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., Bugnion E. & Chappelier J.-C.

CS–202 COMPUTER SYSTEMS

Midterm solution

March 20th, 2024

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 any additional sheets; only this document will be graded.

   Two additional blank pages are provided at the end of the handout. If you use them, please clearly state the question number you are answering.
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 five independent exercises, which can be addressed in any order, but which do not score the same (points are indicated, the total is 90 points); all exercises count for the final grade.

LEAVE THIS EMPTY					
Question 1	Question 2	Question 3	Question 4	Question 5	**TOTAL**
11	15	12	12	40	90

Question 1 Parents and children [11 points]

Assume that the current directory has a single file named `myfork.c` containing the following program. The program is compiled using the following command: `gcc myfork.c -o myfork`

The program compiles without any error. **Background:** The library call `execvp(cmd, arglist)` calls the `exec` system call to execute `cmd` with the argument list `arglist`. As is with the convention, `myargs[0]` below corresponds to `cmd`. `arglist` is passed as `argv` to the `main()` function of the executed program.

Analyze the following program and answer the following questions:

```
1   #include<stdio.h>
2   #include<unistd.h>
3   #include<sys/wait.h>
4   #include<string.h>
5
6   int main(int argc, char *argv[])
7   {
8     int pid = fork();
9     if (pid == 0) {
10      if (strcmp(argv[1], "0") == 0 ) {
11        char *myargs[3];
12        myargs[0] = strdup("./myfork");
13        myargs[1] = strdup("1");
14        myargs[2] = NULL;
15        execvp(myargs[0], myargs);
16      } else {
17        char *myargs[2];
18        myargs[0] = strdup("ls");
19        myargs[1] = NULL;
20        execvp(myargs[0], myargs);
21      }
22      printf("Child started\n");
23    } else {
24      wait(NULL);
25      printf("Child finished\n");
26    }
27    return 0;
28  }
```

On the terminal, you launch the following command: `./myfork 0`

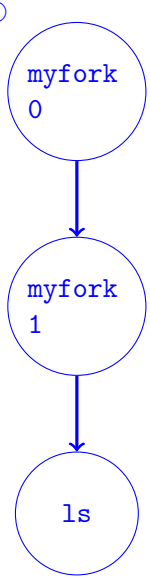
① **[6 points]** Draw a process diagram that includes parent-child relationship and the program name and argument in each node (after `exec`, when applicable)

② **[1 point]** How many processes will be created during the execution (including the original instance of `myfork`)?

③ **[4 points]** What is the output of the above command? Is the output deterministic?

Answers:

①



② 3

③

```
myfork myfork.c
Child finished
Child finished
```

Note: any order/layout of myfork and myfork.c is acceptable
Note: alternative: mentioning <content of directory> rather than `myfork myfork.c`

Question 2 Fibonacci [15 points]

Assume a 64-bit architecture with `sizeof(long long) = 8`. Assume a calling convention where all arguments are pushed on the stack, and no compiler optimizations such as use of registers to store temporary variables. Everything that logically belongs on the stack is on the stack.

Considered the following code, compiled to `./fibo`:

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

long long steps;

long long fibonacci(long long n)
{
  steps += 1;
  if (n < 2) return 1;
  long long x = fibonacci(n-1);
  long long y = fibonacci(n-2);
  return x + y;
}

