

**Lösningförslag till tentamensskrivning —
Realtidsprogrammering — Java 2001-03-07, kl 14.00-19.00**

Uppgifter (vissa sammanfattade) med lösningar

1. Vad är prioritetsinversion? Illustrera med ett enkelt exempel. Redogör kortfattat för hur prioritetsinversion kan undvikas. (2p)

Prioritetsinversion är ett fenomen som uppstår då en högprioriterad tråd blir blockerad av en lägre prioriterad tråd utan att de (för tillfället) delar på någon resurs. Exempel: En lägprioriterad tråd exekverar och tar en resurs. En högprioriterad tråd börjar exekvera och försöker i sin tur ta resursen, vilket misslyckas eftersom den är låst. Den högprioriterade tråden blockerar i väntan på att resursen ska släppas av den lägprioriterade tråden. Under tiden blir en tredje, mellanprioriterad tråd körbar och tar över CPU:n i kraft av sin prioritet och blockerar den lägprioriterade tråden. Indirekt blockeras därmed även den högprioriterade som får vänta på den lägre prioriterade (mellanprioriterade) tråden.

Lösningen på problemet är att använda något slags prioritetsärvningsprotokoll som gör att prioriteten för den lägprioriterade tråden tillfälligt höjs när den blockerar en mer högprioriterad tråd. På så vis kan den mellanprioriterade tråden inte komma in och ta över CPU:n medan resursen som den högprioriterade tråden vill ha är låst.

2. I ett realtidssystem har vi oftast strikta krav på korta svarstider. Detta innebär till exempel att vi måste kunna garantera att systemet svarar på viktiga insignaler inom en kort tidsrymd.

a) Ange ett krav som ställs på schemalagningen i en realtidskärna som skall klara strikta/hårda tidskrav, men som vi inte ställer på tidsdelning utan realtidsskrav. (1p)

Vi måste fördela CPU-tiden strikt efter trådarnas prioritet. Högprioriterade trådar måste alltid få företräde före mera lägprioriterade för att vi ska kunna garantera korta svarstider för dessa.

b) Vadställer det för krav på hurensemafor och dess köavväntande trådar implementeras i systemet? (1p)

För att inte högprioriterade trådar ska behöva vänta godtyckligt lång tid på att komma in i monitorn respektive bli väckta efter en händelse måste semaforens vänteköer vara ordnade i prioritetsordning.

3. Vilka fyra villkor måste vara uppfyllda för att dödläge ska kunna uppstå? (2p)

De fyra villkoren är:

- Mutual exclusion - ömsesidig uteslutning. Endast en tråd tillåts använda en gemensam resurs åt gången, vilket medför att en annan tråd kan behöva vänta.
- Hold-and-wait, dvs att en tråd ska kunna ta en resurs och sedan blockeras när den försöker ta ytterligare en.
- No resource preemption - ingen yttre mekanism som kan tvinga en tråd att släppa en resurs i förtid.
- Circular wait - cirkulär väntan. Det måste finnas en kedja av hold-and-wait-situationer som är cirkulär.

4. En programmerare har skrivit följande programrader för att åstadkomma ömsesidig uteslutning (mutual exclusion) runt anropet av operationen "doSomething".

```
while (mutex==1); // Wait for mutex lock to become free
mutex = 1;        // Acquire mutex lock
doSomething();   // Perform task
mutex = 0;        // Release lock
```

Variabeln `mutex` är en heltalsvariabel som är deklarerad "`int mutex = 0;`".

a) Vad brukar man kalla sättet att vänta på att den delade resursen ska bli tillgänglig ("`while (mutex==1);`")? Vilken nackdel har denna metod? (1p)

Man kallar detta sätt att vänta för "busy-wait". Nackdelen är att man ödslar en mängd CPU-tid på att inte göra annat än att vänta på att villkoret blir uppfyllt. I värsta fall kan det innebära att den tråd som ska uppfylla villkoret aldrig får köra - den kanske har lägre prioritet än den väntande tråden.

b) Garanterar koden ovan ömsesidig uteslutning över anropet av `doSomething`? Motivera ditt svar. (1p)

Nej, koden garanterar ej ömsesidig uteslutning. Antag att två trådar samtidigt försöker gå in i den kritiska regionen. Tråd ett exekverar `while`-satsen och konstaterar att resursen är ledig, men innan den hinner markera resursen som upptagen ("`mutex = 1;`") blir det ett trådbyte och tråd två kommer igång. Den exekverar också `while`-satsen och eftersom `mutex` fortfarande är lika med noll fortsätter tråd två och går in i operationen `doSomething`. Medan tråd två exekverar `doSomething` blir det ett nytt trådbyte och tråd ett fortsätter även den in i `doSomething`.

