ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
CS–202 Computer Systems
Argyraki K., Kashyap S. & Chappelier J.-C.

CS–202 COMPUTER SYSTEMS

Midterm solution

April 7th, 2025

INSTRUCTIONS (please read carefully)

IMPORTANT! Please strictly follow these instructions, otherwise your exam may be canceled.

1. You have one hour and forty-five minutes to complete this examination (1:15–3:00pm).
2. You must **use black or dark blue ink**, neither pencil nor any other color.
3. This is closed book exam.
   Personal notes, two times dual-sided A4 sheets (4 sides in total), allowed.

   On the other hand, you may not use any personal computer, mobile phone or any other electronic equipment.
4. Answer the questions directly on the exam sheet; do not attach any additional sheets; only this document will be graded.
5. Carefully and *completely* read each question so as to do only what we actually ask for. If the statement seems unclear, or if you are in any doubt, ask one of the assistants for clarification.
6. The exam consists of four independent exercises, which can be addressed in any order, but which do not score the same (points are indicated, the total is 105 points); all exercises count for the final grade.

LEAVE THIS EMPTY				
Question 1	Question 2	Question 3	Question 4	**TOTAL**
20	25	25	35	105

Question 1 Accessing a file [20 points]

Consider a file system with the following properties:

- block size of 4KiB (4096 bytes);
- each inode contains metadata and 10 direct pointers to blocks that contain data.

Consider the following C program excerpt:

```
 1  ...
 2  const char* filename = "/foo/bar/testfile.txt";
 3
 4  int main()
 5  {
 6      const size_t N = ...;
 7      char buffer[N];
 8
 9      for (size_t i = 0; i < N; ++i) { buffer[i] = ...; }
10
11      int fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
12      ...
13
14      ssize_t bytes_written = write(fd, buffer, N);
15      if (bytes_written < 0) {
16          return 1;               // Write failed.
17      }
18      ...
19
20      close(fd);
21      return 0;
22  }
```

Suppose you compile it and invoke it on your computer. Assume you have all the necessary permissions and that file "`/foo/bar/testfile.txt`" already exists.

① **[1 point]** How many processes and how many threads does the Operating System (OS) of your computer create as a result of you invoking this program?

② **[2 points]** At the moment when the CPU is executing the first instruction of this program:
(a) Where is the Instruction Pointer (IP) register pointing at?
(b) In what mode/level is the CPU?

③ **[2 points]** Identify all the calls that cause the CPU to switch from one mode/level to a different one while executing this program. Next to each call, say why it causes a switch in one sentence.

④ **[10 points]** Let's define a "disk access" as an instance where the CPU tells the disk controller that it wants to read or write a sequence of bytes from/into a contiguous area of the disk.

Assuming the `if` statement on line `15` is false, how many disk accesses does this program trigger? For each disk access, state what is read from or writen to the disk.

⑤ **[5 points]** Given the properties of this file system, what is the smallest value of `N` that will necessarily make the `if` statement on line `15` true? **Justify** your answer in 1–3 sentences.

Answers:

① One process and one thread (the process's main thread). Since this program does not fork, the OS does not create more than one process.

② **(a)** The IP register is pointing inside the text segment of the process.
(b) The CPU is in user mode, because it is executing an instruction from a process's text segment.

③
- `open`: It cases a mode switch because it is a system call.
- `write`: Same reason.
- `close`: Same reason.

④ The `open` syscall triggers:
- A read of the inode of "/" (root directory).
- A read of a data block of "/".
- A read of the inode of "/foo".
- A read of a data block of "/foo".
- A read of the inode of "/foo/bar".
- A read of a data block of "/foo/bar".
- A read of the inode of "/foo/bar/testfile.txt".
- (Not expected) A write to the inode of "/foo/bar/testfile.txt" because the file is being truncated.

The `write` syscall triggers:
- A write to each of $\lceil \frac{N}{4096} \rceil$ data blocks.
- A write to the inode of "/foo/bar/testfile.txt".

⑤ The minimum value of N that will certainly cause an error is 40961.

Rationale: The maximum file size is $10 \times 4096 = 40960$ bytes, because each inode contais 10 direct ponters to data blocks, and each block fits 4096 bytes. Hence, any $N > 40960$ will cause an error.

