IT University of Copenhagen

Operativsystemer og C

BOSC

Obligatorisk Opgave 3

Author:
Omar Khan (omsh@itu.dk)
Mads Ljungberg (malj@itu.dk)

November 19, 2015

Contents

1 Introduktion

Hukommelse er en vigtig del af et operativ system, da programmer skal indlæses i hukommelsen for at kunne køre.

I moderne operativ systemer er der typisk to former for hukommelse, nemlig den fysiske og den virtuelle hukommelse. Den fysiske hukommelse er det vi kender som RAM(Random Access Memory) og det er i denne hukommelse et program skal indlæses før kørsel. Virtuel hukommelse er derimod en proces der står for at udskifte data mellem den fysiske hukommelse og lagerenheden.

I denne rapport fokuseres der på teorien bag virtuel hukommelse, særligt omkring udskiftning af data mellem fysisk hukommelse og lager, samt hvordan det kan implementeres i et operativ system.

2 Teori

Virtuel hukommelse giver operativ systemet muligheder som fysisk hukommelse ikke kan give, såsom indikationen af mere hukommelse end der reel er, ved brug af sider. En side er en blok af data med en given størrelse. Den virtuelle hukommelse benytter sider, således at den side et program efterspørger indlæses til den fysiske hukommelse via lagerenheden og derefter til den virtuelle hukommelses side. Dette giver operativ systemet mulighed for at lave en mængde sider og alt efter processers behov indlæse og skrive data til lagerenheden. Denne teknik er også kaldet "Demand Paging".

Den virtuelle hukommelse består typisk af en sidetabel ptd, f, b med kollonerne, data for siden, sidens plads i den fysiske hukommelse(hvis den er indlæst) og et flag der indikerer om siden skal indlæses, skrives til eller eksekveres. Den fysiske hukommelses plads kaldes også for rammer i virtuel hukommelse.

Hvis der er mere fysisk hukommelse eller præcist den samme størrelse som den virtuelle hukommelse, er virtuel hukommelse ligeså hurtig som den fysiske hukommelse, da der ikke skal håndteres for side udskiftninger(bortset fra den første indlæsning af hver side). Dette er dog ikke altid tilfældet da der af flere grunde kan forekomme det som kaldes en "page fault", hvor en process tilgår en side, der ikke er i den fysiske hukommelse mere eller den fysiske hukommelse er nået sin grænse.

Dette skal den virtuelle hukommelses sideudskiftnings algoritme håndtere, da der skal tages en beslutning om hvilken ramme skal frigives. Hypotetisk set burde antallet af page faults formindskes desto tættere antallet af sider og rammer er, men dette er ikke altid tilfældet som er blevet påvist af Belady's anomalitet.

Til denne opgave fokuseres der på en tilfældig algoritme, en FIFO(First-In-First-Out) algoritme og en custom algoritme af eget valg.

2.1 Tilfældig udskiftning

Den tilfældige sideudskiftnings algoritme er en meget simpel algoritme, da den kræver at der generes et tal mellem 0 og antallet af rammer. Da det er tilfældigt givet ramme lokationer, kan antallet af page faults variere, da den ikke ved om den ramme bliver brugt eller skal til at bruges, hvilket i et senere tilfælde vil skabe endnu en page fault.

2.2 FIFO udskiftning

Denne algoritme er også meget lige til, da man skal give den ramme der er blevet indlæst data i først. Dette kræver at der er behov for en tæller, der holder styr på hvilken ramme der skal frigives. Hver gang en ramme er frigivet forhøjes tælleren med en. Det skal dog huskes at for hver gang tælleren forhøjes skal den stadig være mellem 0 og antallet af rammer. Til dette kan modulo bruges.

2.3 Custom udskiftning

Til custom udskiftnings algoritmen ser vi nærmere på en udvidet form af FIFO udskiftnings algoritmen, Second-Chance algoritmen(også kaldet Clock algoritmen). Dette er på baggrund af at den i værste tilfælde stadig vil have samme antal page faults som FIFO algoritmen og dette mener vi er en acceptabel præmise.

Selve algoritmen gør brug af en reference bit til hver ramme, der sættes til 0 når et element indlæses i hukommelsen med læse flaget og 1 når et element indlæses med skrivnings flaget. Desuden bruger den også en tæller ligesom FIFO.