5. Förutom eventuella skillnader i schemalaggningsen, nämn två skillnader mellan realtids-trådar (även kallade lättviktsprocesser) och processer (även kallade OS-processer). (1p)

- Trådar kommunicerar normalt via delat minne i samma adressrymd, vilket innebär att man rent tekniskt kan referera till gemensamma variabler/objekt på samma sätt som i sekventiella program. Processer kommunicerar normalt via sekundärminne (filer) eller serialiserat via sockets och pipes. (Direkt primärminneskommunikation är oftast möjligt (via systemanrop såsom `mmap`) men minnet nås då från olika adressrymder vilket försvårar programmering och synkronisering.)
- Till en OS-process hör resursallokering på systemnivå såsom öppna filer, minnesrymder och användarrättigheter. Inom en OS-process kan finnas flera trådar som då inte har egna OS-resurser utan delar den omgivande processens resurser. Trådbyte blir därför billigare än processbyte.
- Trådar opererar normalt sett i en snabbare tidsskala än processer, typiskt snabbare respektive långsammare än 10ms.

6. I ett realtidssystem finns tre stycken trädar, T1, T2 och T3. Deras run-metoder innehåller nedanstående kodsekvenser. S1, S2, S3, S4 och S5 är alla semaforer med initialvärdet ett (1).

```

T1.run()
S5.take();
useS5();
S5.give();
S2.take();
S5.take();
useS2S5();
S2.give();
useS5();
S5.give();
S1.take();
S2.take();
useS1S2();
S2.give();
S1.give();

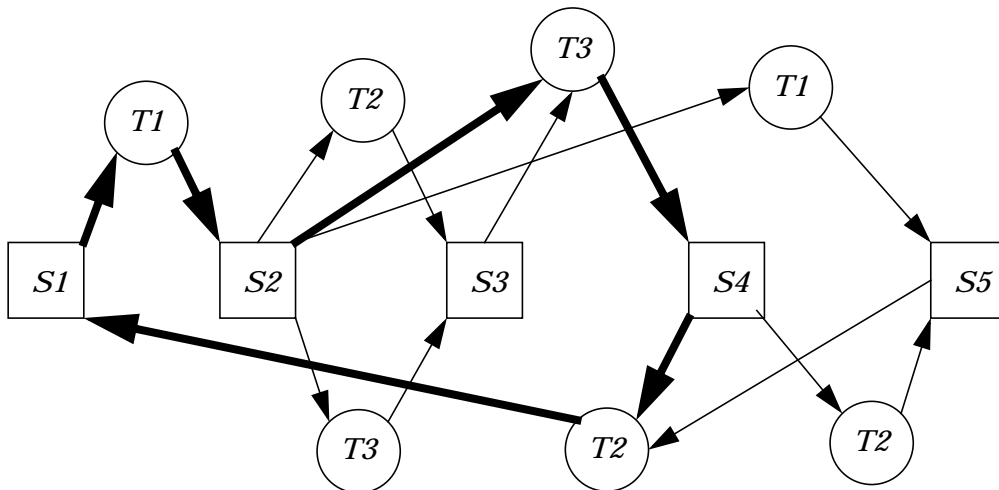
T2.run()
S4.take();
useS4();
S5.take();
S1.take();
useS1S4S5();
S1.give();
S5.give();
S4.give();
S2.take();
S3.take();
useS2S3();
S3.give();
S2.give();

T3.run()
S1.take();
useS1();
S1.give();
S2.take();
S3.take();
useS2S3();
S4.take();
useS2S3S4();
S4.give();
S3.give();
S2.give();

```

a) Rita en resursallokeringsgraf för systemet.

(2p)



b) Finns det risk för dödläge i systemet? Motivera ditt svar.

