Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №2

«Моделирование гонки процессов»

Группа: ИУ7-41М

Студент: Дубовицкая Ольга Николаевна

Дисциплина: Математические основы верификации ПО

Преподаватель: Кузнецова Ольга Владимировна

Москва, 2025 г.

Задание

Необходимо описать взаимодействие двух процессов, работающих с одними данными. Затем место возникновения гонки необходимо дополнить мьютексами.

Отчёт должен содержать:

- описание модели взаимодействия процессов,
- демонстрация логов SPIN, в которых видна гонка,
- описание модели с мьютексом,
- результат корректного взаимодействия процессов – логи SPIN,
- вывод по работе.

Описание модели взаимодействия процессов

Модель включает в себя два процесса, которые инкрементируют общий счётчик *counter* и отслеживают прогресс выполнения с помощью массива *progress* (каждый элемент массива соответствует одному процессу и показывает, завершил ли он свою работу). Promela-код данной модели приведён в Листинге 1.

Процесс *incrementer* является основным в рамках рассматриваемой модели и отвечает за инкрементирование счётчика *counter*. Отметим, что каждый процесс внутри него получает свой идентификатор (*procs_id*) в качестве аргумента. В пределах *incrementer* реализуется чтение текущего значения *counter*, его инкрементирование и обновление соответствующего элемента массива *progress*.

Главный процесс *init* управляет созданием и запуском процессов *incrementer*. Этот процесс содержит *atomic*-блок, в первой части которого происходит создание двух процессов (в соответствии с предварительно заданным *NUM_PROCESSES*) с присвоением каждому из них уникального идентификатора, а во второй реализуется суммирование значений в массиве *progress* для того, чтобы убедиться, что либо все процессы завершили свою работу, либо счётчик равен количеству процессов. Атомарность в данном случае не будет влиять на результаты проведения гонки процессов, поскольку

эти блоки не являются частью основного инкрементирующего алгоритма, но в то же время помогают избежать ненужного увеличения пространства состояний.

Взаимодействие процессов включает в себя:

1. **Запуск процессов:**

В блоке инициализации запускаются два процесса *incrementer*. Каждый из них работает независимо, но они оба будет обращаться к общей переменной *counter*.

2. **Конкуренция за ресурс:**

Оба процесса пытаются одновременно прочитать и изменить значение переменной *counter*. Это создаёт потенциальные условия гонки, когда два процесса могут одновременно считать одно и то же значение *counter*, инкрементировать его и записать обратно, что приводит к неверному конечному значению счётчика и непредсказуемому поведению в целом.

3. **Отслеживание прогресса:**

После завершения инкрементации каждый процесс обновляет свой статус в массиве *progress*. Это позволяет главному процессу (*init*) отслеживать, сколько процессов завершили свою работу.

4. **Проверка условий:**

В конце инициализационного блока происходит проверка с помощью assert, которая гарантирует, что либо все процессы завершили свою работу (*sum < NUM_PROCESSES*), либо общий счётчик равен количеству процессов (*counter == NUM_PROCESSES*). Это условие используется для проверки корректной работы модели – если все процессы были завершены, то все подсчёты были записаны верно.

Листинг 1

```
#define NUM_PROCESSES 2

byte counter = 0;
byte progress[NUM_PROCESSES];

proctype incrementer(byte procs_id){
    int tmp;
    tmp = counter;
    counter = tmp + 1;
    progress[procs_id] = 1;
}

init {
    int i = 0;
    int sum = 0;

    atomic {
        i = 0;
        do
        :: i < NUM_PROCESSES ->
            progress[i] = 0;
            run incrementer(i);
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
    }

    atomic {
        i = 0;
        sum = 0;
        do
        :: i < NUM_PROCESSES ->
            sum = sum + progress[i];
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
        assert(sum < NUM_PROCESSES || counter == NUM_PROCESSES)
    }
}
```

В результате запуска кода из Листинга 1 будет получен вывод, приведённый в Листинге 2. Первая строка этого вывода говорит о том, что утверждение из assert-проверки было нарушено (описание нарушения записывается в trial-файл). В частности, о наличии ошибок говорит отметка «errors: 1».

