

Практическая работа №5.

Тема: «Структуры данных «стек» и «очередь».

Цель работы: изучить СД «стек» и «очередь», научиться их программировать и реализовывать.

Реализовать систему, представленную на рисунке 1.

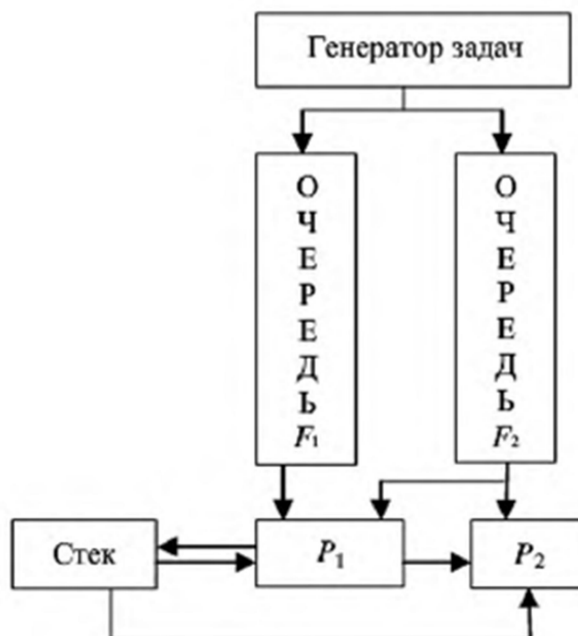


Рисунок 1. Система для реализации.

Задачи последовательно извлекаются из случайной очереди. Задачи первого типа выполняются только на первом процессоре, задачи второго типа выполняются на обоих процессорах, но в приоритете на втором. Задачи первого типа приоритетнее, чем второго типа на первом процессоре. Если оба процессора, способные выполнять какую-либо задачу, заняты, задача помещается в стек. Задачи из стека выполняются только после опустошения очередей.

Реализуем класс задачи, который предоставляет доступ к полям данных задачи (Листинг 1). Содержит поля двух типов: тип задания и время на выполнение задания, которые заполняются при инициализации класса.

					<i>АИСД.09.03.02.100000 ПР</i>		
Изм.	Лист	№ докум.	Подпись	Дат			
Разраб.		Кузнецов Д.В.			Практическая работа №5 «Структуры данных «стек» и «очередь».		
Провер.		Береза А.Н.					
Реценз							
Н. Контр.							
Утверд.							
					Лит.	Лист	Листов
						2	
					ИСОиП (филиал) ДГТУ в г.Шахты ИСТ-Тб21		

Листинг 1. Класс задачи.

```
@dataclass()
class TaskData:
    time: int = None
    type: int = None

class Task:
    def __init__(self, task_type, task_time):
        self.current_task = TaskData()
        self.current_task.time = task_time
        self.current_task.type = task_type

    def __str__(self):
        return '[' + str(self.get_type()) + ', ' +
str(self.get_time()) + ']'

    def get_time(self):
        return self.current_task.time

    def get_type(self):
        return self.current_task.type
```

Реализуем генератор задач (Листинг 2). Класс инициализируется двумя очередями для каждого типа задач. Публичный метод `gen_task` позволяет генерировать задачи, инициализируя класс `Task` случайными значениями из заданного диапазона и помещая его в соответствующую очередь. Публичный метод `get_task` позволяет получить задачу для выполнения. Диаграмма деятельности для этого метода представлена на рисунке 2. Публичный метод `none_task` возвращает истинное значение, если обе очереди пусты.

Листинг 2. Класс генератора задач.

```
class TaskGenerator:
    def __init__(self):
        self.queue1 = MyQueue()
        self.queue2 = MyQueue()
```

					<i>АуСД.09.03.02.100000 ПР</i>	Лист
Изм.	Лист	№ докум.	Подпись	Дата		3

```

def __str__(self):
    out = str(self.queue1) + '\n' + str(self.queue2)
    return out + '\n'

def gen_task(self):
    task = Task(rd.randint(1, 2), rd.randint(4, 8))
    if task.get_type() == 1:
        self.queue1.push(task)
    else:
        self.queue2.push(task)

def get_task(self):
    queue = rd.randint(1, 2)
    if queue == 1 and not self.queue1.check_empty():
        task = self.queue1.pop()
    elif queue == 2 and not self.queue2.check_empty():
        task = self.queue2.pop()
    elif queue == 1 and self.queue1.check_empty():
        task = self.queue2.pop()
    elif queue == 2 and self.queue2.check_empty():
        task = self.queue1.pop()
    else:
        task = None
    return task

def none_task(self):
    return self.queue1.check_empty() and
self.queue2.check_empty()

