ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E SISTEMAS DIGITAIS

Relatório da P1 de Sistemas de Programação

Turma 50

Lucas Haug

No. USP

10773565

São Paulo
2021

Relatório da P1 de Sistemas de Programação

Relatório apresentado como requisito para avaliação na disciplina PCS3216 - Sistemas de Programação, no curso de Engenharia Elétrica oferecido pela Escola Politécnica da Universidade de São Paulo.

São Paulo

2021

SUMÁRIO

Desenvolvimento do Projeto

Problema

Foi solicitado o desenvolvimento de um programa para fazer o loading de conteúdos de um arquivo para a memória e de um programa para fazer o dumping da memória para um arquivo.

Estrutura geral

Para o desenvolvimento do loader e do dumper foi utilizada programação orientada a eventos para construir motores de eventos que lidassem com as necessidades dos dois programas. Para cada um dos dois programas foram separados para cada nível de complexidade motores de eventos distintos, os quais se comunicavam entre si a partir da adição de eventos nas filas dos outros.

Além disso, foi modelada uma memória a partir de um vetor de números inteiros sem sinal de 8 bits. Essa memória, indexada de 0x000 até 0xFFF pode ser acessada diretamente tanto pelo loader quanto pelo dumper.

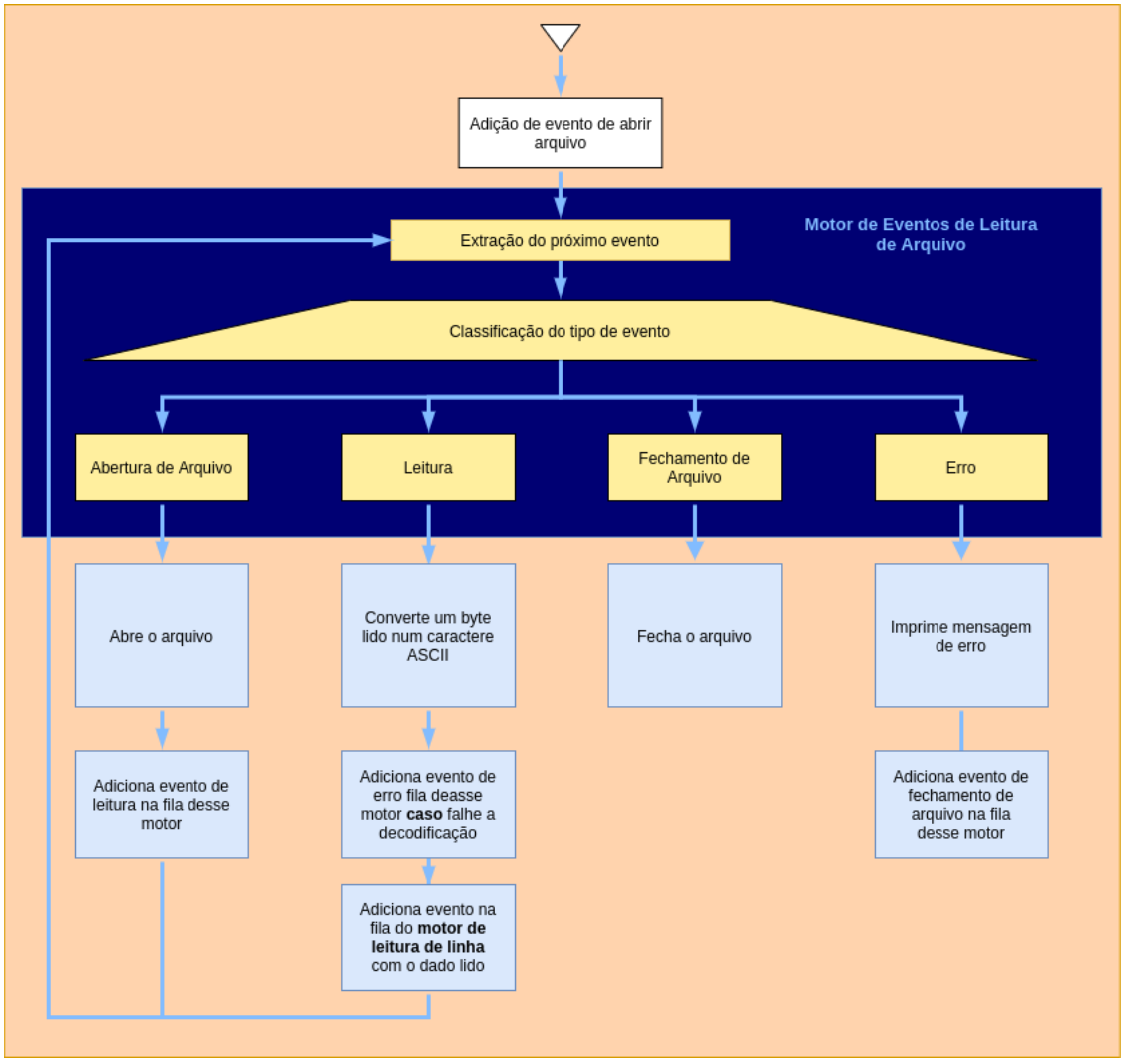
Por fim, foi desenvolvida uma interface de usuário simples para a escolha de qual programa rodar, para poder visualizar os dados da memória e também zerá-la.

