Labb 2: Binär räknare i x86

VT2020

I den här laborationen kommer ni lära er mer om hur man arbetar med assembly på ett sätt som påminner om hur man arbetar mot hårdvara som t.ex. LED lampor och knappar. Uppgiften är att skriva ett program som räknar upp en räknare då en tangent trycks på tangentbordet, och skriva ut värdet av denna räknare binärt i terminalen på emulerade LEDar. Vi har siktat på att emulera hur detta hade gått till om ni arbetat mot riktiga LEDar och knappar kopplade till en Raspberry Pi. Hur den emulerade hårdvaran fungerar beskrivs på sida 5.

För att skapa den binära räknaren kommer ni behöva skriva specifika bitmönster till specifika bitar i en array som representerar GPIO-minnet[1] hos en Raspberry Pi. Detta innebär att ni under laborationens gång kommer bli väl bekanta med bitwise operationer. Formlerna ni behöver finns beskrivna på sida 4.

1 Uppgift

Uppgiften är att initiera och skriva en emulerad interrupt-rutin. Interrupt-rutinen körs vid knapptryck, och ska räkna upp en räknare som skrivs ut binärt på fyra emulerade LEDar. Räknaren ska räkna från 0 till 15 modulo 16, och alltså börja om på 0 efter 15.

Innan ni kan skriva ut värdet på räknaren på LEDarna, eller ens köra interrupt rutinen, behöver ni initiera GPIO minnet. Ni behöver ange att en GPIO pin (index 0) används som input, och att fyra pins (index $1-4$) används som output.

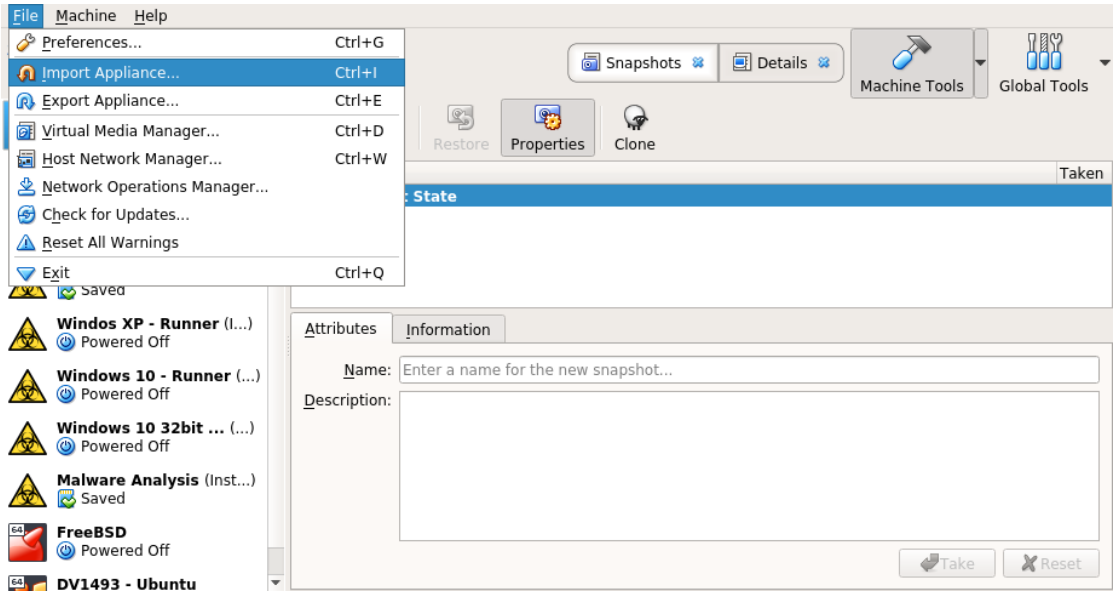
All er kod ska skrivas i x86_64 assembly i `counter_S.S` , och ni får under inga omständigheter direkt referera till funktioner eller variabler som definieras i C-koden.

Inlämningen består av *all* din kod, inklusive `counter_c.c` och `Makefile` som ni inte får röra.

