IT University of Copenhagen

Bachelor: Operating Systems and C, Handin for Group A

Virtual Memory

Anders Edelbo Lillie
aedl@itu.dk

Andreas Bjørn Hassing Nielsen
abhn@itu.dk

Markus Thomsen
matho@itu.dk

November 22, 2016

Contents

1 Introduction and background

This paper reports a project regarding virtual memory using the C language, written in November 2016, in the ITU course "Operating Systems and C". The purpose is to understand how virtual and physical memory work together, and how page faults can be handled by employing different page-swapping algorithms. The project description can be found in the appendix, section 7.1.

The overall tasks to be completed are listed below:

- Construct a page table to keep track of pages to be loaded.
- Construct a page fault handler to swap pages when the page table is full.
- Implement the following algorithms for page swapping:
  - First-in-first-out (FIFO) ordered: swap with the least recently swapped page.
  - Random: choose a random page to be swapped.
  - Custom: should access the disk less times than the other algorithms.
- Answer the multi-programming task (see project description page 6).

A code skeleton was handed out with fully functioning logic for the assignment, only with the above mentioned tasks missing. The only code implemented in the project is found in the `main.c` file. A template for this file was given, handling argument-parsing and program calls for the `main` function. The program code can be found in appendix, section 7.2 and `main.c` in subsection 7.2.2.

2 Problem analysis

In this section, it is discussed how solutions to the different tasks could be constructed, such that they fulfill the requirements listed in the formerly mentioned project description.

2.1 The page fault handler

A page fault handler is called in the case that a page fault occurs. Page faults come in two forms: either the virtual memory accessed was not found in the physical memory, i.e. a page miss, or the virtual memory accessed did not have the proper protection bits set for the wanted operation.

In the case of a page miss, the given page should be swapped into memory, and another page should be swapped out, unless the physical memory has a free frame. The decision on which page to remove from physical memory should be handled by a page swapping algorithm, which is discussed in section 2.2 below. When a page to swap with has been found, it should be saved to disk if the memory is dirty, otherwise it will simply be overridden. Dirty memory means that the variable in a page has been altered since being read from disk, thus it will need to be written to disk when swapped, to keep the variable value consistent throughout the lifetime of the program.

In the case of improper protection bits, one should simply set the bits to `PROT_READ | PROT_WRITE`, as the page fault that occurs will be a miss if the `PROT_READ` bit has not already been set on the page, thus it can be concluded that this page fault must have been due to improper protection bits.

This means that there are actually three cases a page fault handler must support; alter protection bits, insert page into free physical memory, and finally swap a page from physical memory with a page from the disk.

The page fault handler could be implemented with the logic found in the following pseudo-code:

```
func page_fault_handler(page) {
    if (page is readonly)
        set write flag
    else {
        if (free frame available) insert page
        else swap page
    }
}
```

A page table data structure is updated on each page fault, such that when a page is swapped into memory, the page table will contain the current permission bits for that page. Only having this data structure however would force each page-swapping algorithm to verify that the selected page is in physical memory, and if not, select another. This can take $\Theta(N)$ time, where N is the count of pages. As $N \geq F$, F being the count of frames, having a data structure that allows reverse look-ups from frames to the page table would seem beneficial.

2.2 Page swapping algorithms

Several different algorithms for page swapping are developed and used in different operating systems, e.g. Linux uses the least-recently-used (LRU) scheme[1], which basically keeps track of when the pages are used/accessed, and thereby swapping out the "oldest" page in memory. This project requires the 3 algorithms mentioned in the introduction, section 1, to be implemented.

The FIFO algorithm is simple in the sense that the basic requirement is to know which page was loaded into memory first, when memory is full. This could be implemented with a linked list structure of the frames

[1] http://www.thegeekstuff.com/2012/02/linux-memory-swap-cache-shared-vm/ (Accessed Tuesday, November 15)

in physical memory, *adding* pages to the end of the list and *removing* pages from the head of the list, thereby always choosing the head of the list to be swapped out. This does however introduce some overhead as there must be a certain structure of the list with links between the pages, and a pointer to the head and tail. The frames as an array could be used instead of a list structure. Starting with the index of the first frame in memory, a function could increment a counter to know the index of the next frame to be swapped, and perform some modulus operation on the counter to the value of the count of total frames. This implementation would require little code and less overhead in the incremented integer than would have been with the list structure.

The random algorithm is also a simple implementation. Having some function find a random number in the range from 0 to the number of frames in physical memory, and using that frame index to select for swapping. One could then discuss which RNG[2] function to use, but that might be irrelevant as long as the output of is *random*.

The custom algorithm's only requirement is to be more efficient than the other two algorithms in regards to disk access, i.e. read/write operations, and essentially page faults. As mentioned above, many popular and efficient algorithms exist to do this job, but from the project description it is derived that only the `main.c`-file is "allowed" to be modified, thus disallowing any modifications/extensions to e.g. simple memory access operations. This means e.g. that one cannot keep track of pages that are accessed and "used", which the LRU algorithm requires. The only "handle" in this case is when page faults occur. One can think of many ways to construct an algorithm to choose a page to be swapped from physical memory. The difficult thing is to make it more efficient than especially the "random algorithm". You would intuitively go for an algorithm, which makes sure pages most used stay in memory, thus minimizing page faults. This could be done by keeping track of the least swapped page, and always using this to swap. This algorithm however depends much on the program's operations on the data. This observation is discussed further in the conclusion, section 6. Another discussed custom algorithm to make up for the disadvantages of the "least-swapped" algorithm, is the "most-swapped" algorithm, which quite opposite would choose the most swapped page to be swapped. This would however always choose the same page to swap, if no guard prevents two same-frame consecutive swaps. If guarded against this, the property of choosing the most swapped would still favor distinct pages, resulting in many page faults and poor performance.

2.3 Multi-programming

As mentioned in the introduction, section 1, a requirement to the project, was to also consider what it would take for more than one program to use the physical memory concurrently. The main problem to this question is race conditions in regards to writing to frames and swapping out frames. One program could be writing to a frame in memory, while another program writes to it before the first program terminates its execution. The result would be overwritten data stored in the frame. Likewise, if one program is accessing a frame in memory, while another one swaps it with a page from the disk, the program could crash.

Above mentioned problems could first and foremost be solved by putting locks on all critical sections in the data structure. A `mutex` lock could for example be put around the page fault handler, thus only allowing sequential operations of swaps, allocation and modification of protection bits. Using the pseudo code from section 2.1, it would be implemented like the example below. Of course it would also be necessary to have synchronized writing operations for data in memory.

```
global mutex
func page_fault_handler(page) {
    lock(mutex)
    if (page is readonly)
        set write flag
    else {
```

[2]Random Number Generator, a science in itself on deterministic machines.

```
        if (free frame available) insert page
        else swap page
    }
    unlock(mutex)
}
```

Another way to solve it could be to divide the address space and allocate each portion to each program. This would allow concurrency, granted that the programs don't operate on the same input data set. This is depicted in figure 2.3 below, which illustrates that no problems occur when the data and memory is split.