(1p)

Ja, det finns risk för dödläge! Om T1 är i begrepp att ta semafor S2 samtidigt som T2 försöker ta semafor S1 och T3 försöker ta semafor S4 så uppstår cirkulär väntan och risk för dödläge. Detta ser man i resursallokeringsgrafen genom den cirkulära struktur som markerats med breda pilar. Det är värt att notera att det även går att hitta andra cirkulära strukturer i grafen, men dessa skulle i så fall kräva att det fanns flera identiska trädar i systemet. Enligt uppgiften fanns det bara tre trädar i systemet - följaktligen en av vardera T1, T2 och T3.

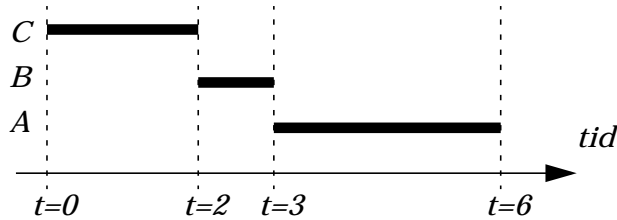
7. Betrakta ett system bestående av tre oberoende periodiskt exekverande trädar med nedanstående karaktäristika (C = värstafallsexekveringstid, D = deadline, T = period).

Tråd	C (ms)	D (ms)	T (ms)
A	3	4	10
B	1	8	8
C	2	6	6

a) Vad blir svarstiderna för de tre olika trädarna om man tillämpar Rate Monotonic Scheduling (RMS)? Är systemet schemalägningsbart, dvs klarar systemet av alla sina deadlines?

(2p)

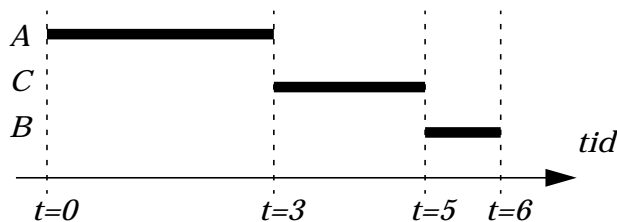
Enligt Rate Monotonic Scheduling ska vi ge trådarna prioritet efter deras perioder - ju kortare period, desto högre prioritet. Detta ger att C ska ha högst prioritet, därefter B och A ska ha lägst prioritet. Om vi ritar upp hur trådarna exekverar vid det s.k. "kritiska ögonblicket", dvs när alla trådarna vill börja köra samtidigt får vi svarstiderna för trådarna genom att se när de olika trådarna har kört färdigt första gången. Vi ritar:



Svarstiderna (i värsta fall) blir således för A: 6 ms, B: 3 ms och för C: 2 ms. Tråd A och B klarar sina tidskrav, dvs svarstiderna är mindre eller lika med deadline, men det gör inte tråd C. Systemet är alltså **inte** schemalägningsbart med RMS!

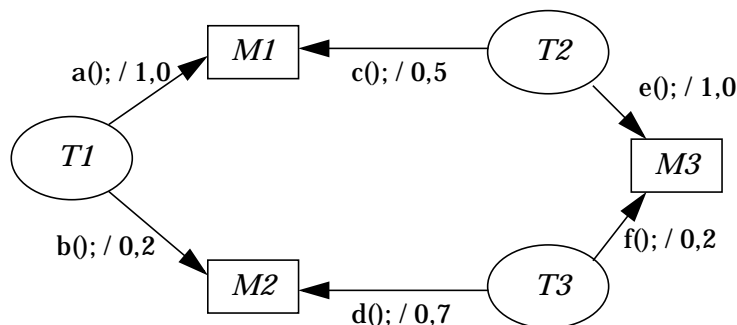
b) Vad blir svarstiderna för de tre trådarna om man i stället använder sig av Deadline Monotonic Scheduling (DMS)? Är systemet schemalägningsbart? (2p)

Vi gör samma analys som i uppgift a). Enligt Deadline Monotonic Scheduling ska man sätta prioriteterna på trådarna efter hur korta deadlines de har. Ju kortare deadline, desto högre prioritet. Vi får då att A ska ha högst prioritet, C därefter och B ska ha lägst prioritet. Vi ritar schemalägningsdiagram:



Svarstiderna blir nu för A: 3 ms, B: 6 ms och för C: 5 ms. Alla trådar klarar nu sina tidskrav! Systemet är således schemalägningsbart!

8. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning och prioritetsarv) innehåller tre trådar (T1, T2 och T3) som kommunicerar med varandra genom att anropa monitoroperationerna a, b, c, d, e och f i monitorerna M1, M2 och M3 och med maximala exekveringstider (i millisekunder) enligt figur. Trådarna anropar en monitoroperation i taget. T1 har högst prioritet och T3 har lägst prioritet.