Modelamento

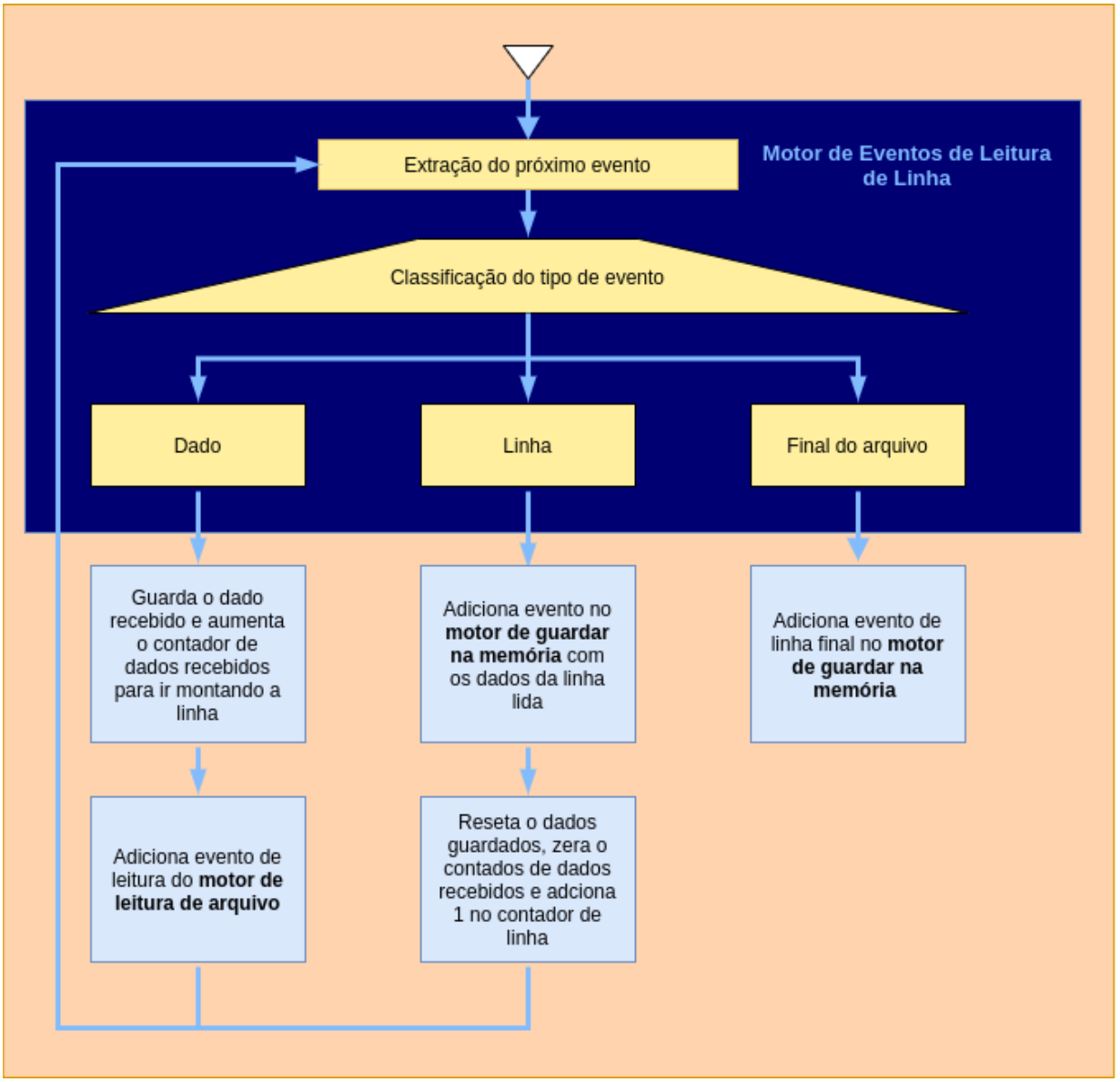
Loader

Para o desenvolvimento do loader foram utilizados três motores de evento para executar todas as tarefas. Foi feito um motor para a leitura do arquivo byte a byte e codificá-los para caracteres a partir do código ASCII, um motor para receber cada um dos caracteres e juntá-los em linhas e por fim um motor para pegar cada uma das linhas lidas e a partir das informações delas escrever os dados na memória, codificando os caracteres recebidos a partir do código ASCII. O modelamento dos motores pode ser visto abaixo, junto com as suas tabelas de eventos e reações.

