TSEA28 Datorteknik Y (och U)

Föreläsning 7

Kent Palmkvist, ISY

LINKÖPINGS UNIVERSITET

Dagens föreläsning

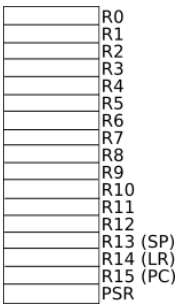
- Avbrott på ARM
  - Stänga av/slå på avbrott
- Kort info lab2 och lab3
- Funktioner för stöd i operativsystem
  - Samtidiga processer

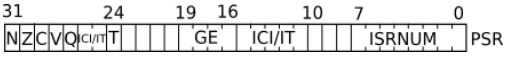
Praktiska kommentarer

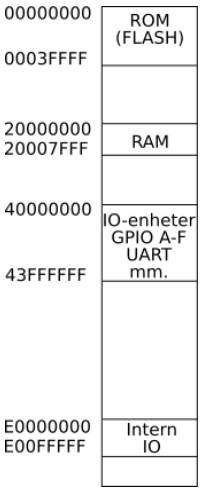
- Ordning i labbet?
  - Koppla ned anslutningar gjorda på DARMA (mot tangentbord etc.)
    - Ta INTE bort sladd mellan DARMA och strömförsörjning (lådan)
  - Vrid INTE skärmen fysiskt
    - Se noteringar på whiteboard vid dörren
- Redovisning kvarvarande lab1
  - Ta det i början av lab 2
- Lab2a anmälan deadline
  - 16 Februari 23.30

Programmerarens vy av DARMA

- CPU med 13 generella register, PC, LR, SP samt statusregister PSR
  - Varje register består av 32 bitar (4 byte)
- 256 KB Flash programminne (ROM)
  - Programkod (4 byte per instruktion)
  - Konstanter
- 32KB RAM för data
  - Variabler
  - Stack etc.
- Minnesmappad I/O





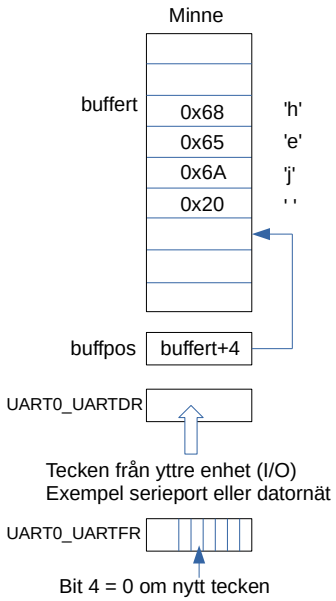


Avbrott i en dator

- Ingång till processorn som indikerar behov av avbrott
- Stoppar pågående program
  - Programmet som stoppas märker inget
- En avbrottsrutin startas
  - Kan finnnas många olika beroende på orsak till avbrott
  - Ser ut som en subrutin

Exempel motivation till avbrott

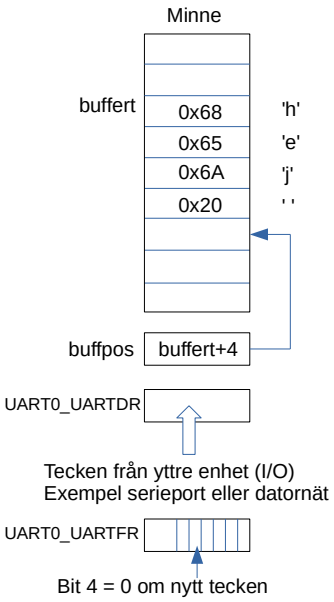
- Antag vi har en I/O enhet som tar emot tecken i hög hastighet.
  - Om UART0_UARTDR inte töms innan nästa tecken från I/O så tappas tecken bort
  - Spara inkommande tecken (finns i UART0_UARTDR) i en buffert i minnet
- Subrutin för att spara värde i buffert
  - Variabel buffpos pekar på nästa lediga plats i buffert
  - Subrutinen kontrollerar om nytt tecken finns, och i så fall placera det på lediga platsen i bufferten och uppdatera buffpos till nästa lediga plats.