Листинг 2

```
verification result:
C:/Spin/spin.exe -a  race.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 20744
pan:1: assertion violated ((sum<2)||(counter==2)) (at depth 22)
pan: wrote race.pml.trail

(Spin Version 6.5.0 -- 1 July 2019)
Warning: Search not completed
       + Partial Order Reduction

Full statespace search for:
       never claim               - (not selected)
       assertion violations      +
       cycle checks       - (disabled by -DSAFETY)
       invalid end states +

State-vector 48 byte, depth reached 24, errors: 1
       45 states, stored
       13 states, matched
       58 transitions (= stored+matched)
       53 atomic steps
hash conflicts:         0 (resolved)

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



pan: elapsed time 0.001 seconds
To replay the error-trail, goto Simulate/Replay and select "Run"
```

Если обратиться к выводу, отражающему последовательность действий процессов (Листинг 3), то можно заметить, что первая часть блока *init* создала оба процесса-инкрементатора, оба из которых сначала извлекли значение общего счётчика, а затем увеличили и сохранили его, потеряв согласованность между вычислениями. В результате этого в дальнейшем срабатывает нарушение *assert*-утверждения.

Листинг 3

```
using statement merging
  1:   proc  0 (:init::1) race.pml:19 (state 1)     [i = 0]
  2:   proc  0 (:init::1) race.pml:21 (state 2)     [((i<2))]
  2:   proc  0 (:init::1) race.pml:22 (state 3)     [progress[i] = 0]
Starting incrementer with pid 1
  3:   proc  0 (:init::1) race.pml:23 (state 4)     [(run incrementer(i))]
  4:   proc  0 (:init::1) race.pml:24 (state 5)     [i = (i+1)]
  5:   proc  0 (:init::1) race.pml:21 (state 2)     [((i<2))]
  5:   proc  0 (:init::1) race.pml:22 (state 3)     [progress[i] = 0]
Starting incrementer with pid 2
  6:   proc  0 (:init::1) race.pml:23 (state 4)     [(run incrementer(i))]
  7:   proc  0 (:init::1) race.pml:24 (state 5)     [i = (i+1)]
  8:   proc  0 (:init::1) race.pml:25 (state 6)     [((i>=2))]
  9:   proc  0 (:init::1) race.pml:20 (state 10)    [break]
 10:   proc  2 (incrementer:1) race.pml:9 (state 1) [tmp = counter]
 11:   proc  1 (incrementer:1) race.pml:9 (state 1) [tmp = counter]
 12:   proc  2 (incrementer:1) race.pml:10 (state 2)[counter = (tmp+1)]
 13:   proc  2 (incrementer:1) race.pml:11 (state 3)[progress[procs_id] = 1]
 14: proc 2 terminates
 15:   proc  1 (incrementer:1) race.pml:10 (state 2)[counter = (tmp+1)]
 16:   proc  1 (incrementer:1) race.pml:11 (state 3)[progress[procs_id] = 1]
 17: proc 1 terminates
 18:   proc  0 (:init::1) race.pml:30 (state 12)    [i = 0]
 18:   proc  0 (:init::1) race.pml:31 (state 13)    [sum = 0]
 19:   proc  0 (:init::1) race.pml:33 (state 14)    [((i<2))]
 19:   proc  0 (:init::1) race.pml:34 (state 15)    [sum = (sum+progress[i])]
 19:   proc  0 (:init::1) race.pml:35 (state 16)    [i = (i+1)]
 20:   proc  0 (:init::1) race.pml:33 (state 14)    [((i<2))]
 20:   proc  0 (:init::1) race.pml:34 (state 15)    [sum = (sum+progress[i])]
 20:   proc  0 (:init::1) race.pml:35 (state 16)    [i = (i+1)]
 21:   proc  0 (:init::1) race.pml:36 (state 17)    [((i>=2))]
 22:   proc  0 (:init::1) race.pml:32 (state 21)    [break]
spin: race.pml:38, Error: assertion violated
spin: text of failed assertion: assert(((sum<2)||(counter==2)))
#processes: 1
 23:   proc  0 (:init::1) race.pml:38 (state 22)
3 processes created
Exit-Status 0
```

Варианты исправления проблемы

В качестве первого варианта можно избавиться от гонки процессов посредством **добавления *atomic*-блока** в процесс *incrementer* (Листинг 4). Такой подход позволяет оградить процессы от ситуации, когда они одновременно пытаются изменить общую переменную – когда один процесс выполняет операцию над общей переменной, другие процессы не могут вмешиваться. Это помогает поддерживать согласованность данных.

