ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Высшая школа программной инженерии

КУРСОВАЯ РАБОТА

Моделирование и верификация распределительных алгоритмов
по дисциплине «Распределенные алгоритмы»

Студент
3530202/90202 Потапова А. М.

Преподаватель Шошмина И. В.

Санкт-Петербург
2022 г.

ЗАДАНИЕ
НА ВЫПОЛНЕНИЕ КУРСОВОГО ПРОЕКТА (КУРСОВОЙ РАБОТЫ)

студенту
группы *3530202/90202* (номер группы) *Потаповой Алине Михайловне* (фамилия, имя, отчество)

1. Тема проекта (работы) Моделирование и верификация распределительных алгоритмов

2. Срок сдачи студентом законченного проекта (работы) *23.05.2022*

3. Исходные данные к проекту (работе)

4. Содержание пояснительной записки (перечень подлежащих разработке вопросов: введение, основная часть (раскрывается структура основной части), заключение, список использованных источников, приложения).
Введение. Основная часть. Заключение. Список источников. Приложение

Примерный объем пояснительной записки *15-20* страниц машинописного текста

5. Перечень графического материала (с указанием обязательных чертежей и плакатов) *не предоставляется*

6. Консультанты

7. Дата получения задания: «1» апреля 2022 г.

Руководитель (подпись) *Шошмина И.В.* (инициалы, фамилия)

Задание принял к исполнению (подпись) *Потапова А.М.* (инициалы, фамилия)

18.05.2022 (дата)

Содержание

Введение 4
Основная часть 5
Постановка задачи 5
Описание решения 7
Верификация 10
Заключение 12
Список источников 13
Приложение 14

Введение

В рамках данной курсовой работы рассматривается решение двух задач на синхронизацию с мониторами. Задача читателей и писателей заключается в обеспечении согласованного доступа нескольких потоков к разделяемым данным. Задача обедающих философов хороша для моделирования процессов, которые соревнуются за исключительный доступ к ограниченному количеству ресурсов.

Для моделирования распределенной системы и ее верификации используется программное обеспечение SPIN [3].

Основная часть

Постановка задачи

1. Задача «Читатели-писатели»

Algorithm 7.4: Readers and writers with a monitor

```
monitor RW
   integer readers ← 0
   integer writers ← 0
   condition OKtoRead, OKtoWrite

   operation StartRead
      if writers ≠ 0 or not empty(OKtoWrite)
         waitC(OKtoRead)
      readers ← readers + 1
      signalC(OKtoRead)

   operation EndRead
      readers ← readers − 1
      if readers = 0
         signalC(OKtoWrite)

   operation StartWrite
      if writers ≠ 0 or readers ≠ 0
         waitC(OKtoWrite)
      writers ← writers + 1

   operation EndWrite
      writers ← writers − 1
      if empty(OKtoRead)
         then signalC(OKtoWrite)
         else signalC(OKtoRead)
```

reader		writer	
p1	RW.StartRead	q1	RW StartWrite
p2	read the database	q2	write to the database
p3	RW.EndRead	q3	RW.EndWrite

Рисунок 1. Алгоритм решения задачи Читателей и писателей с мониторами.

Рассмотрим алгоритм 7.4, представленный в книге Ben-Ari [1]. Имеем несколько потоков, которые пытаются получить доступ к общему ресурсу.

Таким образом, для решения задачи необходимо синхронизировать действия процессов над ресурсом и обеспечить взаимное исключение соответствующих критических секций. Процессы делятся на два класса:

1. Процессы читатели, которые исключают писателей, но не других читателей
2. Процессы писатели, которые исключает как читателей, так и других писателей

В данной работе я приведу решение, которое позволит обеспечить работу этих двух потоков с условием отсутствия приоритета.

2. Задача «Обедающие философы»

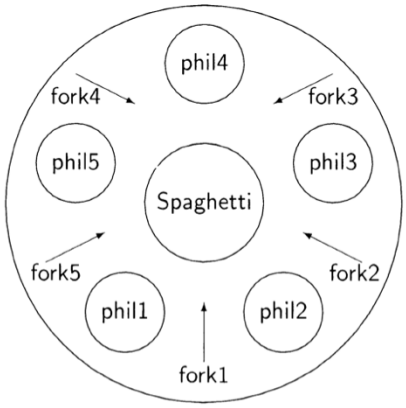

Рисунок 2. Иллюстрация задачи «Обедающие философы».

Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов. Каждый философ может либо есть, либо размышлять.

Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева. Каждый философ может взять ближайшую вилку (если она доступна) или положить — если он уже держит её. Взятие каждой вилки и возвращение её на стол являются раздельными действиями, которые должны выполняться одно за другим. Таким образом, сформулируем основные требования к решению:

1. Философ ест, только если у него есть 2 вилки.
2. Взаимное исключение. Никаких два философа не могут одновременно придерживаться одной и той же вилки.
3. Свобода от голодания. Т. е. все философы будут вечно чередовать прием пищи и размышления.
4. Свобода от взаимной блокировки.

Описание решения

1. Задача «Читатели-писатели» (за основу взят алгоритм 7.4 [1])

Используем монитор со следующими параметрами: *int readers, int writers* (количество читателей и писателей), *Condition OktoRead, Condition OktoWrite* (условные переменные).

```
typedef Monitor {
    int readers;
    int writers;
    Condition OKtoRead;
    Condition OKtoWrite;
}
```

Инициализация монитора с количеством читателей и писателей происходит при помощи функции *initMonitor(M, r, w).*

```
inline initMonitor(M, r, w) {
    M.readers = r;
    M.writers = w;
}
```

Condition в мониторах обычно является очередью, где процессы разблокируются в порядке поступления, но в данной задаче Condition реализован как счетчик и разблокирует тот процесс, который первым пытается прийти.

```
typedef Condition {
    bool gate;
    int waiting;
}
```

Функция *empty(C)* возвращает логическое значение, указывающее, является ли очередь (канал) C пустой или нет.

```
#define emptyC(C) (C.waiting == 0)
```

В функции *waitC(C)* мы блокируем процесс, добавляя его в очередь (канал) командой *C.waiting*++, освобождаем монитор и затем ждем разблокировки процесса.

```
inline waitC(C){
atomic {
    C.waiting++;
    lock = false;
    C.gate;
    C.gate = false;
    C.waiting--;
}
}
```

В функции *signalC*(C) мы проверяем пуста ли очередь (канал). Далее существуют два случая:

- Если он не пуст, то разблокируем процесс и затем ждем возможность зайти в монитор.
- Если он пуст, то операция *signalC(C)* не имеет никакого эффекта.

```
inline signalC(C) {
atomic {
    if
    ::(C.waiting > 0) ->
        C.gate = true;
        !lock;
        lock = true;
    ::else;
    fi;
}
}
```

Также в программе создаем функции *Start Read, Start Write и End Read, End Write* (приложение 1). Каждая из этих функции начинается с попадания в монитор, благодаря функции *enterMonitor()*, и заканчивается выходом из монитора, функцией *leaveMonitor().*

```
inline enterMonitor(){
atomic {
    !lock;
    lock = true;
}
}

inline leaveMonitor(){
    lock = false;
}
```

Вызов функции *enterMonitor()* позволяет процессу убедиться, что он может сделать ход, сняв это право у других процессов, благодаря флагу блокировки *lock*. Благодаря этой функции мы уверены, что никакой другой процесс не будет сейчас делать ход. После того, как функция завершилась, флаг *lock* снимается, что позволяет другим процессам запускать свои функции.

Когда писатель закончил писать, то он разблокирует читателя. Читатели во время исполнения функции *signalC(C)* каскадно передают управление другим заблокированным читателям, чтобы прочитать изменившуюся информацию. Вход разделяет читателей на тех, кто был заблокирован в момент начала каскадной разблокировки и тех, кто был заблокирован позже для того, чтобы у читателей не было приоритета.

2. Задача «Обедающие философы» (за основу взят алгоритм [4][5])

Используем монитор со следующими параметрами: *mtype state[nPhil]* (состояния философов: думает, голоден, ест), *Condition self[nPhil]* (условные переменные), где *nPhil* – количество философов.

```
typedef Monitor {
    mtype state[nPhil];
    Condition self[nPhil];
}
```

Инициализация монитора с состояниями философов происходит при помощи функции *initMonitor(M, st).*

```
inline initMonitor(M, st) {
    int y = 0;
    for (y : 0..4) {
        M.state[y] = st;
    }
}
```

EmptyC(C), waitC(C), signalC(C), enterMonitor(), releaseMonitor(), Condition реализованы аналогично пункту 1. Также в программе создаем функции *takeForks(i), releaseForks(i), test(i)*.