Exempel motivation till avbrott

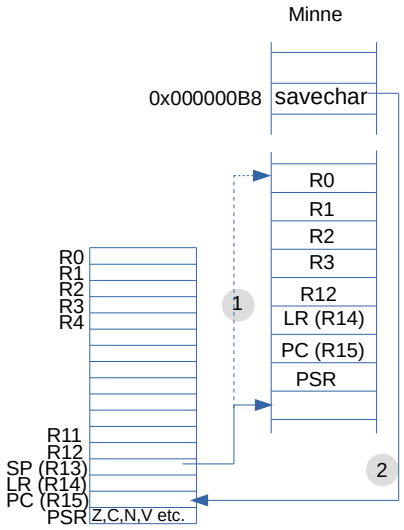
- Subrutin för att spara värde i buffert

```
savechar push {r0,r1,r2}                ; Kommer använda registren
         mov r1,#(UART0_UARTFR & 0xffff) ; Peka på
         movt r1,#(UART0_UARTFR >> 16)   ; statusregister
         ldr  r2,[r1]                    ; Hämta status
         ands r2,r2,#0x10                ; Finns tecken?
         bne skipcopy                    ; nej (bit 4= 1), hoppa ur
         mov r1,#(UART0_UARTDR & 0xffff) ; peka på
         movt r1,#(UART0_UARTDR >> 16)   ; dataregistret
         ldr  r2,[r1]                    ; läs tecken
         mov r1,#(buffpos & 0xffff)      ; peka på variabeln
         movt r1,#(buffpos >> 16)        ; buffpos
         ldr  r0,[r1]                    ; hämta värdet i buffpos
         strb r2,[r0],#1                 ; spara tecken i buffert, öka r0
         str  r0,[r1]                    ; öka adress i buffpos
skipcopy pop {r0, r1, r2}                ; återställ register
         bx lr
```



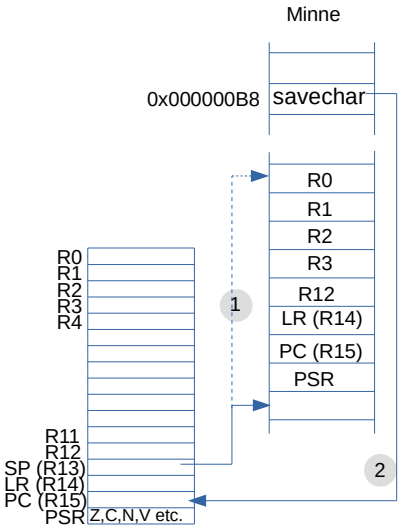
Avbrott på ARM Cortex-M4 (Darma)

- När avbrottssignal aktiveras
  - Automatisk push av PSR, PC, LR, R12, R3, R2, R1, R0 på stack (SP (R13) minskas) (steg 1)
  - Läs fix minnexadress (t ex 0x000000B8-0x000000BB) som innehåller adress till avbrottrutin (steg 2)
    - Avbrottets orsak väljer exakt vilken rad i en tabell
  - Hantera avbrottet (t ex läs tangentbord)
  - Stäng av källan till avbrott
  - Återställ automatiskt register med pop av PSR, PC, LR, R12, R3, R2, R1 och R0 när bx lr körs. (steg 1 åt andra hållet)



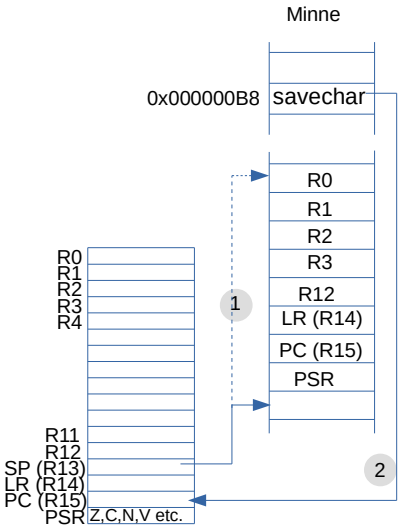
Avbrott på ARM Cortex-M4 (Darma)

