Практическая работа №5

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

Цель работы: изучить структуры данных «стек» и «очередь», научиться их программно реализовывать и использовать.

Реализовать систему, представленную на рисунке 1.

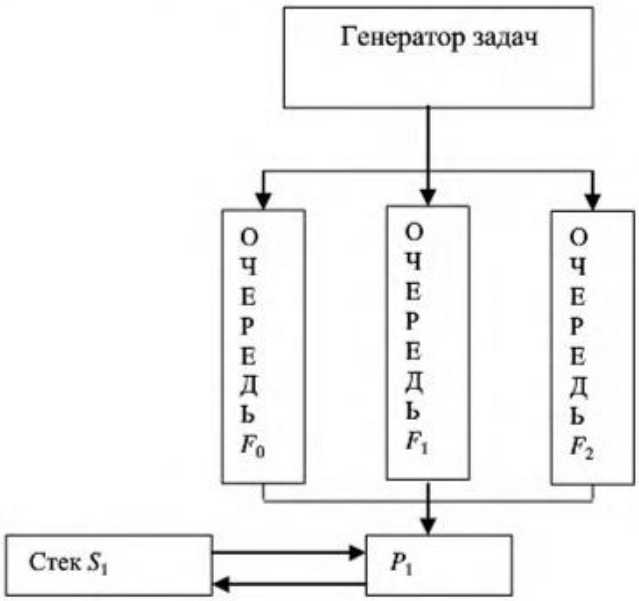

Рисунок 1. Система для реализации.

Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F0. Если она пуста, можно обрабатывать задачи из очереди F1. Если и она пуста, то можно обрабатывать задачи из очереди F2. Если все очереди пусты, то система находится в ожидании поступающих задач, либо в режиме обработки предыдущей задачи. Если поступает задача с более высоким приоритетом, чем обрабатываемая в данный момент, то обрабатываемая помещается в стек и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.

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.

Листинг 1. Класс задачи.

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


class Task:
    def __init__(self, task_priority, task_time):
        self.current_task = TaskData()
        self.current_task.time = task_time
        self.current_task.priority = task_priority

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

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

    def get_priority(self):
        return self.current_task.priority
```

Реализуем генератор задач (Листинг 2). Класс инициализируется тремя 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. Публичный метод none_task возвращает истинное значение, если все очереди пусты.

Листинг 2. Класс генератора задач.

```
class TaskGenerator:
    def __init__(self):
        self.queue1 = Queue()
        self.queue2 = Queue()
        self.queue3 = Queue()

    def __str__(self):
```

```
        out = str(self.queue1) + '\n' + str(self.queue2) + '\n' + str(self.queue3)
        return out + '\n'

    def gen_task(self):
        task = Task(rd.randint(1, 3), rd.randint(4, 8))
        if task.get_priority() == 1:
            self.queue1.push(task)
        elif task.get_priority() == 2:
            self.queue2.push(task)
        else:
            self.queue3.push(task)

    def get_task(self):
        if not self.queue1.check_empty():
            task = self.queue1.pop()
        elif not self.queue2.check_empty():
            task = self.queue2.pop()
        elif not self.queue3.check_empty():
            task = self.queue3.pop()
        else:
            task = None
        return task

    def none_task(self):
        return self.queue1.check_empty() and self.queue2.check_empty() and
self.queue3.check_empty()
```

Реализуем класс процессора. Данный класс инициализируется потоками класса данных Thread (хранит значения типа задачи, времени её выполнения и состояние простоя), соответствующих процессору и стеком для отброшенных задач (Листинг 3). Публичный метод add_task позволяет добавлять задания на потоки. Приватный метод run_task_thread как бы выполняет задачу, уменьшая значение времени выполнения на единицу за шаг цикла. Публичный метод running эти приватные методы для имитации работы процессора. Публичный метод idle_proc для проверки состояния простоя хотя бы одного ядра в первом случае, и процессора во втором.

Листинг 3. Класс процессора.

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

```
    task_priority: int = None
    idle: bool = True


class Processor:
    def __init__(self):
        self.thread = Thread()
        self.wait = Stack()

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

    def add_task(self, task: Task):
        if self.thread.idle:
            self.thread.task_priority = task.get_priority()
            self.thread.work_time = task.get_time()
            self.thread.idle = False
        else:
            self.wait.push(task)

    def __run_task_thread(self):
        self.thread.work_time -= 1
        if self.thread.work_time <= 0:
            self.thread.idle = True
            self.thread.task_priority = None
            self.thread.work_time = None

    def running(self):
        if not self.thread.idle:
            self.__run_task_thread()
        else:
            self.thread.idle = True

    def idle_proc(self):
        return self.thread.idle
```

Исходный код программы представлен на листинге 4.

Листинг 4. Исходный код программы.

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

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

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

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