Graded Exercises

COMP 310 / ECSE 427 Winter 2025

1 Syscalls [10 points]

1.1 Syscall Wrappers [5 points]

When a process makes a system call through the kernel API library with a pointer as an argument, does the validity of that pointer need to be checked in the library, in the kernel, or in both places? Justify your answer.

1.2 Forking [5 points]

How many times will the below program print `hello5`? Explain why (e.g. by drawing the process tree).

```
1  int main() {
2    if (fork() != 0) {
3      if (fork () != 0) {
4        printf("hello1\n");
5      } else {
6        fork();
7      }
8      printf("hello2\n");
9    } else {
10     if (fork() != 0) {
11       printf("hello3\n");
12     } else {
13       fork();
14     }
15     printf("hello4\n");
16   }
17   printf("hello5\n"); // <-- this one!
18 }
```

2 Synchronization [10 points]

Assume that a finite number of resources of a single type must be managed in a multi-threaded program. Threads acquire a number of these resources and – once finished – return them. The following program segment is used to manage a finite number of instances (`MAX_RESOURCES`) of an available resource. When a thread wishes to obtain a number of resources, it invokes the `decrease_count` function. When a thread wants to return a number of resources it calls the `increase_count` function.

```
1  #define MAX_RESOURCES 5
2  int available_resources = MAX_RESOURCES;
3
4  int decrease_count(int count) {
5    if (available_resources < count) {
6      return -1;
7    } else {
8      available_resources -= count;
9      return 0;
10   }
11 }
12
13 int increase_count(int count) {
14   available_resources += count;
15   return 0;
16 }
```

2.1 A Race Everyone Loses [1 point]

Unfortunately, the above program fragment causes race conditions. Identify the data involved in the race condition(s).

2.2 Don't Bet on the Trifecta [2 points]

Clearly identify the location(s) in the code where the race condition(s) occur.

2.3 Save the Track [7 points]

Fix the race condition. You can either write **pseudo-code** to replace the program above, or clearly explain how to modify the code. (You can refer to C library functions and values seen in class.)

3 Compaction [5 points]

This question asks you to do a bit of math. Specifically, we want to see that you understand *what compaction is doing* and how that affects the scenario.[1] We do not need to see that you can accurately do arithmetic. Use a calculator if you wish, and give an approximate answer (say, two digits of precision).

3.1 Turbocharge the matter compressor [5 points]

(Yes, that is a Futurama joke.)
A memory management system eliminates holes by compaction. Assume that a random distribution of holes and many segments means that very nearly the entire memory must be copied. Assume we can read an 8-byte value from physical storage in 4 nanoseconds (ns) and that we can also write 8-byte values in 4ns.
How long does it take to compact 4GB of memory? Explain your reasoning.
Hint: does a copy require only a read, only a write, or both?

4 Virtual Memory: Segmentation [10 points]

For this question, all numbers prefixed by `0b` and in that font are in binary. Therefore, `0b100` is the decimal number 4.
A CPU boasts 5-bit virtual addresses and 16 bytes of byte-addressable physical memory.[2] There are three segments currently mapped, with the following segment table:

Seg#	Base	Bound	R W
0	`0b1100`	`0b100`	0 1
1	`0b0000`	`0b000`	0 0
2	`0b0001`	`0b011`	1 1
3	`0b1000`	`0b010`	1 0

4.1 Cartography [1 point each]

Give either the corresponding physical address, or the phrase "segmentation fault," for each of the following accesses. If the access faults, briefly explain why. (You may draw a diagram of the physical memory if that makes it easier to explain.)

1. `0b00000` read
2. `0b11001` read
3. `0b01000` read
4. `0b10010` read
5. `0b11010` read
6. `0b00111` write
7. `0b10001` write
8. `0b00000` write
9. `0b11000` write
10. `0b10100` write

[1] Note also that these numbers are not realistic.
[2] Quite the boast, huh? Don't tell them.

5 Virtual Memory: Paging [15 points]

We will consider two machines in this question. (There is a third part on the next page.) The first machine has 32-bit virtual addresses. The machine supports paging with a 4KB page size and a 2-level paging scheme. The page directory[3] contains 1024 entries.

5.1 This Sounds Oddly Realistic [2 points]

How many entries are there in a single inner-level page table? Justify your answer.

5.2 Spatial Locality [5 points]

A program running on this first machine is only sparsely using its address space. The addresses between 0 (inclusive) and 18MB (exclusive) are mapped. Additionally, the addresses between 90MB and 117 MB are also mapped.

What is the minimum number of inner-level page tables needed? Justify your answer and take special care to consider whether or not 90 and 117 are on page directory boundaries. (Again, we want to see that you know how to approach the problem, not that your arithmetic is accurate.)

[3] Recall "page directory" is the name for the outer-level page table.

5.3 Chasing Stars [8 points, 2 each]

This question uses some hexadecimal numbers, formatted like `0xf3`.

A second machine has 6-bit addresses (both virtual and physical). It has 64 bytes of byte-addressable physical memory. The designers heard about the cool two-level paging technique and decided to implement it. The page size is 4 bytes; each entry is a single byte. Like usual, the page tables and the page directory are themselves page-sized. Therefore, each one contains 4 entries. Overall, this was probably a bad idea, because the page tables take up a large proportion of the machine's small memory.

The physical memory contains only $64/4 = 16$ frames. Therefore, page table entries use only the the bottom four bits (one hexit) to store a frame number. The other bits are used for metadata.[4] **If the metadata bits are all zero, the entry is invalid.**

At right, the first 8 frames of physical memory are shown. The page directory occupies **frame number 2**. (Frame numbers are 0-indexed, so the frame numbered 2 is **not** the second frame!) For each of the following **virtual** addresses, perform the page table walk to determine either the corresponding physical address or "address invalid." Briefly justify each step of the walk by giving the physical address of **each** lookup. Giving only the final translated address is not enough for full credit.

You may give addresses either in binary or as (hexadecimal) "frame number, offset" pairs. For example, `0b110110` and `d,2` denote the same address.

Addr	Value
0b000000	0xd1
0b000001	0xf7
0b000010	0x6f
0b000011	0x00
0b000100	0x53
0b000101	0x65
0b000110	0x63
0b000111	0x72
0b001000	0x00
0b001001	0xf0
0b001010	0xf5
0b001011	0xf3
0b001100	0x00
0b001101	0xca
0b001110	0x44
0b001111	0xc8
0b010000	0xaf
0b010001	0x05
0b010010	0xf5
0b010011	0x45
0b010100	0xd9
0b010101	0x00
0b010110	0xfc
0b010111	0x5d
0b011000	0xef
0b011001	0x59
0b011010	0xee
0b011011	0xd6
0b011100	0x65
0b011101	0x74
0b011110	0x21
0b011111	0x00
...	...

Hint 1: The virtual addresses are given in binary so that you can easily separate them into parts. There are two bits for each of the offset, page table index, and page directory index. (You could have deduced this from the other given information.)

Hint 2: The top 4 bits of all of the physical addresses in a frame are the same. That is not a coincidence! That's the frame number in binary.

1. `0b000000`
2. `0b110010`
3. `0b101101`
4. `0b010111`

By the way, there is a secret hidden somewhere in the virtual address space. See if you can find it!

[4] For example, read/write permissions. But we will ignore that here.