DV1464/DV1493 Datorteknik – Lab3

Bibliotek för in- och utmatning i Intel assembler

2015-03-11

Blekinge Tekniska Högskola
Carina Nilsson

Inledning

Syftet med det här lilla projektet är att bli bekant med en annan processortyp, Intel (eller AMD som är kompatibel) . Assemblerspråk för denna processortyp finns i olika syntaxvarianter. I projektuppgiften ska AT&T-syntax som dominerar i *Unix*-världen användas. Det är den som `gcc` (`gas`) använder. Det finns en annan varuant som kallas *Intel*-syntax, som är omvänd, och som används i Windows-världen.

Uppgiften består i att skriva ett litet bibliotek med rutiner för in- och utmatning av text som skulle kunna anropas från ett högnivåprogram skrivet i t ex programspråket C. Det är då viktigt att hålla sig till samma konventioner som kompilatorn gör för hur register och stack används. Annars kommer rutinerna inte att fungera tillsammans med högnivåkoden. In- och utmatning på lägsta nivå är ganska besvärligt och innefattar hantering av avbrott, men själva avbrottshanteringen ingår inte i den här uppgiften.

I Unix/Linux betraktas alla enheter som filer, och att läsa från tangentbordet är att läsa från ”filen” `stdin`, och att skriva till skärmen är att skriva till ”filen” `stdout`. För den råa överföringen kan färdiga C-rutiner användas, `fgets` och `puts`. De rutinerna blir gränssnittet mot systemet och avbrottshanteringen, och gör att avbrott inte behöver hanteras av de biblioteksrutiner som ska skrivas inom ramen för uppgiften.

Rutinen `fgets` används på följande sätt: `fgets(buffert, antal, fil)` i C, där `buffert` är adressen till ett minnesutrymme till vilket den lästa raden ska sparas, `antal` är det maximala antalet tecken att läsa och `fil` anger varifrån läsningen ska ske. Rutinen läser normalt till dess att ny rad (Newline = EOL) eller filslut (End of File = EOF) påträffas. Strängen läggs i bufferten och avslutas med NULL-tecknet (ASCII-kod 0). I assemblerspråk för Intel x64 kan ett anrop av `fgets` se ut så här:

```
:
        movq    $buf, %rdi  # lägg i buf, där buf är en bit reserverat
        minne
        movq    $5,%rsi         # högst 5-1=4 tecken (NULL räknas ju också)
        movq    stdin, %rdx     # från standard input stdin=$0 om ej def.
        call    fgets
:
```

Det finns en motsvarande rutin `puts` för utmatningen, som i C anropas enligt: `puts(buffert)`.

Uppgiften går ut på att implementera rutiner för att hantera inmatning och utmatning av data. För att klara av detta behöver man reservera plats för två olika systembuffertar, en för inmatning och en för utmatning. Varje buffert behöver också en variabel som håller reda på aktuell position i respektive buffert.

Uppgiften görs lämpligen i grupper om två. Annan gruppstorlek ska godkännas av handledaren. Grupper med tre eller fler deltagare accepteras inte! Observera att uppgiften tar väsentligt mycket mer tid i anspråk än de schemalagda handledningstillfällena. När hela uppgiften är implementerad laddas koden upp i inlämningsmappen på It’s Learning.

Uppgift

Att skriva ett bibliotek med rutiner för in- och utmatning av olika typer av data. Eftersom er kod ska användas som ett bibliotek ska de ligga i en separat fil som kan länkas med andra program. Det är också viktigt att specifikationen följs så att det fungerar tillsammans med testprogrammet Mprov64 när sluttestet sker. De rutiner som ska implementeras för inmatning är följande:

inImage – läser in en ny textrad från tangentbordet till er inläsningsbuffert för indata och nollställer positionen till den aktuella bufferten. Andra inläsningsrutiner kommer sedan att jobba mot den här bufferten. Om positionen står vid buffertens slut när någon av de andra inläsningsrutinerna nedan anropas ska inimage anropas automatiskt (i den aktuella rutinen), så att det alltid finns data att arbeta med.

getInt – returnerar ett heltal. Rutinen tolkar en sträng som börjar på aktuell buffertposition och fortsätter tills ett tecken som inte kan ingå i ett heltal påträffas. Den lästa substrängen översätts till heltalsformat och returneras. Positionen i bufferten ska vara det första tecken som inte ingick i talet när rutinen lämnas. Inledande blanktecken i talet ska vara tillåtna. Plustecken och minustecken ska kunna inleda talet (men inte tvunget) och vara direkt följd av en eller flera heltalssiffror.

getText(buf, n) – överför maximalt *n* tecken från aktuell position i inbufferten och framåt till *buf*. När rutinen lämnas ska aktuell position i inbufferten vara första tecknet efter den överförda strängen. Om det inte finns *n* st. tecken kvar i inbufferten avbryts överföringen vid slutet av den. Returnera antalet verkligt överförda tecken.

getChar – returnerar ett tecken och flyttar fram aktuell position ett steg

getInPos – returnera aktuell buffertposition för inbufferten

setInPos(n) – sätt aktuell buffertposition för inbufferten till n. n måste dock ligga i intervallet [0,*MAXPOS*]. Om *n*<0sätt den till 0, om *n*>*MAXPOS* sätts den till *MAXPOS*.