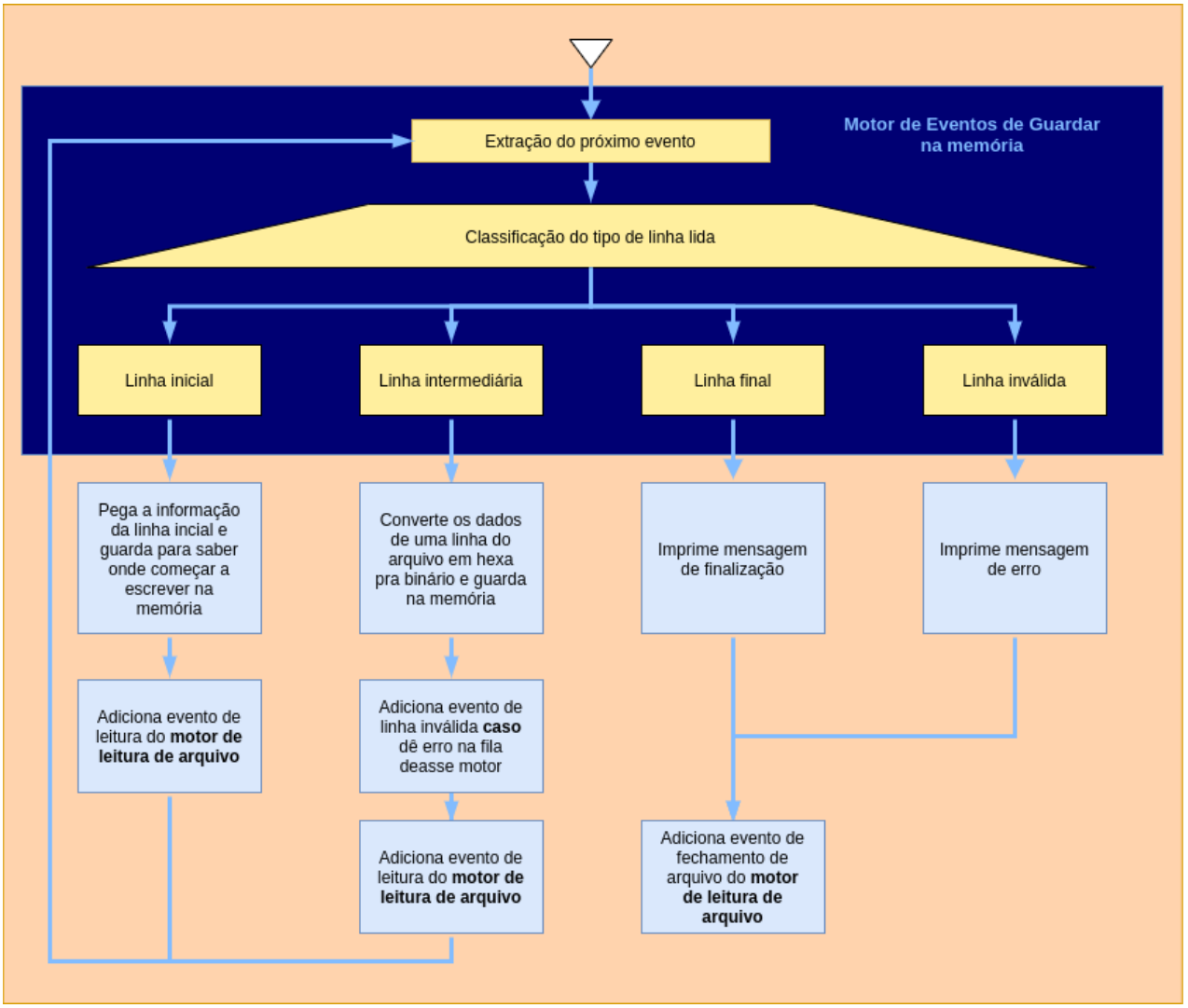
Motor de Eventos de Leitura de Arquivo



Motor de Eventos de Leitura de Linha



Motor de Eventos de Guardar na memória

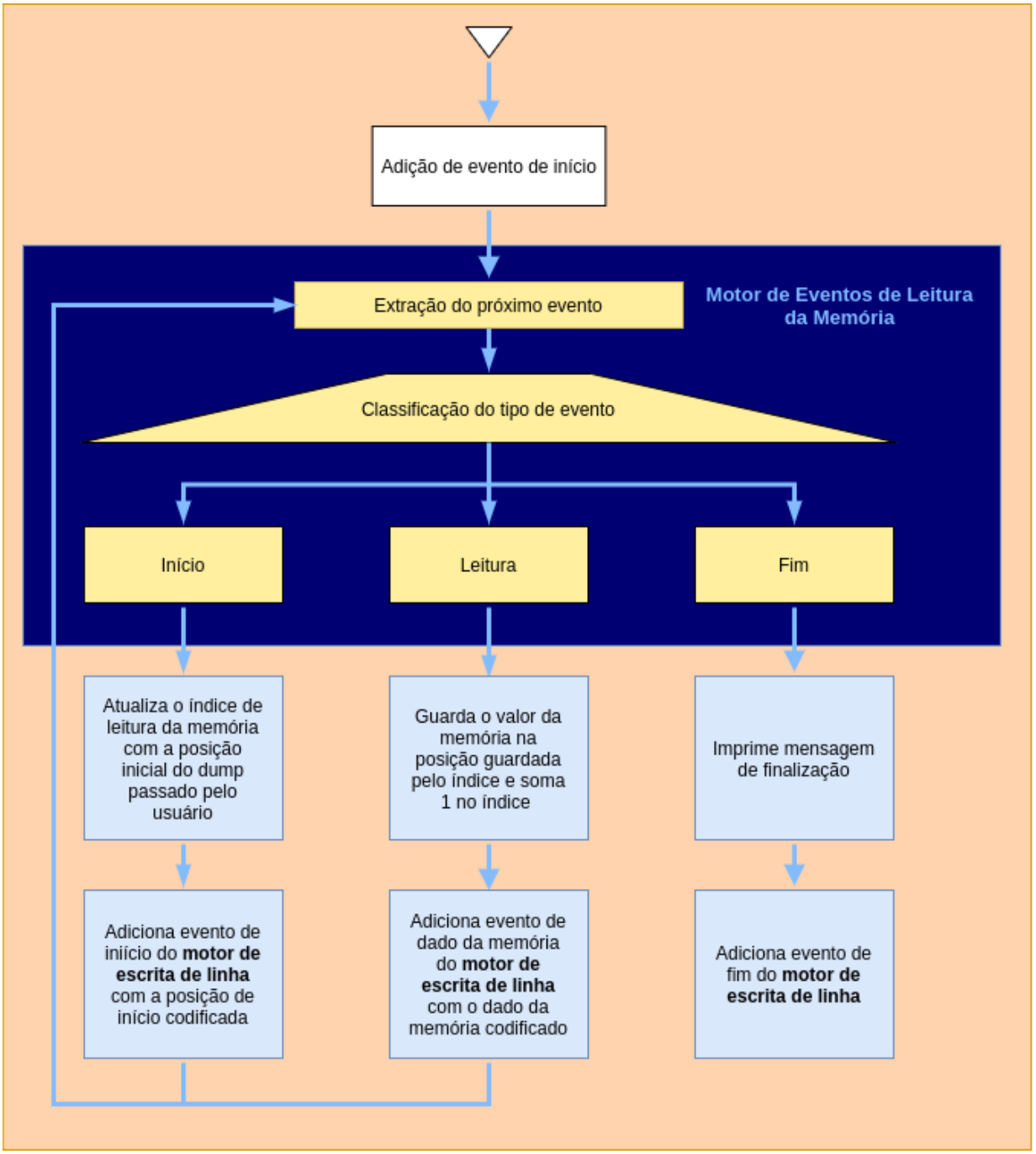


Dumper

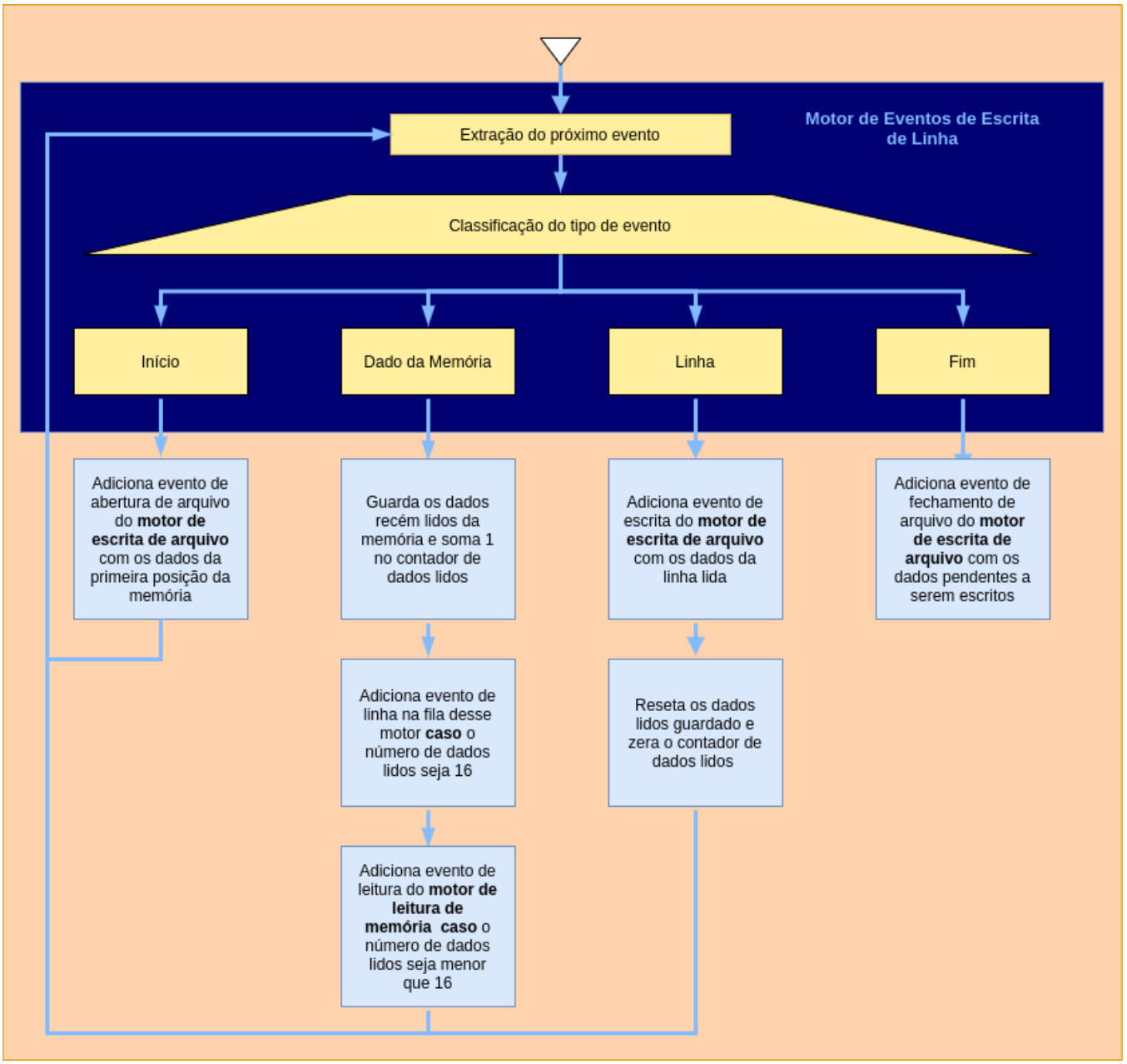
Para o desenvolvimento do dumper foram utilizados também três motores de evento para executar todas as tarefas. Foi feito um motor para a leitura da memória nas posições passadas pelo usuário e a codificação dos dados para caracteres a partir do código ASCII, um motor para receber todos os dados codificados