Лабораторная работа 11. Модель системы массового обслуживания $M|M|1$

11.1. 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.

11.2. Построение модели с помощью CPNTools

1. Будем использовать три отдельных листа: на первом листе опишем граф системы (рис. 11.1), на втором — генератор заявок (рис. 11.2), на третьем — сервер обработки заявок (рис. 11.3).

1.1. 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).

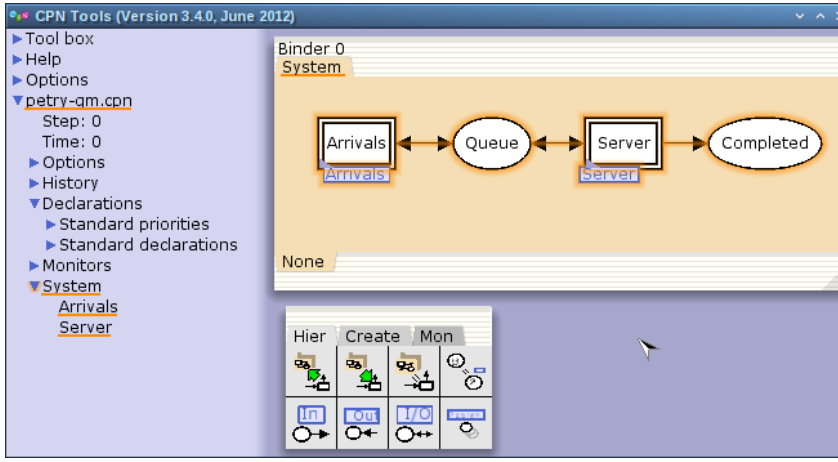

Рис. 11.1. Граф сети системы обработки заявок в очереди

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.

1.2. 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).

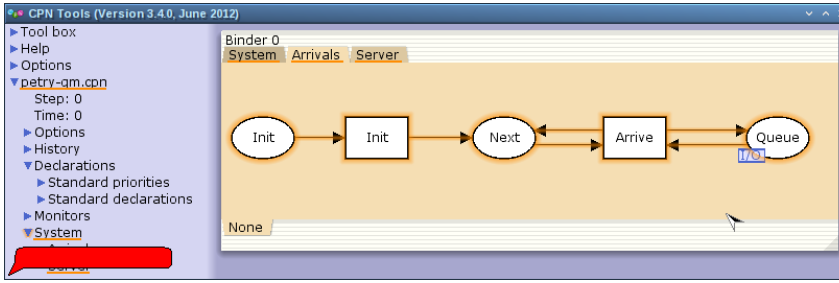

Рис. 11.2. Граф генератора заявок системы

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

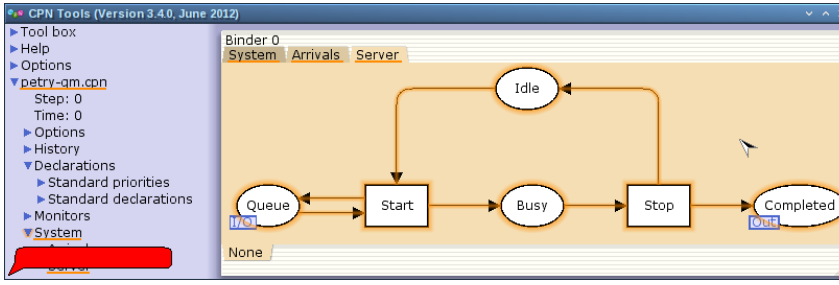

Рис. 11.3. Граф процесса обработки заявок на сервере системы

2. Зададим декларации системы.

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 — `A` и `B`;
– 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.

```
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;
```

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

```
var proctime : INT;
var job: Job;
var jobs: Jobs;
```

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`).

```
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()}
```

3. Зададим параметры модели на графах сети.

3.1. На листе `System` (рис. 11.4):
- у позиции `Queue` множество цветов фишек — `Jobs`; начальная маркировка `1`[]` определяет, что изначально очередь пуста.
- у позиции `Completed` множество цветов фишек — `Job`.

