

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	3
2	Teori	3
2.1	Tilfældig udskiftning	4
2.2	FIFO udskiftning	4
2.3	Custom udskiftning	4
3	Implementation	5
3.1	Page Fault Håndtering	5
3.2	Udskiftning af sider	5
3.2.1	Tilfældig udskiftning	5
3.2.2	FIFO udskiftning	6
3.2.3	Custom udskiftning	6
4	Testing	7
4.1	Diskussion	7
5	Reflektion	8
6	Konklusion	8
7	Appendix A - Test Resultater	9
7.1	Tilfældig udskiftning	9
7.1.1	Sort	9
7.1.2	Scan	9
7.1.3	Focus	10
7.2	FIFO udskiftning	11
7.2.1	Sort	11
7.2.2	Scan	12
7.2.3	Focus	13
7.3	Custom(Second-Chance) udskiftning	14
7.3.1	Sort	14

7.3.2	Scan	15
7.3.3	Focus	16
8	Appendix B - Sourcecode	18

1 Introduktion

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

I moderne operative 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 kollerne, 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 vigtigt 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 variabelen `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 form 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 tre variabler `fault_counter`, `write_count` og `read_count`, der angiver henholdsvis antallet af page faults, disk læsninger og disk skrivninger. 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 de tre variabler 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. Resultaterne af disse test kan ses i Appendix A.

4.1 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 `rand48()` ikke genereret et uniform tilfældigt tal og før hver kørsel blev seedet et tal, da dette kunne forudsage et værste tilfælde hvor det tilfældige tal er det samme i alle tilfælde. Der er dog tilfælde med mindre sider og rammer hvor Second-Chance kan have færre page faults, indlæsninger og skrivninger.

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. Skrivninger og indlæsninger er også meget sammenligneligt med FIFO.

5 Refleksion

Denne opgave har givet os et større indblik i hvordan et virtuelt hukommelses system kan implementeres i praksis, samt udfordret os med hensyn til valget af en udskiftnings algoritme, da der netop er så mange måder at gøre det på. Netop det har gjort at vi har prøvet at implementere flere forskellige algoritmer såsom NFU og LRU, men til sidst valgte Second-Chance på baggrund af dens, i vores øjne, acceptable værste tilfælde.

Det har dog ikke lykkedes os at lave en bedre algoritme i forhold til rand, men vi mener hvis der blev seedet et tal før hver kørsel, såsom tidspunktet på dagen, ville der forekomme tilfælde hvor Second-Chance ville være bedre, men da vi ikke har nået at lave test til dette er det blot en hypotese.

6 Konklusion

Der er mange løsninger til at håndtere problemerne vedrørende hukommelsesstyring med virtuel hukommelse og ikke alle løsninger er gode til hver situation. Vi har blot set på nogle få udskiftningsalgoritmer nogle bedre i visse situationer end andre.

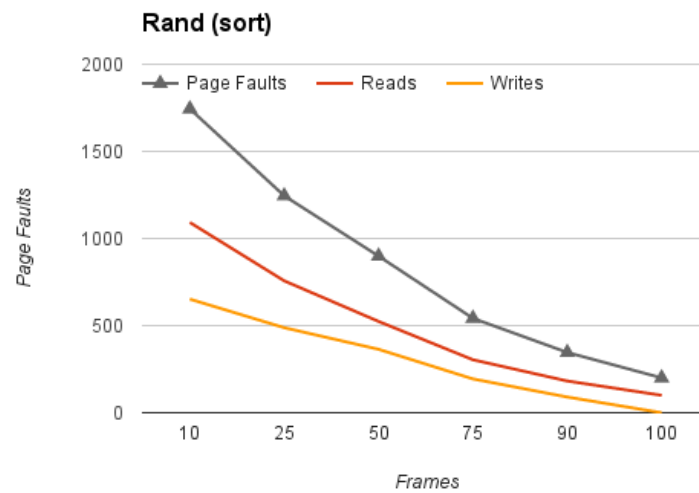
Målet var at kun foretage ændringer i en fil, main.c, og dette er også blevet overholdt. I filen er der implementeret en tilfældig, en FIFO og en custom(Second-Chance) udskiftningsalgoritme, samt håndteringen af sider generelt. Disse er blevet implementeret i forhold til den beskrevet teori og testet grundigt, hvorved vi kan konkludere at den tilfældige algoritme er hurtigere, hvis tallet ikke seedes, og Second-Chance er mere optimeret end FIFO.