- Notera att PC också sparas!
  - Vanlig subrutinanrop placerar PC i LR. Men inte i vid avbrott!
- Hur vet processorn att BX LR ska hämta många värden på stacken?
  - Kan använda subrutiner i avbrottet
- Ersätter LR-värdet med speciella adresser
  - 0xFFFFFFxx
  - BX LR med t ex LR = 0xFFFFFFF1 => hämtar R0-3,R12,LR,PC,PSR



Oväntad detalj: 8-byte alignment av stack

- Vid avbrott läggs alltid första registret på en adress jämnt delbar med 8
  - Kan se ut som extra värde lagts på stacken
- Bit 9 i PSR (statusregistret) sätts om extra värde lagts på stacken
  - PSR placeras också på stacken
  - Måste kontrolleras för att veta hur stacken ser ut
    - Måste läsa stack uppifrån



I/O-enheternas anslutning till avbrott

- De flesta I/O-enheter kan ge avbrott
  - Kan konfigureras på många sätt
    - Vilka pinnar
    - nivå/flank
- Sortera och prioritera avbrott med NVIC (Nested Vector Interrupt Controller)
  - Samlar in avbrottsbegäran (signaler) från alla I/O-enheter
  - Avgör vilket avbrott är viktigast
    - Kan konfigureras av programmet
  - Skickar vidare information till processor om begäran + vilken enhet
    - Rad i tabell med avbrottsrutin väljs beroende på enhet

Hur fungerar avbrott från GPIO?

- Varje pinne kan trigga avbrott
  - Låg nivå, hög nivå, positiv flank, negativ flank eller båda flanker
  - Styrs med kontrollregister i GPIO
    - GPIOIS, GPIOIBE, GPIOIEV
    - 1 bit per pinne i kontrollregistren
- Om flank används ligger begäran kvar tills den nollställs
  - Skriv 1:or i GPIOICR (GPIO Interrupt Clear Register) för att nollställa
- Om orsak inte åtgärdad (nivå eller nollställt flank) fås nytt avbrott direkt
  - Måste nollställa orsak för att undvika nytt avbrott

Flank respektive nivåavbrott från GPIO

- Jämför metoder att få hjälp i lab (1 = hand upp, 0 = hand nere)
- Nivå: Sträcka upp handen. Så länge hand uppe vill du ha hjälp
  - Motsvarar avbrott baserat på nivå.
  - I/O-enheten tar bort nivån själv.
- Flank: Skriva upp sig på whiteboard.
  - Motsvarar avbrott på flank (att gå fram och skriva upp sig motsvarar att ändra till uppräckt hand)
  - Önskan om hjälp finns kvar till den suddas bort (när ni fått hjälp)
  - I/O-enheten kan ta bort nivån, men begäran finns kvar
  - Kräver explicit nollställning (suddas) när hjälp fåtts (skriva ICR-registret)

Avbrottsrutinens struktur i Darma

- Avbrottsrutinen startas med R0-R3, R12 redan sparade
  - LR innehåller viktigt värde! Kan behöva sparas om subrutiner ska användas
  - Spara ytterligare register om dom ska användas (push)
- Stäng av anledning till avbrott, beror på vilken enhet
  - Lämpligen i starten av avbrottet
- Utför vad som ska göras vid avbrottet
  - Subrutiner får användas om LR sparas först
- Arottsrutinen avslutas med bx lr
  - Återställ register först om R4-R11 och/eller LR använts (pop)

Att ignorera avbrott

- För att förhindra avbrott (maska bort avbrott)
  - Behövs t ex när hårdvarukretsar konfigureras
  - Avbrott automatiskt avstängt vid start av program
- Låt processorn ignorera avbrott
  - När programmet inte får avbrytas