Ange för varje tråd (T1, T2 och T3) hur lång tid tråden i värsta fall kan bli blockerad av lägre prioriterade trådar under en och samma körning. (2p)

I systemet förekommer blockering av två typer: direkt blockering (normal blocking) och "push-through-blocking".

För tråd T1, som har högst prioritet, förekommer endast direkt blockering eftersom "push-through-blocking" kräver att det finns minst en tråd som har högre prioritet (och sådan finns inte) och en som har lägre prioritet samt att dessa två kommunicerar med varandra. T1 kan blockeras först av T2 i monitorn M1 i högst 0,5 ms (tiden det kan tänkas ta för T2 att köra färdigt operationen c()). Sedan kan T1 tänkas anropa b() i monitorn M2 och då bli blockerad av T3 i högst 0,7 ms. Sammanlagd blockeringstid i värsta fall blir då $0,5+0,7=1,2$ ms.

Tråd T2 kan inte bli blockerad av T1 därför att T1 har högre prioritet. Visserligen kan T1 avbryta körningen av T2, men detta räknar vi inte som blockering av T2. Då återstår T3. T3 kan precis ha anropat operationen f() i M3 när T2 börjar köra och när T2 i sin tur anropar e() i samma monitor blir T2 blockerad i högst 0,2 ms. T3 kan dock alternativt precis ha anropat operationen d() i monitorn M2. Om nu T1 kommer in och vill börja köra och också den försöker gå in i M3 kommer prioriteten för T3 att höjas till samma prioritet som T1 har så länge T3 är inne i operationen d(). Detta kan den vara i högst 0,7 ms och i och med att prioriteten har höjts över prioriteten för T2 kommer T3 att "tränga sig före" T2 i högst 0,7 ms. Detta kallas för "push-through-blocking". Eftersom T3 enligt uppgiften inte kan vara både i monitor M2 och M3 samtidigt ska vi ta den längsta möjliga blockeringen som blockeringstid. $0,7 > 0,2$ så blockeringstiden blir **0,7 ms**.

Tråden T3 har lägst prioritet och det finns alltså inga trådar med lägre prioritet som kan blockera den. Blockeringstiden för T3 blir således trivialt lika med **0 ms**.

SVAR: $B_1=1,2$ ms, $B_2=0,7$ ms och $B_3=0$ ms.

9. Väntan på brev (förkortad uppgiftsformulering)

I klassen `RTEventBuffer`, finns det två olika operationer för att hämta meddelanden. Den första, `doFetch()`, väntar tills ett meddelande verkligen finns tillgängligt och returnerar detta. Finns inget meddelande i bufferten för stunden så blockerar den anropande tråden tills ett meddelande anländer (någon gör `doPost(...)/tryPost(...)`). Den andra operationen, `tryFetch()`, återvänder omedelbart oavsett om det fanns något meddelande i brevlådan eller inte. Fanns det ett meddelande returneras detta, annars returneras `null`.

I olika sammanhang vore det användbart med en tredje variant, nämligen `timedFetch(long timeout)`, med följande funktion: Om inget meddelande finns tillgängligt i brevlådan blockerar den anropande tråden tills dess ett sådant anländer - dock högst i det av `timeout`-parametern angivet antal millisekunder. Anländer inget meddelande till brevlådan inom denna tidsrymd returneras `null`.

På motsvarande sätt vore det praktiskt att ha en operation `timedPost(...)` som fungerar som `doPost(...)` fast väntar högst ett angivet antal millisekunder om bufferten är full (`doPost(...)` blockerar ju tills det finns plats i bufferten).

Din uppgift är att utvidga klassen `RTEventBuffer` med en operation `timedFetch(long timeout)` och en operation `timedPost(RTEvent e, long timeout)` enligt nedanstående anvisningar:

- Vi antar i denna uppgift att `wait(long timeout)` **inte** finns tillgänglig.
- Komplettera med egna operationer/attribut/extra hjälpklasser efter behov.
- Lösningen bör ej medföra att fler trådar än nödvändigt skapas.
- Operationen `timedPost(...)` ska returnera `true` eller `false` beroende på om operationen lyckades eller vi fick en `timeout`.
- Operationen `timedFetch(...)` ska returnera ett inkommande meddelande utan onödig fördröjning.