7 Appendix A - Test Resultater

7.1 Tilfældig udskiftning

7.1.1 Sort

Pages	Frames	Faults	Reads	Writes
100	10	1744	1092	652
100	25	1245	757	488
100	50	899	524	364
100	75	542	304	194
100	90	346	182	90
100	100	200	100	0



Diagrammet foroven viser sort programmet med den tilfældige udskiftnings algoritme.

7.1.2 Scan

Pages	Frames	Faults	Reads	Writes
100	10	1200	1100	100
100	25	1176	1076	100
100	50	982	882	100
100	75	661	561	100

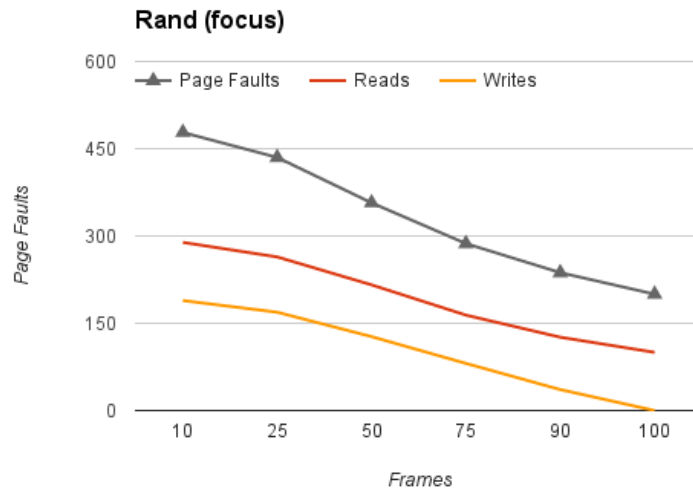
Pages	Frames	Faults	Reads	Writes
100	90	401	381	88
100	100	200	100	0



Diagrammet foroven viser scan programmet med den tilfældige udskiftnings algoritme.

7.1.3 Focus

Pages	Frames	Faults	Reads	Writes
100	10	478	289	189
100	25	435	264	169
100	50	357	216	127
100	75	287	164	81
100	90	237	126	36
100	100	200	100	0

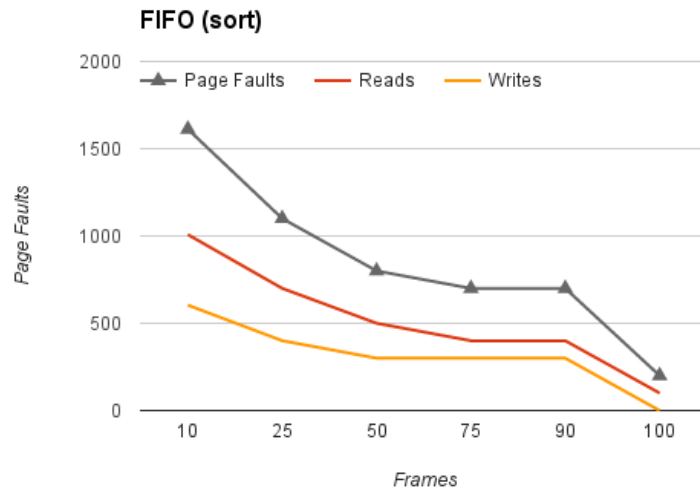


Diagrammet foroven viser focus programmet med den tilfældige udskiftnings algoritme.

7.2 FIFO udskiftning

7.2.1 Sort

Pages	Frames	Faults	Reads	Writes
100	10	1612	1008	604
100	25	1100	700	400
100	50	800	500	300
100	75	700	400	300
100	90	700	400	300
100	100	200	100	0



Diagrammet foroven viser sort programmet med FIFO.

