Eksamen 2012  
3. semester  
I3MPS - Microprocessor systemer  
Kasper Nissen

**Spørgsmål 5**

**Interrupts**

**Spørgsmål**

* Forklar de forskellige interrupttyper
* Hvad er de typiske opgaver for en interrupt service routine?
* Hvordan virker interrupt håndtering i Linux?
* Hvilke funktioner skal vi benytte i en driver for at understøtte interrupts?
* Hvordan kan vi optimere vores ISR?

**Pensum:**

* Linux Device Drivere – 147-155 og 258-273
* Simon 81-111
* Gpio text (afsnittet om gpio\_to\_irq og irq\_to\_gpio)

**Exercise 6**

**Hvorfor bruge interrupts?**

Uforudsigelige hændelser. (Vi ved ikke hvornår nogen ringer på døren? Hvornår nogen sender noget over et serielt kabel? )

Vi vil altså gerne have en metode til at bryde den synkrone eksekvering af kode, med en asynkron metode. Hvilket altså vil medføre et context-switch.

Det er tit nogle vigtige informationer udefra, som kræver omgående opmærksomhed.

**Interrupt typer**

Software interrupts

* Event skrevet kode, f.eks. GUI
* Exceptions, speciel fejl der bliver smidt et sted.

Hardware interrupts

* Internt  
  Interne interrupts kan f.eks. være at den interne UART giver besked at RX bufferen er fuld
* Eksternt  
  Interrupts fra eksterne enheder som er tilsluttet CPU’ens speciele interrupt pins

Andre typer interrupts

* Level-triggered – interupt på et bestemtniveau. F.eks. 5 V (F.eks. benzintank)
* Edge-triggered – interrupt på falling eller rising edge af en clock.

**Hvad er de typiske opgaver for en interrupt service routine?**

Når CPU’en modtager et interrupt request (IRQ) stopper microcontrolleren det den er i gang med, gemmer adressen på den næste kommando på stakken og hopper til interrupt service rutine. En typisk opgave for en interrupt service rutine også kaldet ISR, kan være:

* Rengøringsopgaver
* Genindstille interrupt detekt hardwaren i microcontrolleren, så den er klar til næste ISR.
* Hvis vi snakker et Interrupt fra f.eks. en serial port chip, som har modtaget en char fra serial porten, skal ISR’en læse denne karakter og overføre den til hukommelsen.
* En interrupt service routine kaldes også sommetider interrupt handler.

En typisk ISR skal gøre følgende:

* Bekræfte at der er et interrupt
* Vække enheden
* Hente data fra enhed
* Vågne en sovende function I driveren, som f.eks. read
* Exit med IRQ\_HANDLED eller IRQ\_NONE

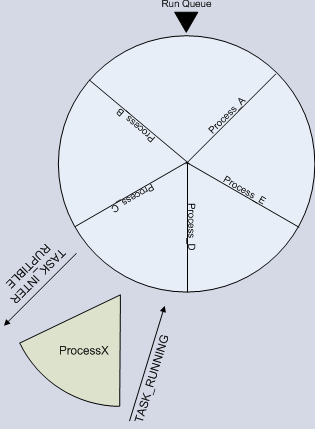
**Hvordan virker interrupt håndtering i Linux?**

Det vi ønsker er at et event skaber et interrupt i vores applikation, men vi ønsker ikke at hardwaren skal kende vores applikation, måden hvorpå man gør det er ved hjælp af Blocking I/O.

Blocking I/O

Blocking I/O er altså en metode hvorpå vi kan adskille driver og applikation. Hvad sker der hvis en driver f.eks. kalder read funktionen og der ikke er noget data tilgængeligt? Eller write hvis bufferen er fuld? Vi skal altså sørge for lave noget der blokere processen, ligge den til at sove, indtil den kan få lov til at udføre det den skal. Måden hvorpå dette er implemeneret i kernelspace, er ved at lade read og write metoderne sove indtil de skal bruges. Man kan gøre noget lignende i user space, ved at blokere fil adgangen, men vi kan jo ikke tilgå interrupts herfra.

Måden hvorpå man laver dette i linux er ved hjælp af en data struktur kaldet for wait queue. Det er en liste med processer som alle venter på et specifikt event. Linux kernen har en speciel kø til processer som alle venter på at blive eksekveret, de processer der ligger sig til at sove, bliver taget ud af denne run queue, og lagt i en wait queue:

  
Statisk allokering af wait queue:

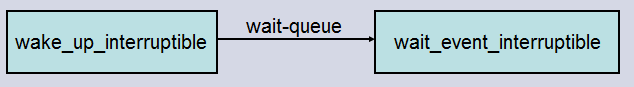


Dynamisk allokering af wait queue:

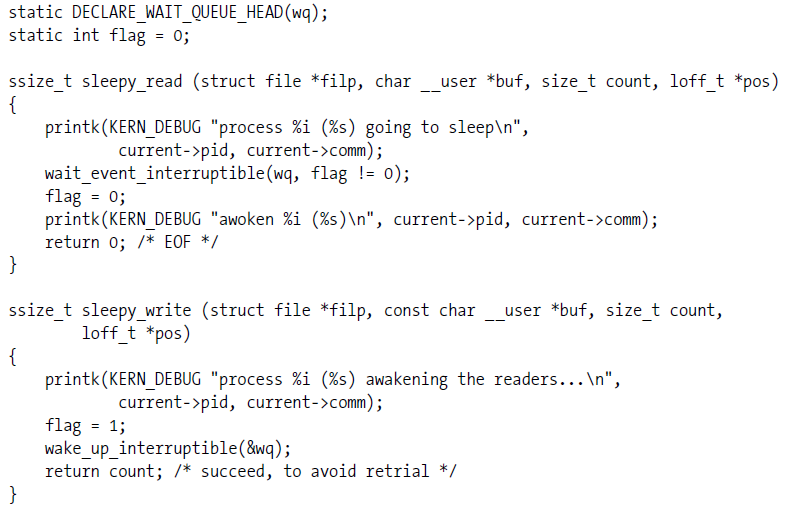


**Måden hvorpå vi laver sleep og wake up i Linux er:**

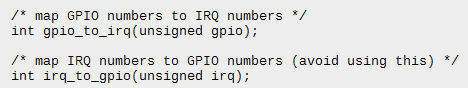
Der findes flere former for wait\_events i Linux, men den vi har brugt er den der hedder wait\_event\_interruptible. Wait\_interruptible får altså tråden til at sove, mens wake\_up\_interruptible er en metode til at få tråden til at vågne op på.

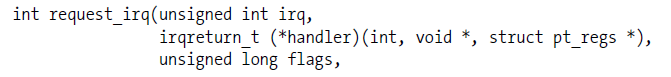


Eksempel på implementering af wait og wake i read og write metoder:



For at kunne bruge interrupts i en device driver, skal man først requeste IRQ linier. Et IRQ nummer er ligesom et GPIO nummer en unsigned integer, man kan nemt mappe et gpio nr. til et IRQ nummer og omvendt, ved hjælp af følgende metoder:



Når man har mappet sit irq nummer, skal linier requestes ved følgende metode: 



Et eksempel på det kunne være:

int result = request\_irq(irq\_nummer, ISR, IRQF\_TRIGGER\_RISING, "mygpio interupt is alive!!!", NULL);

Det første parameter, er det irq nummer vi vil requeste, den næste er den ISR routine vi vil kalde, dernæst hvad er det vi skal interrupte på, det næste er en eller anden form tekst som bliver skrevet ud, og den sidste kan bruges til hvis man vil sende noget med ned i sin ISR routine.

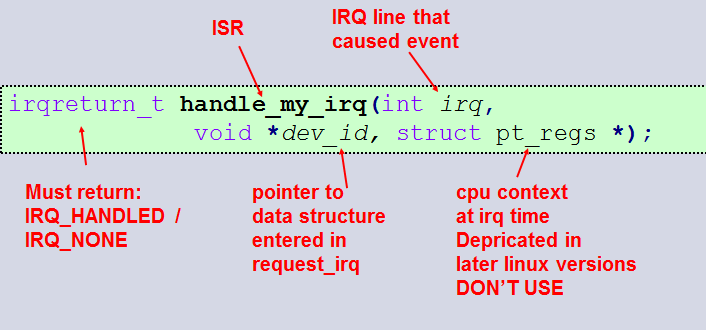
Når man requeste et irq nummer, sker dette jo i init-metoden, man skal derfor huske at frigive numret når modullet tages ud af kernen igen, dette gøres ved:



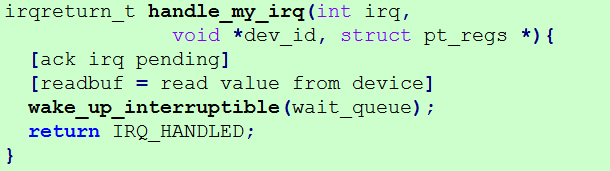
En bedre praksis er at requeste et irq nummer i open metoden, da hvis open ikke kan åbne skal vi jo ikke bruge IRQ’en alligevel, og linien vil altså blive optaget fra det sekund som kernen indsættes i modullet, og ikke fra open metoden, hvis den er i init. Derfor er det bedst at implementere IRQ reqeust i open-metoden. Ligeledes er det bedste sted at frigive irq i release/close metoden.

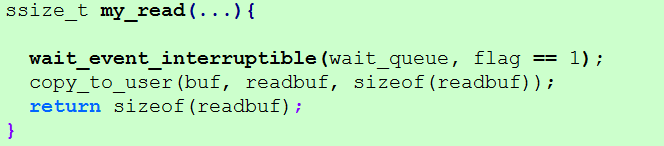
**ISR routinen**

Prototypen for selve ISR routinen er som følger:



**Eksempel på ISR routine:**

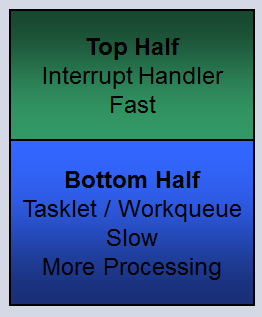




**Top and bottom halves**

En af de store problemer med interrupts er hvor lang tid det tager at udføre koden i interrupten. For ikke at blokere hele programmet hvergang der kommer et interrupt, skal man sørge for at lave så hurtig kode som muligt så vi ikke blokere for lang tid. Der er altså 2 ting vi skal opveje mod hinanden work og speed.

I Linux løser dette ved at splitte interrupt handleren op i 2 halve, top half er den egentlige ISR routine, som bliver kørt ved et interrupt, hvorimod bottom half er en routine som top half schedulere til at kører på et mere sikkert tidspunkt.



* Top-Half:
  + ISR created with request\_irq()
  + Ack device irq\_pending
  + Read device data to buffer
  + Schedule tasklet/workqueue
  + IRQ can be disabled
* Bottom-Half
  + Scheduled by the kernel
  + Handles all heavier processing
  + IRQs enabled during processing
  + Implemented as tasklet or workqueue