

samansafn verkefna og svara

TÖL401G - Stýrikerfi

Sverrir Sigfúrssón
Bjarni Þór

Sigríður Birna
Hákon Ingi

Þorvaldur Tumi
Sigurður Örn

Yfirlit

1. Umræða um stýrikerfi	3
2. Posix standardinn	4
2.1. Hvað er Posix?	4
2.2. Um hvað fjalla eftirfarandi kaflar í POSIX staðalinum?	4
2.3. Finndu dæmi um... ..	4
3. Hypervisors	5
3.1. Tímastýring	5
3.2. Sýndarvélar	5
4. Boot Process	6
5. Scheduler	7
6. Stýrikerfis apar (APIs)	8
6.1. Windows API	8
6.2. Posix API	8
7. Connection less sockets	9
8. Connection oriented sockets	10
9. Scheduling reiknirit	11
9.1. Raðið ferlum eftir reikniritum og reiknið tíma	11
9.2. Hversvegna er ómögulegt fyrir tvo ferla að klára á sama tíma?	12
10. Ferlaröðun	13
10.1. Raðið eftir reglum	13
10.2. Reiknið meðalþjónustutíma	13
11. Þræðir og race condition	14
11.1. Útfærsla á race condition	14
11.2. Gildi sem á að koma	14
12. Peterson reikniritið	15
13. Semaphores	16
14. Java semaphores	17
15. Meiri semaphores	18
16. Monitor-semahpores	19
17. Deadlock prevention	20
17.1. Réttthentu heimspekingarnir	20
17.2. Örvhenti félaginn	20
18. Deadlock Algorithms	21
18.1. Safety reikniritið	21
18.2. Bankers reikniritið	22
18.3. Bankers aftur?	22
19. Dynamic memory managment systems	23
19.1. First fit	23
19.2. Next fit	23
19.3. Best fit	24

19.4. Worst fit	24
20. Paged memory management	25
20.1. Í hversu margar síður skiptist rökvisfangssvæðið?	25
20.2. Í hversu marga ramma skiptist raunvisfangssvæðið?	25
20.3. Hversu mörgum forritum má hlaða inn í minni með skorðum?	25
20.4. Hversu mörgum forritum má hlaða í minni án skorða?	25
20.5. Hversu margar færslur (<i>entries</i>) hefur síðutafla kerfisins?	25
20.6. Hversu marga bita þarf til að tákna línu í síðutöflu?	25
20.7. Hvað er raunvisfang rökvisfangsins 42?	25
20.8. Hvað er raunvisfang rökvisfangsins 191?	25
21. Úthlutunarreiknirit	26
21.1. FIFO	26
21.2. Second-Chance / Clock	26
21.3. Enhanced Second-Chance (NRU)	26
21.4. Least-Recently-Used (LRU)	27
21.5. Optimal	27
22. Paging í praktík	28
22.1. Hversu margar síður eru nýttar undir data í sýndarminni?	28
22.2. Hversu mörgum page faults lendum við í upphaflega?	28
22.3. Hvernig má minnka fjölda page-faulta?	28
23. FAT systems	29
23.1. Uppskipting skráa	29
23.2. Kemst eftirfarandi fyrir?	29
24. I-node file system	30
24.1. Hámarksstærð kerfis	30
24.2. Leit að forriti	30
24.3. Skráarkerfis uppfærslur	30

1. UMRÆÐA UM STÝRIKERFI

Verkefni: Ræðið hvort stýrikerfi séu nauðsynleg. Þælið 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-hópin* 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 staðallinn 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 skrána á kerfi OS_2 keyrandi á VM_2 . Lýsið mögulegri aðferð til að veita vélunum möguleika á að deila skrá

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

4. BOOT PROCESS

Verkefni: Lýsið skrefunum í uppstillingu 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 frumstillingarforrit sem setur upp kerfiseiningar, net, skráarkerfi og þessháttar. Að lokum keyrir frumstylliborritið 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*/
        if (running != null) { addToTail(running, waiting); }
        temp_completed = findAndRemove(ioCompleted, waiting);
        addToTail(temp_completed, ready)
    }

    // 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 býður 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.

```
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();
        }
    }
}
```

```
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());
            }
        }
    }
}
```

