IT University of Copenhagen

IT-Universitet i København

Operativsystemer og C

Obligatorisk aflevering 3

Virtuel hukommelse

Holger Borum (thgn@itu.dk)
Tróndur Høgnason (mfrm@itu.dk)
Frederik Madsen (hstb@itu.dk)

November 19, 2015

Contents

1 Opgave beskrivelse

I denne opgave bliver der implementeret en simpel virtuel hukommelse. Den virtuelle hukommelse skal kunne håndtere demand paging. Løsningen baserer sig på udleveret kode.

Den udleverede kode består af en virtuel side tabel, en virtuel disk og en ufuldstændig main metode. Det der mangler i main metoden er en måde at håndtere page faults på, og en måde at parse valget af sideskiftnings algoritme.

Main metoden tager fire parametre som input

- Antal virtuelle sider (pages)
- Antal fysiske sider (frames)
- Hvilken sideskiftning algoritme der skal bruges (rand|fifo|custom)
- Hvilken simuleret programkørsel der skal udføres(focus|sort|scan)

Vores opgaver er således at:

- Implementere en page fault handler, der skifter fysiske sider til ind og ud i den fysiske hukommelse, når sidetabellen er fuld.
- Implementere tre sideskiftnings algoritmer:

  fifo - Algoritmen skal virke som en FIFO kø, hvor den side der er kommet i brug først, er den første der skiftes ud.

  rand - Ikke en engentlig algoritme, da den bare skal vælge en vilkårlig side, der skal skiftes ud.

  custom - Vores egne algoritme, herefter Custom, der skal lave færre disk tilgange end de to andre algoritmer.

- Teste vores sideudskiftningsalgoritmer
- Ændre main metoden, så den parser tekststrengen fra kommandolinjen, så man kan vælge hvilken sideudskiftningsalgoritme, programmet skal bruge.
- Lave noget statistik over hvor mange disktilgange hver algoritme laver, så vi kan se sammenligne vores custom algoritme med de to andre.

2 Design

2.1 Page fault handler

Der er lavet to forskellige page fault handlers, da vores custom-algoritme, en LRU-tilnærmelse, kræver, at nogle datastrukturer bliver oprettet og opdateret. Kørslen af Custom og de to andre algoritmer fraviger ikke meget fra hinanden, men LRU kræver, at der opretholdes en datastruktur, som der ikke er behov for i de andre sideudskiftningsalgoritmer. Dette var ikke ønskværdigt, og der blev derfor oprettet to forskeliige handlere. Der er truffet et designvalg om, at den kodeduplikation, der nu må optræde, bliver opvejet af simplere kode der er tale om når datastrukturen ikke indgår.

2.2 Sideudskifningsalgoritmer

Herunder findes en kort beskrivelse af hver algoritmes design. Det skal nævnes, at det er gældende for dem alle, at alle frames først bliver mappet til pages sekventielt. Sagt på en anden måde: Når der tilgåes en ny page, så bliver den først ledige frame sekventielt tildelt til den page. Det er først når der ikke er flere frames at nedenstående træder i kraft.

2.2.1 Random

Random er, som navnet antyder, en algoritme, der vælger en tilfældig frame ud til page-swappet. Her er det, i sagens natur, svært at give et kvalificeret bud på et optimalt kørselsforløb. Dog kan det siges at det som regel vil fungere bedre end værste tilfældet hos de fleste algoritmer, da algoritmen ikke har en systematisk svaghed. Dermed ikke sagt at Random ikke kan ende i værste tilfældet.

2.2.2 FIFO

Fist-in-first-out algoritmen fungerer ved, at den frame, der er givet i den første frame-anmodning, også er den frame, der bliver frigivet først. Der er konceptuelt tale om en kø-struktur, hvor hver page-til-frame mapping lægges bagerst i køen når, den bliver oprettet. Således skal alle frames mappes til andre pages, før den første page-til-frame mapping bliver fjernet, og en frame, der allerede har været brugt, kan gives til en ny page. Dette betyder, at alle page-til-frame mappings får lov til at eksistere i det samme antal page-swaps, hvilket bør give færre disk tilgange for programmer, hvor det faktum at et data område lige er blevet brugt, er en indikation på, at det snart benyttes igen. Med andre ord, så virker algoritmen godt, hvis der er større sandsynlighed for at data der lige er blevet brugt, skal bruges igen, end at data der er blevet brugt for et stykke tid siden skal bruges igen.

2.2.3 Least Recently Used

LRU (også kaldet custom) er en algoritme der returnerer den page-til-frame-mapping der er brugt for længst tid siden indenfor et givet tidsinterval. Vi har fundet det optimale tidsinterval til være ~1.1 s (~35 ms * 32 bits). Da vi ikke har adgang til CPU-bits, er der tale om en tilnærmelse, og algoritmen kan således kun finde ud om en page er blevet brugt indenfor ~35 ms, men ikke hvor mange gange. Ideen bag algoritmen er, at pages der er blevet brugt meget indenfor de seneste par instruktioner, også vil blive brugt meget i de kommende par instruktioner. Således vil algoritmen give det optimale resultat i samme situation som FIFO-køen, dog blot bedre, da netop de meget brugte pages får lov at blive i hukommelsen, og ikke blot dem der lige er blevet tildelt. Således tages der også højde for at en side kan hentes ind, for så kun at blive brugt en enkelt gang.

2.2.4 Optimeret random

Efter at have analyseret resultaterne af at køre de udleverede programmer (se afsnit 5.1) med de tre ovenstående algoritmer, opdagede vi, at Random generelt havde betydeligt færre disktilgange. Så for at udnytte denne viden, er der blevet implementeret en fjerde algoritme, der vælger en 1/3 af alle frames ud tilfældigt, og, om muligt, finder en page-til-frame-mapping, der ikke har nogen skriverettigheder. Dette gøres med henblik på at skifte færre af de dyre mappings med skriverettigheder, og flere af de billige mapping med kun læserettigheder, ud. Det giver dog også en bias mod mappings med skriverettigheder, hvilket betyder, at flere og flere mappings vil have skriverettigheder, og mappings med læserettigheder hurtigere vil blive skiftet ud. Så for lange programkørsler vil algoritmen sandsynligvis ikke køre ligeså godt. [1]

[1] Det skal også nævnes, at denne algoritme er inspireret af en anden gruppes løsning, efter at vi i fællesskab havde diskuteret resultaterne af random, og vi derfor ikke kan tage hele æren for at være kommet på ideen.

