МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**ОТЧЕТ ПО ЛАБО**РАТОРНОЙ РАБО**ТЕ № 5**

**по дисциплине**  
 **«МНОГОАГЕНТНОЕ МОДЕЛИРОВАНИЕ»**

Выполнил студент группы 45/2                                  Т. Э. Айрапетов

Направление подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем

Курс    4

Отчет принял доктор физико-математических наук, профессор                                                                                       А.И. Миков

Краснодар

2024 г.

**Задание**:

1. Разработать алгоритм распределения и динамического перераспределения нагрузки в многоагентной системе. Входными (изменяемыми) данными являются граф нагрузки и граф МАС (многоагентной системы).

2. Реализовать модель нагрузки, модель многоагентной системы.

З. Реализовать алгоритм распределения и динамического перераспределения нагрузки в многоагентной системе для совместной работы с моделями нагрузки и МАС.

1. Продемонстрировать работу алгоритма на 3 - 4 тестовых примерах с различными моделями нагрузки и МАС (графы нагрузки и МАС и поэтапное изменение нагрузки агентов модулями приложения). Для каждого теста определить общее время выполнения приложения.

**Решение.**

Для решения задачи были разработаны классы LP и Agent. Класс LP представляет собой программный модуль и соответствующий ему логический процесс. В нем реализованы следующие поля:

* *load* - величина нагрузки;
* *loading* - время с начала запуска задачи;
* *before* - множество процессов-родителей;
* *after* - множество процессов-потомков;
* *denial\_time* - время отказа, либо -1;
* *agent* - агент, выполняющий процесс.

Также в классе LP реализован метод *possible\_to\_start*, возвращающий True, если все процессы-родители были выполнены и False иначе. Для определения времени отказа (*denial\_time*) была реализована функция, которая с вероятностью 0.05 возвращает время отказа и с вероятностью 0.95 возвращает -1 (отказа не будет).

В классе Agent реализованы следующие поля:

* *ended\_tasks* - количество выполненных задач;
* *cur\_task* - ссылка на текущий LP;
* *tasks* - список набранных задач;
* *neighbors* - множество соседей агента.

Также в классе реализована функция подсчёта общей нагрузки агента (сумма нагрузок всех задач в списке + нагрузка текущей задачи).

Модель нагрузки представляется ориентированным нагруженным графом. Для создания тестовых используется функция *gen\_lp*, принимающая на вход кол-во «слоёв» программы (независимые друг от друга программные модули, находящиеся на одном уровне от стартовых модулей), минимальную нагрузку модуля и максимальную нагрузку модуля. Функция случайным образом в заданных диапазонах создает граф нагрузки, связывая каждый следующий слой с предыдущим так, чтобы у каждой задачи в слое был как минимум предок. На рисунке 1 можно увидеть пример создания графа нагрузки. Для каждого процесса в указана информация в формате:

(<нагрузка>) LP<индекс> i/o=<степень по входу>/<степень по выходу>

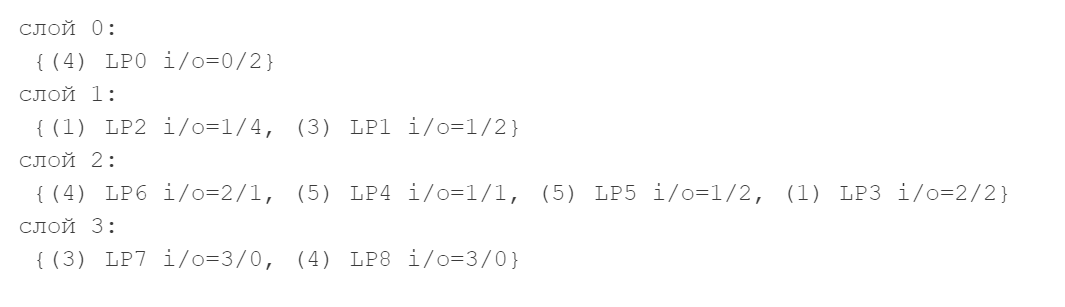


Рисунок 1 - Пример графа нагрузки (без указания отдельных ребёр)

Модель многоагентной системы представляется неорентированным графом, поэтому для создания тестовых примеров была реализована функция *gen\_agents*, принимающая на вход кол-во агентов в системе. Функция создает *n* агентов и связывает их таким образом, чтобы каждый агент имел как минимум одного соседа. При этом не гарантируется, что в графе будет только 1 компонента связности. На рисунке 2 можно увидеть пример создания графа многоагентной системы.