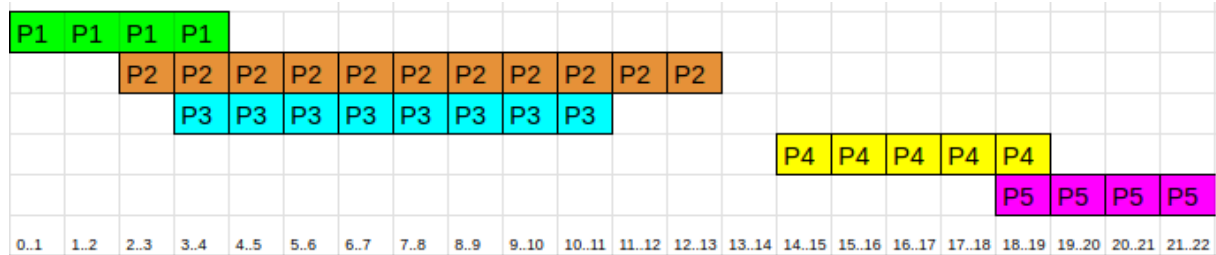
```
import java.net.*;
import java.io.*;

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

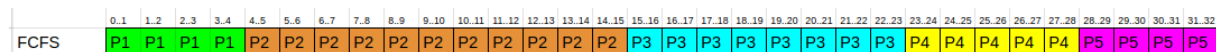


9.1. RAÐIÐ FERLUM EFTIR REIKNIRITUM OG REIKNIÐ TÍMA

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

9.1.A. FCFS (FIRST COME FIRST SERVED)

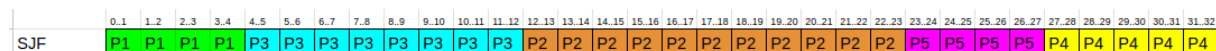
Hér er klárað ferla í þeirri röð sem þeir koma inn. Ferlar eru geymdir í einhverskonar biðröð þar sem næsti í röðinni er tekinn út eftir að sá á undan klárar.



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

9.1.B. SJF (SHORTEST JOB FIRST)

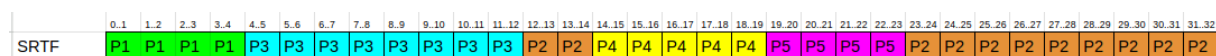
Hér er svipað og FCFS við höfum biðröð þar sem valið er úr eftir að núverandi ferill klárar, nema núna veljum við minnsta feril sem er í biðröðinni og klárum hann.



- 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 biðum 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.

	0.1	1.2	2.3	3.4	4.5	5.6	6.7	7.8	8.9	9.10	10.11	11.12	12.13	13.14	14.15	15.16	16.17	17.18	18.19	19.20	20.21	21.22	22.23	23.24	24.25	25.26	26.27	27.28	28.29	29.30	30.31	31.32
RR	P1	P1	P1	P1	P2	P2	P2	P2	P3	P3	P3	P3	P2	P2	P2	P2	P3	P3	P3	P3	P4	P4	P4	P4	P2	P2	P2	P5	P5	P5	P5	P4

- 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 þælingar 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.

	0.1	1.2	2.3	3.4	4.5	5.6	6.7	7.8	8.9	9.10	10.11	11.12	12.13	13.14	14.15	15.16	16.17	17.18	18.19	19.20	20.21	21.22	22.23	23.24	24.25	25.26	26.27	27.28	28.29	29.30	30.31	31.32
RR-Prio	P1	P1	P1	P1	P3	P3	P3	P3	P3	P3	P3	P3	P2	P2	P2	P2	P4	P4	P4	P4	P5	P5	P5	P5	P4	P2	P2	P2	P2	P2	P2	P2

- 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 volatile 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 þræðum 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 þræðin
        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 þræðurinn 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.

```
Semaphore toilet = new Semaphore(1);
Semaphore food = new Semaphore(0);
Semaphore 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()
    // fyrri barn látið vita
    food.signal()
    // seinna barn látið vita
    food.signal()

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

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

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

    // fyrri 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 semaphore 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)
}
```

16. MONITOR-SEMAPHORES

Verkefni: Útfærið ykkar eigin semaphores, í stað innbyggðu java semaphora, með því að nota **Monitor** hugmyndarfræðina. Notið þá svo til að uppfæra verkefni 11 / 12

```
public class Counter {
    private volatile long in = 0;

    private MonitorSemaphore myMonSem =
        new MonitorSemaphore(1);

    public void increment() {
        long next_free_slot;

        myMonSem.op1MyWait();

        next_free_slot = in;
        next_free_slot = next_free_slot + 1;
        in = next_free_slot;

        myMonSem.op2MySignal();
    }

    public long getIn() {
        return in;
    }
}
```

```
public class MonitorSemaphore {
    private volatile int count;

    public MonitorSemaphore(int init) {
        count = init;
    }

    public synchronized void op1MyWait() {
        count = count - 1;
        try {
            if (count < 0) {
                this.wait();
            }
        } catch (Exception e) {
            // Error
        }
    }

    public synchronized void op2MySignal() {
        count = count + 1;
        try {
            if (count <= 0) {
                this.notify();
            }
        } catch (Exception e) {
            // Error
        }
    }
}
```

