samansafn verkefna og svara

TÖL401G - Stýrikerfi

Sverrir Sigfússon Sigríður Birna Þorvaldur Tumi Bjarni Þór

Yfirlit

1. Umræða um stýrikerfi

Verkefni: Ræðið hvort stýrikerfi séu nauðsynleg. Pælið til dæmis í tækjum sem keyra aðeins eitt forrit án nokkurs inntaks frá notanda. Er nauðsynlegt fyrir þessi tæki að hafa stýrikerfi?

Pínulitlar tölvur með jafnvel bara eitt forrit þurfa ekki endilega stýrikerfi.

Þar sem það er bara eitt forrit að keyra þarf ekki að halda vera með minnistýringu, það er bara einn aðili að nota minnið þannig það þarf ekki að passa að taka það frá og gefa það til baka.

Það þarf ekki að halda utan ferlavinnslu, *"process management"*, þar sem það er bara einn ferill í gangi á hverjum tíma

Segjum að tilgangur þessarar litlu tölvu okkar sé að mæla rakastig í herbergi, það eina sem hún gerir er að senda út hvort rakastigið sé yfir ákveðnu marki. Hér þarf aldrei að skrifa gögn þannig það þarf ekki að halda utan um "mass storage".

Í stuttu máli, þegar verið er að vinna á mjög láum level með kerfi sem eru vel hönnuð ætti ekki að þurfa stýrikerfi fyrir tölvur.

2. Posix standardinn

2.1. Hvað er Posix?

2.1.a. Hvaða stofnanir (fleiri en ein) sjá um skilgreiningu á POSIX staðalinum?

Stöðlunarhópar fyrir POSIX innihalda *Austin Group*, *ISO-hópinn* og *The Open Group*, sjá heimild

2.1.b. Hvað er nafn og númer núverandi POSIX staðals?

Nýjasti staðallinn er **POSIX.1-2017** líka þekktur sem **IEEE Std 1003.1-2017** og var gefinn út 2017.

2.1.c. Um hvað fjallar POSIX í stuttu máli?

POSIX er samansafn af stöðlum til þess að auðvelda tengingu á milli stýrikerfa, þá sérstaklega stýrikerfa byggð ofan á unix.

2.2. Um hvað fjalla eftirfarandi kaflar í POSIX staðalinum?

2.2.a. Base Definitions

"Base Definitions" fjalla um almenn kerfi og "interface" sem eru þekkt í langflestum kerfum. Þetta eru meðal annars staðlar um reglulegar segðir, hausa í C tungumálinu og skilgreiningar á gagnagrunnum.

2.2.b. System Interfaces

"System Interfaces" fjallar um, eins og nafnið gefur til kynna, viðmót sem kerfi sem uppfylla POSIX staðalinn geta gert ráð fyrir. Hér er skilgreint hvernig á að setja fram föll inn í forritum ásamt "macros" og mörgu fleira.

2.2.c. Shell & Utilities

"Shell and Utilities" kaflinn fjallar um skipanair og nytjagögn sem forrit á POSIX kerfum geta nýtt sér. Hér er talað um hvernig á að stýra skipanalínu, finna hvar í skráarkerfi maður er og fleira.

2.3. Finndu dæmi um...

2.3.a. Stýrikerfi sem uppfyllir POSIX staðalinn

Mac OS X 10.8 og seinni útgáfur af Mac OS eru POSIX vottuð

2.3.b. Stýrikerfi sem uppfyllir megnið af POSIX staðalinum

Flest linux stýrikerfi eru mestmegnis POSIX vottuð, mörg þeirra eru ábyggilega alveg POSIX vottuð en staðfesting á því er dýr og það stoppar flest distro í því að ná sér í vottun.

3. Hypervisors

Í báðum verkefnum er verið að skoða *"VM hypervisor"* sem sér um að keyra tvö stýrikerfi OS_1 og OS_2 innan tveggja virtual véla á kerfi með einn kjarna.