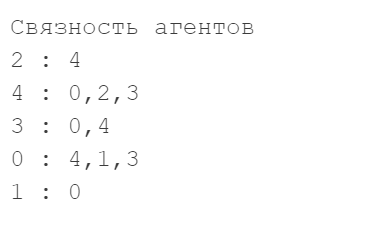


Рисунок 2 - Список связности графа многоагентной системы

Для моделирования распределения нагрузки в системе была создана функция *system*, проводящая как первичное распределение, так и перераспределение.

При распределении нагрузки используется функция *get\_most\_idle\_agent*, которая для всех агентов в системе вычисляет наиболее свободного по 2 показателям: его общая нагрузка (метод *total\_load*) и кол-во соседей. То есть вначале функция смотрит кто из агентов менее загружен задачами, а если есть совпадения, то выбирается агент с наибольшим числом соседей. Таким образом, агент с большим числом соседей будет иметь возможность перераспределить нагрузку более эффективно.

Далее в цикле с условием на завершение всех задач, происходит проверка всех агентов на предмет завершения какой-либо задачи, либо отказ в обслуживании. При отказе происходит перераспределение задачи на наиболее свободного агента. Пример моделирования можно увидеть на рисунке 3. Во время работы программы в консоль выводятся логи событий (завершение задачи, отказ в обслуживании). Система завершает свою работу с завершением последнего процесса. После завершения системы также выводится и максимальное время работы (время выполнения всех задач при последовательном выполнении).

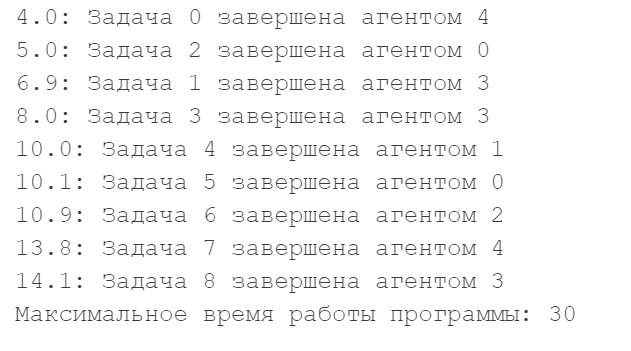


Рисунок 3 - Пример запуска системы

Также после завершения работы можно вывести количество решенных задач по агентам. На рисунке 4 представлен вывод этих данных. Можно обратить внимание, что агенты с меньшим кол-вом соседей получают задачи в среднем реже.

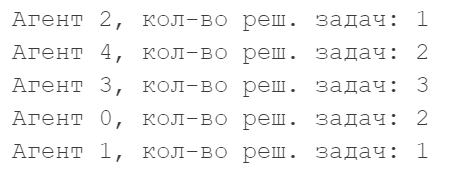


Рисунок 4 - Количество решенных задач агентами

Далее представлены еще несколько запусков при разных графах нагрузки и многоагентной системы.

