

Лабораторная работа №11

Задание для самостоятельного выполнения

Игнатенкова Варвара Николаевна

Содержание

1	Цель работы.....	1
	Реализовать модель $M M 1$ в CPN tools.	1
2	Задание	1
	- Реализовать в CPN Tools модель системы массового обслуживания $M M 1$	1
	- Настроить мониторинг параметров моделируемой системы и нарисовать графики очереди.	1
3	Выполнение лабораторной работы.....	1
4	Выводы	11

1 Цель работы

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

2 Задание

- Реализовать в CPN Tools модель системы массового обслуживания $M|M|1$.

- Настроить мониторинг параметров моделируемой системы и нарисовать графики очереди.

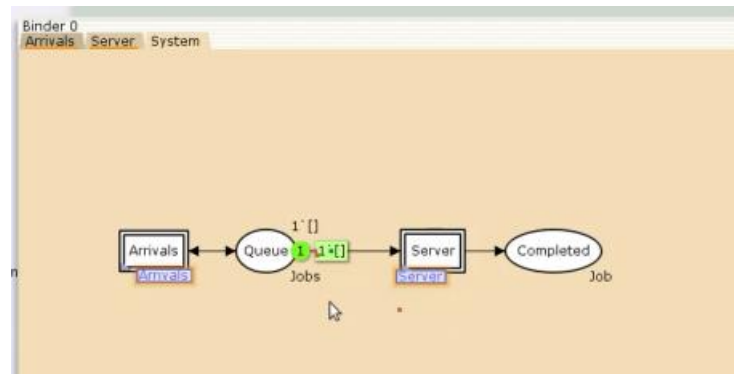
3 Выполнение лабораторной работы

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 на втором — генератор заявок, на третьем — сервер обработки заявок.

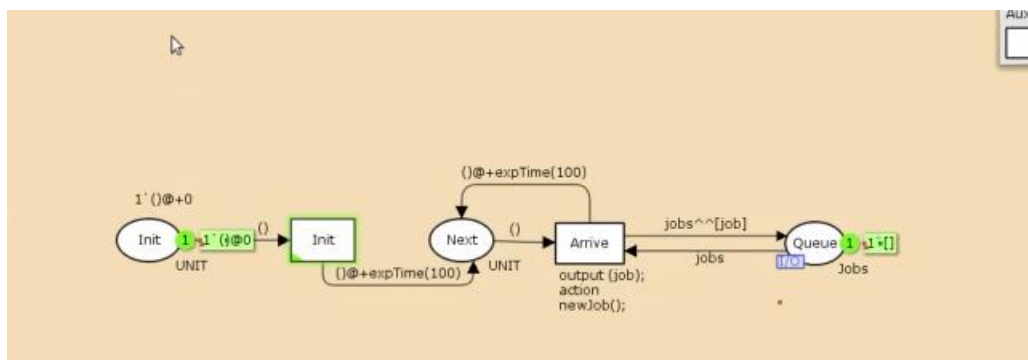
Сеть имеет 2 позиции (очередь — `Queue`, обслуженные заявки — `Completed`) и два перехода (генерировать заявку — `Arrivals`, передать заявку на обработку серверу — `Server`). Переходы имеют сложную иерархическую структуру, задаваемую на отдельных листах модели (с помощью соответствующего инструмента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



Hierarchy).

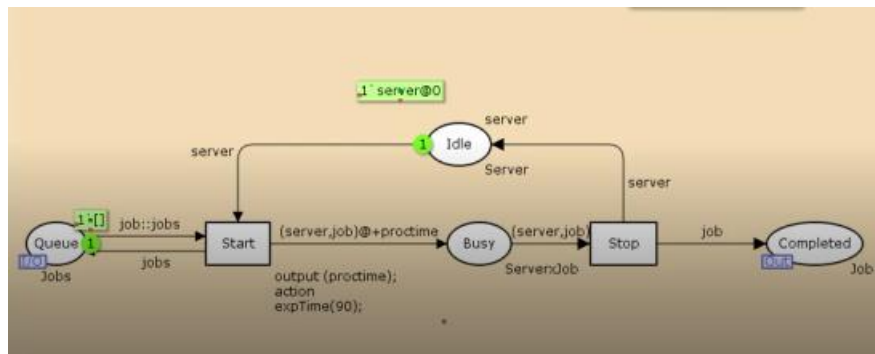
Между переходом `Arrivals` и позицией `Queue`, а также между позицией `Queue` и переходом `Server` установлена дуплексная связь. Между переходом `Server` и позицией `Completed` — односторонняя связь.