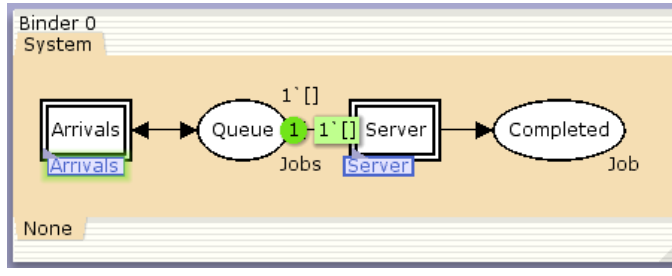

Рис. 11.4. Параметры элементов основного графа системы обработки заявок в очереди

3.2. На листе `Arrivals` (рис. 11.5):
- у позиции `Init`: множество цветов фишек — `UNIT`; начальная маркировка `1`()@0` определяет, что поступление заявок в систему начинается с нулевого момента времени;
- у позиции `Next`: множество цветов фишек — `UNIT`;

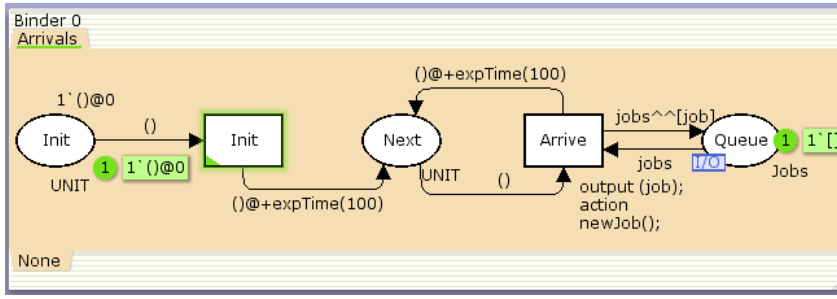

Рис. 11.5. Параметры элементов генератора заявок системы

– на дуге от позиции `Init` к переходу `Init` выражение `()` задаёт генерацию заявок;
– на дуге от переходов `Init` и `Arrive` к позиции `Next` выражение `()@+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 `jobs^^[job]` задает поступление заявки в очередь;
– на дуге от позиции `Queue` к переходу `Arrive` выражение `jobs` задаёт обратную связь.

3.3. На листе `Server` (рис. 11.6):

– у позиции `Busy`: множество цветов фишек — `Server`, начальное значение маркировки — `1`server@0` определяет, что изначально на сервере нет заявок на обслуживание;
– у позиции `Idle`: множество цветов фишек — `ServerxJob`;
– 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.

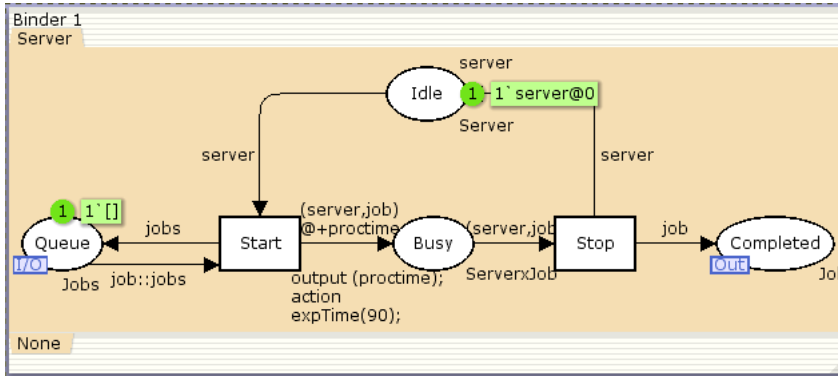

Рис. 11.6. Параметры элементов обработчика заявок системы

11.3. Мониторинг параметров моделируемой системы

Цель: мониторинг параметров очереди системы $M|M|1$.