3 Implementation

I dette afsnit vil der gås lidt mere i dybden med hvordan de nævnte løsninger er implementeret (se afsnit 2).

3.1 Page fault handler

Page handleren er den metode, der bliver kaldt, når der smides en page fault. Først bliver der checket, om page fault'en er en write request, hvilket gør sideudskiftning unødvendig.

Sidenhen checkes der, om der er nogen fri frame, ved at gennemløbe vores frametable (map). Her chekkes der om der er nogen frame, der aldrig har været brugt; at den i frametable er -1.[2]

Hvis der ikke er nogen frame, der ikke er i brug, kommer frameSelecter i brug. Den bruger en af sideudskiftningsalgoritmerne til at finde den optimale frame at skifte ud. Hvis pagen der tidligere var mappet til det pågældende frame har skrive-rettigheder, bliver dataet på framet skrevet til disken, og den pågældende frame sættes til at være fri. Dette sker fordi vi antager at enhver page der har fået skriverettigheder også rent faktisk har skrevet til framet, og således er data blevet ændret ift. til hvad der ligger på disken. Derefter opdateres vores frametable og den frie frame mappes den page, der har fremprovokeret en page fault.

Til sidst læses det ønskede data ind fra disken til den nu mappede frame.

3.2 Sideudskifningsalgoritmer

Der er implementeret tre sideudskiftningsalgoritmer. Deres implementation bliver kort beskrevet nedenfor.

3.2.1 Random

Rand algoritmen sætter bare pointeren til den frie frame til et vilkårligt tal fra 0-N, hvor N er tallet af frames. Funktionen srand48 bliver brugt til seeding og lrand48 bliver brugt til at generere tal.

3.2.2 FIFO

FIFO algoritmen virker som en FIFO kø. Hvis ingen frame er tilgængelig og en frame skal skiftes ud, så skifter algoritmen den tidligst mappede frame ud med en ny, og flytter framen bagerst i køen.

Det gøres i praksis ved at opretholde datastrukturen FIFOData, der rent faktisk kun består af et heltal. Det heltal bliver inkrementeret hver gang der er brug for at loade nyt data ind på en frame indtil vi rammer tallet nframes,

[2]Her kunne en optimering være ikke at gennemløbe map'et, efter alle frames er blevet tildelt første gang, eftersom de aldrig vil blive frie igen før programmets afslutning

antallet af frames, så starter vi fra 0, den første side, igen ved hjælp af modulo operatoren.

Implementationen virker som beskrevet, under den antagelse at sider aldrig frigives og at frie frames uddeles i rækkefølge. Og da alle frie frames først bliver tildelt sekventielt og aldrig frigivet, må antagelsen holde vand.

3.2.3 Least Recently Used

Custom algoritmen er en version af en tilnærmet "Least recently used"-algoritme (LRU). For at kunne få et billede af hvornår en page sidst har læst eller skrevet til hukommelsen, så fjernes alle pages's skriverettigheder med et fast interval (LRUTIME). Alle pages, der nu enten vil skrive eller læse fra hukommelsen, vil nu blive fanget i page_fault_handler(). Her registreres det, at pagen er blevet tilgået indenfor denne periode. For at undgå at disse fremprovokerede faults ikke ender i flere disk tilgange, holdes der styr på, om en pågældende page allerede er mappet til den fysiske hukomelse, og hvilke rettigheder den i så fald havde før rettighederne blev fjernet.

Hver page i den virtuelle hukommelse bliver tildelt to heltal, som bruges til at holde styr på, hvornår pagen sidst er blevet tilgået, dens historie og hvilke rettigheder pagen havde sidst, den havde adgang til den fysiske hukommelse. Historikken bliver brugt som beskrevet i "Operating System Concepts" på page 410. Hvis flere pages har samme laveste historik, så vælges den som er koblet til den laveste frame. Det betyder, at en fifo-struktur ved samme historik ikke er garanteret. Det kunne være løst ved at bruge en hægtet liste som historik.

3.2.4 Optimeret random

Den optimerede random algoritme er implementeret ved en simpel for-løkke, som køres nframes/3 gange. Antallet nframes/3 er fundet ved flere gennemkørsler for at finde et nogenlunde optimalt antal. I for-løkken vælges en tilfældig frame, og så tjekkes om den mappede page har skrive-rettigheder. Hvis den har skriverettigheder, prøves igen. Hvis den kun har læserettigheder, så vælges den frame til at blive frigivet. I tilfælde af at alle pages har skrive rettigheder, vælges blot den sidste tilfældigt valgte frame. Som ses i graferne i afsnit 5.1, kører den optimerede random væsentligt bedre end både FIFO, random og LRU algoritmerne. Det kan skyldes en blanding af 3 ting i de programmer som der er testet med:

1. De har en overvægt af variabler som der kun læses fra få gange, end de har variabler som der kun skrives til få gange

2. De har en overvægt af variabler som der skrives til mange gange, end de har variabler som der læses fra mange gange

3. De er forholdsvis korte, således at den omtalte overvægt af pages med skriverettigheder 2 ikke når at blive en realitet

4 Test

Den virtuelle hukommelse er blevet testet på to måder, som burde være bevis for, at i hvert fald størstedelen af programmet fungerer korrekt.

For det første er de forskellige programkørsler blevet kørt uden nogen sideudskiftning, og resultatet af disse kørsler er blevet noteret for et par værdier. Efter en sideudskiftningsalgoritme er blevet implementeret, er det blevet tjekket at programkørslerne, nu med sideudskiftninger, giver samme resultat. Når dette virker, er det et tegn på, at selve sideudskiftningen virker uden at overskrive data i brug.