7.2.2 Scan

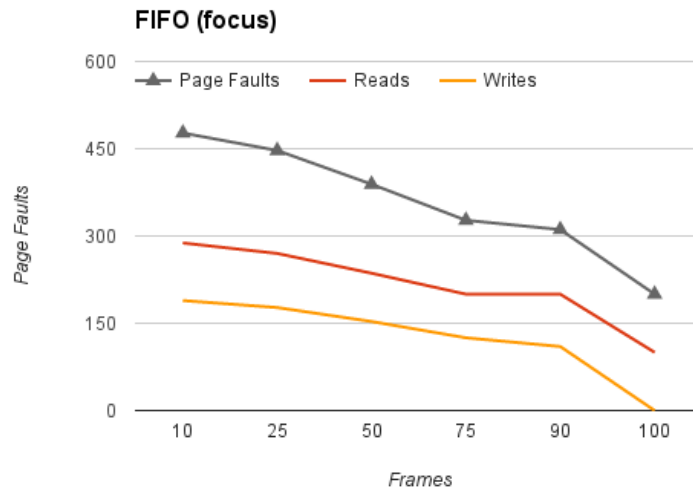
Pages	Frames	Faults	Reads	Writes
100	10	1200	1100	100
100	25	1200	1100	100
100	50	1200	1100	100
100	75	1200	1100	100
100	90	1200	1100	100
100	100	200	100	0



Diagrammet foroven viser scan programmet med FIFO.

7.2.3 Focus

Pages	Frames	Faults	Reads	Writes
100	10	477	288	189
100	25	447	270	177
100	50	389	236	153
100	75	327	200	125
100	90	311	200	110
100	100	200	100	0

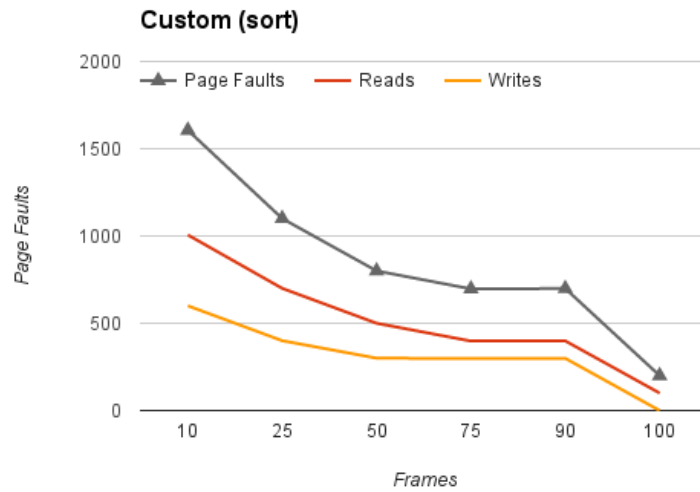


Diagrammet foroven viser focus programmet med FIFO.

7.3 Custom(Second-Chance) udskiftning

7.3.1 Sort

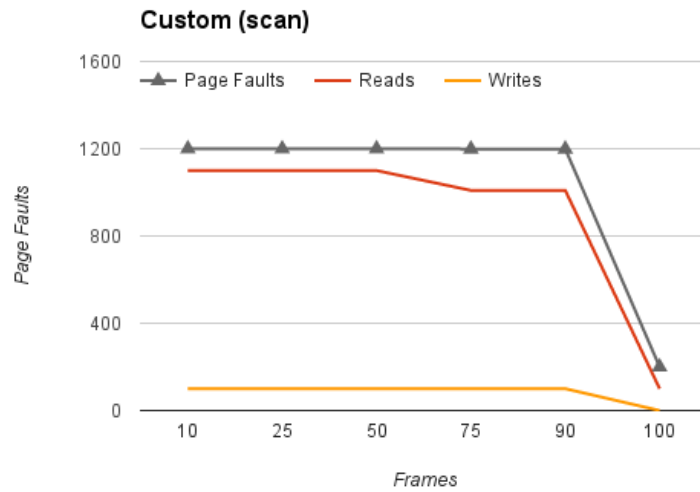
Pages	Frames	Faults	Reads	Writes
100	10	1606	1006	600
100	25	1100	700	400
100	50	800	500	300
100	75	697	398	298
100	90	699	399	299
100	100	200	100	0



Diagrammet foroven viser sort programmet med Second-Chance algoritmen.

