OPERATING SYSTEMS, DV1628/DV1629

—

LAB 2: MEMORY SYSTEMS

Håkan Grahn
Blekinge Institute of Technology

2022 fall
updated: 2022-10-10

The objective of this laboratory assignment is to study how the memory system impacts the performance, with a focus on the virtual memory system and page replacement.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students! It is ok to do them individually, but groups of more than two students are not allowed.

Home Assignment 1. Preparations

Read through these laboratory instructions and do the **home assignments**. The purpose of the home assignments is to do them *before* the lab sessions to prepare your work and make more efficient use of the lab sessions.

Read the following sections in the course book [1]:

- Section 3.2: A memory abstraction: Address spaces
- Section 3.3: Virtual memory
- Section 3.4: Page replacement algorithms
- Section 3.5: Design issues for paging systems
- Section 3.6: Implementation issues
- Section 10.4: Memory management in Linux

End of home assignment 1.

Home Assignment 2. Plagiarism and collaboration

You are encouraged to work in groups of two. Groups larger than two are not accepted.

Discussions between laboratory groups are positive and can be fruitful. It is normally not a problem, but watch out so that you do not cross the border to cheating. For example, you are *not allowed* to share solution approaches, solutions to the different tasks, source code, output data, results, etc.

The submitted solution(s) to the laboratory assignments and tasks, should be developed by the group members only. You are not allowed to copy code from somewhere else, i.e., neither from your student fellows nor from the internet or any other source. The only other source code that is allowed to use is the one provided with the laboratory assignment.

End of home assignment 2.

1 Introduction

An important part of any computer system is the memory system. It contains both a physical main memory (a.k.a. primary memory) and a secondary memory for long term and persistent storage (usually on some kind of disk). When a program is executed, the code and data of the program needs to be in the main memory in order for the processer to access it. However, the main memory is a limited resource and cannot hold all programs' code and data at the same time. Therefore, the secondary memory is used as a backing storage for parts of the program during execution. The technique is called virtual memory.

In this laboratory we will look into some of the aspects in the memory system that affects performance as well as how virtual memory works.

The programs in this lab assignment are tested on Ubuntu Linux, and thus work in the lab room (G332). However, they should also work on any Linux distribution you may have at home. The source code files and programs necessary for the laboratory are available on Canvas. Note that the data sizes, number of iterations, etc. in the test programs are tuned to give reasonable / interesting execution times on the computers in the lab room (G332). If you run on other hardware, the execution times etc. may vary and the observed effects may be different.

Further, if nothing else is said/written, you shall compile your programs using:

```
gcc -O2 -o output_file_name sourcefile1.c ... sourcefileN.c
```

All commands are written in a Linux terminal or console window.

2 Examination and grading

Present and discuss your solutions with a teacher / lab assistant when all tasks are done.

When you have discussed your solutions orally, prepare and submit a `tar`-file or `zip`-file containing:

- **Source code:** The source-code for working solutions to the tasks in this laboratory assignment. Specifically, your well-commented source code for **Task 3**, **Task 6**, and **Task 9**.
- **Written report:** You should write a short report (approximately 2-3 pages) describing your answers to the questions in the assignment and your implementations, as outline below. The format of the report must be pdf.
  - **Answers to questions:** Submit answers to all questions posed in the home assignments and the tasks, including measurements, complete tables with number of page faults, etc.
  - **Implementation:** A short, general description of your implementations of the FIFO, LRU, and Optimal page replacement algorithms, i.e., general code structure / execution flow, how the data structures are organized, etc.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

3 CPU-bound vs. I/O-bound processes

In this first part of laboratory 2, we will study the differences between CPU-bound and I/O-bound processes, and the some aspects of the memory system behavior.

Home Assignment 3. Test programs

Study the small test programs in Listing 1 (`test1.c`) and Listing 2 (`test2.c`), so you understand what they are doing.

How much dynamic memory is allocated in the `test1` program (i.e., how large is `struct link`)?

How much data is written to the temporary file `/tmp/file1.txt` in each iteration of the program `test2`?

End of home assignment 3.

In order to understand the performance of the virtual memory system and the disk I/O system, we need to have some way of measuring what is going on in the operating system. To our help we will use two programs, i.e., `vmstat` and `top`.

- `vmstat` provides information about, e.g., swapping activity, allocated memory, and blocks read and written to a block device.
- `top` provides information about, e.g., how much execution time, cpu utilization, and memory each process is using.

Home Assignment 4. `vmstat` and `top`

Study the man pages of `vmstat` and `top` so you understand how these two commands work. Specifically, understand the parameters that you can give to the commands and the output format of the commands.

In which output columns of `vmstat` can you find the amount of memory a process uses, the number of swap ins and outs, and the number of I/O blocks read and written?

Where in the output from `top` do you find how much cpu time and cpu utilization a process is using?

End of home assignment 4.