For det andet er de forskellige sideudskiftningsalgoritmer blevet testet med simple unit-tests. Her er der kun tale om FIFO og Custom (LRU), da det ikke gav mening at skrive unit test for Random - den returnerer tilfældige kun værdier. Testene er små, men nok til at overbevise os selv om, at sider der skal udskiftes vælges korrekt. Test koden findes i Appendix 6.6

Tilsammen er de to test nok til at overbevise os om, at den virtuelle hukommelse virker korrekt i de fleste henseender. Dog er der noget opdatering og vedligeholdelse af data i Custom implementationen, som ikke direkte er testet.

5 Diskusion

Vi har implementeret en virtuel hukommelse, der har de tre foreslåede sideudskiftningsalgoritmer random, fifo og custom til rådighed. Derudover har vi implementeret en fjerde algoritme, som vi kalder Optimeret random, der laver færre disktilgange end de andre tre.

Der er lavet statistik over hvor ofte disken tilgås, når sideudskiftningsalgoritmerne kører de tre testprogrammer, der alle har vidt forskellige hukommelseadgangsmønstre.

5.1 Statistik for sideudskiftningsalgoritmer

De tre simulerede programkørsler rand, scan og focus er blevet kørt 100 gange med de tre forskellige sideskiftningsalgoritmer. Der er samlet statistik over hvor mange disktilgange de forskellige algoritmer har med et skiftende antal fysiske sider. Programkørslerne resulterede i meget forskellig opførsel i antallet af disk tilgange. I dette afsnit beskrives diskuteres de forskellige resultater, alle kørslerne i en graf kan ses i figur 1. Vi vil udelukkende forholde os til disktilgange, selvom det også er relevant at se på hvilket overhead en algoritme tilføjer systemet.[3]

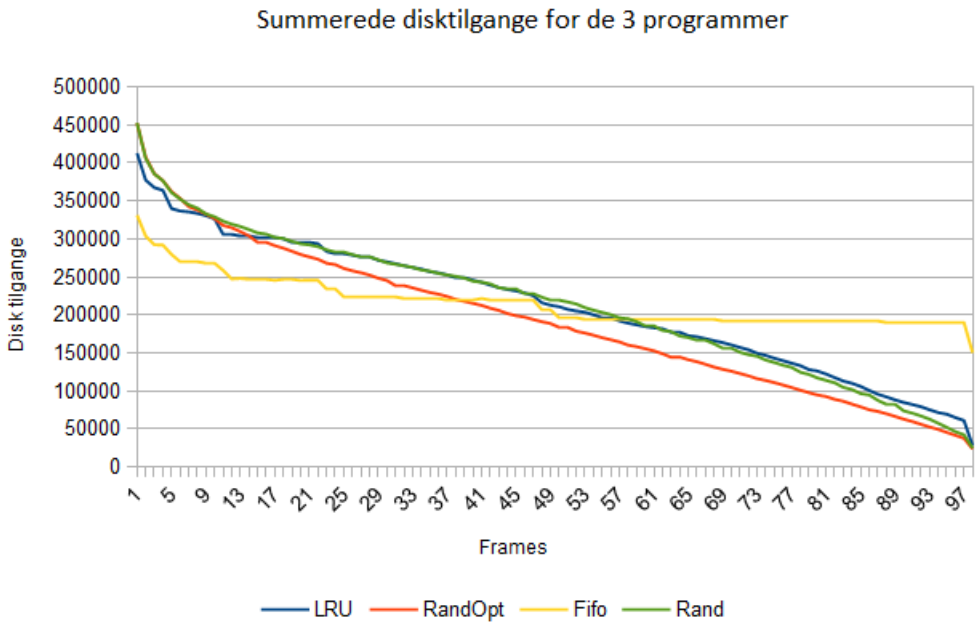

Figure 1: Summeret kørsler for alle programmer med 100 sider

[3]Koden der er brugt til at lave statistik er i appendix i afsnit 6.7

5.1.1 Sekventielle løb

Programmet "Scan" består af 10 sekventielle løb gennem alt data. Det kan hverken FIFO eller LRU håndtere særlig godt, da begge algoritmer bygger på den antagelse, at der er størst sandsynlighed for, at data, der lige er blevet brugt, snart skal bruges igen. Ved sekventielle gennemløb er det det omvendte, der er tilfældet. Dette viste sig også at være tilfældet for FIFO- LRU klarede sig derimod bedre end forventet, hvilket skyldes, at LRU implementation ikke nødvendigvis benytter sig af FIFO i tilfælde af, at to sider har den samme historik (Se. 3.2.3).