```
public class MyAssignment16 extends Thread {
    private long theNumberOfIterations;
    private Counter theCounter;

    public MyAssignment16(Counter counter, long iterationsPerThread) {
        theCounter = counter;
        theNumberOfIterations = iterationsPerThread;
    }

    public void run() {
        for (long i = 0; i < theNumberOfIterations; ++i) {
            System.err.print(this.getName());
            theCounter.increment();
        }
    }

    public static long main(long iterationsPerThread) {
        Counter counter = new Counter();
        MyAssignment16 raceDemo0 = new MyAssignment16(counter, iterationsPerThread);
        MyAssignment16 raceDemo1 = new MyAssignment16(counter, iterationsPerThread);
        raceDemo0.setName("0");
        raceDemo1.setName("1");

        raceDemo0.start();
        raceDemo1.start();
        try {
            raceDemo0.join();
            raceDemo1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return counter.getIn();
    }
}
```

17. DEADLOCK PREVENTION

Verkefni: Við höfum 5 heimspekinga sem haga sér allir svipað, þeir sitja allir í hring með prjóna báðum megin við sig. Það eru þannig 5 prjónar í heildina. Hver heimspekingur hugsar í smá tíma og bíður þangað til að prjóninn hægra megin við hann er laus og tekur hann upp, síðan bíður hann eftir því að prjóninn á vinstri hönd sé laus áður en hann getur borðað.

17.1. RÉTTHENTU HEIMSPEKINGARNIR

Fyrst hugsum við um heimspekinga sem haga sér allir eins

Við getum séð strax að ef allir grípa sinn hægri prjón þá endum við með deadlock vegna þess að enginn getur fengið vinstri prjóninn sem þeir þurfa til að borða

- Þetta er dæmi um circular wait
- Þetta er líka dæmi um no preemption þar sem heimspekingarnir geta ekki skilað prjónum fyrr en eftir að þeir borða
- Það má líka sjá hold and wait þar sem hluturinn sem heimspekingarnir þurfa eru í notkun af öðrum heimspekingum
- Þar sem að aðeins einn heimspekingur getur haldið á hverjum prjóni í einu þá er þetta líka mutual exclusion

Ef heimspekingarnir myndu sýna einhverja mannasiði og taka tillit til hægri handar nágranna síns þá myndi þetta hugsanlega ganga upp en þar sem dæmið uppfyllir öll fjögur deadlock skilyrðin þá má segja að þetta endi í deadlock

17.2. ÖRVHENTI FÉLAGINN

Nú hugsum við um alla eins nema einn örvhentan

Gerum ráð fyrir því að þeir hagi sér eins og í fyrra dæminu, núna grípa þeir allir prjóninn sér á hægri hönd nema sá örvhenti, hann reynir að grípa prjóninn sér á vinstri hönd en nágranni hans er nú þegar búinn að grípa hann Þetta eru góðar fréttir fyrir vinstri nágranna örvhenta heimspekingsins því nú getur hann tekið sinn vinstri prjón og fengið sér að borða Þegar nágranninn klárar skilar hann prjónunum sínum og vinstri réttthenti nágranni hans borðar og setur af stað keðju Þessi keðja endar þegar örvhenti nær loksins að fá vinstri prjón og allir geta borðað húrra :)

Það að hafa örvhenta heimspekingin kemur alltaf í veg fyrir deadlock

18. DEADLOCK ALGORITHMS

Verkefni: Við fáum gefið þrjá ferla $[P_1, P_2, P_3]$ sem nota í sameiningu tvær auðlindir R_1 og R_2 . Báðar R_i breytur hafa 9 stök í kerfinu til nýtingar. Hámarksnýting fyrir ferla $[P_1, P_2, P_3]$ er skilgreind á forminu $P_i = (M_1, M_2)$, þar sem ferill X_i notar að hámarki M_1 af R_1 osfr.

$A = (6, 3), B = (8, 2), C = (4, 1)$

Í upphafi eru allar auðlindir nothæfar en síðan taka ferlarnir frá eftirfarandi auðlindir:

$P_1(2, 1), P_2(3, 1), P_3(2, 1)$

Við endum þá með nýtingarfylki (*allocation matrix*), auðlindarvektor (*avail. resource vec.*) og beiðnafylki (*request matrix*):