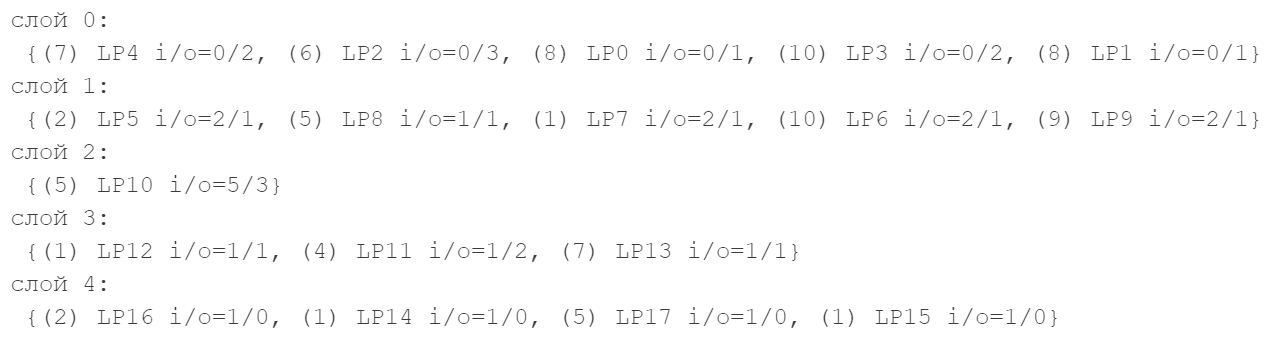


Рисунок 5 - Граф нагрузки 5 слоев, мин. нагрузка 1, макс. нагрузка 10

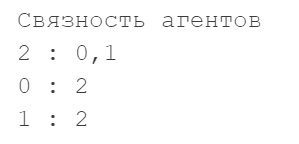
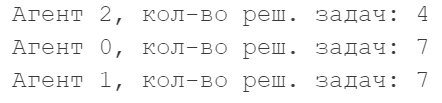
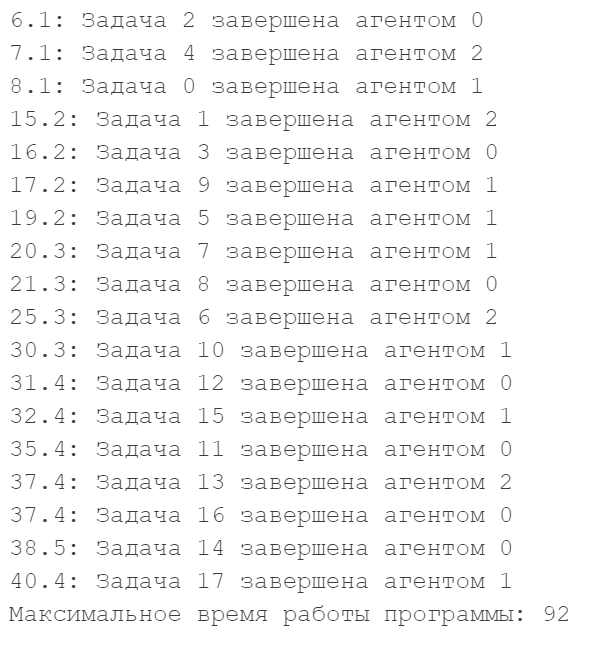