```

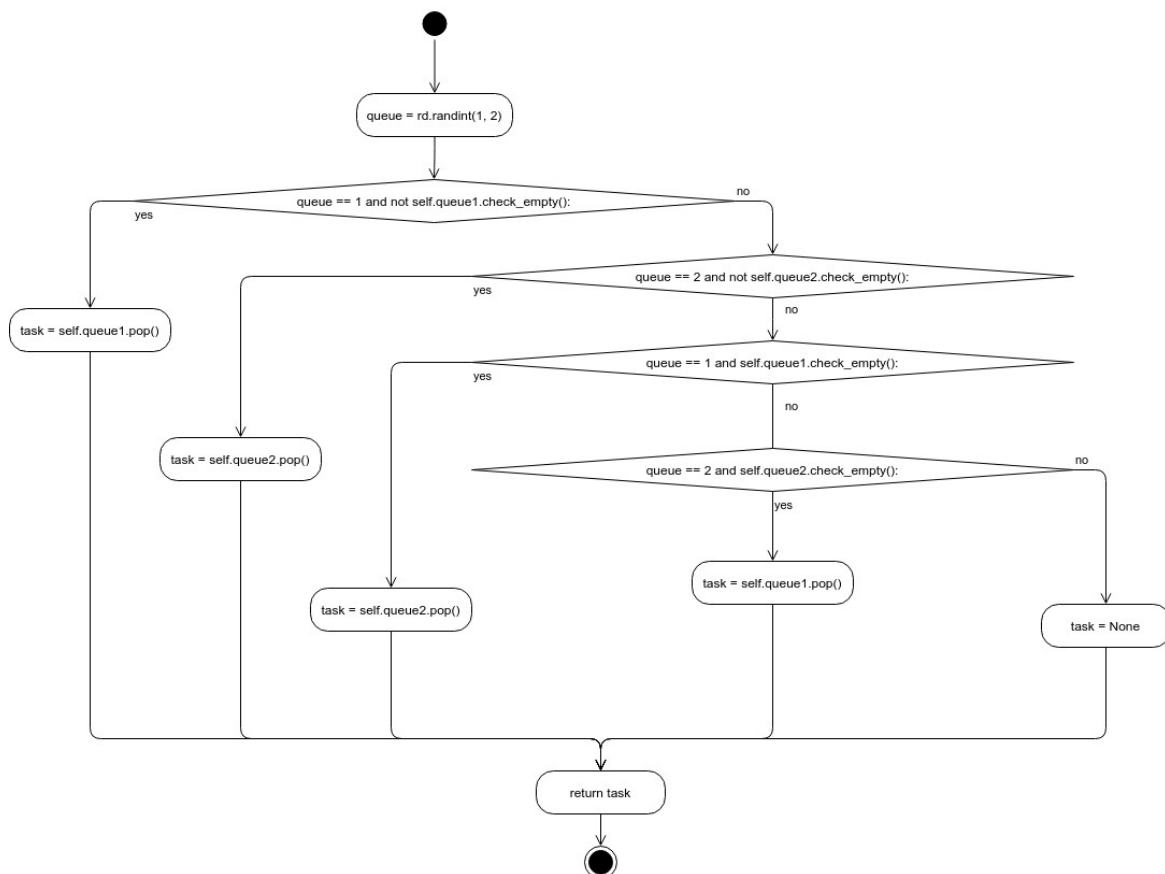


Рисунок 2. Диаграмма деятельности для метода get_task.

Реализуем класс процессора. Данный класс инициализируется двумя потоками класса данных Thread (хранит значения типа задачи, времени её выполнения и состояние простоя), соответствующих первому и второму процессору и стеком для отброшенных задач (Листинг 3). Публичный метод add_task позволяет добавлять задания на потоки. Его диаграмма деятельности представлена на рисунке 3. Приватные методы run_task_t1 и run_task_t2 как бы выполняют задачу, уменьшая значение времени выполнения на единицу за шаг цикла. Публичный метод running эти приватные методы для имитации работы процессора. Публичные методы idle_thread и idle_proc для проверки состояния простоя хотя бы одного ядра в первом случае, и всего процессора во втором.