    CPSID I
- Tillåt avbrott
  - När all konfigurering är klar

    CPSIE I

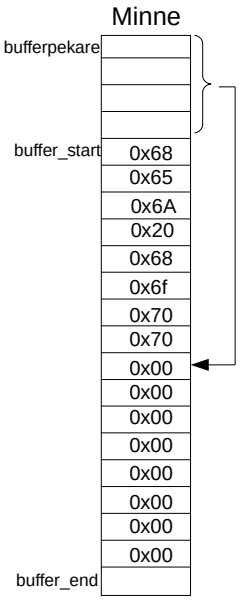
Applikationsexempel, varför hindra avbrott

- Två aktiviteter
  - Huvudprogram som utför långa beräkningar, där inmatning sker via tangentbord
  - Tangentbordet liknar labbens tangentbord, det håller bara reda på aktuell nedtryckt tangent
- För att slippa tappa bort tangenttryck medan huvudprogrammet räknar används en buffert
  - Bufferten är några minnesplatser plus en pekare till nästa lediga plats
  - Huvudprogrammet hämtar tecken i bufferten
  - Avbrottsprogrammet sparar undan aktuellt tecken från tangentbordet i bufferten (liknar förra exemplet)

Exempel, behov maska bort avbrott

- Rutin som sparar tangenttryck i buffert
  - Tangent lagras i buffert (skiftas när ny tangent tryckts).
  - Avbrottsrutinen sparar senaste tecknet i en kö i buffert
    - Exempel visar minne efter ”hej hopp” matats in

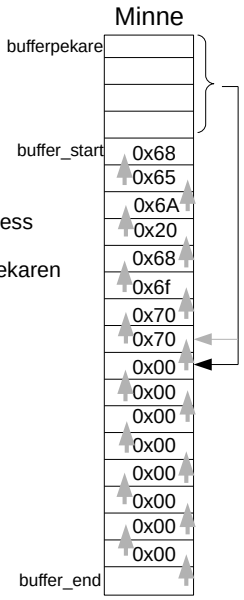
```
buffered_addkey: mov r0,#(bufferpekare & 0xffff)
                 movt r0,#(bufferpekare >> 16)
                 ldr  r1,[r0]              ; hämta aktuell ledig plats
                 mov  r2,#(buffer_end_adr & 0xffff)
                 movt r2,#(buffer_end_adr >> 16)
                 cmp  r2,r1                ; fyllt buffert?
                 beq  buffer_full          ; Ja, skippa
                 mov  r2,#(UART0_DR & 0xffff)   ; adress till I/O
                 movt r2,#(UART0_DR >> 16)      ; för tangentvärde
                 ldr  r3,[r2]              ; hämta tangentvärde
                 strb r3,[r1],#1           ; spara I buffert, öka r1 med 1
                 str  r1,[r0]              ; spara nya buffertpekaren
buffer_full:bx   lr                        ; återhopp från avbrott
```



Exempel, behov maska bort avbrott, forts.

- Huvud-programmet tar hand om tangenttryck från bufferten
  - Returnera äldsta tecknet (0x68) och ta bort det i buffert (skiftas bort)

```
buffered_getkey:
      mov  r1,#(bufferpekare & 0xffff)  ; konstant bufferpekare adress
      movt r1,#(bufferpekare >> 16)
      add  r2,r1,#4     ; konstant buffer_start adress direkt efter pekaren
      ldr  r3,[r1]      ; nuvarande värde på bufferpekare
      cmp  r3,r2        ; pekar bufferpekare på buffer_start?
      beq  tom          ; Tom buffer!
      mov  r4,#15             ; 15 tecken att flytta
      ldrb r0,[r3],#1   ; Äldsta tangent i r0, minska r3 med 1
      str  r3,[r1]            ; spara nya buffertpekare
loop: ldrb r5,[r2,#1]!  ; Öka r2, Hämta föregående tecken
      strb r5,[r2,#-1]  ; placera på adressen innan
      subs r4,r4,#1     ; flyttat alla?
      bne  loop         ; nej, ta nästa
      b    slut         ; Returnerar tryckt knapp i d0.b
tom:  mov  r0,#0        ; Om tom buffer. Returnera 0
slut: bx   lr
```