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:

```
fun pred (bindelem) =
let
  fun predBindElem (Server'Start (1, {job,jobs,proctime}))
    = true
    | predBindElem _ = false
in
  predBindElem bindelem
end
```

Изначально, когда функция начинает работать, она возвращает значение `true`, в противном случае — `false`. В теле функции вызывается процедура `predBindElem`, которую определяем в предварительных декларациях.

Зададим число шагов, через которое будем останавливать мониторинг. Для этого `true` заменим на `Queue_Delay.count()=200` (рис. 11.8):

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

Необходимо определить конструкцию `Queue_Delay.count()`. С помощью палитры *Monitoring* выбираем `Data Call` и устанавливаем на переходе `Start`. Появившийся в меню монитор называем `Queue Delay` (без подчеркивания).

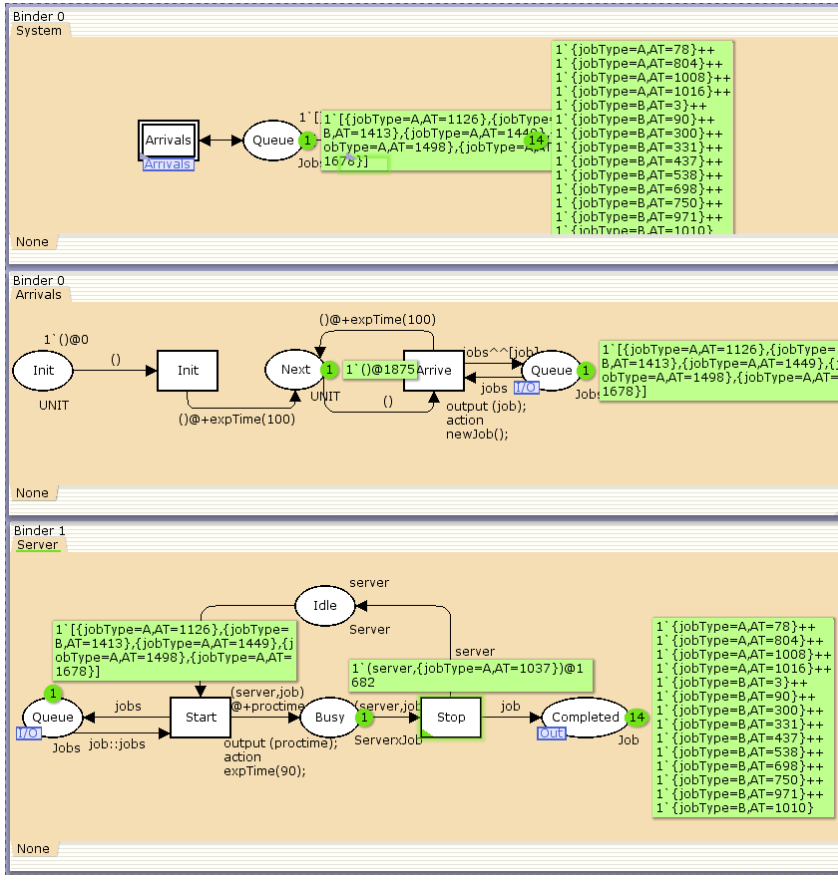

Рис. 11.7. Запуск системы обработки заявок в очереди

Функция `Obsever` выполняется тогда, когда функция предикатора выдаёт значение `true`. По умолчанию функция выдаёт 0 или унарный минус (`~1`), подчёркивание обозначает произвольный аргумент.