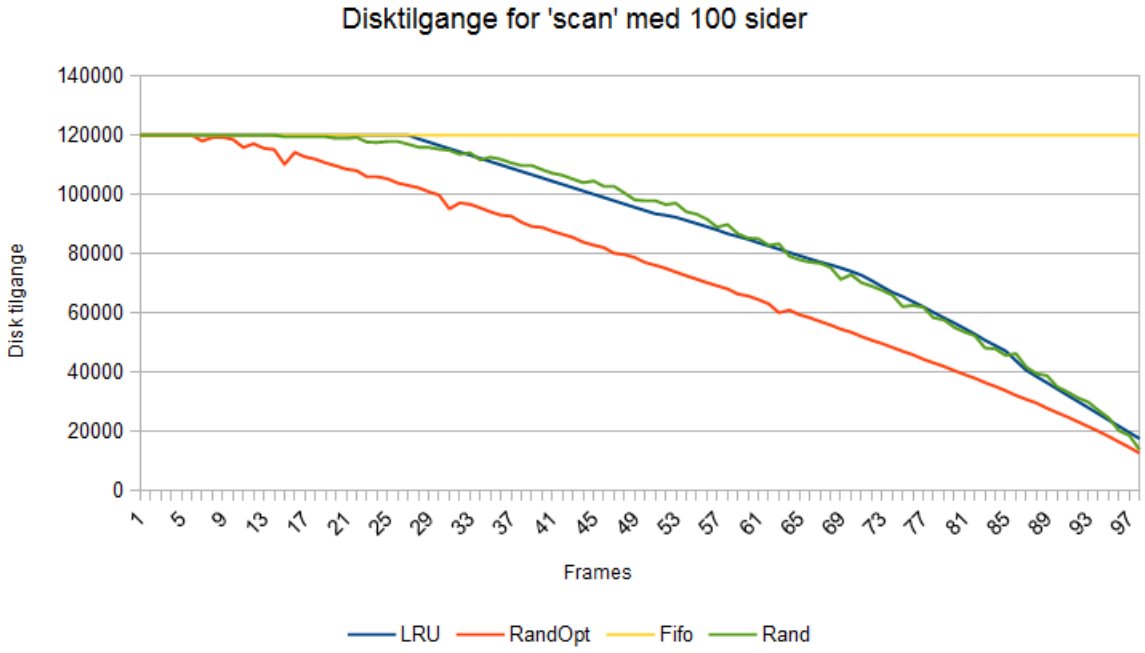

Figure 2: 100 Scan kørsler for hvert antal frames summeret

5.1.2 Anden programkørsel

Selvom det ikke er unormalt for programmer at have sekventielle gennemløb af data, så er det ikke nødvendigvis normal programkørsel. Programmerne "Focus" og "Sort" har en anden hukommelsesopførsel, der stadigvæk har sekventielle løb, men indenfor mindre dataområder[4]. Med disse adgangsmønstre fungerer både FIFO og LRU væsentligt bedre som set i nedenstående figurer. Dog virker de to

[4]Ingen af programmerne har dog et mønster, der minder om det, der ses i "Operating System Concepts", side 420

algoritmer, der basere sig på tilfældighed stadigvæk bedre når 50%+ af siderne er dækket af frames.

