IT University of Copenhagen

Operativsystemer og C

BOSC

Obligatorisk Opgave 3

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

November 20, 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 har vi valgt at se på to algoritmer, den ene som er en udvidet form af FIFO udskiftnings algoritmen, Second-Chance algoritmen(også kaldet Clock algoritmen) og den anden værende LRU, Least Recently Used algoritmen som skulle være den mest optimerede end alle førnævnte algoritmer.

2.3.1 Second-Chance

Second-Chance ser vi nærmere på pga. at den i værste tilfælde stadig vil have samme antal page faults som FIFO algoritmen og dette mener vi er en acceptabel præmise. Udover dette er den også en approximation til LRU, så det ville være spændene at måle forskellen mellem disse.

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.

2.3.2 LRU

Least Recently Used algoritmen går ud på at erstatte den side i den fysiske hukommelse der er blevet brugt mindst. Dette kan mindske antallet af page faults,

indlæsninger og skrivninger, da man kan antage at det mindst anvendte side nok ikke bliver refereret igen foreløbigt. Teoretisk set så bliver denne algoritme bedre desto flere rammer der er og er en undtagelse for Belady's anomalitet.

LRU algoritmen består af en liste over rammer i den fysiske hukommelse. For hver gang en side opdateres eller tilføjes i rammen sættes denne rammes værdi i listen til en "tids" værdi, hvilket kunne være en tæller. For at finde den mindst anvendte element i den fysiske hukommelse findes det element med den mindste tidsværdi eller hvis man benytter en tæller så det element med den højeste tæller værdi.

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()`. Dette ordnes i metoden `page_fault_helper()`.

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 (Second-Chance)

Denne udskiftningsalgoritme kræver 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_helper()` sættes den udskiftede sides ramme til 0, da der sættes 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.

3.2.4 Custom (LRU)

For at implementere LRU benyttes en liste ligesom i Second-Chance `clock`. Denne skal dog benyttes som en tæller for hver ramme. Følges teorien så skal værdien for en ramme i `clock` sættes til 0 når der tilføjes eller opdateres et element.

Uret tikker dog opad for hver gang en ramme opdateres dvs. alle ramme pladsers værdier i `clock` skal stige med en undtagen den opdateret rammes plads. Dette kan ses i `get_swap_frame()` under `case 3`, hvor der først løbes igennem `clock` for at finde den mindst brugte ramme, og derefter tikke uret for alle undtagen denne.

Desuden skal `clock` også opdateres når `page_fault_handler()` får en `PROT_READ`, da dette betyder at siden i rammen bliver opdateret.

