Department of Computer Science

TDT4186 - Operating Systems

Solution 8

Exam Prep: A bit of everything

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman
Daniel Hansen

Handout: 18.04.2023

Introduction

Since this is the last exercise of the semester, it will contain questions with respect to a wide variety of topics discussed in the course. If you have problems with one or the other question, it might be an indicator of which subjects you should revise again before the exam.

For the multiple-choice questions, there might be multiple correct options. Select all that are correct.

1 C Programming

In the exam, we won't ask you to program a syntactically correct and compileable program. However, we expect that you know, can read, and understand C program code.

Includes have been omitted for brevity in the following samples.

1. **Pointer arithmetic:** Which of the following statements is true? (+1P/correct selected answer, -0.5/wrong answer, no negative points overall)

```c
typedef struct proc
{
    uint64_t pid;
    char name[16];
    uint64_t parent_pid;
} proc_t;

int main(int argc, char *argv[])
{
    proc_t procs[4];

    for (int i = 0; i < 3; i++)
    {
        procs[i].pid = i;
        strncpy(procs[i].name, "proc ", 16);
        procs[i].name[5] = (uint8_t)(i + 48);
        procs[i].name[6] = '\0';
        if (i > 0)
            procs[i].parent_pid = i - 1;
        else
            procs[i].parent_pid = 0;
    }
    proc_t *elem = (proc_t *)((procs + 1));
    *(((uint32_t *)((&(elem->parent_pid)) + 2)) + 2) = UINT32_MAX;

    for (int i = 0; i < 3; i++)
    {
        printf("%s (%ld), parent: %ld\n",
               procs[i].name, procs[i].pid, procs[i].parent_pid);
    }
}
```

Listing 1: Pointer Arithmetic Sample

☒ It outputs
```
proc 0 (0), parent:  0
proc 1 (1), parent:  0
proc 2 (2), parent:  1
```

☐ First two lines as above, but it crashes with a SEGFAULT when trying to print the third line

☐ It segfaults when assigning on line 24

☐ It segfaults on line 16

☐ First two lines as in the first option, but the third looks like this
`proc□□□□ (2), parent:  1`

☐ It does not compile because on line 23 an array is assigned to a pointer

2. **Bitwise Operators:** Which of the following statements is true? (+1P/correct selected answer, -0.5/wrong answer, no negative points overall)

```c
1  typedef uint32_t pte_t;
2  #define FLAG_MASK 0x3FF
3  #define FLAG_BITS 0x8
4  #define PA(pte) ((pte >> FLAG_BITS) << FLAG_BITS)
5  #define PTE_V (0x1 << 0)
6  #define PTE_R (0x1 << 1)
7  #define PTE_W (0x1 << 2)
8  #define PTE_X (0x1 << 3)
9  
10 int main(int argc, char *argv[])
11 {
12     pte_t entry1 = 0xC9CF4BD3;
13     pte_t entry2 = 0xC9CFFFD3;
14 
15     if (!(PTE_V | entry1) || !(PTE_V | entry2))
16     {
17         return 1;
18     }
19     else
20     {
21         entry1 = entry1 | PTE_X;
22         entry2 = entry2 | PTE_W;
23     }
24 
25     uint32_t pa1 = PA(entry1);
26     uint32_t pa2 = PA(entry2);
27 
28     uint32_t diff = pa1 ^ pa2;
29     if (diff == 0)
30     {
31         return 2;
32     }
33     uint8_t len = 0;
34     while (diff > 0)
35     {
36         diff = diff >> 1;
37         len++;
38     }
39     printf("%d\n", len);
40 }
```