7.3.2 Scan

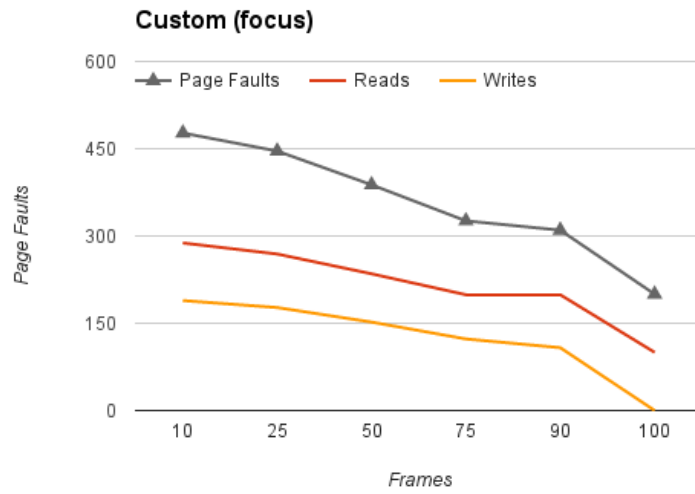
Pages	Frames	Faults	Reads	Writes
100	10	1200	1100	100
100	25	1200	1100	100
100	50	1200	1100	100
100	75	1199	1009	100
100	90	1199	1009	100
100	100	200	100	0



Diagrammet foroven viser scan programmet med Second-Chance algoritmen.

7.3.3 Focus

Pages	Frames	Faults	Reads	Writes
100	10	477	288	189
100	25	446	269	177
100	50	388	235	152
100	75	326	199	123
100	90	310	199	108
100	100	200	100	0



Diagrammet foroven viser focus programmet med Second-Chance algoritmen.

8 Appendix B - Sourcecode

main.c

```
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, write_count = 0,
    read_count = 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                 }
70                 else
71                 {
72                     clock[i] = 0;
73                     i++;
74                     i = i % nframes;
75                 }
76             }
77             return;
78     }
79 }
80
81 void page_fault_handler( struct page_table *pt, int page )
82 {
83     fault_counter++;
84     int flag;
85     int frame;
86
87     //get frame and flag for the page

```

```

88     page_table_get_entry(pt, page, &frame, &flag);
89
90     //page_table_print_entry(pt,page);
91
92     int i;
93     switch(flag)
94     {
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                     read_count++;
106
107                     //page_table_print_entry(pt,page);
108                     //printf("\n");
109
110                     return;
111                 }
112             }
113             //variables for victim
114             int vFrame, vPage, vFlag;
115
116             //get the victim frame
117             get_swap_frame(&vFrame);
118
119             //set the victim page
120             vPage = loaded_pages[vFrame];
121
122             //get the victim flag
123             page_table_get_entry(pt, vPage, &vFrame, &vFlag);
124
125             //check for RW flag
126             int rw = (PROT_READ|PROT_WRITE);
127             if(vFlag == rw)
128             {
129                 //write victim from physmem to disk
130                 disk_write(disk, vPage, &physmem[vFrame*PAGE_SIZE]);
131                 write_count++;
132             }
133

```

```

134 //read from disk to victim frame
135 disk_read(disk, page, &physmem[vFrame*PAGE_SIZE]);
136 read_count++;
137
138 //update page table entries
139 page_table_set_entry(pt, page, vFrame, PROT_READ);
140 page_table_set_entry(pt, vPage, 0, 0);
141 //update loaded_pages
142 loaded_pages[vFrame] = page;
143
144 if(pageswap == 2)
145 {
146     clock[vFrame] = 0;
147 }
148
149 //print_second_chance();
150 //page_table_print_entry(pt,page);
151 //printf("\n");
152
153 return;
154 case PROT_READ:
155     page_table_set_entry(pt, page, frame, PROT_READ|PROT_WRITE);
156
157     //page_table_print_entry(pt,page);
158     //printf("\n");
159
160     if(pageswap == 2)
161     {
162         clock[frame] = 1;
163     }
164     return;
165 }
166 printf("page fault on page %d\n",page);
167 exit(1);
168 }
169
170 int main( int argc, char *argv[] )
171 {
172     if(argc!=5)
173     {
174         printf("use: virtmem <npages> <nframes> <rand|fifo|custom>
175             <sort|scan|focus>\n");
176         return 1;
177     }
178     npages = atoi(argv[1]);