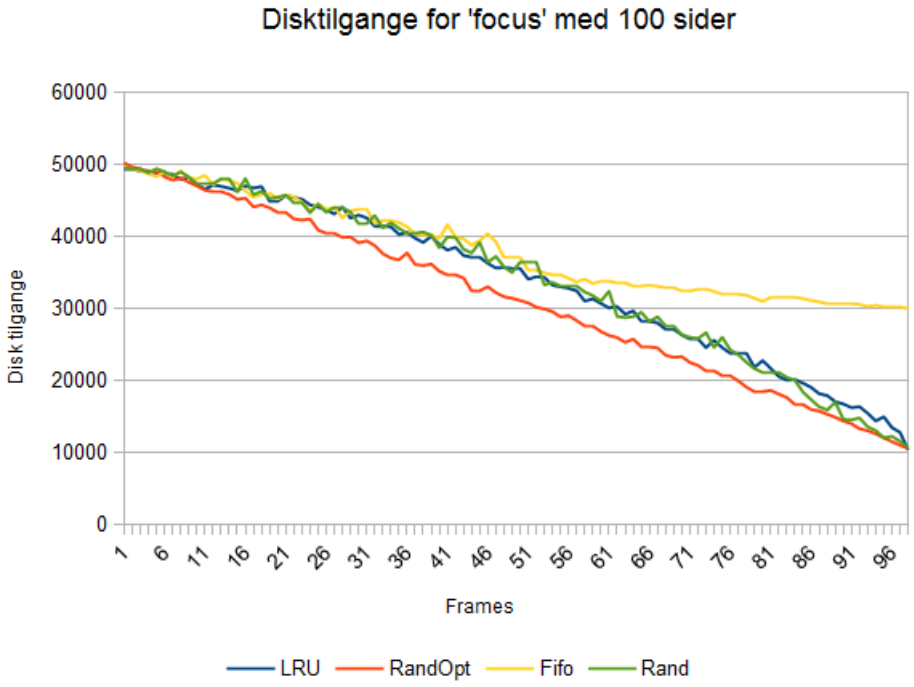

Figure 3: 100 focus kørsler for hvert antal frames summeret

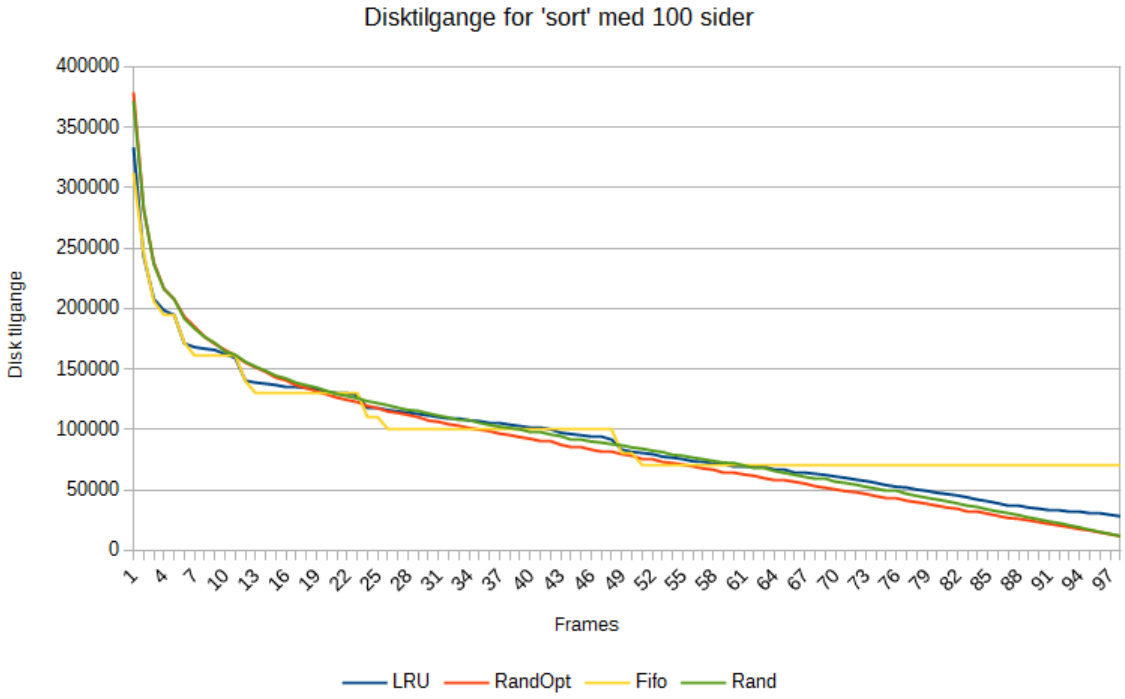

Figure 4: 100 sort kørsler for hvert antal frames summeret

5.2 Konklusion

Umiddelbart lader det til, at algoritmerne, der er baseret på tilfældighed, resulterer i færre disktilgange. Der er dog to forbehold, som gør resultatet usikkert. For det første ser det ud til at FIFO og LRU klarer sig godt, når der er få frames i forhold til sider. For det andet mener vi, at en algoritme som Optimeret Random kan få problemer ved lange programkørsler, fordi at det vil ende med, at der ville være få pages med læserettigheder, som hurtigt ville blive skiftet ud igen. Det vil givetvis føre til et øget antal sideudskiftninger, som vi måske ikke har observeret. Grundlæggende må vi konkludere, at det er de konkrete programkørsler, der er bestemmende for, hvornår en given algoritme er god eller dårlig.

6 Appendix

6.1 Makefile

```
virtmem: main.o page_table.o disk.o program.o frameSelecter.o
  gcc main.o page_table.o disk.o program.o frameSelecter.o -o
      virtmem

main.o: main.c
  gcc -w -Wall -g -c main.c -o main.o

frameSelecter.o: frameSelecter.c frameSelecter.h
  gcc -w -Wall -g -c frameSelecter.c -o frameSelecter.o

disk.o: disk.c
  gcc -w -Wall -g -c disk.c -o disk.o

program.o: program.c
  gcc -w -Wall -g -c program.c -o program.o

test: page_table.o disk.o frameSelecter.o test.o
  gcc test.o page_table.o disk.o frameSelecter.o -o test

test.o: test.c
  gcc -w -Wall -g -c test.c -o test.o

clean:
  rm -f *.o virtmem
```

