TSEA81 Datorteknik och realtidsystem

Föreläsning 6

LINKÖPINGS UNIVERSITET

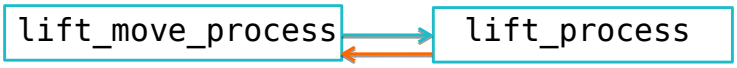
Agenda

- Reflektion Lab4
- Information Lab5
- Scheduling + Real-time Linux
- Information till Lab6
- Tentan

Reflektion Lab 4

Reflektion Lab4

- Ändrade väntetider i liftmove_process eller lift_process:




```
while(1) {
  sleep(4);
  send_message(LIFT_MOVE);
}
```

```
case LIFT_MOVE:
  sleep(1); // ”close doors”
  lift_next_floor(...);
  lift_move(...);
  sleep(1); // ”open doors”
  ...
```

- Vad händer om `lift_move_process` skickar ett `LIFT_MOVE`-meddelande varannan sekund (eller oftare) istället för var fjärde sekund?
- Jo, `lift_process` hinner ju inte med. Meddelanden läggs på hög.
- En snygg lösning vore kanske ett returmeddelande, t ex `LIFT_MOVE_DONE`

Information Lab 5

Information Lab5

- Mät tiden så här för Lab3 och Lab4:

```
for (x=10; x<=???; x+=10)
{
  gettimeofday(&starttime, NULL);
  // make 10000 journeys each
  // with x persons
  gettimeofday(&endtime, NULL);
  timediff = ...
  printf(timediff);
}
```

- Mät tiden så här för individuella resor:

```
#define MAX_LOG 10001
struct timespec timestamp[MAX_LOG];
For (i=0; i<10000; i++) {
  clock_gettime(CLOCK_MONOTONIC,
                &timestamp[i]);
  lift_travel(…);
}
clock_gettime(CLOCK_MONOTONIC,
                &timestamp[i]);
```

Scheduling

Priority based scheduling

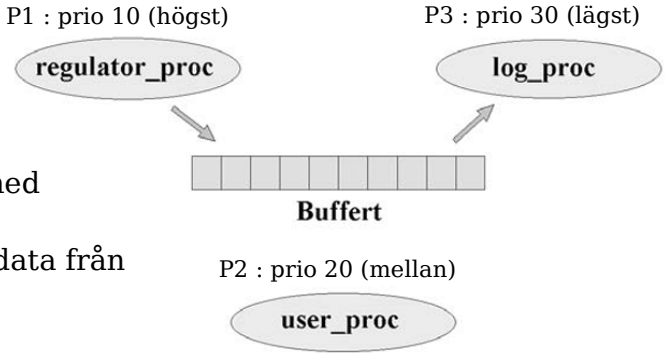
- **Prioritetsbaserad schemaläggning**
  - Den process, som är redo, med högst prioritet får köra
  - Om flera processer har samma prioritet:
    - Tidskvanta : den som varit körklar längst får köra (s k Round-Robin scheduling, körklara processer med samma prioritet får dela på CPU-tiden)
    - Köordning : den som ligger först i Ready-lista får köra (FIFO : First In First Out)
  - Användbar för både statisk och dynamisk prioritet

Frivillig / non-preemptive (cooperative):
- Processen avslutas bara frivilligt, dvs man får vänta på att den blir klar

Påtvingad / preemptive:
- Processbyte kan ske när som helst, t ex via klockavbrott

Priority based preemptive scheduling är vanligt i realtidssystem, just för att få bättre kontroll även *när* processer ska köras.

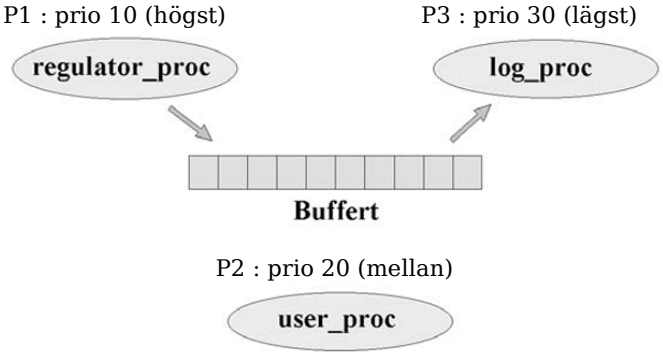
Prioritetsinversion



- **P1: Regulator_proc**, prio 10 (högst) skriver data till buffert
- **P2: User_proc**, prio 20, interagerar med användaren, använder inte buffert
- **P3: Log_proc**, prio 30 (lägst), loggar data från reglersystemet, använder buffert
- Antag följande scenario:
  - **log_proc** har tillgång till buffert och kör
  - **user_proc** blir aktiv, och får köra eftersom **user_proc** har högre prio än **log_proc**
  - **regulator_proc** blir körklar, vill skriva i buffert, men får vänta då buffert är reserverad av **log_proc**
  - nu använder inte **user_proc** buffert, men den är reserverad, så **regulator_proc** får vänta indirekt även på **user_proc** (med lägre prio) som är den som nu kör
- Fenomenet kallas *prioritetsinversion (priority inversion)*