Task 1. CPU-bound vs. I/O-bound processes

Compile the test programs `test1` and `test2`.

Execute "`top`" and "`vmstat 1`" in two new terminal windows.

Execute the test program `test1`. How much memory does the program use? Does it correspond to your answer in Home Assignment 3?

What is the cpu utilization when executing the `test1` program? Which type of program is `test1`?

Execute the test program `test2`. How much memory does the program use? How many blocks are written out by the program? How does it correspond to your answer in Home Assignment 3?

What is the cpu utilization when executing the `test2` program? Which type of program can we consider `test2` to be?

End of task 1.

When measuring the execution time, it looks a bit different if you have `bash` och `tcsh` as default shell in your terminal / console. In the lab room (G332) `tcsh` is default, but in a standard Ubuntu/Linux distribution `bash` is default. You usually can find out which shell you run by executing the command (if the environment variable `SHELL` is set correctly, which is unfortunately not always done):

```
echo $SHELL
```

If you are running `bash`, you should measure the execution time with:

```
time command
```

The value of "real" is the wall clock time that we are interested in.

If you are running `tcsh`, you should measure the execution time with:

```
/usr/bin/time command
```

The value of "elapsed" is the wall clock time that we are interested in.

Task 2. Overlapping CPU and I/O execution

Study the two scripts `run1` (Listing 3) and `run2` (Listing 4).

What is the difference between them in terms of how they execute the test programs?

Execute the script `run1` and measure the execution time. Study the cpu utilization using `top` during the execution. How long time did it take to execute the script and how did the cpu utilization vary?

Execute the script `run2` and measure the execution time. Study the cpu utilization using `top` during the execution. How long time did it take to execute the script and how did the cpu utilization vary?

Which of the two cases executed fastest?

In both cases, the same mount of work was done. I which case was the system best utilized and why?

End of task 2.

4 Virtual memory and page replacement algorithms

In this part of the laboratory, we are going to study the effects of different page replacement algorithms.

Among the files provided to you, there are two files with the extension `.mem` (`mp3d.mem` and `mult.mem`). These two files contains 100,000 memory references each, where each row in the file corresponds to a memory address that the program has generated. The traces are captured when running two different programs (mp3d, a particle-based wind tunnel simulation, and matrix multiplication). Both applications are from a classic benchmark suite of numerical applications.

Task 3. Implementation of FIFO page replacement

Your task is to write a program that calculates the number of page faults for a sequence of memory references (i.e., the memory reference trace in the `.mem`-files) when using the FIFO (First-In-First-Out) page replacement policy.

Your program shall take as input the number of physical pages, the page size, and the name of the trace file, see the example below:

```
./fifo no_phys_pages page_size filename
```

The program shall write the resulting number of page faults for that specific combination of number of pages and page size, for either `mp3d.mem` or `mult.mem`.

Example execution:

```
mycomputer$ ./fifo 4 256 mp3d.mem
No physical pages = 4, page size = 256
Reading memory trace from mp3d.mem...  Read 100000 memory references
Result:  11940 page faults
```

End of task 3.

Task 4. Page fault measurements for the FIFO page replacement policy

Use the program that you developed in Task 3 to fill in the number of pages faults in Table 1 and Table 2. To your help, we have filled in some of the numbers in the tables (so you can use them to check that you program generate the correct number of page faults).

End of task 4.

Table 1: Number of page faults for `mp3d.mem` when using FIFO as page replacement policy.

Page size	Number of pages							
	1	2	4	8	16	32	64	128
128								
256			11940					
512							417	
1024								

Table 2: Number of page faults for `mult.mem` when using FIFO as page replacement policy.

Page size	Number of pages							
	1	2	4	8	16	32	64	128
128						249		
256								
512								
1024		16855						

Task 5. Evaluation of the FIFO page replacement policy

Based on the values in Table 1 and Table 2, answer the following questions.

What is happening when we keep the number of pages constant and increase the page size? Explain why!

What is happening when we keep the page size constant and increase the number of pages? Explain why!

If we double the page size and halve the number of pages, the number of page faults sometimes decrease and sometimes increase. What can be the reason for that?

Focus now on the results in Table 2 (`matmul`). At some point decreases the number page faults very drastically. What memory size does that correspond to? Why does the number of page faults decrease so drastically at that point?

At some point the number of page faults does not decrease anymore when we increase the number of pages. When and why do you think that happens?

End of task 5.

Task 6. Implementation of LRU page replacement

Your task is to write a program that calculates the number of page faults for a sequence of memory references (i.e., the memory reference trace in the `.mem`-files) when using the LRU (Least-Recently-Used) page replacement policy.

Your program shall take as input the number of physical pages, the page size, and the name of the trace file, see the example below:

```
./lru no_phys_pages page_size filename
```

The program shall write the resulting number of page faults for that specific combination of number of pages and page size, for either `mp3d.mem` or `mult.mem`.