В *test(i),* если соседи не едят и философ голоден, мы присваиваем состояние «ест» и вызываем signalC().

В *takeForks(i)* мы присваиваем состояние «голоден», вызываем *test(i)* в случае, если состояние не изменилось на «ест», вызываем waitC(C).

В *releaseForks(i)* мы присваиваем состояние «думает» и вызываем *test(i)* для соседей.

```
inline test(i) {
    if
    :: ((DP.state[(i + 1) % nPhil] != eating) &&
       (DP.state[(i + nPhil - 1) % nPhil] != eating) &&
       (DP.state[i] == hungry)) ->
          DP.state[i] = eating;
          signalC(DP.self[i]);
    :: else -> skip;
    fi;
}


inline takeForks(i) {
    enterMonitor();
    DP.state[i] = hungry;
    test(i);
    if
    :: (DP.state[i] != eating) -> waitC(DP.self[i]);
    :: else -> skip;
    fi;
    leaveMonitor();
}


inline releaseForks(i) {
    enterMonitor();
    DP.state[i] = thinking;
    test((i + 1) % nPhil);
    test((i + nPhil - 1) % nPhil);
    leaveMonitor();
}
```

Верификация

1. Проверка на свободу от взаимной блокировки.

```
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> spin -a rw.pml
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> gcc -o pan pan.c
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> ./pan -x -n

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 80 byte, depth reached 5014, errors: 0
    18133 states, stored
    29762 states, matched
    47895 transitions (= stored+matched)
     3114 atomic steps
hash conflicts:        16 (resolved)

Stats on memory usage (in Megabytes):
    1.868       equivalent memory usage for states (stored*(State-vector + overhead))
    1.936       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  130.390       total actual memory usage
```

Рисунок 3. Верификация с проверкой на deadlock.

2. Проверка на взаимоисключающий доступ. LTL–формула [1]: (RW.readers > 0 → RW.writers==0) && (RW.writers <= 1) && (RW.writers == 1 → RW.readers == 0)

```
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> ./pan -a -x -n -N mutex2
pan: ltl formula mutex2

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (mutex2)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 96 byte, depth reached 9733, errors: 0
    18133 states, stored
    29762 states, matched
    47895 transitions (= stored+matched)
     3114 atomic steps
hash conflicts:        10 (resolved)

Stats on memory usage (in Megabytes):
    2.144       equivalent memory usage for states (stored*(State-vector + overhead))
    2.124       actual memory usage for states (compression: 99.05%)
                state-vector as stored = 95 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  130.585       total actual memory usage
```

Рисунок 4. Верификация с проверкой на взаимоисключающий доступ.

3. Проверка свободы от голодания. LTL–формула:
((RW.OKtoRead.waiting > 0) → <>(RW.OKtoRead.gate == true)) &&
((RW.OKtoWrite.waiting > 0) →<>(RW.OKtoWrite.gate == true))

```
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> ./pan -a -x -n -N starvAll
pan: ltl formula starvAll

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (starvAll)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 96 byte, depth reached 9733, errors: 0
    20132 states, stored (22131 visited)
    39778 states, matched
    61909 transitions (= visited+matched)
     4272 atomic steps
hash conflicts:        22 (resolved)

Stats on memory usage (in Megabytes):
    2.381       equivalent memory usage for states (stored*(State-vector + overhead))
    2.416       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  130.878       total actual memory usage
```

Рисунок 5. Верификация с проверкой на свободу от голодания.

