Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра ВС

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Моделирование»

**Моделирование дискретно-событийных систем**

|  |  |  |  |
| --- | --- | --- | --- |
| Выполнил студент | Марков В.А. | | |
| Защищена |  | Оценка |  |
|  | | | |
| Выполнил студент | Мосин И.А. | | |
| Защищена |  | Оценка |  |

|  |  |
| --- | --- |
| Группы | МГ-165 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу принял |  | доцент д.т.н. А.С. Родионов |

Новсибирск 2017 г.

**СОДЕРЖАНИЕ**

1. Цель курсового проекта 3
2. Дискретно-событийная система 4
3. Календарь событий 5
4. Эксперименты 7

ЗАКЛЮЧЕНИЕ 8

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 9

ПРИЛОЖЕНИЕ 10

1. **Цель курсового проекта**

Программно реализовать моделирование дискретно-событийных систем. Сравнить среднее время пребывания заявки в системе, а также среднюю длину очередей.

Дана дискретно-событийная система, состоящая из двух линий обработки задач (см. рис. 1). В первую очередь поступает заявка с интенсивностью *τ*1, распределенной по экспоненциальному закону, в вторую с интенсивностью *τ*2, также распределенной по экспоненциальному закону.

Далее эти заявки поступают на обработку, в том случае если рабочий 1 и 2 не заняты соответственно. Рабочие берут заявку в первом случае – с интенсивностью, распределенной по экспоненциальному закону, для второго случая – равномерно. Далее после того как заявка обработается первым рабочим она поступает во вторую очередь и наоборот. Заявка, обработанная обоими рабочими, покидает систему.



Рисунок 1 – Исходный граф событий

1. Приход новой заявки

2. Приход новой заявки

3. Взятие заявки на обработку

4. Взятие заявки на обработку

5. Окончание обслуживания

6. Окончание обслуживания

(I) – рабочий 1 свободен

(II) – рабочий 2 свободен

(III) – рабочий 1 свободен

(IV) – рабочий 2 свободен

(V) – заявка обрабатывалась однажды

(VI) – заявка обрабатывалась однажды

1. **Дискретно-событийное моделирование**

Дискретно-событийное моделирование (discrete event modeling) – подход к моделированию, предлагающий абстрагироваться от непрерывной природы событий и рассматривать процессы, протекающие в моделируемой системе, как последовательность отдельных событий. Основан Джеффри Гордоном в 1960-х годах.

В дискретно-событийном моделировании выделяются состояния системы и описываются действия, которые переводят ее из одного состояния в другое. Состояние системы может меняться только в моменты свершения событий. Так как состояние системы не изменяется между этими моментами, полный динамический портрет состояний системы может быть получен путем продвижения имитационного времени от одного события к другому. При дискретной имитации значения зависимых переменных изменяются дискретно в моменты совершения событий и, как правило, не изменяются в промежутках между моментами совершения событий.

Термин "дискретно-событийное моделирование", однако, обычно используется в более узком смысле для обозначения "процессного" моделирования, где динамика системы представляется как последовательность операций над некими сущностями. Эти сущности пассивны, они сами не контролируют свою динамику, но могут обладать определёнными атрибутами, влияющими на процесс их обработки или накапливающими статистику. Процессное моделирование используется на среднем или низком уровне абстракции: каждый объект моделируется индивидуально, как отдельная сущность, но множество деталей "физического уровня" опускается.

Дискретно-событийное моделирование наиболее развито и имеет огромную сферу приложений – от логистики и систем массового обслуживания до транспортных и производственных систем. Этот вид моделирования наиболее подходит для моделирования производственных процессов.

Примеры систем: Extend, Arena, ProModel, Witness, Taylor, Gpss/H-Proof, AutoMod, QUEST, SIMFACTORY II.5, SIMPLE++, eM-Plant, и др [1].

1. **Календарь событий**

При дискретной имитации состояние системы может меняться только в моменты свершения событий. Так как состояние системы не изменяется между этими моментами, полный динамический портрет состояний системы может быть получен путем продвижения имитационного времени от одного события к другому.