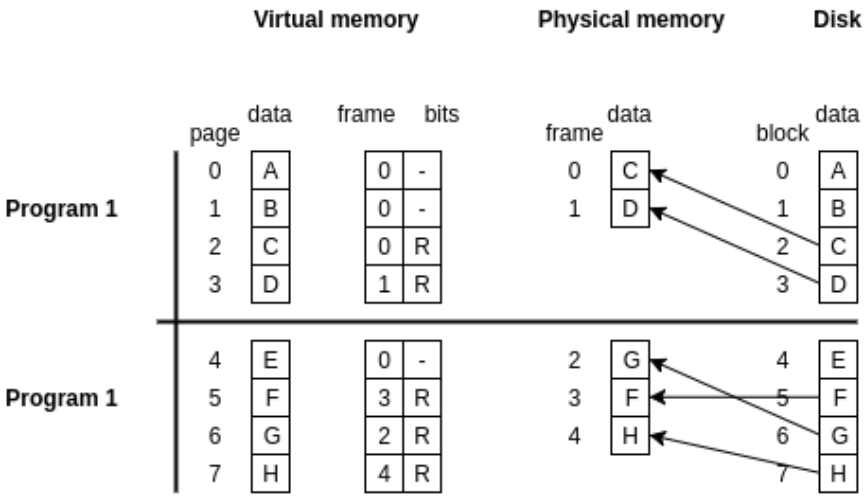


Figure 1: Illustration of divided address space for multi-programming

3 Examples

In this section, the functionalities of the program and the algorithms implemented therein are described and illustrated through some examples.

3.1 virtmem

Running `virtmem` from a console window will yield the result of running the specified program with the selected page swapping algorithm.

An example of running `virtmem` with the `sort` program, and each of the three page swapping algorithms, can be seen below. The number of pages and frames are 100 and 10 respectively:

```
$ ./virtmem 100 10 custom sort
sort result is -311357
$ ./virtmem 100 10 rand sort
sort result is -311357
$ ./virtmem 100 10 fifo sort
sort result is -311357
```

Furthermore, a shell script is constructed, which runs all the programs with each swapping algorithm, and prints statistics to `stdout`. The program takes 1 argument, being the number of frames, which makes it ideal for testing purposes. The script can be found in appendix, section 7.2.9, and is run like below, 10 being the number of frames (number of pages is set to 100):

```
./run_tests.sh 10
```

3.2 virtmem with statistics

To get statistics along with the output of `virtmem`, the program should be compiled with the `stats` make-target:

```
$ make stats
gcc -DSTATS -Wall -g -c main.c -o main.o
gcc -DSTATS -Wall -g -c page_table.c -o page_table.o
gcc -DSTATS -Wall -g -c disk.c -o disk.o
gcc -DSTATS -Wall -g -c program.c -o program.o
gcc -DSTATS main.o page_table.o disk.o program.o -o virtmem
$ ./virtmem 100 10 custom sort
sort result is -311357
Page faults: 3468
Disk reads: 2536
Disk writes: 921
```

4 Technical description of the program

In this section, the specified functionality is described with a technical description, both covering how they work, and how they are implemented in the source code.

4.1 Page-fault handler

When a page fault occurs, the page fault handler is called. A page fault occurs in the following scenarios:

1. Attempted to write to a page in memory, that only has a read permission bit.
2. Attempted to read a page in memory, that did not exist in the physical memory.

The first of the above scenarios is implemented as such: allow the page to be written to, by setting the permission bit to `PROT_READ | PROT_WRITE`. When the program regains control it can write to the page again without fault.

The second scenario consist of two cases: Either the program is just initialized, in which case physical memory is "free", or there are no empty frames. The former is handled by setting the given page table entry to point to an empty frame, updating the protection bits to `PROT_READ`, and returning control to the program. The latter is handled by running the page swapping algorithm, which swaps some page from physical memory to disk, and then swaps the wanted page from disk into memory. This is described in more detail in section 4.2 below.

A "reverse look-up table" is implemented to support the operations of swapping. This includes the `frame_state` struct, and a global variable containing the frame states in an array: `struct frame_state *frame_states;`. This data structure is updated every time a page is swapped in and out of physical memory, and allows the page-swapping algorithms to return a frame number to swap, rather than doing the swapping then and there. As discussed in section 2.1, the reverse look-up table, mapping frames to their respective pages, actually increases the effectiveness of the search for a page swap candidate from $\Theta(N)$ to $\Theta(F)$ (N being the number of pages, and F being the number of frames), which, although still linear, is faster than looking through the page table (unless $N = F$, but come on, we're not savages).

The page swapping function calls a function defined as a type and a global variable as such:

```
typedef int (*frame_select_algo_t) (struct page_table *pt);
frame_select_algo_t frame_select_algo;
```

This function variable is set in the `main` function, as specified by the user of the program, and is in fact a pointer to one of the page-swapping algorithms. This design yielded more readable code and a lower degree of coupling.

4.2 Page-swapping algorithms

Three page-swapping algorithms were implemented for the project. The first two had their functionality defined in advance, those being the random page-swapping algorithm and a page-swapping algorithm employing a first in, first out principle. The last one is a page-swapping algorithm by our own design, referred to as the custom algorithm.

Each page-swapping algorithm takes the page table as parameter, and returns the frame number it nominates to be swapped out. A general page-swap function is implemented that calls the wanted page-swapping algorithm to find a frame to swap with, and then completes the page-swap with that frame.

4.2.1 Random

The random page-swapping algorithm is simple in its approach. Whenever a page-fault occurs, it merely picks a random frame, and swaps the page in that frame with the requested one. The implementation is really just

one line of code as illustrated below. `lrand48()` generates non-negative long integers uniformly distributed between 0 and 2^{31} [3], and `nframes` represents the number of frames in physical memory:

```
return lrand48() % nframes;
```

Using the modulo operator (%), the number returned will always be within the range of available frames.

4.2.2 FIFO

The FIFO page-swapping algorithm employs a FIFO queue to decide which page to swap when a page fault occurs. The implementation of the algorithm doesn't implement an actual queue data structure, but instead uses an incremental queue pointer, to increase performance. This is implemented as illustrated below. `current_frame` is a static counter, and again, `nframes` represents the number of frames in physical memory and the modulo operator assures the range is within the number of physical memory frames.

```
current_frame = (current_frame + 1) % nframes;
```

4.2.3 Custom

The designed custom algorithm keeps in memory the amount of times that a page has been swapped out of physical memory via page faults (in the array `int page_swap_count[n_pages]`). This data is used to find the least swapped page in the frames, and then swapping that one out. The implementation is not illustrated here, as the function is roughly 25 lines of code, but can be found in appendix, section 7.2.2, with the function signature `get_custom_frame(..)`.

This approach favors pages that have been swapped out more often than not, which means that they stay in memory longer than pages that have not been swapped out as much. The idea is that if a page needs to be swapped out often, it must also be read from memory often, thus it should stay in memory longer than other pages.

The custom algorithm is conservative, but only because it avoids using the previously swapped frame, thereby negating a page fault cycle that occurs if two entries are required in memory at the same time, and both have the lowest value in `page_swap_count` of all the pages in physical memory. The cycle would break once the page swap counts of the two pages becomes high enough, but useless swaps would then have been committed in the meantime.

[3] https://linux.die.net/man/3/lrand48

5 Test

Having described how the programs could be implemented, their functionality through examples, as well as their technical descriptions, finally this section shows to what extend the program solves the given problems.

5.1 Requirements validation

Refer to the examples in section 3 for runs of the program that validates the output of the different page-swapping algorithms with each other.

5.2 Performance comparison

Below are graphs comparing the page swapping algorithms to one another, for each of the programs found in the hand-out (`sort`, `scan` and `focus`). The ranges used; N-pages 100, N-frames from 10-100, are expected to have been tested, as specified in the assignment hand-out's minimum demands. To create these graphs, as described in section 3 a shell script was created along with a modified version of the program, tabulating the page-fault statistics output, rather than pretty printing. This work can be found in a separate branch[4]. The tabularized version let us copy and paste the program output and insert it into a spreadsheet for further analysis and graph generation.