Lösningförslag

Kraven på lösningen gör att vi behöver någon mekanism som gör att en tråd som anropar `timedFetch/timedPost` kan blockera tills antingen någon annan tråd stoppar in ett meddelande respektive hämtar ett meddelande (någon anropar `notify`) eller att vi får en `timeout`. För detta vore varianten av `wait` med `timeout` (`wait(long timeout);`) idealiskt, men eftersom denna inte finns tillgänglig får vi hitta en annan lösning.

Lösningar baserade på att man inuti `timedFetch/timedPost` gör `sleep`, antingen hela timeouttiden eller i korta intervall, fungerar inte eftersom monitorn (vår `TimeoutEventBuffer`) är låst under tiden och ingen annan tråd kan gå in i monitorn för att stoppa in/hämta meddelande.

Lämpligen gör man i stället så att man anropar `wait` utan timeout samt har en extra tråd som ansvarar för att `notify` anropas när väntetiden går ut. En variant är att skapa en ny tråd varje gång en tråd behöver vänta i `timedFetch/timedPost`, men detta strider mot kravet att vi inte ska skapa fler trådar än nödvändigt. Bättre är alltså att ha en tråd som håller reda på alla beställda timeouter och anropar `notify` när det är dags för en tråd att få timeout.

Följande lösningsförslag (marginellt modifierat) lämnades in av Fredrik Olofsson, E98. Hans lösning bygger på att det finns ett attribut i bufferten som anger tiden för nästa timeout (`deadline`). Trådar som vill beställa en timeout kontrollerar denna tid och om dess egen beställning är tidigare sätter tråden attributet till sin egen deadline samt informerar en Supervisor-tråd genom att göra `interrupt` på denna. Supervisor-tråden bevakar attributet och gör `sleep` den tid som angetts varefter den går in i `TimeoutEventBuffer`-objektet och gör `notifyAll`. Om den blir avbruten via `interrupt` gör den ett nytt `sleep` angiven tid. När en tråd vaknar efter att ha gjort `wait` kontrollerar den om det är dags för timeout. Om den ska fortsätta vänta kontrollerar den återigen attributet och sätter den om nödvändigt till sin egen timeout.

```
class TimeoutEventBuffer extends RTEventBuffer {
    protected long deadline = Long.MAX_VALUE;
    protected RTThread supervisor;

    public TimeoutEventBuffer(int maxsize) {
        super(maxsize);
        supervisor = new Supervisor(this);
        supervisor.start();
    }

    public synchronized RTEvent timedFetch(long timeout) {
        long myDeadline = System.currentTimeMillis()+timeout;
        RTEvent ev = tryFetch();
        while ((System.currentTimeMillis()<myDeadline) && (ev==null)) {
            if (myDeadline<deadline) {
                deadline = myDeadline;
                supervisor.interrupt();
            }
            try { wait();
            } catch (InterruptedException e) { error("timedFetch interrupted"); }
            ev = tryFetch();
        }
        return ev;
    }

    public synchronized boolean timedPost(RTEvent e, long timeout) {
        long myDeadline = System.currentTimeMillis()+timeout;
        e = tryPost(e);
        while ((System.currentTimeMillis()<myDeadline) && (e!=null)) {
            if (myDeadline<deadline) {
                deadline = myDeadline;
                supervisor.interrupt();
            }
            try { wait();
            } catch (InterruptedException e) { error("timedFetch interrupted"); }
            e = tryPost(e);
        }
        return e==null;
    }

    public synchronized void wakeUp() {
        deadline = Long.MAX_VALUE;
        notifyAll();
    }
}
```

```

}
class Supervisor extends RTThread {

    private TimeoutEventBuffer buffer;

    public Supervisor(TimeoutEventBuffer theBuffer) {
        buffer = theBuffer;
    }

    public void run() {
        while (true) {
            try {
                int toSleep = buffer.deadline-System.currentTimeMillis();
                if (toSleep>0)
                    sleep(toSleep);
                buffer.wakeUp();
            } catch (InterruptedException e) {}
        }
    }
}