Листинг 3. Класс процессора.

```

@dataclass()
class Thread:
    work_time: int = None
    task_type: int = None
  
```

```

        idle: bool = True

class Processor:
    def __init__(self):
        self.thread1 = Thread()
        self.thread2 = Thread()
        self.wait = MyStack()

    def __str__(self):
        out = '|thread|type|time|idle |\n'
        out += '{:<9}{:<5}{:<5}{:<6}'.format('1',
str(self.thread1.task_type), str(self.thread1.work_time),

str(self.thread1.idle)) + '\n'
        out += '{:<9}{:<5}{:<5}{:<6}'.format('2',
str(self.thread2.task_type), str(self.thread2.work_time),

str(self.thread2.idle))
        return out

    def add_task(self, task: Task):
        if task.get_type() == 1:
            if self.thread1.idle:
                self.thread1.task_type = task.get_type()
                self.thread1.work_time = task.get_time()
                self.thread1.idle = False
            elif self.thread1.task_type == 2:
                denied_task = Task(self.thread1.task_type,
self.thread1.work_time)
                self.thread1.task_type = task.get_type()
                self.thread1.work_time = task.get_time()
                self.wait.push(denied_task)
            else:
                self.wait.push(task)
        elif task.get_type() == 2:

```

```

        if self.thread2.idle:
            self.thread2.task_type = task.get_type()
            self.thread2.work_time = task.get_time()
            self.thread2.idle = False
        elif self.thread1.idle:
            self.thread1.task_type = task.get_type()
            self.thread1.work_time = task.get_time()
            self.thread1.idle = False
        else:
            self.wait.push(task)

def __run_task_t1(self):
    self.thread1.work_time -= 1
    if self.thread1.work_time <= 0:
        self.thread1.idle = True
        self.thread1.task_type = None
        self.thread1.work_time = None

def __run_task_t2(self):
    self.thread2.work_time -= 1
    if self.thread2.work_time <= 0:
        self.thread2.idle = True
        self.thread2.task_type = None
        self.thread2.work_time = None

def running(self):
    if not self.thread1.idle:
        self.__run_task_t1()
    else:
        self.thread1.idle = True
    if not self.thread2.idle:
        self.__run_task_t2()
    else:
        self.thread2.idle = True

def idle_thread(self):

```

```
return self.thread1.idle or self.thread2.idle
```

```
def idle_proc(self):
    return self.thread1.idle and self.thread2.idle
```

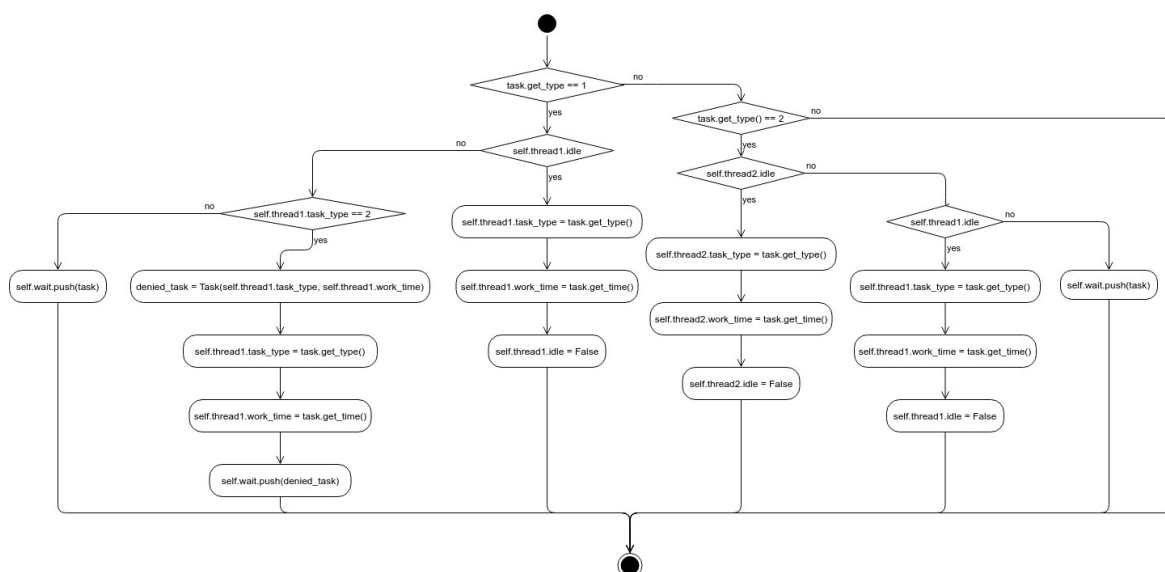


Рисунок 3. Диаграмма деятельности для метода add_task.

Исходный код программы представлен на листинге 4. Логика работы приведена на диаграмме деятельности (Рисунок 4).