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|custom2> <sort|scan|focus>
```

`custom` er Second-Chance algoritmen og `custom2` er LRU algoritmen.

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 test resultaterne kan det ses at LRU er den bedste side udskiftningsalgoritme af de tre andre, da den har færre indlæsninger og skrivninger til disk, samt færre page faults. LRU algoritmen følger også teorien med at den bliver bedre idet den får tildelt flere rammer. Med et mindre antal rammer er der dog ikke den store forskel mellem algoritmerne.

Den tilfældige algoritme er bedre end de to resterende algoritmer, men det skal dog bemærkes, at selvom denne er bedre med disse programmer, så kunne det blive værre hvis `lrand48()` ikke genereret et uniform tilfældigt tal eller 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 sider og rammer 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 Reflektion

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. Vi valgte to ekstra algoritmer, da det ville være spændene at se forskellen mellem dem.

Testene viste dog en meget større forskel end vi havde regnet med, men vi mener at dette skyldes at programmerne danner page faults sekventielt hvilket i sidste ende vil få Second-Chance til at være ligesom FIFO. Det kunne være interessant at se et program med tilfældige siders page fault og sammenligne Second-Chance og LRU.

Vi har også gjort overvejelser om hvorvidt vi skulle implementere et tjek under `page_fault_handler()`'s `PROT_READ` sag, for at se om siden der skal ænderes til et læse og skrive flag, er i rammetabellen. For at implementere dette kræver det blot en `if` erklæring omkring den nuværende metode for at lave tjekket, og blot bruge metoderne `get_swap_frame()` og `page_fault_helper()`, på samme måde som de benyttes i `0` sagen. Grunden til at vi ikke har implementeret dette er på baggrund af opgavebeskrivelsens illustrations eksempel, hvor vi fortolker en side for en `PROT_READ` sag som allerede værende i rammetabellen.

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 for opgaven 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 to custom udskiftningsalgoritmer, Second-Chance og LRU, selvom kravet var en. Desuden skulle håndteringen af page fault laves.

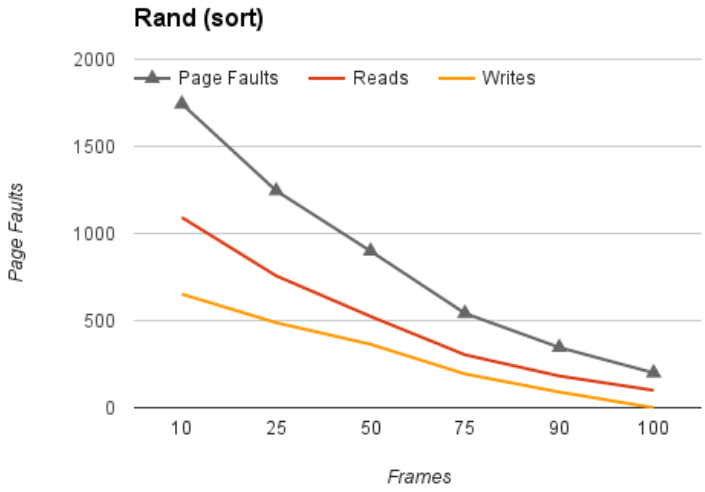
Kraevne foroven er blevet implementeret i forhold til den beskrevet teori og testet grundigt, og dermed kan vi konkludere at LRU er den bedste af de tre andre udskiftningsalgoritmer og at den tilfældige algoritme viste sig at være bedre med disse programmer over både FIFO og Second-Chance.