Når udskiftningsalgoritmen kaldes tjekkes der for et element med 0 som reference bit. Dette tjek startes fra tællerens position. Under gennemløbet sættes de reference bit der er 1 til 0, da dette er deres anden chance, idet da gennemløbet er cirkulært og det møder dette element igen vil den miste sin plads.

3 Implementation

I dette afsnit beskrives hvorledes implementationen af en virtuel hukommelses side håndtering og udskiftnings algoritmerne beskrevet i teorien.

3.1 Page Fault Håndtering

Til at starte med er det vigtig at implementere basis page fault håndtering, altså hvordan der skal indlæses data fra disken og skrives til disken.

Dette gøres i `page_fault_handler()` metoden. Vi husker fra teorien at en side i en sidetabel har et flag, der kan benyttes til at afgøre hvad sidens behov er. Dette implementere vi med en `switch` erklæring med tre sager.

Den første sag er 0, altså et flag der hverken har læse eller skrive rettigheder, denne indikerer at denne side ikke er indlæst i hukommelsen. For at indlæse data fra disken benyttes metoderne `page_table_set_entry()`, som sætter sidens rettigheder og ramme, og `disk_read()`, der indlæser data fra disken til den tildelte ramme. For at finde ud af hvilken ramme siden skal til, tjekkes listen `loaded_pages`, der er en liste over indlæste sider i rammerne, om der er en ledig plads, som indikeres ved -1.

Hvis det ikke er muligt at finde en ledig ramme, skal en sideudskiftnings algoritme afgøre om hvilken ramme der skal tildeles. Efter en ramme er tildelt, er det nødvendigt at se om det har `PROT_READ|PROT_WRITE` flaget sat, da disse skal skrives til disken med `disk_write()` før frigivelse. Desuden skal den udskiftede side opdateres i sidetabellen med `page_table_set_entry()`.

Den anden sag i `switch` erklæringen er `PROT_READ`, der er læse flaget, når dette er tilfældet skal denne side blot have læse samt skrive rettigheder, men ikke decideret skrives til disken med det samme.

3.2 Udskiftning af sider

Når der skal sider fra den fysiske hukommelse benyttes metoden `get_swap_frame()`, der afgør hvilken udskiftningsalgoritme brugeren ville benytte med variablen `pageswap`. Når denne er 0 skal den tilfældige udskiftning foretages, 1 for FIFO udskiftning og 2 for custom(Second-Chance).

3.2.1 Tilfældig udskiftning

Den tilfældige algoritme gør brug af metoden `lrand48()` for at genere et tilfældigt tal hvorefter rammen findes ved brug af modulo med `nframes`, det maksimalt antal af rammer.

3.2.2 FIFO udskiftning

FIFO er implementeret præcist efter teorien med en tæller `fifo_counter`, der forhøjes med en efter hver ramme tildeling og derefter sættes til at være mellem 0 og `nframes` ved brug af modulo.

3.2.3 Custom udskiftning

Denne udskiftningsalgoritme kræver en lidt mere i sin implementering da den har behov for en reference bit til hver ramme, som er implementeret i from af en liste `clock`. Det der skal tages højde for ved implementeringen af denne algoritme er hvordan gennemløbet skal være og hvornår skal reference bitten sættes til 0 og 1.

Ved initialisering af `clock`, sættes alle bit til 0. I `page_fault_handler()` under indlæsning af data efter udskiftning at sidens ramme skal sættes til 0, da side udskiftning foretages i `switch` erklæringens sag 0, hvilket foroven er der hvor siden får et læse flag. I `switch` erklæringens `PROT_READ` sag, skal `clock` sættes til 1 da denne sides flag nu er et læse og skrive flag.

Under selve udskiftningen skal der gennemløbes cirkulært gennem `clock` hvor startpunktet er tælleren `fifo_counter`'s værdi, her benyttes en while-løkke. I løkken tjekkes reference bittens værdi. Hvis den er 0 kan denne ramme godt tildeles og tælleren forhøjes på samme måde som i FIFO udskiftningen. I tilfælde af at den `clock`'s værdi er 1, sættes denne til 0, da den nu får en anden chance.

4 Testing

For at teste implementationen af udskiftnings algoritmerne er der implementeret en variable `fault_counter`, der angiver antallet af page faults der har været igennem programmets kørsel. Bemærk at når man første gang laver programmet med `make` vil der forekomme to advarsler, som kommer fra page_table.c og program.c, hvilket er filer der ikke er foretaget ændringer.