Рисунок 6 - Граф МАС для 3 агентов



Риснуок 7 - Запуск системы при условиях рис. 5 и 6

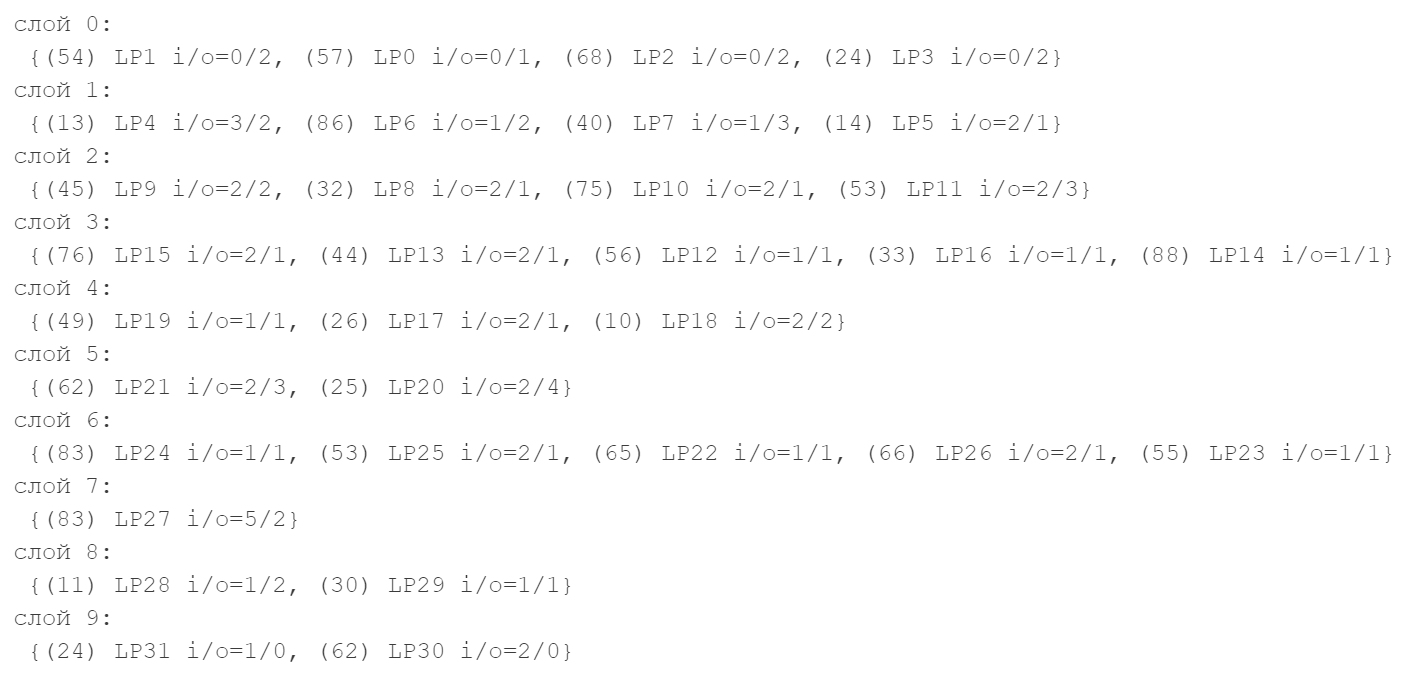


Рисунок 8 - Граф нагрузки 10 слоев, мин. нагрузка 10, мак. нагрузка 100

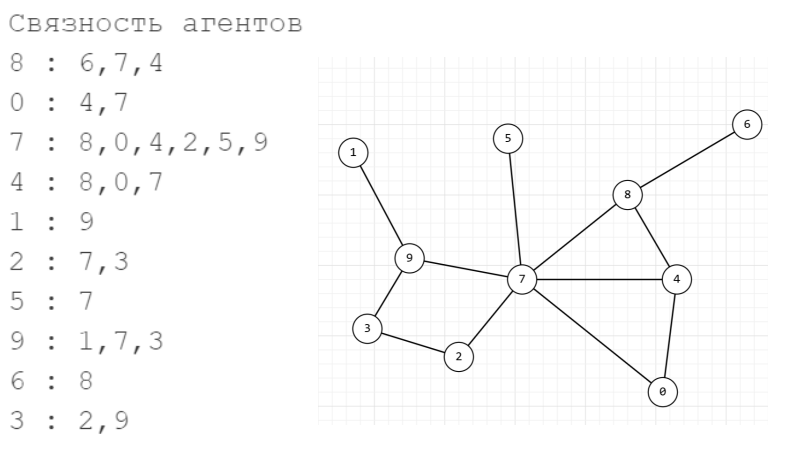
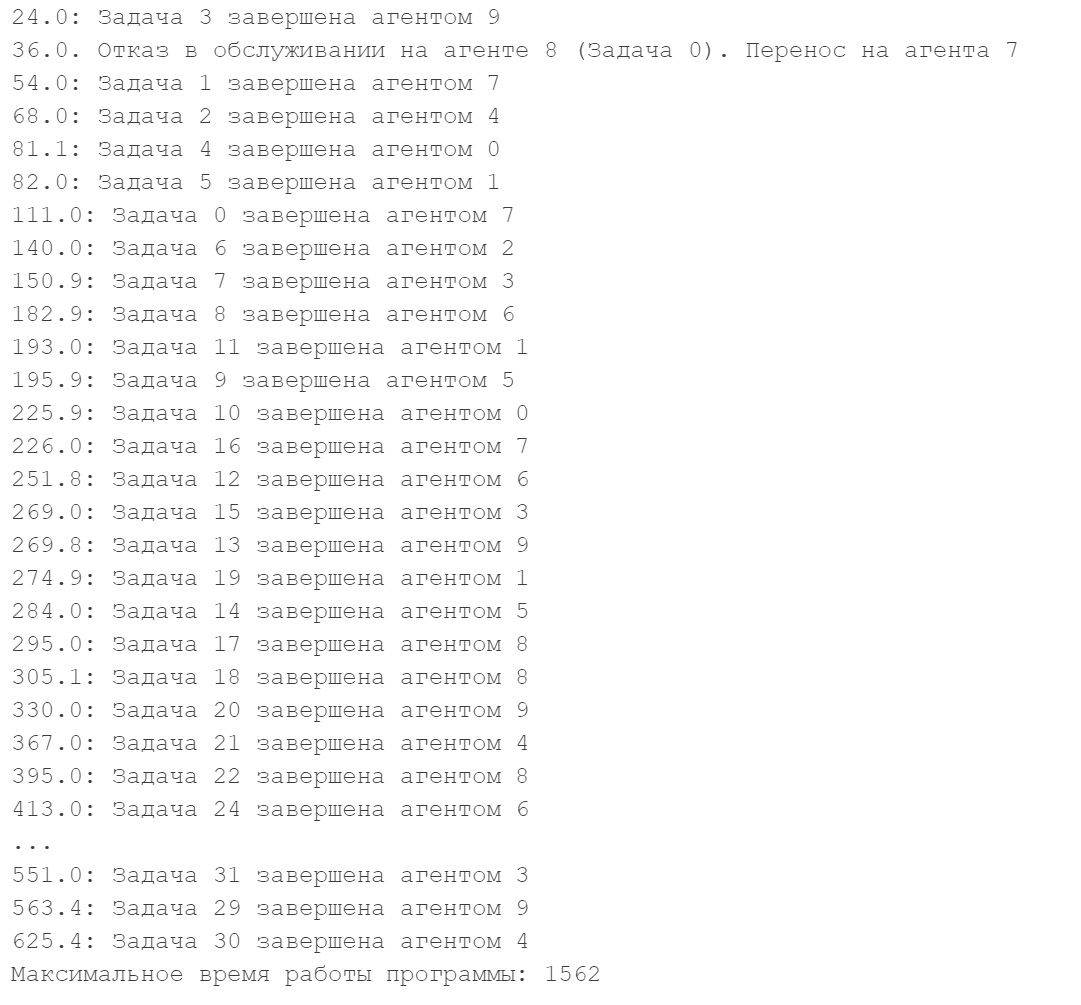