Listing 2: Bitwise Operations

☒ Line 21 and 22 each change a single bit in the respective entries

☐ The variable diff contains the value 1 at the end of the execution

☐ It will not compile because it can't cast `pte_t` to `uint32_t` in line 25

☐ It will compile but will exit on line 17, returning 1

☐ It will compile but it will exit on line 31, returning 2

☒ It will compile and reach the printf on line 39

3. What output will the program in Listing 2 generate? (3 Points)

   Answer: *Let's go through the program one by one: entry1 and entry2 both have the least significant bit set to 1, so we do no return on line 17. In the else clause we set bits 4 resp. 3 to one, but that does not have an effect on the rest of the execution, as in the next step, we shift both entries 8 bits to the right, before shifting them back 8 bits, effectively zeroing out the 8 least significant bits. On line 28, we calculate the exclusive or (a bit is only 1 if one of the corresponding bits is 0 and the other is 1). If we take a look at the definitions in line 12 and 13, we now see that the first four numbers match, so those will be zero. We zeroed out 8 bits previously, so the last two digits will also be 0. Thus we have only the two digits in the middle that we need to compare. Entry2 is all 1's in those bits, so we can just invert the binary representation of 0x4B, which yields 0xB4. So diff has the value 0xB400 (remember the correct position). Now this is shifted bit by bit, so we are counting how many bits there are in 0xB400. Since we are using hexadecimal representation, each digit represents four bits, so len will have the value 16 in the end.*

4. **SPOILER ALERT: IF YOU HAVEN'T SOLVED THE FIRST EXERCISE, THE SOLUTION IS CONTAINED IN THE FOLLOWING QUESTION**
   Please explain where the value that is assigned in Listing 1 is gone. Why don't we see it in the output? (3 Points)

   Answer: *We are assigning the value UINT32_MAX to the third byte of the third entry in the procs array (on line 23 we get the pointer to the second entry of the procs array, on line 24 we get the pointer to the parent pid in that struct, increase the pointer by two uin64_t pointers, thus now pointing to the name array. Then we cast it to a uint32_t pointer, to which we add another two (so we now have a pointer to the 8th element of the name array), which we cast to be reinterpreted as a uint32_t and assign our value to those 4 bytes). In C, strings are null-terminated, so we know where the string stops (remember, C arrays don't have a length). Since there is a null character at position 6 in the array, the printf function will stop reading the array at that point, ignoring any values that are stored later on.*

2 Scheduling, Processes, and Threads

1. Which of the following statements is true? (+1P/correct selected answer, -0.5/wrong answer, no negative points overall)

   ☐ Cooperative scheduling relies on hardware support to turn over control to the scheduler

   ☒ Direct execution always runs a program to completion/forever

   ☐ Processes and Threads are different names for the same concept

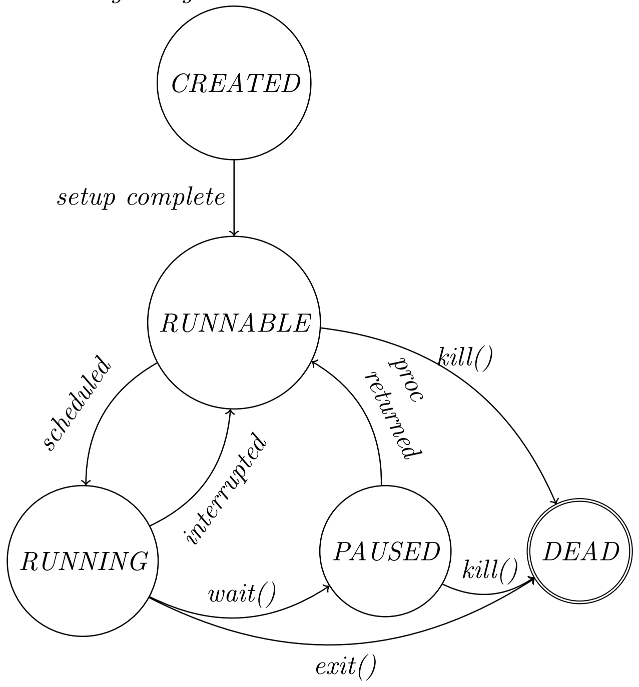
   ☐ The OS kernel runs in its own process

