

Лабораторная работа №11

Модель системы массового обслуживания $M|M|1$

Шуваев Сергей Александрович

Содержание

| | | |
|----------|--|-----------|
| 1 | Введение | 4 |
| 2 | Выполнение лабораторной работы | 5 |
| 2.1 | Мониторинг параметров моделируемой системы | 11 |
| 3 | Выводы | 19 |

Список иллюстраций

| | | |
|------|---|----|
| 2.1 | Граф сети системы обработки заявок в очереди | 6 |
| 2.2 | Граф генератора заявок системы | 6 |
| 2.3 | Граф процесса обработки заявок на сервере системы | 7 |
| 2.4 | Задание деклараций системы | 8 |
| 2.5 | Параметры элементов основного графа системы обработки заявок в очереди | 9 |
| 2.6 | Параметры элементов генератора заявок системы | 10 |
| 2.7 | Параметры элементов обработчика заявок системы | 11 |
| 2.8 | Функция Predicate монитора Ostanovka | 11 |
| 2.9 | Функция Observer монитора Queue Delay | 12 |
| 2.10 | Файл Queue_Delay.log | 13 |
| 2.11 | График изменения задержки в очереди | 14 |
| 2.12 | Функция Observer монитора Queue Delay Real | 14 |
| 2.13 | Содержимое Queue_Delay_Real.log | 15 |
| 2.14 | Функция Observer монитора Long Delay Time | 16 |
| 2.15 | Определение longdelaytime в декларациях | 16 |
| 2.16 | Содержимое Long_Delay_Time.log | 17 |
| 2.17 | Периоды времени, когда значения задержки в очереди превышали заданное значение | 18 |

1 Введение

Цель работы

Реализовать модель $M|M|1$ в CPN tools.

Задание

- Реализовать в CPN Tools модель системы массового обслуживания $M|M|1$.
- Настроить мониторинг параметров моделируемой системы и нарисовать графики очереди.

2 Выполнение лабораторной работы

Постановка задачи

В систему поступает поток заявок двух типов, распределённый по пуассоновскому закону. Заявки поступают в очередь сервера на обработку. Дисциплина очереди - FIFO. Если сервер находится в режиме ожидания (нет заявок на сервере), то заявка поступает на обработку сервером.

Будем использовать три отдельных листа: на первом листе опишем граф системы (рис. 2.1), на втором — генератор заявок (рис. 2.2), на третьем — сервер обработки заявок (рис. 2.3).

Сеть имеет 2 позиции (очередь — Queue, обслуженные заявки — Complited) и два перехода (генерировать заявку — Arrivals, передать заявку на обработку серверу — Server). Переходы имеют сложную иерархическую структуру, задаваемую на отдельных листах модели (с помощью соответствующего инструмента меню — Hierarchy).

Между переходом Arrivals и позицией Queue, а также между позицией Queue и переходом Server установлена дуплексная связь. Между переходом Server и позицией Complited — односторонняя связь.

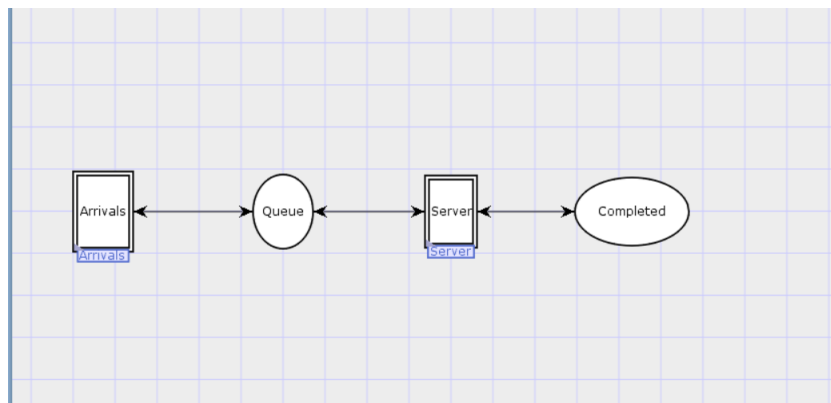


Рис. 2.1: Граф сети системы обработки заявок в очереди

Граф генератора заявок имеет 3 позиции (текущая заявка — Init, следующая заявка — Next, очередь — Queue из листа System) и 2 перехода (Init — определяет распределение поступления заявок по экспоненциальному закону с интенсивностью 100 заявок в единицу времени, Arrive — определяет поступление заявок в очередь).