e juntá-los em linhas e um motor para escrita do arquivo byte a byte, após codificar os caracteres de uma linha a partir do código ASCII.

O modelamento dos motores pode ser visto abaixo, junto com as suas tabelas de eventos e reações.

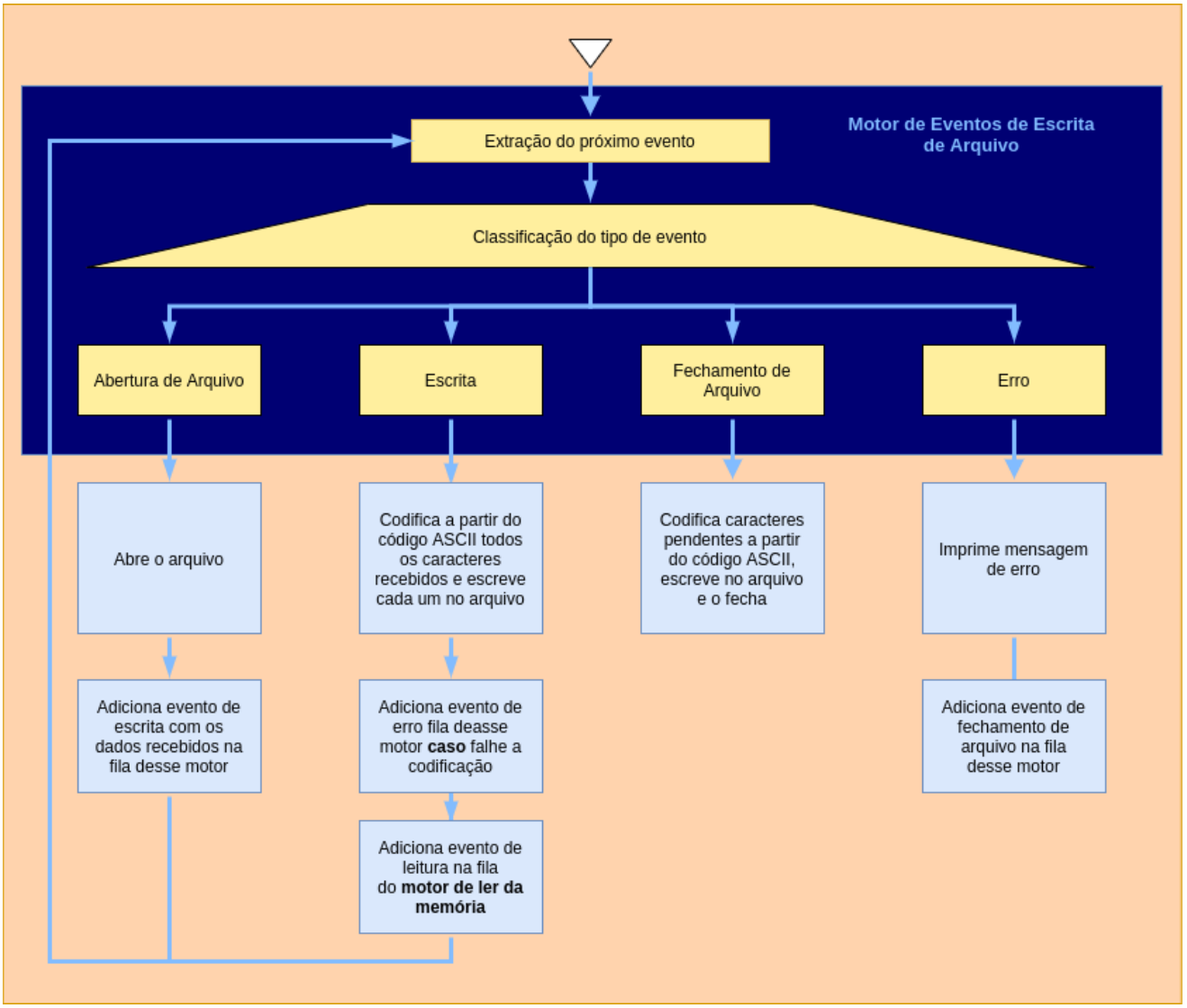
Motor de Eventos de Leitura da Memória



Motor de Eventos de Escrita de Linha



Motor de Eventos de Escrita de Arquivo



Software

Geral

Para o desenvolvimento dos programas foi utilizado python e foi utilizado orientação a objetos para modelar os objetos dos programas.