```

```

179 nframes = atoi(argv[2]);
180 const char *algorithm = argv[3];
181 const char *program = argv[4];
182
183 loaded_pages = malloc(sizeof(int) * nframes);
184 int i;
185 for(i = 0; i < nframes; i++)
186 {
187     //indicate that there is no pages loaded yet
188     loaded_pages[i] = -1;
189 }
190
191 disk = disk_open("myvirtualdisk", npages);
192 if(!disk)
193 {
194     fprintf(stderr, "couldn't create virtual disk:
195         %s\n", strerror(errno));
196     return 1;
197 }
198
199 struct page_table *pt = page_table_create( npages, nframes,
200     page_fault_handler );
201 if(!pt)
202 {
203     fprintf(stderr, "couldn't create page table: %s\n", strerror(errno));
204     return 1;
205 }
206
207 char *virtmem = page_table_get_virtmem(pt);
208
209 physmem = page_table_get_physmem(pt);
210
211 if(!strcmp(algorithm, "rand"))
212 {
213     pageswap = 0;
214 }
215 else if(!strcmp(algorithm, "fifo"))
216 {
217     pageswap = 1;
218     fifo_counter = 0;
219 }
220 else if(!strcmp(algorithm, "custom"))
221 {
222     pageswap = 2;
223     fifo_counter = 0;
224     clock = malloc(sizeof(int) * nframes);

```

```

223     for(i = 0; i < nframes; i++)
224     {
225         clock[i] = 0;
226     }
227 }
228 else
229 {
230     fprintf(stderr, "unknown algorithm: %s\n", argv[2]);
231 }
232
233 if(!strcmp(program, "sort"))
234 {
235     sort_program(virtmem, npages*PAGE_SIZE);
236 }
237
238 else if(!strcmp(program, "scan"))
239 {
240     scan_program(virtmem, npages*PAGE_SIZE);
241 }
242
243 else if(!strcmp(program, "focus"))
244 {
245     focus_program(virtmem, npages*PAGE_SIZE);
246 }
247
248 else
249 {
250     fprintf(stderr, "unknown program: %s\n", argv[3]);
251 }
252 printf("Faults: %d Reads: %d Writes: %d\n", fault_counter,
        read_count, write_count);
253 page_table_delete(pt);
254 disk_close(disk);
255
256 return 0;
257 }

```


disk.h

```
1
2 /*
3 Do not modify this file.
4 Make all of your changes to main.c instead.
5 */
6
7 #ifndef DISK_H
8 #define DISK_H
9
10 #define BLOCK_SIZE 4096
11
12 /*
13 Create a new virtual disk in the file "filename", with the given
14   number of blocks.
15 Returns a pointer to a new disk object, or null on failure.
16 */
17 struct disk * disk_open( const char *filename, int blocks );
18
19 /*
20 Write exactly BLOCK_SIZE bytes to a given block on the virtual disk.
21 "d" must be a pointer to a virtual disk, "block" is the block number,
22 and "data" is a pointer to the data to write.
23 */
24
25 void disk_write( struct disk *d, int block, const char *data );
26
27 /*
28 Read exactly BLOCK_SIZE bytes from a given block on the virtual disk.
29 "d" must be a pointer to a virtual disk, "block" is the block number,
30 and "data" is a pointer to where the data will be placed.
31 */
32
33 void disk_read( struct disk *d, int block, char *data );
34
35 /*
36 Return the number of blocks in the virtual disk.
37 */
38
39 int disk_nblocks( struct disk *d );
40
41 /*
42 Close the virtual disk.
43 */
```

```
44  
45 void disk_close( struct disk *d );  
46  
47 #endif
```