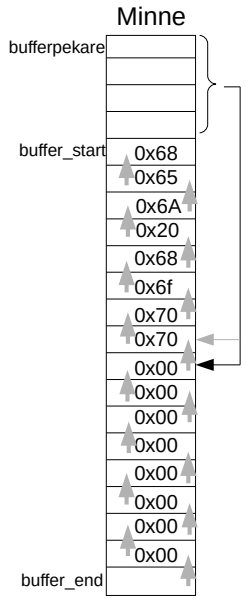


Exempel, behov maska bort avbrott, forts.

- Om avbrott sker när buffered_getkey kör ändras bufferpekare, men inte kopian i r3. När buffered_getkey avslutas tappas ett tecken bort eller förstör näst senaste tecknet

Kopia i register

```
buffered_getkey:
        mov  r1,#(bufferpekare & 0xffff)    ; konstant bufferpekare adress
        movt r1,#(bufferpekare >> 16)
        add  r2,r1,#4      ; konstant buffer_start adress direkt efter pekaren
        ldr  r3,[r1]       ; nuvarande värde på bufferpekare
        cmp  r3,r2         ; pekar bufferpekare på buffer_start?
        beq  tom           ; Tom buffer!
        mov  r4,#15              ; 15 tecken att flytta
        ldrb r0,[r3],#1    ; Äldsta tangent i r0, minska r3 med 1
        str  r3,[r1]             ; spara nya buffertpekare
loop:   ldrb r5,[r2,#1]!   ; Öka r2, Hämta föregående tecken
        strb r5,[r2,#-1]   ; placera på adressen innan
        subs r4,r4,#1      ; flyttat alla?
        bne  loop          ; nej, ta nästa
        b    slut          ; Returnerar tryckt knapp i d0.b
tom:    mov  r0,#0         ; Om tom buffer. Returnera 0
slut:   bx   lr
```



Lösning på problemet

- buffered_getkey innehåller kritisk sektion som inte får avbrytas
  - Löses genom att maska bort avbrott
  - Glöm inte tillåta avbrott i båda fall av avslutning

Tillåt inte avbrott

Tillåt avbrott igen

```
buffered_getkey:
        cpsid i
        mov  r1,#(bufferpekare & 0xffff)    ; konstant bufferpekare adress
        movt r1,#(bufferpekare >> 16)
        add  r2,r1,#4      ; konstant buffer_start adress direkt efter pekaren
        ldr  r3,[r1]       ; nuvarande värde på bufferpekare
        cmp  r3,r2         ; pekar bufferpekare på buffer_start?
        beq  tom           ; Tom buffer!
        mov  r4,#15              ; 15 tecken att flytta
        ldrb r0,[r3],#1    ; Äldsta tangent i r0, minska r3 med 1
        str  r3,[r1]             ; spara nya buffertpekare
loop:   ldrb r5,[r2,#1]!   ; Öka r2, Hämta föregående tecken
        strb r5,[r2,#-1]   ; placera på adressen innan
        subs r4,r4,#1      ; flyttat alla?
        bne  loop          ; nej, ta nästa
        b    slut          ; Returnerar tryckt knapp i d0.b
tom:    mov  r0,#0         ; Om tom buffer. Returnera 0
slut:   cpsie i
        bx   lr
```