4. LTL–формула на проверку корректности [1]:
((RW.writers > 0) → ((RW.writers == 1) && (RW.readers == 0)))

```
alinapotapova@MacBook-Pro-Alina-2 ~/D/j/spin-files> ./pan -a -x -n -N book
pan: ltl formula book

(Spin Version 6.5.2 -- 6 December 2019)
        + Partial Order Reduction

Full statespace search for:
        never claim             + (book)
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 96 byte, depth reached 9733, errors: 0
    18133 states, stored
    29762 states, matched
    47895 transitions (= stored+matched)
     3114 atomic steps
hash conflicts:        13 (resolved)

Stats on memory usage (in Megabytes):
    2.144       equivalent memory usage for states (stored*(State-vector + overhead))
    2.123       actual memory usage for states (compression: 98.99%)
                state-vector as stored = 95 byte + 28 byte overhead
  128.000       memory used for hash table (-w24)
    0.534       memory used for DFS stack (-m10000)
  130.585       total actual memory usage
```

Рисунок 6. Проверка корректности.

Заключение

В ходе выполнения данной курсовой работы была рассмотрена проблема задач: «Читатели и писатели» и «Обедающие философы». На языке Promela системы верификации SPIN была реализована модель, которая проверялась с помощью формул LTL. Проверка показала, что все формулы выполнились.

Таким образом, удалось на практике убедиться, что система SPIN с встроенным языком Promela являются удобным инструментом для верификации моделей с помощью выражения условий корректности формулами темпоральной логики линейного времени (Linear Time Logic, LTL).

Список источников

1. M. Ben-Ari. 2006. Chapter 7.6 The problem of the readers and writers. Chapter 7.8 A monitor solution for the dining philosophers. // Principles of Concurrent and distributed programming.
2. M. Raynal. 2013. A Construct for Imperative Languages: the Monitor. // Concurrent Programming: Algorithms, Principles, and Foundations
3. И.В. Шошмина, Ю.Г. Карпов. 2009. Введение в язык Promela и систему комплексной верификации Spin.
4. Dining Philosophers, Monitors, and Condition Variables [Электронный ресурс]: Monitor-based Solution to Dining Philosophers. URL: https://home.cs.colorado.edu/~rhan/CSCI_3753_Spring_2005/CSCI_3753_Spring_2005/Lectures/02_22_05_dp_mon_cv.pdf (дата обращения 16.05.2022).
5. Dining-Philosophers Solution Using Monitors [Электронный ресурс]. URL: https://www.geeksforgeeks.org/dining-philosophers-solution-using-monitors/ (дата обращения 16.05.2022).

Приложение

1. Код программы «Читатели и писатели»

```
#define nWriters 2
#define nReaders 2

#define f1 ((cR >= 1 -> cW == 0) && (cW <= 1) && (cW == 1 -> cR == 0))
#define f2 ((RW.readers > 0 -> RW.writers == 0) && (RW.writers <= 1) && (RW.writers
== 1 -> RW.readers == 0))
#define f3 ((RW.OKtoRead.waiting > 0) -> <>(RW.OKtoRead.gate == true)) &&
((RW.OKtoWrite.waiting > 0) -> <>(RW.OKtoWrite.gate == true))
#define f4 ((RW.writers > 0) -> ((RW.writers == 1) && (RW.readers == 0)))

ltl mutex  { []f1 }
ltl mutex2 { []f2 }
ltl starv  { []f3 }
ltl book   { []f4 }

#define emptyC(C) (C.waiting == 0)

typedef Condition {
   bool gate;
   int waiting;
}

typedef Monitor {
   int readers;
   int writers;
   Condition OKtoRead;
   Condition OKtoWrite;
}

inline initMonitor(M, r, w) {
   M.readers = r;
   M.writers = w;
}

inline enterMonitor(){
atomic {
   !lock;
   lock = true;
}
}

inline leaveMonitor(){
   lock = false;
}
```

```
inline waitC(C){
atomic {
   C.waiting++;
   lock = false;
   C.gate;
   C.gate = false;
   C.waiting--;
}
}

inline signalC(C) {
atomic {
   if
   ::(C.waiting > 0) ->
      C.gate = true;
      !lock;
      lock = true;
   ::else -> skip;
   fi;
}
}

Monitor RW;
bool lock = false;
bool wantR[nReaders], wantW[nWriters], critR[nReaders], critW[nWriters];
int cR = 0, cW = 0;

inline StartRead() {
   enterMonitor();
   if
   :: (RW.writers != 0 || !emptyC(RW.OKtoWrite)) -> waitC(RW.OKtoRead);
   :: else -> skip;
   fi;
   RW.readers++;
   signalC(RW.OKtoRead);
   leaveMonitor();
}

inline StartWrite() {
   enterMonitor();
   if
   :: (RW.writers != 0 || RW.readers != 0) -> waitC(RW.OKtoWrite);
   :: else -> skip;
   fi;
   RW.writers++;
   leaveMonitor();
}

inline EndRead() {
   enterMonitor();
   RW.readers--;
```

```
    if
    :: (RW.readers == 0) -> signalC(RW.OKtoWrite);
    :: else -> skip;
    fi;
    leaveMonitor();
}

inline EndWrite() {
    enterMonitor();
    RW.writers--;
    if
    :: (emptyC(RW.OKtoRead)) -> signalC(RW.OKtoWrite);
    :: else -> signalC(RW.OKtoRead);
    fi;
    leaveMonitor();
}

proctype reader (int i) {
do
::
    wantR[i] = true;
    StartRead();
    critR[i] = true;
    cR++;
    cR--;
    critR[i] = false;
    EndRead();
    wantR[i] = false;
od;
}

proctype writer (int i) {
do
::
    wantW[i] = true;
    StartWrite();
    critW[i] = true;
    cW++;
    cW--;
    critW[i] = false;
    EndWrite();
    wantW[i] = false;
od;
}

init {
atomic {
    lock = false;
    initMonitor(RW, 0, 0);
    int i;
    for (i : 0..1) {
```

```
        run writer (i);
        run reader (i);
    }
}
}
```