3.1. Tímastýring

Verkefni: Verkraðari yfirstýrikerfisins *(scheduler, hypervisor)* hefur gefið OS_1 20ms af CPU tíma. Á meðan OS_1 er að keyra í sínum gefna tímaramma klárast biðtími OS_2 og það stýrikerfi lætur sinn CPU verkraða vita. Lýsið tveimur möguleikum fyrir yfirstýrikerfið til að takast á þessu.

Ein lausn væri fyrirbyggjandi plönun *(preemptive scheduling)*. Hver hlutur fær að keyra í ákveðin tíma, og eftir það er tekinn í burtu og settur í röð. Þannig gæti yfirstýrikerfið truflað keyrslu á OS_1 og farið að keyra OS_2.

Önnur lausn væri að tímastýra kerfunum *(time-sharing)*. Þá leyfir yfirstýrikerfið OS_1 að keyra í 20ms, en minnkar tíma sem það fær í næsta tímaramma til að bæta upp fyrir tímann sem OS_2 missti.

3.2. Sýndarvélar

Verkefni: Þegar verið er að vinna með sýndarvélar eru sýndarvélarnar ótengdar hvor annari. OS_1 keyrandi á vél VM_1 getur ekki nálgast skráarkerfi ytri vélarinnar né skránna á kerfi OS_2 keyrandi á VM_2. Lýsið mögulegri aðferð til að veita vélunum möguleika á að deila skrám

Það er mögulegt fyrir fleiri en eina sýndarvél að deila skrárkerfis-staðsetningu *(file-system volume)* sem gerir þessum sýndarvélum kleyft að deila skrám.

4. Boot Process

Verkefni: Lýsið skrefunum í uppstyllingu kerfisins *(boot process)* alveg þangað til að innskráningar gluggi stýrikerfisins birtist. Leggið áherslu á hvað gerist þegar innra ræsiforritið *(bootstrap)* hefur komið kjarna *(kernel)* inn í RAM

Við getum gert ráð fyrir því að búið sé að hlaða kernel inn í minni eftir skref tvö á bootloader ferlinu. Í hausnum á kernel myndinni er örlítill kóði sem setur upp lágmarks tengingu við vélbúnaðinn, þetta gerir vélinni kleift að uncompressa kernelinn og setja hann í high memory.

Eftir að búið er að klára vinnslu á kernel, er skráarkerfið fest sem leyfir kernelinu að sjá og nálgast nauðsynlegar skrár. Eftir það er keyrt frumstyllingarforrit sem setur upp kerfiseiningar, net, skráarkerfi og þessháttar. Að lokum keyrir frumstylliforritið upp notendaviðmótið á samt fleiri kerfiseiningum. Þegar öll þessi skref eru klárað er ferill 1 keyrt upp.

5. Scheduler

Verkefni: Útfærðu scheduler aðferð sem tekst á eftirfarandi aðstæðum:
- Timer interrupt - Aðferðin fær merki um að tímarammi hafi klárast
- I/O syscall - Aðferð sem er að keyra biður um **I/O**
- I/O interrupt - Tæki lætur vita að **I/O** hafi klárast

```
// kóðinn sem var gefinn
schedulerUnfinished() {
  if (called by timer interrupt) {
    // Time slice of current process expired
   } else if (called by I/O system call) {
    // I/O request by current process
  } else if (called by I/O interrupt) {
    // I/O of process ioCompleted completed
  }

  // Further code outside if statements (if required)
}

// kóðinn sem var skilað
schedulerFinished() {
    if (called by timer interrupt) { // Time slice of current process expired
        addToTail(running, ready);
    } else if (called by I/O system call) { // I/O request by current process
        addToTail(running, waiting);
    } else if (called by I/O interrupt) { // I/O of process ioCompleted completed
        addToTail(running, waiting);
        findAndRemove(ioCompleted, waiting);
    }

    // Further code outside if statements (if required)
    interruptOn()
    if (ready == null) {
        sleep()
    } else {
        running = removeFromHead(ready);
        switchTo(running)
    }
}
```

eftirfarandi aðferðir og hlutir voru gefnir:
- `addToTail(pcb, queue)`: Append pcb to the tail of the queue queue.
- `removeFromHead(queue)`: Remove the element at the head of the queue queue and return this element. If the queue is empty, NULL is returned
- `findAndRemove(pid, queue)`: Find and return PCB entry with PId pid in the queue queue. Removes that PCB entry from the queue.
- `interruptsOn()`: Enables all interrupts again.
- `sleep()`: Puts the CPU into sleep mode, only an interrupt will wake up the CPU.
- `switchTo(pcb)`: Restarts the timer of the time slice expiration interrupt and switches to the

context stored in pcb.