Generella riktlinjer för avbrott

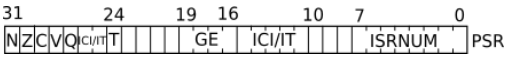
- Maska bort avbrott endast under mycket korta stunder
  - Stor risk att tappa information om avstängt längre tid
- Se till att avbrottsrutiner går snabbt att köra (når bx lr inom få instruktioner)
- Om möjligt skriv program så strömsparfunktioner kan användas
  - Stänga av programmet, vänta på interrupt

Många fler anledningar till avbrott

- Varje orsak har en egen rutin
  - Lägsta adresserna i minnet innehåller tabell med adresser
    - Tabell 2.8 i Tiva TM4C123GH6PM Microcontroller Data Sheet
    - Filen tm4c123gh6pm_startup_ccs.c
  - Exempel
    - Reset hämtar stackvärde på adress 0x00000000 och startadress på adress 0x00000004
    - Externa avbrott (vektorer adress 0x00000040 t om 0x00000268)
    - SVCALL (speciell maskininstruktion SVC som tvingar fram avbrott)
    - Usage faults
      - Division med 0
      - Illegal instruktion: Om instruktionen inte känns igen kan ett litet program kanske utföra den istället
        - Kan implementera nya instruktioner som inte fanns när processorn byggdes (göra gammal utrustning kompatibel med ny programvara).

Cortex-M4 (Darma) statusregister (PSR)

- Flaggor
  - C, V, Z, N enligt tidigare beskrivningar
    - Använd villkorliga hopp för att testa deras värden
- T: Thumb flag, visar om Thumb mode används (alltid = 1)
- ISRNUM: Anger vilken avbrottsrutin som körs
  - 0x02=NMI, 0x03=Hard Fault, 0x05=bus fault, 0x10=avbrottsvektor 0, etc.



Darma, sammanfattning avbrottssystemet

- GPIO-pin kan skapa avbrottsbegäran från GPIO-porten
  - Flanktriggad eller nivå
  - Begäran skickas till NVIC
- NVIC samlar in alla enheters avbrottsbegäran
  - NVIC håller reda på vilken som har högst prioritet
  - Skickar vidare en signal till processorn om begäran av avbrott och vilket avbrottsnummer (vilken rad i tabell med avbrottsrutiner som ska startas)
- Processorn lyssnar på avbrottsbegäran från NVIC
  - Kan sluta lyssna mha CPSID I
  - Om avbrott begärs och tillåts (CPSIE I använt) så sparar processorn register på stacken, hämtar rätt adress i tabellen, och starta avbrottsrutinen.

Intro laboration 2 och 3

Laboration 2: Avbrott

- Skriv avbrottsrutiner och undersök stack
  - Testa prioritet
  - Förstå innehåll på stack
    - Vilket register har kopierats vart
    - Se var programmet avbröts av avbrottsbegäran
- Gametime!
  - Skriv ping-pong spel
  - Avbrottsbaserat
    - Varje spelare har var sin knapp
  - Spelplan: 8-pixel ”skärm”

Laboration 3: digitalur

- Styr en 4-siffrig 7-segment display
  - Antal tillgängliga signaler räcker inte till (7*4=28 olika pinnar)
    - Multiplexing (växla snabbt mellan siffrorna)
  - Avbrott genererade av klocka
- Två delfunktioner
  - Räkna tid (ett avbrott per sekund)
  - Uppdatera utseende på skärm (hundratals avbrott per sekund)
    - Visa bara 1 siffra per gång
- Kräver även sparande av energi
  - Stanna processorn mellan varje avbrott

Funktioner för operativsystem

En vanlig dators funktion

- Olika program kan laddas in
  - Assemblera inte nytt varje gång
- Flera program (verkar) köras samtidigt
  - Programmen kan samsas på samma maskin
- Hantering av I/O kan variera
  - Tangentbord
  - Skärm
  - Lagringsmedia