В большинстве языков дискретной имитации используется механизм продвижения времени, основанный на поиске следующего ближайшего события.

Функционирование дискретной имитационной модели можно задать следующим образом: определяя изменения состояния системы, происходящие в момент свершения событий; описывая действия, в которых принимают участие элементы системы, или процесс, через который проходят элементы.

Событие происходит в тот момент, когда принимается решение о начале или окончании действия. Процесс – это ориентированная во времени последовательность событий, которая может состоять из нескольких действий. Эти представления лежат в основе трех альтернативных методологических подходов к построению дискретных имитационных моделей, называемых обычно событийным подходом, подходом сканирования активностей и процессно-ориентированным подходом.

Для реализации событийного подхода требуется реализовать календарь событий. Ниже предоставлен псевдокодов минимального функционала календаря событий [2].

|  |  |
| --- | --- |
| **Алгоритм работы календаря событий** | |
| 1 | **function** Simulate(*calendar*, *currentTime*, *modelingTime*) |
| 2 | **while** NotEmpty(*calendar*) **and** *currentTime* < *modelTime* |
| 3 | *event* = CalendarPop(*calendar*) */\* Извлекаем из календаря \*/* |
| 4 | *currentTime* = *currentEvent*.*time* */\* Меняем модельное время \*/* |
| 5 | *event*.Procedure() */\* Запускаем обработчик \*/* |
| 6 | Free(*event*) */\* Освобождаем память \*/* |
| 7 | **end while** |
| 8 | **end function** |

|  |  |
| --- | --- |
| **Алгоритм обработчика события** | |
| 1 | **function** EventHandler(*currentTime*, *lambda*) |
| 2 | */\* Код обработчика \*/* |
| 3 | Schedule(EventHandler, *currentTime* + GetRandExp(*lambda*)) |
| 4 | **end function** |

|  |  |
| --- | --- |
| **Алгоритм планировщика календаря событий** | |
| 1 | **function** Schedule(*FuncPointer*, *time*, *calendar*) |
| 2 | *event* = Malloc() |
| 3 | *event*.*time* = *time* |
| 4 | *event*.*proc* = *FuncPointer* |
| 5 | СalendarPush(*calendar*, *event*) */\* Помещаем в календарь \*/* |
| 6 | **end function** |

|  |  |
| --- | --- |
| **Описание структуры события на языке Си** | |
| 1 | **struct** *event* |
| 2 | { |
| 3 | **double** *time*; */\* Время \*/* |
| 4 | **void** (\**proc*)(*/\* args \*/*); */\* Указатель на функцию обработчика \*/* |
| 5 | **struct** *event* \**next*; */\* Указатель на следующий узел \*/* |
| 6 | **struct** *event* \**prev*; */\* Указатель на предыдущий узел \*/* |
| 7 | }; |

1. **Эксперименты**

Необходимо провести экспериментальное исследование и сравнить средний размер входных очередей и среднее время пребывания заявки в системе от заданных входные данных.

Результаты проведения экспериментов вариант1:

* Время моделирования 1000 (ед. мод. времени)
* Интенсивность прихода заявок в очередь 1 *λ*(1.0)
* Интенсивность прихода заявок в очередь 2 *λ*(1.0)
* Интенсивность обработки заявок рабочим1 *λ*(2.1)
* Интенсивность обработки заявок рабочим 2 *λ*(3.0)
* Генерация времени распределена по экспоненциальному закону
* **Средний размер очередей: 458 (заявок)**
* **Среднее время пребывания заявки в системе: 348.14 (ед. мод. времен)**

Вариант2:

* Время моделирования 1000 (ед. мод. времени)
* Интенсивность прихода заявок в очередь 1 *λ*(1.0)
* Интенсивность прихода заявок в очередь 2 *λ*(1.0)
* Интенсивность обработки заявок рабочим1 *λ*(1.0 / 2.1)
* Интенсивность обработки заявок рабочим 2 *λ*(1.0 / 3.0)
* Генерация времени распределена по экспоненциальному закону (только для входных очередей)
* **Средний размер очередей: 424 (заявки)**
* **Среднее время пребывания заявки в системе: 306.72 (ед. мод. времен)**

**Заключение**

В результате выполнения данного курсового проекта была реализована модель дискретно-событийной системы, также было проведено экспериментальное исследование зависимости среднего размера входных очередей и среднего времени пребывания заявки в системе от заданных данных (интенсивность поступления на обработку, интенсивность обработки заявок).