De rutiner som ska implementeras för utmatning är följande:

outImage – skriv ut strängen i utbufferten i terminalen. Om någon av nedanstående utdatarutiner når buffertens slut så ska ett anrop till *outimage* göras automatiskt så att man får en tömd utbuffert att jobba mot.

putInt(n) – lägg ut talet *n* som sträng i utbufferten från och med den aktuella positionen. Glöm inte att uppdatera aktuell position.

putText(buf) – lägg texten som finns i *buf* från och med den aktuella positionen i utbufferten. Glöm inte att uppdatera aktuell position.

putChar(c) – lägg tecknet *c* i utbufferten och flytta fram aktuell position ett steg.

getOutPos - returnera aktuell buffertposition för utbufferten

setOutPos(n) - sätt aktuell buffertposition för utbufferten till *n*. *n* måste dock ligga i intervallet [0,*MAXPOS*]. Om *n*<0 sätt den till 0, om *n*>*MAXPOS* sätts den till *MAXPOS*.

Editering, kompilering och länkning

Skriver program gör vi i en texteditor, t.ex. gedit.

Att översätta ett program till ett körbart kan göras med hjälp av verktyget `gcc` på följande sätt. Anta att vi har ett assemblerprogram i filen `filnamn.s`:

```
gcc -o fil filnamn.s
```

Eftersom namnet slutar på `.s` förutsätter kompilatorverktyget `gcc` att det är en assemblerfil och anropar översättaren `as`. Därefter länkas programmet med lämpliga bibliotek och läggs i en fil med namnet `fil`. Programmet kan sedan köras med:

```
./fil
```

I exemplet ovan hade finns bara en källkodsfil, `filnamn.s`, men `gcc` kan ta godtyckligt många parametrar (filnamn) som slutar på `.s`, `.c` eller `.o`.

I följande exempel kommer `filnamn.s` kompileras och länkas tillsammans med `Mprov.s` till en exekverbar fil som heter `fil2`:

```
gcc -o fil2 filnamn.s Mprov.s
```

Exempel på ett program att köra skrivet i Intel-x64 assembler

Här följer ett litet kodexempel som kan testas som uppvärmning för att komma igång. Programmet kallar på en rekursivt implementerad subrutin `fak` som beräknar fakultet på det tal som skickas med som parameter via register `%rdi`. Resultatet returneras i register `%rax`. Resultatet skrivs sedan ut. Därefter görs en inläsning från terminalen av några tecken (max 4 st), som sedan skrivs ut igen.

```
        .text
        .global         main
main:   sub     $8, %rsp                #fixa stack alignment
        movq    $5, %rdi                # Beräkna 5!
        call    fak
        movq    %rax, %rsi      #Flytta returvärdet till argumentregistret
        movq    $resMsg, %rdi           # skriv ut fak= ”resultat”
        call    printf
                        # läs med fgets(BUF,5,stdin)
        movq    $buf, %rdi              # lägg i BUF
        movq    $5,%rsi         # högst 5-1=4 tecken
        movq    stdin, %rdx             # från standard input
        call    fgets
        movq    $buf, %rdi
        call    printf          # skriv ut buffert

        movq    $endMsg, %rdi           # följd av slut
        call    printf
        add     $8, %rsp        #återställ stack
        call    exit            # avsluta programmet
```

```
                                # Här finns funktionen f = n! (rekursiv)
fak:    pushq   %rbp            # spara frame-pekare på stack, 8 byte
        movq    %rsp,%rbp       # ny frame-pekare = nuvarande stackpekare
        cmpq    $1,%rdi         # if n>1
        jle     L1
        pushq   %rdi            #lägg anropsvärde på stacken, 8 byte
        decq    %rdi            #räkna ned värdet med 1
        call    fak             # temp=f(n-1)
        popq    %rdi            #hämta från stack
        imul    %rdi,%rax               # return n*temp
        movq    %rbp,%rsp               # Återställ stackpekare (=frame)
        popq    %rbp            # Återställ frame-pekare från stacken
        ret                     # Återvänd
L1:     movq    $1,%rax         # else return 1
        movq    %rbp,%rsp       # Återställ stackpekare (=frame)
        popq    %rbp            # Återställ frame-pekare från stacken
        ret                     # Återvänd

        .data
resMsg: .asciz  "fak=%d\n"
buf:    .asciz  "xxxxxxxxx"
endMsg: .asciz  "slut\n"
```