Para o modelamento da memória foi desenvolvida a classe abaixo.

```python
import numpy as np

class Memory():
    def __init__(self, size) -> None:
        self.array = np.zeros(size, dtype=np.uint8)


    def __getitem__(self, index):
        return self.array[index]


    def __setitem__(self, index, value):
        self.array[index] = value


    def clear(self):
        self.array *= 0


    def display(self):
        np.savetxt("image.txt", self.array, fmt='%02X')
```

Para o motor de eventos foi feita uma classe mãe de motor de eventos, a qual pode ser vista abaixo. Da mesma forma foi feita uma classe mãe de eventos, a qual pode ser vista em seguida.

```python
from abc import ABC, abstractmethod
from typing import Tuple

class EventsMotor(ABC):
    def __init__(self) -> None:
        self.events_queue = []

        self.reactions_table = {
            "unknown": None
        }
```

```
        self.active = True


    def activate(self):
        self.active = True

        self.events_queue = []


    def deactivate(self):
        self.active = False


    def is_active(self):
        return self.active


    def add_event(self, event: Event) -> None:
        self.events_queue.append(event)


    def extract_event(self) -> Tuple[bool, Event]:
        empty_queue = not self.events_queue

        if empty_queue:
            event = None
        else:
            event = self.events_queue.pop(0)

        return (not empty_queue, event)


    @abstractmethod
    def categorize_event(self, event: Event) -> str:
        pass
```

```
    def react_to_event(self, event_type: str, event: Event) -> None:
        self.reactions_table[event_type](event)


    def run(self) -> bool:
        has_pending_event, event = self.extract_event()

        if has_pending_event:
            event_type = self.categorize_event(event)

            self.react_to_event(event_type, event)

            return True
        else:
            return False
```

Classe mãe de eventos:

```
from abc import ABC

class Event(ABC):
    def __init__(self) -> None:
        pass
```

A partir dessa classe foram feitas classes para cada um dos motores de eventos descritos no modelamento, as quais podem ser vistas abaixo.

Eventos e Motores do Loader

Eventos do loader:

```
class FileReadingEvent(Event):
    """
    Event type handled by the FileReadingMotor
    """

    def __init__(self, event_type) -> None:
        super().__init__()

        self.type = event_type


class DataReadingEvent(Event):
    """
    Event type handled by the LineReadingMotor
    """

    def __init__(self, event_data) -> None:
        super().__init__()

        self.data = event_data


class LineReadingEvent(Event):
    """
    Event type handled by the MemStoringMotor
    """

    def __init__(self, first_line, line_data, line_size) -> None:
        super().__init__()

        self.first_line = first_line
        self.line_data = line_data
```

```
        self.line_size = line_size
```

Motores do loader:

```
class FileReadingMotor(EventsMotor):
    def __init__(self) -> None:
        super().__init__()

        self.reactions_table["open_file"] = self._open_file
        self.reactions_table["read"] = self._read
        self.reactions_table["close_file"] = self._close_file
        self.reactions_table["error"] = self._error

        self.decoding_table = {
            b'': "",
            b'\x0A': "\n",
            b'\x20': " ",
            b'\x30': "0",
            b'\x31': "1",
            b'\x32': "2",
            b'\x33': "3",
            b'\x34': "4",
            b'\x35': "5",
            b'\x36': "6",
            b'\x37': "7",
            b'\x38': "8",
            b'\x39': "9",
            b'\x41': "A",
            b'\x42': "B",
            b'\x43': "C",
            b'\x44': "D",
            b'\x45': "E",
            b'\x46': "F"
        }
```

```
def set_file_name(self, file_name):
    self.file_name = file_name


def set_line_reading_motor(self, line_reading_motor) -> None:
    self.line_reading_motor = line_reading_motor


def categorize_event(self, event: FileReadingEvent) -> str:
    if event.type == "open_file":
        return "open_file"
    elif event.type == "read":
        return "read"
    elif event.type == "close_file":
        return "close_file"
    else:
        return "error"


def _open_file(self, event: FileReadingEvent) -> None:
    self.file = open(self.file_name, 'rb')

    next_event = FileReadingEvent("read")

    self.add_event(next_event)


def _read(self, event: FileReadingEvent) -> None:
    data = self.file.read(1)

    decoded_data = self._decode(data)

    if decoded_data is not None:
        next_event = DataReadingEvent(decoded_data)
```

```
            self.line_reading_motor.add_event(next_event)
        else:
            next_event = FileReadingEvent("error")

            self.add_event(next_event)


def _close_file(self, event: FileReadingEvent) -> None:
    self.file.close()

    self.deactivate()


def _error(self, event: FileReadingEvent) -> None:
    print("[ERROR] Incorrect file content")

    next_event = FileReadingEvent("close_file")

    self.add_event(next_event)


def _decode(self, read_byte):
    if read_byte in self.decoding_table:
        return self.decoding_table[read_byte]
    else:
        return None
```

```