2. Код программы «Обедающие философы»

```
#define nPhil 5

mtype = { thinking, hungry, eating };

typedef Condition {
    bool gate;
    int waiting;
}

typedef Monitor {
    mtype state[nPhil];
    Condition self[nPhil];
}

inline initMonitor(M, st) {
    int y = 0;
    for (y : 0..4) {
        M.state[y] = st;
    }
}

inline waitC(C){
atomic {
    C.waiting++;
    lock = false;
    C.gate;
    C.gate = false;
    C.waiting--;
}
}

inline signalC(C) {
atomic {
    if
    ::(C.waiting > 0) ->
        C.gate = true;
        !lock;
        lock = true;
    ::else;
    fi;
}
}
```

```
inline enterMonitor(){
atomic {
   !lock;
   lock = true;
}
}

inline leaveMonitor(){
   lock = false;
}

Monitor DP;
bool lock = false;

inline test(i) {
   if
   :: ((DP.state[(i + 1) % nPhil] != eating) &&
      (DP.state[(i + nPhil - 1) % nPhil] != eating) &&
      (DP.state[i] == hungry)) ->
            DP.state[i] = eating;
            signalC(DP.self[i]);
   :: else -> skip;
   fi;
}

inline takeForks(i) {
   enterMonitor();
   DP.state[i] = hungry;
   test(i);
   if
   :: (DP.state[i] != eating) -> waitC(DP.self[i]);
   :: else -> skip;
   fi;
   leaveMonitor();
}

inline releaseForks(i) {
   enterMonitor();
   DP.state[i] = thinking;
   test((i + 1) % nPhil);
   test((i + nPhil - 1) % nPhil);
   leaveMonitor();
}

proctype philosopher (int i){
do
::
   takeForks(i);
   releaseForks(i);
od;
}
```

```
inline enterMonitor(){
atomic {
   !lock;
   lock = true;
}
}

inline leaveMonitor(){
   lock = false;
}

Monitor DP;
bool lock = false;

inline test(i) {
   if
   :: ((DP.state[(i + 1) % nPhil] != eating) &&
      (DP.state[(i + nPhil - 1) % nPhil] != eating) &&
      (DP.state[i] == hungry)) ->
            DP.state[i] = eating;
            signalC(DP.self[i]);
   :: else -> skip;
   fi;
}

inline takeForks(i) {
   enterMonitor();
   DP.state[i] = hungry;
   test(i);
   if
   :: (DP.state[i] != eating) -> waitC(DP.self[i]);
   :: else -> skip;
   fi;
   leaveMonitor();
}

inline releaseForks(i) {
   enterMonitor();
   DP.state[i] = thinking;
   test((i + 1) % nPhil);
   test((i + nPhil - 1) % nPhil);
   leaveMonitor();
}

proctype philosopher (int i){
do
::
   takeForks(i);
   releaseForks(i);
od;
}
```