Programmet køres på følgende måde:

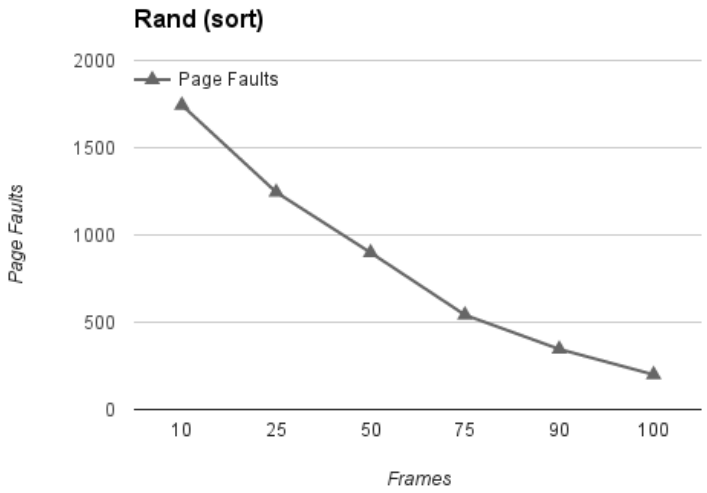
```
./virtmem npages nframes <rand|fifo|custom> <sort|scan|focus>
```

Generelt er der testet ved brug af udskrifter af variabler ved brug af `page_table_print_entry()` og `print_second_chance`, men for at afgøre og måle de forskellige algoritmer mod hinanden har vi kørt hver algoritme igennem hvert program med 100 sider i alt med varierende rammer og aflæst `fault_counter` der udskrives til sidst i programmet. Med denne information opstilles tabeller og der udarbejdes diagrammer for hver af de forskellige programmer med de forskellige algoritmer.

4.1 Tilfældig udskiftning

4.1.1 Sort

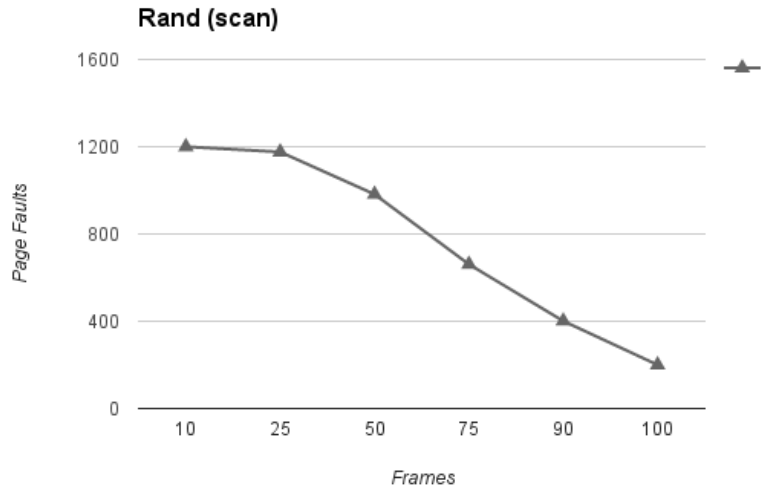
Pages	Frames	Faults
100	10	1744
100	25	1245
100	50	899
100	75	542
100	90	346
100	100	200



Diagrammet foroven viser sort programmet med den tilfældige udskiftnings algoritme.

4.1.2 Scan

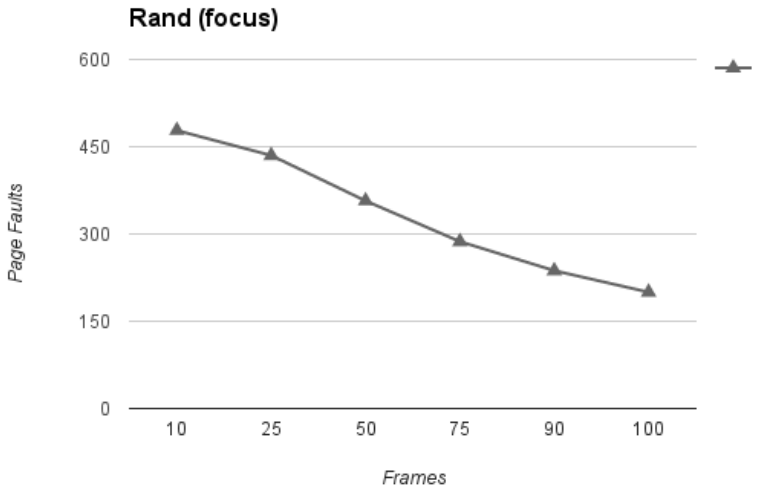
Pages	Frames	Faults
100	10	1200
100	25	1176
100	50	982
100	75	661
100	90	401
100	100	200



Diagrammet foroven viser scan programmet med den tilfældige udskiftnings algoritme.