Листинг 4

```
#define NUM_PROCESSES 2

byte counter = 0;
byte progress[NUM_PROCESSES];

proctype incrementer(byte procs_id){
    int tmp;

    atomic {
        tmp = counter;
        counter = tmp + 1;
    }

    progress[procs_id] = 1;
}

init {
    int i = 0;
    int sum = 0;

    atomic {
        i = 0;
        do
        :: i < NUM_PROCESSES ->
            progress[i] = 0;
            run incrementer(i);
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
    }

    atomic {
        i = 0;
        sum = 0;
        do
        :: i < NUM_PROCESSES ->
            sum = sum + progress[i];
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
        assert(sum < NUM_PROCESSES || counter == NUM_PROCESSES)
    }
}
```

В данном случае запуск кода будет происходить без ошибок, что подтверждают выводы из Листинга 5 и Листинга 6. Как можно заметить, запуск модифицированной модели из Листинга 4 позволяет получить безошибочный обход пространства состояний.

Листинг 5

```
verification result:
C:/Spin/spin.exe -a  atomic.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 2524

(Spin Version 6.5.0 -- 1 July 2019)
       + Partial Order Reduction

Full statespace search for:
       never claim               - (not selected)
       assertion violations      +
       cycle checks        - (disabled by -DSAFETY)
       invalid end states +

State-vector 48 byte, depth reached 22, errors: 0
       52 states, stored
       21 states, matched
       73 transitions (= stored+matched)
       68 atomic steps
hash conflicts:         0 (resolved)

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


unreached in proctype incrementer
       (0 of 5 states)
unreached in init
       (0 of 24 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Листинг 6

```
  0:   proc  - (:root:) creates proc  0 (:init:)
  1:   proc  0 (:init::1) atomic.pml:22 (state 1)   [i = 0]
  3:   proc  0 (:init::1) atomic.pml:24 (state 2)   [((i<2))]
  4:   proc  0 (:init::1) atomic.pml:25 (state 3)   [progress[i] = 0]
Starting incrementer with pid 1
  5:   proc  0 (:init::1) creates proc  1 (incrementer)
  5:   proc  0 (:init::1) atomic.pml:26 (state 4)   [(run incrementer(i))]
  6:   proc  0 (:init::1) atomic.pml:27 (state 5)   [i = (i+1)]
  7:   proc  0 (:init::1) atomic.pml:24 (state 2)   [((i<2))]
  8:   proc  0 (:init::1) atomic.pml:25 (state 3)   [progress[i] = 0]
Starting incrementer with pid 2
  9:   proc  0 (:init::1) creates proc  2 (incrementer)
  9:   proc  0 (:init::1) atomic.pml:26 (state 4)   [(run incrementer(i))]
 10:   proc  0 (:init::1) atomic.pml:27 (state 5)   [i = (i+1)]
 11:   proc  0 (:init::1) atomic.pml:28 (state 6)   [((i>=2))]
 12:   proc  0 (:init::1) atomic.pml:23 (state 10)  [break]
 13:   proc  2 (incrementer:1) atomic.pml:10 (state 1)    [tmp = counter]
```

```
 14:   proc  2 (incrementer:1) atomic.pml:11 (state 2)     [counter = (tmp+1)]
 15:   proc  1 (incrementer:1) atomic.pml:10 (state 1)     [tmp = counter]
 16:   proc  1 (incrementer:1) atomic.pml:11 (state 2)     [counter = (tmp+1)]
 17:   proc  0 (:init::1) atomic.pml:33 (state 12)  [i = 0]
 18:   proc  0 (:init::1) atomic.pml:34 (state 13)  [sum = 0]
 19:   proc  0 (:init::1) atomic.pml:36 (state 14)  [((i<2))]
 20:   proc  0 (:init::1) atomic.pml:37 (state 15)  [sum = (sum+progress[i])]
 21:   proc  0 (:init::1) atomic.pml:38 (state 16)  [i = (i+1)]
 22:   proc  0 (:init::1) atomic.pml:36 (state 14)  [((i<2))]
 23:   proc  0 (:init::1) atomic.pml:37 (state 15)  [sum = (sum+progress[i])]
 24:   proc  0 (:init::1) atomic.pml:38 (state 16)  [i = (i+1)]
 25:   proc  0 (:init::1) atomic.pml:39 (state 17)  [((i>=2))]
 26:   proc  0 (:init::1) atomic.pml:35 (state 21)  [break]
 27:   proc  0 (:init::1) atomic.pml:41 (state 22)
       [assert(((sum<2)||(counter==2)))]
 28:   proc  2 (incrementer:1) atomic.pml:14 (state 4)     [progress[procs_id] = 1]
 28:   proc  2 (incrementer:1) terminates
 29:   proc  1 (incrementer:1) atomic.pml:14 (state 4)     [progress[procs_id] = 1]
 29:   proc  1 (incrementer:1) terminates
 29:   proc  0 (:init::1) terminates
3 processes created
```

Второй вариант избавления от гонки процессов (Листинг 7) заключается в использовании **механизма блокировок** (он будет реализован с помощью макросов *spin_lock* и *spin_unlock*).

spin_lock использует цикл, который продолжается до тех пор, пока блокировка не будет получена. Внутри этого цикла есть конструкция *atomic*, которая гарантирует, что операции внутри него выполняются как единое целое. Если *mutex* равен 0 (т.е. блокировка свободна), то она устанавливается в 1 и процесс выходит из цикла. Если же *mutex* уже равен 1 (т.е. блокировка занята), то процесс просто пропускает выполнение (операция *skip*) и продолжает выполнение цикла.

Так, перед тем как инкрементировать общий счётчик *counter*, процесс будет вызывать *spin_lock(mutex)*, что в данный момент времени гарантирует доступ только одного процесса к коду, который изменит счётчик. После получения блокировки процесс считывает текущее значение *counter*, инкрементирует его и записывает обратно. Затем будет вызвано *spin_unlock(mutex),* освобождающее блокировку, чтобы другой процесс теперь тоже мог получить доступ к общему счётчику.

Листинг 7