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

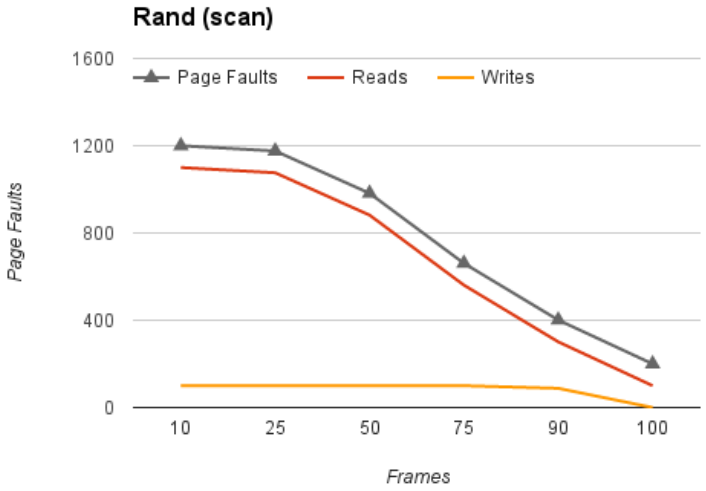


Diagrammet foroven viser sort programmet med den tilfældige udskiftnings algoritme.

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

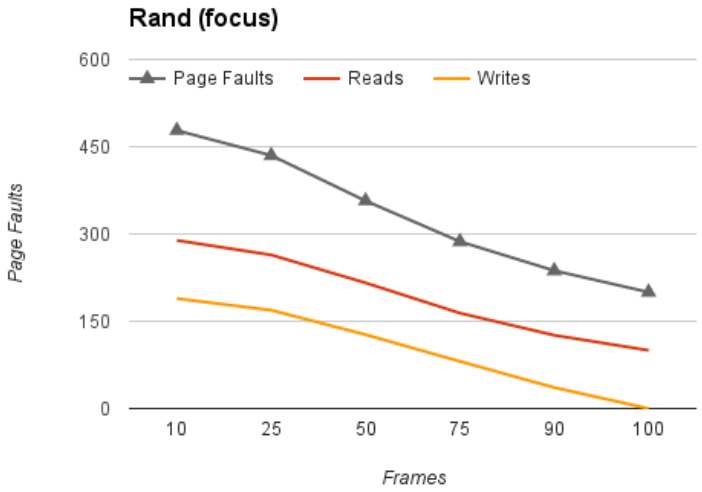
Pages	Frames	Faults	Reads	Writes
100	90	401	381	88
100	100	200	100	0



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

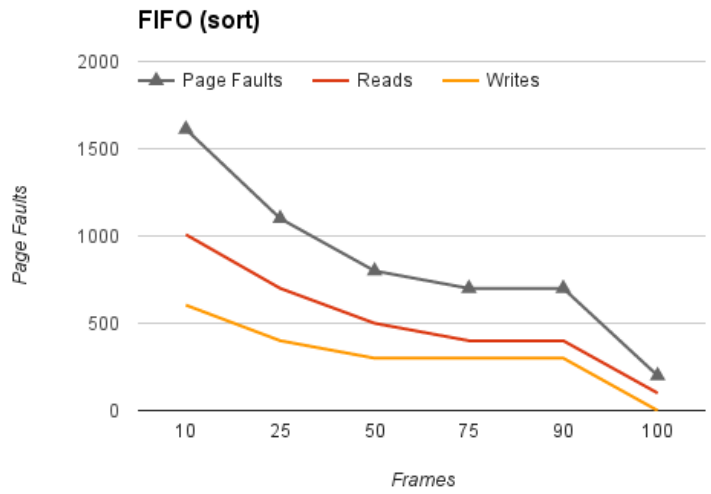


Diagrammet foroven viser focus programmet med den tilfældige udskiftnings algoritme.

7.2 FIFO udskiftning

7.2.1 Sort

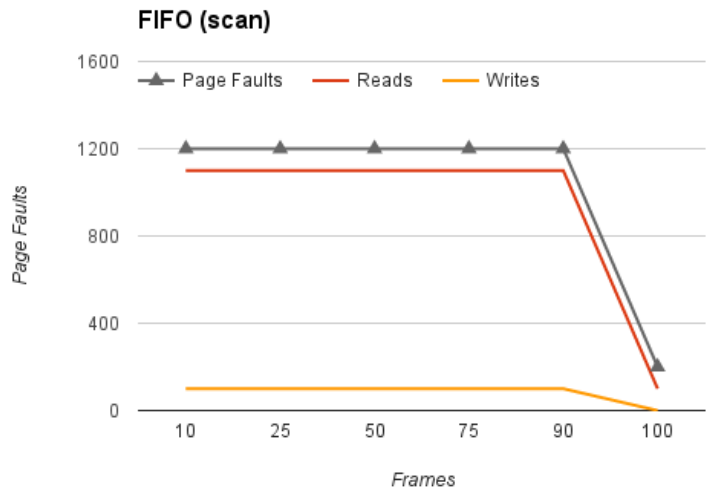
Pages	Frames	Faults	Reads	Writes
100	10	1612	1008	604
100	25	1100	700	400
100	50	800	500	300
100	75	700	400	300
100	90	700	400	300
100	100	200	100	0



Diagrammet foroven viser sort programmet med FIFO.

7.2.2 Scan

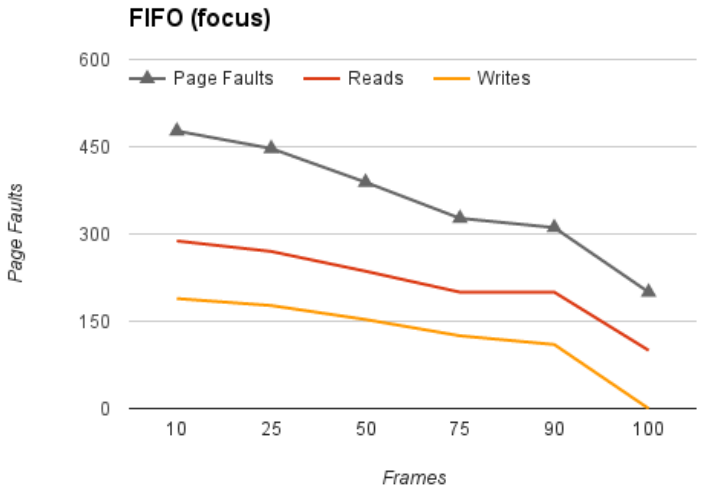
Pages	Frames	Faults	Reads	Writes
100	10	1200	1100	100
100	25	1200	1100	100
100	50	1200	1100	100
100	75	1200	1100	100
100	90	1200	1100	100
100	100	200	100	0



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

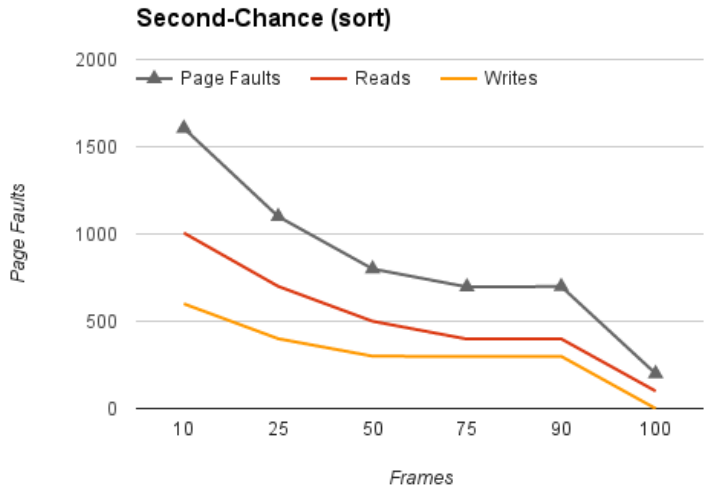


Diagrammet foroven viser focus programmet med FIFO.

7.3 Custom(Second-Chance) udskiftning

7.3.1 Sort

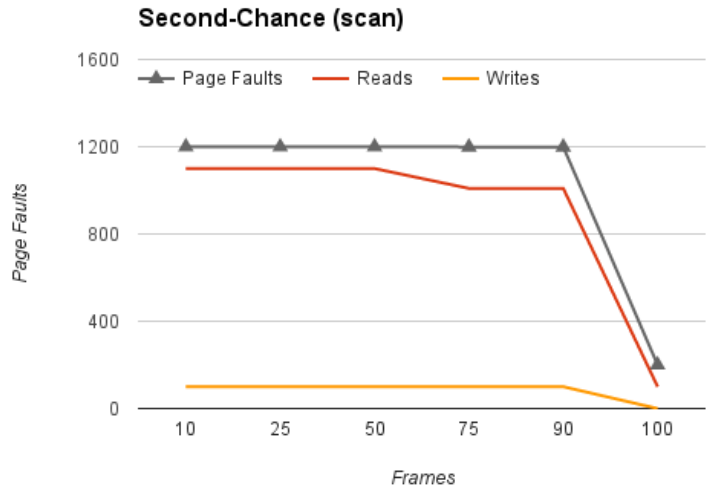
Pages	Frames	Faults	Reads	Writes
100	10	1606	1006	600
100	25	1100	700	400
100	50	800	500	300
100	75	697	398	298
100	90	699	399	299
100	100	200	100	0



Diagrammet foroven viser sort programmet med Second-Chance algoritmen.

7.3.2 Scan

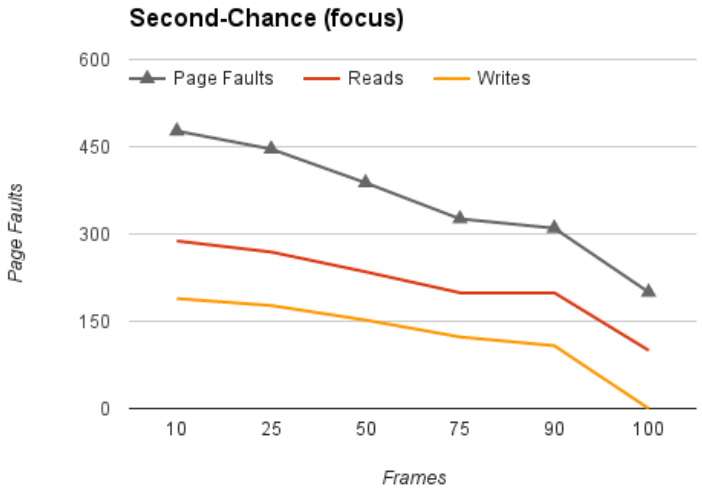
Pages	Frames	Faults	Reads	Writes
100	10	1200	1100	100
100	25	1200	1100	100
100	50	1200	1100	100
100	75	1199	1009	100
100	90	1199	1009	100
100	100	200	100	0



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

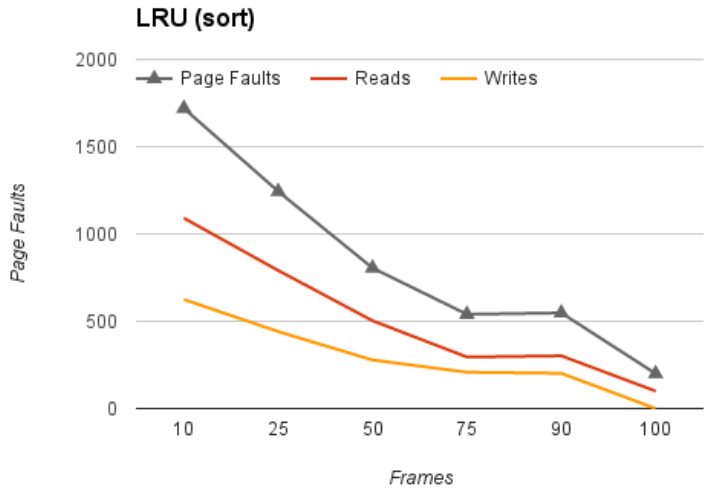


Diagrammet foroven viser focus programmet med Second-Chance algoritmen.

7.4 Custom (LRU) udskiftning

7.4.1 Sort

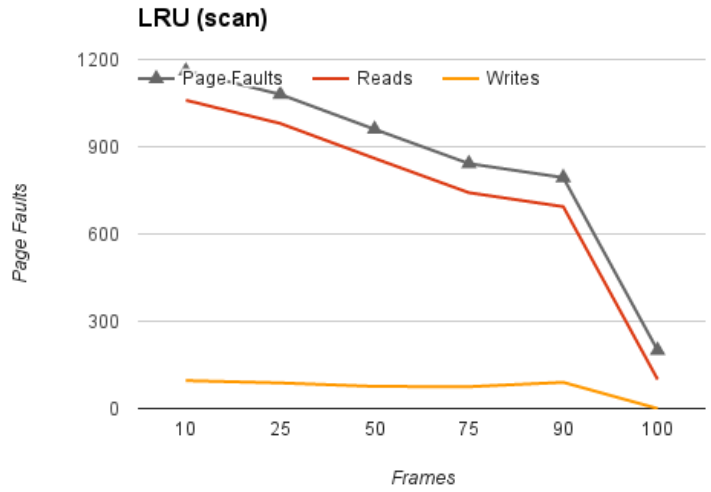
Pages	Frames	Faults	Reads	Writes
100	10	1720	1091	625
100	25	1244	791	441
100	50	805	503	278
100	75	540	295	208
100	90	548	302	202
100	100	200	100	0



Diagrammet foroven viser sort programmet med LRU algoritmen.

7.4.2 Scan

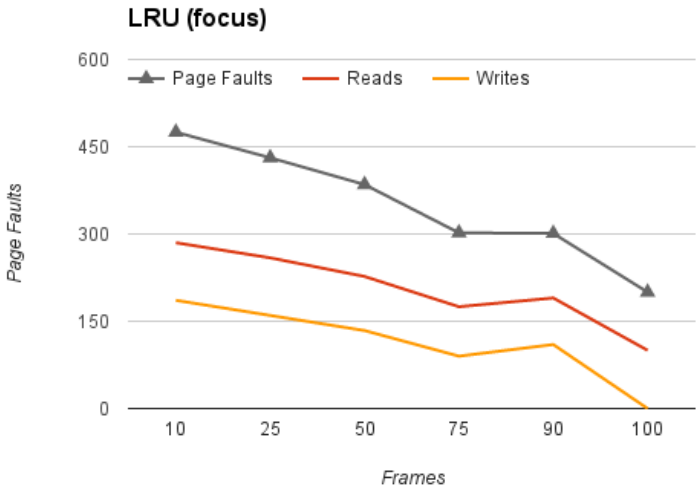
Pages	Frames	Faults	Reads	Writes
100	10	1160	1060	96
100	25	1080	980	88
100	50	960	860	76