6.2 main.c

```c
/*
Main program for the virtual memory project.
Make all of your modifications to this file.
You may add or rearrange any code or data as you need.
The header files page_table.h and disk.h explain
how to use the page table and disk interfaces.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>

#include "page_table.h"
#include "disk.h"
#include "program.h"
#include "frameSelecter.h"

#define LRUTIME 40

//ressources
struct frame_table *ft;
struct disk *disk;
struct LRUData *LRUData;
char *physmem;


//statisics
int diskWrites = 0, diskReads = 0, pageReq = 0, writeReq = 0,
    LRUFaults = 0;

//frameselection:
void* fsData;

void (*frameSelecter)(struct frame_table*, int, int, int*,  void*);

char *int2bit(int a, char *buffer, int buf_size) {
    buffer += (buf_size - 1);
    int i;
    for (i = 31; i >= 0; i--) {
        *buffer-- = (a & 1) + '0';

        a >>= 1;
    }

    return buffer;
}

void print_mapping(struct page_table *pt){
  int npages = page_table_get_npages(pt);
  int nframes = page_table_get_nframes(pt);

  // 1. Find a free frame
  // a. If there is a free frame
```

```
55    int p, frame, bits;
56    printf("P - F - B - Pb\n");
57    for(p = 0; p < npages; p++){
58      page_table_get_entry(pt, p, &frame, &bits);
59
60      printf("%d - %d - %d - %d\n", p, frame, bits, LRUData->
            page_bits[p]);
61    }
62    printf("Frame: \n" );
63
64    for(p = 0; p < nframes; p++){
65      printf("%d\n", ft->map[p]);
66    }
67  }
68
69  int findFreeFrame(struct page_table *pt, int* retFrame){
70    int f, nframes = page_table_get_nframes(pt);
71
72    for(f = 0; f < nframes; f++){
73      if(ft->map[f] == -1){
74        *retFrame = f;
75        return 1;
76      }
77    }
78    return 0;
79  }
80
81  void standard_page_fault_handler( struct page_table * pt, int page
      ){
82    pageReq++; //Statistik
83
84    int bits, frame, history;
85    page_table_get_entry(pt, page, &frame, &bits );
86
87    int npages, nframes;
88    npages = page_table_get_npages(pt);
89    nframes = page_table_get_nframes(pt);
90
91
92    //Check if this is a "write-request"
93    if((bits & PROT_READ) == PROT_READ){
94      page_table_set_entry(pt, page, frame, PROT_READ | PROT_WRITE );
95      writeReq++;
96      return;
97    }
98
99    // 1. Find a frame
100   //  a. If there is a free frame
101   int freeFrame;
102
103   //  b. If there is no free frame
104   //    b1. Use a page-replacement algorithm to select a victim
          frame
105   if(!findFreeFrame(pt, &freeFrame)){
106     frameSelecter(ft, npages, nframes,&freeFrame, fsData);
107
108     int tempFrame, bits, oldPage = ft->map[freeFrame];
```

```
109     page_table_get_entry(pt, oldPage, &tempFrame, &bits);
110
111     //  b2. Write the victim frame to the diske; update the page
            and frame tables accordingly
112     if((bits & PROT_WRITE) == PROT_WRITE ){
113       disk_write(disk, oldPage, &physmem[freeFrame * PAGE_SIZE]);
114       diskWrites++;
115     }
116     page_table_set_entry(pt, oldPage, 0, 0);
117   }
118
119   // 2. Read the desired page into the selected frame; change the
        page and frame tables.
120   ft->map[freeFrame] = page;
121
122   disk_read(disk, page, &physmem[freeFrame * PAGE_SIZE]);
123   page_table_set_entry(pt, page, freeFrame, PROT_READ);
124     diskReads++;
125
126   // 3. Continue the user process
127 }
128
129 void LRU_page_fault_handler( struct page_table * pt, int page ){
130   pageReq++; //Statistik
131
132   int bits, frame, history;
133   page_table_get_entry(pt, page, &frame, &bits );
134
135
136   int npages, nframes;
137   npages = page_table_get_npages(pt);
138   nframes = page_table_get_nframes(pt);
139
140     //saetter alle skriveretigheder til 0 og bitshifter alt
          page_history
141   int p, tempbits, tempFrame;
142   double c = clock();
143   if(c - LRUData->timestamp > LRUTIME){
144     for(p = 0; p < npages; p++){
145       LRUData->page_history[p] = LRUData->page_history[p]>>1;
146       page_table_get_entry(pt, p, &tempFrame, &tempbits);
147       LRUData->page_bits[p] = (LRUData->page_bits[p] > tempbits) ?
              LRUData->page_bits[p] : tempbits;
148       page_table_set_entry(pt, p, tempFrame, 0);
149     }
150     LRUData->timestamp = c;
151   }
152
153   //Markere at denne page er blevet efterspurgt i perioden, ved at
        saette leftmost bit til 1
154   LRUData->page_history[page] = LRUData->page_history[page] | (0
        x8000000);
155
156   //Checking if this request is caused by a LRU reset
157   if(bits == 0 && LRUData->page_bits[page] > 0){
158     page_table_set_entry(pt, page, frame, LRUData->page_bits[page])
          ;
```

```
159     //LRUData->page_bits[page] = 0;
160     LRUFaults++;
161     return;
162   }
163
164   //Check if this is a "write-request"
165   if((bits & PROT_READ) == PROT_READ){
166     page_table_set_entry(pt, page, frame, PROT_READ | PROT_WRITE );
167     writeReq++;
168     return;
169   }
170
171   // 1. Find a frame
172   //  a. If there is a free frame
173   int freeFrame;
174
175   //  b. If there is no free frame
176   //    b1. Use a page-replacement algorithm to select a victim
          frame
177   if(!findFreeFrame(pt, &freeFrame)){
178     frameSelecter(ft, npages, nframes,&freeFrame, fsData);
179
180     int bits, oldPage = ft->map[freeFrame];
181     page_table_get_entry(pt, oldPage, &tempFrame, &bits);
182
183     // b2. Write the victim frame to the diske; update the page and
            frame tables accordingly
184     if((bits & PROT_WRITE) == PROT_WRITE ||
185        (LRUData->page_bits[oldPage] & PROT_WRITE) == PROT_WRITE ){
186
187       disk_write(disk, oldPage, &physmem[freeFrame * PAGE_SIZE]);
188       diskWrites++;
189     }
190     page_table_set_entry(pt, oldPage, 0, 0);
191     LRUData->page_bits[oldPage] = 0;
192   }
193
194   // 2. Read the desired page into the selected frame; change the
        page and frame tables.
195   ft->map[freeFrame] = page;
196   disk_read(disk, page, &physmem[freeFrame * PAGE_SIZE]);
197   page_table_set_entry(pt, page, freeFrame, PROT_READ);
198
199     diskReads++;
200   // 3. Continue the user process
201 }
202
203 int main( int argc, char *argv[] )
204 {
205   srand48(time(NULL));
206   int freeLRU = 0;
207   if(argc!=5) {
208     printf("use: virtmem <npages> <nframes> <rand|fifo|custom> <
          sort|scan|focus>\n");
209     return 1;
210   }
211
```

```
212   int npages = atoi(argv[1]);
213   int nframes = atoi(argv[2]);
214   const char *algorithm = argv[3];
215   const char *program = argv[4];
216
217   //Initialising Frame table
218   if(!(ft = createFrameTable(nframes))){
219     printf("Frame table couldn't be allocated\n");
220     return 1;
221   }
222   int f;
223   for(f = 0; f < nframes; f++){
224     ft->map[f] = -1;
225   }
226   struct page_table* pt;
227
228   if (!strcmp(algorithm,"custom")){
229     //printf("%s\n", "Custom algorithm - LRU:");
230
231     //Initialising LRUData
232     if(! (LRUData = createLRUData(npages))){
233       printf("LRUData couldn't be allocated\n");
234       return 1;
235     }
236     LRUData->timestamp = clock();
237
238
239     //setting frameselecter
240     frameSelecter = getCustom();
241     fsData = LRUData;
242
243     //setting pagetable
244     pt = page_table_create( npages, nframes, LRU_page_fault_handler
           );
245
246     //eventually we should free LRU:
247     freeLRU = 1;
248   }
249   else if (!strcmp(algorithm,"fifo")){
250     struct FIFOData* fifdat = malloc(sizeof(struct FIFOData));
251
252     fifdat->nextFrame = 0;
253     fsData = (void*)fifdat;
254     //printf("%s\n", "Fifo algorithm:");
255     frameSelecter = getFifo();
256
257     //setting pagetable
258     pt = page_table_create( npages, nframes,
           standard_page_fault_handler );
259   }
260   else if (!strcmp(algorithm,"rand"))
261   {
262     //printf("%s\n", "Random algorithm:");
263     frameSelecter = getRand();
264
265     //setting pagetable
```

```
266       pt = page_table_create( npages, nframes,
              standard_page_fault_handler );
267     }else if (!strcmp(algorithm,"randopt"))
268     {
269       //printf("%s\n", "Random algorithm:");
270       frameSelecter = getRandOpt();
271
272       //setting pagetable
273       pt = page_table_create( npages, nframes,
              standard_page_fault_handler );
274
275       //setting data
276       struct RANDData* dat = malloc(sizeof(struct RANDData));
277       dat->pt = pt;
278       fsData = (void*)dat;
279     }
280     else{
281       printf("Algorithms to choose from are rand|fifo|custom|randopt\
              n");
282       return 1;
283     }
284
285     disk = disk_open("myvirtualdisk",npages);
286
287     if(!disk) {
288       fprintf(stderr,"couldn't create virtual disk: %s\n",strerror(
              errno));
289       return 1;
290     }
291
292     if(!pt) {
293       fprintf(stderr,"couldn't create page table: %s\n",strerror(
              errno));
294       return 1;
295     }
296
297     char *virtmem = page_table_get_virtmem(pt);
298     physmem = page_table_get_physmem(pt);
299
300     if(!strcmp(program,"sort")) {
301 //      printf("sort:\n");
302       sort_program(virtmem,npages*PAGE_SIZE);
303
304     } else if(!strcmp(program,"scan")) {
305 //      printf("scan:\n");
306       scan_program(virtmem,npages*PAGE_SIZE);
307
308     } else if(!strcmp(program,"focus")) {
309 //      printf("focus:\n");
310       focus_program(virtmem,npages*PAGE_SIZE);
311
312     }else if(!strcmp(program,"test")){
313       test_program(virtmem,npages*PAGE_SIZE);
314
315     } else{
316       fprintf(stderr,"unknown program: %s\n",argv[3]);
317     }
```

```
318
319   //printf("PageRequests: %d\n", pageReq);
320   //printf("writeReq: %d\n", writeReq);
321  printf("%d;", diskWrites);
322  printf("%d\n", diskReads);
323  //printf("LRUFaults: %d\n", LRUFaults);
324
325    //freeing mem
326    free(ft->map);
327    free(ft);
328    if(freeLRU){
329      free(LRUData->page_history);
330      free(LRUData->page_bits);
331      free(LRUData);
332    }
333
334    page_table_delete(pt);
335    disk_close(disk);
336
337    return 0;
338  }
```