class LineReadingMotor(EventsMotor):
    def __init__(self) -> None:
        super().__init__()

        self.reactions_table["data"] = self._save_data
        self.reactions_table["line"] = self._read_line
        self.reactions_table["file_end"] = self._file_end

        self.read_data = []

        self.line_count = 0
        self.data_count = 0


    def set_file_reading_motor(self, file_reading_motor) -> None:
        self.file_reading_motor = file_reading_motor


    def set_mem_storing_motor(self, mem_storing_motor) -> None:
        self.mem_storing_motor = mem_storing_motor


    def activate(self) -> None:
        super().activate()

        self.read_data = []
        self.line_count = 0
        self.data_count = 0


    def categorize_event(self, event: DataReadingEvent) -> str:
        if event.data == "\n":
            return "line"
        elif event.data == "":
            return "file_end"
        else:
```

```
            return "data"


    def _save_data(self, event : DataReadingEvent) -> None:
        self.read_data.append(event.data)
        self.data_count += 1

        next_event = FileReadingEvent("read")

        self.file_reading_motor.add_event(next_event)


    def _read_line(self, event : DataReadingEvent) -> None:
        first_line = True if self.line_count == 0 else False

        next_event = LineReadingEvent(first_line, self.read_data,
self.data_count)

        self.line_count += 1
        self.data_count = 0
        self.read_data = []

        self.mem_storing_motor.add_event(next_event)

    def _file_end(self, event : DataReadingEvent) -> None:
        # Add end line event
        next_event = LineReadingEvent(False, [], 0)

        self.mem_storing_motor.add_event(next_event)

        self.deactivate()
```

```
ADDRESS_SIZE = 3
DATA_SIZE = 2

NUM_OF_DATA_PER_LINE = 16
NUM_OF_SPACES_SEPARATORS = 15
MAX_BYTES = NUM_OF_DATA_PER_LINE * DATA_SIZE +
NUM_OF_SPACES_SEPARATORS

class MemStoringMotor(EventsMotor):
   def __init__(self) → None:
       super().__init__()

       self.reactions_table["initial_line"] = self._initial_line
       self.reactions_table["middle_line"] = self._middle_line
       self.reactions_table["last_line"] = self._last_line
       self.reactions_table["invalid_line"] = self._invalid_line

       self.memory_pointer = None
       self.memory_position = 0

       self.decoding_table = {
           "0" : 0x0,
           "1" : 0x1,
           "2" : 0x2,
           "3" : 0x3,
           "4" : 0x4,
           "5" : 0x5,
           "6" : 0x6,
           "7" : 0x7,
           "8" : 0x8,
           "9" : 0x9,
           "A" : 0xA,
           "B" : 0xB,
           "C" : 0xC,
           "D" : 0xD,
           "E" : 0xE,
```

```
        "F" : 0xF
    }


def set_file_reading_motor(self, file_reading_motor) -> None:
    self.file_reading_motor = file_reading_motor


def set_memory_pointer(self, memory_pointer) -> None:
    self.memory_pointer = memory_pointer


def activate(self):
    super().activate()

    self.memory_position = 0


def categorize_event(self, event: LineReadingEvent) -> str:
    if event.first_line:
        if event.line_size != ADDRESS_SIZE:
            return "invalid_line"

        return "initial_line"
    elif event.line_size > MAX_BYTES:
        return "invalid_line"
    elif event.line_size == 0:
        return "last_line"
    else:
        return "middle_line"


def _initial_line(self, event: LineReadingEvent):
    data = event.line_data
    decoded_data = self._decode(data, ADDRESS_SIZE)
```

```
        self.memory_position = decoded_data

        next_event = FileReadingEvent("read")

        self.file_reading_motor.add_event(next_event)


    def _middle_line(self, event: LineReadingEvent):
        line_data = event.line_data
        line_size = event.line_size

        # Needed for data padronization
        line_data.append(" ")
        line_size += 1

        data = []

        for i in range(line_size):
            if line_data[i] == " ":
                if len(data) == DATA_SIZE:
                    decoded_data = self._decode(data, DATA_SIZE)

                    self.memory_pointer[self.memory_position] =
decoded_data

                    data = []

                    self.memory_position += 1
                else:
                    # Send ivalid line event
                    next_event = LineReadingEvent(False, [],
MAX_BYTES + 1)

                    self.add_event(next_event)
            else:
                data.append(line_data[i])
```

```
        next_event = FileReadingEvent("read")

        self.file_reading_motor.add_event(next_event)


    def _last_line(self, event: LineReadingEvent):
        print("[INFO] Finished loading")

        next_event = FileReadingEvent("close_file")

        self.file_reading_motor.add_event(next_event)

        self.deactivate()


    def _invalid_line(self, event: LineReadingEvent):
        print("[ERROR] Make sure to follow the input file
specification")

        next_event = FileReadingEvent("close_file")

        self.file_reading_motor.add_event(next_event)

        self.deactivate()


    def _decode(self, data, data_size):
        decoded_data = 0

        for i in range(data_size):
            decoded_data = decoded_data << 4
            decoded_data += self.decoding_table[data[i]]

        return decoded_data
```