```
#define spin_lock(mutex) \
    do \
    :: 1 -> atomic { \
        if \
        :: mutex == 0 -> \
            mutex = 1 ; \
            break \
        :: else -> skip \
        fi \
    } \
    od

#define spin_unlock(mutex) \
    mutex = 0

#define NUM_PROCESSES 2

byte counter = 0;
byte progress[NUM_PROCESSES];
byte mutex = 0;

proctype incrementer(byte procs_id){
    int tmp;

    spin_lock(mutex);
    tmp = counter;
    counter = tmp + 1;
    spin_unlock(mutex);

    progress[procs_id] = 1;
}

init {
    int i = 0;
    int sum = 0;

    atomic {
        i = 0;
        do
        :: i < NUM_PROCESSES ->
            progress[i] = 0;
            run incrementer(i);
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
    }

    atomic {
        i = 0;
        sum = 0;
        do
        :: i < NUM_PROCESSES ->
            sum = sum + progress[i];
            i++;
        :: i >= NUM_PROCESSES -> break
        od;
        assert(sum < NUM_PROCESSES || counter == NUM_PROCESSES)
    }
}
```

Аналогично первому варианту запуск кода также будет происходить без ошибок, что подтверждают выводы из Листинга 8 и Листинга 9.

Листинг 8

```
verification result:
C:/Spin/spin.exe -a  mutex.pml
gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c
./pan -m10000
Pid: 20260

(Spin Version 6.5.0 -- 1 July 2019)
       + Partial Order Reduction

Full statespace search for:
       never claim               - (not selected)
       assertion violations      +
       cycle checks        - (disabled by -DSAFETY)
       invalid end states  +

State-vector 48 byte, depth reached 36, errors: 0
      130 states, stored
       80 states, matched
      210 transitions (= stored+matched)
      195 atomic steps
hash conflicts:         0 (resolved)

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


unreached in proctype incrementer
       (0 of 17 states)
unreached in init
       (0 of 24 states)

pan: elapsed time 0 seconds
No errors found -- did you verify all claims?
```

Листинг 9

```
  0:   proc  - (:root:) creates proc  0 (:init:)
  1:   proc  0 (:init::1) mutex.pml:38 (state 1)    [i = 0]
  3:   proc  0 (:init::1) mutex.pml:40 (state 2)    [((i<2))]
  4:   proc  0 (:init::1) mutex.pml:41 (state 3)    [progress[i] = 0]
Starting incrementer with pid 1
  5:   proc  0 (:init::1) creates proc  1 (incrementer)
  5:   proc  0 (:init::1) mutex.pml:42 (state 4)    [(run incrementer(i))]
  6:   proc  0 (:init::1) mutex.pml:43 (state 5)    [i = (i+1)]
  7:   proc  0 (:init::1) mutex.pml:40 (state 2)    [((i<2))]
  8:   proc  0 (:init::1) mutex.pml:41 (state 3)    [progress[i] = 0]
```