The result of the tests is discussed in the conclusion.

[4] As the GitHub repository is private, a diff of the branch compared to the master branch can be found in appendix section 7.2.10, and a modified version of the `run_tests` script can be found in appendix section 7.2.10

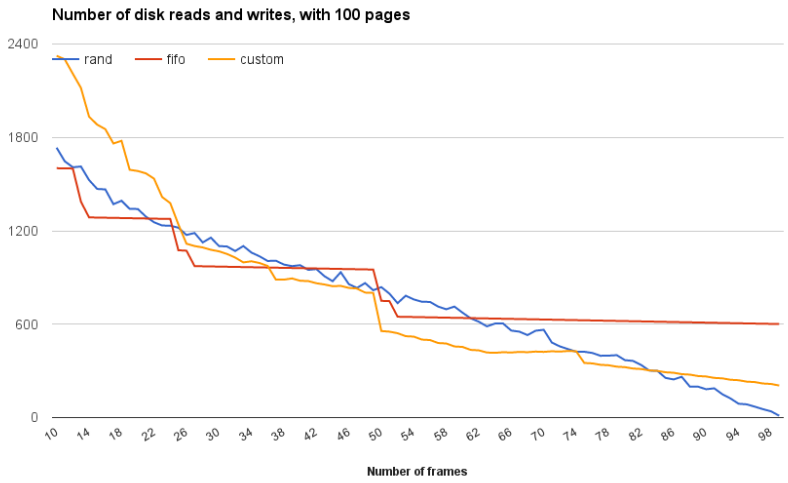


Figure 2: Comparison of (diskreads + diskwrites) for the `sort` program. 100 pages, and increasing amounts of frames along the x-axis.

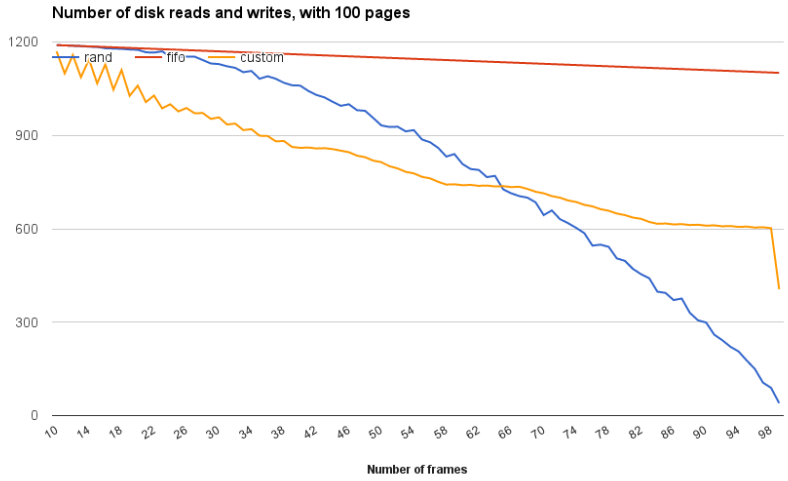


Figure 3: Comparison for the `scan` program.

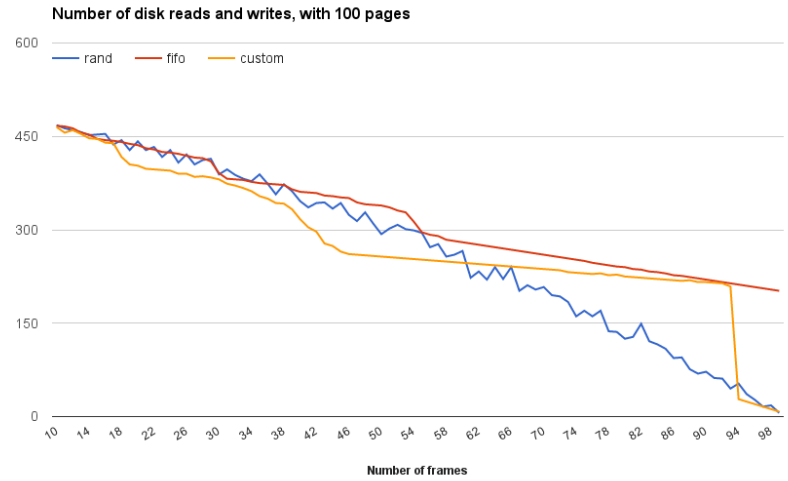


Figure 4: Comparison for the `focus` program.

6 Conclusion

The tasks have been completed, and the program functions as expected, fulfilling the project requirements. Although not much code was needed to implement the solution, most of the time spent was put into discussing approaches to building the page table and the swapping algorithms, and in particular the custom algorithm. This section concludes the performance of the custom algorithm implemented, as well as notes what *could* have been done differently, and finally what *can* be done to improve the program.

6.1 Algorithm performance

The performance difference of the page-swapping algorithms depends on the program they run, and the page to frame ratio the algorithms have available. This subsection concludes on the graphs presented in section 5.2

When running the **focus** program, which has an arbitrary memory access pattern, the different algorithms should approach each others performance, as more memory is accessed. This is not the case with the pre-specified random seed in use in the program, but if the random function was re-seeded with a new seed every time, the performance of the algorithms should be indistinguishable.

When running the **scan** program, the performance difference becomes much more noticeable. This is because the scan program has a linear memory access pattern. The FIFO algorithm performs dreadfully, as it hits a page-fault on every new memory access, after the physical frames are full. It will do this as long as `nframes` < `npages`. The random-algorithm performs almost as bad as the FIFO algorithm, when it has access to few `nframes`, but the number of disk accesses decrease exponentially, as `nframes` approaches `npages`. The custom algorithm performs better than the FIFO and random algorithm to begin with, but is overtaken by the random algorithm around `nframes` = `npages`*2/3.

The **sort** program has an interesting memory access pattern. Memory is accessed pseudo-randomly, that is, accessed in the random order of the input parameter. However, every time an element is accessed, it will be less likely to be accessed in the future, as it becomes more likely that the element is in its sorted state (stale). The FIFO algorithm performs remarkably well with the sort program. This makes sense, as every time an element is accessed, it's less likely to need to be accessed again. The random algorithm performs as in the other cases, steadily reducing diskreads- and writes as `nframes` approaches `npages`. The custom algorithm intially performs terribly, producing a lot of disk reads- and writes at a low `nframes` to `npages` ratio. However, as the number of `nframes` increases, the chance of having an element in memory do so as well. This is a product of "quicksort"'s locality, which keeps sorting smaller and smaller partitions, and as such, will usually access a subset of the elements multiple times.

6.2 Retrospective

Not much would have been done differently in retrospect. It is worth mentioning however that more time could have been spent comparing the current custom-algorithm with alternatives. It would have been interesting to implement a "second-chance" algorithm, which would select a frame that only has the read protection bit set, and set everything else in physical memory to read-only. This would allow frames with write protection to have a *second chance*. Furthermore, it would have been interesting to "mix" this implementation with the current one, by utilizing the "least-swapped" concept, to select the frame from the set of pages that only has the read protection bit set.

6.3 Extensions and improvements

The most impacting improvements to the program can be made in the swapping algorithms. As mentioned in section 6.1, the performance of the algorithms vary greatly depending on the programs they are run with. The current implementation of the project only takes page faults into account, for each program, whereas it could have been interesting and beneficial to also know *when* the programs access pages, without page faults. This

statistic would certainly improve the custom swapping algorithm. Instead of keeping track of least *swapped* pages, the algorithm could swap out those pages least *read*. The projects description dictated however that no changes could be made to any files but `main.c`, as mentioned in section 1.