int main(int argc, char *argv[])
{
  steps = 0;
  long long n = atoi(argv[1]);
  if (n > 0) {
    long long f = fibonacci(n);
    printf("Fibonacci of n=%lld is %lld"
           "and requires %lld invocations\n",
           n, f, steps);
  }
  return 0;
}
```

① **[1.5 points]** What is the output of `./fibo 3`?

② **[6 points]** Assume a constrained machine so that the stack can only be 8 KiB (8×1024 bytes) in size. For which approximate value of `n` is there a likely stack overflow (stack pointer exceeds stack size)? **State your assumptions clearly.**

③ **[5 points]** Currently the program only uses the stack. Modify the above program to additionally use the data segment (without changing the algorithmic complexity of the program).
Put your modifications directly on the above provided code.

④ **[2.5 points]** For your modified program, does your answer to the subquestion ② above change? **State your assumptions clearly.**

Answers:

① `Fibonacci of n=3 is 3 and requires 5 invocations`

② Each call as two arguments, the RIP and two local variable, thus five times 8 bytes.

Stack will thus be exhausted after roughly $8 \times 1024/(5 \times 8) \simeq 204$ calls.

Notice that the calls are depth-first, thus the first branch (`fibonacci(n-1, steps)`) exhausts the stack before anything else.

③ see above

④ Not a big change: `step` is no longer on the stack, thus the above 5 turns into a 4: $1024/4 \simeq 256$ calls.

Question 3 A few pages [12 points]

Consider a system with 32-bit virtual and physical addresses, with pages of 4 KiB size and the MMU uses a linear page table. Recall that in hexadecimal notation, `0x1000` equals 4 KiB.

A process in this system require 4 virtual pages, out of which only 3 virtual pages are mapped to physical pages in the following manner:

- virtual page 1, containing the text section of the process, maps to physical page 6;
- virtual page 2, containing the data section of the process, maps to physical page 3;
- virtual page 3, containing the stack section of the process, maps to physical page 11;
- virtual page 5, containing the heap, is not currently mapped as the content has been swapped out on disk;
- all the other pages in the virtual address space of the process are invalid.

The MMU is given a pointer to the base address of this page table for address translation. Further, the MMU has a small TLB that stores two translations. At times $t = T$, the TLB holds translations for virtual pages 1 and 3.

Assume that each virtual address given below is accessed independently and the state of the system (page table, MMU, TLB) is reset to the state given above.
For each virtual address given below, describe what happens when that address is accessed by the CPU. Specifically, you must answer:

- Will it result in a TLB hit or miss?
- Will it result in a trap or not?
  - If there is no trap, which physical address is accessed?
  - If there is a trap, what does the operating system do?

① at $t = T$, Virtual address `0x00001011`;

② or at $t = T$, Virtual address `0x00002F1E`;

③ or at $t = T$, Virtual address `0x000052AB`;

④ or at $t = T$, Virtual address `0x00004C79`.

Justify your answers.

Answers:

① Virtual page 1 (or text)

- TLB hit
- No trap, physical address: `0x00006011`

② Virtual page 2 (or data)

- TLB miss
- No trap, physical address: `0x00003F1E`

③ Virtual page 5 (or heap)

- TLB miss
- Trap. OS will *swap in* the page from disk. (swap in = bring page from disk).

④ Virtual page 4 (Invalid)

- TLB miss
- Trap. OS will *kill the process* (process doesn't run etc.) using `SegFault`.

Question 4 Disk access [12 points]

Consider the following program:

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include <unistd.h>
5   #include <fcntl.h>
6
7   int main(void)
8   {
9     int fd = open("/home/cs202/out.txt", O_CREAT | O_RDWR | O_TRUNC, S_IRWXU);
10    if (fd > 0) {
11      char buffer[20] = { 0 };
12      write(fd, "hello world", 11);
13      lseek(fd, 2, SEEK_SET);
14      read(fd, buffer, 5);
15      printf("read data: %s\n", buffer);
16      close(fd);
17    }
18    return 0;
19  }
```

The file system is mounted at the root directory `"/"` and the user has full access to it. Assume the following about the file system and its content:

- the inode structure has only the following:
  - a length field indicating the length of the file;
  - a modification time field;
  - 10 entries for direct blocks;
  - entries for indirect blocks (unused in the question);
- a single directory entry resides in a single data block;
- the root directory `"/"` consists of a single directory called `"home"`;
- the `"home"` directory consists of a single directory called `"cs202"`;
- the file `"out.txt"` does not exist in the `"cs202"` directory;
- the OS has a very large file system buffer cache, initially empty, which is used to cache inode and data blocks.

Based on the above program and assumptions about the file system, answer the following questions:

① **[2 points]** What is the output of this program? **State your assumptions.**

② **[10 points]** In the tables on the next page, enter the number of blocks associated with each inode (inode blocks, indirect blocks, data blocks) which are read or written from disk for each syscall.

Mark read/write accesses to inode in table 1 and read/write accesses to data blocks in table 2. Creating a new inode/data block or updating an existing block is counted as a write.

In these tables, all blank answers will be interpreted as "not answered" rather than 0.
State your assumptions, if any.

Answers:

Table 1: inode blocks

	"/"		"home"		"cs202"		"out.txt"	
	Read	Write	Read	Write	Read	Write	Read	Write
`open`	1	0	1	0	1	1	0	1
`write`	0	0	0	0	0	0	0	1
`lseek`	0	0	0	0	0	0	0	0
`read`	0	0	0	0	0	0	0	0
`close`	0	0	0	0	0	0	0	0

Table 2: data blocks

	"/"		"home"		"cs202"		"out.txt"	
	Read	Write	Read	Write	Read	Write	Read	Write
`open`	1	0	1	0	1	1	0	0
`write`	0	0	0	0	0	0	0	1
`lseek`	0	0	0	0	0	0	0	0
`read`	0	0	0	0	0	0	0	0
`close`	0	0	0	0	0	0	0	0

① `read data:  llo w`

Explain: reads 5 bytes (thus 5 `char`) from position index 2 (thus third byte/`char`).

② see above.

