Lösning till Hemtentamen, EDAF85 Realtidssystem

2020–10–30, 08.00–13.00

1. Dödläge

Dödläge kan uppstå när två eller flera trådar väntar på varandra på ett sådant sätt att ingen av trådarna kan komma vidare i sin exekvering. Normalt uppstår det om trådarna har låst var sin resurs som de inte släpper förrän de fått tillgång till en annan resurs som är låst av någon av de andra i dödläget inblandade trådarna och de trådarna i sin tur väntar på att få tillgång till den första resursen.

Man brukar säga att det finns fyra villkor för att dödläge ska kunna uppstå:

Ömsesidig uteslutning – Enbart en tråd kan komma åt en resurs åt gången. Detta är nödvändigt för att garantera att trådarna inte stör varandra. Svårt att göra något åt.

Hold and wait – En tråd kan låsa en resurs och sedan vänta på att låsa en annan resurs. För att undvika detta skulle man inte få använda mer än en resurs samtidigt, vilket skulle vara begränsande för vad man kan göra effektivt.

Ingen resource preemption" – Vi kan inte ta ifrån en tråd en resurs den har låst innan tråden är klar med den för det skulle kunna göra att vi får data som hamnar i ett tillstånd som inte är konsistent. Inte mycket vi kan (vill) göra något åt heller.

Cirkulär väntan – De olika möjliga hold-wait-situationerna i programmet är sådana att det kan skapas en cirkulär kedja av trådar som väntar på resurser som är låsta av andra trådar i kedjan. Genom att vara noggranna med i viken ordning vi tar semaforerna kan vi undvika att det skapas sådana cirkulära kedjor av väntan. Så detta är det villkoret som det är enklast att se till att det inte är uppfyllt och därigenom garantera att dödläge inte uppstår i programmet.

2. Semaforer

För att åstadkomma ömsesidig uteslutning så behöver vi en semafor med startvärdet 1 (ett körtillstånd). När en tråd sedan vill gå in i en kritisk region så tar tråden semaforen med ett `acquire()`. När de lämnar regionen återställer samma tråd semaforens körtillstånd med anrop av `release()`. Det är alltså samma tråd som gör `acquire()`. som sedan gör `release()`. Vid signalering vill vi istället från en tråd meddela en annan tråd att något har inträffat. Därför har vi här en semafor med startvärdet 0 som den mottagende tråden väntar på (`acquire()`). När tråden som vill skicka signalen gör det anropar den `release()`, varvid den mottagande tråden kommer loss från anropet av `acquire()`.

Ömsesidig uteslutning

Tråd 1	Tråd 2
s.acquire();	
//kritisk region	s.acquire()
//kritisk region	// blockerad
s.release();	// kommer igång
...	// kritisk region
...	s.release();

Signalering

Tråd 1	Tråd 2
s.acquire();	
// blockerad	...
...	s.release();
// signal mottagen	

3. Generaliserad schemaläggningsanalys

Denna uppgift gavs i fyra olika varianter som skiljde sig åt i ett par detaljer. Dels varierade exekveringstiden för operationerna `v()` och `x()` något och dels hade trådarna fått olika bokstavsbeteckningar. Här följer lösningen till den variant som hade följande tabell över trådarnas egenskaper och där maxtiden för `v()` var 0,1 ms och för `x()` 0,2 ms. Övriga varianter kräver exakt samma beräkningar och svaret kommer bara att skilja sig när det gäller trådarnas bokstavsbeteckning samt de två mest högprioriterade trådarnas responstider som kommer att variera med 0,1 ms.