6. Stýrikerfis apar (APIs)

6.1. Windows API

> **Verkefni:** Finnið hvaða system calls Windows apinn bíður upp á til að búa til og eyða prósessum og lýsið stuttlega nafni, inntaki og högun aðferðanna

Það eru þrjú **system calls** til að búa til **process** í windows apanum.
1. `CreateProcessA()`
2. `CreateProcessAsUserA()`
3. `CreateProcessAsUserW()`
4. `CreateProcessW()`

Allar þessar aðferðir búa til feril helsti munurinn á milli þeirra eru réttindin sem veitt eru til þess sem kallar á þær.

Parametrarnir sem þær taka eru eftirfarandi
- `lpApplicationName`: nafn forritsins sem kallar á fallið **(getur verið `Null`)**
- `lpCommandLine`: skipunin sem á að keyra
- `lpProcessAttributes`: bendir sem segir til um öryggisreglur fyrir processin
- `lpThreadAttributes`: bendir sem segir til um öryggisreglur fyrir aðalþráð processins
- `bInheritHandles`: boolean, segir til um hvort nýji process eigi að erfa frá process sem kallaði á sig
- `dwCreationFlags`: auka flögg sem stilla uppsetninguna
- `lpEnvironment`: bendir á svæðið þar sem processinn fær að lifa, ef `Null` lifir á svæði kallfallsins
- `lpCurrentDirectory`: slóð að working directory
- `lpStartupInfo`: bendir á upplýsingar um kerfið, monitora ofl.
- `lpProcessInformation`: bendir á svæði sem fær upplýsingar um nýja processinn

Það eru tvær aðferðir til að fjarlægja process:
1. `TerminateProcess()`
2. `ExitProcess()`

báðar aðferðirnar hafa parameterinn `uExitCode` sem er einfaldlega exit kóði ferilsins þegar hann er fjarlægður `TerminateProcess()` hefur líka `hProcess` sem er handle fyrir þann feril sem á að drepa

6.2. Posix API

> **Verkefni:** Hvaða tvær skipanir innan POSIX kerfa eru sambærilegar skipunum úr Windows apanum til þess að búa til nýjan feril

Í **POSIX** stöðluðu kerfi eru skipanirnar til þess að búa til nýtt feril í sameiningu, `fork()` og svo `exec()`. Fork býr til copy af feril og exec keyrir hann af stað.

7. Connection less sockets

Verkefni: Útbúið lausn fyrir producer / consumer vandamálið í java með *zero capacity buffer* þ.e. buffer sem ekkert getur geymt. Annað forritið starfar sem producer og sendir gögn, vaxandi heiltölur, á consumer sem tekur við þeim. Hitt forritið skal starfa sem consumer, það tekur við gögnum frá producer og prentar þau út.

```java
import java.net.*;
import java.io.*;

public class Producer {
  public static void main(String args[]) {
    DatagramSocket aSocket = null;

    try {
      aSocket = new DatagramSocket(6789);
      byte[] buffer = new byte[1000];
      byte[] out = new byte[1000];
      int item = 0;

      while (true) {
        item++;
        DatagramPacket request =
          new DatagramPacket(
            buffer,
            buffer.length
          );
        aSocket.receive(request);
        out =
          Integer.toString(item).getBytes();
        DatagramPacket reply =
          new DatagramPacket(
            out,
            out.length,
            request.getAddress(),
            request.getPort()
          );
        aSocket.send(reply);
        System.out.println(
          "item --> " + item
        );
      }
    } catch (SocketException e) {
      System.out.println(
        "Socket: " + e.getMessage()
      );
    } catch (IOException e) {
      System.out.println(
        "IO: " + e.getMessage()
      );
    } finally {
    if (aSocket != null) aSocket.close();
    }
  }
}
```

```java
import java.io.*;
import java.net.*;

public class Consumer {
  public static void main(String args[]) {
    // args[0]: message contents
    // args[1]: destination hostname
    DatagramSocket aSocket = null;
      try {
        while (true) {
          aSocket = new DatagramSocket();
          byte[] message = args[0].getBytes();
          InetAddress aHost =
            InetAddress.getByName(args[1]);
          int serverPort = 6789;
          DatagramPacket request =
            new DatagramPacket(
              message,
              message.length,
              aHost,
              serverPort
            );
          aSocket.send(request);
          byte[] buffer = new byte[1000];
          DatagramPacket reply =
            new DatagramPacket(
              buffer,
              buffer.length
            );
          aSocket.receive(reply);
          System.out.println(
            "Reply: " +
            new String(reply.getData())
          );
        }
      } catch (SocketException e) {
        System.out.println(
          "Socket: " + e.getMessage()
        );
      } catch (IOException e) {
        System.out.println(
          "IO: " + e.getMessage()
        );
      } finally {
      if (aSocket != null) aSocket.close();
    }
  }
}
```