disk.c

```
1  /*
2  Do not modify this file.
3  Make all of your changes to main.c instead.
4  */
5
6  #include "disk.h"
7
8  #include <unistd.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <errno.h>
13 #include <fcntl.h>
14
15 extern ssize_t pread (int __fd, void *__buf, size_t __nbytes, __off_t
    __offset);
16 extern ssize_t pwrite (int __fd, const void *__buf, size_t __nbytes,
    __off_t __offset);
17
18
19 struct disk {
20     int fd;
21     int block_size;
22     int nblocks;
23 };
24
25 struct disk * disk_open( const char *diskname, int nblocks )
26 {
27     struct disk *d;
28
29     d = malloc(sizeof(*d));
30     if(!d) return 0;
31
32     d->fd = open(diskname, O_CREAT|O_RDWR, 0777);
33     if(d->fd<0) {
34         free(d);
35         return 0;
36     }
37
38     d->block_size = BLOCK_SIZE;
39     d->nblocks = nblocks;
40
41     if(ftruncate(d->fd, d->nblocks*d->block_size)<0) {
42         close(d->fd);
```

```

43     free(d);
44     return 0;
45 }
46
47     return d;
48 }
49
50 void disk_write( struct disk *d, int block, const char *data )
51 {
52     if(block<0 || block>=d->nblocks) {
53         fprintf(stderr,"disk_write: invalid block %#d\n",block);
54         abort();
55     }
56
57     int actual = pwrite(d->fd,data,d->block_size,block*d->block_size);
58     if(actual!=d->block_size) {
59         fprintf(stderr,"disk_write: failed to write block %#d:
60             %s\n",block,strerror(errno));
61         abort();
62     }
63 }
64
65 void disk_read( struct disk *d, int block, char *data )
66 {
67     if(block<0 || block>=d->nblocks) {
68         fprintf(stderr,"disk_read: invalid block %#d\n",block);
69         abort();
70     }
71
72     int actual = pread(d->fd,data,d->block_size,block*d->block_size);
73     if(actual!=d->block_size) {
74         fprintf(stderr,"disk_read: failed to read block %#d:
75             %s\n",block,strerror(errno));
76         abort();
77     }
78 }
79
80 int disk_nblocks( struct disk *d )
81 {
82     return d->nblocks;
83 }
84
85 void disk_close( struct disk *d )
86 {
87     close(d->fd);
88     free(d);

```


page_table.h

```
1 #ifndef PAGE_TABLE_H
2 #define PAGE_TABLE_H
3
4 #include <sys/mman.h>
5
6 #ifndef PAGE_SIZE
7 #define PAGE_SIZE 4096
8 #endif
9
10 struct page_table;
11
12 typedef void (*page_fault_handler_t) ( struct page_table *pt, int page
    );
13
14 /* Create a new page table, along with a corresponding virtual memory
15 that is "npages" big and a physical memory that is "nframes" bit
16 When a page fault occurs, the routine pointed to by "handler" will be
    called. */
17
18 struct page_table * page_table_create( int npages, int nframes,
    page_fault_handler_t handler );
19
20 /* Delete a page table and the corresponding virtual and physical
    memories. */
21
22 void page_table_delete( struct page_table *pt );
23
24 /*
25 Set the frame number and access bits associated with a page.
26 The bits may be any of PROT_READ, PROT_WRITE, or PROT_EXEC
    logical-ored together.
27 */
28
29 void page_table_set_entry( struct page_table *pt, int page, int frame,
    int bits );
30
31 /*
32 Get the frame number and access bits associated with a page.
33 "frame" and "bits" must be pointers to integers which will be filled
    with the current values.
34 The bits may be any of PROT_READ, PROT_WRITE, or PROT_EXEC
    logical-ored together.
35 */
36
```

```

37 void page_table_get_entry( struct page_table *pt, int page, int
    *frame, int *bits );
38
39 /* Return a pointer to the start of the virtual memory associated with
    a page table. */
40
41 char * page_table_get_virtmem( struct page_table *pt );
42
43 /* Return a pointer to the start of the physical memory associated
    with a page table. */
44
45 char * page_table_get_physmem( struct page_table *pt );
46
47 /* Return the total number of frames in the physical memory. */
48
49 int page_table_get_nframes( struct page_table *pt );
50
51 /* Return the total number of pages in the virtual memory. */
52
53 int page_table_get_npages( struct page_table *pt );
54
55 /* Print out the page table entry for a single page. */
56
57 void page_table_print_entry( struct page_table *pt, int page );
58
59 /* Print out the state of every page in a page table. */
60
61 void page_table_print( struct page_table *pt );
62
63 #endif