100	75	842	742	75
100	90	794	694	90
100	100	200	100	0



Diagrammet foroven viser scan programmet med LRU algoritmen.

7.4.3 Focus

Pages	Frames	Faults	Reads	Writes
100	10	475	285	186
100	25	431	259	160
100	50	385	227	134
100	75	302	175	90
100	90	301	190	110
100	100	200	100	0



Diagrammet foroven viser focus programmet med LRU algoritmen.

8 Appendix B - Sourcecode

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

char *physmem;

struct disk *disk;
int npages, nframes;
int *loaded_pages, *clock;
int pageswap, fifo_counter, fault_counter = 0, write_count = 0,
    read_count = 0;

void print_second_chance()
{
  int i;
  printf("+---+---+\n");
  for(i = 0; i < nframes; i++)
  {
    if(fifo_counter == i)
    {
      printf("| %d | %d | <-\n", loaded_pages[i], clock[i]);
    }
    else
    {
      printf("| %d | %d |\n", loaded_pages[i], clock[i]);
    }
    printf("+---+---+\n");
  }
}
```

```
42
43 void get_swap_frame(int *vFrame)
44 {
45   int i;
46   switch(pageswap)
47   {
48     /* random */
49     case 0:
50       *vFrame = lrand48() % nframes;
51       return;
52
53     /* FIFO */
54     case 1:
55       *vFrame = fifo_counter;
56       fifo_counter++;
57       fifo_counter = fifo_counter % nframes;
58       return;
59
60     /* second chance */
61     case 2:
62       //print_second_chance();
63       i = fifo_counter;
64       int do_repeat = 1;
65       while(do_repeat == 1)
66       {
67         /* check if it's reference bit is 0 */
68         if(clock[i] == 0)
69         {
70           do_repeat = 0;
71           *vFrame = i;
72           fifo_counter++;
73           fifo_counter = fifo_counter % nframes;
74         }
75         else
76         {
77           clock[i] = 0;
78           i++;
79           i = i % nframes;
80         }
81       }
82       return;
83
84     /* LRU */
85     case 3:;
86       int frame = 0;
87       for(i = 0; i < nframes; i++)
```

```
88        {
89          if(clock[i] > frame)
90          {
91            frame = i;
92          }
93        }
94        for(i = 0; i < nframes; i++)
95        {
96          if(i == frame)
97          {
98            clock[i] = 0;
99          }
100         else
101         {
102           clock[i] += 1;
103         }
104       }
105       *vFrame = frame;
106       return;
107   }
108 }
109
110 void page_fault_helper(struct page_table *pt, int page, int vPage, int
      vFrame, int flag)
111 {
112   int vFlag;
113
114   /* get the victim flag */