$$C = \begin{pmatrix} 2 & 1 \\ 3 & 1 \\ 2 & 1 \end{pmatrix}, A = (2 \ 6), R = \begin{pmatrix} 4 & 2 \\ 5 & 1 \\ 2 & 0 \end{pmatrix}$$

18.1. SAFETY REIKNIRITIÐ

Verkefni: Notið safety reikniritið til að athuga hvort þessi staða leiðir til deadlocks

Finnið feril þar sem hámarksnýting - núverandi nýting er minni eða jöfn því sem auðlindafylkið hefur upp á að bjóða

C hefur $\max = (4, 1)$ og nýtingu $= (2, 1)$ og þar sem $(4 - 2, 1 - 1) = (2, 0) \leq (2, 6)$ þá getum við merkt C sem **klárað** og tekið auðlindir þess inn í auðlindafylkið. Skoðum núna fylkin okkar:

$$C = \begin{pmatrix} 2 & 1 \\ 3 & 1 \\ 0 & 0 \end{pmatrix}, A = (4 \ 7), R = \begin{pmatrix} 4 & 2 \\ 5 & 1 \\ 0 & 0 \end{pmatrix}$$

Gerum nú það sama fyrir það sem eftir er af nýtingarfylkinu: $A_{\max} - A_C = (4, 2) \leq (4, 7)$, merkjum því A sem klárað og tökum auðlindirnar

$$C = \begin{pmatrix} 0 & 0 \\ 3 & 1 \\ 0 & 0 \end{pmatrix}, A = (6 \ 8), R = \begin{pmatrix} 0 & 0 \\ 5 & 1 \\ 0 & 0 \end{pmatrix}$$

Reynum að klára með því að gera líka við B : $B_{\max} - B_C = (5, 1) \leq (6, 8)$ við getum því klárað B og sagt með fullvissu að þessi staða endi ekki í deadlock, lokastaða:

$$C = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, A = (9 \ 9), R = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

18.2. BANKERS REIKNIRITIÐ

Verkefni: Höldum áfram með upphafsstöðuna hér fyrir ofan nema breytum aðeins P_1 vill sleppa einum úr R_1 og P_2 vill svo fá tvo af R_1 í staðin, ef báðar af þessum beiðnum eru samþykktar þá endum við með upphafsstöðu sem lítur svona út:

$$C = \begin{pmatrix} 1 & 1 \\ 5 & 1 \\ 2 & 1 \end{pmatrix}, A = (1 \ 6), R = \begin{pmatrix} 5 & 2 \\ 3 & 1 \\ 2 & 1 \end{pmatrix}$$

Notið núna bankers reikniritið til þess að ákvarða hvort það eigi að veita þessar beiðnir

Bankers reikniritið snýst um að skoða beiðnir og athuga hvort, ef samþykktar, staðan sem endar verði örugg með því að keyra safety reikniritið eins og fyrir ofan.

Við sjáum hér strax að ef við samþykkjum beiðni P_2 þá endum við strax í deadlock þar sem við getum ekki uppfyllt neinar kröfur í R

Hinsvegar ef við skoðum fylkin þar sem við samþykktum bara beiðni P_1 :

$$C = \begin{pmatrix} 1 & 1 \\ 3 & 1 \\ 2 & 1 \end{pmatrix}, A = (3 \ 6), R = \begin{pmatrix} 5 & 2 \\ 5 & 1 \\ 2 & 1 \end{pmatrix}$$

Núna, eftir að við klárum P_3 getum við klárað bæði P_1 og P_2 . Niðurstaðan er því að samþykkja beiðnina frá P_1 og geyma beiðnina frá P_2 . Niðurstaðan er því að samþykkja beiðnina frá P_1 og geyma beiðnina frá P_2 .

18.3. BANKERS AFTUR?

Endurtökum síðasta verkefni nema rétt eftir að við höfnum P_2 þá hættir P_3 og skilar öllum sínum auðlindum. Þá þarf að keyra bankers fyrir beiðni P_2 aftur, virkar það í þetta skipti?

Upphafsstaðan okkar er þá núna:

$$C = \begin{pmatrix} 1 & 1 \\ 5 & 1 \\ 0 & 0 \end{pmatrix}, A = (5 \ 7), R = \begin{pmatrix} 5 & 2 \\ 3 & 1 \\ 0 & 0 \end{pmatrix}$$

Nú vinna auðlindirnar úr P_3 á móti því sem P_2 bað um og við getum klárað bæði P_1 og P_2 þannig þessi staða er ekki deadlock og við getum samþykkt beiðnina.

19. DYNAMIC MEMORY MANAGMENT SYSTEMS

Verkefni: Komið eftirfarandi gildum inn í rétt hólfið samkvæmt tilheyrandi reikniriti

	m1=16 KB	m2=8 KB	m3=20 KB	m4=14 KB	m5=5 KB
Fill in allocations as follows:	Initially free:	26 KB	30 KB		
	1. Allocation	m2=8 KB	m1=16 KB		
	2. Allocation	18 KB free	m5=5 KB		
	3. Allocation		9 KB free		

Gildin eru sett inn í vaxandi röð, þe m_1 fer inn á undan m_2 , skref eru talin þegar verið er að leyta að plássi til að passa inn í, sama hvort bendir sé hreyfður eða ekki.

19.1. FIRST FIT