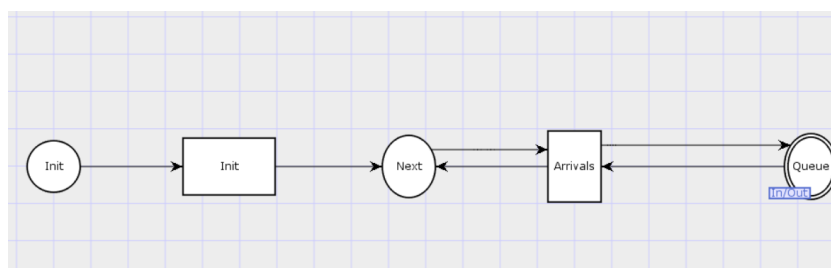


Рис. 2.2: Граф генератора заявок системы

Граф процесса обработки заявок на сервере имеет 4 позиции (Busy — сервер занят, Idle — сервер в режиме ожидания, Queue и Completed из листа System) и 2 перехода (Start — начать обработку заявки, Stop — закончить обработку заявки).

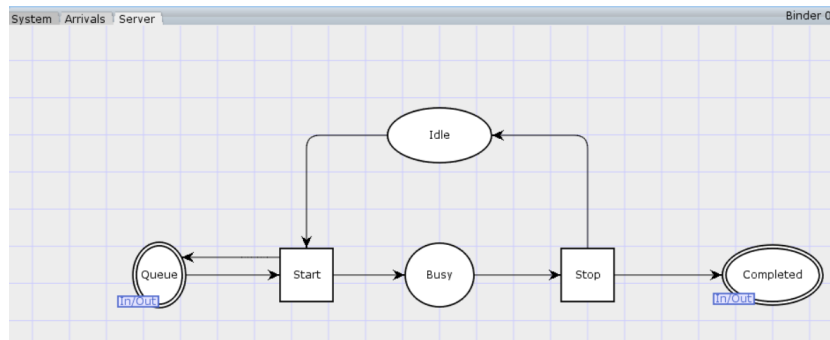


Рис. 2.3: Граф процесса обработки заявок на сервере системы

Зададим декларации системы (рис. 2.4).

Определим множества цветов системы (colorset):

- фишки типа UNIT определяют моменты времени;
- фишки типа INT определяют моменты поступления заявок в систему.
- фишки типа JobType определяют 2 типа заявок — А и В;
- кортеж Job имеет 2 поля: jobType определяет тип работы (соответственно имеет тип JobType, поле AT имеет тип INT и используется для хранения времени нахождения заявки в системе);
- фишки Jobs — список заявок;
- фишки типа ServerxJob — определяют состояние сервера, занятого обработкой заявок.

Переменные модели:

- proctime — определяет время обработки заявки;
- job — определяет тип заявки;
- jobs — определяет поступление заявок в очередь.

Определим функции системы:

- функция expTime описывает генерацию целочисленных значений через интервалы времени, распределённые по экспоненциальному закону;
- функция intTime преобразует текущее модельное время в целое число;

- функция `newJob` возвращает значение из набора `Job` — случайный выбор типа заявки (A или B).

```

▼ Declarations
  ► Standard priorities
  ▼ Standard declarations
    ► colset BOOL
    ► colset INTINF
    ▼ colset TIME = time;
    ► colset REAL
    ► colset STRING
  ▼ SYSTEM
    ▼ colset UNIT = unit timed;
    ▼ colset INT = int;
    ▼ colset Server = with server timed;
    ▼ colset JobType = with A | B;
    ▼ colset Job = record jobType : JobType *
      AT : INT;
    ▼ colset Jobs = list Job;
    ▼ colset ServerxJob = product Server * Job timed;
    ▼ var proctime : INT;
    ▼ var job : Job;
    ▼ var jobs : Jobs;
    ▼ fun expTime (mean: int) =
      let
        val realMean = Real.fromInt mean
        val rv = exponential((1.0/realMean))
      in
        floor (rv+0.5)
      end;
    ▼ fun intTime() = IntInf.toInt (time());
    ▼ fun newJob() = {jobType = JobType.ran(),
      AT = intTime()}
  ► Monitors

```

Рис. 2.4: Задание деклараций системы

Зададим параметры модели на графах сети.