8. Connection oriented sockets

Verkefni: Útfærið Java server sem tekur við streng en túlkar hann sem heiltölu n og framkvæmir síðan endurkvæmt Fibonacci reiknirit til að finna n-tu Fibonacci töluna. Breytið síðan útfærslunni ykkar til að nota marga þræði *(multithreaded)*.

```
import java.net.*;
import java.io.*;

public class ConnectionOrientedClient {
  public static void main(String args[]) {
    // args[0]: message contents, args[1]: destination hostname
    Socket aSocket = null;
    try {
      int serverPort = 7896;
      aSocket = new Socket(args[1], serverPort);
      DataInputStream in =
        new DataInputStream(aSocket.getInputStream());
      DataOutputStream out =
        new DataOutputStream(aSocket.getOutputStream());
      out.writeUTF(args[0]); // UTF is a string encoding
      String data = in.readUTF();
      // read a line of data from the stream
      System.out.println("Received: " + data);
    } catch (UnknownHostException e) {
      System.out.println("Socket:" + e.getMessage());
    } catch (EOFException e) {
      System.out.println("EOF:" + e.getMessage());
    } catch (IOException e) {
      System.out.println("readline:" + e.getMessage());
    } finally {
      if (aSocket != null)
      try { aSocket.close(); }
      catch (IOException e) {
        System.out.println("close:" + e.getMessage());
      }
    }
  }
}
```

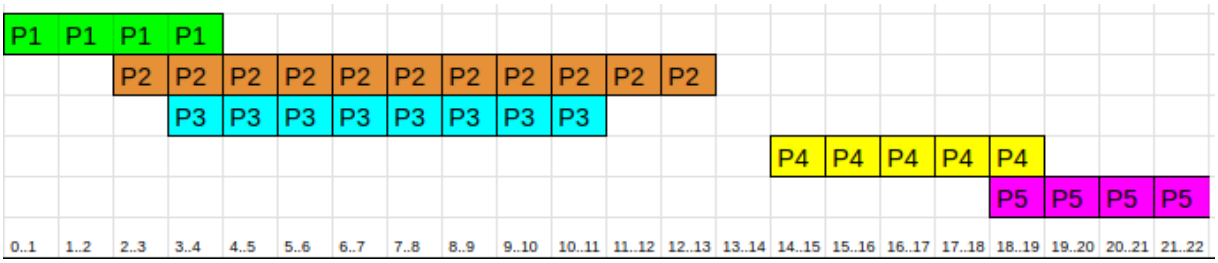
```
import java.net.*;
import java.io.*;

public class ConnectionOrientedServer {
  public static void main(String args[]) {
  try {
    int serverPort = 7896; // the server port
    ServerSocket listenSocket = new ServerSocket(serverPort);
    while (true) {
      Socket clientSocket = listenSocket.accept();
      Connection c = new Connection(clientSocket); // Handle request
      c.start();
    }
  } catch (IOException e) {
    System.out.println(
      "Listen socket:" + e.getMessage());
    }
  }
}
```