4.1.3 Focus

Pages	Frames	Faults
100	10	478
100	25	435
100	50	357
100	75	287
100	90	237
100	100	200

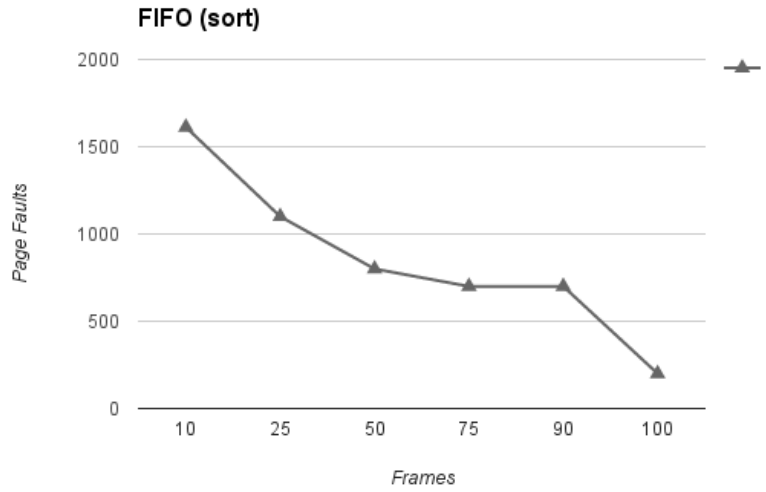


Diagrammet foroven viser focus programmet med den tilfældige udskiftnings algoritme.

4.2 FIFO udskiftning

4.2.1 Sort

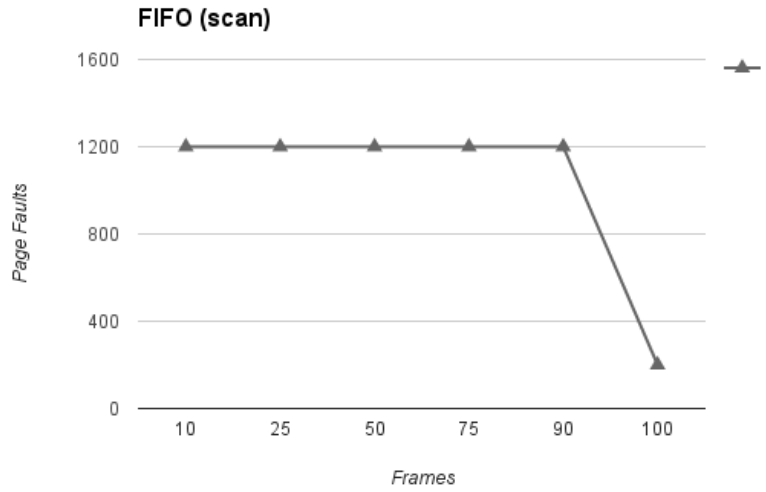
Pages	Frames	Faults
100	10	1612
100	25	1100
100	50	800
100	75	700
100	90	700
100	100	200



Diagrammet foroven viser sort programmet med FIFO.

4.2.2 Scan

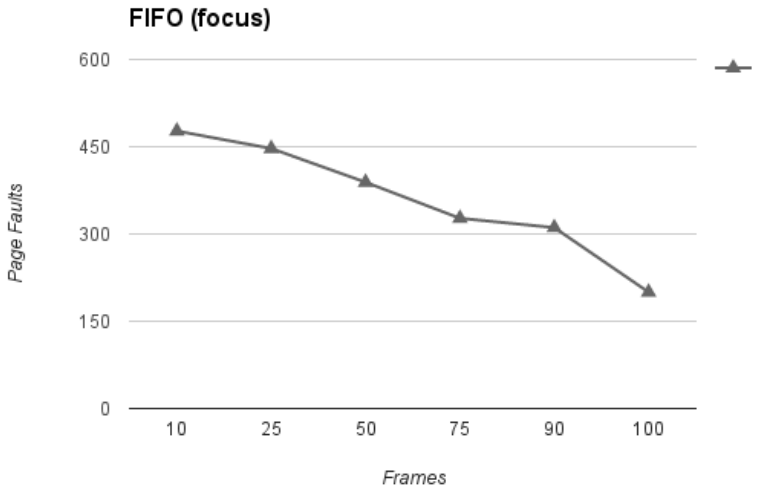
Pages	Frames	Faults
100	10	1200
100	25	1200
100	50	1200
100	75	1200
100	90	1200
100	100	200