Another aspect to keep in mind is the performance of the actual algorithms. Not looking at disk writes and disk reads; is the random-algorithm's execution considerably *faster* than that of the custom-algorithm's? If this is the case, one would strive to improve the running time of the custom-algorithm to speed up the overall swapping of pages.

7 Appendix

7.1 Project description

Intentional blank space, as a separate PDF file is included in the report, and begins on the next page.

IT-Universitetet i København 8. november 2016

Operativsystemer og C

Obligatorisk Opgave 3: Virtuel Hukommelse

Rapporten - som **pdf-fil** - samt kode skal pakkes og uploades til LearnIT
Senest onsdag den 23. November, klokken 23:59
Baseret på original af Prof. Douglas Thain ved University of Notre Dame

---

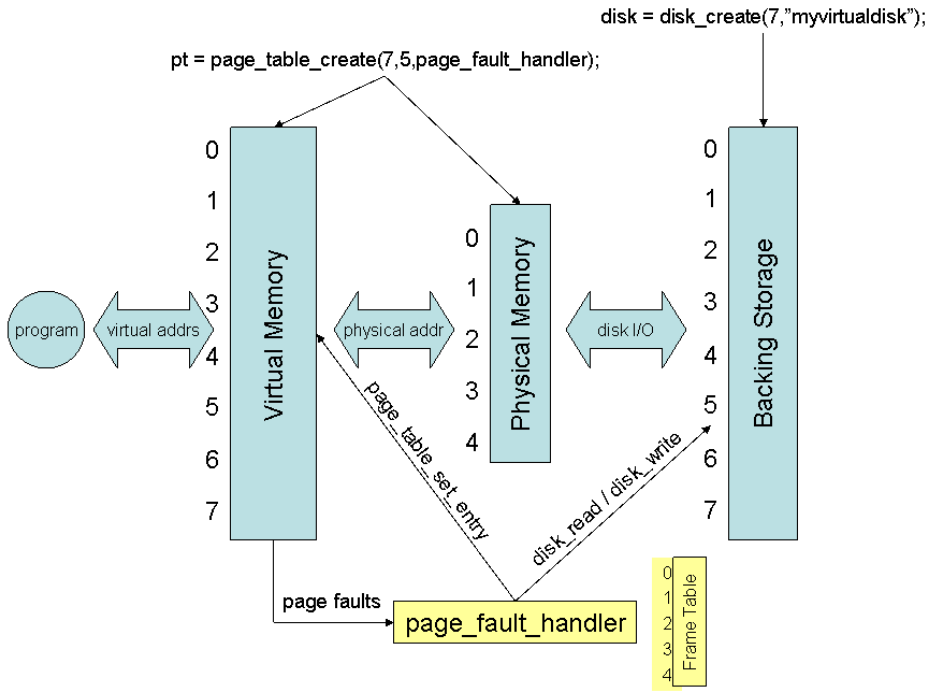
Baggrund

Målet med denne opgave er at udvikle en dyb forståelse for og implementere virtuel hukommelse samt signal håndtering.
I dette projekt skal du implementere en simpel men fuldt funktionel virtuel hukommelse baseret på demand paging. Virtuel hukommelse er som oftest implementeret i operativ systemets kerne uden adgang for brugeren, men det er faktisk også muligt at implementere det på bruger niveau. Det er denne teknik som bruges i moderne virtuelle maskiner, så du vil faktisk lære en brugbar og avanceret teknik uden at skulle have hovedpinen med at arbejde med kerne kode.
Figur 1 giver et overblik over de elementer der indgår i opgaven. Vi giver dig koden til en virtuel side tabel og en virtuel disk (disken er virtuel fordi den ligger i hukommelsen). Når du opretter en virtuel side tabel vil der både blive oprettet en virtuel og en fysisk hukommelse. Du har endvidere adgang til metoder til at opdatere indgange i side tabellen og deres beskyttelses bits. Når et program tilgår den virtuelle hukommelse er det implementeret sådan at det resulterer i en side fejl som bliver fanget og givet til en speciel page handler som du skal implementere. Det betyder generelt at du skal opdatere side tabellen og/eller flytte data frem og tilbage imellem disken og den fysiske hukommelse. Når du er nået i mål med denne del skal du implementere tre sideudskiftningsalgoritmer og dokumentere deres performance samt overveje en mulig udvidelse med multiprogrammering.

Opvarmning

Du skal starte med at hente kilde koden fra LearnIT kløgtigt navngivet oo3.zip, pakke den ud og bygge programmet (hint: kræver Linux og make). Kig nu i main.c og dan dig et overblik over koden. Bemærk at programmet bare laver en virtuel disk og en side table og derefter forsøger at køre en af de medfølgende tre programmer ved brug af den netop oprettede virtuelle hukommelse. Du kan endda køre hele programmet:

```
./virtmem 100 10 rand sort
```



Figur 1: En skematisk oversigt over elementerne i det virtuelle hukommelsessystem som skal implementeres i denne opgave.

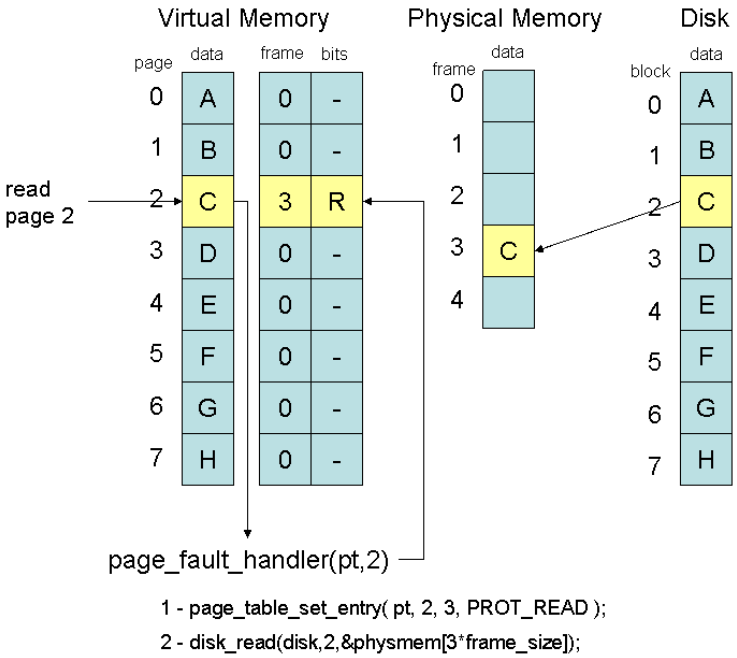
Da du ikke har lavet din opgave endnu og der dermed ikke er koplet noget fysisk hukommelse til den virtuelle hukommelse opstår en side fejl med det samme:

```
% page fault on page #0
```

Nu går vi lige så roligt igang med opgaven. Hvis du kører programmet med et identisk antal fysiske og virtuelle sider har vi ikke brug for disken. Du skal blot få N fysiske sider til at pege direkte på N virtuelle sider. Det gøres ved at indsætte nedenstående i page fault handleren i *main.c*.

```
page_table_set_entry(pt,page,page,PROT_READ|PROT_WRITE);
```

Hver gang der spørges efter en side der ikke er set før vil der genereres en side fejl som fra vores kode kalder din page fault handler overfor. Din page fault handler vil nu lave en direkte mapping fra den virtuelle hukommelse side nummer *page* til den fysiske hukommelse side nummer *page*. Flagene oplyser operativ systemet om at vi både har læst og skrevet til den netop oprettede virtuelle side (hvilket ikke er tilfældet endnu men er et lille hack for at få det til at køre indtil du har lavet opgaven). Hvis vi nu kører programmet skulle det gerne køre og give *page* antal side fejl. Du er nu godt på vej.