   ☒ Timer interrupts are only required for preemptive schedulers

   ☐ A process can directly access hardware and shared resources

2. Assume that we have a simple operating system which supports processes. Those processes can have the following states: CREATED, RUNNABLE, RUNNING, PAUSED, DEAD. Draw a state diagram of a processes lifecycle. Please label the edges in your diagram with which actions will lead to this transition. (2 points)

   Answer: *There might be more than one possible answer here, you should have come up with* *something along these lines:*



3. You are tasked with writing a small operating system for a handheld device. The user will be able to install and run third-party software (coming from developers you don't know). The device should be low power, and thus can at any point in time only run two processes to improve battery live. Which approach do you choose to enable multi-processing? What do you need to pay attention to?

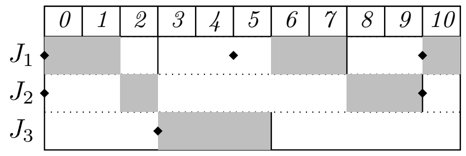
   Answer: *The key is that there will be third-party software running on our operating system, that we cannot verify. Thus, we need to have a preemptive scheduler, which enables our OS to regain control, even if a process does not work as intended. Additionally, we need to synchronise on the accesses to the process data structures, as there are two execution units (two processes can run, which also means that two scheduling decisions might be taken at the same time).*

4. You are working on a embedded system for self-driving cars. It needs to run a set of tasks periodically and complete each task before its deadline. Is it possible to schedule the jobs in Table 1 such that all jobs meet their deadline or do you need a second device to make it work? The arrival time describes when a task is added to the system for the first time, the execution time describes how long a task takes, the deadline describes how many time units after arriving a job needs to be finished, and the interval describes after how many time units another instance of the same job is added to the system (e.g. for Job 3, the next instance will be added at timeslot 13).

Job ID	Arrival Time	Execution Time	Deadline	Interval
1	0	2	3	5
2	0	3	10	10
3	3	3	3	10

Table 1: Jobs with deadline and interval

   Answer: *We use the STCF modification from exercise 2 and extend it in the sense that we do no longer assume the arrival time to be 0 for all the jobs. The schedule that we arrive at looks like the following one and will repeat indefinitely.*



3 Memory Management

1. Take a look at the small program snippet below. Which of the following statements are true? Select all that you think are true. (1P for each correctly selected/not selected option, -0.5P for each wrongly selected/not selected option, minimum points overall: 0)

```
1  int add(int a, int b) {
2      int sum = 0;
3      sum += a;
4      sum += b;
5      return sum;
6  }
7
8  int main(int argc, char *argv[])
9  {
10     int size = 2;
11     if (argc > 1)
12         size = argc;
13     int *all_ints = malloc(argc * sizeof(int));
14     for (int i = 0; i < size; i++)
15         *(all_ints + i) = add(argc, i);
16     printf(”all␣ints:␣”);
17     for (int i = 0; i < size; i++)
18         printf(”%d,␣”, *(all_ints + i));
19     free(all_ints);
20 }
```

Listing 3: Sample Program, includes are left out for brevity

Answer: *Quick explanation to why the options below are correct/incorrect:*

(a) The string for printf on line 16 is a constant, thus stored in the code section

(b) The memory pointed to by all_ints is allocated using malloc, which allocates memory on the heap

(c) all_ints is actually a local variable, which stores a pointer. The pointerr is stored on the stack (but points to the heap)

(d) The size variable is a normal local variable that is stored on the stack

(e) The sum variable is a normal local variable thus stored on the stack

(f) Functions don't have their "own" stack, they use different parts of the stack (by pushing new frames onto the stack)

☐ The string for the printf on line 16 is stored in the heap

☒ The memory pointed to by all_ints lives on the heap

☒ Since all_ints is a local pointer variable it lives on the stack

☐ The size variable is allocated on the heap because it is initialised with an argument value

☒ The sum variable is allocated on the stack

☐ The sum function has its own stack that is independent of the main function stack

2. We learned that the stack grows downwards whenever new things are pushed onto the stack and shrinks upwards once they are popped off. How does the heap grow and shrink?