115   page_table_get_entry(pt, vPage, &vFrame, &vFlag);
116
117   /* check for RW flag  */
118   int rw = (PROT_READ|PROT_WRITE);
119   if(vFlag == rw)
120   {
121     /* write victim from physmem to disk */
122     disk_write(disk, vPage, &physmem[vFrame*PAGE_SIZE]);
123     write_count++;
124   }
125
126   /* read from disk to victim frame */
127   disk_read(disk, page, &physmem[vFrame*PAGE_SIZE]);
128   read_count++;
129
130   /* update page table entries */
131   page_table_set_entry(pt, page, vFrame, flag);
132   page_table_set_entry(pt, vPage, 0, 0);
```

```
133
134   /* update loaded_pages */
135   loaded_pages[vFrame] = page;
136
137   /* Second-Chance clock setting */
138   if(pageswap == 2 && flag == PROT_READ)
139   {
140     clock[vFrame] = 0;
141   }
142 }
143
144 void page_fault_handler( struct page_table *pt, int page )
145 {
146   fault_counter++;
147   int flag;
148   int frame;
149
150   /* get frame and flag for the page */
151   page_table_get_entry(pt, page, &frame, &flag);
152
153   //page_table_print_entry(pt,page);
154
155   int i;
156   switch(flag)
157   {
158     case 0:
159       /* check for free frame*/
160       for(i = 0; i < nframes; i++)
161       {
162         if(loaded_pages[i] == -1)
163         {
164           /* read from disk to physmem */
165           page_table_set_entry(pt, page, i, PROT_READ);
166           disk_read(disk, page, &physmem[i*PAGE_SIZE]);
167           loaded_pages[i] = page;
168           read_count++;
169
170           //page_table_print_entry(pt,page);
171           //printf("\n");
172
173           return;
174         }
175       }
176
177       /* variables for victim */
178       int vFrame, vPage;
```

```
179
180       /* get the victim frame */
181       get_swap_frame(&vFrame);
182
183       /* set the victim page */
184       vPage = loaded_pages[vFrame];
185
186       /* call the helper to settle the pages */
187       page_fault_helper(pt, page, vPage, vFrame, PROT_READ);
188
189       //print_second_chance();
190       //page_table_print_entry(pt,page);
191       //printf("\n");
192
193       return;
194     case PROT_READ:
195       page_table_set_entry(pt, page, frame, PROT_READ|PROT_WRITE);
196
197       //page_table_print_entry(pt,page);
198       //printf("\n");
199
200       if(pageswap == 2)
201       {
202         clock[frame] = 1;
203       }
204
205       if(pageswap == 3)
206       {
207         clock[frame] = 0;
208       }
209       return;