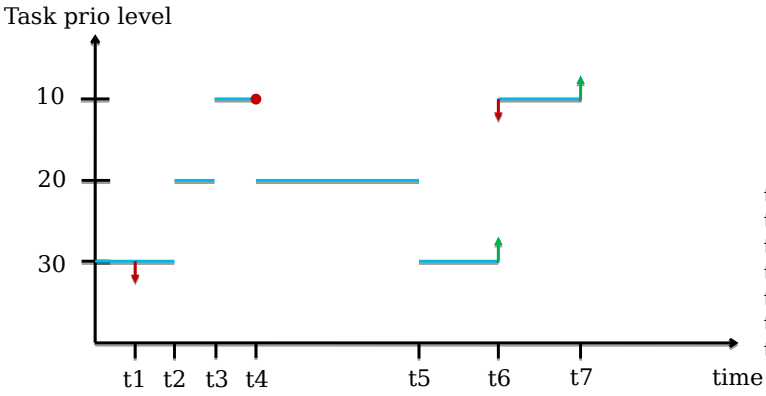
LINKÖPINGS UNIVERSITET

Prioritetsinversion



↓ Reservera semafor

↑ Släpp semafor

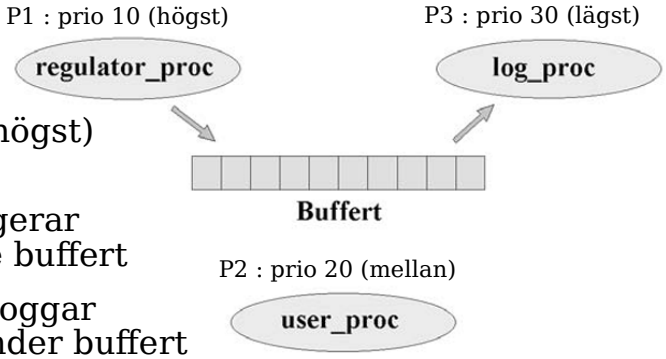


t1: P3 reserverar semafor
t2: P2 avbryter P3 (preempted)
t3: P1 avbryter P2 (preempted)
t4: P1 försöker reservera semafor (prio inversion)
t5: P2 klar, P3 återupptas
t6: P3 släpper semafor, P1 reserverar semafor
t7: P1 släpper semafor

LINKÖPINGS UNIVERSITET

Tips: googla på ”mars pathfinder priority inversion”
https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

Prioritetsärvning

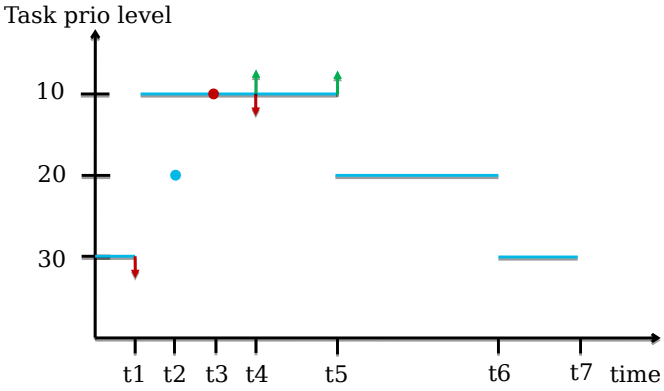


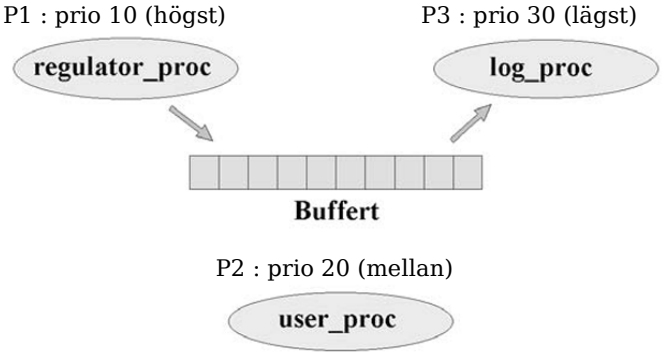
- **P1: Regulator_proc**, prio 10 (högst) skriver data till buffert
- **P2: User_proc**, prio 20, interagerar med användaren, använder inte buffert
- **P3: Log_proc**, prio 30 (lägst), loggar data från reglersystemet, använder buffert
- Lösning: *Prioritetsärvning*: Höj tillfälligt (under användning av buffert) prio på **log_proc** till samma prio som **regulator_proc**, dvs 10. Då kan **log_proc** köra klart (utan avbrott från **user_proc**), sedan släppa buffert (samtidigt återta sin tidigare prio, dvs 30) varpå **regulator_proc** kan köra.
- Därmed slipper **regulator_proc** vänta onödigt länge.

