Operating Systems

Tutorial 6

Fabian Klopfer

14. Dezember 2020

Intro

- Pingo Polls

Exercise Sheet 5

Exercise 1

Exercise 1 I

1. What problem does paging solve?
   - Virtual address space larger than physical
   - Paging splits virtual address space into chunks
   - Loads them in available frames in RAM.
2. What is the difference between swapping and paging?
   - Paging splits virtual address space into smaller chunks and loads them as needed. $\Rightarrow$ Small parts of processes address space.
   - Swapping stores currently inactive processes to disk and loads them if active again. $\Rightarrow$ Complete address space of process.
3. What is the difference between overlays and paging?
   - Overlays: Splitting & loading is done by hand by application programmer
   - Paging: Splitting & loading is done automagically by OS

Exercise 1 II

4. What is a page and what is a page frame?
   - Page: A chunk of continuous virtual address space
   - Frame: A chunk of continuous physical address space/RAM
5. What is the relation between the size of pages and the size of page frames?
   Normally same size.
6. Why do pages usually have a size that corresponds to a power of two?
   Addressing & Address translation in binary format/with binary ops
7. What is the function of page tables?
   Stores mapping from virtual pages to physical frames.
   Page frame address + offset = physical address

Exercise 1 III

8. What is a page fault?
   Is what happens when page is looked up in page table & the absent bit is set.
   On page fault, OS executes page fault handler to load the page from disk to frame.

9. What problem does segmentation solve?
   Eases MM by providing multiple separate address spaces to a process

10. Can paging and segmentation be combined?
    Yes e.g. Intel x86

Exercise 2

Exercise 2 I

Two-dimensional array `int X[64][64]`, four page frames each has space for 128 values of type `int`.
Calculate the number of page faults:

Array uses $\frac{64^2}{128} = \frac{(32 \cdot 2) \cdot 64}{128} = \frac{128 \cdot 32}{128} = 32$ Pages.

Exercise 2 II

```
// Fragment A
for (int j = 0; j < 64; ++j)
    for (int i = 0; i < 64; ++i)
        X[i][j] = 0;
```

- Inner loop iterates over rows, outer over columns.
- only two elements of page are accessed before next page needed
- each second access leads to a page fault

$$\frac{64^2}{2} = \frac{2 \cdot 32 \cdot 64}{2} = 64 \cdot 32 = 2^6 \cdot 2^5 = 2^{11} = 2048$$

Exercise 2 III

```
// fragment B
for (int i = 0; i < 64; ++i)
    for (int j = 0; j < 64; ++j)
        X[i][j] = 0;
```

- Inner loop iterates over columns, outer over rows.
- Two rows $X[i][]$ and $X[i+1][]$ are accessed without page fault.
- Overall 32 page faults occur.

Exercise 3

Exercise 3 I

Expression	Description
`v`	value of variable `v`
`&v`	address of variable `v`
`*p`	value at address stored by `p`
`a[2]`	value of third element of the array. Also value of the address `a + 2 * sizeof(*a)`
`**p`	value at address stored by value at address stored by `p`

Exercise 4

Exercise 4 I

Translate the following virtual addresses into physical addresses, or indicate that a page fault would occur:

1. 5555
2. 4100
3. 2048
4. 2047

Page	P/A Bit	R-Bit	M-Bit	Page frame
0	1	1	1	4
1	1	1	0	7
2	1	0	0	3
3	1	0	0	2
4	0	0	0	—
5	1	1	1	0

Exercise 4 II

1. $5555_{10} = 1010110110011_2$.

   Page ID = (Bit Mask & Virtual Address) >> Offset length

   $$(1110000000000_2 \ \& \ 1010110110011_2) >> 10 = 101_2 = 5$$

   Page Table Lookup: Page ID 5 is in Page frame 0.

   Offset = Bit Mask & Virtual Address

   $$0001111111111_2 \ \& \ 1010110110011_2 = 0110110011_2$$

   Physical address = (Page frame in binary << Offset length) | Offset

   $$\Rightarrow 0000110110011_2 = 435_{10}$$

Exercise 4 III

2. $4100_{10} = 1000000000100_2$.

   Page ID = (Bit Mask & Virtual Address) >> Offset length

   $(1110000000000_2 \ \& \ 1000000000100_2) >> 10 = 100_2 = 4$

   Page ID 4 is not present.

   $\Rightarrow$ Page fault.

Exercise 4 IV

3. $2048_{10} = 0100000000000_2$.

   Page ID = (Bit Mask & Virtual Address) >> Offset length

   $(1110000000000_2 \ \& \ 0100000000000_2) >> 10 = 010_2 = 2$

   Page Table Lookup: Page ID 2 is in Page frame 3.

   Offset = Bit Mask & Virtual Address

   $0001111111111_2 \ \& \ 0100000000000_2 = 0000000000_2$

   Physical address = (Page frame in binary << Offset length) | Offset

   $\Rightarrow 0110000000000_2 = 3072_{10}$

Exercise 4 V

4. $2047_{10} = 0011111111111_2$.

   Page ID = (Bit Mask & Virtual Address) >> Offset length

   $$(1110000000000_2 \ \& \ 0011111111111_2) >> 10 = 001_2 = 1$$

   Page Table Lookup: Page ID 1 is in Page frame 7.