Diagrammet foroven viser scan programmet med FIFO.

4.2.3 Focus

Pages	Frames	Faults
100	10	477
100	25	447
100	50	389
100	75	327
100	90	311
100	100	200

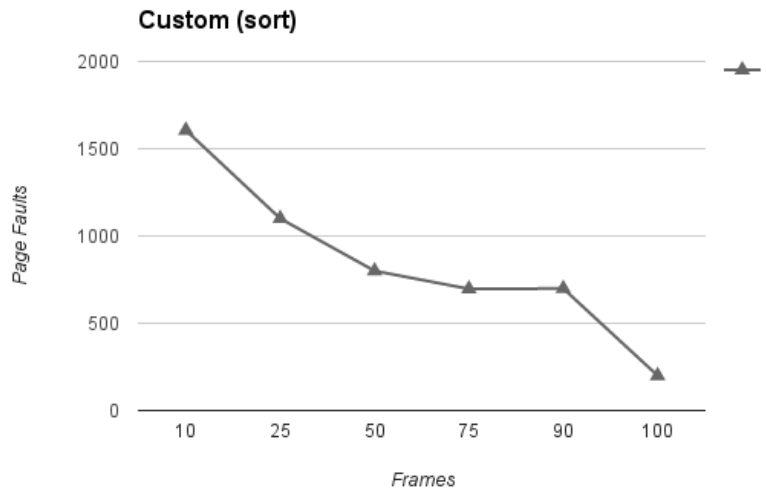


Diagrammet foroven viser focus programmet med FIFO.

4.3 Custom(Second-Chance) udskiftning

4.3.1 Sort

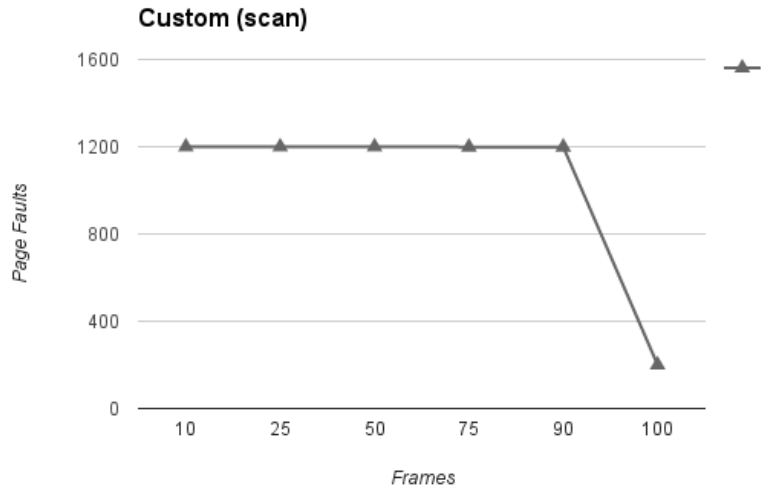
Pages	Frames	Faults
100	10	1611
100	25	1100
100	50	800
100	75	697
100	90	699
100	100	200



Diagrammet foroven viser sort programmet med Second-Chance algoritmen.