Prioritetsärvning

↓ Reservera semafor

↑ Släpp semafor

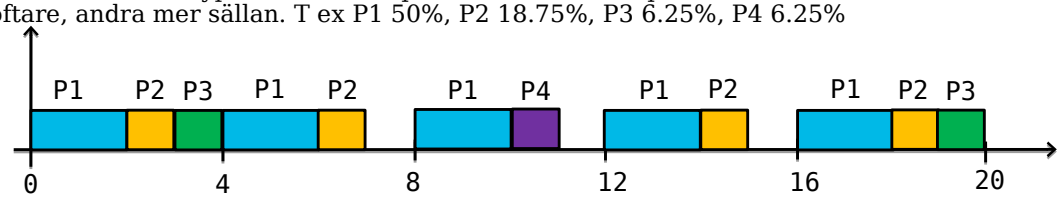




t1: P3 reserverar semafor, och prio höjs till samma som P1
t2: P2 blir redo försöker köra, men P3 har nu högre prio
t3: P1 försöker reservera semafor, blockerad då P3 har den
t4: P3 släpper semafor, P1 reserverar semafor
t5: P1 släpper semafor, P2 får köra
t6: P2 klar, P3 får köra
t7: P3 klar

Time-driven scheduling

- **Schemaläggning sker vid bestämda tidpunkter**
  - **Static Cyclic Scheduling** (cyclic executive):
  - Processer anropas som funktioner i en viss ordning med ett visst periodiskt tidsintervall, ofta orsakat av ett klockavbrott.
  - Processerna har typiskt olika anropsfrekvenser för att på så sätt fördela körtiden. Vissa kör oftare, andra mer sällan. T ex P1 50%, P2 18.75%, P3 6.25%, P4 6.25%



  - Periodiska anrop kräver att processerna kör färdigt inom sitt tidsintervall eller att arbetet kan delas upp i delar. Alternativt kan man använda ett background/foreground-system, där tidskritiska/periodiska processer styrs via tidsavbrott och övriga processer körs i bakgrunden.
  - Nackdel: En reserverad gemensam resurs kan hindra efterkommande processer att köra, och tidsintervallet blir outnyttjat, dvs det går förlorat.

Körordning

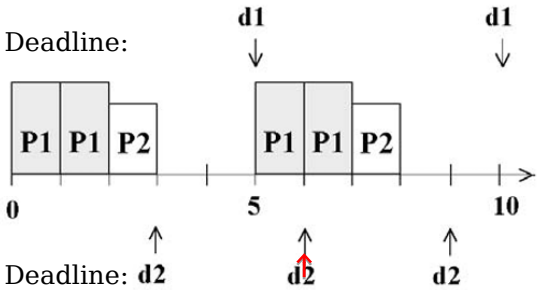
```
0:P1
1:P1
2:P2
3:P3
4:P1
5:P1
6:P2
7:[ ]
8:P1
9:P1
10:P4
11:[ ]
12:P1
13:P1
14:P2
15:[ ]
```


Periodiska processer

- Periodiska processer ska starta med en viss periodicitet Det finns tidskrav på att processens arbete ska slutföras inom en föreskriven sluttid, en **dead-line**
- Låt e vara tiden det tar att köra do_work()
  Låt w vara tiden processen väntar
  Låt p = e+w vara periodtiden
- Processens CPU-utnyttjande blir e/p
- Systemets CPU-utnyttjande blir $U_e = \sum_{i=1}^{n} \left(\frac{e_i}{p_i}\right)$
- **Antag två processer**:
- -P1 skall under tidsintervallet [i5, (i+1)5] köra 2 tidsenheter (i är ett heltal >=0)
- -P2 skall under tidsintervallet [i3, (i+1)3] köra 1 tidsenhet (i är ett heltal >= 0)
- Utnyttjandegraden blir `2/5 + 1/3 = 11/15 ≈ 0.73`

```
// princip för periodisk process
while (1) {
    do_work(); // e time units
    sleep(w);  // w time units
}
```