9. Scheduling reiknirit

Skoðið eftirfarandi töflu og / eða mynd:

Process	P1	P2	P3	P4	p5
Arrival time	0	2	3	14	18
Service time	4	11	8	5	4
Priority	mid	low	high	mid	high

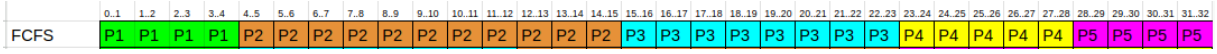


9.1. Raðið ferlum eftir reiknirítum og reiknið tíma

- Þjónustutími *(residance time)* fyri feril er reiknaður sem:
  - completion time − arrival time
  - waiting time + service time
- Biðtími *(waiting time)* fyrir feril er reiknaður sem:
  - residance time - service time

9.1.a. FCFS (first come first served)

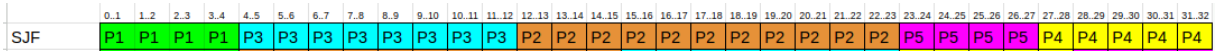
Hér er klárað ferla í þeirri röð sem þeir koma inn. Ferlar eru geymdir í einhverskonar biðröð þar sem næsti í röðinni er tekinn út eftir að sá á undan klárar.



- Meðalþjónustutími = $\frac{4-0+15-2+23-3+28-14+32-18}{5} = 13$
- Meðalbiðtími = $\frac{0+4-2+15-3+23-14+28-18}{5} = 6.6$

9.1.b. SJF (shortest job first)

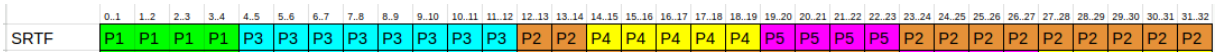
Hér er svipað og FCFS við höfum biðröð þar sem valið er úr eftir að núverandi ferill klárar, nema núna veljum við minnsta feril sem er í biðröðinni og klárum hann.



- Meðalþjónustutími = $\frac{4+23-2+12-3+32-14+27-18}{5} = \frac{4+21+9+18+9}{5} = 12.2$
- Meðalbiðtími = $\frac{0+12-2+4-3+27-14+23-18}{5} = \frac{0+10+1+13+5}{5} = 5.8$

9.1.c. SRTF (shortest remaining time first)

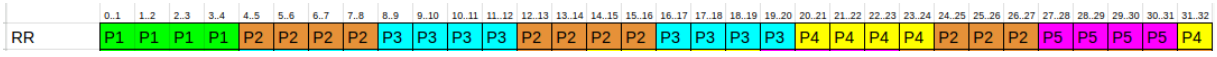
Hérna höfum við biðröðina okkar nema hvað við bíðum ekki endilega eftir því að fyrri ferill hafi klárað til að byrja á nýjum ferli. Þegar nýr ferill kemur inn þá athugum við hvort hann sé með styttri vinnslutíma og ef svo er geymum við núverandi feril og byrjum þennan nýja.



- Meðalþjónustutími = $\frac{4+32-2+12-3+19-14+23-18}{5} = \frac{4+30+9+5+5}{5} = 10.6$
- Meðalbiðtími = $\frac{0+30-11+9-8+5-5+5-4}{5} = \frac{0+19+1+0+1}{5} = 4.2$

9.1.d. RR (round robin)

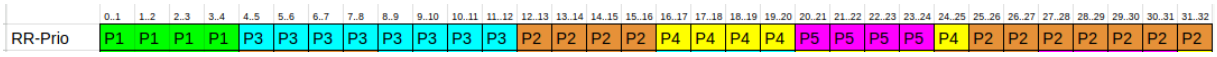
Round robin notast við fyrirfram skilgreindan *"time quantum "* sem í þessu tilfelli er 4. Það þýðir að hver ferill fær að vinna í 4 einingar af tíma og þá er næsti ferill í biðröðinni valinn. Þetta er endurtekið þar til allir ferlar eru kláraðir.



