Lösningar, EDAF85 Realtidssystem

2024–08–29, 14.00–19.00

1. 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

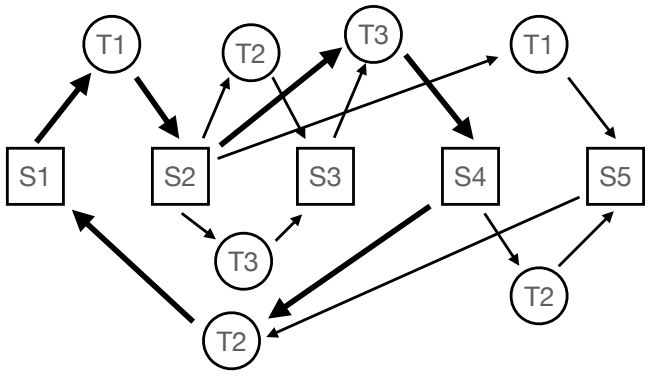
Tråd 1	Tråd 2
s.acquire();	
//kritisk region	s.acquire()
//kritisk region	// blockerad
s.release();	// kommer igång
...	// kritisk region
...	s.release();

   Signalering

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

2.

   a)



   b) Om det finns en cykel i grafen som innefattar högst det antal trådinstanser (av respektive förekommande typ) som systemet innehåller kan det bli dödläge. Det förutsätter också att de i cykeln inblandade trådarna kan befinna sig i de av cykeln antydda exekveringstillstånden samtidigt.

   c) Det finns bara en cykel (och därmed scenario) som uppfyller alla kraven ovan.

   d) T1: rad 11, T2: rad 4, T3: rad 7

3.

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. Summan tar hänsyn till högre prioriterade trådars effekt.

För att kunna ge en rimlig övre gräns för tiden som trådar med lägre prioritet kan fördröja tråden behövs någon typ av prioritetsarvsprotokoll. Då kan vi räkna ut denna övre tid, vilken benämns B_i.

Lägre prioriterade trådar kan fördröja en högre prioriterad tråd på två sätt. Om en tråd försöker ta ett lås (semafor/monitor) som är låst av en lägre prioriterad tråd måste tråden vänta tills låset släpps. Detta kallas för direkt blockering. Prioritetsarvsprotokollet ser till att denna tid blir begränsad. Prioritetsarvsprotokoll har dock också den effekten att trådar som normalt sett har lägre prioritet än den aktuella tråden tillfälligt kan ärva en högre prioritet och därmed tränga sig före den aktuella tråden. Detta kallas för indirekt blockering eller *push-through blocking*.

b) Vi kan inte dra några slutsatser alls i detta specifika fallet eftersom CPU-utnyttjandet bara kan användas för schemaläggningsbarhetstest ifall systemet har oberoende trådar (ingen blockering) och deadline är lika med periodtid för alla trådarna. Det är inte fallet här.

c) Först måste vi avgöra prioritetsordningen bland trådarna. Eftersom RMS skulle användas ges det av kolumn D i tabellen. Vi får då (prioritet efter stigande periodtid): B, C, D, A. 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. B_B ges då av exekveringstiden för `v()`: $B_B = 0{,}4ms$.

Tråd C kan råka ut för både direkt (via M2) och indirekt blockering (tråd A ärver prioriteten av tråd B): $B_C = 0{,}4 + 0{,}3 = 0{,}7ms$.

Tråd D kan bara bli indirekt blockerad (tråd A ärver prioriteten av tråd B): $B_C = 0{,}4ms$

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

d) Vi använder iterationsmetoden:

$R_B = 1 + 0{,}4 = 1{,}4$

$R_C = 3 + 0{,}7 = 3{,}7$
$R_C = 3 + 0{,}7 + \left\lceil \frac{3{,}7}{4} \right\rceil \cdot 1 = 4{,}7$
$R_C = 3 + 0{,}7 + \left\lceil \frac{4{,}7}{4} \right\rceil \cdot 1 = 5{,}7$
$R_C = 3 + 0{,}7 + \left\lceil \frac{5{,}7}{4} \right\rceil \cdot 1 = 5{,}7$

$R_D = 2 + 0{,}4 = 2{,}4$
$R_D = 2 + 0{,}4 + \left\lceil \frac{2{,}4}{4} \right\rceil \cdot 1 + \left\lceil \frac{2{,}4}{12} \right\rceil \cdot 3 = 6{,}4$
$R_D = 2 + 0{,}4 + \left\lceil \frac{6{,}4}{4} \right\rceil \cdot 1 + \left\lceil \frac{6{,}4}{12} \right\rceil \cdot 3 = 7{,}4$
$R_D = 2 + 0{,}4 + \left\lceil \frac{7{,}4}{4} \right\rceil \cdot 1 + \left\lceil \frac{7{,}4}{12} \right\rceil \cdot 3 = 7{,}4$

$R_A = 4 + 0 = 4$
$R_A = 4 + 0 + \left\lceil \frac{4}{4} \right\rceil \cdot 1 + \left\lceil \frac{4}{12} \right\rceil \cdot 3 + \left\lceil \frac{4}{24} \right\rceil \cdot 2 = 10$
$R_A = 4 + 0 + \left\lceil \frac{10}{4} \right\rceil \cdot 1 + \left\lceil \frac{10}{12} \right\rceil \cdot 3 + \left\lceil \frac{10}{24} \right\rceil \cdot 2 = 12$
$R_A = 4 + 0 + \left\lceil \frac{12}{4} \right\rceil \cdot 1 + \left\lceil \frac{12}{12} \right\rceil \cdot 3 + \left\lceil \frac{12}{24} \right\rceil \cdot 2 = 12$

Svar: $R_A = 12$, $R_B = 1,4$, $R_C = 5{,}7$ och $R_D = 7{,}4$.

e) Eftersom varje tråds responstid är mindre än dess deadline är systemet schemaläggningsbart.