```
Starting incrementer with pid 2
  9:   proc  0 (:init::1) creates proc  2 (incrementer)
  9:   proc  0 (:init::1) mutex.pml:42 (state 4)    [(run incrementer(i))]
 10:   proc  0 (:init::1) mutex.pml:43 (state 5)    [i = (i+1)]
 11:   proc  0 (:init::1) mutex.pml:44 (state 6)    [((i>=2))]
 12:   proc  0 (:init::1) mutex.pml:39 (state 10)   [break]
 13:   proc  2 (incrementer:1) mutex.pml:25 (state 1)     [(1)]
 14:   proc  1 (incrementer:1) mutex.pml:25 (state 1)     [(1)]
 15:   proc  0 (:init::1) mutex.pml:49 (state 12)   [i = 0]
 16:   proc  0 (:init::1) mutex.pml:50 (state 13)   [sum = 0]
 17:   proc  0 (:init::1) mutex.pml:52 (state 14)   [((i<2))]
 18:   proc  0 (:init::1) mutex.pml:53 (state 15)   [sum = (sum+progress[i])]
 19:   proc  0 (:init::1) mutex.pml:54 (state 16)   [i = (i+1)]
 20:   proc  0 (:init::1) mutex.pml:52 (state 14)   [((i<2))]
 21:   proc  0 (:init::1) mutex.pml:53 (state 15)   [sum = (sum+progress[i])]
 22:   proc  0 (:init::1) mutex.pml:54 (state 16)   [i = (i+1)]
 23:   proc  0 (:init::1) mutex.pml:55 (state 17)   [((i>=2))]
 24:   proc  0 (:init::1) mutex.pml:51 (state 21)   [break]
 25:   proc  0 (:init::1) mutex.pml:57 (state 22)
       [assert(((sum<2)||(counter==2)))]
 26:   proc  1 (incrementer:1) mutex.pml:25 (state 2)     [((mutex==0))]
 27:   proc  1 (incrementer:1) mutex.pml:25 (state 3)     [mutex = 1]
 28:   proc  1 (incrementer:1) mutex.pml:25 (state 12)    [break]
 29:   proc  1 (incrementer:1) mutex.pml:26 (state 13)    [tmp = counter]
 30:   proc  2 (incrementer:1) mutex.pml:25 (state 5)     [else]
 31:   proc  2 (incrementer:1) mutex.pml:25 (state 6)     [(1)]
 32:   proc  2 (incrementer:1) mutex.pml:25 (state 1)     [(1)]
 33:   proc  2 (incrementer:1) mutex.pml:25 (state 5)     [else]
 34:   proc  2 (incrementer:1) mutex.pml:25 (state 6)     [(1)]
 35:   proc  2 (incrementer:1) mutex.pml:25 (state 1)     [(1)]
 36:   proc  2 (incrementer:1) mutex.pml:25 (state 5)     [else]
 37:   proc  2 (incrementer:1) mutex.pml:25 (state 6)     [(1)]
 38:   proc  2 (incrementer:1) mutex.pml:25 (state 1)     [(1)]
 39:   proc  1 (incrementer:1) mutex.pml:27 (state 14)    [counter = (tmp+1)]
 40:   proc  1 (incrementer:1) mutex.pml:28 (state 15)    [mutex = 0]
 41:   proc  1 (incrementer:1) mutex.pml:30 (state 16)    [progress[procs_id] = 1]
 42:   proc  2 (incrementer:1) mutex.pml:25 (state 2)     [((mutex==0))]
 43:   proc  2 (incrementer:1) mutex.pml:25 (state 3)     [mutex = 1]
 44:   proc  2 (incrementer:1) mutex.pml:25 (state 12)    [break]
 45:   proc  2 (incrementer:1) mutex.pml:26 (state 13)    [tmp = counter]
 46:   proc  2 (incrementer:1) mutex.pml:27 (state 14)    [counter = (tmp+1)]
 47:   proc  2 (incrementer:1) mutex.pml:28 (state 15)    [mutex = 0]
 48:   proc  2 (incrementer:1) mutex.pml:30 (state 16)    [progress[procs_id] = 1]
 48:   proc  2 (incrementer:1) terminates
 48:   proc  1 (incrementer:1) terminates
 48:   proc  0 (:init::1) terminates
3 processes created
```

Выводы

В результате выполнения лабораторной работы была описана модель взаимодействия двух конкурирующих процессов, которые инкрементируют один счётчик.

Были рассмотрены варианты избавления от ошибок и непредсказуемого поведения, возникающих в местах гонки процессов. Реализация обоих вариантов привела к корректному взаимодействию процессов.