Operating Systems

Tutorial 7

Fabian Klopfer

7. Dezember 2022

Intro

- Pingo Polls
- This time: extended pointer treatment
- Starting next time: Loaders

Exercise Sheet 6

Exercise 1

Exercise 1 I

1. Does the difficulty of a software implementation of the Least-Recently-Used algorithm lie in converting virtual addresses to physical addresses or in handling page faults? Explain your answer.
   - Address translation: needs to be fast, that's why it's usualy impl. in HW.
   - Alter field of page table on each address translation
   - How to make it so fast with SW?
   - Page faults: Occur less often, allowed to be slower
2. Name two paging algorithms that can be considered approximations of the Least-Recently-Used algorithm.
   Not frequently used, not recently used

Exercise 1 II

3. Name two key challenges in implementing paging systems.
   Speed & Size:
   - Speed as address translation happens often
   - Size as address spaces are large but the page table should be small in-memory
4. What property must a process have in terms of its memory accesses for a translation lookaside buffer to increase the speed of memory accesses?
   It has to make a large number of references to a small number of pages. **Locality** of access

Exercise 1 III

5. Name three advantages of virtualizing operating systems.
   - Simpler Maintenance
   - Simpler Migration
   - Simpler Testing
   - Less prone to crashes of the whole system/isolation
6. What is a “world switch”?
   Transition between hypervisor and guest

Exercise 1 IV

7. Name two differences between virtualization and containers.
   Virtualization: Virtualizes HW, isolates OSes
   Containers: Virtualize OSes, isolate processes
8. For what reason can a hardware-based approach to virtualization (CPU support) be slower than a binary translation-based software approach?
   Many traps when using HW-based virtualization

Exercise 2

Exercise 2 I

1. A machine instruction that loads a 32-bit word from memory into a register on the CPU can cause up to four page faults. How?

   Operation is on boundary $\Rightarrow$ 2 page faults
   both operands may be on boundary $\Rightarrow$ 2 page fault for each

2. 48-bit addresses, 8 KiB sized pages. How many pages does a single-level linear page table contain?

   $\frac{2^{48}}{8 \cdot 1024} = 2^{48}/2^{13} = 2^{35} = 34,359,738,368 \approx 34$ million.

Exercise 2 II

3. virtual address space 64 KiB, page size 4 KiB. A process has 32 768 B program code, 16 386 B data and 15 870 B stack. Does the process fit into the address space?

   System has $64/4 = 16$ pages.
   Code, data and stack have each
   $32\,768\,\mathrm{B} = 8 \cdot 4096\,\mathrm{B} + 0\,\mathrm{B}$,
   $16\,386\,\mathrm{B} = 4 \cdot 4096\,\mathrm{B} + 2\,\mathrm{B}$,
   $15\,870\,\mathrm{B} = 3 \cdot 4096\,\mathrm{B} + 3582\,\mathrm{B}$
   Thus $8 + 5 + 4 = 17$ pages
   Process does **not** fit into the address space.

Exercise 2 III

4. Now page size 512 B. Does the process now fit into the address space?

   System has $65,536/512 = 128$ pages.
   Code, data and stack have each
   $32\,768\,\text{B} = 64 \cdot 512\,\text{B} + 0\,\text{B}$,
   $16\,386\,\text{B} = 32 \cdot 512\,\text{B} + 2\,\text{B}$,
   $15\,870\,\text{B} = 30 \cdot 512\,\text{B} + 510\,\text{B}$
   Thus $64 + 33 + 31 = 128$ pages
   Process does fit into the address space.

Exercise 3

Exercise 3 I

Which page is evicted when using the following strategies?