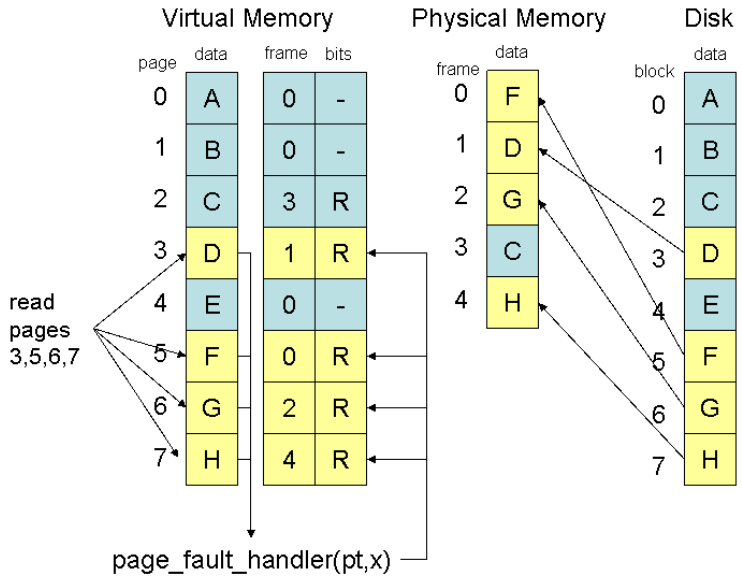


Figur 2: Indlæsning af logisk side 2 fra disken til fysisk side 3.

Illustrativt eksempel

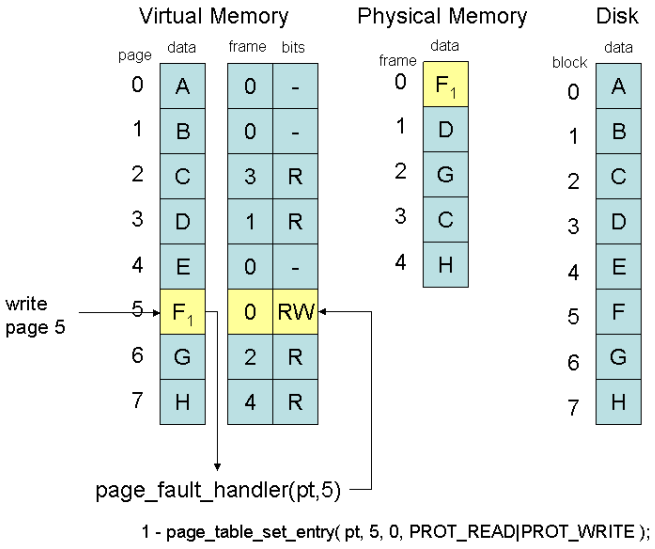
Hvis der er færre fysiske sider end virtuelle virker ovenstående ikke. Du er istedet nødt til at holde de sider som er brugt fornyeligt i hukommelsen og have resten på disken. Du skal også opdatere sidetabellen i takt med at siderne bliver flyttet frem og tilbage. Lad os kigge på et eksempel illustreret i figur 2 og se hvordan dette burde virke. Antag at vi ikke har noget i den fysiske hukommelse (eller den virtuelle). Hvis programmet starter med at forsøge at læse virtuelle side 2 vil det give en side fejl. Page fault handleren vælger nu en ledig fysisk side f.eks. side 3. Den opdatere så side tabellen så indgang 2 peger på side 3 og sætter **kun** læse flaget. Den læser nu side 2 fra disken ind på side 3 (du kan med fordel tage et kig på *disk.h* for at se hvilke operationer du kan bruge på den virtuelle disk.). Når page fault handleren er færdig, køres læse operationen igen og denne gang lykkedes den.

Vi forsætter i figur 3 hvor vi antager at programmet kører videre og læser siderne 3, 5, 6 og 7 som via side fejl bliver læst ind i den fysiske hukommelse. Lad os så antage at den fysiske hukommelse er fyldt.
I figur 4 prøver programmet at skrive til virtuel side 5. Den er i den fysiske hukommelse, men den har kun læse flaget sat hvilket derfor resulterer i en side fejl. Det

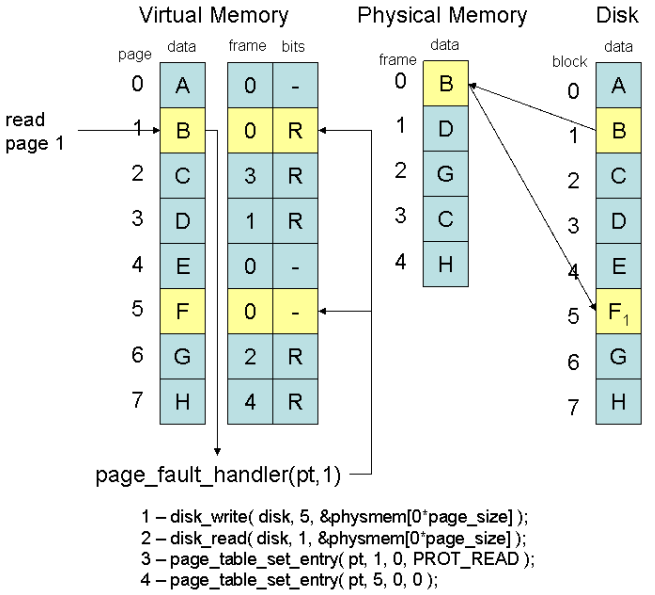


Figur 3: Indlæsning af logisk side 3,5,6, og 7 fra disken til henholdsvis fysisk side 1, 0, 2, og 4 vha. page fault handleren.

ordnes snildt i page fault handleren ved også at sætte skrive flaget. Efter page fault handleren er færdig kører programment fint videre.
Nu kører programmet videre og læser virtuel side 1 illustreret i figur 5. Denne side er ikke på nuværende tidspunkt i den fysiske hukommelse. Nu må page fault handleren beslutte hvilken at de fysiske sider der skal fjernes. Lad os antage at den vælger virtuel side 5, svarende til fysisk side 0. Vi ved at side 5 er blevet ændret fordi den har skrive flaget sat. Page fault handleren må nu skrive side 5 tilbage til disken og læse side 1 ind i dens sted. To indgange i side tabellen skal nu tilrettes for at det hele passer og er konsistent.



Figur 4: Skrivning til virtuel side 5 og resulterende metode kald.



Figur 5: Udskiftning af virtuel side 5 med 1 og resulterende metode kald.

Multiprogrammering

Dette spørgsmål kan med fordel besvares efter resten af opgaven er løst.
Ovenfor antager vi at kun et program bruger hukommelsen. Det er jo ikke realistisk idet flere programmer typisk skal køre samtidigt i et operativ system. Du bedes skitsere (ikke programmere) vha af f.eks. tekst, figurer, eller pseudo kode hvilke ændringer det kræver i det ovenfor beskrevne system for at flere programmer kan bruge den fysiske hukommelse samtidigt.