2 Plattformen

Laborationen ska genomföras på en virtuell maskin som kör Ubuntu 18.04 LTS. En `.ova` fil med detta färdiginstallerat finns tillgängligt på Canvas. Den virtuella maskinen

[1] General-Purpose Input/Output, tillåter datorn att prata med hårdvara: `https://en.wikipedia.org/wiki/General-purpose_input/output`



innehåller också labb-filerna. Användandet av en virtuell maskin säkerställer att alla får samma miljö att arbeta i, och det innebär också att ni slipper konfigurera maskinen innan ni börjar labba. För att få igång VMen är allt ni behöver göra att installera VirtualBox[2], ladda ner `.ova` filen, och importera denna. Detta kan ta ett par minuter.

En av de saker som har ändrats, jämfört med Ubuntu out-of-the-box, är att *Address Space Layout Randomization* (ASLR) är avstängt. Detta är en funktion som i normala fall är påslaget, och ser till att den virtuella adressrymden som en process arbetar med är slumpad. Detta görs för att minska risken för att säkerhetshål som *buffer overflows* kan utnyttjas. Dock är en stor del i denna labb att beräkna minnesadresser, och av denna anledning har vi valt att slå av funktionaliteten. Annars hade ni fått gissa er fram till rätt adresser.

2.1 Virtuell Maskin

Den virtuella maskinen innehåller alla labbfiler ni behöver. Dessa finns i mappen `lab02` i er hem-mapp. Inloggning sker automatiskt när ni startar maskinen, men skulle ni låsa er ute är användarnamnet `student` och lösenordet `letmein`.

2.2 Projektfiler

I mappen `lab02` på den virtuella maskinen finns tre filer; `counter_c.c` , `counter_S.S` och `Makefile` Den fil ni ska arbeta i är `counter_S.S` de andra filerna ska förbli orörda (men titta gärna i dem så att ni vet vad de gör).

`counter_c.c` innehåller koden som emulerar knappar, LEDar samt GPIO minnet. Den ser till att knappar och LEDar enbart fungerar om de är påslagna och i rätt läge,

[2]Vi har testat med v5.2.26 till Ubuntu samt 6.0.20 till Windows

och den ser till att knappar och LEDar tänds och släcks som de ska.

`Makefile` är bygginstruktioner för projektet. För att kompilera m.h.a. filen öppnar ni en terminal, navigerar till mappen där `Makefile` ligger och skriver `make`. För att ta bort alla gamla objekt och exekverbara filer skriver ni `make clean`.

I `counter_S.S` finns ett par funktions-skelett som ni måste använda (dessa anropas ifrån C-koden), vad de ska göra beskrivs nedan. Förutom dessa funktioner är ni mer än välkomna att definiera egna funktioner om ni behöver det. Undvik dock att definiera nya variabler, då detta kan flytta GPIO arrayen, och då blir resten av labben väldigt komplicerad. Om ni måste lägga till variabler, t.ex. för att kunna använda `printf` under tiden ni debuggar, kan ni ta en titt i avsnitt 5 på sida 5 för att se hur ni kan åstadkomma det. Men lättast är om ni kan undvika det.

2.2.1 `counter_S.S`

Filen `counter_S.S` är den fil ni ska arbeta i. Den innehåller ett par funktionsskal för att hjälpa er på vägen. Här följer en kort genomgång om vad som behöver göras i varje funktion. Ni får mer än gärna definiera fler funktioner än dessa, men dessa måste finnas i programmet och fungera enligt nedan.

2.2.1.1 `interrupt`

Detta är interrupt-rutinen. Den kallas när avbrottet sker, d.v.s. när ni trycker en tangent på tangentbordet. Funktionen ska räkna upp en räknare från 0 till 15 modulo 16, och uppdatera de GPIO-pins som är kopplade till LEDarna (index $1-4$), så att de visar räknarens värde binärt.

2.2.1.2 `setup`

Denna rutin kallas när programmet initieras. Det som behöver göras är att beräkna och spara undan minnesadressen till GPIO och initiera alla GPIO-pinnar som ska användas till rätt värden. Fyra GPIO-pins behöver alltså sättas till output (index $1-4$), och en behöver sättas till input (index 0).