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Дискретно-событийное моделирование URL: https://ru.wikipedia.org/wiki/Дискретно-событийное\_моделирование
2. Методическое пособие к лекциям по курсу Моделирование. СибГУТИ.

**ПРИЛОЖЕНИЕ**

**Calendar.cpp**

#include "calendar.h"

#include <queue>

#include <string>

#include <chrono>

#include <random>

#include <iostream>

static double currentTime = 0.0;

static struct event \*calendar = NULL;

static std::queue<Task> queue1;

static std::queue<Task> queue2;

double getRandExp(double lambda)

{

// Obtain a time-based seed

unsigned seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

// Initialize random generator

std::default\_random\_engine generator(seed);

// Initialize space

std::uniform\_real\_distribution<double> distribution(0.0, 1.0);

return -(std::log(distribution(generator)) / lambda);

}

void calendarPush(struct event \*newEvent)

{

auto \*event = calendar;

if (!calendar)

{

newEvent->next = NULL;

newEvent->prev = NULL;

calendar = newEvent;

return;

}

while (event)

{

if (event->time > newEvent->time)

{

newEvent->next = event;

newEvent->prev = event->prev;

if (event->prev)

{

event->prev->next = newEvent;

}

else

{

calendar = newEvent;

}

event->prev = newEvent;

return;

}

if (!event->next)

{

newEvent->next = NULL;

newEvent->prev = event;

event->next = newEvent;

return;

}

event = event->next;

}

}

struct event \*calendarPop()

{

struct event \*event = calendar;

event->next->prev = NULL;

calendar = event->next;

return event;

}

void schedule(void (\*f)(Statistic &s, int t, double l), double time)

{

auto \*newEvent = new struct event;

if (!newEvent)

{

throw std::string("Memory allocation error!"); // Throw exeption

}

newEvent->time = time;

newEvent->proc = f;

calendarPush(newEvent);

}

int simulate(Statistic &statistic,

int testNum,

double modelTime,

std::vector<double> lambda)

{

auto periodicTime = 0.0;

while (calendar != NULL && currentTime <= modelTime)

{

auto currentEvent = calendarPop();

// Changing time to current event time

currentTime = currentEvent->time;

if (currentEvent->proc == E1)

currentEvent->proc(statistic, testNum, lambda[0]);

else if (currentEvent->proc == E2)

currentEvent->proc(statistic, testNum, lambda[1]);

else if (currentEvent->proc == E3)

currentEvent->proc(statistic, testNum, lambda[2]);

else if (currentEvent->proc == E4)

currentEvent->proc(statistic, testNum, lambda[3]);

delete currentEvent;

periodicTime += currentTime;

if (periodicTime > 1.0)

{

periodicTime = 0.0;

statistic.computeLengths(queue1, queue2);

}

}

if (!calendar)

{

std::cout << "[" << \_\_FUNCTION\_\_ << "]"

<< " Monitor: ending simultation (calendar is empty)"

<< std::endl;

return CALENDAR\_EMPTY;

}

std::cout << "[" << \_\_FUNCTION\_\_ << "]"

<< " Monitor: ending simultation (modeling time expired)"

<< std::endl;

return TIME\_EXPIRED;

}

void E1(Statistic &statistic, int testNum, double lambda)

{

queue1.push(Task(currentTime, -1.0, true));

schedule(E1, currentTime + getRandExp(lambda));

}

void E2(Statistic &statistic, int testNum, double lambda)