Example execution:

```
mycomputer$ ./lru 4 256 mp3d.mem
No physical pages = 4, page size = 256
Reading memory trace from mp3d.mem...  Read 100000 memory references
Result:  9218 page faults
```

End of task 6.

Task 7. Page fault measurements for the LRU page replacement policy

Use the program that you developed in Task 6 to fill in the number of pages faults in Table 3. To your help, we have filled in some of the numbers in the table (so you can use them to check that you program generate the correct number of page faults).

End of task 7.

Table 3: Number of page faults for `mp3d.mem` when using LRU as replacement policy.

Page size	Number of pages							
	1	2	4	8	16	32	64	128
128								
256			9218					
512							362	
1024								

Task 8. Comparison of the FIFO and LRU page replacement policies

Based on the values in Table 1 and Table 3, answer the following questions.

Which of the page replacement policies FIFO and LRU seems to give the lowest number of page faults? Explain why!

In some of the cases, the number of page faults are the same for both FIFO and LRU. Which are these cases? Why is the number of page faults equal for FIFO and LRU in those cases? Explain why!

End of task 8.

Task 9. Implementation of Optimal page replacement (Bélády's algorithm)

Your task is to write a program that calculates the minimum number of page faults for a sequence of memory references, i.e., you should implement the Optimal page replacement policy (which is also known as Bélády's algorithm).

Your program shall take as input the number of physical pages, the page size, and the name of the trace file, see the example below:

```
./optimal no_phys_pages page_size filename
```

The program shall write the resulting number of page faults for that specific combination of number of pages and page size,

for either `mp3d.mem` or `mult.mem`.

Example execution:

```
mycomputer$ ./optimal 32 128 mp3d.mem
No physical pages = 32, page size = 128
Reading memory trace from mp3d.mem...  Read 100000 memory references
Result:  824 page faults
```

End of task 9.

Task 10. Page fault measurements for the Optimal (Bélády's) page replacement policy

Use the program that you developed in Task 9 to fill in the number of pages faults in Table 4. To your help, we have filled in some of the numbers in the table (so you can use them to check that you program generate the correct number of page faults).

End of task 10.

Table 4: Number of page faults for `mp3d.mem` when using Optimal (Bélády's algorithm) as replacement policy.

Page size	Number of pages							
	1	2	4	8	16	32	64	128
128						824		
256								
512								
1024			3367					

Task 11. Comparison of the FIFO, LRU, and Optimal page replacement policies

Based on the values in Table 1, Table 3, and Table 4, answer the following questions.

As expected, the Optimal policy gives the lowest number of page faults. Explain why!

Optimal is considered to be impossible to use in practice. Explain why!

Does FIFO and/or LRU have the same number of page faults as Optimal for some combination(s) of page size and number of pages? If so, for which combination(s) and why?

End of task 11.

References

[1] Andrew S. Tanenbaum and Herbert Bos, "Modern Operating Systems, 4th ed", *Pearson Education Limited*, ISBN-10: 0-13-359162-X, 2015.

A Program listings

Listing 1. `test1.c`

```
#include <stdio.h>
#include <stdlib.h>
/* test1.c */

#define SIZE (32*1024)
#define ITERATIONS 10

int main(int argc, char **argv)
{
    struct link {
        int x[SIZE][SIZE];
    };
    struct link *start;
    int iteration, i, j;
    start = (struct link *) calloc(1, sizeof(struct link));
    if (!start) {
        printf("Fatal error: Can't allocate memory of %d x %d = %lu\n", SIZE, SIZE, (unsigned long)SIZE*SIZE);
        exit(-1);
    }
    for (iteration = 0; iteration < ITERATIONS; iteration++) {
        printf("test1, iteration: %d\n", iteration);
        for (i = 0; i < SIZE; i++)
            for (j = 0; j < SIZE; j++)
                start->x[i][j] = 0;
    }
}
```

Listing 2. `test2.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
/* test2.c */

#define SIZE (16*1024)
#define ITERATIONS 5

int main(int argc, char **argv)
{
    struct link {
        int x[SIZE][SIZE];
    };
    struct link *start;
    int iteration;
    FILE *f;
    start = (struct link *) calloc(1, sizeof(struct link));
    if (!start) {
        printf("Fatal error: Can't allocate memory of %d x %d = %lu\n", SIZE, SIZE, (unsigned long)SIZE*SIZE);
        exit(-1);
    }
    for (iteration = 0; iteration < ITERATIONS; iteration++) {
        printf("test2, iteration: %d\n", iteration);
        f = fopen("/tmp/file1.txt", "w");
        fwrite(start->x, sizeof(start->x[0][0]), SIZE*SIZE, f);
        fclose(f);
    }
}
```

Listing 3. `run1`

```
#!/bin/bash
./test1; ./test1; ./test1; ./test2; ./test2
```

Listing 4. `run2`

```
#!/bin/bash
(./test1; ./test1; ./test1) &
(./test2; ./test2)
```