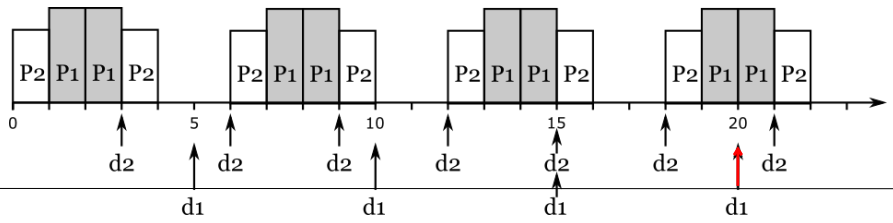
Periodiska processer



- **Antag att prioritetsbaserad påtvingad schemaläggning används**
- **Antag att P1 har högst prioritet**
- Följande händer:
  - Båda processerna anses körklara vid tidpunkt 0, och P1 får köra 2 tidsenheter och inleder sedan väntan i 3 tidsenheter
  - Vid tidpunkt 2 får P2 köra 1 tidsenhet och inleder sedan väntan i 2 tidsenheter
  - Båda processerna möter/klarar sina dead-lines/tidskrav under första varvet
  - Båda processerna blir åter körklara vid tidpunkt 5, och P1 (högst prio) får köra
  - Eftersom P1 under andra varvet är klar först vid tidpunkt 7, så missar P2 sin dead-line vid tidpunkt 6
- Mönstret upprepar sig efter 5 tidsenheter (för då är båda processerna körklara igen) och den verkliga utnyttjandegraden blir = 3/5 = 0.60 (jfr med förväntad =0.73)

Periodiska processer

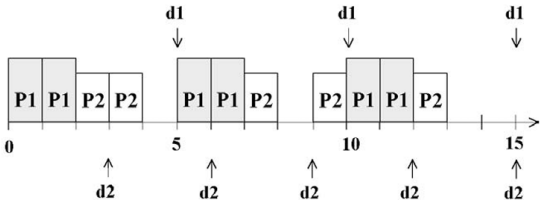
- Antag att prioritetsbaserad påtvingad schemaläggning används
- **Antag att P2 har högst prioritet**
- Följande händer:
  - Båda processerna anses körklara vid tidpunkt 0, och P2 får köra 1 tidsenhet och inleder sedan väntan i 2 tidsenheter
  - Vid tidpunkt 1 får P1 köra 2 tidsenheter och inleder sedan väntan i 3 tidsenheter
  - Vid tidpunkt 3 kör P2 1 tidsenhet och väntar 2 tidsenheter
  - Båda processerna blir åter körklara vid tidpunkt 6, och P2 (högst prio) får köra
  - P2 klarar sina deadlines, men P1 missar sin deadline vid t=20
  - Mönstret upprepar sig alltså efter 6 tidsenheter och den verkliga utnyttjandegraden blir = 4/6 = 0.67 (jfr med förväntad =0.73)



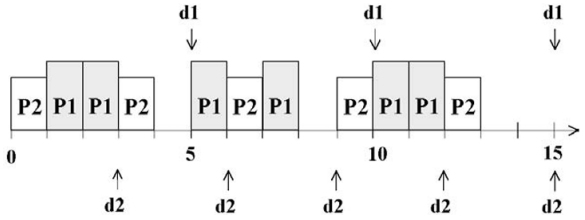
Periodiska processer

- Systemet klarar inte tidskraven för bådaprocesserna då väntetiden är konstant, bättre vore att vänta fram till periodtidens slut:
  - P1 högst prio: = 11/15 0.73

```
// bättre princip för
// periodisk process
const int TASK_PERIOD=...;
t = get_time();
while (1) {
    do_work(); // e time units
    t = t + TASK_PERIOD;
    sleep_until(t);
}
```



P1 högst prio



P2 högst prio

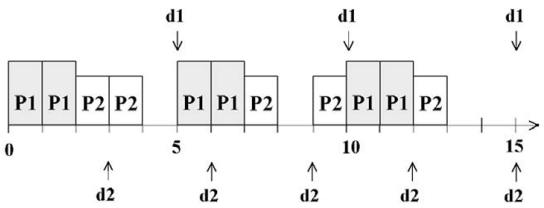
Periodiska processer

- Systemet klarar inte tidskraven för båda Processerna då väntetiden är konstant, bättre vore att vänta fram till periodtidens slut:

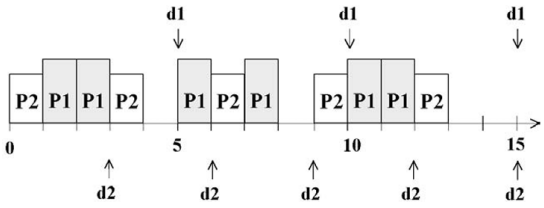
  -P1 högst prio: = 11/15 0.73

  -P2 högst prio: = 11/15 0.73

```
// bättre princip för
// periodisk process
const int TASK_PERIOD=...;
t = get_time();
while (1) {
    do_work(); // e time units
    t = t + TASK_PERIOD;
    sleep_until(t);
}
```



P1 högst prio



P2 högst prio

Periodiska processer : EDF

- **EDF – Earliest Deadline First**:

  Den körklara process som har kortast tid kvar till nästa sluttid får köra.
  - Den *relativa sluttiden* definieras som tiden från körklar fram till nästa sluttid.
  - Då de relativa sluttiderna sammanfaller med processernas periodtider och förväntad $U_e \leq 1$ kommer EDF leda till att processerna håller sina sluttider. [Liu och Leyland, 1973]
- Förutsättning: att påtvingad schemaläggning används.

