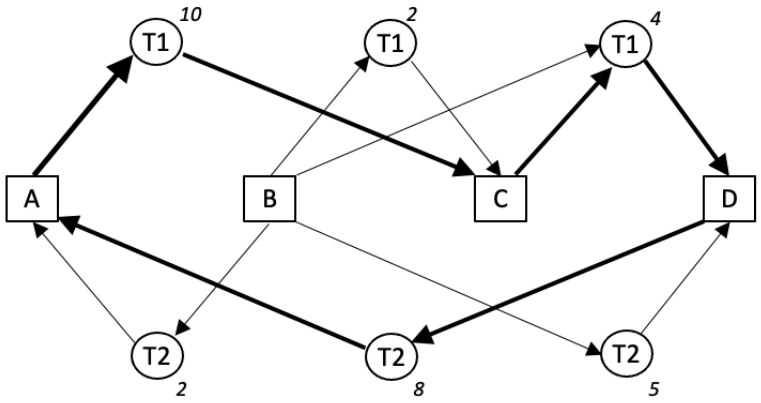
Lösningsförslag till Tentamen, EDAF85 Realtidssystem

2021–10–30, 08.00–13.00

1. a) Busy-wait. Nackdelen är att all tillgänglig CPU-tid går åt till att kontrollera om villkoret har ändrats, vilket kan göra att andra trådar fördröjs eller inte får köra alls. I värsta fall får tråden som ska uppfylla villkoret aldrig tillfälle att göra detta.

b) Nej, koden garanterar inte ömsesidig uteslutning. Som motexempel kan följande scenario användas: Två trådar försöker exekvera den aktuella koden. Den första tråden kommer fram till while-satsen och konstaterar att resursen är ledig och går vidare till nästa rad i programmet, men innnan denna rad hinner utföras sker ett kontextbyte så att den andra tråden får börja köra. Den andra tråden kommer fram till whilesatsen och konstaterar även den att resursen är ledig. Den fortsätter därför till nästa rad, markerar resursen som upptagen och går därefter in i den kritiska regionen. Om ett kontextbyte sker medan den gör det så att den första tråden återfår kontrollen kommer även denna att fortsätta in i den kritiska regionen. Vi har alltså ej ömsesidig uteslutning.

2. a) Resursallokeringsgraf med cykel markerad samt radnummer för varje hold-wait-situation.



b) Vi måste ha två instanser av T1, som exekverar rad 4 respektive 10, samt en instans av T2 som exekverar rad 8 för att det ska bli dödläge.

c) Byt ordning på rad 9 och 10 i T1 så att C låses före A. Då vänder vi på T1-beroendet så att går från C till A i stället för tvärtom och cykeln bryts.

3. a) Dynamisk schemaläggning betyder att det finns en schemaläggare som kontinuerligt under programmets körning bestämmer vilken tråd som ska få köra just nu. Det finns inget fast schema. För att avgöra vilken tråd som är den som ska få köra tittar schemaläggaren på trådarnas prioriteter och väljer vid varje tillfälle den tråd som har högst prioritet bland de körbara trådarna.

b) Dynamisk schemaläggning där det alltid är den tråd som har närmast till sin nästa deadline som får köra. Man har alltså inga fixa prioriteter.

c) Dynamisk prioritetsbaserad schemaläggning där prioriteterna sätts så att den tråd som kör oftast får högst prioritet. Sedan sjunker prioriteten med ökande periodtid.

d) Vid statisk schemaläggning bestämmar man redan på designstadiet exakt i vilken ordning de olika trådarna ska köra. Man gör ett schema som sedan upprepas cyklist vid körning.

4. DMS ger prioritetsordningen B, C, A för trådarna.

a) Blockeringstiden för A, B_A, är 0 ms eftersom A har lägst prioritet och det därmed inte finns några trådar med lägre prioritet som kan blockera den.

För B påverkas blockeringstiden B_B endast av direktblockeringar då B har högst prioritet. Det finns två fall då A kan läsa antingen M1 eller M2, men inte båda samtidigt. Om A låser M2 kan C inte låsa denna. Vi får då: $B_B = max((0,1+0,4),(0,2)) = 0,5\ ms$.

I fallet med C kan potentiellt både direkt och indirekt (push-through) blockering förekomma. Vi får två fall beroende på om A låser monitorn M1 eller M2 (indirekt respektive direkt blockering) vilket ger: $B_C = max((0,4),(0,2)) = 0,4\ ms$.

b) Vi använder iterationsmetoden för att beräkna värstafallssvarstiderna:

$R_B = 2 + 0,5 = 2,5$ Eftersom vi inte har något R_B i högerledet finns det ingen anledning att fortsätta iterationen.

