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

For at udskifte sider skal

3.2.1 Tilfældig udskiftning

3.2.2 FIFO udskiftning

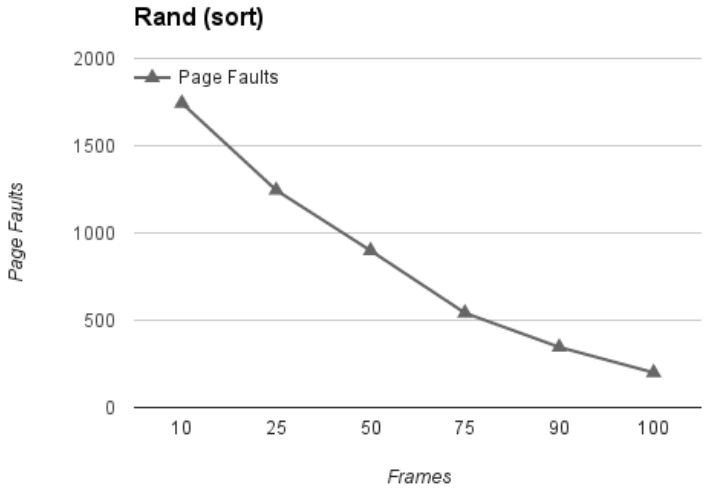
3.2.3 Custom udskiftning

4 Testing

4.1 Tilfældig udskiftning

4.1.1 Sort

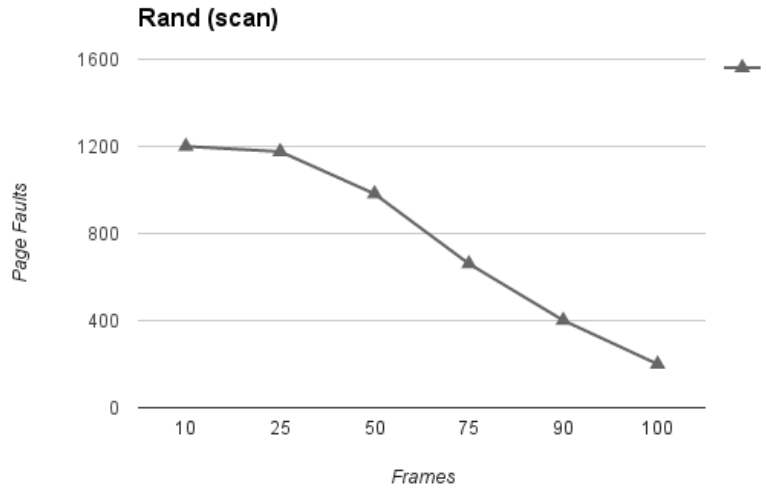
Pages	Frames	Faults
100	10	1744
100	25	1245
100	50	899
100	75	542
100	90	346
100	100	200



Diagram

4.1.2 Scan

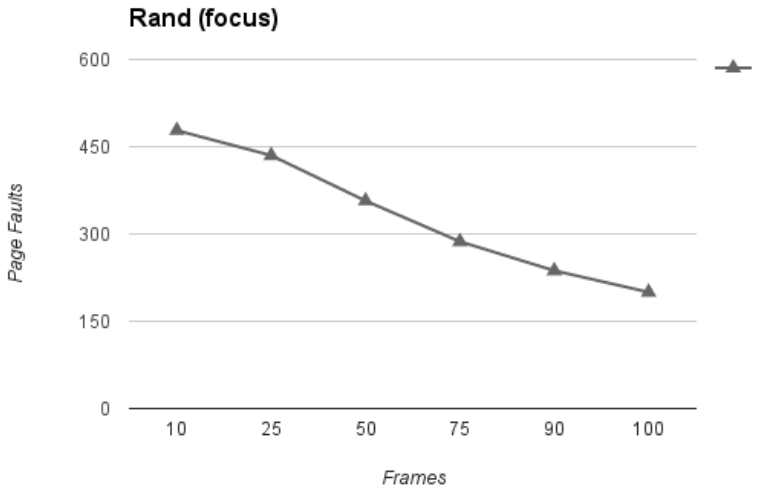
Pages	Frames	Faults
100	10	1200
100	25	1176
100	50	982
100	75	661
100	90	401
100	100	200