На листе `System` (рис. 2.5):

- у позиции `Queue` множество цветов фишек — `Jobs`; начальная маркировка `1[]` определяет, что изначально очередь пуста.
- у позиции `Completed` множество цветов фишек — `Job`.

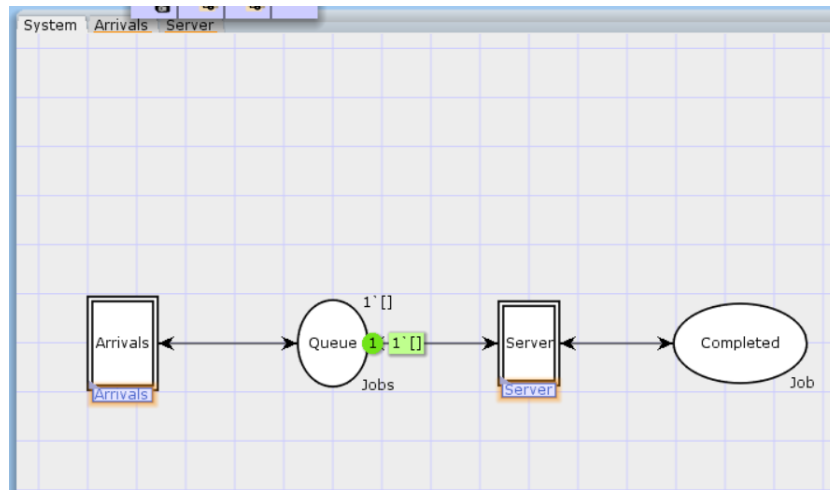


Рис. 2.5: Параметры элементов основного графа системы обработки заявок в очереди

На листе Arrivals (рис. 2.6):

- у позиции Init: множество цветов фишек — UNIT; начальная маркировка $1' \text{ } () @ 0$ определяет, что поступление заявок в систему начинается с нулевого момента времени;
- у позиции Next: множество цветов фишек — UNIT;
- на дуге от позиции Init к переходу Init выражение $()$ задаёт генерацию заявок;
- на дуге от переходов Init и Arrive к позиции Next выражение $() @ \text{expTime}(100)$ задаёт экспоненциальное распределение времени между поступлениями заявок;
- на дуге от позиции Next к переходу Arrive выражение $()$ задаёт перемещение фишки;
- на дуге от перехода Arrive к позиции Queue выражение $\text{jobs}^{\wedge}[\text{job}]$ задает поступление заявки в очередь;
- на дуге от позиции Queue к переходу Arrive выражение jobs задаёт обратную связь.

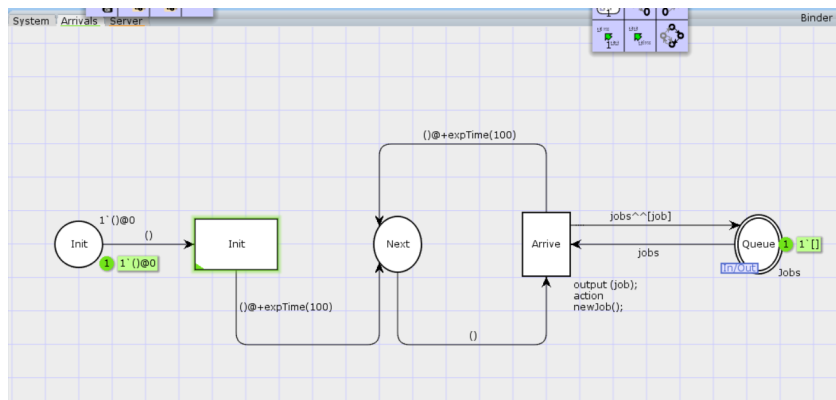


Рис. 2.6: Параметры элементов генератора заявок системы

На листе Server (рис. 2.7):

- у позиции Busy: множество цветов фишек — Server, начальное значение мар-кировки — $1' \text{ server} @ 0$ определяет, что изначально на сервере нет заявок на обслуживание;
- у позиции Idle: множество цветов фишек — $\text{Server} \times \text{Job}$;
- переход Start имеет сегмент кода `output (proctime); action expTime(90);` определяющий, что время обслуживания заявки распределено по экспоненциальному закону со средним временем обработки в 90 единиц времени;
- на дуге от позиции Queue к переходу Start выражение `job : : jobs` определяет, что сервер может начать обработку заявки, если в очереди есть хотя бы одна заявка;
- на дуге от перехода Start к позиции Busy выражение `(server , job) @ +proctime` запускает функцию расчёта времени обработки заявки на сервере;
- на дуге от позиции Busy к переходу Stop выражение `(server , job)` говорит о завершении обработки заявки на сервере;
- на дуге от перехода Stop к позиции Completed выражение `job` показывает, что заявка считается обслуженной;
- выражение `server` на дугах от и к позиции Idle определяет изменение состояние сервера (обрабатывает заявки или ожидает);

- на дуге от перехода Start к позиции Queue выражение jobs задаёт обратную СВЯЗЬ.