$$R_C = 4 + 0,4 = 4,4$$
$$R_C = 4 + 0,4 + \left\lceil \frac{4,4}{5} \right\rceil \cdot 2 = 6,4$$
$$R_C = 4 + 0,4 + \left\lceil \frac{6,4}{5} \right\rceil \cdot 2 = 8,4$$
$$R_C = 4 + 0,4 + \left\lceil \frac{8,4}{5} \right\rceil \cdot 2 = 8,4$$

$$R_A = 3 + 0 = 3$$
$$R_A = 3 + 0 + \left\lceil \frac{3}{5} \right\rceil \cdot 2 + \left\lceil \frac{3}{20} \right\rceil \cdot 4 = 9$$
$$R_A = 3 + 0 + \left\lceil \frac{9}{5} \right\rceil \cdot 2 + \left\lceil \frac{9}{20} \right\rceil \cdot 4 = 11$$
$$R_A = 3 + 0 + \left\lceil \frac{11}{5} \right\rceil \cdot 2 + \left\lceil \frac{11}{20} \right\rceil \cdot 4 = 13$$
$$R_A = 3 + 0 + \left\lceil \frac{16}{5} \right\rceil \cdot 2 + \left\lceil \frac{10}{20} \right\rceil \cdot 4 = 13$$

Svarstiderna blir alltså: $R_B = 2,5\ ms$, $R_C = 8,4\ ms$ och $R_A = 13\ ms$.

5.

```
class AudioController extends ActorThread<AudioRequest> {

    @Override
    public void run() {
        try {
            Buzzer buzzer = new Buzzer();

            int halfPeriod = 0;   // initially silent

            // time when last period was planned to start
            long periodStart = -1;

            while (true) {
                // use receive() or receiveWithTimeout(delay),
                // depending on state
                // -----------------------------------------
                AudioRequest message;
                if (halfPeriod <= 0) {
                    message = receive();
                } else {
                    // handle clock drift
                    // (including state 'periodStart')
                    // -------------------------------
                    periodStart += 2 * halfPeriod;
                    long delay = Math.max(0, periodStart - System.nanoTime());
                    message = receiveWithTimeout(delay);
                }

                // update state if a message was received
                if (message != null) {
                    halfPeriod = message.getHalfPeriod();
                    periodStart = System.nanoTime();
                }

                // play one period, then go back to
                // receiveWithTimeout
                if (halfPeriod > 0) {
                    buzzer.on();
                    TimeUnit.NANOSECONDS.sleep(halfPeriod);
                    buzzer.off();
                }
            }
        } catch (InterruptedException unexpected) {
            throw new Error(unexpected);
        }
    }
}
```

6.
```
// Attempt to buy the requested units in o for the given maximum price or less
// before System.currentTimeMillis() reach validUntil.
// If the total order cannot be satisfied before then the order is cancelled.
// Returns the total amount paid or 0.0 if order cancelled.
public synchronized double buy(BuyOrder o,long validUntil)  {
    try {
        o.pricePaid = 0.0;
        buyOrders.insert(o);
        notifyAll();
        while (o.pricePaid==0.0 && System.currentTimeMillis()<validUntil) {
            long to = validUntil-System.currentTimeMillis();
            if (to>0) {
                wait(to);
            }
        }
        if (o.pricePaid==0.0) {
            buyOrders.remove(o);
            System.out.println("Buy order cancelled for client "+o.clientId+".");
        }
    } catch (InterruptedException unexpected) {
        throw new Error(unexpected);
    }
    return o.pricePaid;
}

// Attempt to sell the amount of units in o for a price no less than the given
// price before System.currentTimeMillis() reach validUntil.
// Return the total number of units sold actually sold.
public synchronized int sell(SellOrder o,long validUntil) {
    try {
        o.unitsSold = 0;
        sellOrders.insert(o);
        notifyAll();
        while (o.units>0 && System.currentTimeMillis()<validUntil) {
            long to = validUntil-System.currentTimeMillis();
            if (to>0) {
                wait(to);
            }
        }
        if (o.units>0) {
            sellOrders.remove(o);
            System.out.println("Sell order cancelled for client "+o.clientId
                               +". "+o.units+" remaining unsold.");
        }
    } catch (InterruptedException unexpected) {
        throw new Error(unexpected);
    }
    return o.unitsSold;
}
```

```
// Return a Match object describing a buy order that can be fully satisfied
// together with a list of the sell orders used to satisfy the order.
// If no match is found, null is returned.
public synchronized void matchBuyersAndSellers() {
    try {
        Match m;
        while (true) {
            while ((m=nextBuyOrder())==null) {
                wait();
            }
            for(SellOrder s: m.sellers) {
                int n = m.buyer.units;
                if (n>s.units) {
                    n = s.units;
                }
                m.buyer.pricePaid += s.pricePerUnit*n;
                s.units -= n;
                s.unitsSold += n;
                if (s.units==0) {
                    sellOrders.remove(s);
                }
                System.out.println("Client "+m.buyer.clientId+" buys "+n
                                   +" units from client "+s.clientId
                                   +" at price "+s.pricePerUnit+".");
            }
            buyOrders.remove(m.buyer);
            notifyAll();
        }
    } catch (InterruptedException unexpected) {
        throw new Error(unexpected);
    }
}
```