Eventos e Motores do Dumper

Eventos do dumper:

```
class MemReadingEvent(Event):
    """
    Event type handled by the MemReadingMotor
    """

    def __init__(self, event_type) -> None:
        super().__init__()

        self.type = event_type


class DataWritingEvent(Event):
    """
    Event type handled by the LineWritingMotor
    """

    def __init__(self, event_type, data, data_size) -> None:
        super().__init__()

        self.type = event_type
        self.data = data
        self.data_size = data_size


class LineWritingEvent(Event):
    """
    Event type handled by the FileWritingMotor
    """

    def __init__(self, event_type, line_data, line_size) -> None:
```

```
        super().__init__()

        self.type = event_type
        self.line_data = line_data
        self.line_size = line_size
```

Motores do dumper:

```
ADDRESS_SIZE = 3
DATA_SIZE = 2

class MemReadingMotor(EventsMotor):
    def __init__(self) -> None:
        super().__init__()

        self.reactions_table["start"] = self._start
        self.reactions_table["read"] = self._read
        self.reactions_table["end"] = self._end

        self.memory_pointer = None
        self.memory_position = 0
        self.start_poistion = 0
        self.end_position = 0

        self.encoding_table = {
            0x0 : "0",
            0x1 : "1",
            0x2 : "2",
            0x3 : "3",
            0x4 : "4",
            0x5 : "5",
            0x6 : "6",
            0x7 : "7",
            0x8 : "8",
            0x9 : "9",
```

```
            0xA : "A",
            0xB : "B",
            0xC : "C",
            0xD : "D",
            0xE : "E",
            0xF : "F"
        }


    def set_line_writing_motor(self, line_writing_motor) -> None:
        self.line_writing_motor = line_writing_motor


    def set_memory_pointer(self, memory_pointer) -> None:
        self.memory_pointer = memory_pointer


    def set_dumping_area(self, start_poistion, end_position) ->
None:
        self.start_poistion = start_poistion
        self.end_position = end_position


    def activate(self):
        super().activate()

        self.memory_position = 0


    def categorize_event(self, event: MemReadingEvent) -> str:
        if event.type == "start":
            return "start"
        elif event.type == "end" or self.memory_position ==
(self.end_position + 1):
            return "end"
        elif event.type == "read":
```

```
            return "read"


    def _start(self, event: MemReadingEvent):
        self.memory_position = self.start_poistion

        encoded_data = self._encode(self.start_poistion,
ADDRESS_SIZE)

        next_event = DataWritingEvent("start", encoded_data,
ADDRESS_SIZE)

        self.line_writing_motor.add_event(next_event)


    def _read(self, event: MemReadingEvent):
        mem_data = self.memory_pointer[self.memory_position]

        self.memory_position +=1

        encoded_data = self._encode(mem_data, DATA_SIZE)

        next_event = DataWritingEvent("mem_data", encoded_data,
DATA_SIZE)

        self.line_writing_motor.add_event(next_event)


    def _end(self, event: MemReadingEvent):
        print("[INFO] Finished dumping")

        next_event = DataWritingEvent("end", [""], 1)

        self.line_writing_motor.add_event(next_event)
```

```
    def _encode(self, data, data_size):
        decoded_data = []

        for i in reversed(range(data_size)):
            partial_data = data >> (4 * i)
            partial_data &= 0xF

            decoded_data.append(self.encoding_table[partial_data])

        return decoded_data
```

```
DATA_SIZE = 2

NUM_OF_DATA_PER_LINE = 16
NUM_OF_SPACES_SEPARATORS = 15
MAX_BYTES = NUM_OF_DATA_PER_LINE * DATA_SIZE +
NUM_OF_SPACES_SEPARATORS

class LineWritingMotor(EventsMotor):
    def __init__(self) -> None:
        super().__init__()

        self.reactions_table["start"] = self._start
        self.reactions_table["mem_data"] = self._mem_data
        self.reactions_table["line"] = self._line
        self.reactions_table["end"] = self._end

        self.read_data = []

        self.data_count = 0


    def set_file_writing_motor(self, file_writing_motor) -> None:
        self.file_writing_motor = file_writing_motor
```

```
def set_mem_reading_motor(self, mem_reading_motor) → None:
    self.mem_reading_motor = mem_reading_motor


def activate(self) → None:
    super().activate()

    self.read_data = []
    self.data_count = 0


def categorize_event(self, event: DataWritingEvent) → str:
    if event.type == "start":
        return "start"
    elif event.type == "mem_data":
        return "mem_data"
    elif event.type == "line":
        return "line"
    else:
        return "end"


def _start(self, event : DataWritingEvent) → None:
    data = event.data
    data.append("\n")

    next_event = LineWritingEvent("open_file", data, len(data))

    self.file_writing_motor.add_event(next_event)


def _mem_data(self, event : DataWritingEvent) → None:
    for i in range(event.data_size):
        self.read_data.append(event.data[i])
```

```
        self.data_count += 1

        if self.data_count < NUM_OF_DATA_PER_LINE:
            self.read_data.append(" ")

            next_event = MemReadingEvent("read")

            self.mem_reading_motor.add_event(next_event)
        else:
            next_event = DataWritingEvent("line", [], 0)

            self.add_event(next_event)


    def _line(self, event : DataWritingEvent) -> None:
        self.read_data.append("\n")

        next_event = LineWritingEvent("write", self.read_data,
MAX_BYTES + 1)

        self.file_writing_motor.add_event(next_event)

        self.read_data = []
        self.data_count = 0


    def _end(self, event : DataWritingEvent) -> None:
        if len(self.read_data) =/= 0:
            self.read_data[-1] = "\n"

        self.read_data.append("")

        next_event = LineWritingEvent("close_file", self.read_data,
len(self.read_data))

        self.file_writing_motor.add_event(next_event)
```

```
class FileWritingMotor(EventsMotor):
   def __init__(self) -> None:
       super().__init__()

       self.reactions_table["open_file"] = self._open_file
       self.reactions_table["write"] = self._write
       self.reactions_table["close_file"] = self._close_file
       self.reactions_table["error"] = self._error

       self.encoding_table = {
           ""   : b'',
           "\n" : b'\x0A',
           " "  : b'\x20',
           "0"  : b'\x30',
           "1"  : b'\x31',
           "2"  : b'\x32',
           "3"  : b'\x33',
           "4"  : b'\x34',
           "5"  : b'\x35',
           "6"  : b'\x36',
           "7"  : b'\x37',
           "8"  : b'\x38',
           "9"  : b'\x39',
           "A"  : b'\x41',
           "B"  : b'\x42',
           "C"  : b'\x43',
           "D"  : b'\x44',
           "E"  : b'\x45',
           "F"  : b'\x46'
       }


   def set_file_name(self, file_name):
       self.file_name = file_name
```

```
    def set_mem_reading_motor(self, mem_reading_motor) -> None:
        self.mem_reading_motor = mem_reading_motor


    def categorize_event(self, event: LineWritingEvent) -> str:
        if event.type == "open_file":
            return "open_file"
        elif event.type == "write":
            return "write"
        elif event.type == "close_file":
            return "close_file"
        else:
            return "error"


    def _open_file(self, event: LineWritingEvent) -> None:
        self.file = open(self.file_name, 'wb')

        next_event = LineWritingEvent("write", event.line_data,
event.line_size)

        self.add_event(next_event)


    def _write(self, event: LineWritingEvent) -> None:
        for i in range(event.line_size):
            encoded_data = self._encode(event.line_data[i])

            if encoded_data is not None:
                self.file.write(encoded_data)
            else:
                next_event = LineWritingEvent("error", [], 0)

                self.add_event(next_event)
```

```
        next_event = MemReadingEvent("read")

        self.mem_reading_motor.add_event(next_event)


    def _close_file(self, event: LineWritingEvent) -> None:
        # Write pending data
        for i in range(event.line_size):
            encoded_data = self._encode(event.line_data[i])

            if encoded_data is not None:
                self.file.write(encoded_data)

        self.file.close()

        self.deactivate()


    def _error(self, event: LineWritingEvent) -> None:
        print("[ERROR] Incorrect data type")

        next_event = LineWritingEvent("close_file", [], 0)

        self.add_event(next_event)


    def _encode(self, read_char):
        if read_char in self.encoding_table:
            return self.encoding_table[read_char]
        else:
            return None
```

Para juntar todas as execuções dos motores foram feitos os seguintes programas para o loader e para o dumper.