210   }
211   printf("page fault on page #%d\n",page);
212   exit(1);
213 }
214
215 int main( int argc, char *argv[] )
216 {
217   if(argc!=5)
218   {
219     printf("use: virtmem <npages> <nframes> <rand|fifo|custom>
          <sort|scan|focus>\n");
220     return 1;
221   }
222
223   npages = atoi(argv[1]);
```

```
224   nframes = atoi(argv[2]);
225   const char *algorithm = argv[3];
226   const char *program = argv[4];
227
228   loaded_pages = malloc(sizeof(int) * nframes);
229   int i;
230   for(i = 0; i < nframes; i++)
231   {
232     /* indicate that there is no pages loaded yet */
233     loaded_pages[i] = -1;
234   }
235
236   disk = disk_open("myvirtualdisk",npages);
237   if(!disk)
238   {
239     fprintf(stderr,"couldn't create virtual disk:
          %s\n",strerror(errno));
240     return 1;
241   }
242
243   struct page_table *pt = page_table_create( npages, nframes,
       page_fault_handler );
244   if(!pt)
245   {
246     fprintf(stderr,"couldn't create page table: %s\n",strerror(errno));
247     return 1;
248   }
249
250   char *virtmem = page_table_get_virtmem(pt);
251
252   physmem = page_table_get_physmem(pt);
253
254   if(!strcmp(algorithm, "rand"))
255   {
256     pageswap = 0;
257   }
258   else if(!strcmp(algorithm, "fifo"))
259   {
260     pageswap = 1;
261     fifo_counter = 0;
262   }
263   else if(!strcmp(algorithm, "custom"))
264   {
265     pageswap = 2;
266     fifo_counter = 0;
267     clock = malloc(sizeof(int) * nframes);
```

```
268     for(i = 0; i < nframes; i++)
269     {
270       clock[i] = 0;
271     }
272   }
273   else if(!strcmp(algorithm, "custom2"))
274   {
275     pageswap = 3;
276     clock = malloc(sizeof(int) * nframes);
277     for(i = 0; i < nframes; i++)
278     {
279       clock[i] = 0;
280     }
281   }
282   else
283   {
284     fprintf(stderr,"unknown algorithm: %s\n",argv[2]);
285   }
286
287   if(!strcmp(program,"sort"))
288   {
289     sort_program(virtmem,npages*PAGE_SIZE);
290
291   }
292   else if(!strcmp(program,"scan"))
293   {
294     scan_program(virtmem,npages*PAGE_SIZE);
295
296   }
297   else if(!strcmp(program,"focus"))
298   {
299     focus_program(virtmem,npages*PAGE_SIZE);
300
301   }
302   else
303   {
304     fprintf(stderr,"unknown program: %s\n",argv[3]);
305   }
306   printf("Faults: %d Reads: %d Writes: %d\n", fault_counter,
        read_count, write_count);
307   page_table_delete(pt);
308   disk_close(disk);
309
310   return 0;
311 }
```