6.3 frameSelecter.h

```
#ifndef FRAME_SELECTER_H
#define FRAME_SELECTER_H
#include "page_table.h"

struct FIFOData {
  int nextFrame;
};

struct LRUData{
  unsigned int *page_history;
  int *page_bits;
  double timestamp;
};
struct RANDData{
  struct page_table* pt;
};


struct frame_table{
  int *map;
};

void (*getFifo())   (struct frame_table*, int, int, int*, void*);
void (*getRand())   (struct frame_table*, int, int, int*, void*);
void (*getRandOpt())(struct frame_table*, int, int, int*, void*);
void (*getCustom()) (struct frame_table*, int, int, int*, void*);
struct LRUData* createLRUData(int);
struct frame_table* createFrameTable(int);

#endif
```

6.4 frameSelecter.c

```
1  #include "frameSelecter.h"
2  #include <stdlib.h>
3  #include <time.h>
4  #include "page_table.h"
5
6  void frameSelectFifo(struct frame_table *ft, int npages, int
       nframes, int* freeFrame, void* data){
7    struct FIFOData* fifdat = data;
8    *freeFrame = fifdat->nextFrame;
9    fifdat->nextFrame = (*freeFrame + 1) % nframes;
10 }
11
12 void frameSelectRand(struct frame_table *ft, int npages, int
       nframes, int* freeFrame, void* data){
13   *freeFrame = lrand48() % nframes;
14 }
15
16 void frameSelectRandOpt(struct frame_table *ft, int npages, int
       nframes, int* freeFrame, void* data){
17   struct RANDData* randdat = data;
18   int frame, page, bits, i;
19   for(i = 0; i < (nframes+1 / 3); i++){
20     frame = lrand48() % nframes;
21     page = ft->map[frame];
22     page_table_get_entry(randdat->pt, page, &frame, &bits );
23     if((bits & PROT_WRITE) != PROT_WRITE){
24       *freeFrame = frame;
25       return;
26     }
27   }
28   *freeFrame = frame;
29
30 }
31
32 void frameSelectCust(struct frame_table *ft, int npages, int
       nframes, int* freeFrame, void* data){
33   struct LRUData* LRUData = data;
34   int f, frame, bits;
35
36   //Find den page der har den laveste history, af pages der er
         mappet til frames.
37   unsigned int min = 0xffffffff;
38   double c;
39   for(f = 0; f < nframes; f++){
40     int page = ft->map[f];
41     unsigned int hist = LRUData->page_history[page];
42     if(hist <= min){
43       *freeFrame = f;
44       min = hist;
45     }
46   }
47 }
48
49
50 struct LRUData* createLRUData(int pages){
```

```
51    struct LRUData *LRUData;
52
53    if (!(LRUData = malloc(sizeof (struct LRUData) ) ) ){
54      printf("Fejl\n");
55      return 0;
56    }
57
58    if (!(LRUData->page_history = malloc(sizeof (int) * pages) ) ){
59      printf("Fejl\n");
60      return 0;
61    }
62
63    if (!(LRUData->page_bits = malloc(sizeof (int) * pages) ) ){
64      printf("Fejl\n");
65      return 0;
66    }
67
68    return LRUData;
69  }
70
71  struct frame_table* createFrameTable(int frames){
72    struct frame_table* ft;
73
74    if(!(ft = malloc(sizeof (struct frame_table)))){
75      printf("Fejl\n");
76      return 0;
77    }
78    if(!(ft->map = malloc(sizeof (int) * frames))){
79      printf("Fejl\n");
80      return 0;
81    }
82
83    return ft;
84  }
85
86  void (*getFifo()) (struct frame_table*, int nframes, int npages,
        int*, void*){
87    return &frameSelectFifo;
88  }
89
90  void (*getRand()) (struct frame_table*, int nframes, int npages,
        int*, void*){
91    return &frameSelectRand;
92  }
93
94  void (*getRandOpt()) (struct frame_table*, int nframes, int npages,
         int*, void*){
95    return &frameSelectRandOpt;
96  }
97
98  void (*getCustom()) (struct frame_table*, int nframes, int npages,
        int*, void*){
99    return &frameSelectCust;
100 }
```

6.5 minunit.h

```
1 //this is copied from http://www.jera.com/techinfo/jtns/jtn002.html
2 /* file: minunit.h */
3 #define mu_assert(message, test) do { if (!(test)) return message;
      } while (0)
4 #define mu_run_test(test) do { char *message = test(); tests_run++;
       \
5                                 if (message) return message; }
                                      while (0)
6 extern int tests_run;
```