4.3.2 Scan

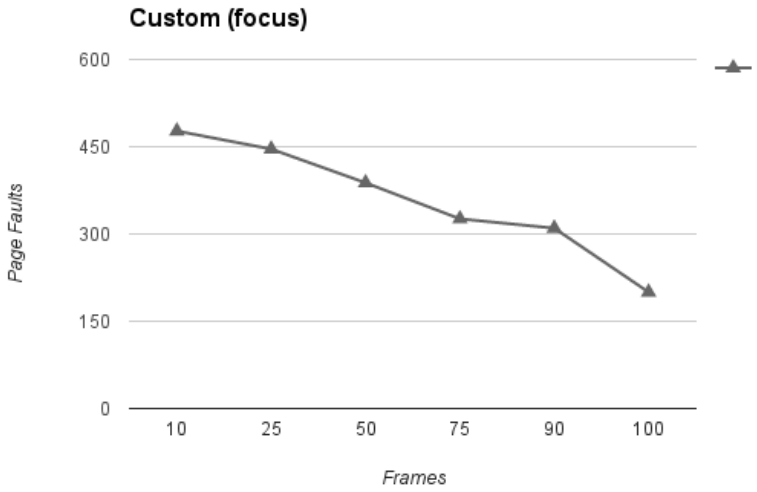
Pages	Frames	Faults
100	10	1200
100	25	1200
100	50	1200
100	75	1199
100	90	1199
100	100	200



Diagrammet foroven viser scan programmet med Second-Chance algoritmen.

4.3.3 Focus

Pages	Frames	Faults
100	10	477
100	25	446
100	50	388
100	75	326
100	90	310
100	100	200



Diagrammet foroven viser focus programmet med Second-Chance algoritmen.

4.4 Diskussion

Ud fra resultaterne kan man se at den tilfældige algoritme er meget hurtigere end de to andre. Det skal dog bemærkes, at selvom denne er hurtigere med disse programmer, så kunne det blive værre hvis ikke `lrand48()` genereret et uniform tilfældigt tal, da dette kunne forudsage et værste tilfælde hvor det tilfældige tal er det samme i alle tilfælde.

Desuden er det værd at bemærke at Second-Chance kun er bedre end FIFO med 1 page fault i de fleste tilfælde. Nærmere udforskning med mindre tal har dog vist at der kan være større forskel, som f.eks. at køre Second-Chance og FIFO med 4 sider og 3 rammer med sort. Dette passer efter teorien at Second-Chance vil i værste tilfælde have lige så mange page faults som FIFO.