Consolidado do Loader

```
class Loader():
    def __init__(self) -> None:
        self.file_reading_motor = FileReadingMotor()
        self.line_reading_motor = LineReadingMotor()
        self.mem_storing_motor = MemStoringMotor()


self.file_reading_motor.set_line_reading_motor(self.line_reading_mo
tor)


self.line_reading_motor.set_file_reading_motor(self.file_reading_mo
tor)

self.line_reading_motor.set_mem_storing_motor(self.mem_storing_moto
r)


self.mem_storing_motor.set_file_reading_motor(self.file_reading_mot
or)


    def run(self, file_name, memory_pointer):
        self.file_reading_motor.set_file_name(file_name)
        self.mem_storing_motor.set_memory_pointer(memory_pointer)

        self.file_reading_motor.activate()
        self.line_reading_motor.activate()
        self.mem_storing_motor.activate()


self.file_reading_motor.add_event(FileReadingEvent("open_file"))

        all_motors_activeted = True
```

```
        while all_motors_activeted:
            self.file_reading_motor.run()
            self.line_reading_motor.run()
            self.mem_storing_motor.run()

            all_motors_activeted =
self.file_reading_motor.is_active() and \

self.line_reading_motor.is_active() and \

self.mem_storing_motor.is_active()

        self.file_reading_motor.deactivate()
        self.line_reading_motor.deactivate()
        self.mem_storing_motor.deactivate()
```

Consolidado do Dumper

```
class Dumper():
    def __init__(self) → None:
        self.file_writing_motor = FileWritingMotor()
        self.line_writing_motor = LineWritingMotor()
        self.mem_reading_motor = MemReadingMotor()


self.file_writing_motor.set_mem_reading_motor(self.mem_reading_moto
r)


self.line_writing_motor.set_file_writing_motor(self.file_writing_mo
tor)

self.line_writing_motor.set_mem_reading_motor(self.mem_reading_moto
r)
```

```
self.mem_reading_motor.set_line_writing_motor(self.line_writing_mot
or)


   def run(self, file_name, memory_pointer, start_poistion,
end_position):
       self.file_writing_motor.set_file_name(file_name)
       self.mem_reading_motor.set_memory_pointer(memory_pointer)
       self.mem_reading_motor.set_dumping_area(start_poistion,
end_position)

       self.file_writing_motor.activate()
       self.line_writing_motor.activate()
       self.mem_reading_motor.activate()

       self.mem_reading_motor.add_event(MemReadingEvent("start"))

       all_motors_activeted = True

       while all_motors_activeted:
           self.file_writing_motor.run()
           self.line_writing_motor.run()
           self.mem_reading_motor.run()

           all_motors_activeted =
self.file_writing_motor.is_active() and \

self.line_writing_motor.is_active() and \

self.mem_reading_motor.is_active()

       self.file_writing_motor.deactivate()
       self.line_writing_motor.deactivate()
       self.mem_reading_motor.deactivate()
```

Interface de usuário

Para o usuário escolher o que rodar, foi desenvolvida o seguinte programa:

```
def main():
    loader = Loader()
    dumper = Dumper()

    memory = Memory(4096)

    general_user_msg = """\nDentre as seguintes opções:\n
    l - Para fazer load de um arquivo
    d - Para fazer dump da memória
    z - Para zerar o conteúdo da memória
    i - Para imprimir o conteúdo da memória
    s - Para sair\n"""

    avaible_options = ["l", "d", "z", "i", "s"]

    stop = False

    try:
        while not stop:
            print(general_user_msg)
            option = input("Qual delas deseja executar? ")

            if option not in avaible_options:
                print("Opção inválida, escolha outra\n\n")
            else:
                if option == "l":
                      file_name = input("Qual o nome do arquivo a ser
carregado? ")

                    loader.run(file_name, memory)
```

```
            elif option == "d":
                    file_name = input("Qual o nome do arquivo no
qual fazer o dump? ")
                      start_position = int(input("Qual a posição
inicial da memória para fazer o dump? "), 16)
                    end_position = int(input("Qual a posição final
da memória para fazer o dump? "), 16)

                    dumper.run(file_name, memory, start_position,
end_position)
            elif option == "z":
                memory.clear()

                print("Memória zerada\n")
            elif option == "i":
                memory.display()

                  print("Dados da memória disponíveis no arquivo
image.txt\n")
            else:
                stop = True
    except KeyboardInterrupt:
        print("[INFO] Ending Program")


if __name__ == '__main__':
    main()
```

Testes

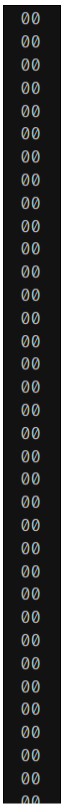
Conversão manual de testes

Convertendo-se o programa fornecido para o formato especificado pela prova se obteve o seguinte resultado:

010
80 32 90 37 90 34 90 36 80 37 50 35 10 30 80 37
40 32 90 37 80 34 40 33 90 34 40 36 90 36 00 18
C0 30 01 02 00 04 00 00

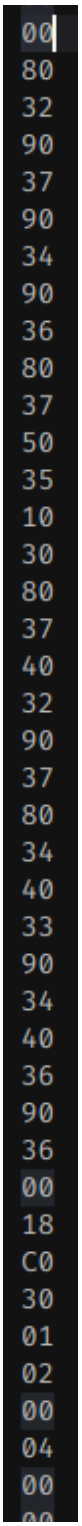
Conteúdo inicial da memória

Como a memória foi inicializada inicialmente com zeros, seu conteúdo inicial foi zero também, como se pode ver no seguinte trecho inicial da memória:



Conteúdo da memória após o loading do arquivo inicial

Após se carregar na memória, se obteve os seguintes dados no trecho em que a memória foi gravada:



Conteúdo do arquivo de dump

O conteúdo obtido após o dump foi o seguinte:

```
dump.txt
010
80 32 90 37 90 34 90 36 80 37 50 35 10 30 80 37
40 32 90 37 80 34 40 33 90 34 40 36 90 36 00 18
C0 30 01 02 00 04 00 00

```

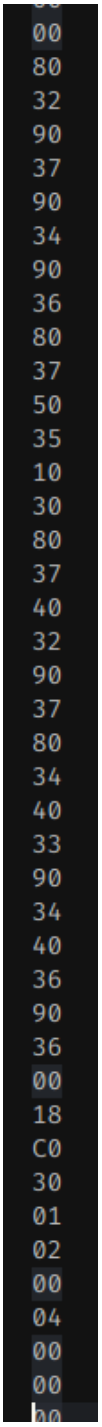
Conteúdo da memória após zerar

Após zerar a memória, todas as posições previamente escritas ficaram zero, como pode ser visto abaixo:

```
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
00
```

Conteúdo da memória após o loading do arquivo de dump

Após se carregar a memória novamente com o arquivo de dump gerado, se obteve os seguinte resultados na posições escritas:



Comparações e comentários

Os resultados obtidos ao imprimir o conteúdo inicial da memória e após zerá-la foram os mesmo, uma vez que a memória foi criada com vários zeros.

Já os resultados obtidos ao se fazer o load do código convertido na primeira etapa e do código gerado pelo dump foram iguais, uma vez que ambos usam o mesmo formato e são complementares, a saída de um é a entrada de outro, dessa forma se obteve os mesmos dados na memória nos dois casos