Diagram

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

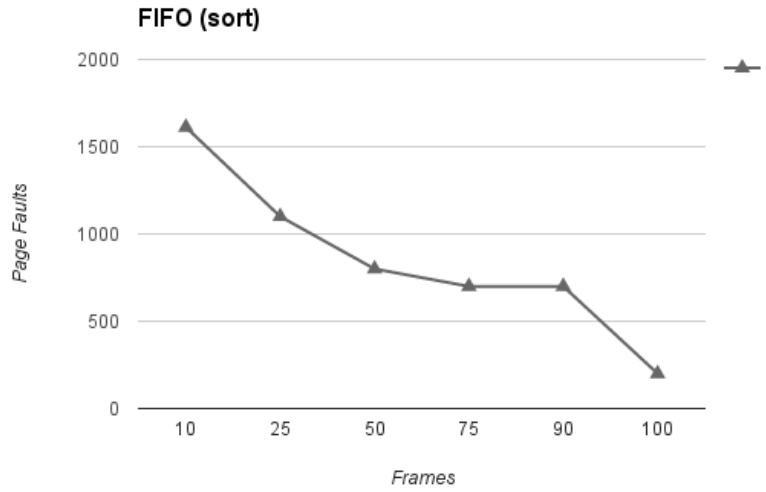


Diagram

4.2 FIFO udskiftning

4.2.1 Sort

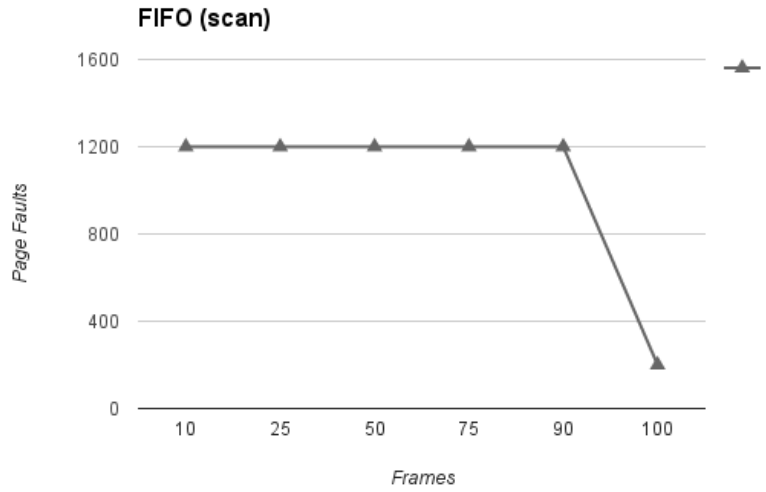
Pages	Frames	Faults
100	10	1612
100	25	1100
100	50	800
100	75	700
100	90	700
100	100	200



Diagram

4.2.2 Scan

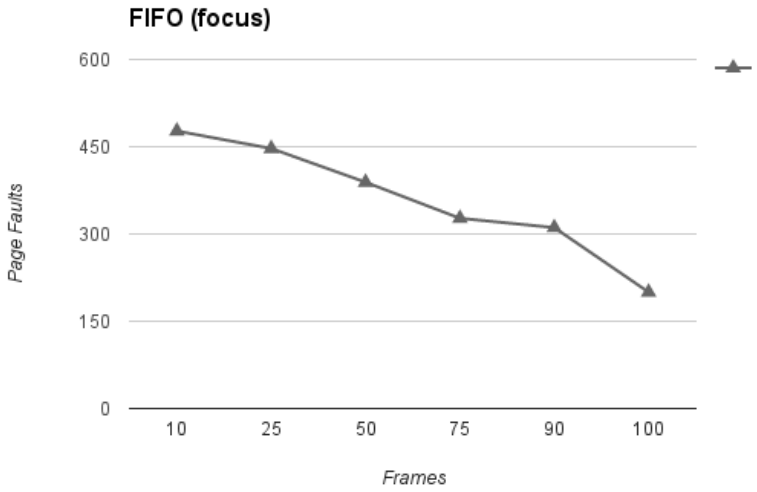
Pages	Frames	Faults
100	10	1200
100	25	1200
100	50	1200
100	75	1200
100	90	1200
100	100	200