5 Reflektion

6 Konklusion

7 Appendix A - Sourcecode

```
1 /*
2 Main program for the virtual memory project.
```

```
3  Make all of your modifications to this file.
4  You may add or rearrange any code or data as you need.
5  The header files page_table.h and disk.h explain
6  how to use the page table and disk interfaces.
7  */
8
9  #include "page_table.h"
10 #include "disk.h"
11 #include "program.h"
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16 #include <errno.h>
17
18 char *physmem;
19
20 struct disk *disk;
21 int npages, nframes;
22 int *loaded_pages, *clock;
23 int pageswap, fifo_counter, fault_counter = 0;
24
25 void print_second_chance()
26 {
27   int i;
28   printf("+---+---+\n");
29   for(i = 0; i < nframes; i++)
30   {
31     if(fifo_counter == i)
32     {
33       printf("| %d | %d | <-\n", loaded_pages[i], clock[i]);
34     }
35     else
36     {
37       printf("| %d | %d |\n", loaded_pages[i], clock[i]);
38     }
39     printf("+---+---+\n");
40   }
41 }
42
43 void get_swap_frame(int *vFrame)
44 {
45   int i;
46   switch(pageswap)
47   {
48     case 0:
```

```
      *vFrame = lrand48() % nframes;
      return;
    case 1:
      *vFrame = fifo_counter;
      fifo_counter++;
      fifo_counter = fifo_counter % nframes;
      return;
    case 2:
      //print_second_chance();
      i = fifo_counter;
      int do_repeat = 1;
      while(do_repeat == 1)
      {
        //check if it's reference bit is 0
        if(clock[i] == 0)
        {
          do_repeat = 0;
          *vFrame = i;
          fifo_counter++;
          fifo_counter = fifo_counter % nframes;
        }
        else
        {
          clock[i] = 0;
          i++;
          i = i % nframes;
        }
      }
      return;
  }
}

void page_fault_handler( struct page_table *pt, int page )
{
  fault_counter++;
  int flag;
  int frame;

  //get frame and flag for the page
  page_table_get_entry(pt, page, &frame, &flag);

  //page_table_print_entry(pt,page);

  int i;
  switch(flag)
  {
```

```
95      case 0:
96        //check for free frame
97        for(i = 0; i < nframes; i++)
98        {
99          if(loaded_pages[i] == -1)
100         {
101           //read from disk to physmem
102           page_table_set_entry(pt, page, i, PROT_READ);
103           disk_read(disk, page, &physmem[i*PAGE_SIZE]);
104           loaded_pages[i] = page;
105
106           //page_table_print_entry(pt,page);
107           //printf("\n");
108
109           return;
110         }
111       }
112       //variables for victim
113       int vFrame, vPage, vFlag;
114
115       //get the victim frame
116       get_swap_frame(&vFrame);
117
118       //set the victim page
119       vPage = loaded_pages[vFrame];
120
121       //get the victim flag
122       page_table_get_entry(pt, vPage, &vFrame, &vFlag);
123
124       //check for RW flag
125       int rw = (PROT_READ|PROT_WRITE);
126       if(vFlag == rw)
127       {
128         //write victim from physmem to disk
129         disk_write(disk, vPage, &physmem[vFrame*PAGE_SIZE]);
130       }
131
132       //read from disk to victim frame
133       disk_read(disk, page, &physmem[vFrame*PAGE_SIZE]);
134
135       //update page table entries
136       page_table_set_entry(pt, page, vFrame, PROT_READ);
137       page_table_set_entry(pt, vPage, 0, 0);
138       //update loaded_pages
139       loaded_pages[vFrame] = page;
140
```

```
141       if(pageswap == 2)
142       {
143         clock[vFrame] = 0;
144       }
145
146       //print_second_chance();
147       //page_table_print_entry(pt,page);
148       //printf("\n");
149
150       return;
151     case PROT_READ:
152       page_table_set_entry(pt, page, frame, PROT_READ|PROT_WRITE);
153
154       //page_table_print_entry(pt,page);
155       //printf("\n");
156
157       if(pageswap == 2)
158       {
159         clock[frame] = 1;
160       }
161       return;
162   }
163   printf("page fault on page #%d\n",page);
164   exit(1);
165 }
166
167 int main( int argc, char *argv[] )
168 {
169   if(argc!=5)
170   {
171     printf("use: virtmem <npages> <nframes> <rand|fifo|custom>
          <sort|scan|focus>\n");
172     return 1;
173   }
174
175   npages = atoi(argv[1]);
176   nframes = atoi(argv[2]);
177   const char *algorithm = argv[3];
178   const char *program = argv[4];
179
180   loaded_pages = malloc(sizeof(int) * nframes);
181   int i;
182   for(i = 0; i < nframes; i++)
183   {
184     //indicate that there is no pages loaded yet
185     loaded_pages[i] = -1;
```

```
186   }
187
188   disk = disk_open("myvirtualdisk",npages);
189   if(!disk)
190   {
191     fprintf(stderr,"couldn't create virtual disk:
          %s\n",strerror(errno));
192     return 1;
193   }
194
195   struct page_table *pt = page_table_create( npages, nframes,
       page_fault_handler );
196   if(!pt)
197   {
198     fprintf(stderr,"couldn't create page table: %s\n",strerror(errno));
199     return 1;
200   }
201
202   char *virtmem = page_table_get_virtmem(pt);
203
204   physmem = page_table_get_physmem(pt);
205
206   if(!strcmp(algorithm, "rand"))
207   {
208     pageswap = 0;
209   }
210   else if(!strcmp(algorithm, "fifo"))
211   {
212     pageswap = 1;
213     fifo_counter = 0;
214   }
215   else if(!strcmp(algorithm, "custom"))
216   {
217     pageswap = 2;
218     fifo_counter = 0;
219     clock = malloc(sizeof(int) * nframes);
220     for(i = 0; i < nframes; i++)
221     {
222       clock[i] = 0;
223     }
224   }
225   else
226   {
227     fprintf(stderr,"unknown algorithm: %s\n",argv[2]);
228   }
229
```

```
230   if(!strcmp(program,"sort"))
231   {
232     sort_program(virtmem,npages*PAGE_SIZE);
233
234   }
235   else if(!strcmp(program,"scan"))
236   {
237     scan_program(virtmem,npages*PAGE_SIZE);
238
239   }
240   else if(!strcmp(program,"focus"))
241   {
242     focus_program(virtmem,npages*PAGE_SIZE);
243
244   }
245   else
246   {
247     fprintf(stderr,"unknown program: %s\n",argv[3]);
248   }
249   printf("Faults: %d\n", fault_counter);
250   page_table_delete(pt);
251   disk_close(disk);
252
253   return 0;
254 }
```