```

page_table.c

```
1
2 /*
3 Do not modify this file.
4 Make all of your changes to main.c instead.
5 */
6
7 #include <sys/types.h>
8 #include <unistd.h>
9 #include <sys/mman.h>
10 #include <limits.h>
11 #include <stdio.h>
12 #include <fcntl.h>
13 #include <stdlib.h>
14 #include <ucontext.h>
15
16 #include "page_table.h"
17
18 struct page_table {
19     int fd;
20     char *virtmem;
21     int npages;
22     char *physmem;
23     int nframes;
24     int *page_mapping;
25     int *page_bits;
26     page_fault_handler_t handler;
27 };
28
29 struct page_table *the_page_table = 0;
30
31 static void internal_fault_handler( int signum, siginfo_t *info, void
    *context )
32 {
33
34 #ifdef i386
35     char *addr = (char*)((((struct ucontext *)context)->uc_mcontext.cr2));
36 #else
37     char *addr = info->si_addr;
38 #endif
39
40     struct page_table *pt = the_page_table;
41
42     if(pt) {
43         int page = (addr-pt->virtmem) / PAGE_SIZE;
```



```

44
45     if(page>=0 && page<pt->npages) {
46         pt->handler(pt,page);
47         return;
48     }
49 }
50
51 fprintf(stderr,"segmentation fault at address %p\n",addr);
52 abort();
53 }
54
55 struct page_table * page_table_create( int npages, int nframes,
56     page_fault_handler_t handler )
57 {
58     int i;
59     struct sigaction sa;
60     struct page_table *pt;
61     char filename[256];
62
63     pt = malloc(sizeof(struct page_table));
64     if(!pt) return 0;
65
66     the_page_table = pt;
67
68     sprintf(filename, "/tmp/pmem.%d.%d",getpid(),getuid());
69
70     pt->fd = open(filename,O_CREAT|O_TRUNC|O_RDWR,0777);
71     if(!pt->fd) return 0;
72
73     ftruncate(pt->fd,PAGE_SIZE*npages);
74
75     unlink(filename);
76
77     pt->physmem =
78         mmap(0,nframes*PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,pt->fd,0);
79     pt->nframes = nframes;
80
81     pt->virtmem =
82         mmap(0,npages*PAGE_SIZE,PROT_NONE,MAP_SHARED|MAP_NORESERVE,pt->fd,0);
83     pt->npages = npages;
84
85     pt->page_bits = malloc(sizeof(int)*npages);
86     pt->page_mapping = malloc(sizeof(int)*npages);
87
88     pt->handler = handler;
89 }

```

```

87     for(i=0;i<pt->npages;i++) pt->page_bits[i] = 0;
88
89     sa.sa_sigaction = internal_fault_handler;
90     sa.sa_flags = SA_SIGINFO;
91
92     sigfillset( &sa.sa_mask );
93     sigaction( SIGSEGV, &sa, 0 );
94
95     return pt;
96 }
97
98 void page_table_delete( struct page_table *pt )
99 {
100     munmap(pt->virtmem,pt->npages*PAGE_SIZE);
101     munmap(pt->physmem,pt->nframes*PAGE_SIZE);
102     free(pt->page_bits);
103     free(pt->page_mapping);
104     close(pt->fd);
105     free(pt);
106 }
107
108 void page_table_set_entry( struct page_table *pt, int page, int frame,
109     int bits )
110 {
111     if( page<0 || page>=pt->npages ) {
112         fprintf(stderr,"page_table_set_entry: illegal page %d\n",page);
113         abort();
114     }
115
116     if( frame<0 || frame>=pt->nframes ) {
117         fprintf(stderr,"page_table_set_entry: illegal frame %d\n",frame);
118         abort();
119     }
120
121     pt->page_mapping[page] = frame;
122     pt->page_bits[page] = bits;
123
124     remap_file_pages(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,0,frame,0);
125     mprotect(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,bits);
126 }
127
128 void page_table_get_entry( struct page_table *pt, int page, int
129     *frame, int *bits )
130 {
131     if( page<0 || page>=pt->npages ) {
132         fprintf(stderr,"page_table_get_entry: illegal page %d\n",page);