```

Alternativt lösningsförslag

En alternativ lösning till problemet är att lägga in alla beställda timeouter i en länkad lista, vilket dock blir mer komplicerat att hantera jämfört med Fredriks lösning:

```

class TimeoutEventBuffer extends REventBuffer {
    Timeouter myTimeouter;

    public TimeoutEventBuffer(int maxsize) {
        super(maxsize);
        myTimeouter = new Timeouter(this);
        myTimeouter.start();
    }

    public synchronized REvent timedFetch(long timeout) {
        long stoptime = System.currentTimeMillis()+timeout;
        REvent e = tryFetch();
        if (e==null) {
            myTimeouter.requestNotification(stoptime);
            while (e==null && System.currentTimeMillis(<stoptime) {
                try { wait();
                } catch (InterruptedException e) {
                    error("timedFetch interrupted"); }
                e = tryFetch();
            }
        }
    }

    public synchronized boolean timedPost(REvent e, long timeout) {
        long stoptime = System.currentTimeMillis()+timeout;
        e = tryPost(e);
        if (e!=null) {
            myTimeouter.requestNotification(stoptime);
            while (e!=null && System.currentTimeMillis(<stoptime) {
                try { wait();
                } catch (InterruptedException e) {
                    error("timedFetch interrupted"); }
                e = tryPost(e);
            }
        }
        return e==null;
    }
}

```

```

    public synchronized void doNotify() {
        notifyAll();
    }
}

class Timeouter extends Thread {
    TimeoutEventBuffer myBuffer;
    TimeoutNotice orders;

    public Timeouter(TimeoutEventBuffer buf) {
        myBuffer = buf;
    }

    public void run() {
        while (true) {
            long tosleep = calculateSleepTime();
            try {
                if (tosleep>0)
                    sleep(tosleep);
            } catch (InterruptedException e) { }
            if (performNotify())
                myBuffer.doNotify();
        }
    }

    private synchronized boolean performNotify() {
        boolean n = false;
        long now = System.currentTimeMillis();
        while (orders!=null && orders.when<=now) {
            n = true;
            orders = orders.next;
        }
        return n;
    }

    private synchronized long calculateSleepTime() {
        while (orders==null) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        return orders.when-System.currentTimeMillis();
    }

    public synchronized void requestNotification(long notificationTime) {
        TimeoutNotice o = new TimeoutNotice();
        o.when = notificationTime;
        if (orders==null) {
            o.next = null;
            orders = o;
            notifyAll();
        } else {
            if (orders.when>notificationTime) {
                o.next = orders;
                orders = o;
                interrupt();
            } else {
                TimeoutNotice c = orders;
                TimeoutNotice n = orders.next;
            }
        }
    }
}

```

```

        while (n!=null && n.when<notificationTime) {
            c = n;
            n = n.next;
        }
        o.next = c.next;
        c.next = o;
    }
}

class TimeoutNotice {
    TimeoutNotice next;
    long when;
}

```

10. Färddator (förkortad uppgiftsformulering)

Uppgiften går ut på att skriva styrprogrammet till en färddator i en bil. Färddatorn kan presentera en av följande uppgifter åt gången på en liten display:

- Körd sträcka i kilometer (trippmätare).
- Aktuell bensinförbrukning i liter/100km (medelvärde över de senaste tio sekunderna).
- Medelförbrukning (liter/100km) sedan början på trippen.
- Mängd bensin som finns kvar i tanken (liter).
- Uppskattad kvarvarande körsträcka utan tankning baserad på medelförbrukningen (kilometer).

Displayen kan visa ett decimaltal samt en ikon som indikerar vilket presentationsläge datorn befinner sig i. Vidare finns två knappar som föraren kan trycka på: *Reset* och *Mode*. Vid tryckning på *Reset* nollställs datorns trippmätare samt medelförbrukning av bensin. När man trycker på *Mode* växlar datorn mellan att visa körsträcka, aktuell bensinförbrukning, medelförbrukning, bensinmängd och uppskattad kvarvarande körsträcka.

Lösningsförslag

Om vi betraktar parallellismen i problemet finner vi följande parallella aktiviteter:

1. Bevaka knapptryckningar (via blockerande anrop av `CarIO.waitForButtonPress()`).
2. Uppdatera körd sträcka (vänta på "tick" från hjulet via blockerande anrop i `CarIO`).
3. Läsa av aktuell bränsleförbrukning 10 gånger i sekunden och medelvärdesbilda/beräkna ackumulerad förbrukning.
4. Beräkna och visa ett uppdaterat värde på displayen en gång i sekunden.