Page	Load time	Last Access	R	M
0	170	437	0	0
1	335	436	0	1
2	65	512	1	1
3	99	501	1	0

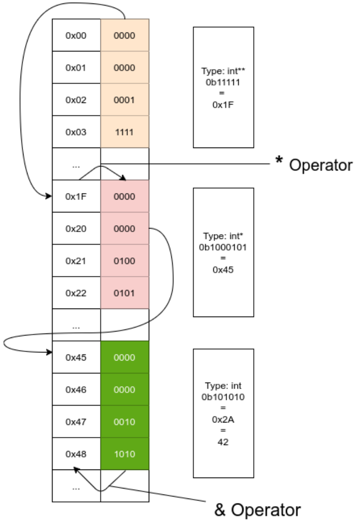
1. First-In First-Out
   min load time $= 65$
   $\Rightarrow$ Page 2
2. Second Chance
   min load time$|_{R=0} = 170$
   $\Rightarrow$ Page 0
3. Least-Recently-Used
   min last access $= 436$
   $\Rightarrow$ Page 1
4. Not-Recently-Used
   random$|_{R=0 \wedge M=0}$
   $\Rightarrow$ Page 0

Exercise 4

Pointers Again I

- Pointer $\equiv$ Variable storing an address
- $*$ Operator returns the value at the address stored by the variable/pointer
- & Operator returns the address of the variable itself

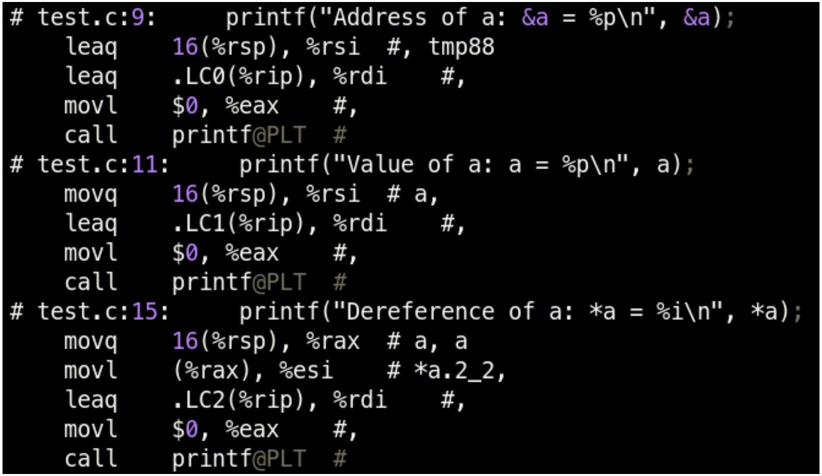
Pointers Again II



Pointers Again III

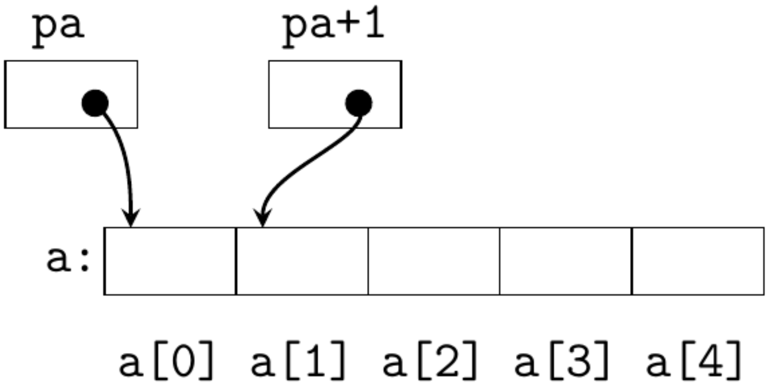
```c
#include <stdio.h>

int main(void) {
    int* a;
    int b;
    a = &b;
    *a = 6;

    printf("Address of a: &a = %p\n", &a);
    printf("Value of a: a = %p\n", a);
    printf("Dereference of a: *a = %i\n", *a);

    return 0;
}
```

Pointers Again IV

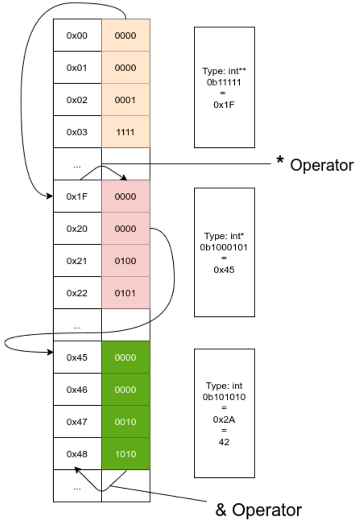


```
# test.c:9:      printf("Address of a: &a = %p\n", &a);
    leaq    16(%rsp), %rsi  #, tmp88
    leaq    .LC0(%rip), %rdi    #,
    movl    $0, %eax    #,
    call    printf@PLT  #
# test.c:11:      printf("Value of a: a = %p\n", a);
    movq    16(%rsp), %rsi  # a,
    leaq    .LC1(%rip), %rdi    #,
    movl    $0, %eax    #,
    call    printf@PLT  #
# test.c:15:      printf("Dereference of a: *a = %i\n", *a);
    movq    16(%rsp), %rax  # a, a
    movl    (%rax), %esi    # *a.2_2,
    leaq    .LC2(%rip), %rdi    #,
    movl    $0, %eax    #,
    call    printf@PLT  #
```