Листинг 4. Исходный код программы.

```
from Stack_and_Queue.processor import Processor
from Stack_and_Queue.task import TaskGenerator
```

```
generator = TaskGenerator()
processor = Processor()
```

```
for i in range(50):
    generator.gen_task()
```

```
while True:
    task = generator.get_task()
    if processor.idle_thread():
        if not generator.none_task():
            processor.add_task(task)
        elif not processor.wait.check_empty():
```

Изм.	Лист	№ докум.	Подпись	Дата

АуСД.09.03.02.100000 ПР

Лист

8

```

        processor.add_task(processor.wait.pop())
processor.running()
print('Tasks\n', generator)
print('Processor:\n', processor)
print('Stack:', processor.wait)
if          generator.none_task()          and
processor.wait.check_empty() and processor.idle_proc():
    break

```

					<i>AuCD.09.03.02.100000 IP</i>	Лист
						9
Изм.	Лист	№ докум.	Подпись	Дата		

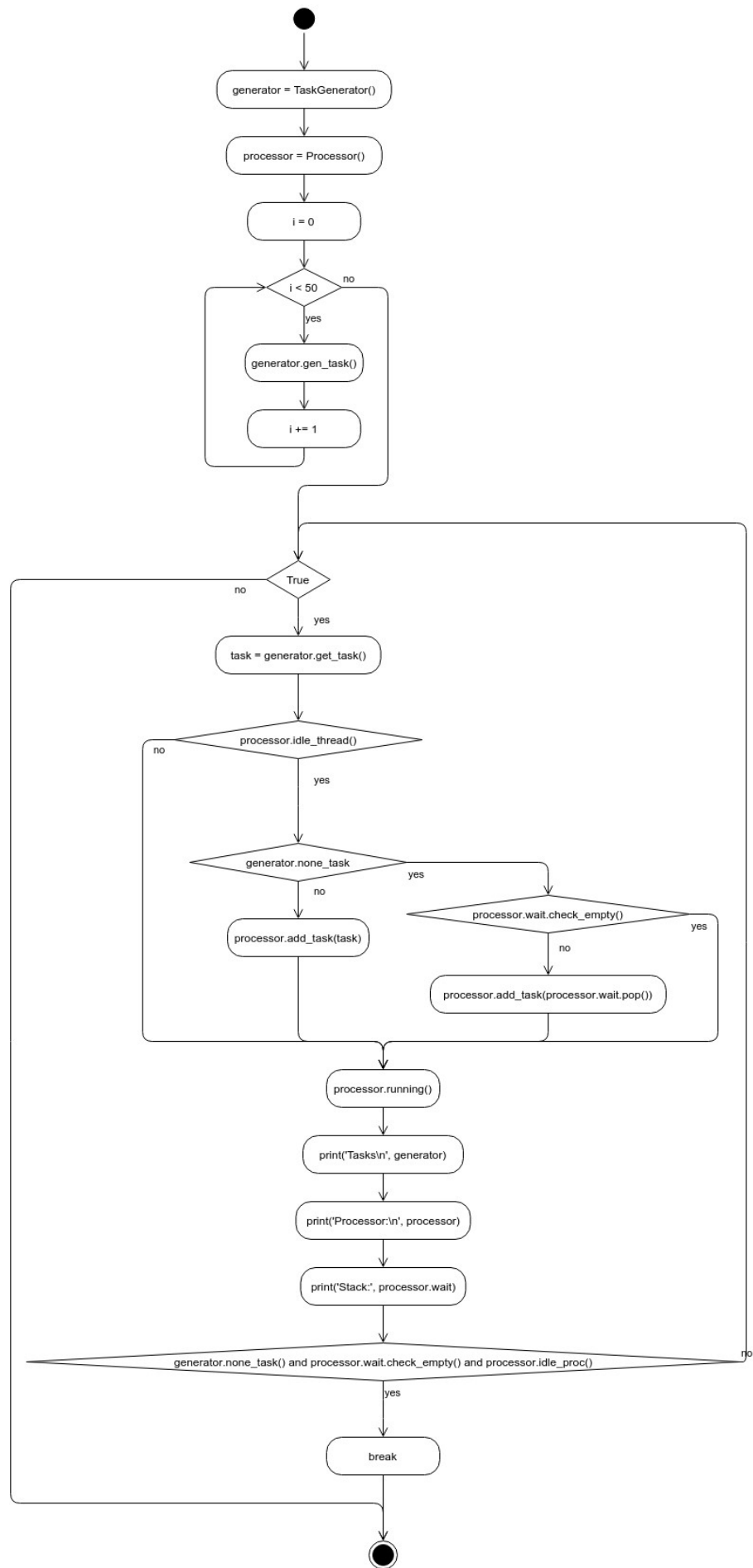


Рисунок 4. Диаграмма деятельности для программы.

Вывод: в ходе выполнения данной практической работы были изучены структуры данных «стек» и «очередь», и их программные реализации и использование.