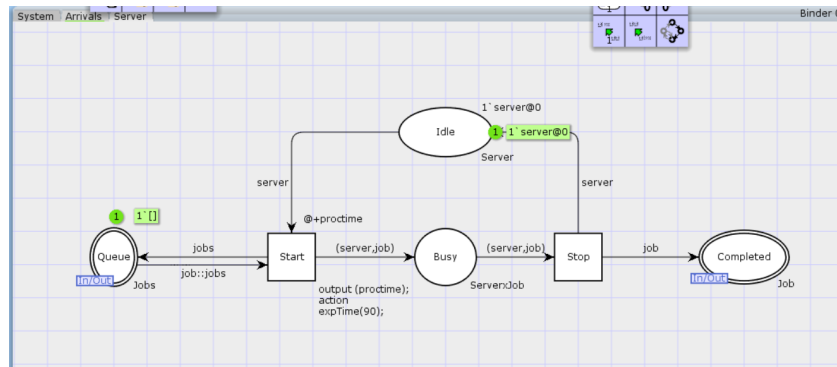


Рис. 2.7: Параметры элементов обработчика заявок системы

2.1 Мониторинг параметров моделируемой системы

Потребуется палитра Monitoring. Выбираем Break Point (точка останова) и устанавливаем её на переход Start. После этого в разделе меню Monitor появится новый подраздел, который назовём Ostanovka. В этом подразделе необходимо внести изменения в функцию Predicate, которая будет выполняться при запуске монитора. Зададим число шагов, через которое будем останавливать мониторинг. Для этого true заменим на `Queue_Delay.count()=200`.

В результате функция примет вид (рис. 2.8):

```

▼ Predicate
fun pred (bindelem) =
let
fun predBindElem (Server'Start (1, {job,jobs,proctime}))
= Queue_Delay.count()=200
| predBindElem _ = false
in
predBindElem bindelem
end

```

Рис. 2.8: Функция Predicate монитора Ostanovka

Необходимо определить конструкцию `Queue_Delay.count()`. С помощью палитры Monitoring выбираем Data Call и устанавливаем на переходе Start. Появившийся

ся в меню монитор называем Queue Delay (без подчеркивания). Функция Observer выполняется тогда, когда функция предикатора выдаёт значение true. По умолчанию функция выдаёт 0 или унарный минус (~1), подчёркивание обозначает произвольный аргумент. Изменим её так, чтобы получить значение задержки в очереди. Для этого необходимо из текущего времени `intTime()` вычесть временную метку `AT`, означающую приход заявки в очередь.

В результате функция примет вид (рис. 2.9):

```
▼ Observer
  fun obs (bindelem) =
  let
  fun obsBindElem (Server'Start (1, {job, jobs, proctime})) = 0
  | obsBindElem _ = ~1
  in
  obsBindElem bindelem
  end
```

Рис. 2.9: Функция Observer монитора Queue Delay

После запуска программы на выполнение в каталоге с кодом программы появится файл `Queue_Delay.log` (рис. 2.10), содержащий в первой колонке — значение задержки очереди, во второй — счётчик, в третьей — шаг, в четвёртой — время.

```
#data counter step time
0 1 3 3
0 2 6 78
103 3 9 193
17 4 12 317
0 5 15 331
224 6 19 661
214 7 23 752
87 8 25 785
107 9 28 857
174 10 31 978
14 11 33 985
182 12 40 1190
301 13 42 1311
373 14 44 1389
414 15 48 1451
```

Рис. 2.10: Файл Queue_Delay.log

С помощью gnuplot можно построить график значений задержки в очереди (рис. 2.11), выбрав по оси x время, а по оси y — значения задержки:

```
#!/usr/bin/gnuplot -persist
# задаём текстовую кодировку,
# тип терминала, тип и размер шрифта

set encoding utf8
set term pngcairo font "Helvetica,9"

# задаём выходной файл графика
```

```
set out 'window_1.png'
plot "Queue_Delay.log" using ($4):($1) with lines
```