Question 2 Memory virtualization [25 points]

Consider a computer system that uses paging for memory virtualization and has the following properties:

- each page consists of 128 bytes;
- each virtual address consists of 10 bits in total
  (including the bits used to identify the offset within each page).

① **[1 point]** How many bits are used to identify the offset within each page?
Justify your answer in 1–2 sentences.

② **[1 point]** How many bits are used to index into a page table?
Justify your answer in 1–2 sentences.

③ **[1 point]** Assume linear (single-level) page tables.
Compute the minimum size of each page table.

④ **[2 points]** With the information you have until this point, can you compute the size of the computer's physical memory? If yes, compute it. If not, explain why not in 1–2 sentences.

⑤ **[10 points]** The following table shows you the inputs and outputs of the Memory Management Unit (MMU) at different points in time (input and output are in binary):

Time tick	MMU Input	MMU Output
0	0010000000	00110000000
1	0110000111	00010000111
2	0001010100	01011010100
3	1000011001	00100011001
4	0100111000	01100111000
5	1100011011	01110011011
6	1110000000	01000000000
7	1011000100	00001000100
8	0001001100	10101001100
9	0011110000	10001110000
10	0011010000	10001010000

From time tick 0 up to and including time tick 7, the CPU runs the same process, P1.
Draw as much as you can of P1's page table.

⑥ **[5 points]** If all processes running in this computer system have **similar memory images** as process P1, is there any benefit in using two-level page tables (as opposed to linear page tables)?
Justify your answer in 1-3 sentences.

⑦ **[5 points]** By looking at the table above, can you tell whether the CPU continues to run process P1 or switches to a different process from time tick 8 and onward?

Suppose the timer interrupt occurs every X time ticks. Do you have enough information to compute an upper bound for X?
If yes, compute it. If not, explain what extra assumptions you would need to make about process P1 in order to compute it.

Answers:

① Each page contains 128 bytes, hence we need $log_2 128 = 7$ bits to identify each byte within a given page.

② Given the previous answer, there remain $10 - 7 = 3$ bits for indexing a page table.

③ Each page table contains $2^3 = 8$ entries. Assuming that each PTE has a minimum size of 1 byte, each page table has a minimum size of 8 bytes.

④ No, at this point there is not enough information for computing the size of physical memory. We only know the size of a virtual memory address (which may not be the same as the size of a physical memory address), as well as the size of the offset (which gives us the size of a physical frame, but not the total number of frames).

⑤

Physical Frame Number	Valid bit
0101	1
0011	1
0110	1
0001	1
0010	1
0000	1
0111	1
0100	1

⑥ No! The benefit of multi-level page tables is that the take less space *when processes use only a fraction of their virtual pages*. Process P1 uses all its virtual pages, which means that its page table takes the same amount of space, whether it is a linear or a multi-level page table.

⑦ The CPU switches to a different process. We can tell, because—according to the MMU table—at time tick 8 virtual page number `000` maps to physical frame number 1010, whereas at time tick 2, the same virtual page number mapped to physical frame number 1010. It is not possible that the same virtual page *of one process* maps to two different physical frames.

The fact that the CPU switches to a different process at time tick 8 implies that the OS scheduler runs at time tick 8. However, we cannot be sure that the OS scheduler runs *because of a timer interrupt*. To be sure of this, we need to assume that process P1 does not terminate or block before time tick 8 (so, it does not cause the OS scheduler to run before a timer interrupt occurs). Under this assumption, the time interrupt occurs at least every 9 time ticks.

Question 3 Forking and scheduling [25 points]

Consider an OS scheduler that follows a Multi-Level Feedback Queue (MLFQ) policy with the following properties:

- there are two priority levels, H (higher) and L (lower);
- both levels have a time slice of 2 time ticks;
- when a thread arrives, it is placed in level H;
- if a thread is in level H and exhausts its time slice, it is demoted to level L;
- all threads are boosted to level H every 5 time ticks (so, at time tick 5, 10, 15, etc.).