Tid	Δd_1	Δd_2	*Process*	*Kommentar*
0-1	5	3	*P2*	*P2* kortast tid till sluttid
1-2	4	2	*P1*	*P2* ej körklar
2-3	3	1	*P1*	*P2* ej körklar
3-4	2	3	*P2*	*P1* ej körklar
4-5	1	2		*P1* och *P2* ej körklara
5-6	5	1	*P1*	*P2* ej körklar
6-7	4	3	*P2*	*P2* kortast tid till sluttid
7-8	3	2	*P1*	*P2* ej körklar
8-9	2	1		*P1* och *P2* ej körklara
9-10	1	3	*P2*	*P1* ej körklar
10-11	5	2	*P1*	*P2* ej körklar
11-12	4	1	*P1*	*P2* ej körklar
12-13	3	3	*P2*	*P1* ej körklar
13-14	2	2		*P1* och *P2* ej körklara
14-15	1	1		*P1* och *P2* ej körklara

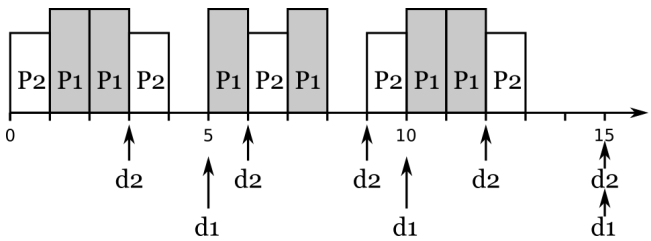
P1: kör 2 tidsenheter i intervallet `[i5,(i+1)5]`
P2: kör 1 tidsenhet i intervallet `[i3,(i+1)3]`

Periodiska processer : EDF

- EDF – Earliest Deadline First:

  Den körklara process som har kortast tid kvar till nästa sluttid får köra.
  - Vid varje tidpunkt ställ frågan:

    Vilka processer är redo att köra, och vilken av dessa har kortast tid kvar till sin deadline?
  - Om flera processer (som är redo att köra) har samma tid kvar till sin respektive deadline spelar det ingen roll vem som kör, men om en av dom redan kör låter man den fortsätta för att slippa ett processbyte.



P1: kör 2 tidsenheter i intervallet `[i5,(i+1)5]`
P2: kör 1 tidsenhet i intervallet `[i3,(i+1)3]`

Periodiska processer : RMS

RMS – Rate Monotonic Scheduling:

- Processer får prioritet utefter hur ofta de ska köras. Dvs, kortare periodtid ger högre prioritet.
- Därefter används prioritetsbaserad påtvingad schemaläggning.
- Då RMS och påtvingad schemaläggning används för n processer där de relativa sluttiderna sammanfaller med processernas periodtider och utnyttjandegraden uppfyller $U_e \leq n(2^{1/n}-1)$, så kommer processernas sluttider att hållas. [Liu och Leyland, 1973]
- T ex: För processerna P1 och P2 med RMS gäller: $2(2^{1/2}-1) \approx 0.83$
- P2 har kortast periodtid och om P2 får högst prio uppfylls tidskraven.
- Då $n \rightarrow \infty$ så $n(2^{1/n}-1) \rightarrow \ln 2 \approx 0.69$
- Dvs, tumregel: Om CPU-utnyttjandet är mindre än 69% kommer tidskraven att uppfyllas.
- Förutsättning: processerna får inte vara beroende av varandra, t ex via mutexar som orsakar prioritsinversion.

Periodiska processer

- Deadlines för processerna är ofta given i krav/specifikation.
  Hur får man tag i körtiderna?
- För simpla mikrokontrollers (AVR el liknande):
  - Räkna klockcykler i den assemblerade koden, och ta hänsyn till eventuella avbrott
- För en mer avancerad processor:
  - Svårt att mäta/beräkna tiden exakt:
    - Cacheminne och DMA-access påverkar åtkomsttider
    - Pipelining, branch prediction, out-of-order execution påverkar cykeltider
    - Synkronisering av data mellan flera kärnor är svårberäknad
  - Slutsats: Räkna med att få ut ungefärliga tider
- Vid mycket hårda realtidskrav, använd FPGA och bygg en speciallösning för uppgiften

Scheduling : Realtidssystem

Slutliga tips, vid konstruktion av realtidssystem:

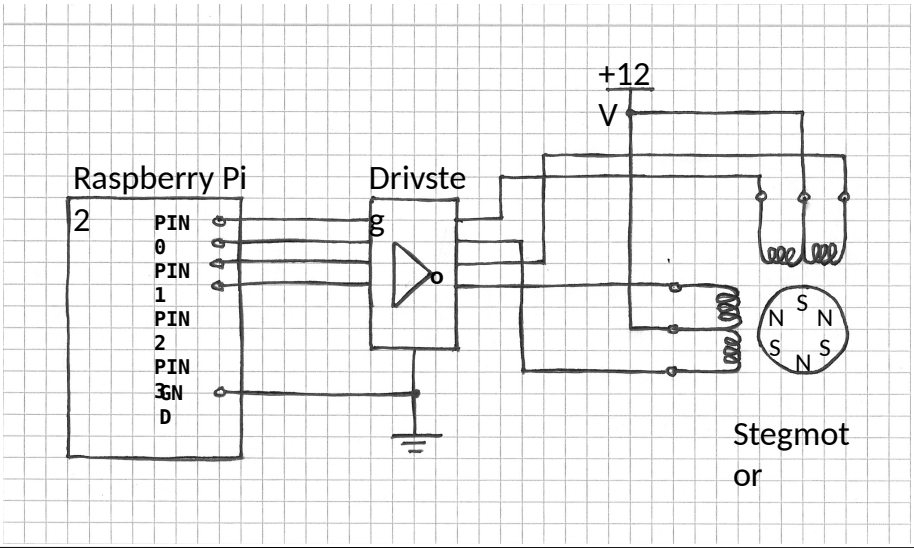
- Håll schemaläggningen enkel. Går det att använda statisk schemaläggning utan trådar/processer: Gör det!
  - Använd bara trådar om nödvändigt.
- Använd inte onödigt många lås, tillståndsvariabler, semaforer och dylikt.
  - Håller man det enklare blir det lättare att felsöka.
- Ha gärna gott om marginal för processorutnyttjande, dvs snåla inte med processorkraften vid val av processor. Det blir förstås dyrare vid stora produktionsserier, men felsökningstid kostar också pengar.
- Det kan ofta vara lönsamt att flytta över realtidsfunktionaliteten till en enklare processor (typ AVR) som kör ett enklare realtids-OS (om ens det).
- Vid mycket hårda realtidskrav, använd ren hårdvara (t ex en FPGA).

Real-time Linux

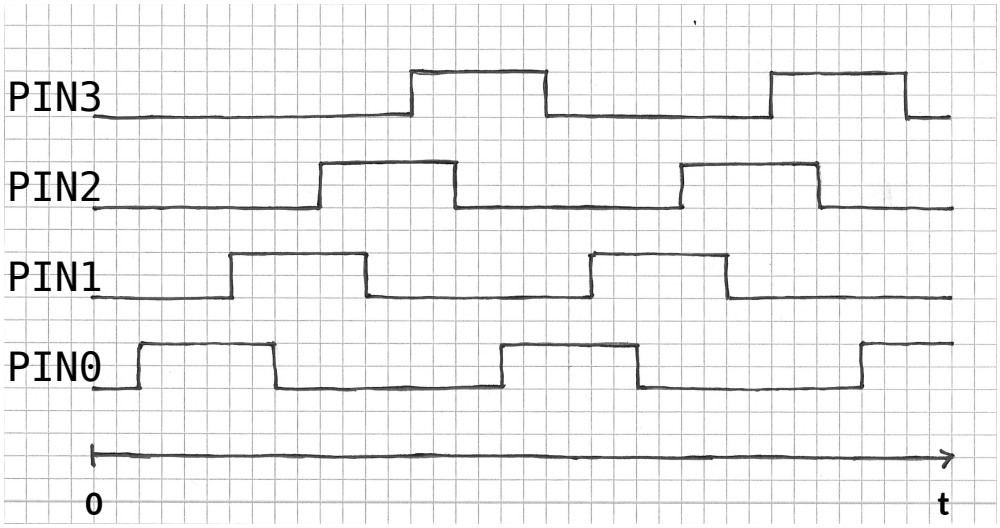
Real-time Linux

- Olika sätt att åstadkomma Real-time Linux:
  - Linux 2.6 har CONFIG_PREEMPT option, dvs även processer som gör systemanrop kan preemptas.
  - Linux 2.6 kan köras med O(1)-scheduler, dvs schemaläggning kan göras på konstant tid, och CFS (Completely Fair Scheduler) som maximerar CPU-utnyttjande och interaktivitet.
  - Patch : CONFIG_PREEMPT_RT, gör så att hela Linux-kärnan kan preemptas.
  - Thin kernel : Kör Linux som en lågprioriterad process i en s k thin kernel, vilken kör som ett ”RTOS”.