Explain:

- nothing is written but the file `out.txt` and its parent directory;
- the `open()` syscall creates a new file which requires a write operation on both inode (to increase the len) and data-block of the parent directory (to add the mapping of "out.txt" to its inode);
- during `open()`, the whole path is read;
- the `write()` syscall writes to the disk: to the datablock for the content and to the inode to set the length of the file;
- `lseek()` and `close()` do not perform any disk access;
- the subsequent *read* operation does not access the disk itself because of the cache;

Note: this answers assumes that all operations are synchronous; a filesystem with asynchronous write-back could delay some disk write operations at a later point (this was not covered in class, but could be an acceptable answer)

Question 5 Small value set [40 points]

In this exercise, we are interested in a simulation program for a (small) value set. We start by providing an quick overview of the desired program, although, for the sake of simplicity, we do **not** ask you to write the whole program, only some specific parts of it.

5.1 General description [nothing to do]

The simulated value set can simply be seen as a "black box" into which unique values (`int` here for simplicity) can be added (function "`set()`") or retreived (function "`get()`"). But the set capacity is *limited*, thus some overwriting occurs. The main question is then, which value should be overwriten when the set is full and a new value shall be added. This "value overwritting" (by the `set()` function when there is no more space) depends on the chosen management policy: it can be the last value entered (`set()`), the last value read (`get()`), the first value entered, etc.

For the sake of simplicity, we here consider only two possible management policies:

- "*Most Recently Get*" (MRG): the overwritten value is the last value *read* (`get()`, independently of any `set()`); see example below;
- "*Last Recently Get*" (LRG) the overwritten value is the value *read* the least recently (longest ago, independently of any `set()`).

For example, let's assume that we can only enter *four* values in the set and we enter the values `1`, `2`, `3`, `4`; then read out (`get()`) the values `4` then `1`; and that we finally insert the value `5`. Where does this `5` go?

In other words, after the instruction sequence:
`set(1) set(2) set(3) set(4)  get(4)  get(1)  set(5)`
what are the values still in the set?

In the case of MRG management, `5` overwrites `1` ; and the set contains therefore the values `2`, `3`, `4`, `5` (we don't bother the order here).

In the case of LRG management, `5` overwrites `4`; and the set contains therefore the values `1`, `2`, `3`, `5` (order doesn't matter).

5.2 Set data structure [23 points]

The implemented set data structure will be organized by "lines" containing data (`int` here for simplicity). For example:

```
line 0: 4 8 5 0
line 1: 0 0 0 0
line 2: 12 1 7 3
```

Although all lines have the same size, `MAX_PER_LINE`, your program will *dynamically* allocate them. The array of lines is also dynamic: it's empty at the beginning and dynamically fills up necessary lines, all initialized to `0`, without leaving any "hole" in the array of lines.

For example, if, from an empty array of lines, we require line number 2 to store the value `12`, the array of lines will then change from empty to:

```
line 0: 0 0 0 0
line 1: 0 0 0 0
line 2: 12 0 0 0
```

i.e. the lines `0` and `1` will also be added (to avoid any "hole").

In addition, the set data structure will store an array (say "`used`") containing the number of values present in each line. In the above example, this array `used` would contain `0 0 1` because we haven't yet put any value in the first two lines and that we've put one value (`12`) in the third line.

5.2.1 Data structure [6 points]

Propose here a data structure to represent a set of values as explained above:

Answer:

```
typedef int data_t;
typedef struct {
  data_t** lines;   // array of arrays of data
  size_t*  used;    // array of size_t
  size_t   nlines;  // size of the arrays
} set_t;
```

The `typedef` for `data_t` is a good practice, optional here.

The `typedef` of `set_t` is itself optional for those who like to write `struct` everywhere.

We could do even better by grouping a line and its `used` in the same substructure to ensure that they have the same number:

```
typedef int data_t;

typedef struct {
  data_t* row_elements;
  size_t  nb_used;
} row_t;

typedef struct {
  row_t* lines;    // array of lines
  size_t nlines;
} set_t;
```

As a result, with such a substructure, we could implement the flexible-array-member trick:

```
typedef int data_t;

typedef struct {
  data_t* row_elements;
  size_t  nb_used;
} row_t;

typedef struct {
  size_t nlines;
  row_t  lines[1];
} set_t;
```

which is not possible with the first s since we have *two* arrays.