First fit snýst um að setja gildi inn í fyrsta hólfið sem hefur nóg pláss, þegar hólfið er fundið er leit hafin aftur fyrir næsta á byrjunarreit.

First Fit							
Initially free:	15 KB	10 KB	26 KB	30 KB	21 KB	13 KB	17 KB
1. Allocation	m2=8 KB		m1=16 KB	m3=20 KB	m4=14 KB		
2. Allocation	m5=5 KB		10 KB	10 KB	7 KB		
3. Allocation	2 KB						
Number of steps:	14	<- Fill in here number of check steps (you can use next line for your internal calculations)					
	3	1	4	5	1		

19.2. NEXT FIT

Next fit er svipað og first fit nema í stað þess að fara aftur á byrjun eftir að finna hólfið, þá er haldið áfram frá síðasta fundna hólfi.

Next Fit							
Initially free:	15 KB	10 KB	26 KB	30 KB	21 KB	13 KB	17 KB
1. Allocation			m1=16 KB	m3=20 KB	m4=14 KB		
2. Allocation			m2=8 KB	10 KB	m5=5 KB		
3. Allocation			2 KB		2 KB		
Number of steps:	9	<- Fill in here number of check steps (you can use next line for your internal calculations)					
	3	1	2	2	1		

19.3. BEST FIT

Hér er alltaf valið úr öllum opnum hólfum, það sem mun skilja minnst eftir ef sett er í það hólf, þar sem það er alltaf vitað um alla er skrefafjöldi jafn fjölda hólfanna sinnum fjölda staka.

Best Fit							
Initially free:	15 KB	10 KB	26 KB	30 KB	21 KB	13 KB	17 KB
1. Allocation	m4=14 KB	m2=8 KB			m3=20 KB	m5=5 KB	m1=16 KB
2. Allocation	1 KB	2 KB			1 KB	8 KB	1 KB
3. Allocation							
Number of steps:	35	<- Fill in here number of check steps (you can use next line for your internal calculations)					
	7	7	7	7	7		

19.4. WORST FIT

Öfugt við best fit, finnur hólfid sem mun skylda mest eftir þegar stak er sett þar.

Worst Fit							
Initially free:	15 KB	10 KB	26 KB	30 KB	21 KB	13 KB	17 KB
1. Allocation			m2=8 KB	m1=16 KB	m3=20 KB		m5=5 KB
2. Allocation			m4=14 KB	14 KB	1 KB		12 KB
3. Allocation			4 KB				
Number of steps:	35	<- Fill in here number of check steps (you can use next line for your internal calculations)					
	7	7	7	7	7	7	

20. PAGED MEMORY MANAGEMENT

Verkefni: Við fáum gefið PMMU sem styður 10 bita raunlæg vistföng (*physical addresses*) en getur samt aðeins notað 8 bita rökivistföng (*logical addresses*). Síðustærðin er 64 bæti og stærð orða er 1 bæti.

20.1. Í HVERSU MARGAR SÍÐUR SKIPTIST RÖKVISTFANGSSVÆÐIÐ?

Rökivistfangssvæðið inniheldur $2^8 = 256$ vistföng og hver síða inniheldur 64 orð, ef við deilum fjölda vistfanga með fjölda orða í síðu fáum við fjölda síða sem er $\frac{256}{64} = 4$

20.2. Í HVERSU MARGA RAMMA SKIPTIST RAUNVISTFANGSSVÆÐIÐ?

Gerum ráð fyrir því að rammastærð sé sú sama og síðustærð 64 bæti, við höfum $2^{10} = 1024$ raunvistföng sem skiptast yfir $\frac{2^{10}}{2^6} = 2^4 = 16$ ramma

20.3. HVERSU MÖRGUM FORRITUM MÁ HLAÐA INN Í MINNI MEÐ SKORÐUM?

Verkefni: Við höfum þær skorður að ekkert swapping sé notað, gerum ráð fyrir því að allt raunvistfangssvæðið sé í boði fyrir forrit sem nota allt rökivistfangssvæðið sitt.

Þá einfaldlega deilum við fjölda ramma með fjölda síðna sem er $\frac{16}{4} = 4$

20.4. HVERSU MÖRGUM FORRITUM MÁ HLAÐA Í MINNI ÁN SKORÐA?

Verkefni: Núna getum við notað swapping, er möguleiki að segja til um hversu mörg forrit má hlaða inn í minni?

Það er hægt að koma fleiri forritum inn í minni ef notast er við swapping eða demand paging. Hinsvegar fer það eftir stærð raunminnisins og minniskrafa forritana hversu mörg forrit má hlaða inn.

20.5. HVERSU MARGAR FÆRSLUR (*ENTRIES*) HEFUR SÍÐUTAFLA KERFISINS?

Færslurnar í síðutöflu samsvara skiptingu á rökivistföngum, þar sem þetta kerfi hefur skiptir sínum rökivistföngum niður á 4 síður er greinilegt að það hafi aðeins fjórar færslur í síðutöflunni sinni.