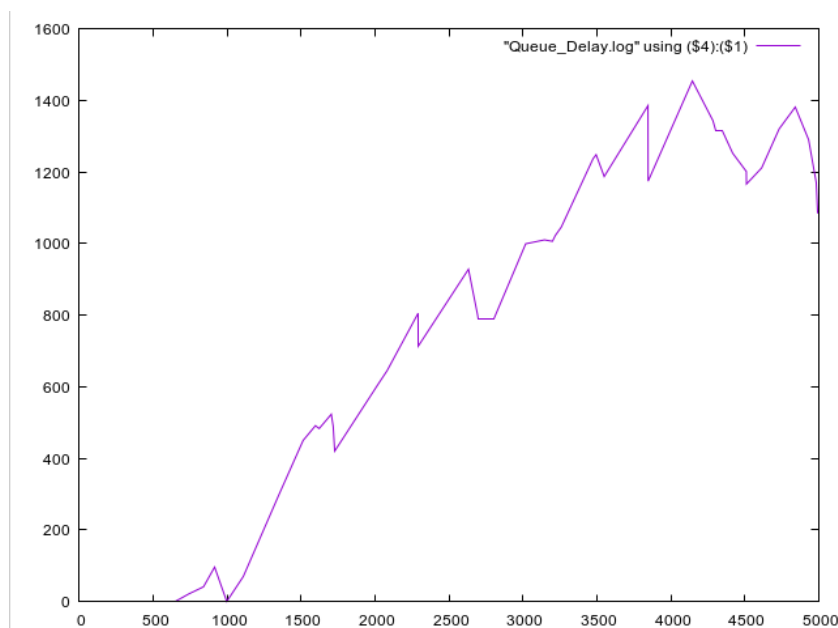


Рис. 2.11: График изменения задержки в очереди

Посчитаем задержку в действительных значениях. С помощью палитры Monitoring выбираем Data Call и устанавливаем на переходе Start. Появившийся в меню монитор называем Queue Delay Real. Функцию Observer изменим следующим образом(рис. 2.12):

```
▼ Observer
  fun obs (bindelem) =
  let
  fun obsBindElem (Server'Start (1, {job, jobs, proctime}))
  = Real.fromInt(intTime() - (#AT job))
  | obsBindElem _ = ~1.0
  in
  obsBindElem bindelem
  end
  ▶ Init function
```

Рис. 2.12: Функция Observer монитора Queue Delay Real

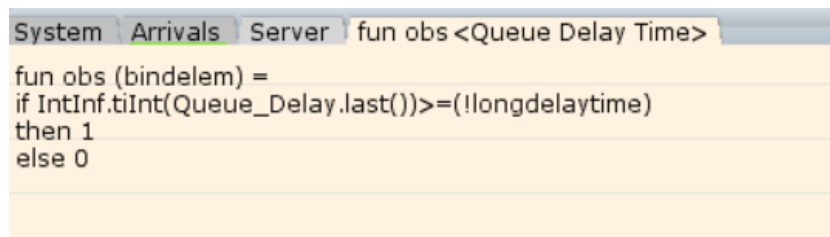
По сравнению с предыдущим описанием функции добавлено преобразование значения функции из целого в действительное, при этом obsBindElem _ принимает значение ~1.0. После запуска программы на выполнение в каталоге с кодом

программы появится файл Queue_Delay_Real.log с содержимым, аналогичным содержимому файла Queue_Delay.log, но значения задержки имеют действительный тип (рис. 2.13):

```
#data counter step time
0.000000 1 3 69
89.000000 2 6 176
188.000000 3 10 386
173.000000 4 13 490
102.000000 5 15 584
0.000000 6 18 692
0.000000 7 21 815
0.000000 8 24 933
0.000000 9 27 996
0.000000 10 30 1039
6.000000 11 33 1067
3.000000 12 36 1090
184.000000 13 41 1400
184.000000 14 43 1422
192.000000 15 46 1554
171.000000 16 50 1723
```

Рис. 2.13: Содержимое Queue_Delay_Real.log

Посчитаем, сколько раз задержка превысила заданное значение. С помощью палитры Monitoring выбираем Data Call и устанавливаем на переходе Start. Монитор называем Long Delay Time. Функцию Observer изменим следующим образом (рис. 2.14):