5.2.2 Add a line [11 points]

Now define the function `new_line_at()` which creates and initializes a new line, respecting all the above conditions. For example, starting from an empty `set` a call to

```
new_line_at(&set, 2);
```

will create the following value set (assuming `MAX_PER_LINE` is equal to 4):

```
line 0: 0 0 0 0
line 1: 0 0 0 0
line 2: 0 0 0 0
```

Answer:

```
void new_line_at(set_t *set, size_t index) {

  if (set == NULL) return; // optionnel

  // Bonus: do overflow check for multiplications

  data_t** temp1 = realloc(set->lines, (index+1) * sizeof(data_t*));
  if (temp1 == NULL) return;

  size_t* temp2 = realloc(set->used, (index+1) * sizeof(size_t));
  if (temp2 == NULL) return;

  set->lines = temp1;
  set->used  = temp2;

  for (size_t i = set->nlines; i <= index; ++i) {
    set->lines[i] = calloc(MAX_PER_LINE, sizeof(data_t));
    if (set->lines[i] == NULL); // could do some error handling
    set->used[i] = 0;
  }
  set->nlines = index + 1;
}
```

Notes:

- Protection against a bad index (such as "`if (index < set->nlines)`") may well have been made on call and is not strictly necessary here, but it's good to think to think about it. Notice, BTW, that the above code is perfectly safe if `index` is smaller than `set->nlines`.
- It's also worth thinking about the protection against the risk of overflow in the multiplication in the arguments of `realloc()` (but it's also conceivable, especially in an examination context, different from producing a complete stand-alone library, that the context of use of this function is protected against this (`index` not too big)).
- We could have `int` as a return type (e.g. error code), or even `int*` (the new line).

5.2.3 Free memory [6 points]

Define here the function `free_set()` which frees all the memory allocated by a set (but not the set itself).

Answer:

```
void free_set(set_t *set) {

  if (set == NULL) return; // Nothing to do

  for (size_t i = 0; i < set->nlines; ++i) {
    free(set->lines[i]);
  }

  free(set->lines);
  free(set->used);
  memset(set, 0, sizeof(*set));
}
```

Note: `free()` is protected against `NULL` (it's in the standard).

5.3 Management policies [9 points]

The different management policies ("MRG", "LRG"; see introduction) can each be described by two specific actions: one to be performed during a `get()` and the other to be performed during a `set()`. The chosen implementation is to use function pointers to represent each of the possible management policies. This is what this subquestion is about.

5.3.1 Policy types [2 points]

Define an `enum` type, named `Policy`, to indicate the possible management policies (`MRG` and `LRG` in our case):

```
enum Policy { MRG = 0, LRG, LAST };
```

`LAST` and `"= 0"` are not needed here at all: it's just a useful trick for browsing or marking the end of a `enum`.

We can also use `typedef` here.

5.3.2 Actions [3 points]

The actions to be made in `get()` and in `set()` by the various management policies all have the same format: they receive as arguments a value set, which can be modified, a line number and a position in the line, and don't return anything.

Define here the type `policy_function` which is a pointer to such a function:

```
typedef void (*policy_function)(set_t *set, size_t line, size_t index);
```

5.3.3 Description of the different management policies [4 points]

We can therefore summarize the various management policies (`MRG` or `LRG` for us) by a two-dimensional array of `policy_functions`. Assuming that respectively

the action to be performed in ...	with management ...	is implemented in a function[1] ...
`get()`	MRG	`move_to_front()`,
`set()`	MRG	`front_to_back()`,
`get()`	LRG	`move_to_back()`,
`set()`	LRG	`do_nothing()`,

define here (and initialize/fill) a (two-dimensional) array named `policies` representing the above information (**hint**: maybe read also the next question):

```
static const policy_function policies[LAST][2] = {
    { move_to_front, front_to_back }, // MRG
    { move_to_back , do_nothing    }  // LRG
};
```

`static` and `const` (as well as the use of `LAST`) are optional here.

The array could be transposed.

5.4 Function `get()` [8 points]

Assume the existence of:

- the "management policies" table, `policies`, as defined in 5.3.3;
- a function[2] `hash()` giving the line number in the set for a given value (for example `hash(12)` will give `2` in the example subquestion 5.2, page 10),
- a function[2] `search()` which receives (at least) a value and an array, and returns:
  - the index of the value in the array, if that value is in the array;
  - or `"(size_t) -1"` if it isn't.

Define the `get()` function for the set.
This function receives as arguments a set, a management policy and a value, and returns `1` if the value you're looking for is present in the set and `0` otherwise.

Answer:

```
int get(const set_t* set, const enum Policy pol, const data_t val) {

    if (set == NULL) return 0; // optionnel

    const size_t ln = hash(val);
    if (ln >= set->nlines) return 0;
    const size_t index = search(set->lines[ln], val);

    if (index < set->used[ln]) { // notice this includes proper handling of -1
        policies[pol][GET](set, ln, index);
        return 1;
    }

    return 0;
}
```

Notes:

- even if it is a very good practice, in the context of an exam, the `assert` or any other equivalent check is **optional** (it could very well be assumed that this is an internal function and that all checks have been made beforehand)
- the first three (real) lines could still be better modularized, e.g. by a `set_search(set, val)`, which makes `get()` even more concise (call in the last `if`).

[2] We will not bother with its definition; you do not have to write this function.