20.6. HVERSU MARGA BITA ÞARF TIL AÐ TÁKNA LÍNU Í SÍÐUTÖFLU?

Hver færsla vísar í ramma og segir til hvort hann sé *valid* eða ekki, við höfum 16 ramma sem er táknað í binary með 4 tölum þannig við þurfum þá fjóra bita fyrir ramma vistfang + 1 bita fyrir valid, við þurfum því 5 bita í heildina.

20.7. HVAÐ ER RAUNVISTFANG RÖKVISTFANGSINS 42?

Byrjum á að finna síðunúmer og offset: Við finnum síðunúmer með því að deila rökivistfanginu með síðustærð og námunda niður, síðunúmerið verður því $\lfloor \frac{42}{64} \rfloor = 0$, gildið á fyrst staki í síðutöflu[0], við finnum offsetið á svipaðan máta, nema núna moddum við rökivistfangið við síðustærðina og fáum $42 \% 64 = 42$

Raunvistfangið verður því $0 * 64 + 42 = 42$

20.8. HVAÐ ER RAUNVISTFANG RÖKVISTFANGSINS 191?

Gögnum í gegnum sömu skref og fyrir ofan Síðunúmer: $\lfloor \frac{191}{64} \rfloor = 2$, gildið á staki [2] í síðutöflu er 10, offset: $191 \% 64 = 63$

Raunvistfang: $10 * 64 + 63 = 703$

21. ÚTHLUTUNARREIKNIRIT

Verkefni: Við fáum í hendurnar töflu sem sýnir les/skrifa köll á síður

Reference number	1.	2.	3.	4.	5.	6.	7.	8.
Page reference	0	1	3	1	2	0	1	2
Read or Write	R	R	W	W	R	R	R	W

Notið síðan tilheyrandi reiknirit til þess að fylla inn í töflur með þessum upplýsingur

21.1. FIFO

Hér er aðgangi stjórnað með biðröð. Bendirinn bendir á það sem kom fyrst inn, ef það kemur inn reference sem er ekki í töflunni nú þegar er því reference skipt út fyrir það sem er undir bendinum og hann síðan færður einn niður, ef hann er á botni færast hann efst upp.

FIFO (using either FIFO-queue or circular buffer)																
Reference number		1		2		3		4		5		6		7		8
Page reference string		0		1		3		1		2		0		1		2
Read/Write access		R		R		W		W		R		R		R		W
Frame 0	->	0	->	0	->	0	->	0		2		2	->	2	->	2
Frame 1				1		1		1	->	1		0		0		0
Frame 2						3		3		3	->	3		1		1
Number of page faults:		6														

21.2. SECOND-CHANCE / CLOCK

Hvert stak fær R bita þegar þau komast inn í röðina. Bendirinn færast ekki nema á page faulti. Þegar það kemur page fault ítrar bendirinn yfir öll stök og tekur burt R bitann, ef það finnur stak sem hefur ekki R bita, þá er því staki skipt út.

Second-Chance/Clock																			
Reference number		1		2		3		4		5		6		7		8			
Page reference string		0		1		3		1		2		0		1		2			
Read/Write access		R		R		W		W		R		R		R		W			
Frame 0	->	0 R		0 R		0 R		->	0 R		0		0		0		0		
Frame 1				->	1 R		1 R		1 R		->	2 R		->	2 R		2 R		
Frame 2						->	3 R		3 R		3		3		->	1 R		->	1 R
Number of page faults:		6																	

21.3. ENHANCED SECOND-CHANCE (NRU)

Nú geta stök haft fjögur "state" 1 : (,), 2 : (, M), 3 : (R,), 4 : (R, M), í þessu dæmi eru reference endurstíllt fyrir fjórða og sjöunda. Bendirinn færast bara niður með page faulti og ítrar yfir stökin, lækkandi þau um eitt state í hverri yfirfærslu. Þegar það kemur að staki í state 1 eða nær að lækka niður í state 1 þá er því staki skipt út.

Enhanced Second-Chance (NRU)																		
Reference number		1		2		3		4		5		6		7		8		
Page reference string		0		1		3		1		2		0		1		2		
Read/Write access		R		R		W		W		R		R		R		W		
Frame 0	->	0 R		0 R		0 R		0		->	2 R		2 R		2		2 M	
Frame 1				->	1 R		1 R		1 RM		1 RM		->	1 RM		->	1 RM	
Frame 2						->	3 RM		->	3 M		3 M		0 R		0		0
Number of page faults:		5																
Note: A periodic reset of the																		

Note: A periodic reset of the

22. PAGING Í PRAKTÍK

Verkefni:

Við fáum gefið kerfi sem notar demand pagint til að úthluta sýndarminni, síður og/eða rammar eru 1KB, stærð raunskyndiminnisins er bara 1 rammi.

Forritið notar LRU til að skipta síðum og keyrir eftirfarandi C-kóða