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).



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).



Граф процесса обработки заявок на сервере системы

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 — 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 declarations
    ▼ 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 ()};
      ▼ globref longdelaytime = 200;
```

Задание деклараций системы

Зададим параметры модели на графах сети.

На листе `System`:

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

На листе Arrivals:

- 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`;
- 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.

На листе Server:

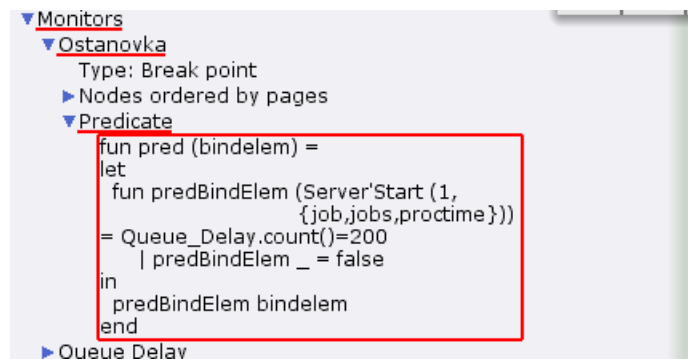
- у позиции `Busy`: множество цветов фишек — `Server`, начальное значение маркировки — ``1`server@0`` определяет, что изначально на сервере нет заявок на обслуживание;
- у позиции `Idle`: множество цветов фишек — ``ServerxJob``;
- переход `Start` имеет сегмент кода ``output (proctime); action expTime(90);`` 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.

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:



Функция Predicate монитора Ostanovka

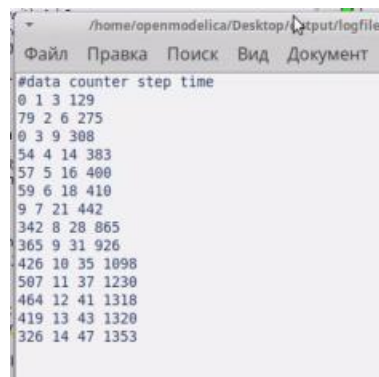
Необходимо определить конструкцию Queue_Delay.count(). С помощью палитры Monitoring выбираем Data Call и устанавливаем на переходе Start. Появившийся в меню монитор называем Queue Delay (без подчеркивания). Функция Observer выполняется тогда, когда функция предикатора выдаёт значение true. По умолчанию 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:

```
▼ Queue Delay
  ▶ Type: Data collection
  ▶ Nodes ordered by pages
  ▶ Predicate
  ▼ Observer
    fun obs (bindelem) =
      let
        fun obsBindElem (Server'Start (1, {job,jobs,proctime}))
          = (intTime() - (#AT job))
          | obsBindElem _ = ~1
        in
          obsBindElem bindelem
        end
      in
        Init function
      Stop
```

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.