   Answer: *The responsibility for the heap resides with the programmer or the runtime. In C, the programmer uses malloc to allocate new memory and free to give it back again, in other languages such as Java or Python the runtime manages that part, but under the hood uses the same syscalls as malloc and free do to (de)allocate memory. Memory is usually allocated by the page (as this is the smallest unit that can be mapped in the pagetables), thus if smaller memory regions are requested, this is not handled by the operating system but by the userlibrary.*

3. You are developping a new small operating system for an embedded device that runs a predefined set of processes. How do you partition the memory within the operating system between the processes? Give an explanation for your answer.

   Answer: *Possible answer: Since the set of processes is predefined, we might have a pretty good grasp of how much memory each of these processes need. Thus, I would opt for a fixed partitioned memory, where we assign a predefined chunk of memory to each process. That reduces the complexity and thus also the energy requirements (embedded devices are often energy constraint).*

4 Paging, Page Tables & Translation Lookaside Buffer

In the exercises in this section we use the following page tables and page directory in tables 2 to 5.

Index	20	12
	. . .	
0x034	0xDEAD0	flags
	. . .	
0x300	0xBEEF0	flags
	. . .	
0x3FB	0x2BAD2	flags
	. . .	

Table 2: Page Directory stored at 0xBADCAFE

	20	12
	. . .	
0x000	0x00000	flags
	. . .	
0x06F	0x11111	flags
	. . .	
0x0CC	0x22222	flags
	. . .	

Table 3: Page Table stored at 0xDEAD0000

	20	12
	. . .	
0x000	0x33333	flags
0x001	0x66666	flags
	. . .	
0x10F	0xCCCCC	flags
	. . .	

Table 4: Page Table stored at 0xBEEF0000

	20	12
	. . .	
0x1CE	0x99998	flags
	. . .	
0x2D2	0x33330	flags
	. . .	
0x2DC	0x66660	flags
	. . .	

Table 5: Page Table stored at 0x2BAD2000

1. Looking up physical addresses in a page table structure for every access is inefficient, especially for deeper nested page table structures. Suggest a possible solution to improve the virtual to physical address translation speed for the most commonly used addresses.

   Answer: *We could use some structure to cache a translation to reuse it later. Many accesses happen one after another (e.g. looping through an array), so we will hit the same page repeatedly. Thus a simple one-to-one can go a long way in reducing the latency for the translation of virtual to physical addresses, which is what translation look-aside buffers (TLB)* *do for us.*

2. Translate the virtual address `0xC010FF00` to a physical address and explain how you end up at your solution. The Page Directory and the Page Tables have the same amount of entries and a Page contains 4096 bytes.

   Answer: *We are examining the most significant ten bits first, which equates to* ***`0x11 0000 0000`****. Translating this to hexadecimal gives us 0x300, which is the index into the page directory. The value at this index gives us the address of the page table for the next step of the translation. So we are looking at Page Table at address 0xBEEF0000. To index this table, we use the next ten bits of the virtual address, which are* ***`0x01 0000 1111`****, in hex 0x10F. Looking at the address stored there, we now got the upper part of the address to be 0xCCCCC and take the lower 12 bits of the virtual address (index into the page) to obtain the full address 0xCCCCCF00. We divide the address into 10/10/12 blocks for indexing and the offset. The offset needs to be 12 bits, since we need to be able to address every single byte in a page (*$lg2(4096) = 12$*). After that we know that both, Page Directory and Page Table have the same amount of entries and thus require the same amount of index bits, thus we split the remaining 20 bits in half for each index.*

3. Given the physical address 0x00000115, what is the corresponding virtual address?

   Answer: *We split the address into offset (the last three digits, 0x115), and the entry that we expect in the page table (0x00000). Now we check which page table contains a corresponding entry (the page table at 0xDEAD0000 does). We get the index for that entry (0x000). Then we check which entry in the Page Directory points to the page table and get the index (0x034). Now we can shift all the indexes into place and add them up together with the offset to get the physical address:* $0x034 \ll 22 + 0x000 \ll 12 + 0x115 = 0x0D000115$

4. You are given the following Page Directory/Page Table configuration. How many bits must an address contain for this configuration to be sound?