Diagram

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

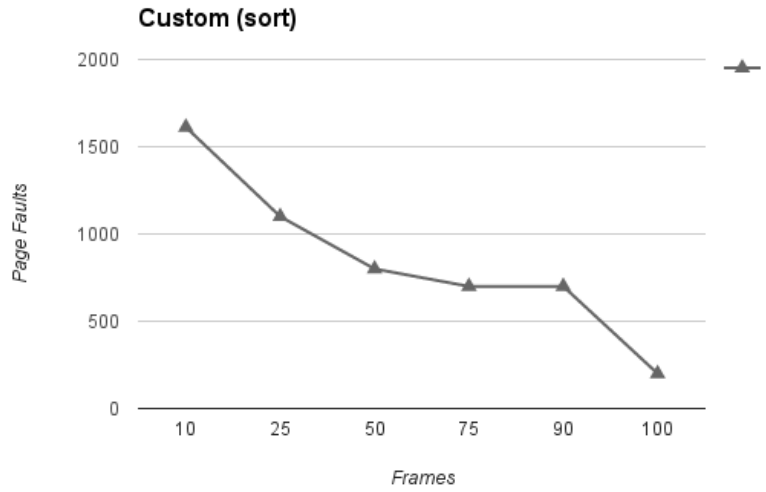


Diagram

4.3 Custom(Second-Chance) udskiftning

4.3.1 Sort

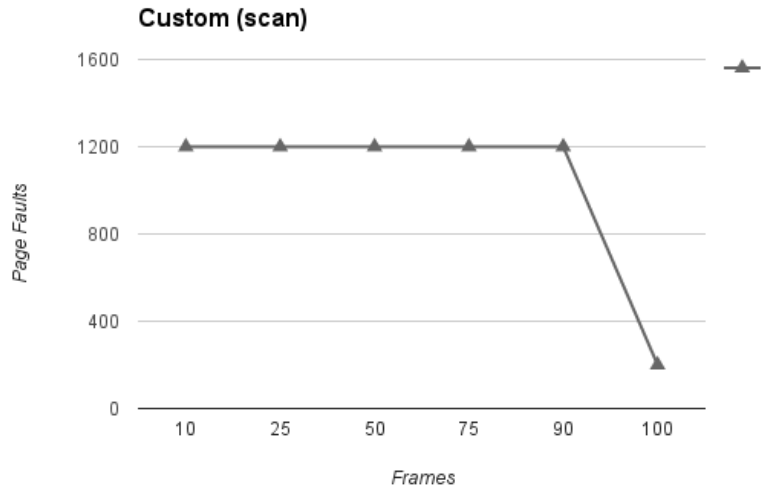
Pages	Frames	Faults
100	10	1612
100	25	1100
100	50	800
100	75	700
100	90	700
100	100	200



Diagram

4.3.2 Scan

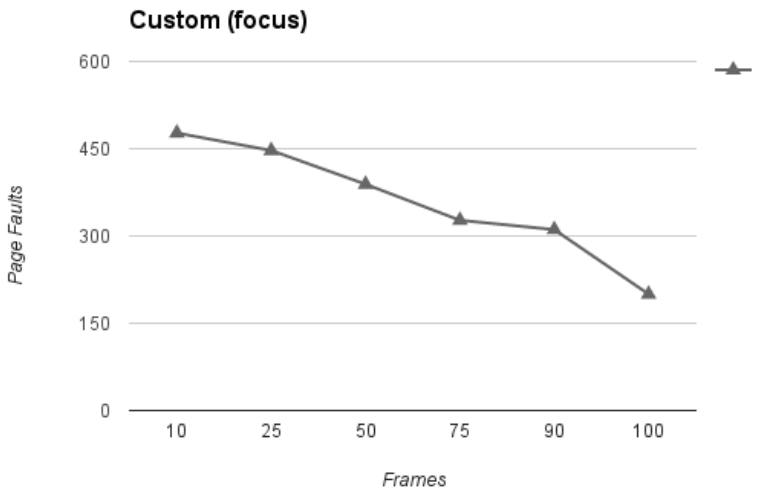
Pages	Frames	Faults
100	10	1200
100	25	1200
100	50	1200
100	75	1200
100	90	1200
100	100	200



Diagram

4.3.3 FOCUS

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



Diagram

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:
49       *vFrame = lrand48() % nframes;
50       return;
51     case 1:
52       *vFrame = fifo_counter;
53       fifo_counter++;
54       fifo_counter = fifo_counter % nframes;
55       return;
56     case 2:
57       //print_second_chance();
58       i = fifo_counter;
59       int do_repeat = 1;
60       while(do_repeat == 1)
61       {
62         //check if it's reference bit is 0
63         if(clock[i] == 0)
```

```
64          {
65            do_repeat = 0;
66            *vFrame = i;
67            fifo_counter++;
68            fifo_counter = fifo_counter % nframes;
69            clock[i] = 1;
70          }
71          else
72          {
73            clock[i] = 0;
74            i++;
75            i = i % nframes;
76            //second chance used
77            if(i == fifo_counter)
78            {
79              do_repeat = 0;
80              *vFrame = fifo_counter;
81              fifo_counter++;
82              fifo_counter = fifo_counter % nframes;
83              clock[i] = 1;
84            }
85          }
86        }
87        return;
88    }
89  }
90
91  void page_fault_handler( struct page_table *pt, int page )
92  {
93    fault_counter++;
94    int flag;
95    int frame;
96
97    //get frame and flag for the page
98    page_table_get_entry(pt, page, &frame, &flag);
99    page_table_print_entry(pt,page);
100   int i;
101   switch(flag)
102   {
103     case 0:
104       //check for free frame
105       for(i = 0; i < nframes; i++)
106       {
107         if(loaded_pages[i] == -1)
108         {
109           //read from disk to physmem
```