Tråd	*C (ms)*	*T (ms)*	*D (ms)*
A	3	15	10
B	1	5	2
C	4	20	15
D	2	8	4

a) Med hjälp av formeln kan vi beräkna en tråds maximala responstid, R_i. Responstiden beror dels på trådens egen maximala exekveringstid, dels på hur lång tid högre prioriterade trådar kan fördröja tråden och slutligen på den tid trådar som normalt sett har lägre prioritet kan fördröja oss. Detta avspeglas i formelns tre termer i högerledet. C_i är den aktuella trådens egen exekveringstid och B_i är den maximala tiden som trådar med lägre prioritet kan fördröja tråden. För att kunna ge en rimlig övre gräns för hur stor den senare effekten kan bli behövs någon typ av prioritetsarvsprotokoll. Då kan vi räkna ut denna övre tid, vilken benämns B_i. Då återstår effekten av de högre prioriterade trådarna att te med i beräkningen och detta kommer in i form av den tredje termen som är en summa av de enskilda högre prioriterade trådarnas effekt på trådens responstid. Tiden en enskild högre prioriterad tråd kan fördröja den aktuella tråden ges i sin tur av antalet gånger den kommer att köra innan den aktuella tråden har slutfört sitt arbete multiplicerat med de högre trådens värstafallsxekveringstid.

   Eftersom R_i förekommer inuti takfunktionen (avrundning uppåt till närmsta högre heltal) går det inte att lösa ut R_i ur formeln. Man får därför ta till en numerisk lösning, nämligen den så kallade iterationsmetoden. Man börjar med att gissa ett lågt värde på R_i, sätter in det i formeln och får ett nytt värde på R_i som man på nytt sätter in ända tills värdet inte ändrar sig längre. Då har vi hittat den korrekta svarstiden.

b) Först måste vi avgöra prioritetsordningen bland trådarna. Eftersom RMS skulle användas ges det av kolumn T i tabellen. Vi får då (prioritet efter stigande periodtid): B, D, A, C. Vi kan därefter resonera oss fram till blockeringstiderna genom att analysera den givna blockeringsgrafen.

För B (högst prioriterad) gäller att bara direkt blockering kan vara aktuell. Eftersom både M1 och M2 skulle kunna vara blockerade när B vill köra måste vi addera ihop de enskilda värstafallsblockeringstiderna (givna av metoderna `v()` och `z()`. Vi får då $B_B = 0,1 + 0,4 = 0,5ms$.

För D är ingen direkt blockering möjlig eftersom den inte använder någon av monitorerna. Däremot kan den råka ut för indirekt blockering om antingen C eller A låser sina respektive monitorer när A vill köra. Då kommer de tillfälligt att ärva Bs prioritet och tränga sig före D. Detta kan återigen hända i båda monitorerna och vi får då åter en blockeringstid som är summan av de två möjliga indirekta blockeringarna: $B_D = 0,1 + 0,4 = 0,5ms$.

För A gäller också att enbart indirekt blockering kan komma ifråga. Det är så på grund av att den inte kommunicerar direkt med en lägra prioriterad tråd via någon monitor. Indirekt blockering kan dock fortfarande förekomma om C ärver Bs prioritet (i max 0,4 ms). Vi får alltså: $B_A = 0,4ms$.

Till slut har vi den lägst prioriterade tråden kvar, C. Eftersom det inte finns några trådar med lägre prioritet kan den heller inte blockeras av lägre prioriterade trådar. Vi får alltså: $B_C = 0ms$.

c) Vi använder iterationsmetoden som beskrivet ovan:

$R_B = 1 + 0,5 = 1,5$ Eftersom vi inte har något R_B i högerledet finns det ingen anledning att fortsätta iterationen.

$$R_D = 2 + 0,5 = 2,5$$
$$R_D = 2 + 0,5 + \left\lceil \frac{2,5}{5} \right\rceil \cdot 1 = 3,5$$
$$R_D = 2 + 0,5 + \left\lceil \frac{3,5}{5} \right\rceil \cdot 1 = 3,5$$

$$R_A = 3 + 0,4 = 3,4$$
$$R_A = 3 + 0,4 + \left\lceil \frac{3,4}{5} \right\rceil \cdot 1 + \left\lceil \frac{3,4}{8} \right\rceil \cdot 2 = 6,4$$
$$R_A = 3 + 0,4 + \left\lceil \frac{6,4}{5} \right\rceil \cdot 1 + \left\lceil \frac{6,4}{8} \right\rceil \cdot 2 = 7,4$$
$$R_A = 3 + 0,4 + \left\lceil \frac{7,4}{5} \right\rceil \cdot 1 + \left\lceil \frac{7,4}{8} \right\rceil \cdot 2 = 7,4$$