Stegmotor-demo : uppkoppling



Stegmotor-demo : uppkoppling



Stegmotor-demo : Linux GPIO

```
// Export the pin to the GPIO directory
FILE *fp=open("/sys/class/gpio/export","w");
fprintf(fp,"%d",gpioport);
fclose(fp);

// Set the pin as an output
char filename[256];
sprintf(filename,"/sys/class/gpio/gpio%d/direction",gpioport);
fp = fopen(filename,"w");
if(!fp){
  panic("Could not open gpio file");
}
fprintf(fp,"out");
fclose(fp);
```

Stegmotor-demo : Linux GPIO

```
// Open the value file and return a pointer to it.
sprintf(filename,"/sys/class/gpio/gpio%d/value",gpioport);
fp = fopen(filename,"w");
if (!fp) {
  panic("Could not open gpio file");
}
return fp;

// Set the I/O pin to the specified value.
void setiopin(FILE *fp, int val) {
  fprintf(fp,"%d\n",val);
  fflush(fp);
}
```


Stegmotor-demo : stepper0.c

```
main()....
{
  FILE *pin0 = init_gpio(14);
  FILE *pin1 = init_gpio(15);
  FILE *pin2 = init_gpio(17);
  FILE *pin3 = init_gpio(18);
  signal(SIGINT, dumptimestamps);                          // see ”man signal”
  while(1){                                                // and ”man 7 signal”
    usleep(delay); logtimestamp(); setiopin(pin0,1);
    usleep(delay); logtimestamp(); setiopin(pin3,0);
    usleep(delay); logtimestamp(); setiopin(pin1,1);
    usleep(delay); logtimestamp(); setiopin(pin0,0);
    usleep(delay); logtimestamp(); setiopin(pin2,1);
    usleep(delay); logtimestamp(); setiopin(pin1,0);
    usleep(delay); logtimestamp(); setiopin(pin3,1);
    usleep(delay); logtimestamp(); setiopin(pin2,0);
  }
}
```

Stegmotor-demo : stepper0.c

- Problem med stepper0.c:
  - Vi väntar en relativ tid med usleep()
  - Tiden mellan varje händelse på utsignalerna blir delay+X, där X beror på körtiden för logtimestamp() och setiopin()
- Lösning:
  - Använd clock_nanosleep

Stegmotor-demo : stepper1.c

```
void sleep_until(struct timespec *ts, int delay)
{
  ts->tv_nsec += delay;
  if(ts->tv_nsec > 1000*1000*1000){
    ts->tv_nsec -= 1000*1000*1000;
    ts->tv_sec++;
  }
  clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, ts,  NULL);
}


struct timespec {
  time_t tv_sec;    // seconds
  long tv_nsec;     // nanoseconds [0 .. 999999999]
};
```

Stegmotor-demo : stepper1.c

```
main()....
{
  FILE *pin0 = init_gpio(14);
  FILE *pin1 = init_gpio(15);
  FILE *pin2 = init_gpio(17);
  FILE *pin3 = init_gpio(18);
  signal(SIGINT, dumptimestamps);
  clock_gettime(CLOCK_MONOTONIC, &ts);
  while(1){
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin0,1);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin3,0);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin1,1);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin0,0);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin2,1);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin1,0);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin3,1);
    sleep_until(&ts, delay); logtimestamp(); setiopin(pin2,0);
  }
}
```


Stegmotor-demo : stepper1.c

- Problem med stepper1.c:
  - Utsignalerna kommer även nu att uppvisa kraftigt jitter (fasbrus) på grund av övrig aktivitet i systemet, särskilt vid hög belastning.
- Lösning:
  - Sätt realtidsprioritet

Stegmotor-demo : stepper2.c

```
main()....
{

  ....

  // SCHED_FIFO: Processen kör tills en högre prioriterad process
  // vill köra. Finns även med SCHED_RR.
  // Kräver oftast root!
  struct sched_param sp;
  sp.sched_priority = 30;
  pthread_setschedparam(pthread_self(), SCHED_FIFO, &sp);

  While(1) {
    // samma som i stepper1.c
  }
}
```


Stegmotor-demo : stepper2.c

- Problem med stepper2.c:
  - Andra saker kan påverka realtidsprestandan:
    - Minnesanvändning hos andra processer kan störa realtidsprocesser.
    - Processen får inte bli utswappad, dvs dess använda minne får inte läggas ut i swap-minnet (swap space) som har ”lång” access-tid.
  - Linux använder ”overcommit” vid minnesanvändning:
    - malloc() garanterar inte att det minne som allokeras faktiskt existerar. Man får ett virtuellt minne tilldelat, och minnet allokeras fysiskt egentligen först när det används.
- Lösning:
  - Lås minnet till processen med mlockall()