```
110             page_table_set_entry(pt, page, i, PROT_READ);
111             disk_read(disk, page, &physmem[i*PAGE_SIZE]);
112             loaded_pages[i] = page;
113 
114             page_table_print_entry(pt,page);
115             printf("\n");
116             if(pageswap == 2)
117             {
118               clock[i] = 0;
119             }
120             return;
121           }
122         }
123         printf("SIDESWAPPING\n");
124         //variables for victim
125         int vFrame, vPage, vFlag;
126 
127         //get the victim frame
128         get_swap_frame(&vFrame);
129 
130         //set the victim page
131         vPage = loaded_pages[vFrame];
132 
133         //get the victim flag
134         page_table_get_entry(pt, vPage, &vFrame, &vFlag);
135 
136         //check for RW flag
137         int rw = (PROT_READ|PROT_WRITE);
138         if(vFlag == rw)
139         {
140           //write victim from physmem to disk
141           disk_write(disk, vPage, &physmem[vFrame*PAGE_SIZE]);
142         }
143 
144         //read from disk to victim frame
145         disk_read(disk, page, &physmem[vFrame*PAGE_SIZE]);
146 
147         //update page table entries
148         page_table_set_entry(pt, page, vFrame, PROT_READ);
149         page_table_set_entry(pt, vPage, 0, 0);
150         page_table_print_entry(pt,page);
151         printf("\n");
152         //update loaded_pages
153         loaded_pages[vFrame] = page;
154 
155         if(pageswap == 2)
```

```
156         {
157           clock[vFrame] = 0;
158         }
159 
160         return;
161       case PROT_READ:
162         page_table_set_entry(pt, page, frame, PROT_READ|PROT_WRITE);
163         page_table_print_entry(pt,page);
164         printf("\n");
165         if(pageswap == 2)
166         {
167           clock[frame] = 1;
168         }
169         return;
170   }
171   printf("page fault on page #%d\n",page);
172   exit(1);
173 }
174 
175 int main( int argc, char *argv[] )
176 {
177   if(argc!=5)
178   {
179     printf("use: virtmem <npages> <nframes> <rand|fifo|custom>
          <sort|scan|focus>\n");
180     return 1;
181   }
182 
183   npages = atoi(argv[1]);
184   nframes = atoi(argv[2]);
185   const char *algorithm = argv[3];
186   const char *program = argv[4];
187 
188   loaded_pages = malloc(sizeof(int) * nframes);
189   int i;
190   for(i = 0; i < nframes; i++)
191   {
192     //indicate that there is no pages loaded yet
193     loaded_pages[i] = -1;
194   }
195 
196   disk = disk_open("myvirtualdisk",npages);
197   if(!disk)
198   {
199     fprintf(stderr,"couldn't create virtual disk:
          %s\n",strerror(errno));
```

```
200     return 1;
201   }
202
203   struct page_table *pt = page_table_create( npages, nframes,
          page_fault_handler );
204   if(!pt)
205   {
206     fprintf(stderr,"couldn't create page table: %s\n",strerror(errno));
207     return 1;
208   }
209
210   char *virtmem = page_table_get_virtmem(pt);
211
212   physmem = page_table_get_physmem(pt);
213
214   if(!strcmp(algorithm, "rand"))
215   {
216     pageswap = 0;
217   }
218   else if(!strcmp(algorithm, "fifo"))
219   {
220     pageswap = 1;
221     fifo_counter = 0;
222   }
223   else if(!strcmp(algorithm, "custom"))
224   {
225     pageswap = 2;
226     fifo_counter = 0;
227     clock = malloc(sizeof(int) * nframes);
228     for(i = 0; i < nframes; i++)
229     {
230       clock[i] = 0;
231     }
232   }
233   else
234   {
235     fprintf(stderr,"unknown algorithm: %s\n",argv[2]);
236   }
237
238   if(!strcmp(program,"sort"))
239   {
240     sort_program(virtmem,npages*PAGE_SIZE);
241
242   }
243   else if(!strcmp(program,"scan"))
244   {
```

```
245     scan_program(virtmem,npages*PAGE_SIZE);
246 
247   }
248   else if(!strcmp(program,"focus"))
249   {
250     focus_program(virtmem,npages*PAGE_SIZE);
251 
252   }
253   else
254   {
255     fprintf(stderr,"unknown program: %s\n",argv[3]);
256   }
257   printf("Faults: %d\n", fault_counter);
258   page_table_delete(pt);
259   disk_close(disk);
260 
261   return 0;
262 }
```