en del kod som är avsiktligt dåligt implementerad (den implementation som finns från start är på vissa punkter endast till för att det skall gå att alls starta PintOS). Den kod du skriver kommer till större delen att vara rena tillägg till den befintliga koden, och måste samarbeta med resten av systemet. För att din kod skall samarbeta bra med resten av systemet krävs att du förstår hur de centrala modulerna i PintOS fungerar. Dessa finns beskrivna i Stanfords originaldokumentationen till PintOS, Appendix A, Reference Guide. Appendix A är rekommenderad läsning. Även Introduction, Project 1 och Project 2 innehåller bitar du kan ha nytta av. De viktigaste funktionerna finns även beskrivna i detta dokument.

Genom åren har vi naturligtvis haft många studenter som ignorerat allt vad dokumentation heter och gissat sig fram till en lösning. Gemensamt för dem är att de fått lägga ned mycket tid på buggar och felsökning eller ännu inte är klara. Om du använder befintliga funktioner fel, eller inte försår hur t.ex. trådsystemet fungerar, blir det ofta jobbiga, tidskrävande fel. Gissa inte.

PintOS är skrivet modulärt. Varje fil innehåller funktioner som implementerar ett delsystem, och koden är relativt välkommenterad. Ytterst är det kommentarerna och koden som är den fullständigt korrekta dokumentationen av hur det verkligen fungerar. Bläddra i koden för att hitta de funktioner som du behöver för att lösa uppgifterna, och läs på hur den kod du tänker dig att använda fungerar. Min målsättning är att allt du behöver skall finnas beskrivet i denna instruktion till slut, men dit har jag ännu inte kommit. Du kan behöva komplettera med att läsa i Stanfords dokumentation, eller direkt i koden. Är något oklart skall du dokumentera det. Ett formulär för utvärdering av deluppgifter och tidrapportering finns i detta dokument.

Tillåtna och otillåtna lösningar

Som vanligt när du löser laborationer finns det lösningar som inte godkänns. Det är självklart lösningar som kan leda till fel resultat, lösningar som är gravt ineffektiva eller klumpiga, lösningar som läcker minne, samt lösningar som på ett eller annat sätt inte uppfyller god programmeringssed. I PintOS tillåter vi dock att du gör en del lösningar som inte skulle vara godtagbara i ett riktigt system. Anledningen till detta är att det är första gången du kodar ett operativsystem, du är ovan att programmera i C, samt att vi har begränsat med tid. Därför håller vi oss till enklare lösningar:

- Begränsad storlek på datastrukturer: Du behöver hålla reda på öppna filer och startade processer m.m., det är då OK att sätta en övre gräns på hur många, vilket gör att enkla arrayer kan användas. Det skall dock vara lätt att öka eller minska begränsningen vi omkompilering.
- Globala variabler: Operativsystemet kontrollerar systemets globala resurser, och måste ofta använda globala datastruktuer för att hålla reda på dem. Du får alltså lägga datastrukturer globalt, men du måste kunna motivera det, och det måste fungera med godtyckligt antal aktiva trådar. Det är mycket viktigt att initiera dessa rätt.

Några “nya” saker som inte är acceptabla:

- Stänga av interrupts är inte tillåtet: Du skall lösa synkronisering med hjälp av lås och semaforer. De tar i sin tur hand om att stänga av och sätta på interrupts på rätt sätt. Det är bland annat därför dessa abstraktioner finns. Den enda gången lås inte kan användas är då synkronisering skall ske mellan en tråd och ett externt (asynkront) interrupt. Du kommer inte råka ut för någon sådan situation, men du kan ju fundera på varför lås inte fungerar i det fallet.
- Busy-wait är inte tillåtet. Då en tråd behöver vänta på att något villkor skall uppfyllas skall detta lösas genom att tråden placeras på en vänte-kö och låter andra trådar exekveras under tiden. Semaforer och Conditions existerar för att utföra detta på rätt sätt. Använd dem.
- Synkroniseringsproblem och “races” är inte tillåtna. Om det går att konstruera två exekverings-sekvenser (via trådbyten), oavsett hur långsökta, där samma uppgift under samma initiala förutsättningar kan leda till två olika resultat så är koden felaktig.
- Minnesläckor är absolut förbjudna. Använder du *malloc* eller *palloc_get_page* för att reservera minne är det absolut nödvändigt att du använder *free* resp. *palloc_free_page* när du inte behöver minnet mera. Läcker en process minne kan (skall!) operativsystemet ta tillbaka minnet när processen avslutar. Men om operativsystemet läcker minne slutar det med att datorn måste startas om dagligen eller oftare för att få tillbaka minnet. Det är inte acceptabelt.