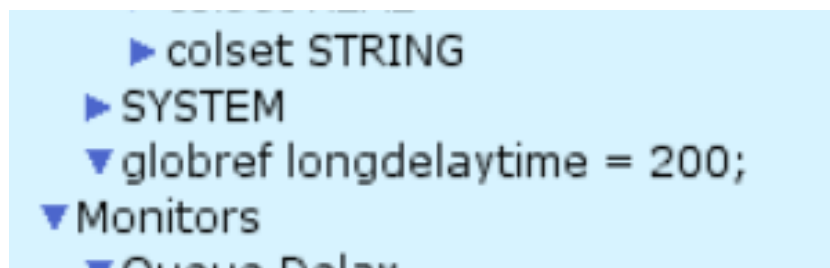
```

System Arrivals Server fun obs <Queue Delay Time>
fun obs (bindelem) =
if IntInf.tiInt(Queue_Delay.last())>=(!longdelaytime)
then 1
else 0

```

Рис. 2.14: Функция Observer монитора Long Delay Time

При этом необходимо в декларациях задать глобальную переменную (в форме ссылки на число 200): longdelaytime (рис. 2.15).



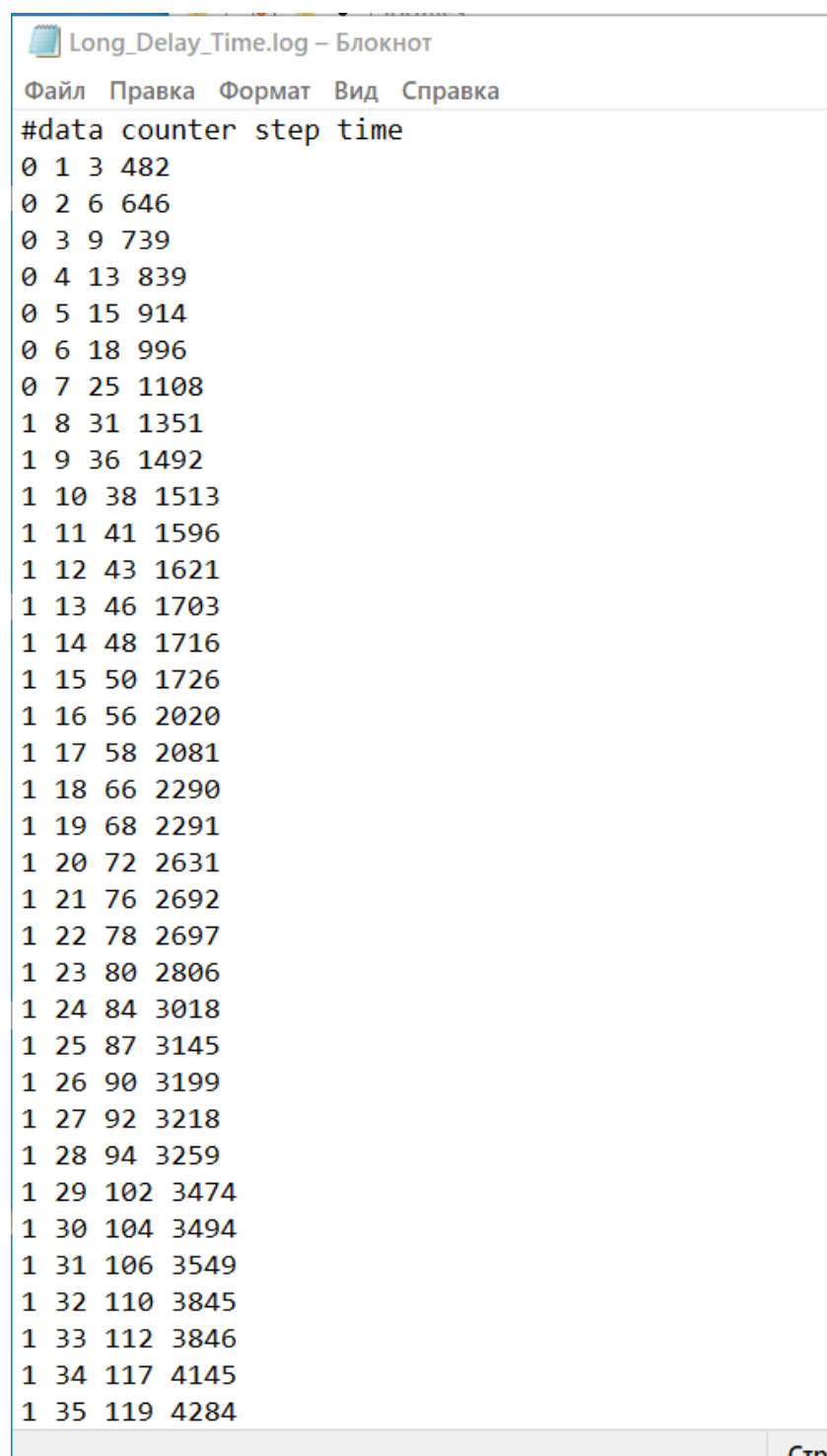
```

► colset STRING
► SYSTEM
▼ globref longdelaytime = 200;
▼ Monitors
▼ Queue Delay