Extra krav för verklig dator

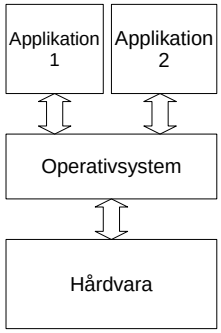
- Behöver gömma detaljfunktion hos I/O
  - Jfr med getkey i lab1, hextangentbord eller datorns tangentbord
  - Subrutiner med väl definierat registeranvändning och funktion
    - T ex printchar, checkcode
- Behöver garantera att ett program inte påverkar ett annat
  - Inte förstöra stacken
  - Inte skriva över andra programmets minne
- Byta snabbt mellan olika program för att ge intryck av samtidigt körande program

Extra krav för vanlig dator

- Behöver gömma detaljfunktion hos I/O
  - Jfr med getkey i lab1, hextangentbord eller datorns tangentbord
  - Subrutiner med väl definierat registeranvändning och funktion
    - T ex printchar, checkcode
- Behöver garantera att ett program inte påverkar ett annat
  - Inte förstöra stacken
  - Inte skriva över andra programmets minne
- Byta snabbt mellan olika program för att ge intryck av samtidigt körande program

Funktion hos operativsystem

- Behöver bygga ut funktion för att bättre stödja ett operativsystem
- Operativsystem hanterar hårdvara och gömmer detaljer från användaren
  - Windows, Linux, respektive MacOS ser likadana ut oberoende av hur datorn är byggd i detalj
  - Ger illusionen av att flera saker sker samtidigt
  - Delar på resurser så att alla användare får tillgång till dom
  - Applikationen ska inte behöva veta alla detaljer om hårdvaran
- Säkerhet
  - Användarprogram får inte krascha datorn (stänga av funtioner, störa andra program)



Hantering av multianvändarsystem

- Separera vanliga program från operativsystem och avbrott
  - Kritisk funktion för att garantera fungerande system
  - Olika lösningar i olika processorer
- Vissa funktioner är kritiska för datorns funktion
  - Avbrottsmask, stackpekare, I/O
- Begränsa programmets möjlighet att påverka dessa funktioner
- Användarprogram ska begära hjälp av operativsystemet för tillgång till hårdvaruspecifika funktioner

Hur hantera slarvig användare

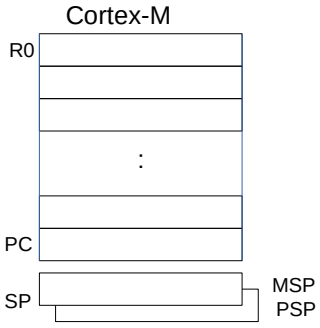
- De flesta funktioner bygger på att stackpekaren är korrekt (pekar på tillgängligt RAM-minne)
- En slarvig användare skulle kunna använda SP till något annat
  - Så länge som inga push/pop görs används inte stacken av användaren
- Operativsystemet behöver fortfarande en fungerande stack, t ex för avbrott
  - Behöver skilja på ”vanlig” programkörning och avbrott
  - Cortex-M: Thread mode (vanligt program) vs Handler mode (avbrottsrutin, operativsystem)

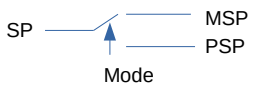
Hantering av multianvändarsystem

- ARM Cortex-A: 7 olika körmoder
  - User, FIQ, IRQ, SVC, Abort, Undef, System.
  - Hopp mellan moder kan växla registerkopior
  - Avbrott byter SP, LR i nästan alla (en uppsättning för varje körmode)
  - FIQ byter även till egna kopior av R8-R12
  - Byter körmode vid avbrott eller återhopp från avbrott
- ARM Cortex-M4: Två olika körmoder, två olika rättighetsmoder
  - Thread (användare) och Handler (avbrott, OS) mode
  - Privileged och unprivileged Thread mode
- x86 (Intel och AMD processorer i Macintosh och IBM PC) har fyra rättighetsnivåer (privilege levels)
  - Vanliga operativsystem använder bara två nivåer