- Meðalþjónustutími = $\frac{4+27-2+20-3+32-14+31-18}{5} = \frac{4+25+17+18+13}{5} = 15.4$
- Meðalbiðtími = $\frac{0+25-11+17-8+18-5+13-4}{5} = \frac{0+14+9+13+9}{5} = 9$

9.1.e. RR-prio (round robin with priority)

Svipaðar pælingar og með round robin nema hvað við skiptum ferlum upp í mismunandi biðraðir eftir forgangi, sjá í töflunni *"priority"* fyrir hvern feril. Þegar tíminn *"time quantum"* er liðinn veljum við næst þá biðröð sem hefur hæstan forgang og vinnum með hana þangað til allir hennar ferlar hafa klárað.



- Meðalþjónustutími = $\frac{4+32-2+12-3+25-14+24-18}{5} = \frac{4+30+11+9+6}{5} = 12$
- Meðalbiðtími = $\frac{0+30-11+11-8+9-5+6-4}{5} = \frac{0+19+3+4+2}{5} = 5.6$

9.2. Hversvegna er ómögulegt fyrir tvo ferla að klára á sama tíma?

Nýr process er búinn til með kalli á CreateProcess í windows eða Fork á Posix kerfi á meðan annar process keyrir. Á kerfi með aðeins einn gjörva (processor system) má aðeins kalla einu sinni á þessar aðferðir og engir tveir processar geta komið á sama tíma

10. Ferlaröðun

10.1. Raðið eftir reglum

Verkefni: Raðið ferlum **[P1, P2, P3]** með Round Robin þar sem time quantum er 4.

Reglur:

arr(t): Tímapunktur þegar ferill kemur inn, fylgir x í (x, y) tímapunkti

cpu_io(d_{cpu}, d_{io}): Ferill vinnur í d_{cpu} tíma og bíður svo í d_{io} eftir að klára

cpu(d): Ferill vinnur í d tíma og hættir

Gildi ferla:

- P1: arr(0), cpu_io(1, 2), cpu_io(2, 3), cpu(5)
- P2: arr(2), cpu_io(5, 2), cpu(1)
- P3: arr(4), cpu_io(1, 1), cpu(2)

Timespan	Running	Ready queue		Waiting Queue		
0..1	P1	---	---	---	---	---
1..2	---	---	---	P1	---	---
2..3	P2	---	---	P1	---	---
3..4	P2	P1	---	---	---	---
4..5	P2	P1	P3	---	---	---
5..6	P2	P1	P3	---	---	---
6..7	P1	P3	P2	---	---	---
7..8	P1	P3	P2	---	---	---
8..9	P3	P2	---	P1	---	---
9..10	P2	---	---	P1	P3	---
10..11	---	---	---	P1	P3	P2
11..12	P1	---	---	P3	P2	---
12..13	P1	P3	---	P2	---	---
13..14	P1	P3	---	P2	---	---
14..15	P1	P3	P2	---	---	---
15..16	P3	P2	P1	---	---	---
16..17	P3	P2	P1	---	---	---
17..18	P2	P1	---	---	---	---
18..19	P1	---	---	---	---	---
19..20	---	---	---	---	---	---
20..21	---	---	---	---	---	---

10.2. Reiknið meðalþjónustutíma

Þjónustutími fyrir hvern feril er lokatími - upphafstími, meðaltíminn verður því:

$$\frac{19-0+18-2+17-4}{3} = \frac{19+16+13}{3} = 16$$

11. Þræðir og race condition

11.1. Útfærsla á race condition

Verkefni: Útfærið forrit í java sem býr til tvo nýja þræði sem eru í race condition. Þræðirnir eiga að hækka breytu **in** um 1 innan lykkju sem keyrir n sinnum þar sem n er heiltala tekin inn af skipanalínu. Þar sem **in** breytan er volitile getur komið upp race condition og líklegra eftir því sem n er stærra.

```
public class MyAssignment11 extends Thread {
  private static volatile long in; // línan sem leyfir þráðum að deila breytu
  private long iterations;

  MyAssignment11(long _in) {
    iterations = _in;
  } // iterations jafngildir n

  public static long main(long iterationsPerThread) { // Do not modify this line!
    MyAssignment11 t1 = new MyAssignment11(iterationsPerThread);
    MyAssignment11 t2 = new MyAssignment11(iterationsPerThread);

    // báðir þræðirnir keyra sitthvort run() samhlið
    t1.start();
    t2.start();

    // join() sameinar þræði aftur inn í main þráðin
    try {
      t1.join();
      t2.join();
    } catch (Exception e) {
      System.out.println(e);
    }

    return in;
  }

  @Override
  public void run() {
    for (long i = 0; i < iterations; i++) {
      ++in;
    }
  }
}
```

11.2. Gildi sem á að koma

Verkefni: Hvert er gildið sem á að prentast út í lokin og hvernig er hægt að sjá að race condition hafi átt sér stað?

Gildið sem í fullkomnum heimi ætti að koma út væri $2 \cdot n$ en við getum séð að það hafi komið upp race condition ef prentaða gildið er minna en $2n$. Þetta gerist vegna þess að báðir þræðirnir reyna að taka **in** frá á sama tíma og hækka það, en bara annar þráðurinn fær að vista breytinguna.

12. Peterson reikniritið

Verkefni: Notið kóðann úr verkefni 11 og breytið þannig að það noti útfærslu Peterson's reikniritsisn og komið þannig í veg fyrir race conditions

```
public class MyAssignment12 extends Thread {
  public static volatile long in;
  private static long n;
  private int id;
  public static volatile boolean[] flag = {false,false};
  public static volatile int turn;

  public MyAssignment12 (long _n, int _id) {
    n = _n;
    in = 0;
    id = _id;
  }


  public static long main(long iterationsPerThread) {
    Thread t1 = new MyAssignment12(iterationsPerThread, 0);
    Thread t2 = new MyAssignment12(iterationsPerThread, 1);

    t1.start();
    t2.start();

    try {
      t1.join();
      t2.join();
    } catch (InterruptedException ie) {
      System.out.println("Interrupted while waiting thread");
    }
    return in;
  }

  public void run() {
    for (int i = 1; i <= myIteration; i++){
      increment();
    }
  }

  public void increment() {
    flag[id] = true;
    turn = 1-id;
    while(flag[1-id] && turn==1-id) { /* waiting for turn */ }
    long next_free_slot = in;
    next_free_slot ++;
    in = next_free_slot;
    flag[id] = false;
  }
}
```

13. Semaphores

Verkefni: Notið semaphores til að leysa eftirfarandi samstilli *(synchronisation)* vandamál:

Fjölskylda sem samanstendur af **mömmu**, **pabba** og **tveimur börnum** ,sem deila sömu hegðun, byrja alla daga eins.

- Allir byrja daginn á því að fara á klósettið (`useToilet()`)
  - Röðin sem þau nota klósettið er ekki sett í stein
  - Það er bara eitt klósett á heimilinu
- Eftir að pabbinn hefur notað klósettið býr hann til drykki fyrir krakkana (`prepareDrinks()`)
- Eftir að mamman hefur notað klósettið býr hún til mat fyrir fjölskylduna (`prepareFood()`)
- Ef bæði matur og drykkir eru til þá borða krakkarnir (`haveBreakfast()`)
- Ef barn er búið að borða tekur mamman það og keyrir í skólann `takeAndDriveToSchool()`
- Ef barn er búið að borða tekur pabbinn af borðinu og gengur frá `(clearTable())`