Hints

- Bemærk at du selv skal parse tekststrengen fra kommandolinjen der angiver hvilken sideudskiftningsalgoritme der skal bruges.
- Det er nødvendigt at oprette nogle globale variable.
- Bemærk at man ikke direkte kan detektere om der læses eller skrives til en side, men hvis rettighedsflagene PROT_READ og PROT_WRITE er sat rigtigt kan man udlede det.
- Tilsvarende kan rettighedsflagene bruges til at afgøre om en side overhovedet er indlæst i hukommelsen.
- Hvis du skal bruge tilfældige tal så brug lrand48() og ikke rand() (gør det nu bare).
- Du må/skal kun rette i *main.c*.

Mindste krav

Du skal kunne starte dit program som vist:

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

npages er antallet af virtuelle sider, $nframes$ er antallet af fysiske sider. Bemærk iøvrigt at i koden bruges konsekvent *pages* i forbindelse med virtuelle sider og $frames$ i forbindelse med fysiske sider. Du skal implementere *rand* som er tilfældig side udskiftning, $fifo$ (first-in-first-out), og endeligt *custom* som er din egen sideudskiftningsalgoritme. *custom* skal tilgå disken færre gange end de andre to. Det sidste argument specificerer hvilket program der skal køres og er implementeret. Programmerne har forskellige hukommelsestilgangsmønstre for at stresse sideudskiftningsalgoritmerne.
Et færdigt og korrekt program kører alle programmerne og laver en linjes output fra hvert program plus en oversigt over antal side fejl, disk læsninger, og disk skrivninger. Brug 100 virtuelle sider og test med en række valg af antal fysiske sider imellem 1 og 100.

Opgave

En mulig tilgang til opgaven kunne være

1. Lav en sidetabel til at holde styr på hvilke sider der loades.
2. Lav en page fault handler som skifter sider ind og ud når sidetabellen er fuld (sidetabellen er fuld når alle indgange har PROT_READ sat).
3. Tag højde for skrivninger og gem ændringer på disk ved side skift.
4. Implementer standard algoritmerne.
5. Implementer custom algoritmen.
6. Svar på multiprogrammeringsopgaven

Rapport

Du skal igen lave en fuld rapport der kan læses som et selvstændigt dokument. I rapporten skal du beskrive hvordan du har implementeret punkterne der spørges til i opgaverne. Desuden skal du redegøre for hvordan du har testet at disse fungerer efter hensigten, evaluere deres performance og diskutere resultater. *main.c* skal inkluderes i et appendix.

7.2 Code

7.2.1 Makefile

```
CCFLAGS=

all: virtmem

debug: CCFLAGS += -DDEBUG -g
debug: virtmem

stats: CCFLAGS += -DSTATS
stats: virtmem

virtmem: main.o page_table.o disk.o program.o
	gcc $(CCFLAGS) main.o page_table.o disk.o program.o -o virtmem

main.o: main.c
	gcc $(CCFLAGS) -Wall -g -c main.c -o main.o

page_table.o: page_table.c
	gcc $(CCFLAGS) -Wall -g -c page_table.c -o page_table.o

disk.o: disk.c
	gcc $(CCFLAGS) -Wall -g -c disk.c -o disk.o

program.o: program.c
	gcc $(CCFLAGS) -Wall -g -c program.c -o program.o

.PHONY: clean

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

7.2.2 main.c

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

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

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

// Variables for statistics
#ifdef STATS
int npage_faults;
int disk_reads;
int disk_writes;

void print_stats() {
        printf("Page faults: %d\n", npage_faults);
        printf("Disk reads: %d\n", disk_reads);
        printf("Disk writes: %d\n", disk_writes);
        printf("Disk reads+writes: %d\n\n", disk_reads+disk_writes);
}
#endif

/**
 * disk facilitates global access to read and write operations on
 * the disk, backing the virtual memory.
 */
struct disk *disk;

/**
 * frame_state contains the bits of a frame (0 if empty), and the
 * id of the page loaded into that frame.
 */
struct frame_state {
        int bits;
        int page;
};

struct frame_state *frame_states;
```

```c
/**
 * Defines an interface for the frame selection algorithms to use.
 * Allows using a global variable containing the selected function,
 * and reduces the amount of switch or if/else blocks needed.
 * @see get_random_frame(struct page_table*)
 */
typedef int (*frame_select_algo_t) (struct page_table *pt);

frame_select_algo_t frame_select_algo;

/**
 * Get an empty frame index, or -1 if no such frame exists.
 * @param nframes number of available frames, empty or not.
 * @return The index of the empty frame, or -1 if no such frame exists.
 */
int get_empty_frame(const int nframes) {
        static int used_frames = 0;
        if (used_frames >= nframes) return -1;

        used_frames++;

        for (int i = 0; i < nframes; i++) {
                if (!frame_states[i].bits) return i;
        }

        return -1;
}

/**
 * Get a random frame and its corresponding page from memory.
 * @param pt the page table to select from.
 * @return the frame number to swap with.
 */
int get_random_frame(struct page_table *pt) {
        int nframes = page_table_get_nframes(pt);

        return lrand48() % nframes;
}

/**
 * Get a frame and its corresponding page from memory, using a FIFO queue approach.
 * @param pt the page table to select from.
 * @return the frame number to swap with.
 */
int get_frame_from_queue(struct page_table *pt) {
        static int current_frame = -1;
        int nframes = page_table_get_nframes(pt);
```

```
        current_frame = (current_frame + 1) % nframes;

        return current_frame;
}

/**
 * Get a frame and its corresponding page from memory, using a custom approach.
 * @param pt the page table to select from.
 * @return the frame number to swap with.
 */
int get_custom_frame(struct page_table *pt) {
        static unsigned int *page_swap_count = NULL;
        static int last_swapped_frame = -1;

        if (page_swap_count == NULL)
                page_swap_count = calloc(page_table_get_npages(pt), sizeof(int));

        int nframes = page_table_get_nframes(pt);

        int least_swapped_page = 0, least_swapped_page_frame = 0;
        for (int i = 0; i < nframes; i++) {
                // avoid page fault cycles when trying to compare two entries
                if (i == last_swapped_frame) continue;
                int swap_count = page_swap_count[frame_states[i].page];
                if (swap_count < page_swap_count[least_swapped_page]) {
                        least_swapped_page = frame_states[i].page;
                        least_swapped_page_frame = i;
                }
        }

        page_swap_count[least_swapped_page]++;
        last_swapped_frame = least_swapped_page_frame;

        return least_swapped_page_frame;
}

/**
 * Swap a page from disk to physical memory.
 * @param pt page table for the virtual memory.
 * @param page page number of the page to swap into memory.
 */
void swap_page(struct page_table *pt, int page) {
        int frame_to_free = frame_select_algo(pt);
        int page_to_free = frame_states[frame_to_free].page;
        int bits_on_frame = frame_states[frame_to_free].bits;

        char *physmem = page_table_get_physmem(pt);
```

```
        // dirty virtual memory in frame?
        if (bits_on_frame == (PROT_READ | PROT_WRITE)) {
                disk_write(disk, page_to_free, &physmem[frame_to_free * PAGE_SIZE]);
#ifdef STATS
                disk_writes++;
#endif
        }

        disk_read(disk, page, &physmem[frame_to_free * PAGE_SIZE]);
        page_table_set_entry(pt, page, frame_to_free, PROT_READ);
        frame_states[frame_to_free].bits = PROT_READ;
        frame_states[frame_to_free].page = page;

        page_table_set_entry(pt, page_to_free, frame_to_free, PROT_NONE);

#ifdef STATS
        disk_reads++;
#endif
}

/**
 * Handles page faults in the virtual memory. Page faults occur when
 * a page in virtual memory is not loaded into a frame in physical
 * memory, or when the protection bits do not allow the attempted operation.
 * @param pt page table for the virtual memory.
 * @param page the page involved in the page fault.
 */
void page_fault_handler(struct page_table *pt, int page)
{
        int frame, bits;
        page_table_get_entry(pt, page, &frame, &bits);

        if (bits == PROT_READ) {
                // the entry exists in physical memory, but is read-only
                page_table_set_entry(pt, page, frame, bits | PROT_WRITE);
                frame_states[frame].bits = bits | PROT_WRITE;
        } else {
                int empty_frame_index = get_empty_frame(page_table_get_nframes(pt));
                if (empty_frame_index != -1) {
                        // there is an empty frame for the entry
                        page_table_set_entry(pt, page, empty_frame_index, PROT_READ);
                        frame_states[empty_frame_index].page = page;
                        frame_states[empty_frame_index].bits = PROT_READ;
                } else {
                        // there were no empty frames, find one to release via frame_select_algo
                        swap_page(pt, page);
                }
```

```
        }

#ifdef STATS
        npage_faults++;
#endif
}

int main(int argc, char *argv[])
{
        if(argc!=5) {
                printf("use: virtmem <npages> <nframes> <rand|fifo|custom> <sort|scan|focus>\n");
                return 1;
        }

        int npages = atoi(argv[1]);
        int nframes = atoi(argv[2]);
        frame_states = malloc(sizeof(struct frame_state) * nframes);
        const char *select_algo = argv[3];
        const char *program = argv[4];

        disk = disk_open("myvirtualdisk",npages);
        if(!disk) {
                fprintf(stderr, "couldn't create virtual disk: %s\n", strerror(errno));
                return 1;
        }

        struct page_table *pt = page_table_create(npages, nframes, page_fault_handler);
        if(!pt) {
                fprintf(stderr, "couldn't create page table: %s\n", strerror(errno));
                return 1;
        }

        char *virtmem = page_table_get_virtmem(pt);

        if (!strcmp(select_algo, "rand")) {
                frame_select_algo = &get_random_frame;
        } else if (!strcmp(select_algo, "fifo")) {
                frame_select_algo = &get_frame_from_queue;
        } else if (!strcmp(select_algo, "custom")) {
                frame_select_algo = &get_custom_frame;
        } else {
                fprintf(stderr, "unknown selection algorithm: %s\n", select_algo);
                exit(EXIT_FAILURE);
        }

        if(!strcmp(program, "sort")) {
                sort_program(virtmem, npages*PAGE_SIZE);
```

```
        } else if(!strcmp(program, "scan")) {
                scan_program(virtmem, npages*PAGE_SIZE);

        } else if(!strcmp(program, "focus")) {
                focus_program(virtmem, npages*PAGE_SIZE);

        } else {
                fprintf(stderr, "unknown program: %s\n", program);
                exit(EXIT_FAILURE);
        }

        page_table_delete(pt);
        disk_close(disk);
        free(frame_states);

#ifdef STATS
        print_stats();
#endif

        return 0;
}
```