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
       number of blocks.
14 Returns a pointer to a new disk object, or null on failure.
15 */
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

```c
/*
Do not modify this file.
Make all of your changes to main.c instead.
*/

#include "disk.h"

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>

extern ssize_t pread (int __fd, void *__buf, size_t __nbytes, __off_t
    __offset);
extern ssize_t pwrite (int __fd, const void *__buf, size_t __nbytes,
    __off_t __offset);


struct disk {
  int fd;
  int block_size;
  int nblocks;
};

struct disk * disk_open( const char *diskname, int nblocks )
{
  struct disk *d;

  d = malloc(sizeof(*d));
  if(!d) return 0;

  d->fd = open(diskname,O_CREAT|O_RDWR,0777);
  if(d->fd<0) {
    free(d);
    return 0;
  }

  d->block_size = BLOCK_SIZE;
  d->nblocks = nblocks;

  if(ftruncate(d->fd,d->nblocks*d->block_size)<0) {
    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:
          %s\n",block,strerror(errno));
60      abort();
61    }
62  }
63
64  void disk_read( struct disk *d, int block, char *data )
65  {
66    if(block<0 || block>=d->nblocks) {
67      fprintf(stderr,"disk_read: invalid block #%d\n",block);
68      abort();
69    }
70
71    int actual = pread(d->fd,data,d->block_size,block*d->block_size);
72    if(actual!=d->block_size) {
73      fprintf(stderr,"disk_read: failed to read block #%d:
          %s\n",block,strerror(errno));
74      abort();
75    }
76  }
77
78  int disk_nblocks( struct disk *d )
79  {
80    return d->nblocks;
81  }
82
83  void disk_close( struct disk *d )
84  {
85    close(d->fd);
86    free(d);
```

```
87 }
```