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

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 (рис. 11.9):

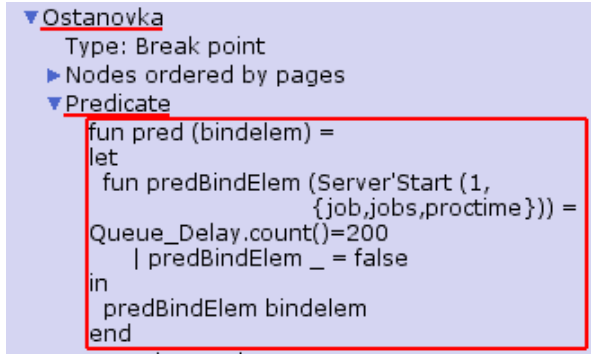

Рис. 11.8. Функция Predicate монитора Ostanovka

```
fun obs (bindelem) =
let
  fun obsBindElem (Server'Start (1, {job, jobs, proctime}))
    = (intTime() - (#AT job))
    | obsBindElem _ = ~1
in
  obsBindElem bindelem
end
```

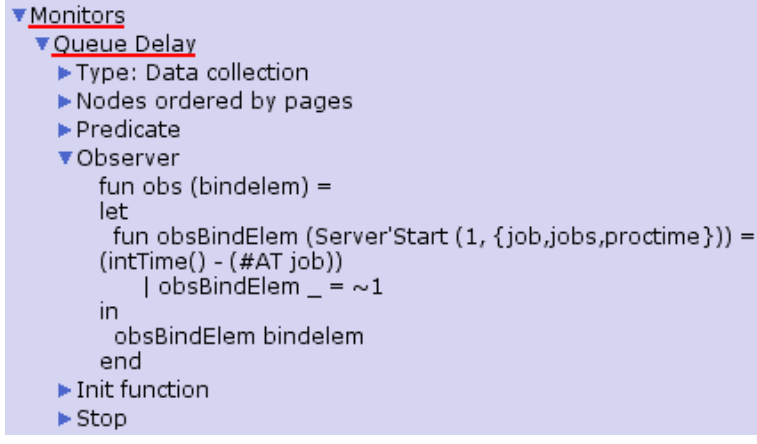

Рис. 11.9. Функция Observer монитора Queue Delay

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

gnuplot можно построить график значений задержки в очереди (рис. 11.10), выбрав по оси x время, а по оси y — значения задержки:

```
gnuplot
plot "Queue_Delay.log" using ($4):($1) with lines
quit
```

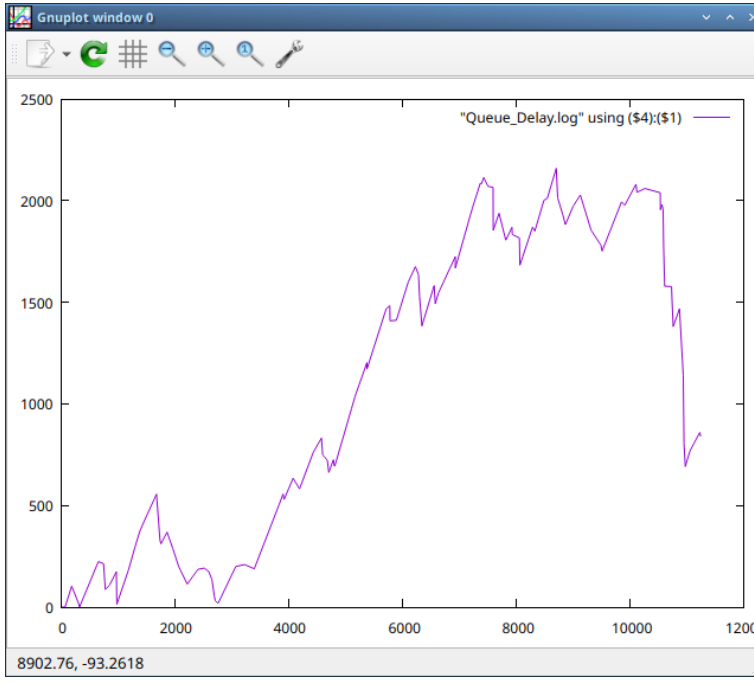

Рис. 11.10. График изменения задержки в очереди

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. 11.11):