   Offset = Bit Mask & Virtual Address

   $$0001111111111_2 \ \& \ 0011111111111_2 = 1111111111_2$$

   Physical address = (Page frame in binary << Offset length) | Offset

   $$\Rightarrow 1111111111111_2 = 8191_{10}$$

Exercise 5

Exercise 5 I

1. Write a function that represents an unsigned integer as a string in binary system
2. Write a function that converts a virtual memory address to a physical memory address, thus performing the core calculation of a MMU:

Exercise 5 II

```c
void binary_to_string(unsigned long int b, unsigned char k, char *s) {
    unsigned long shifted_to_place;
    for (unsigned char i = 0; i < k; ++i) {
        shifted_to_place = (b >> (k - 1 - i));
        s[i] = (shifted_to_place & 1) == 1 ? '1' : '0';
    }
    printf("%.*s\n", k, s);
}
```

Exercise 5 III

$k = 4,\ 0000110 = 6_{10}$

1. Iteration $i = 0$:

$$00000110 >> 3\ \&\ 00000001 = 00000000\ \&\ 00000001 = 0$$

2. Iteration $i = 0$:

$$00000110 >> 2\ \&\ 00000001 = 00000001\ \&\ 00000001 = 1$$

3. Iteration $i = 0$:

$$00000110 >> 1\ \&\ 00000001 = 00000011\ \&\ 00000001 = 1$$

4. Iteration $i = 0$:

$$00000110 >> 0\ \&\ 00000001 = 00000110\ \&\ 00000001 = 0$$

Exercise 5 IV

```
uint_least16_t translate_address(uint_least16_t virtual, uint_least32_t const *page_table) {
    uint_least32_t page_table_entry = *(page_table + (virtual  >> OFFSET_WIDTH));

    if (page_table_entry >> 31 == 0) {
        return PAGE_FAULT;
    }

    return (((uint_least16_t) ((page_table_entry & ~PA_BIT) << OFFSET_WIDTH)) | (virtual & OFFSET_MASK));
}
```

Exercise 5 V

$1025_{10} = 0010000000001_2$.
Page ID = (Bit Mask & Virtual Address) >> Offset length

$$(1110000000000_2 \ \& \ 0010000010001_2) >> 10 = 001_2 = 1$$

Page Table Lookup: Page ID 1 is in Page frame 7.
Offset = Bit Mask & Virtual Address

$$0001111111111_2 \ \& \ 0010000000001_2 = 0000001_2$$

Physical address = (Page frame in binary << Offset length) | Offset

$$\Rightarrow 1110000000000|00000001 = 1110000000001$$

Exercise sheet 6

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

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.
2. Name two paging algorithms that can be considered approximations of the Least-Recently-Used algorithm.
3. Name two key challenges in implementing paging systems.
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?
5. Name three advantages of virtualizing operating systems.
6. What is a "world switch"?
7. Name two differences between virtualization and containers.
8. For what reason can a hardware-based approach to virtualization (CPU support) be slower than a binary translation-based software approach?

- All in the book, later part of paging and virtualization (chapter 7)
- Virtualization vs. containers: VirtualBox vs. Docker

Exercise 2 (address spaces) $(2 + 1 + 2 + 1 = 6$ points)

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?
2. A computing system uses virtual memory with 48-bit wide addresses and 8 KiB sized pages. How many pages does a single-level linear page table contain?
3. A computing system has a virtual address space of 64 KiB, divided into pages of size 4 KiB. Each page can contain either program code, data or stack (no mixed usage). 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? Justify your answer.
4. The same computational system now uses pages of size 512 B. Does the process now fit into the address space? Justify your answer. You can assume that the address space is fully available to the process.

- Tanenbaum p. 236
- What happens if an instruction is at the boarder of two pages?
- Pay attention on internal fragmentation

Exercise 3 (paging algorithms) $(4 \cdot 2 = 8$ points)

The table shows the four page frames of a computer. Loading time and time of last access are given in timer intervals.

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

Which page do the following algorithms replace?

1. First-In First-Out
2. Second Chance
3. Least-Recently-Used
4. Not-Recently-Used

You can assume that the Least-Recently-Used algorithm only uses the R-bit and ignores the M-bit.

- FIFO depends on loading time
- second chance on R-bit and loading time
- LRU on Access time
- NRU on R and M bit.

Exercise 4 (pointers and arrays) $(6 \cdot 1 = 6$ points)

What output does this program produce? Explain your answer.

```
#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;
}
```

- Print it and make sense of it
- remember that increments with pointers: $+1$ might also mean $+4$ bytes or $+$ subarray size.

Bonus exercise 5 (optimal page sizes) $(2 + 1 + 2 + 1 = 6$ extra points)

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). For this, consider the memory consumption by the page table itself as well as the internal fragmentation, that is, unused memory within pages. For the latter, you can assume that (a) the fraction of used memory is uniformly distributed within pages that are not fully used, and (b) the memory has been compacted, so the process occupies only one contiguous block of memory.
2. Interpret your expression in terms of page size p.
3. Determine the optimal page size p in your model.
4. Calculate the optimal page size p according to your model for (a) $s = 1\,\text{MiB}, e = 8\,\text{B}$ and (b) $s = 16\,\text{GiB}, e = 32\,\text{B}$.

- Tanenbaum p. 226/227

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.