page_table.h

```
#ifndef PAGE_TABLE_H
#define PAGE_TABLE_H

#include <sys/mman.h>

#ifndef PAGE_SIZE
#define PAGE_SIZE 4096
#endif

struct page_table;

typedef void (*page_fault_handler_t) ( struct page_table *pt, int page );

/* Create a new page table, along with a corresponding virtual memory
that is "npages" big and a physical memory that is "nframes" bit
 When a page fault occurs, the routine pointed to by "handler" will be called. */

struct page_table * page_table_create( int npages, int nframes, page_fault_handler_t handler );

/* Delete a page table and the corresponding virtual and physical memories. */

void page_table_delete( struct page_table *pt );

/*
Set the frame number and access bits associated with a page.
The bits may be any of PROT_READ, PROT_WRITE, or PROT_EXEC logical-ored together.
*/

void page_table_set_entry( struct page_table *pt, int page, int frame, int bits );

/*
Get the frame number and access bits associated with a page.
"frame" and "bits" must be pointers to integers which will be filled with the current values.
The bits may be any of PROT_READ, PROT_WRITE, or PROT_EXEC logical-ored together.
*/

```

```
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,
        page_fault_handler_t handler )
56  {
57    int i;
58    struct sigaction sa;
59    struct page_table *pt;
60    char filename[256];
61
62    pt = malloc(sizeof(struct page_table));
63    if(!pt) return 0;
64
65    the_page_table = pt;
66
67    sprintf(filename,"/tmp/pmem.%d.%d",getpid(),getuid());
68
69    pt->fd = open(filename,O_CREAT|O_TRUNC|O_RDWR,0777);
70    if(!pt->fd) return 0;
71
72    ftruncate(pt->fd,PAGE_SIZE*npages);
73
74    unlink(filename);
75
76    pt->physmem =
          mmap(0,nframes*PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,pt->fd,0);
77    pt->nframes = nframes;
78
79    pt->virtmem =
          mmap(0,npages*PAGE_SIZE,PROT_NONE,MAP_SHARED|MAP_NORESERVE,pt->fd,0);
80    pt->npages = npages;
81
82    pt->page_bits = malloc(sizeof(int)*npages);
83    pt->page_mapping = malloc(sizeof(int)*npages);
84
85    pt->handler = handler;
86
```

```
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,
      int bits )
109 {
110   if( page<0 || page>=pt->npages ) {
111     fprintf(stderr,"page_table_set_entry: illegal page #%d\n",page);
112     abort();
113   }
114
115   if( frame<0 || frame>=pt->nframes ) {
116     fprintf(stderr,"page_table_set_entry: illegal frame #%d\n",frame);
117     abort();
118   }
119
120   pt->page_mapping[page] = frame;
121   pt->page_bits[page] = bits;
122
123   remap_file_pages(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,0,frame,0);
124   mprotect(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,bits);
125 }
126
127 void page_table_get_entry( struct page_table *pt, int page, int
      *frame, int *bits )
128 {
129   if( page<0 || page>=pt->npages ) {
130     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
157 void page_table_print( struct page_table *pt )
158 {
159   int i;
160   for(i=0;i<pt->npages;i++) {
161     page_table_print_entry(pt,i);
162   }
163 }
164
165 int page_table_get_nframes( struct page_table *pt )
166 {
167   return pt->nframes;
168 }
169
170 int page_table_get_npages( struct page_table *pt )
171 {
172   return pt->npages;
173 }
174
175 char * page_table_get_virtmem( struct page_table *pt )
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

```
/*
Do not modify this file.
Make all of your changes to main.c instead.
*/

#ifndef PROGRAM_H
#define PROGRAM_H

void scan_program( char *data, int length );
void sort_program( char *data, int length );
void focus_program( char *data, int length );

#endif
```

program.c

```c
/*
Do not modify this file.
Make all of your changes to main.c instead.
*/

#include "program.h"

#include <stdio.h>
#include <stdlib.h>

static int compare_bytes( const void *pa, const void *pb )
{
  int a = *(char*)pa;
  int b = *(char*)pb;

  if(a<b) {
    return -1;
  } else if(a==b) {
    return 0;
  } else {
    return 1;
  }

}

void focus_program( char *data, int length )
{
  int total=0;
  int i,j;

  srand(38290);

  for(i=0;i<length;i++) {
    data[i] = 0;
  }

  for(j=0;j<100;j++) {
    int start = rand()%length;
    int size = 25;
    for(i=0;i<100;i++) {
      data[ (start+rand()%size)%length ] = rand();
    }
  }

```

```
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
```