```

Рис. 2.15: Определение longdelaytime в декларациях

После запуска программы на выполнение в каталоге с кодом программы появится файл Long_Delay_Time.log (рис. 2.16)



```
#data counter step time
0 1 3 482
0 2 6 646
0 3 9 739
0 4 13 839
0 5 15 914
0 6 18 996
0 7 25 1108
1 8 31 1351
1 9 36 1492
1 10 38 1513
1 11 41 1596
1 12 43 1621
1 13 46 1703
1 14 48 1716
1 15 50 1726
1 16 56 2020
1 17 58 2081
1 18 66 2290
1 19 68 2291
1 20 72 2631
1 21 76 2692
1 22 78 2697
1 23 80 2806
1 24 84 3018
1 25 87 3145
1 26 90 3199
1 27 92 3218
1 28 94 3259
1 29 102 3474
1 30 104 3494
1 31 106 3549
1 32 110 3845
1 33 112 3846
1 34 117 4145
1 35 119 4284
```

Рис. 2.16: Содержимое Long_Delay_Time.log

С помощью gnuplot можно построить график (рис. 2.17), демонстрирующий, в какие периоды времени значения задержки в очереди превышали заданное

значение 200.

```
#!/usr/bin/gnuplot -persist
# задаём текстовую кодировку,
# тип терминала, тип и размер шрифта

set encoding utf8
set term pngcairo font "Helvetica,9"

# задаём выходной файл графика
set out 'window_1.png'
set style line 2
plot [0:] [0:1.2] "Long_Delay_Time.log" using ($4):($1) with lines
```

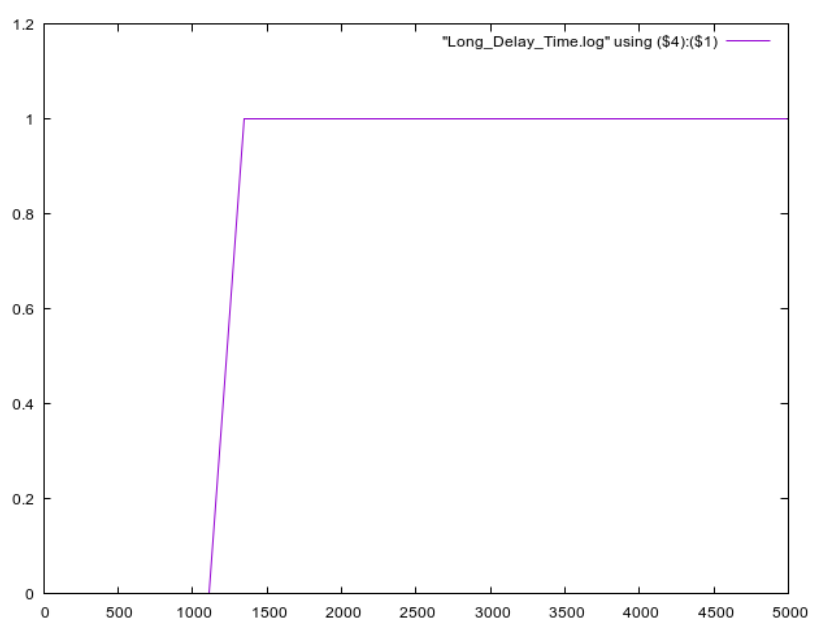


Рис. 2.17: Периоды времени, когда значения задержки в очереди превышали заданное значение

3 Выводы

В процессе выполнения данной лабораторной работы я реализовал модель системы массового обслуживания $M|M|1$ в CPN Tools.