7.2.3 disk.h

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

#ifndef DISK_H
#define DISK_H

#define BLOCK_SIZE 4096

/*
Create a new virtual disk in the file "filename", with the given number of blocks.
Returns a pointer to a new disk object, or null on failure.
*/

struct disk * disk_open( const char *filename, int blocks );

/*
Write exactly BLOCK_SIZE bytes to a given block on the virtual disk.
"d" must be a pointer to a virtual disk, "block" is the block number,
and "data" is a pointer to the data to write.
*/

void disk_write( struct disk *d, int block, const char *data );

/*
Read exactly BLOCK_SIZE bytes from a given block on the virtual disk.
"d" must be a pointer to a virtual disk, "block" is the block number,
and "data" is a pointer to where the data will be placed.
*/

void disk_read( struct disk *d, int block, char *data );

/*
Return the number of blocks in the virtual disk.
*/

int disk_nblocks( struct disk *d );

/*
Close the virtual disk.
*/

void disk_close( struct disk *d );

#endif
```

7.2.4 disk.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);
                free(d);
                return 0;
        }

        return d;
```

```
}

void disk_write( struct disk *d, int block, const char *data )
{
        if(block<0 || block>=d->nblocks) {
                fprintf(stderr,"disk_write: invalid block #%d\n",block);
                abort();
        }

        int actual = pwrite(d->fd,data,d->block_size,block*d->block_size);
        if(actual!=d->block_size) {
                fprintf(stderr,"disk_write: failed to write block #%d: %s\n",block,strerror(errno));
                abort();
        }
}

void disk_read( struct disk *d, int block, char *data )
{
        if(block<0 || block>=d->nblocks) {
                fprintf(stderr,"disk_read: invalid block #%d\n",block);
                abort();
        }

        int actual = pread(d->fd,data,d->block_size,block*d->block_size);
        if(actual!=d->block_size) {
                fprintf(stderr,"disk_read: failed to read block #%d: %s\n",block,strerror(errno));
                abort();
        }
}

int disk_nblocks( struct disk *d )
{
        return d->nblocks;
}

void disk_close( struct disk *d )
{
        close(d->fd);
        free(d);
}
```

7.2.5 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.
*/

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

/* Return a pointer to the start of the virtual memory associated with a page table. */

char * page_table_get_virtmem( struct page_table *pt );

/* Return a pointer to the start of the physical memory associated with a page table. */

char * page_table_get_physmem( struct page_table *pt );

/* Return the total number of frames in the physical memory. */
```

```
int page_table_get_nframes( struct page_table *pt );

/* Return the total number of pages in the virtual memory. */

int page_table_get_npages( struct page_table *pt );

/* Print out the page table entry for a single page. */

void page_table_print_entry( struct page_table *pt, int page );

/* Print out the state of every page in a page table. */

void page_table_print( struct page_table *pt );