```

```

131     abort();
132 }
133
134 *frame = pt->page_mapping[page];
135 *bits = pt->page_bits[page];
136 }
137
138 void page_table_print_entry( struct page_table *pt, int page )
139 {
140     if( page<0 || page>=pt->npages ) {
141         fprintf(stderr, "page_table_print_entry: illegal page #%d\n", page);
142         abort();
143     }
144
145     int b = pt->page_bits[page];
146
147     printf("page %06d: frame %06d bits %c%c%c\n",
148           page,
149           pt->page_mapping[page],
150           b&PROT_READ ? 'r' : '-',
151           b&PROT_WRITE ? 'w' : '-',
152           b&PROT_EXEC ? 'x' : '-'
153     );
154 }
155
156 void page_table_print( struct page_table *pt )
157 {
158     int i;
159     for(i=0; i<pt->npages; i++) {
160         page_table_print_entry(pt, i);
161     }
162 }
163
164 int page_table_get_nframes( struct page_table *pt )
165 {
166     return pt->nframes;
167 }
168
169 int page_table_get_npages( struct page_table *pt )
170 {
171     return pt->npages;
172 }
173
174 char * page_table_get_virtmem( struct page_table *pt )
175 {
176

```

```
177     return pt->virtmem;
178 }
179
180 char * page_table_get_physmem( struct page_table *pt )
181 {
182     return pt->physmem;
183 }
```

program.h

```
1  /*
2  Do not modify this file.
3  Make all of your changes to main.c instead.
4  */
5
6  #ifndef PROGRAM_H
7  #define PROGRAM_H
8
9  void scan_program( char *data, int length );
10 void sort_program( char *data, int length );
11 void focus_program( char *data, int length );
12
13 #endif
```

program.c

```
1  /*
2  Do not modify this file.
3  Make all of your changes to main.c instead.
4  */
5
6  #include "program.h"
7
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 static int compare_bytes( const void *pa, const void *pb )
12 {
13     int a = *(char*)pa;
14     int b = *(char*)pb;
15
16     if(a<b) {
17         return -1;
18     } else if(a==b) {
19         return 0;
20     } else {
21         return 1;
22     }
23 }
24
25
26 void focus_program( char *data, int length )
27 {
28     int total=0;
29     int i,j;
30
31     srand(38290);
32
33     for(i=0;i<length;i++) {
34         data[i] = 0;
35     }
36
37     for(j=0;j<100;j++) {
38         int start = rand()%length;
39         int size = 25;
40         for(i=0;i<100;i++) {
41             data[ (start+rand()%size)%length ] = rand();
42         }
43     }
44 }
```

```

45     for(i=0;i<length;i++) {
46         total += data[i];
47     }
48
49     printf("focus result is %d\n",total);
50 }
51
52 void sort_program( char *data, int length )
53 {
54     int total = 0;
55     int i;
56
57     srand(4856);
58
59     for(i=0;i<length;i++) {
60         data[i] = rand();
61     }
62
63     qsort(data,length,1,compare_bytes);
64
65     for(i=0;i<length;i++) {
66         total += data[i];
67     }
68
69     printf("sort result is %d\n",total);
70 }
71 }
72
73 void scan_program( char *cdata, int length )
74 {
75     unsigned i, j;
76     unsigned char *data = cdata;
77     unsigned total = 0;
78
79     for(i=0;i<length;i++) {
80         data[i] = i%256;
81     }
82
83     for(j=0;j<10;j++) {
84         for(i=0;i<length;i++) {
85             total += data[i];
86         }
87     }
88
89     printf("scan result is %d\n",total);
90 }

```

Makefile

```
1
2 virtmem: main.o page_table.o disk.o program.o
3   gcc main.o page_table.o disk.o program.o -o virtmem
4
5 main.o: main.c
6   gcc -Wall -g -c main.c -o main.o
7
8 page_table.o: page_table.c
9   gcc -Wall -g -c page_table.c -o page_table.o
10
11 disk.o: disk.c
12   gcc -Wall -g -c disk.c -o disk.o
13
14 program.o: program.c
15   gcc -Wall -g -c program.c -o program.o
16
17
18 clean:
19   rm -f *.o virtmem
```