Consider the C program excerpt provided below. Next to each statement, you see how long it takes to execute the corresponding instructions. If there is no timing information written next to a statement, assume that the corresponding instructions take 0 time.

```
...
int main()
{
   int pid1 = fork();   // 1 time tick.
   if (pid1 == 0) {
      ...               // 1 time tick. Initiates I/O that takes 1000 time ticks.
      return 0;
   }
   int pid2 = fork();   // 1 time tick.
   if (pid2 == 0) {
      ...               // 1 time tick. Initiates I/O that takes 1000 time ticks.
      return 0;
   }
   waitpid(pid1, ...);  // 1 time tick.
   waitpid(pid2, ...);  // 1 time tick.
   return 0;
}
```

Suppose you compile this program and invoke it on your computer.
The CPU starts executing the first instruction at time tick 0.
The two I/O operations do not block one another (they can proceed at the same time).

① **[2 points]** How many processes and how many threads does your OS create as a result of you invoking this program? Draw the parent/child process graph.

② **[15 points]** Show which thread(s) are in level H and L, which thread(s) are blocked, which thread(s) are ready, and which thread is running at each time tick <u>during the first 6 time ticks</u>. Any category (including the "running" category) may be empty.
Use Table 1 on the next page.

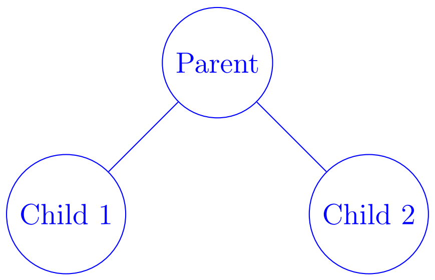
③ **[2 points]** Compute the average turnaround time from time tick 0 until when all threads have terminated.

④ **[6 points]** Would this metric change <u>significantly</u> (and if yes, by how much) if the OS scheduler followed a Shortest Time to Completion First (STCF) policy?
Justify your answer in 1–3 sentences.
Hint: you do <u>not</u> need to draw a table like Table 1 in order to answer this subquestion.

Answers:

① Three processes, each with a single (main) thread: The OS creates one process when the program is invoked; this process forks twice, hence creates two children processes.

Process graph:



Time tick	0	1	2	3	4	5
Arriving threads	0	1	2	-	-	-
Threads in H	0	0,1	1,2	1,2	1,2	1,2,0
Threads in L	-	-	0	0	0	-
Blocked threads	-	-	-	1	1,2	1,2,0
Ready threads	-	1	0,2	0	-	-
Running thread	0	0	1	2	0	-

Table 1: Your answer to sub-question ②.

②

③
- Threads 1 and 2: Each one is in the ready state for 1 time tick, in the running state for 1 time tick, and in the blocked state for 1000 time ticks. Turnadound time = 1002.
- Thread 0: In the ready state for 2 time ticks, in the running state for 4 time ticks, in the blocked state for 998 time ticks (from time tick 5 until the end of time tick 1002, which is when Thread 1 terminates). Turnaround time = 1004 time ticks.
- Average turnaround time = $\frac{1002+1002+1004}{3}$ = 1002.5 time ticks.

④ No,turnaround time could not change significantly, because each of threads 1 and 2 must run for 1000 time ticks anyway, while thread 0 must wait for each of the two other threads to finish. I.e., none of the threads spends any significant amount of time in the ready state, hence none of the threads could benefit significantly from a different scheduling policy.

Question 4 A bit of C [35 points]

This is a set of three independent questions about C programming.
For your answers, you can use whatever C standard up to (and including) C23.

4.1 A bit of memory [13.5 points]

Consider the following C program (which compiles):