   - Page Directory: 64K Entries
   - Page Table: 4K Entries
   - Page Size: 8KB

   Answer: *The address requires 13 bits for the page offset (*$log_2(8K) = 13$*), 12 bits for the page table index (*$log_2(4K) = 12$*), and 16 bits for page directory index (*$log_2(64K) = 16$*). This sums up to 41, so an address requires at least 41 bits to address every single entry in all the tables/bytes in the pages.*

5. What is the maximum size of memory the system could handle in the previous subtask? Calculate it based on the size of the address and the page directory/page table.

   Answer: *If we calculate it by the address size we end up with* $2^{41}B = 2TB$.
   To calculate it based on the page tables, we multiply the number of page directory entries by the number of page table entries and multiply this finally with the page size. That gives us the maximal amount of memory that could be managed with this structure. This yields $64K * 4K * 8KB = 2TB$. *This should be no surprise as we store as much information in the page tables as we use bits in the address.*

5 Processes, Threads & Synchronisation

1. Which of the following statements about locks is true?

   ☐ To make sure only processes that are allowed to access a resource are accessing it

   ☐ We don't really need locks, this is to better communicate the ordering in code with other developers

   ☐ To ensure sequential consistency

☐ We don't necessarily need locks; we can replace all locks by lock-free data structures

☒ To exclude others (processes, threads) from changing certain state during a limited timeframe

☐ Locks are only required when multi-threading as processes are isolated from each other

2. Explain the difference between a process and a thread. What is the main distinguishing factor between the two?

   Answer: *Processes are relatively isolated from each other (e.g. they don't share the same address space), and thus sharing information between multiple processes might be more expensive and cumbersome. On the contrary are the threads that all operate on the same address space and thus information sharing between threads comes pretty much for free, as everything is accessible to each other.*

3. Go to https://deadlockempire.github.io/ and go through the samples there. Samples like that might be part of the exam.

   Answer: *No solution - the website will tell you once you got it right*

6 I/O Scheduling, DMA & RAID

1. We discussed I/O scheduling mainly to reduce seek time on spinning hard disk drives. However, many modern systems use SSDs instead of HDDs. So is I/O scheduling in SSD only systems still important?

   Answer: *While we looked at I/O scheduling mainly to improve performance of HDDs in this course, it has proven to be a useful technique for multiple I/O requests. Modern server systems have more varied storage systems as storage might be attached on different layers and thus expose vastly different properties. So the technique of I/O scheduling is still useful and might be used for scheduling I/O for different devices.*

2. Which of the following statements is true?

   ☐ RAID is a technique to reduce latency when reading data from HDDs

   ☒ RAID 5 improves the reliability and the throughput when reading/writing HDDs

   ☐ DMA allows a device to access HDD storage directly, bypassing the CPU

   ☐ RAID is only useful if we expect a disk to fail

   ☐ RAID 0 improves the reliability of the system

   ☐ RAID 5 gives us more reliability than RAID 4

3. You are tasked with increasing performance in a system. You are allowed to make changes to the hardware as well. You see from the analysis that the CPU spends the most time just moving data from the network card to the memory. What could you do instead? How does this affect the system and how would it complicate the hardware design? Does this increase the systems concurrency?