Pointers Again V



Pointers Again VI



Pointers Again VII

```
int a[2][2]:

__a[0][0]__|__a[0][1]__|__a[1][0]__|__a[1][1]__
   (int)        (int)        (int)        (int)

int **a (e.g. dynamically allocated with nested mallocs)

__a__
(int**)
   |
   v
__a[0]__|__a[1]__
   (int*)   (int*)
     |         |
     |         |
     v         ------------------>
__a[0][0]__|__a[0][1]__        __a[1][0]__|__a[1][1]__
   (int)        (int)             (int)        (int)
```

Pointers Again VIII

Double pointers and 2D arrays and pointer to arrays example

Exercise 4 I

What output does this program produce? Explain your answer.

```c
#include <stdio.h>

int main(void)
{
    int a[3][2] = {{1, 2}, {3, 4}, {5, 6}};

    /* 1 */ printf("%d\n", a[0][0]);
    /* 2 */ printf("%d\n", **a);
    /* 3 */ printf("%d\n", *(*a + 1));
    /* 4 */ printf("%d\n", **(a + 1));
    /* 5 */ printf("%d\n", **(&a[0] + 1));
    /* 6 */ printf("%d\n", *( ((int *)(a + 2)) - 1 ));

    return 0;
}
```

Exercise 4 II

1. 1, Prints the first value of the first subarray
2. 1, prints the starting address of the array which is also the staring address of the first subarray which is the first element of the first subarray
3. 2, adds one integer step to the address of the first element of the first subarray
4. 3, adds one step to the address of the outer array, i.e. gets the address of the first element of the second subarray
5. as above
6. the innermost addition adds two subarray steps to the address of the array, i.e. this is the address of the third subarray. by subtracting one, we arrive at the second element of the second subarray, i.e. 4 is printed

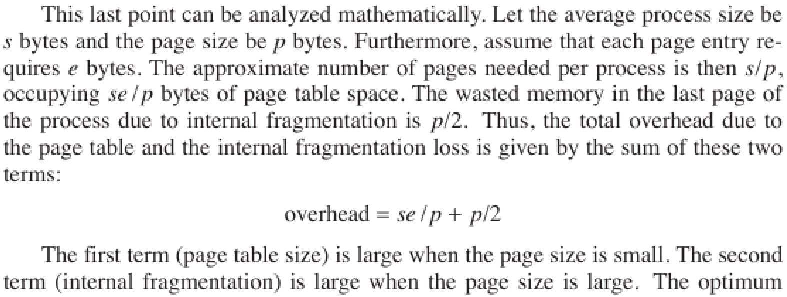
Exercise 5

Exercise 5 I

Let s denote the average process size in bytes, let p denote the size of a page in bytes, and let e denote the size of a page table entry in bytes.

1. Construct an expression (in p, s, e) for the average memory consumption (per process) for paging (single-level page table).
2. Interpret your expression in terms of page size p.
3. Determine the optimal page size p in your model.

Exercise 5 II

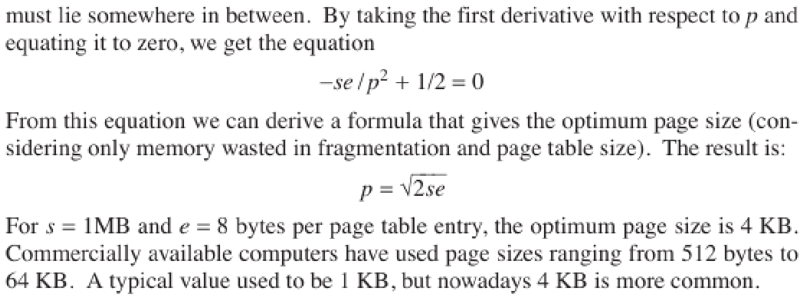


This last point can be analyzed mathematically. Let the average process size be s bytes and the page size be p bytes. Furthermore, assume that each page entry requires e bytes. The approximate number of pages needed per process is then s/p, occupying se/p bytes of page table space. The wasted memory in the last page of the process due to internal fragmentation is $p/2$. Thus, the total overhead due to the page table and the internal fragmentation loss is given by the sum of these two terms:

$$\text{overhead} = se/p + p/2$$

The first term (page table size) is large when the page size is small. The second term (internal fragmentation) is large when the page size is large. The optimum

Exercise 5 III



must lie somewhere in between. By taking the first derivative with respect to p and equating it to zero, we get the equation

$$-se / p^2 + 1/2 = 0$$

From this equation we can derive a formula that gives the optimum page size (considering only memory wasted in fragmentation and page table size). The result is:

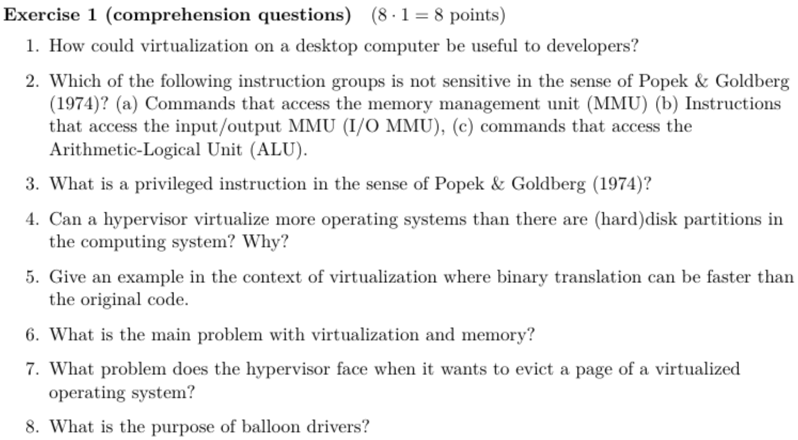
$$p = \sqrt{2se}$$

For s = 1MB and e = 8 bytes per page table entry, the optimum page size is 4 KB. Commercially available computers have used page sizes ranging from 512 bytes to 64 KB. A typical value used to be 1 KB, but nowadays 4 KB is more common.

Exercise 5 IV

4. Calculate the optimal page size p according to your model for (a) $s = 1\,\text{MiB}, e = 8\,\text{B}$
   $p = 4\,\text{KiB}$
   (b) $s = 16\,\text{GiB}, e = 32\,\text{B}$.
   $p = 1\,\text{MiB}$

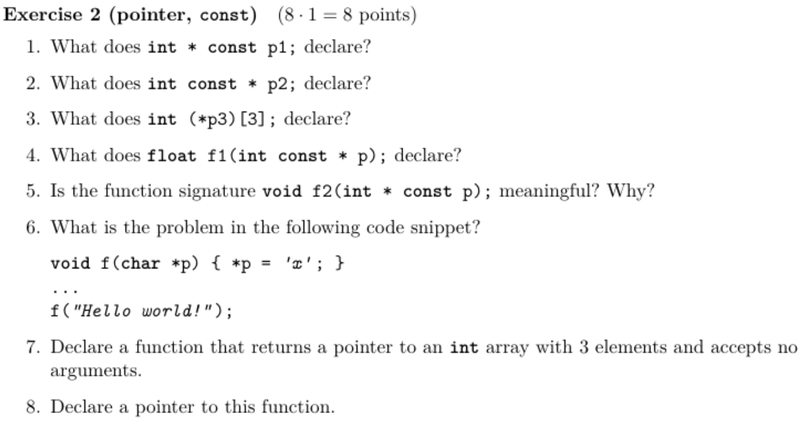
Exercise sheet 7



Exercise 1 (comprehension questions) ($8 \cdot 1 = 8$ points)

1. How could virtualization on a desktop computer be useful to developers?
2. Which of the following instruction groups is not sensitive in the sense of Popek & Goldberg (1974)? (a) Commands that access the memory management unit (MMU) (b) Instructions that access the input/output MMU (I/O MMU), (c) commands that access the Arithmetic-Logical Unit (ALU).
3. What is a privileged instruction in the sense of Popek & Goldberg (1974)?
4. Can a hypervisor virtualize more operating systems than there are (hard)disk partitions in the computing system? Why?
5. Give an example in the context of virtualization where binary translation can be faster than the original code.
6. What is the main problem with virtualization and memory?
7. What problem does the hypervisor face when it wants to evict a page of a virtualized operating system?
8. What is the purpose of balloon drivers?