Känner du dig osäker på om något är acceptabelt eller inte, rådgör med assistent i *förväg*.

Oavsett vilken lösning du väljer skall slutresultatet vara att alla systemanrop fungerar stabilt under en längre tids körning. Det som räknas från kursens sida är inte bara att eventuella test fungerar, utan även att koden är skriven på ett tydligt och korrekt sätt. Testfallen kan fungera trots gravt felaktig kod. Korrekt kod kommer alltid klara alla testfall. Du skall lösa problemet, skriva kod och sedan testa. Om du börjar med att titta på testfallen och sedan skriver kod som löser just dem blir det nästan säkert så att din kod inte löser problemet.

Missuppfattningar och misstag vid kodning i C

Många av de problem föregående års studenter råkat ut för härstammar från misstag och bristande förståelse av programspråket C. Det vanligaste tankesättet är “det kompilerar, alltså är det rätt” eller “det fungerar, alltså är det rätt”. Tyvärr är inget av detta sant, och många gånger blir den upptäckten som en kniv i ryggen efter många timmars felsökning.

Missuppfattningen “det kompilerar, alltså är det rätt“

C är ett språk som tillåter dig som programmerare att göra i princip vilka knasigheter som helst utan att generera fel vid kompilering. Om du som programmerare skrivit koden implicit (t.ex. en implicit pekaromvandling) så genererar kompilatorn en varning (om du har tur), men det är i regel allt du kan hoppas på. Skriver du koden explicit, t.ex. med explicita (kanske felaktiga) typomvandlingar får du inte ens en varning då kompilatorn utgår från att du vet vad du gör. Du är **GUD** över koden och därmed kompilatorn. Du bestämmer vad som är rätt. Är du minsta osäker på någon detalj gäller det alltså att kontrollera hur det skall vara. Bättre spendera en timme med att ta reda på hur det skall vara, än att senare spendera åtta timmar med att felsöka konstiga fel. Det gäller även att vara mycket uppmärksam på kompileringsvarningar. Ofta är de oviktiga, men lika ofta signalerar de direkta felaktigeter. Grundregeln är att om du inte förstår varför en varning uppstår skall du ta reda på det och rätta koden. Rekommendationen är att koda bort alla varningar. Annars är det lätt hänt att missa en viktig varning bland en lång lista på oviktiga.

Missuppfattningen “det fungerar, alltså är det rätt“

Att något fungerar betyder faktiskt bara att det fungerar för precis det du testat. Så om du inte testar **ALLA** (vilket som regel är omöjligt) möjliga trådbytessekvenser och indatasekvenser kan du inte veta om något är rätt genom att bara testa.

När du skriver vanliga program fungerar operativsystemet i viss mån som en sandlåda, men tillåter fortfarande att programmet gör fel internt. Du har inte tillgång till privilegierade instruktioner. Du kan inte läsa och skriva var du vill i minnet. Om du gör grova fel med någon pekare får du “segmentation fault”. Men gör du bara “små” (men allvarliga) fel, t.ex. indexerar utanför en array eller använder avallokerat minne så kommer programmet verka fungera tills minnet du använt fel (t.ex. precis efter arrayen) används till det som det är avsett för. Symtomen på att det är fel kan alltså visa sig långt senare trots att det ser ut fungera till en början.