$$R_C = 4 + 0 = 4$$
$$R_C = 4 + 0 + \left\lceil \frac{4}{5} \right\rceil \cdot 1 + \left\lceil \frac{4}{8} \right\rceil \cdot 2 + \left\lceil \frac{4}{15} \right\rceil \cdot 3 = 10$$
$$R_C = 4 + 0 + \left\lceil \frac{10}{5} \right\rceil \cdot 1 + \left\lceil \frac{10}{8} \right\rceil \cdot 2 + \left\lceil \frac{10}{15} \right\rceil \cdot 3 = 13$$
$$R_C = 4 + 0 + \left\lceil \frac{13}{5} \right\rceil \cdot 1 + \left\lceil \frac{13}{8} \right\rceil \cdot 2 + \left\lceil \frac{13}{15} \right\rceil \cdot 3 = 14$$
$$R_C = 4 + 0 + \left\lceil \frac{13}{5} \right\rceil \cdot 1 + \left\lceil \frac{13}{8} \right\rceil \cdot 2 + \left\lceil \frac{13}{15} \right\rceil \cdot 3 = 14$$

Svarstidena blir alltså: $R_B = 1,5ms$, $R_D = 3,5ms$, $R_A = 7,4ms$ och $R_C = 14ms$.

4. Filtrering av sensorvärden

```
abstract class DistanceSensing {

    private static final double DANGER_DIST = 0.8;

    /* Add your own attributes here */
    Sample val[] = new Sample[3];
    boolean safe = true;
    boolean restore = false;

    /** Constructor */
    public DistanceSensing() {
        /* to be implemented */
        new DangerThread().start();
    }

    /** Stores a new sensor value in the monitor */
    public synchronized void put(Sample s) {
        /* to be implemented */
        val[s.getId()] = s;
        if (safe) {
            safe = isSafe();
        }
        if (restore) {
            restore = !isSafe();
        }
        notifyAll();
    }

    /** Blocks until sensor values at least as new as time
        are available from all sensors anf then returns
        the sample with smallest value.
    */
    public synchronized Sample getMinimum(long time) {
        /* to be implemented */
        Sample ret = null;
        do {
            ret = minimum(time);
            try {
                wait();
            } catch(InterruptedException e) {
                throw new Error("Unexpected InteruptedException!");
            }
        } while(ret==null);
        return ret;
    }

    /** Automatically called whenever it is detected that
        any of the current sensor values are lower than
        DANGER_DIST. Only one invokation of this method can
        be active at any time, i.e., if the method is called
        it can not be called again until the first call has
        finished. A new call also requires that all sensor
        values has returned to a safe state (> DANGER_DIST)
        before it can be called again. A call to the method
        is allowed to take a long time, e.g., waiting for an
        operator to acknowledge an alarm.
    */
    protected abstract void dangerAlert();
```

```
    /* Add your private methods/inner classes here if your */
    /* solution requires any.                              */

    private Sample minimum(long time) {
        Sample ret = null;
        int valid = 0;
        for(int i=0;i<3;i++) {
            if (val[i]!=null) {
                if (val[i].getTimestamp()>=time) {
                    valid++;
                    if (ret==null || val[i].getDistance()<ret.getDistance()) {
                        ret = val[i];
                    }
                }
            }
        }
        if (valid<3) {
            ret = null;
        }
        return ret;
    }

    private boolean isSafe() {
        boolean safe = true;
        for(int i=0;i<3;i++) {
            if (val[i]!=null && val[i].getDistance()<DANGER_DIST) {
                safe = false;
                break;
            }
        }
        return safe;
    }

    private synchronized void awaitUnsafe() {
        while (safe) {
            try {
                wait();
            } catch(InterruptedException e) {
                throw new Error("Unexpected InteruptedException!");
            }
        }
    }

    private synchronized void awaitSafe() {
        if (!isSafe()) {
            restore = true;
            while (restore) {
                try {
                    wait();
                } catch(InterruptedException e) {
                    throw new Error("Unexpected InteruptedException!");
                }
            }
        }
        safe = true;
    }

    private class DangerThread extends Thread {
        public void run() {
            while (true) {
                awaitUnsafe();
                dangerAlert();
                awaitSafe();
            }
        }
    }

}
```