4.

```
class DoorMonitor {
  private boolean open, closed, button;

  public synchronized void reportOpen() {
    open = true;
    notify();
  }

  public synchronized void reportClosed() {
    closed = true;
    notify();
  }

  public synchronized void reportButton() {
    button = true;
    notify();
  }

  public synchronized void handleOpening(DoorIO io) {
    try {
      while (!button) {
        wait();
      }
      button = false;
      io.startOpening();
      open = false;
      while (!open) {
        wait();
      }
      io.stop();
    } catch(InterruptedException e ) {
      System.out.println("Unexpected InterruptedException in DoorMonitor!");
    }
  }

  public synchronized void handleClosing(DoorIO io) {
    try {
      io. startClosing();
      closed = false;
      while (!closed && !button) {
        wait();
      }
      io.stop();
    } catch(InterruptedException e ) {
      System.out.println("Unexpected InterruptedException in DoorMonitor!");
    }
  }
}
```

5. `DoorHandlerThread` har ganska enkel logik som visserligen ganska lätt kan implementeras med nästlade `while`-satser enligt följande:

```
class DoorHandlerThread extends ActorThread<String> {
  private DoorIO io;

  public DoorHandlerThread(DoorIO io) {
    this.io = io;
  }

  public void run() {
    while (true) {
      try {
        String s;
        do {
          s = receive();
        } while (!s.equals("B"));
        while (!s.equals("C")) {
          io.startOpening();
          do {
            s = receive();
          } while (!s.equals("O"));
          io.stop();
          Thread.sleep(5000);
          io.startClosing();
          do {
            s = receive();
          } while (!s.equals("C") && !s.equals("B"));
          io.stop();
        }
      } catch(InterruptedException e) {
        System.out.println("Unexpected InterruptedException!");
      }
    }
  }
}
```

När logiken blir mer komplicerad är det ofta enklare att modellera trådar som en tillståndsmaskiner. Vi rekommenderar generellt denna lösning för trådar som tar emot meddelanden. I vårt fall får vi då i stället:

```
class DoorHandlerThread extends ActorThread<String> {
  private DoorIO io;

  public DoorHandlerThread(DoorIO io) {
    this.io = io;
  }

  public void run() {
    enum State { CLOSED, OPENING, CLOSING }

    State state = State.CLOSED;
    while (true) {
      try {
        String message = receive();
        switch(state) {
          case State.CLOSED:
            if (message.equals("B")) {
              io.startOpening();
              state = State.OPENING;
            }
          break;
```

```
            case State.OPENING:
              if (message.equals("O")) {
                io.stop();
                Thread.sleep(5000);
                io.startClosing();
                state = State.CLOSING;
              }
              break;
            case State.CLOSING:
              if (message.equals("B")) {
                io.stop();
                io.startOpening();
                state = State.OPENING;
              } else if (message.equals("C")) {
                io.stop();
                state = State.CLOSED;
              }
              break;
          }
        } catch(InterruptedException e) {
          System.out.println("Unexpected InterruptedException!");
        }
      }
    }
  }
```

Trådar för att hantera insignalerna:

```
  class ButtonThread extends Thread {
    private DoorIO io;
    private DoorHandlerThread dht;

    public ButtonThread(DoorIO io,DoorHandlerThread dht) {
      this.io = io;
      this.dht = dht;
    }

    public void run() {
      while (true) {
        io.awaitButton();
        try {
          dht.send("B");
        } catch(InterruptedException e) {
          System.out.println("Unexpected InterruptedException!");
        }
      }
    }
  }
```

```
class OpenThread extends Thread {
  private DoorIO io;
  private DoorHandlerThread dht;

  public OpenThread(DoorIO io,DoorHandlerThread dht) {
    this.io = io;
    this.dht = dht;
  }

  public void run() {
    while (true) {
      io.awaitOpen();
      try {
        dht.send("O");
      } catch(InterruptedException e) {
        System.out.println("Unexpected InterruptedException!");
      }
    }
  }
}

class CloseThread extends Thread {
  private DoorIO io;
  private DoorHandlerThread dht;

  public CloseThread(DoorIO io,DoorHandlerThread dht) {
    this.io = io;
    this.dht = dht;
  }

  public void run() {
    while (true) {
      io.awaitClosed();
      try {
        dht.send("C");
      } catch(InterruptedException e) {
        System.out.println("Unexpected InterruptedException!");
      }
    }
  }
}
```