I PintOS skriver du operativsystemet. Där har du tillgång till allt (existerande) minne. Skriver du till “fel” adress är alltså chansen större att allt kommer verka fungera. Tills det som den adressen egentligen används till behövs, vilket sker några dagar senare när du börjar testa andra saker. Om du inte helt förstår hur koden du skriver samarbetar med resten av PintOS är risken att allt till en början verkar fungera. Men ett par dagar senare, när du skrivit flera dussin nya rader och startar ett nytt testfall, då krashar PintOS. Och inte på grund av de två dussin nya rader du just skrev, utan på grund av det du skev i förrgår och trodde var rätt.

Felsökning

Att felsöka är utmanande och krävande. Ibland är det ens väldigt svårt att acceptera att det är fel. Ibland verkar det som om det som händer omöjligt KAN hända. Trots att det händer.

När du felsöker måste du vara “open-minded” och struktureread. Att det fungerade i förrgår betyder inte att det var rätt då. Försök ta tillvara på det som faktiskt händer och spåra felet bakåt. Har en variable fel värde? Ta reda på alla ställen där den variabeln tilldelas. Kontrollera att den alltid initieras. Ta reda på vilken tilldelning som ger variabeln fel värde. Den fick fel värde från någon av de variabler som förekommer i tilldelningsuttrycket. Vilken? ***Skriv ut tydlig spårinformation.*** Fortsätt på samma sätt med nästa variabel som är fel, och nästa, tills du förstår hur felet uppstod. Ibland kör du fast ändå. Fråga då en assistent. Tydlig information är felaktigheters största fiende. Att bara lägga in en spårutskrift “Nu är jag här” och sedan en “Nu är jag här 2“ räcker inte på långa vägar. Ibland måste du även lägga in kod som bara skriver ut spårinformation “vid rätt tillfälle” för att du inte skall drunkna i den.

Se till att testa din kod ofta och noga. Försök skapa testfall för varje möjlig exekveringssekvens. Ibland är det lättare att testa ett delsystem utanför PintOS med ett eget testprogram.

Felsök systematiskt. Skriv ut värdet på alla variabler som kan påverka, även de som du “vet är rätt” (går något fel så är ju något av det du “vet” fel, eller hur?). Skriv noggranna felmeddelanden med mycket information, “kalle1”, “kalle2” osv är inte bra felutskrifter. Det bör framgå av varje utskrift vilken tråd som exekverar, vilken källkodfil, funktion och rad det är, samt namn och värde på de variabler som funktionen använder. Det gäller att hitta den information som inte är som förväntat. Var förutsättningslös. Blir det fel, oavsett hur omöjligt felet är, så är det fel, och det är i din kod felet finns även om symptomen uppstår djupt inne i någon befintlig funktion. Använd backtrace för att spåra sådant tillbaka till din kod.

Om denna instruktion

Denna instruktion började utvecklas parallellt med kursen som gavs våren 2010. Studenterna gavs då möjlighet att direkt följa den nya instruktionen, vilket de flesta valde att göra. Fortfarande finns mycket att önska om instruktionen, barnsjukdomar m.m., och den kommer fortsätta utvecklas under åren som kommer. Det är önskvärt att du kommer med synpunkter, och formulär för detta finns nedan. Till dags dato (Januari 2013) har ännu ingen använt formuläret. Du har chans att bli först.

PintOS används i flera kurser. Den främsta av dessa är TDDB68. Det är den kurs på IDA som först började använda PintOS hösten 2007. Jag har själv undervisat PintOS höst och vår sedan dess. Ursprungliga instruktionen och erfarenheterna sedan dess (i skrivande stund 9 kursomgångar) ligger till grund för denna instruktion. Det är OK att läsa och använda information som t.ex. TDDB68 tilhandahåller, men ytterst är det denna instruktion som gäller. I slutänden är det egentligen inte så viktigt vilka delsteg som gjorts eller i vilken ordning, det som examineras är till slut att din implementation av alla systemanrop som beskrivs fungerar korrekt.

Instruktionen har två delar. De första uppgifterna, fram till deluppgift 9, har syftet att lära känna programspråket C, och att implementera eller använda saker som är bra att ha senare. Från deluppgift 10 lägger du till funktionalitet i PintOS steg för steg.