Handler / Thread mode i Cortex-M

- Vanlig användare kan förstöra stackpekarens värde
  - Lösning i Cortex-M4: En extra stackpekare för Handler mode
  - Byter register beroende på mode
  - Använd MSP i handler mode (och vid avbrott)
  - Använd PSP i thread mode (vanlig stack)
- Två versioner av thread mode
  - Privileged och unprivileged
    - Begränsar access till systemresurser
    - Begränsar access till vissa minnesområden



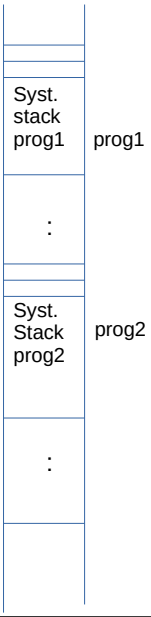


För den nyfikne: Kontextbyte

- Mål: Få det att verka som flera program körs samtidigt på datorn
  - Utan att programmen behöver veta om varandra
  - Tidiga system (t ex Mac) använde kolloborativ multitasking, där varje program aktivt lämnar över till nästa program (tråkigt om ett program inte vill lämna till nästa)
- Låt varje enskilt program köra på processorn
- Efter en viss tid avbryts nuvarande program och nästa startas
- Jämför med en restaurang med 1 anställd
  - Måste vara kock ibland, servitör ibland, städare ibland
  - Fungerar så länge som byten sker tillräckligt ofta

För den nyfikne: Kontextbyte, forts.

- Konfigurera en timer som ger avbrott var 10:e ms (100 Hz)
  - 10 ms per program innan nästa program får tid att köra
- Vid avbrott
  - Spara r0-r15 (inkl PC, SP och LR) samt PSR till kärnans minnesområde (1 per program)
  - Hämta r0-15, PSR från nästa program
  - Bx lr
- Byter helt tillstånd (stack, register, status)



Notering om stackar och LR i samband med subrutiner och avbrott

- Finns inget som kräver att återhoppsadressen som tas från stacken eller vid bx lr har placerats där av en BL och push {lr} eller avbrott
- Manipulation av LR möjlig
  - Direkt via t ex

    mov LR,#$1237 ; ersätt LR med ny adress
- Samma manipulation möjlig vid avbrottsåterhopp. Inget krav att ett ”äkta” avbrott skapat SR och PC kopior på stack
- Dvs bx lr hoppas att det är korrekta återhoppsadresser, men kan inte garantera det

Enklare exempel på kontextbyte, Cortex-M

- Exempelkod på hemsidan (fungerar på lab/distansutrustningen)
  - context_switch.asm (använder även tm4c123gh6pm_startup_ccs.c från lab2)
  - Detaljerad beskrivning av ide och struktur:
    - https://www.adamh.cz/blog/2016/07/context-switch-on-the-arm-cortex-m0/
- Byter om fjärrutrustningens tsea28lab2 trycker på vänster knapp
- Tre rutiner
  - Skriv ut ”Task 1 is running”
  - Skriv ut ”Task 2 is running”
  - Skriv ut alla tecken från '0' till 'z'
- Varje rutin känner bara till sig själv
  - Skriver ut sitt meddelande om och om igen
  - Vet inte när nästa rutin ska startas

```
task1:
        adr   r4,string1
task1_loop:
        ldrb  r0,[r4],#1
        ands r0,r0,r0
        beq task1 ; If at end restart!
        bl      printchar
        b       task1_loop

        .align 4
string1:
        .string       "Task 1 is running",13,10,0
```

Multitasking (processprogrammering)

- Mer information fås i kurser som Datorteknik och realtidssystem (för Y) och (antagligen) motsvarande kurs för U: Processprogrammering och operativsystem
- Blir ett svårare problem att lösa om flera program behöver bearbeta samma data
  - Låsa tillgång (jämför med buffered_getkey)
  - Vanliga program får inte ändra avbrottsnivå
- Många sätt att optimera prestanda och garantera svarstider
  - Vilket program ska väljas som nästa program att köra?