- All in the book, virtualization (chapter 7)



Exercise 2 (pointer, const) ($8 \cdot 1 = 8$ points)

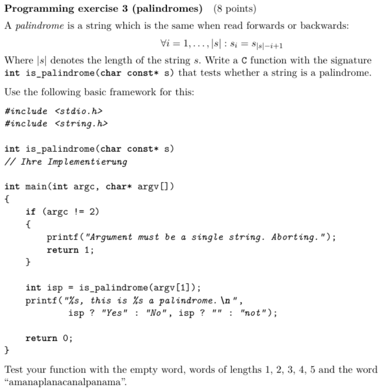
1. What does `int * const p1;` declare?
2. What does `int const * p2;` declare?
3. What does `int (*p3)[3];` declare?
4. What does `float f1(int const * p);` declare?
5. Is the function signature `void f2(int * const p);` meaningful? Why?
6. What is the problem in the following code snippet?

```
void f(char *p) { *p = 'x'; }
...
f("Hello world!");
```

7. Declare a function that returns a pointer to an `int` array with 3 elements and accepts no arguments.
8. Declare a pointer to this function.

- const binds to the next symbol to the right of the type to its left
- `char * const` a is of „type“ pointer and constant
- address the pointer stores may not be changed.

- How about `char const *a`?
- Of type char and `*a` cannot be changed
- Careful with const!



Programming exercise 3 (palindromes) (8 points)

A *palindrome* is a string which is the same when read forwards or backwards:

$$\forall i = 1, \ldots, |s| : s_i = s_{|s|-i+1}$$

Where $|s|$ denotes the length of the string s. Write a `C` function with the signature `int is_palindrome(char const* s)` that tests whether a string is a palindrome.

Use the following basic framework for this:

```
#include <stdio.h>
#include <string.h>

int is_palindrome(char const* s)
// Ihre Implementierung

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        printf("Argument must be a single string. Aborting.");
        return 1;
    }

    int isp = is_palindrome(argv[1]);
    printf("%s, this is %s a palindrome.\n",
           isp ? "Yes" : "No", isp ? "" : "not");

    return 0;
}
```

Test your function with the empty word, words of lengths 1, 2, 3, 4, 5 and the word "amanaplanacanalpanama".

- Start with the string reverse code
- add a condition in the loop to compare the characters from front and back

Bonus exercise 4 (self-study, programming exercise) (4 extra points)

Write a `C` program that outputs a seasonally appropriate animation to the text console, for example, a landscape with snow falling, a Christmas tree, or a candle burning.

For animations, it is helpful to be able to clear the screen and place the cursor back in the upper left corner of the text console. Find out about different approaches to this, for example at cplusplus.com/articles/4z18T05o, and then use ANSI sequences (en.wikipedia.org/wiki/ANSI_escape_code) as a solution available on many platforms.

A keyword to search for suitable motifs can be "ASCII art".
The POSIX function `usleep` pauses the current thread.

```
   .      .
   _\/  \/_
    _\/\/_
_\_\_\/\/_/_/_
 / /_/\/\_\ \
    _/\/\_
    /\  /\
   '      '
```

Happy holidays and a successful new year!
from the team of the "Operating Systems" course.

Image: `asciiart.website` (jgs)

References I

[1] W. Stallings, *Operating systems: internals and design principles*. Upper Saddle River, NJ: Pearson/Prentice Hall, 2009.
[2] A. Silberschatz, P. B. Galvin und G. Gagne, *Operating system principles*. John Wiley & Sons, 2006.
[3] A. S. Tanenbaum und H. Bos, *Modern operating systems*. Pearson, 2015.
[4] B. W. Kernighan und D. M. Ritchie, *The C programming language*. 2006.