{

queue2.push(Task(currentTime, -1.0, true));

schedule(E2, currentTime + getRandExp(lambda));

}

void E3(Statistic &statistic, int testNum, double lambda)

{

static Task localTask;

static bool isFree = true;

if (isFree)

{

if (!queue1.empty())

{

localTask = queue1.front();

queue1.pop();

isFree = false;

}

}

else

{

if (localTask.first)

{

localTask.first = false;

queue2.push(localTask);

}

else

{

localTask.end = currentTime;

statistic.pushTask(localTask);

}

isFree = true;

}

if (testNum == 1)

{

schedule(E3, currentTime + getRandExp(lambda));

}

else if (testNum == 2)

{

lambda = 1.0 / lambda;

schedule(E3, currentTime + lambda);

}

}

void E4(Statistic &statistic, int testNum, double lambda)

{

static Task localTask;

static bool isFree = true;

if (isFree)

{

if (!queue2.empty())

{

localTask = queue1.front();

queue2.pop();

isFree = false;

}

}

else

{

if (localTask.first)

{

localTask.first = false;

queue1.push(localTask);

}

else

{

localTask.end = currentTime;

statistic.pushTask(localTask);

}

isFree = true;

}

if (testNum == 1)

schedule(E4, currentTime + getRandExp(lambda));

else if (testNum == 2)

{

lambda = 1.0 / lambda;

schedule(E4, currentTime + lambda);

}

}

void resetModelingTime()

{

currentTime = 0.0;

}

void resetQueues()

{

while(!queue1.empty())

queue1.pop();

while(!queue2.empty())

queue2.pop();

}

**Main.cpp**

#include <iostream>

#include <fstream>

#include <string>

#include <vector>

#include "color.h"

#include "calendar.h"

#include "statistic.h"

void test(int testNum, int modelingTime, std::vector<double> lambda)

{

Statistic statistic(testNum, lambda[0], lambda[1], lambda[2], lambda[3]);

double startTime = 0.0;

resetModelingTime();

resetQueues();

// Planing events before start

if (testNum == 1)

{

schedule(E1, startTime + getRandExp(lambda[0]));

schedule(E2, startTime + getRandExp(lambda[1]));

schedule(E3, startTime + getRandExp(lambda[2]));

schedule(E4, startTime + getRandExp(lambda[3]));

}

else if (testNum == 2)

{

schedule(E1, startTime + getRandExp(lambda[0]));

schedule(E2, startTime + getRandExp(lambda[1]));

schedule(E3, startTime + (1.0 / lambda[2]));

schedule(E4, startTime + (1.0 / lambda[3]));

}

// Run actual simlation

simulate(statistic, testNum, modelingTime, lambda);

statistic.showStatistic();

return;

}

int main(int argc, char const \*argv[])

{

try

{

Color::Modifier yellow(Color::FG\_YELLOW);

Color::Modifier def(Color::FG\_DEFAULT);

if (argc != 2)

{

throw std::string("usage " + std::string(argv[0]) + " <file-name>");

}

std::ifstream file(argv[1]);

if (!file.is\_open())

{

throw std::string("Can't open file");

}

/\*

\* Read only four lines form file

\* Floating point format required

\* - line 1 : lambda 1

\* - line 2 : lambda 2

\* - line 3 : lambda 3

\* - line 4 : lambda 4

\*/

std::vector<double> lambdas;

for (auto i = 0; i < 4; ++i)

{

std::string::size\_type sz;

std::string line;

getline(file, line);

std::cout << yellow << "Reading from file : " << line << def << std::endl;

lambdas.push\_back(std::stod(line, &sz));

}

test(1, 1000.0, lambdas); // Running test 1

test(2, 1000.0, lambdas); // Running test 2

}

catch (std::string err)

{

Color::Modifier red(Color::FG\_RED);

Color::Modifier def(Color::FG\_DEFAULT);

std::cerr << red << err << def << std::endl;

return -1;

}

return 0;

}