Það sem þarf að passa hér er að hlutir séu ekki gerðir áður en allir þeir sem hluturinn hefur áhrif á hafa klárað sitt. Þetta er gert með því að nota `init()`, `wait()` og `signal()` á réttum stöðum.

```
Semahpore toilet = new Semaphore(1);
Semaphore food = new Semaphore(0);
Semahpore drink = new Semaphore(0);
Semaphore foodFin = new Semaphore(0);
Semaphore drinkFin = new Semaphore(0);

parallel {
  child(),
  child(),
  mother(),
  father()
}
```

```
child() {
  toilet.wait()
  useToilet()
  toilet.signal()
  food.wait()
  drink.wait()

  haveBreakfast()

  foodFin.signal()
  drinkFin.signal()
}
```

```
mother() {
  toilet.wait()
  useToilet()
  toilet.signal()

  prepareFood()
  // fyrra barn látið vita
  food.signal()
  // seinna barn látið vita
  food.signal()

  // fyrra barn klárar
  foodFin.wait()
  // seinna barn klárar
  foodFin.wait()
  takeAndDriveToSchool()
}
```

```
father() {
  toilet.wait()
  useToilet()
  toilet.signal()

  prepareDrinks()
  // fyrra barn látið vita
  drink.signal()
  // seinna barn látið vita
  drink.signal()

  // fyrra barn klárar
  drinkFin.wait()
  // seinna barn klárar
  drinkFin.wait()
  clearTable()
}
```

14. Java semaphores

Verkefni: Breytið lausn úr verkefni 11 eða 12 þannig að vandamálið sé leyst með notkun semaphora úr java

```
import java.util.concurrent.Semaphore;

public class MyAssignment14 extends Thread {
  private static Counter counter;
  private static long max;
  private static Semaphore sem;

  public static long main(long iterationsPerThread) {
    Thread thread1 = new MyAssignment14();
    Thread thread2 = new MyAssignment14();

    thread1.setName("0");
    thread2.setName("1");
    sem = new Semaphore(1);
    max = iterationsPerThread;
    counter = new Counter();

    thread1.start();
    thread2.start();

    try {
      thread1.join();
      thread2.join();
    }
    catch(Exception ex) {
      System.out.println("Exception" + ex);
    }
    return counter.getIn();
  }

  public void run() {
    try {
      for (int i = 1; i <= max; i++) {
        System.err.print(this.getName());
        sem.acquire(); // eins og wait í dæmi 13
        counter.increment(max);
        sem.release(); // eins og signal í dæmi 13
      }
    } catch (InterruptedException ie) { /* error */ }
  }
}
```

```
public class Counter {
  public static volatile long in = 0;
  public void increment(long max) {
    long next_free_slot = in + 1;
    in = next_free_slot;
  }
  public long getIn() {
    return in;
  }
}
```

15. Meiri semaphores

Verkefni: Jói bakari vill betrumbæta bollu-bökunar ferlið í bakaríinu sínu með því að nota nýtt samhliða bakara-ferli *(parallel baker processes)* sem nýtir sér semaphores til að stilla saman bakara. Ferlið hljómar svona:

- Einn yfirbakari skaffar þremur undirbökurum vinnu og hráefnum
- Hver undirbakari framleiðir endalaust af bollum úr þremur hráefnum
- Hver undirbakari hefur endalaust magn af einu hráefni en vantar hin tvö til að baka bollu
- Yfirbakarinn hefur skaffar endalaust handahófskennd pör af hráefnum til undirbakarana
- Undirbakarinn sem á það hráefni sem vantar
  1. Tekur við parinu
  2. Lætur vita að hann hafi tekið parið
  3. Býr til bollu

```
Semaphore on_table = new Semaphore(0);
Semaphore offer_plain = new Semaphore(0);
Semaphore offer_cream = new Semaphore(0);
Semaphore offer_choco = new Semaphore(0);

// master:
while(true) {
  int choice = random(1,3)
  switch(choice) {
    case 1:
      offer(cream, choco);
      offer_plain.signal();
      break;

    case 2:
      offer(choco, plain);
      offer_cream.signal();
      break;

    case 3:
      offer(plain, cream);
      offer_choco.signal();
      break;
  }

  on_table.wait();
}
```

```
// Assistants:
// ass-cream:
while(true) {
  offer_cream.wait();
  ingredients = fetch();
  on_table.signal();

  assemble(ingredients, cream)
}

// ass-choco:
while(true) {
  offer_choco.wait();
  ingredients = fetch();
  on_table.signal();

  assemble(ingredients, choco)
}

// ass-plain:
while(true) {
  offer_plain.wait();
  ingredients = fetch();
  on_table.signal();

  assemble(ingredients, plain)
}
```