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

void f(const char* tab[], size_t n)
{
  char* new = calloc(3,1);
  char* it = new;
  *new++ = **tab;
  *new = tab[n-1][4];
  printf("\"%s\"\n", it);
  free(it);
}

int main(void)
{
  const char* hello = "Hello CS-202!";
  size_t p = 4;
  char c = 'x';
  const char* some[] = { hello + p, strstr(hello, "-2"), &c };

  f(some, 2);

  return 0;
}
```

As a reminder, the function

```
char* strstr(const char* haystack, const char* needle);
```

returns (a pointer to) the position of the substring `needle` inside the string `haystack`.

① **[1 point]** For each of the 8 variables/parameters (`tab`, `n`, `new`, `it`, `hello`, `p`, `c` and `some`), put their name in the appropriate column corresponding to where *they* stay in memory:
(maybe **read the next question first**)

stack	heap	text or data segment
`tab, n`		
`new, it`		
`hello, p`		
`c, some`		

② **[3 points]** For each of the following pointers, say whether they **point to** stack, heap or text/data segment:

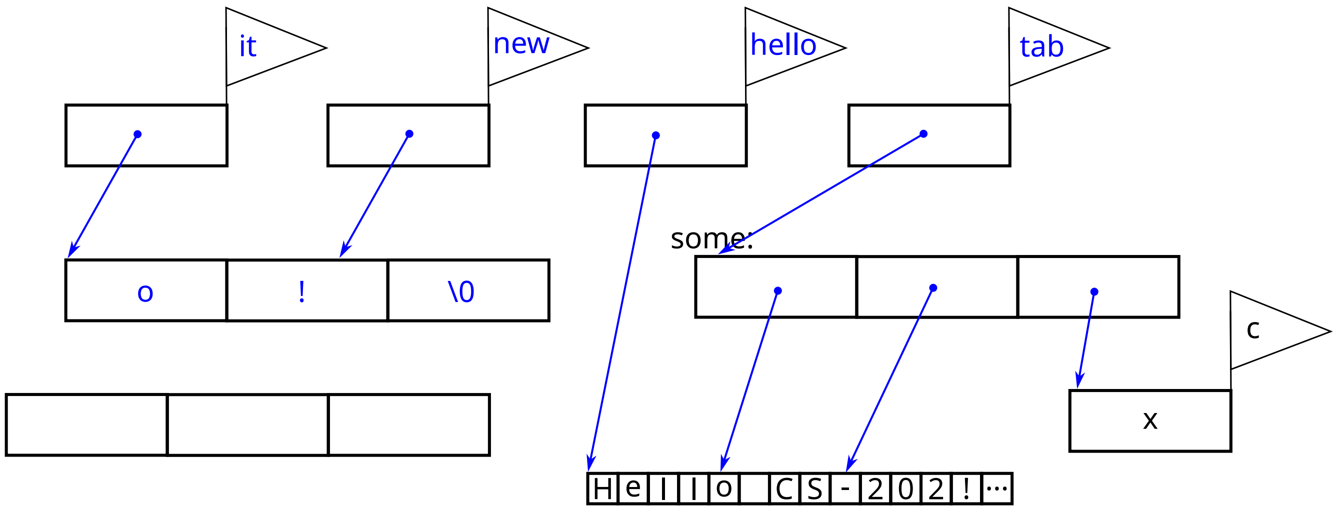
`some[0]`:	`data`	`hello`:	`data`
`some[1]`:	`data`	`new`:	`heap`
`some[2]`:	`stack`	`it`:	`heap`

③ **[4 points]** What does the program print? If there are some unknown values, simply use a question mark (?) for each of them.
Briefly explain your answer.

Answer: It prints `"o!"` (with a newline).

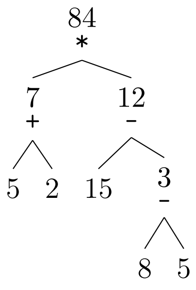
First, `some[0]`, which will become `tab[0]`, is assigned to `"o CS-202!"` (see illustration in next subquestion).
Then in `f()`, we assign `*new` to `**tab`, which is `tab[0][0]`, thus `'o'`.
Then `new` is moved forward and `*new` is assigned to `tab[n-1][4]`, which is `some[2-1][4]`, , which is `strstr("Hello CS-202!", "-2")[4]`, i.e. `"-202!"[4]`, i.e. `'!'`.
The whole `it` string is **null-terminated** because of the initial `calloc()`.

④ **[5.5 points]** Using the template below, draw an abstract picture of the memory state for the six variables (and their relations) `it`, `new`, `tab`, `some`, `c` and `hello` when the `printf()` is done.
There may be more cells than needed; and if you miss cells, simply add your own.



4.2 A bit of arithmetic [14.5 points]

Arithmetic computations can be represented in the form of a tree, each node of which contains an operator, a value, and a left and a right computation (= subtrees).
For instance, the computation $(5 + 2) \times (15 - (8 - 5))$ could be represented as the tree on the right, where for inner nodes, the corresponding value has been displayed above the operator.
For pure values, i.e. leaves of the tree, the "operator" is simply a special operator named "value" and their left and right computations are simply empty.



4.2.1 Types [2.5 points]

Here first define an enumerated type named `Operation` with the values: `ADD`, `SUB`, `MULT`, `DIV` and `VALUE`.

Then define a type `Tree` which contains: an `Operation`, a value (`unsigned int`) and two "`Trees`", `left` and `right`. It's up to you to choose the appropriate type for `left` and `right` (the reason why we quoted it).

Answer:

```
enum Operation { ADD, SUB, MULT, DIV, VALUE };
// could also be/add a typedef

struct Tree {
  enum Operation op;
  unsigned int value;
  struct Tree* left;
  struct Tree* right;
};
// could also be/add a typedef
```

4.2.2 Values [4 points]

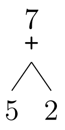
Define a function `value()` that takes a `unsigned int` and returns a pointer to a newly allocated `Tree` which contains only a value, i.e. it's a leaf: operation is `VALUE`, the value is the parameter of the function and `left` and `right` are "empty trees" (it's up to you to choose how to represent that according to your former choices).
This function returns `NULL` in case of error.

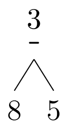
Answer:

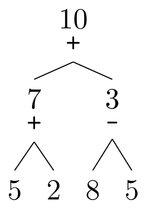
```
struct Tree* value(unsigned int v)
{
  struct Tree* result = malloc(sizeof(struct Tree));
  if (result != NULL) {
    result->op = VALUE;
    result->value = v;
    result->left = NULL; // not needed if calloc()
    result->right = NULL;
  }
  return result;
}
```

4.2.3 Addition [5 points]

Define a function `add()` that takes two pointers to `Tree` and returns a pointer to a newly allocated `Tree` which represents the addition of the two `Tree` received as parameters.
It returns `NULL` in case of error.

For instance, if this function receives the trees 

and 

it will then return the tree 

Answer:

```
struct Tree* add(struct Tree* left, struct Tree* right)
{
  if (left  == NULL) return NULL;
  if (right == NULL) return NULL;

  struct Tree* result = malloc(sizeof(struct Tree));
  if (result != NULL) {
    result->op = ADD;
    result->value = left->value + right->value;
    result->left = left;
    result->right = right;
  }
  return result;
}
```

4.2.4 Garbage collector [3 points]

Define a function `free_tree()` that take a pointer to `Tree` and releases *all* the memory that has been allocated for that tree (assuming it was itself dynamically allocated).
This function does not return anything.

Answer:

```
void free_tree(struct Tree* t)
{
  if (t != NULL) {
    free_tree(t->left);
    free_tree(t->right);
    free(t);
  }
}
```

4.3 A bit of strings [7 points]

Using pointer arithmetic, write a *function*

`char* normalize(const char* string);`

that takes a(n immutable) string and returns a copy of it, where all whitespaces are replaced with `'_'` (underscore) and all characters are put in lowercase.
This function returns `NULL` in case of error.

To test if a character is a whitespace, you must use the standard function

`int isspace(int c);`

and to get the lowercase of a character, you must use the standard function

`int tolower(int c);`

This function is robust (returns the same character if it has no lowercase equivalent).

Note: in C, a `char` can silently be cast to and cast from an `int`.

Answer:

```c
char* normalize(const char* string)
{
  if (string == NULL) return NULL;

  char* returned = strdup(string); // C23 version
  /* alternative:
  char* returned = calloc(strlen(string) + 1, 1);
  * Then either strcpy(string) here,
  *      or add chars from string in the loop below.
  */
  if (returned != NULL) {

    const char* const end = returned + strlen(string);
    // (better) alternative: check for *p not '\0'

    for (char* p = returned; p < end; ++p) {
      if (isspace(*p)) {
        *p = '_';
      } else {
        *p = tolower(*p);
      }
    }
  }

  return returned;
}
```