Рисунок 9 - Граф МАС 10 агентов



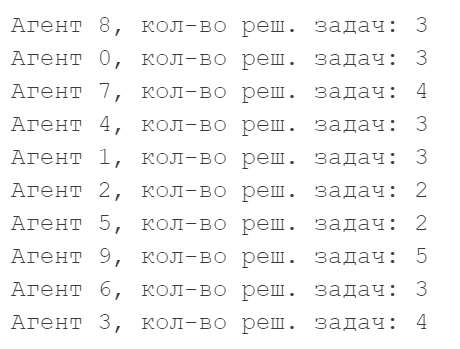


Рисунок 10 - Запуск системы при условиях рис. 8 и 9

Текст программы на языке Python:

*import* numpy *as* np

*# время отказа, либо -1*

def denial\_of\_service(*load*, *p*=0.05):

    b=np.random.choice([0,1], *p*=[1-*p*,*p*])

*if* b:

*return* round(np.random.choice(np.arange(0, *load*, 0.1)), 2)

*return* -1

class LP:

    def \_\_init\_\_(*self*, *ind*, *load*):

*self*.ind=*ind*

*self*.load = *load*

*self*.loading = 0

*self*.before = set()

*self*.after = set()

*self*.denial\_time = denial\_of\_service(*load*)

*self*.agent=None

    def possible\_to\_start(*self*):

*return* all(map(lambda *x*:*x*.load<=*x*.loading, *self*.before))

    def \_\_repr\_\_(*self*):

*return* f"({*self*.load}) LP{*self*.ind} i/o={len(*self*.before)}/{len(*self*.after)}"

    def \_\_str\_\_(*self*):

*return* *self*.\_\_repr\_\_()

class Agent:

    def \_\_init\_\_(*self*, *ind*):

*self*.ind=*ind*

*self*.ended\_tasks = 0

*self*.cur\_task = None

*self*.tasks = list()

*self*.neighbors = set()

    def total\_load(*self*):

*return* sum([i.load *for* i *in* *self*.tasks]) + (*self*.cur\_task.load *if* *self*.cur\_task *else* 0)

    def \_\_repr\_\_(*self*):

*return* f"Agent{*self*.ind}, load={*self*.total\_load()}, {len(*self*.neighbors)} neighbors"

    def \_\_str\_\_(*self*):