3 Manipulera GPIO

För att manipulera GPIO pinnarna behöver man skriva en etta eller nolla till specifika bitar i minnet. I kodstycket nedan finns ett antal formler som används till detta.

```
/*
 * Formel för minnesaddressen till de emulerade GPIO pinnarna
 */
int gpio_addr = ((0x60 << 16) | (0xFFFF & 0x1000) | (~(0x9E) & 0xFF))


/*
 * Sätter GPIO pin med nummer GPIO_NR till att vara output.
 * gpio_addr är basaddressen. Notera att gpio_addr används
 * som en pekare.
 */
*(short*)gpio_addr |= (0x2 << 6) << ((4-GPIO_NR)*2);

/*
** För att sätta en GPIO pin till input används följande formel
*/
*(short*)gpio_addr + (GPIO_NR/8) |= 0x3 << ((((3-GPIO_NR)*2) % 8)+8);

/*
 * För att sätta en GPIO pin till högt läge (alltså LED på)
 * används följande formel
 */
*(int*)gpio_addr + 3 |= ((0x3 << 4)+1) << (8*(GPIO_NR-1));

/*
 * För att sätta en GPIO pin till lågt läge (alltså LED av)
 * används följande formel
 */
*(int*)gpio_addr + 3 &= ~(0xCF << (8*(GPIO_NR-1)));
```

4 Inlämning

Inlämningen är all er kod, inklusive `counter_c.c` och `Makefile` . Detta ska packas till en `.tgz` fil och laddas upp på Canvas. För att göra detta så smidigt som möjligt kan ni köra `make submission` för att packa ihop filerna korrekt.

5 Debugging

Som tidigare nämnt kan ni stöta på problem om ni vill lägga till nya variabler i programmet, t.ex. för att använda `printf` , eftersom detta riskerar att flytta GPIO arrayen i minnet. Detta innebär att beräkningen ni fått angiven inte längre stämmer. Det lättaste är att undvika fler variabler, och debugga med `gdb` (jättebra tillfälle att lära sig använda verktyget!), men det går fortfarande att använda `printf` . För att göra det måste ni dock lägga till ett offset på beräkningen av basadressen. För att hitta detta offset kan ni köra följande, vilket kommer visa minnesadressen som GPIO arrayen läggs på.

```
nm counter | grep 'B gpio'
```

När ni har detta kan ni beräkna hur långt den flyttats ifrån där den ska vara (d.v.s. adressen som beräkningen ni fått resulterar i), och helt enkelt lägga till en `ADD` instruktion i slutet av beräkningen med motsvarande värde. Med andra ord, om originaladressen är `0x401000` och ni med ovanstående får `0x401008` lägger ni till `addq $8, %register` till slutet av beräkningen av adressen.

6 Emulerad Hårdvara

Att förstå exakt hur hårdvaran är emulerad är inte nödvändigt för att klara labben; informationen nedan är inbyggd i formlerna ni ska implementera. Detta är dock en simplifierad variant av hur riktig hårdvara fungerar, så det kan därför vara nyttigt och intressant att sätta sig in lite i det.

Hårdvaran är emulerad i filen `counter_c.c` . Den huvudsakliga komponenten är en array som består av 7 bytes. De två första används som kontroll för GPIO pinnarna, och avgör om de ska vara input eller output, på eller av. Detta görs av ett bit-par, där den första biten avgör om GPIO är på (1) eller av (0), och den andra biten avgör om den är input (1) eller output (0). Bit-paret 10 innebär därför att en pin är påslagen, och används som output. 11 innebär att en pin är på, och satt till input.

Vilken GPIO som kontrolleras bestäms av vilket bit-par i de två kontroll-byten som används; de två först (alltså most significant bits (MSB)) kontrollerar GPIO 0, nästa två kontrollerar GPIO 1. GPIO 0 simulerar att en knapp är kopplad till den, och GPIO 1-4 simulerar LEDar. På ett riktigt system hade det gått att välja helt fritt.

Ctl	Ctl	GPIO 0	GPIO 1	GPIO 2	GPIO 3	GPIO 4

Kod är redan skriven för att lyssna på tangentbordet och ”tända”/”släcka” LEDerna, men detta görs enbart om dessa är påslagna, och LEDarna ”tänds” och ”släcks” enbart om rätt värde är skriva till de bytes som representerar dem i GPIO minnet.