```
int i,j;
int data[128][128];

for (j = 0; j < 128; j++) {
    for (i = 0; i < 128; i++) {
        data[i][j] = 0;
    }
}
```

Gerið ráð fyrir því að breytur i og j séu geymdar í CPU registrum. Hver `int` hlutur tekur 4 byte af minni

22.1. HVERSU MARGAR SÍÐUR ERU NÝTTAR UNDIR `data` Í SÝNDARMINNI?

Við erum að geyma $4 \cdot 128 = 512$ bæti af gögnum inn í hverju innra fylki, það þýðir að hver rammi getur haldið tveimur stökum úr ytra fylkinu, þ.e. $\frac{128}{2} = 64$ síður

22.2. HVERSU MÖRGUM PAGE FAULTS LENDUM VIÐ Í UPPHAFLEGA?

Upphaflega inniheldur fylkið engin gögn og eins og uppsetningin á kóðanum er núna þá ítrum við yfir ytri stökin áður en við ítrum yfir innri stökin. Það þýðir að fyrir 128 innri stök lendum við í $128 \cdot 64 = 1044$ page faultum. Þetta er vegna þess að við þurfum að skipta um síðu fyrir annað hvert ytra stak.

22.3. HVERNIG MÁ MINNKA FJÖLDA PAGE-FAULTA?

Það er frekar einfalt, við getum breytt þessu þannig að við fáum bara page fault fyrir hvert 256-tugasta stak. Við gerum það með því að víxla á i og j , annaðhvort í for-lykkjunum eða í fylkja aðgenginu en ekki báðum.