#endif
```

7.2.6 page_table.c

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

#define _GNU_SOURCE // to avoid implicit declaration warning on remap_file_pages
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <ucontext.h>
#include <sys/mman.h>

#include "page_table.h"

struct page_table {
        int fd;
        char *virtmem;
        int npages;
        char *physmem;
        int nframes;
        int *page_mapping;
        int *page_bits;
        page_fault_handler_t handler;
};

struct page_table *the_page_table = 0;

static void internal_fault_handler( int signum, siginfo_t *info, void *context )
{

#ifdef i386
        char *addr = (char*)(((struct ucontext *)context)->uc_mcontext.cr2);
#else
        char *addr = info->si_addr;
#endif

        struct page_table *pt = the_page_table;

        if(pt) {
                int page = (addr-pt->virtmem) / PAGE_SIZE;

                if(page>=0 && page<pt->npages) {
                        pt->handler(pt,page);
                        return;
```

```
                }
        }

        fprintf(stderr,"segmentation fault at address %p\n",addr);
        abort();
}

struct page_table * page_table_create( int npages, int nframes, page_fault_handler_t handler )
{
        int i;
        struct sigaction sa;
        struct page_table *pt;
        char filename[256];

        pt = malloc(sizeof(struct page_table));
        if(!pt) return 0;

        the_page_table = pt;

        sprintf(filename,"/tmp/pmem.%d.%d",getpid(),getuid());

        pt->fd = open(filename,O_CREAT|O_TRUNC|O_RDWR,0777);
        if(!pt->fd) return 0;

        ftruncate(pt->fd,PAGE_SIZE*npages);

        unlink(filename);

        pt->physmem = mmap(0,nframes*PAGE_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,pt->fd,0);
        pt->nframes = nframes;

        pt->virtmem = mmap(0,npages*PAGE_SIZE,PROT_NONE,MAP_SHARED|MAP_NORESERVE,pt->fd,0);
        pt->npages = npages;

        pt->page_bits = malloc(sizeof(int)*npages);
        pt->page_mapping = malloc(sizeof(int)*npages);

        pt->handler = handler;

        for(i=0;i<pt->npages;i++) pt->page_bits[i] = 0;

        sa.sa_sigaction = internal_fault_handler;
        sa.sa_flags = SA_SIGINFO;

        sigfillset( &sa.sa_mask );
        sigaction( SIGSEGV, &sa, 0 );

        return pt;
```

```
}

void page_table_delete( struct page_table *pt )
{
        munmap(pt->virtmem,pt->npages*PAGE_SIZE);
        munmap(pt->physmem,pt->nframes*PAGE_SIZE);
        free(pt->page_bits);
        free(pt->page_mapping);
        close(pt->fd);
        free(pt);
}

void page_table_set_entry( struct page_table *pt, int page, int frame, int bits )
{
        if( page<0 || page>=pt->npages ) {
                fprintf(stderr,"page_table_set_entry: illegal page #%d\n",page);
                abort();
        }

        if( frame<0 || frame>=pt->nframes ) {
                fprintf(stderr,"page_table_set_entry: illegal frame #%d\n",frame);
                abort();
        }

        pt->page_mapping[page] = frame;
        pt->page_bits[page] = bits;

        remap_file_pages(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,0,frame,0);
        mprotect(pt->virtmem+page*PAGE_SIZE,PAGE_SIZE,bits);
}

void page_table_get_entry( struct page_table *pt, int page, int *frame, int *bits )
{
        if( page<0 || page>=pt->npages ) {
                fprintf(stderr,"page_table_get_entry: illegal page #%d\n",page);
                abort();
        }

        *frame = pt->page_mapping[page];
        *bits = pt->page_bits[page];
}

void page_table_print_entry( struct page_table *pt, int page )
{
        if( page<0 || page>=pt->npages ) {
                fprintf(stderr,"page_table_print_entry: illegal page #%d\n",page);
                abort();
        }
```

```
        int b = pt->page_bits[page];

        printf("page %06d: frame %06d bits %c%c%c\n",
                page,
                pt->page_mapping[page],
                b&PROT_READ  ? 'r' : '-',
                b&PROT_WRITE ? 'w' : '-',
                b&PROT_EXEC  ? 'x' : '-'
        );

}

void page_table_print( struct page_table *pt )
{
        int i;
        for(i=0;i<pt->npages;i++) {
                page_table_print_entry(pt,i);
        }
}

int page_table_get_nframes( struct page_table *pt )
{
        return pt->nframes;
}

int page_table_get_npages( struct page_table *pt )
{
        return pt->npages;
}

char * page_table_get_virtmem( struct page_table *pt )
{
        return pt->virtmem;
}

char * page_table_get_physmem( struct page_table *pt )
{
        return pt->physmem;
}
```

7.2.7 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
```

7.2.8 program.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();
                }
        }

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

```
        printf("focus result is %d\n",total);
}

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

        srand(4856);

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

        qsort(data,length,1,compare_bytes);

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

        printf("sort result is %d\n",total);

}

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

        for(i=0;i<length;i++) {
                data[i] = i%256;
        }

        for(j=0;j<10;j++) {
                for(i=0;i<length;i++) {
                        total += data[i];
                }
        }

        printf("scan result is %d\n",total);
}
```

7.2.9 run_tests.sh

```
#!/bin/bash

make clean
make stats

if [ $# -eq 0 ]
then
        echo "Usage: $0 <nframes>"
        exit 1
fi

echo
echo "--------------------------------"
echo "Sort"
echo "--------------------------------"
echo "Rand:"
./virtmem 100 $1 rand sort
echo "Fifo:"
./virtmem 100 $1 fifo sort
echo "Custom:"
./virtmem 100 $1 custom sort

echo
echo "--------------------------------"
echo "Scan"
echo "--------------------------------"
echo "Rand:"
./virtmem 100 $1 rand scan
echo "Fifo:"
./virtmem 100 $1 fifo scan
echo "Custom:"
./virtmem 100 $1 custom scan

echo
echo "--------------------------------"
echo "Focus"
echo "--------------------------------"
echo "Rand:"
./virtmem 100 $1 rand focus
echo "Fifo:"
./virtmem 100 $1 fifo focus
echo "Custom:"
./virtmem 100 $1 custom focus
```

7.2.10 Graph Stats Generation Branch

Graph Stats Generation Branch Diff

```
commit 202e981f87565aa2e8003eed7eeae59887abbf79
Author: AndreasHassing <andreas@famhassing.dk>
Date:   Tue Nov 15 10:22:54 2016 +0100

    Add sick stats output generation, NOT FOR PRODUCTION

diff --git a/main.c b/main.c
index 97a7ba4..2e88c4c 100644
--- a/main.c
+++ b/main.c
@@ -22,10 +22,11 @@ int disk_reads;
 int disk_writes;

 void print_stats() {
-        printf("Page faults: %d\n", npage_faults);
-        printf("Disk reads: %d\n", disk_reads);
-        printf("Disk writes: %d\n", disk_writes);
-        printf("Disk reads+writes: %d\n\n", disk_reads+disk_writes);
+//        printf("Page faults: %d\n", npage_faults);
+//        printf("Disk reads: %d\n", disk_reads);
+//        printf("Disk writes: %d\n", disk_writes);
+//        printf("Disk reads+writes: %d\n\n", disk_reads+disk_writes);
+        printf("%d\t", disk_reads+disk_writes);
 }
 #endif

diff --git a/program.c b/program.c
index f09d857..930eb6b 100644
--- a/program.c
+++ b/program.c
@@ -46,7 +46,7 @@ void focus_program( char *data, int length )
                 total += data[i];
         }

-        printf("focus result is %d\n",total);
+        //printf("focus result is %d\n",total);
 }

 void sort_program( char *data, int length )
@@ -66,7 +66,7 @@ void sort_program( char *data, int length )
                 total += data[i];
         }

-        printf("sort result is %d\n",total);
+        //printf("sort result is %d\n",total);
```

```
 }

@@ -86,5 +86,5 @@ void scan_program( char *cdata, int length )
                 }
         }

-        printf("scan result is %d\n",total);
+        //printf("scan result is %d\n",total);
 }
```

Graph Stats Generation Branch `run_tests.sh`

```
#!/bin/bash

make clean
make stats

echo "--------------------------------"
echo "Sort"
echo "--------------------------------"
for i in {10..100}
do
        ./virtmem 100 $i rand sort
        ./virtmem 100 $i fifo sort
        ./virtmem 100 $i custom sort
        echo
done
echo "--------------------------------"

echo
echo "--------------------------------"
echo "Scan"
echo "--------------------------------"
for i in {10..100}
do
        ./virtmem 100 $i rand scan
        ./virtmem 100 $i fifo scan
        ./virtmem 100 $i custom scan
        echo
done
echo "--------------------------------"

echo "--------------------------------"
echo "Focus"
echo "--------------------------------"
for i in {10..100}
do
        ./virtmem 100 $i rand focus
        ./virtmem 100 $i fifo focus
        ./virtmem 100 $i custom focus
        echo
done
echo "--------------------------------"
```