```
#data counter step time
0 1 3 129
79 2 6 275
0 3 9 308
54 4 14 383
57 5 16 400
59 6 18 410
9 7 21 442
342 8 28 865
365 9 31 926
426 10 35 1098
507 11 37 1230
464 12 41 1318
419 13 43 1320
326 14 47 1353
```

Файл *Queue_Delay.log*

С помощью gnuplot можно построить график значений задержки в очереди.

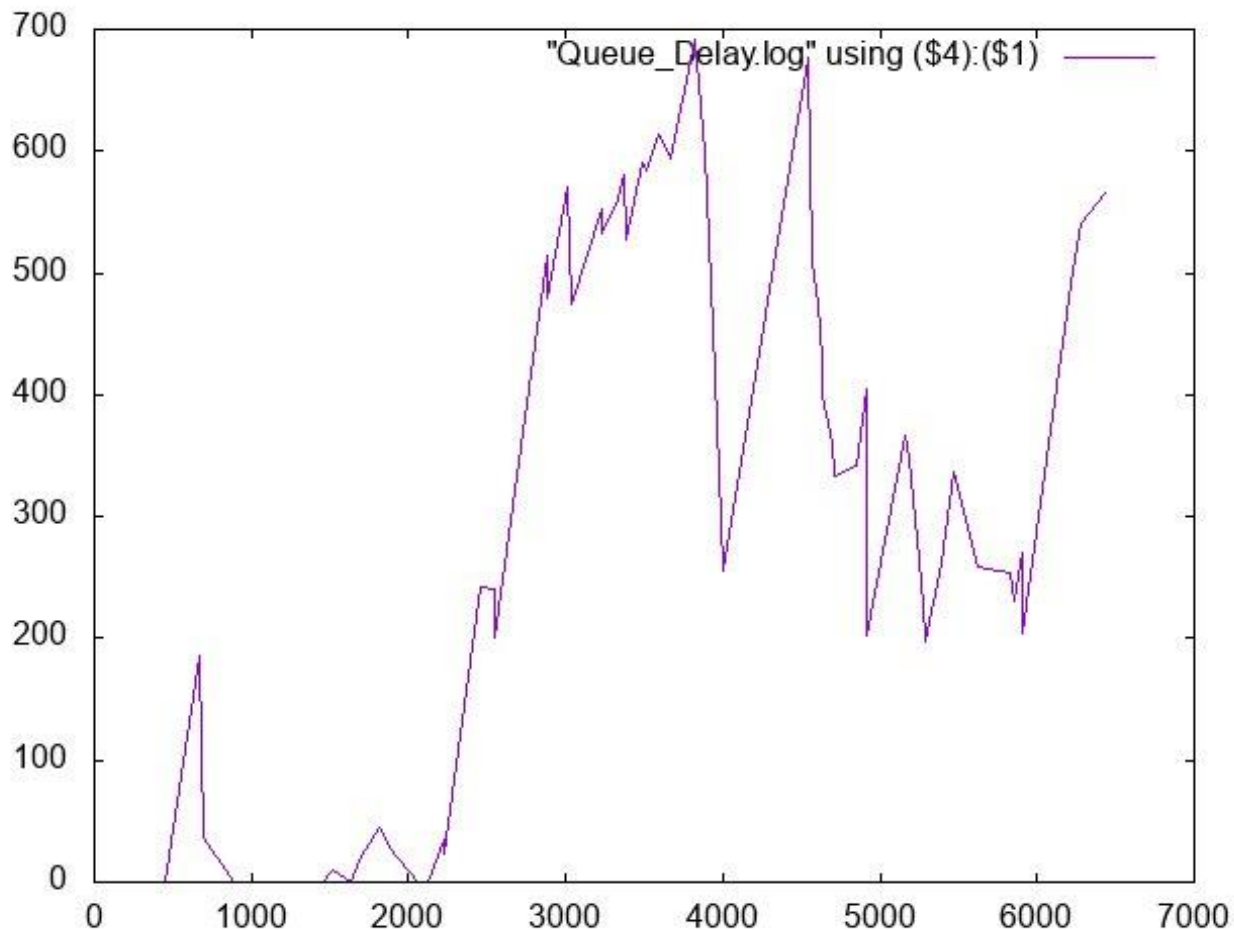


График изменения задержки в очереди

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:

```

    ▶ Stop
    ▼ Queue Delay Real
    ▶ Type: Data collection
    ▶ Nodes ordered by pages
    ▶ Predicate
    ▼ 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
    ▶ Stop
    ▼ Long Delay Time
```

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:

#data	counter	step	time
591.000000	1	102	3483
584.000000	2	104	3507
614.000000	3	106	3589
594.000000	4	108	3670
691.000000	5	111	3827
599.000000	6	114	3887
454.000000	7	116	3934
255.000000	8	118	4006
676.000000	9	128	4538
510.000000	10	130	4567
475.000000	11	132	4601
429.000000	12	134	4635
400.000000	13	136	4637
360.000000	14	138	4701
334.000000	15	140	4708
341.000000	16	145	4847
404.000000	17	147	4911
202.000000	18	149	4912

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:

```

▼ Long Delay Time
  ► Type: Data collection
  ► Nodes ordered by pages
  ► Predicate
  ▼ Observer
    fun obs (bindelem) =
      if IntInf.toInt(Queue_Delay.last()) >= (!longdelaytime)
      then 1
      else 0
  ► Init function
  ► Stop
Init

```

Функция Observer монитора Long Delay Time

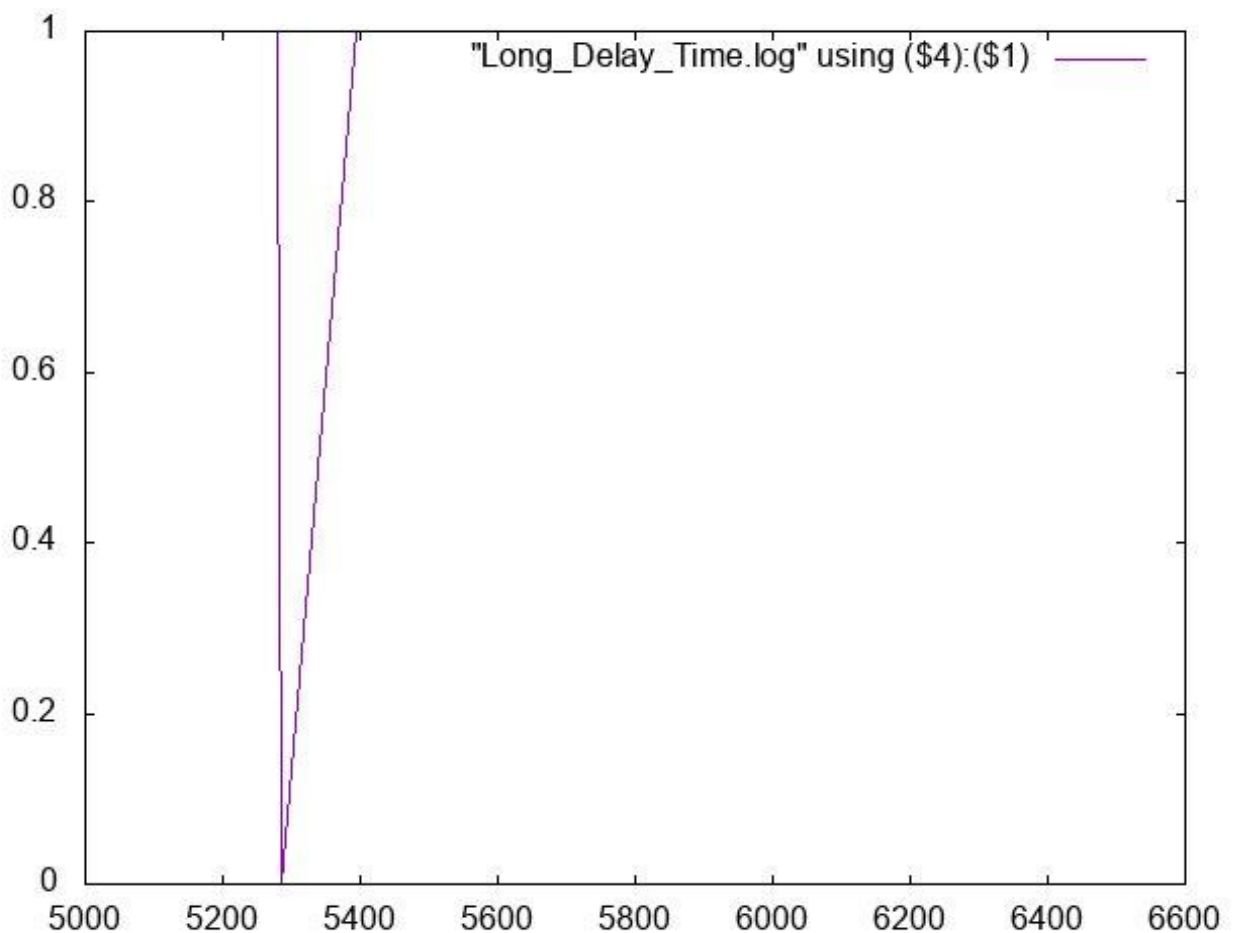
При этом необходимо в декларациях задать глобальную переменную (в форме ссылки на число 200): `longdelaytime`.

Файл	Правка	Поиск	Вид	Документ	Справка
#data	counter	step	time		
1	1	156	5162		
1	2	158	5186		
1	3	160	5255		
1	4	162	5278		
0	5	164	5284		
1	6	167	5395		
1	7	169	5471		
1	8	173	5620		
1	9	179	5824		
1	10	181	5850		
1	11	185	5902		
1	12	187	5911		
1	13	192	6216		
1	14	194	6273		
1	15	200	6439		

Содержимое *Long_Delay_Time.log*

После запуска программы на выполнение в каталоге с кодом программы появится файл *Long_Delay_Time.log*.

С помощью *gnuplot* можно построить график, демонстрирующий, в какие периоды времени значения задержки в очереди превышали заданное значение 200.



Периоды времени, когда значения задержки в очереди превышали заданное значение

4 Выводы

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