```
int i,j;
int data[128][128];

for (i = 0; i < 128; i++) {
    for (j = 0; j < 128; j++) {
        data[i][j] = 0;
    }
}
```

23. FAT SYSTEMS

Verkefni: Við fáum í hendurnar þrjú skráarkerfi sem nota öll FAT, fyrsta hefur bara 10KB clustera, annað 1KB clustera og þriðja 0.5KB clustera.

23.1. UPPSKIPTING SKRÁA

Verkefni: Nú viljum við setja 1000 0.5KB skrár, 100 5KB skrár og 10 50KB skrár inn í hvert kerfi fyrir sig. Hversu mikið pláss í hverju kerfi í KB er ekki hægt að nýta vegna internal fragmentation

Allir þessara skráahópa eru 500KB í heild, ef við byrjum að reikna plássið sem tapast í 10KB skráarkerfinu þá sjáum við að heildartap verður 10000KB:

- $1000 \cdot 0.5\text{KB}$ - hver cluster skilur eftir 9.5KB fyrir hverja skrá, tapað pláss verður 9500KB
- $100 \cdot 5\text{KB}$ - hver cluster skilur eftir 5KB fyrir hverja skrá, tapað pláss verður 500KB
- $10 \cdot 50\text{KB}$ - ekkert tap

Í 1KB cluster kerfinu tapast í heildina $1000 \cdot 0.5\text{KB} = 500\text{KB}$

Í 0.5KB cluster kerfi tapast ekkert

23.2. KEMST EFTIRFARANDI FYRIR?

Verkefni: Nú höfum við hámarka á FAT kerfunum 1500KB, er hægt að nýta kerfin fyrir ofan til þess að geyma allar skrárnar í dæminu fyrir ofan?

23.2.A. FAT KERFIÐ ER EKKI GEYMT Í CLUSTERUNUM

Það væri hægt að geyma skrárnar í 0.5KB clusterunum þar sem þar er ekkert tap.

23.2.B. EF FAT KERFIÐ ER GEYMT Í CLUSTERUM, HVERSU MARGA CLUSTERA TEKUR ÞAÐ?

Við erum að notast við FAT16 sem er 16 bitar að stærð, þ.e. 2 byte

- Í 10KB kerfinu höfum við 150 klustera, $\frac{1500\text{KB}}{10\text{KB}} = 150$, þá þurfum við $150 \cdot 2B = 300B$ af minni fyrir kerfið sjálft, það gera $\lceil \frac{300B}{10240B} \rceil = 1$ cluster
- 1KB kerfið hefur 1500 clusters og 3000B af minni undir FAT sem gera $\lceil \frac{3000B}{1024B} \rceil = 3$ clustera
- Kerfið með 0.5KB gerir 3000 klusters, 6000B fyrir FAT, $\lceil \frac{6000B}{512B} \rceil = 12$ clusters

24. I-NODE FILE SYSTEM

24.1. HÁMARKSSTÆRÐ KERFIS

Við höfum I-node kerfi sem styður að geyma 98 “*direct block*” benda, einn bendi á “*single indirect block*” og einn bendi á “*double indirect block*”. Hvert “*indirect block*” getur geymt 100 *block* benda. Hvert block er 1KB að stærð. Hvað er hámarksstærð skráa sem geyma má í kerfinu?

$$\text{SIB} + \text{DIB} + \text{DB} = 100 \cdot 1_{\text{KB}} + 100^2 \cdot 1_{\text{KB}} + 98 \cdot 1_{\text{KB}} = 100_{\text{KB}} + 10000_{\text{KB}} + 98_{\text{KB}} = 10198_{\text{KB}}$$

24.2. LEIT AÐ FORRITI

Verkefni: Við fáum gefin “*absolute path*” að skránni `/usr/bin/vi`. Hvað þarf að sækja á disk til að athuga hvort þessi skrá sé yfirhöfuð til. Gerið ráð fyrir því að `/` (root) sé núþegar búið að hlaða inn í minni. Sýnið þau skref sem eru tekin.

Fyrst að rótin er í skyndiminninu getum við leitað þar að `usr` möppunni, við þurfum ekki “*disk access*” til þess að athuga þetta en við þurfum *DA* til þess að hlaða inn I-node `usr` möppunnar og síðan annað til þess að hlaða inn kubbum hnútsins. Þetta eru tvö *DA*. Við gerum svo eins til að leita að `bin` möppunni innan `/usr/` og fáum þá aftur tvö *DA*. Að lokum leitum við innan `/usr/bin/` að `vi` skránni. Til þess þurfum við ekki *DA*.

24.3. SKRÁARKERFIS UPPFÆRSLUR

Verkefni: Lýsið því sem gerist þegar eftirfarandi skipanir eru keyrðar

24.3.A. `link("ParentDirectory/existingFile", "ParentDirectory/newFile")`

Skipunin býr til nýja skrá sem er “hard link” á skrá sem nú þegar er til. Harður hlekkur er bendir á I-node fyrir skrá sem nú þegar er til. Skrefin sem eru tekin:

- Ný skrá búin til inn í “ParentDirectory” I-node
- Block/s teknir frá innan “ParentDirectory” sem geyma vísun í:
 - “existingFile” I-node
 - Lengd skráarnafnsins
 - Skráarnafnið sjálft
- #links er hækkað um 1 í “existingFile” metadata

24.3.B. `symlink("ParentDirectory/existingFile", "ParentDirectory/newFile")`

Symlink er skrá sem geymir slóð á aðra skrá, skrefin sem við tökum eru færri. Við þurfum að taka frá block/s til að gera ráð fyrir nýju I-node og slóðinni ásamt því að bæta nýrri færslu inn í “ParentDirectory” I-nodeið.