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

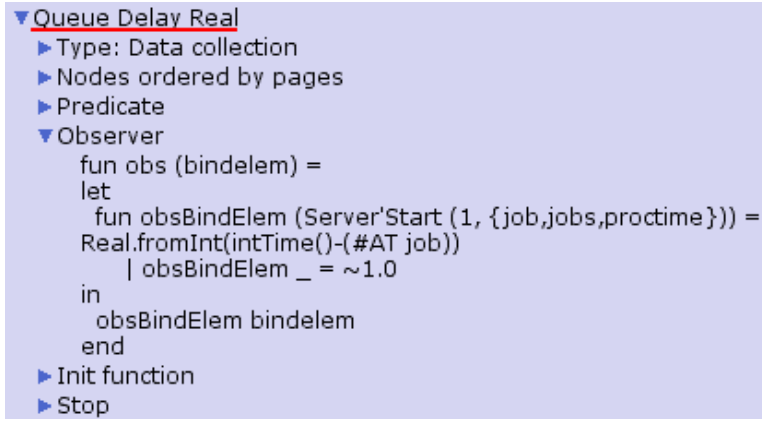

Рис. 11.11. Функция Observer монитора Queue Delay Real

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.

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. 11.12):

```
fun obs (bindelem) =
  if IntInf.tiInt(Queue_Delay.last())>=(!longdelaytime)
  then 1
  else 0
```

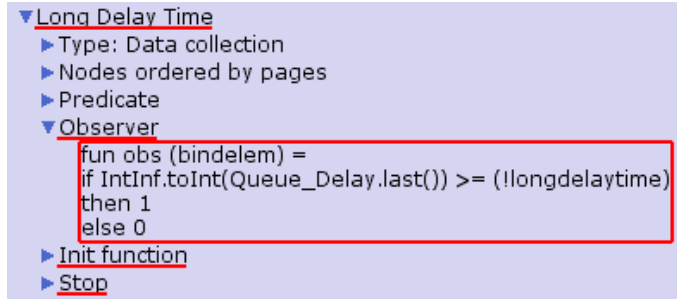

Рис. 11.12. Функция Observer монитора Long Delay Time

Если значение монитора `Queue Delay` превысит некоторое заданное значение, то функция выдаст 1, в противном случае — 0. Восклицательный знак означает разыменование ссылки.

При этом необходимо в декларациях (рис. 11.13) задать глобальную переменную (в форме ссылки на число 200): `longdelaytime`:

```
globref longdelaytime = 200;
```

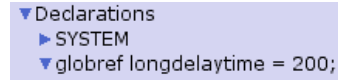

Рис. 11.13. Определение longdelaytime в декларациях

С помощью gnuplot можно построить график (рис. 11.14), демонстрирующий, в какие периоды времени значения задержки в очереди превышали заданное значение 200:

```
gnuplot
plot [0:][0:1.2] "Long_Delay_Time.log" using ($4):($1) with lines
quit
```

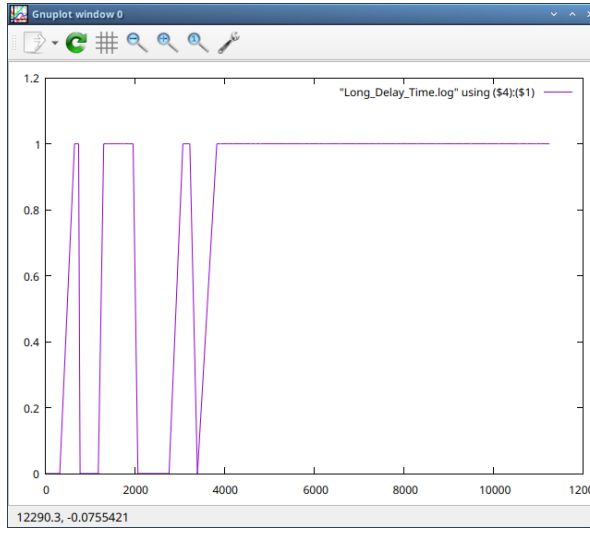

Рис. 11.14. Периоды времени, когда значения задержки в очереди превышали заданное значение