6.6 test.c

```
#include <stdio.h>
#include "minunit.h"
#include <stdlib.h>
#include <math.h>

#include "page_table.h"
#include "disk.h"
#include "frameSelecter.h"

#define FIFORUNS 200
#define FIFOFRAMES 10
#define FIFOPAGES 100

int tests_run = 0;
void (*frameSelecter)(struct frame_table*, int, int, int*, void*);

static char * test_FIFO() {
  int returnFrame;

  struct FIFOData* fifdat = malloc(sizeof(struct FIFOData));
  fifdat->nextFrame = 0;
  void* data = fifdat;

  //mock
  struct frame_table* mockFrameTable;

  int results[FIFORUNS];
  frameSelecter = getFifo();

  int i;
  for(i = 0; i < FIFORUNS; i++){
    frameSelecter( mockFrameTable, FIFOPAGES, FIFOFRAMES, &
        returnFrame, data);
    results[i] = returnFrame;
  }

  for(i = 0; i < FIFORUNS; i++){
    if(results[i] != (i%FIFOFRAMES))
      break;
  }
  //start of error message construction
  char error[44];
  char iAsChar[3];
  sprintf(iAsChar,"%d", i);
  char iModTenChar = (i%FIFOFRAMES) + '0';
  char resultChar = results[i] + '0';
  strcpy(error,"Error, expected result[");
  strncat(error, &iAsChar, 3);
  strcat(error, "] to be ");
  strncat(error, &iModTenChar, 1);
  strcat(error, " but was ");
  strncat(error, &resultChar, 1);
  //end of error message construction

  free(fifdat);
```

```
55
56    mu_assert(error, i == FIFORUNS);
57    return 0;
58  }
59
60  static char * test_custom() {
61    frameSelecter = getCustom();
62    struct LRUData* LRUData;
63    struct frame_table* ft;
64
65     int freeFrame;
66
67    LRUData = createLRUData(5);
68    ft = createFrameTable(3);
69
70
71    LRUData->page_history[0] = 0x8000000;
72    LRUData->page_history[1] = 0xf000000;
73    LRUData->page_history[2] = 0x4000000;
74    LRUData->page_history[3] = 0x2300000;
75    LRUData->page_history[4] = 0x5000000;
76
77    ft->map[0] = 0;
78    ft->map[1] = 1;
79    ft->map[2] = 2;
80
81    frameSelecter(ft, 5, 3, &freeFrame, (void*) LRUData );
82
83    mu_assert("Selected wrong simple", freeFrame == 2);
84
85    ft->map[0] = 0;
86    ft->map[1] = 4;
87    ft->map[2] = 3;
88
89    frameSelecter(ft, 5, 3, &freeFrame, (void*) LRUData );
90    mu_assert("Selected wrong crossed frame table", freeFrame == 2);
91
92    LRUData->page_history[0] = 0;
93    LRUData->page_history[1] = 0;
94    LRUData->page_history[2] = 0;
95    LRUData->page_history[3] = 0;
96    LRUData->page_history[4] = 0;
97    ft->map[0] = 3;
98    ft->map[2] = 2;
99
100   frameSelecter(ft, 5, 3, &freeFrame, (void*) LRUData );
101   mu_assert("Empty history error", freeFrame == 2);
102
103   LRUData->page_history[0] = 0xffffffff;
104   LRUData->page_history[1] = 0xffffffff;
105   LRUData->page_history[2] = 0xffffffff;
106   LRUData->page_history[3] = 0xffffffff;
107   LRUData->page_history[4] = 0xffffffff;
108
109   frameSelecter(ft, 5, 3, &freeFrame, (void*) LRUData );
110   mu_assert("Full history error", freeFrame == 2);
111
```

```
112   free(ft->map);
113   free(ft);
114
115   ft = createFrameTable(1);
116   ft->map[0] = 4;
117
118   frameSelecter(ft, 5, 1, &freeFrame, (void*) LRUData );
119   mu_assert("One frame memory", freeFrame == 0);
120
121   free(ft->map);
122   free(ft);
123   free(LRUData->page_history);
124   free(LRUData->page_bits);
125   free(LRUData);
126   return 0;
127 }
128
129 static char * all_tests() {
130   mu_run_test(test_FIFO);
131   mu_run_test(test_custom);
132   return 0;
133 }
134
135 int main(int argc, char **argv) {
136   char *result = all_tests();
137    if (result != 0) {
138        printf("%s\n", result);
139    }
140    else {
141        printf("ALL TESTS PASSED\n");
142    }
143    printf("Tests run: %d\n", tests_run);
144
145    return result != 0;
146 }
```

6.7 Statistik

6.7.1 run_all.sh

```
a=2
while [ $a -lt 100 ]
do
  d=0
  while [ $d -lt 100 ]
  do
    b=$(./virtmem 100 $a $1 $2)
    OLDIFS="$IFS"
    IFS=';'
    c=0
    for num in $b;
    do
      e[$c]=$((${e[$c]}+$num))
      c=`expr $c + 1`
    done
    IFS="$OLDIFS"
    d=`expr $d + 1`
  done
  printf '%s;' "${e[@]}"
  printf '\n'
  e[0]=0
  e[1]=0
  a=`expr $a + 1`
done
```

6.7.2 run_all.sh

```
echo "Custom sort"
./stat.sh custom sort > stat/custom_sort.txt
echo "Custom focus"
./stat.sh custom focus > stat/custom_focus.txt
echo "Custom scan"
./stat.sh custom scan > stat/custom_scan.txt
echo "FIFO sort"
./stat.sh fifo sort > stat/FIFO_sort.txt
echo "FIFO focus"
./stat.sh fifo focus > stat/FIFO_focus.txt
echo "FIFO scan"
./stat.sh fifo scan > stat/FIFO_scan.txt
echo "rand sort"
./stat.sh rand sort > stat/rand_sort.txt
echo "rand focus"
./stat.sh rand focus > stat/rand_focus.txt
echo "rand scan"
./stat.sh rand scan > stat/rand_scan.txt
echo "randopt sort"
./stat.sh randopt sort > stat/randopt_sort.txt
echo "randopt focus"
./stat.sh randopt focus > stat/randopt_focus.txt
echo "randopt scan"
./stat.sh randopt scan > stat/randopt_scan.txt
```