Stegmotor-demo : stepper3.c

```
main()....
{

  ....

  // CURRENT: Nuvarande sidor låses
  // FUTURE: Framtida allokeringar låses
  // Kräver oftast root!
  if (mlockall(MCL_FUTURE|MCL_CURRENT)) {
    fprintf(stderr,”WARNING: Failed to lock memory\n”);
  }

  ....

  While(1) {
    // samma som i stepper2.c
  }
}
```


Stegmotor-demo : användarinmatning

- Problem:
  - Hur matar vi in data *till* programmet (stepper_thread) utan att blockera? Vi kan ju självklart **inte** göra så här:

```
while(1) {
  sleep_until(&ts, delay); logtimestamp(); setiopin(pin0,1);

  ....

  sleep_until(&ts, delay); logtimestamp(); setiopin(pin2,0);

  printf(”Enter new delay value: ”);
  scanf(”%d”, &delay);
}
```

- Lösning:
  - Använd non-blocking I/O (stökigt!) eller ännu bättre, gör inmatning av data i en annan tråd och kommunicera med realtidstråden via symmetrisk synkronisering, dock ej blockerande sådan!

Stegmotor-demo : stepper.c

```
struct stepper {
  sem_t sem;
  sem_t rt_sem;
  int delay;
  ...
};

void *user_thread(void *arg) {
  struct stepper *step =
    (struct stepper *) arg;
  while(1) {
    // get_delay läser indata
    step->delay = get_delay();
    sem_post(&step->sem);
    // invänta stepper_thread
    sem_wait(&step->rt_sem);
  }
}
```

```
void *stepper_thread(...) {
int delay; // lokal kopia av step->delay
  while(1) {
    sleep_until(&ts, delay);
    setiopin(pin0,1);
    ...
    sleep_until(&ts, delay);
    setiopin(pin2,0);
    // om vi inte kan låsa fortsätter vi
    if (!sem_trywait(&step->sem)) {
      delay = step->delay;
      // Acknowledge
      sem_post(&step->rt_sem);
    }
    ...
```


Stegmotor-demo : stepper.c

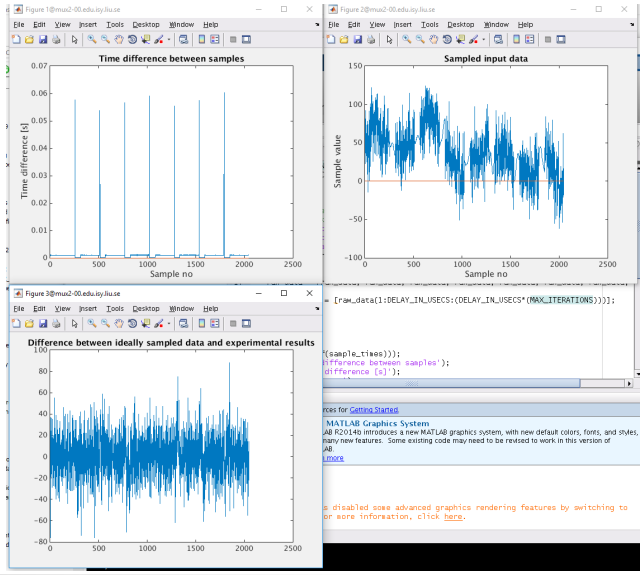
- Problem:
  - Hur hämtar vi data från stepper_thread?
- Lösning:
  - Inför fler fält och flaggor i structen:

```
struct stepper {
  sem_t sem;
  sem_t rt_sem;
  int delay;
  int get_steps; // flagga 0/1
  int num_steps; // antal steg
  ...
};
```

```
void *stepper_thread(...) {
  int num_steps;
  ...
  if (!sem_trywait(&step->sem)) {
    delay = step->delay;
    if (step->get_steps) {
      step->get_steps = 0;
      step->num_steps = num_steps;
    }
  // Acknowledge
  sem_post(&step->rt_sem);
  }
```

Lab 6 : Real-time Linux

Lab6 : Real-time Linux




```
void *maintask(void)
{
  for (i=0; i<2048; i++) {
    sleep(1 ms);
    sample_buffer[i]=
      read_sample(channel);
    if (i==256,512,768...) {
      //signal do_work_task
    }
  }
}
```

```
void *do_work_task(void)
{
  //wait for signal
    do_work(sample_buffer);
}
```

Tentan

Tentan

- Motivera svaren
- Kommentera programkoden
- Är man osäker på det exakta namnet för en funktion, skriv vad ni tror och kommentera vad den gör så att vi förstår vad ni menar.
- Syntaxen är inte superviktig. Om programkoden kan tolkas entydigt så är det okej. Dvs, använd aldrig pseudokod, om det inte specifikt efterfrågas.
- Skriv läsbart! Oläsbar text kan inte bedömas.
- Lycka till!

Ingen Fö 12/12

- Labba istället

LINKÖPINGS UNIVERSITET