   Answer: *Since the CPU is busy moving data, you could instead use a DMA controller to move the data from the network card to main memory and only interrupt the CPU if the copying is complete. This now means that the CPU is free to compute further things while network I/O is in progress. However, moving data from the network card to memory needs to be synchronised with the CPU's accesses to either the BUS or the memory (you can't write/read simultaneously, as the Bus can only hold a single value at a time). This increases the systems concurrency, as now both tasks (computation and I/O) can make progress, but it increases contention on the Bus.*

7 File System & Unix I/O

1. Which of the following statements is true?

   ☐ An inode contains the filename

   ☐ Filesystems relatively constrained in design space as they have to follow the hardware's structure

   ☐ In order to guarantee consistency, the OS must always resolve paths from the inode structure

   ☒ We use bitmaps to indicate if an inode is used or not

   ☐ Multi-level indexes improve performance, since one can use binary search to search for a file in parallel

   ☐ The maximal file size is only limited by the amount of free storage on disk

2. Assume the numbers given for the Barracuda Disk in Figure 37.5 in chapter 37.4[1]. How long does it take to move a folder with 8 files, each file being of the size of 128 KB, into another folder? So how long does it take to execute `mv /a/f /b`, where `/a/f` is a directory, containing 8 files of the size 128 KB.

   Answer: *We are not copying the files physically, but we change the inode information, such that the folder now resides in another folder. For that, the sequence looks like the following:*

   (a) *read the root inode (to find the root data)*

   (b) *read the root data (to find out where `/a` and `/b` is stored)*

   (c) *read the `a` inode to find `a` data*

   (d) *read the `a` data to find `f`*

   (e) *read the `f` inode (Check permissions)*

   (f) *read the `b` data*

   (g) *write the `b` data (adding f to b)*

   (h) *write the `/b` inode*

   (i) *read the `/b/f` inode*

   (j) *write the `/b/f` inode*

   (k) *write the `/a` data*

   (l) *write the `/a` inode*

   This leaves us with 12 accesses to move the folder (independent of the content size), as we only have to rewrite the inode structure. Thus a move on the same device can be very fast. Since the amount of data written is negligible, we are just multiplying the number of accesses with the average access latency (which is most likely an overestimate, as some of the structures live close by and can be read/written in one go). The average access latency is given by the average seek latency (9 ms) plus the average rotational delay (4.16 ms). This yields $12 * 13.16ms = 158ms$. *So moving on the same disk can be done almost instantly since we do not need to copy any data.*

3. How would the speed change in the previous task if instead we were using an SSD? Assume the speeds given in chapter 44 of the ostep book, page 18. If you need latency, you can assume 0.5 ms. Explain why there is a difference in speed.

[1] You don't need to know these numbers by heart, we will provide whatever numbers you need to know during the exam.

Answer: *The main advantage of the SSD over the HDD is the huge decrease in latency. So in this case, we have* $14 * 0.5ms = 7ms$ *overall. The difference in speed is a result of the SSD not having any physical parts that need to be moved in place before an operation can be executed.*

4. Propose an inode structure such that we can store files up to 8 GB, assuming a block size of 4KB and addresses of size 64 bit. How much overhead does this introduce?

   Answer: *A possible solution uses eight double-indirect nodes, which means we can contain (let b be the block size)* $8 * (b/8)^2 * b = 8GB$*. In terms of overhead, we use additional blocks for the indirection. So we need to allocate* $8 * (b/8 + 1)$ *blocks. This results in an overhead of roughly 16 MB overall. This is negligible in comparison to the file size.*
   We can calculate it that way: $8GB/4KB = 2^{21}$ *blocks are needed. A single block can contain 512 pointers (4KB/8B). Using that* $512 = 2^9$*, so by using two levels of indirections, we get that we need* $2^{21}/(2^9)^2 = 2^3 = 8$ *pointers in the first place*

5. You created a new device that accelerates machine learning computations and you want to make it work under Linux, as most of today's machine learning happens on Linux machines. How would you expose that to the userspace? How would this be integrated into the operating system?

   Answer: *In Unix based systems, everything is a file. To be consistent with that interface, we will also expose our new device as a file under* `/dev/`*. We then can provide a library on how to communicate with the device as a user-space process in order to make use of its blazingly fast computations.*