Det verkar alltså som om fyra trådar behövs. Vi noterar dock att de två periodiska aktiviteterna (3 och 4) har harmoniska perioder. Det bör alltså vara möjligt att slå ihop dessa två aktiviteter till en enda tråd med periodtiden 100 ms (10 Hz) som uppdaterar displayen var tionde gång den exekverar. Det är inte helt uppenbart om man ska slå ihop de två aktiviteterna på detta sätt eller inte, men det känns som om det bör bli enklare så vi väljer att slå ihop dem i vårt lösningsförslag. Förutom att vi får en tråd mindre slipper vi fundera på hur de två trådarna annars skulle kommunicera med varandra. Vi får alltså tre trådar:

RevThread - Räknar upp körd sträcka varje gång hjulet har snurrat ett kvarts varv. Eftersom vi får fyra signaler varje rullat varv ger detta en exekveringsfrekvens på ca 120 Hz vid 200 km/h ($200/3,6/(1,85/4)$). Eftersom vi bara har möjlighet att vänta på nästa "tick" från hjulet måste vi se till att tråden hinner utföra sin uppgift och anropa `CarIO.waitForRevolution()` igen innan nästa "tick" anländer.

ButtonThread - Väntar på att föraren trycker på en knapp. Eftersom knapptryckningar förekommer ganska sällan och det inte gör så mycket om datorn inte omedelbart svarar på tryckningen (fördröjningar på säg mindre än 0,2 sekunder är knappast ens märkbara) behöver inte denna aktivitet högprioriteras.

UpdateThread - Kör tio gånger per sekund och läser av aktuell bensinförbrukning. Var tionde exekvering uppdateras dessutom värdet på displayen (en gång per sekund).

Exekveringsfrekvenserna för trådarna samt deras känslighet för missade deadlines gör att **RevThread** bör ha högst prioritet. **UpdateThread** bör ha näst högst prioritet och **ButtonThread** lägst prioritet.

RevThread och **ButtonThread** behöver kunna meddela **UpdateThread** när externa insignaler anländer. Vi väljer att använda en monitor som skyddar gemensamma data som håller reda på körd sträcka samt aktuell visningsmode.

```
import se.lth.cs.realtime.*;

class Salvo {
    public static void main(String args[]) {
        Monitor m = new Monitor();
        new RevThread(m).start();
        new ButtonThread(m).start();
        new UpdateThread(m).start();
    }
}

class Monitor {
    long revs;
    int mode;
    boolean resetP;

    synchronized void reset() {
        revs = 0;
        resetP = true;
        CarIO.displayValue(-1.0,mode);
    }

    synchronized long getRevs() {
        long r = resetP ? -1 : revs;
        resetP = false;
        return r;
    }

    synchronized void incRevs() {
        revs++;
    }

    synchronized void newMode() {
        mode = (mode+1)%5;
        CarIO.displayValue(-1.0,mode);
    }

    synchronized int getMode() {
        return mode;
    }
}

class RevThread extends Thread {
    Monitor mon;

    RevThread(Monitor m) {
        mon = m;
        setPriority(MAX_PRIORITY);
    }
}
```

```

public void run() {
    while (true) {
        CarIO.waitForRevolution();
        mon.incRevs();
    }
}

class ButtonThread extends Thread {
    Monitor mon;

    ButtonThread(Monitor m) {
        mon = m;
        setPriority(MIN_PRIORITY);
    }

    public void run() {
        while (true) {
            int b = CarIO.waitForButtonPress();
            if (b==0)
                mon.reset();
            else
                mon.newMode();
        }
    }
}

class UpdateThread extends PeriodicThread {
    Monitor mon;
    long revs[];
    double cons[];
    double acc,tot;
    int p;

    UpdateThread(Monitor m) {
        super(100);
        setPriority(NORM_PRIORITY);
        mon = m;
        revs = new long[100];
        cons = new double[100];
    }

    public void perform() {
        long deltaR;
        long r = mon.getRevs();
        if (r== -1) {
            for(int t=0;t<100;t++) {
                revs[t] = 0;
                cons[t] = 0.0;
                acc = 0.0;
                tot = 0.0;
            }
        } else {
            p=(p+1)%100;
            deltaR = r-revs[p];
            acc -= cons[p];
            revs[p] = r;
            cons[p] = CarIO.currentConsumption()/10.0;
            acc += cons[p];
            tot += cons[p];
        }
    }
}

```