*return* *self*.\_\_repr\_\_()

*from* numpy.random *import* choice

def gen\_lp(*n\_layers*=4, *min\_load*=1, *max\_load*=5):

    ind = 0

    arch = choice(5, *n\_layers*)+1

    def gen(*k*):

        nonlocal ind

        lps = set()

*for* \_ *in* range(*k*):

            lps.add(LP(ind, np.random.randint(*min\_load*, *max\_load*+1)))

            ind += 1

*return* lps

    def connect(*la*,*lb*):

        n = len(*la*)

*for* i *in* *lb*:

*for* j *in* choice(list(*la*), choice(n//2+1)+1):

                i.before.add(j)

                j.after.add(i)

*for* i *in* *la*:

*if* not i.after:

                t = choice(list(*lb*), choice(1)+1)

*for* j *in* t:

                    i.after.add(j)

                    j.before.add(i)

    st = gen(arch[0])

    la = st

    min\_time = max(la, *key*=lambda *x*:*x*.load).load

*# print("слой 0:\n",la)*

    j = 1

*for* i *in* arch[1:]:

        lb = gen(i)

        connect(la,lb)

        print(f"слой {j-1}:\n",la)

        j+=1

        la = lb

        min\_time += max(la, *key*=lambda *x*:*x*.load).load

    print(f"слой {j-1}:\n",lb)

*return* st, min\_time

def gen\_agents(*n*=4):

    agents = set()

*for* i *in* range(*n*):

        agents.add(Agent(i))

*for* a *in* agents:

*if* not a.neighbors:

            t = choice(list(agents-set([a])), choice(*n*) + 1)

*for* b *in* t:

*if* a != b:

                    a.neighbors.add(b)

                    b.neighbors.add(a)

    print("Связность агентов")

*for* a *in* agents:

        print(a.ind, ': '+','.join([str(i.ind) *for* i *in* a.neighbors]))

*return* agents

def get\_most\_idle\_agent(*agents*):

    n = len(*agents*)

*return* sorted(*agents*, *key*=lambda *x*:(*x*.total\_load(), n-len(*x*.neighbors)))[0]

def system(*st*, *agents*):

    visited = set()

    stack = list(*st*)

    max\_time = 0

*while* stack:

        i = stack.pop(0)

*if* not (i in visited or i.agent):

            max\_time += i.load

            ag = get\_most\_idle\_agent(*agents*)

            ag.tasks.append(i)

            i.agent = ag

            visited.add(i)

            stack.extend(list(i.after))

    dt = 0.1

    t = 0

*while* visited:

        t += dt

*for* a *in* *agents*:

*if* a.tasks or a.cur\_task:

*if* not a.cur\_task:

*for* i *in* range(len(a.tasks)):

*if* a.tasks[i].possible\_to\_start():

                            a.cur\_task = a.tasks.pop(i)

*break*

*else*:

*continue*

                task = a.cur\_task

                task.loading += dt

*if* (task.denial\_time != -1) and (task.denial\_time <= task.loading):

                    ag1 = get\_most\_idle\_agent(a.neighbors)

                    i = 0

*for* i *in* range(len(ag1.tasks)):

*if* not ag1.tasks[i].possible\_to\_start():

*break*

                    ag1.tasks.insert(i, task)

                    a.cur\_task = None

                    task.loading=0

                    task.denial\_time = -1

                    print(f"{t:.1f}. Отказ в обслуживании на агенте {a.ind} (Задача {task.ind}). Перенос на агента {ag1.ind}")

*elif* task.loading >= task.load:

                    print(f"{t:.1f}: Задача {task.ind} завершена агентом {a.ind}")

                    a.ended\_tasks += 1

                    visited.remove(task)

                    a.cur\_task = None

    print("Максимальное время работы программы:", max\_time)

st, min\_time = gen\_lp(10, 10, 100)

agents = gen\_agents(10)

system(st, agents)

**Выводы.** В многоагентных системах стратегия распределения нагрузки является важным аспектом для её работоспособности, так как грамотная балансировка может во много раз уменьшить время выполнения задач. Так, в нашем случае, стратегию можно улучшить, более детально сравнивая агентов между собой. Также можно участить операции по перераспределению, однако это может и увеличить время работы системы.