Визуализация алгоритмов реализации взаимоисключений средствами пиктограммной среды программирования

И.Н.Грибанова[1], А.С. Караваева[2], А.А. Леонов[3], А.Г. Леонов[4], Д.В. Мащенко[5], В.А. Оганисян[6]

[1]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, nig@niisi.com;

[2]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, aleksandrakaravaevaa@yandex.ru;

[3]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, anton11-02@mail.com;

[4]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, МГУ им. М. В. Ломоносова, Москва, Россия, МПГУ, Москва, Россия, Государственный университет управления, Москва, Россия, dr.l@vip.niisi.ru;

[5]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, mashchenko.darya.vlad@yandex.ru;

[6]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, kovyrshina@niisi.ru

Аннотация. Курсы по информатике и информационно-коммуникационным технологиям могут включать широкий набор тем от основ программирования и элементарных знаний об устройстве компьютеров до теории формальных грамматик, языков искусственного интеллекта, операционных систем, 3D-графики и т.п. Основную сложность при этом составляет математическая компонента курсов, что требует соответствующей подготовки слушателей. Однако, и в программировании есть темы, сложные для понимания студентами даже с глубокими академическими знаниями. Одной из таких тем является параллельное программирование, когда с одной стороны требуется изучить и написать учебные высокоэффективные алгоритмы решения задач для выполнения одновременных вычислений на различных ЭВМ или процессорах, с другой стороны, что не менее сложно для понимания - обеспечить параллельную работу различных частей программы с разделяемым доступом к общим данным. Освоение азов параллельного программирования усложняется отсутствием адекватной визуализации параллельной работы процессов на компьютере. Педагогам остается иллюстрировать тему описывая знакомые студентам ситуации из реального мира, например, задачу одновременного проезда четырех автомобилей через нерегулируемый перекресток. С другой стороны, современные тенденции требует понижения возраста начала знакомства растущего ребенка с элементами программирования и алгоритмизации. Дошкольники, дети четвертого года жизни, с успехом осваивают в игре основные понятия последовательного программирования. используя программно-управляемых роботов и их виртуальных двойников в цифровой образовательной среде ПиктоМир. Та же ЦОС ПиктоМир используется для пропедевтики программирования при преподавании для любого возраста, включая студентов университетов. ПиктоМир обладает не только простотой в освоении, но допускает использование в заданиях одновременно нескольких исполнителей-роботов, каждый из которых действует по собственной программе. Наглядность процесса выполнения программ в ПиктоМире привела авторов к мысли использовать ЦОС ПиктоМир для демонстрации работы классических алгоритмов взаимоисключений в курсах по параллельному программированию. Опыт авторов показал высокую эффективность подобного подхода, что позволило студентам за короткий срок понять проблему и успешно решить все задачи темы. Указанный подход апробировался в курсах Операционные системы (ИИС ГУУ) и Алгоритмы и структуры данных (Институт Детства МПГУ).

Ключевые слова: взаимное исключение, процесс, deadlock, критическая секция, ПиктоМир. операционные системы, алгоритмы.

1. Введение

Вопросы параллельных вычислений стали активно исследоваться с 60-х годов. Необходимость таких работ легко поясняется примерами того времени. В СССР, например, в Москве в больших киноконцертных залах всегда было несколько касс. В случае больших загрузки, с целью исключить большие очереди на текущий киносеанс, билеты продавались сразу в двух и более кассах. В отсутствии компьютеров и информационных систем, кассиры должны были договариваться между собой, кто какие места продает. В случае, если билеты не содержали отпечатанный типографским способом ряд и место в зале, кассирам приходилось отмечать проданные места на одном листе схемы киноконцертного зала, но и даже в случае, когда билет содержал напечатанное место и ряд создавались определенные сложности. Так, например, пара кассиров делила пачки билетов примерно пополам, и

каждая касса продавала свою часть. Кассиры работали с разной скоростью, поэтому в процессе продажи у одной из касс могла выстроиться очередь, а другая вообще оказывалась свободной в случае распродажи всех отведенных кассе билетов. Аналогичные трудности возникали и при продаже билетов в железнодорожных кассах дальнего следования. Если касса не находилась непосредственно на вокзале, то кассир вынужден был заранее получать список билетов, которые он может реализовать, или ему требовалось регулярно дозваниваться до вокзала, сообщая в центральную кассу о факте продаже билета на конкретный поезд или спрашивать о наличии места на данное направление. Естественно, что такая работа протекала чрезвычайно медленно, создавая очереди, часто возникали сбои, когда, например, несколько билетов на одно и то же место продавали одновременно разным пассажирам. Можно говорить, что в информационной модели системы по продаже билетов, кассиры и выполняемая ими работа представляли независимые процессы, а список билетов – общий ресурс. При этом работа процессов – касс была организована параллельно, и если не учитывать общий ресурс, то независимо друг от друга.

Задачу синхронизации параллельной работы процессов можно решить и для N процессов (или ЭВМ), связанных между собой только обращением к общему ресурсу [1]. Для этого процесс рассматривается как отдельная программа, в которой выделяется часть, называемая, обычно, критической секцией или критическим интервалом. В этом месте программы осуществляется доступ к общим данным, который должен быть монопольным, то есть одновременно в своей критической секции может находиться только один процесс.

Считается, что первой статьей, решающей эту проблему, называемую проблемой взаимного исключения, была статья Эдсгера Дейкстры [2], в которой автор сформулировал условия задачи обеспечения взаимоисключающего доступ к критической секции среди конкурирующих процессов.

- каждый процесс выполняет последовательность инструкций (команд) в бесконечном цикле.
- инструкции разделены на четыре части: код остаточная часть (не критично одновременное выполнение), вход в критическую секцию (код, выполняющийся перед входом в критический интервал), критический интервал и выход (код, который выполняется после критического интервала).

Для решения поставленной задачи необходимо написать программный код для двух разделов входа и выхода, чтобы удовлетворялось условие, что никакие два процесса не находятся в своих критических секциях одновременно. Предполагается, что код критической секции и остальной части процесса не будет меняться при обеспечении взаимоисключений. Естественно также предположить, что решение будет идентичным для всех процессов и будет использовать общие переменные [3].
Исходный код каждого процесса выглядит так:

```
loop forever
    remainder code;
    entry code;
    critical section;
    exit code;
end loop
```

На рис. 1 представлено решение Эдсгера Дейкстры из указанной выше статьи.

В дальнейшем алгоритм неоднократно улучшался. Так в 1974 Лесли Лампортом был предложен так называемый алгоритм Пекарни [4], работающий по аналогии с настоящим заведением. Покупатели-процессы, заходят в пекарню, и каждый входящий по очереди получает номерок, то есть на единицу больше предыдущего покупателя. Специальное табло показывает номер клиента, обслуживаемого в данный момент времени. Остальные посетители ожидают в очереди, пока не закончится обслуживание клиента и табло не покажет следующий номер. Клиент, который уже совершил покупку, отдает свой номерок продавцу и тот не только вызывает следующего по очереди, но и увеличивает на единицу допустимое число входных номеров. Покупатели-процессы получают номера из некоторой глобальной величины [5].

Алгоритм неоднократно дорабатывался, из-за проблем с одновременным получением несколькими процессами одного и того же номера [6,7].

Несмотря на наличие большого количества статей по данной тематике представленные алгоритмы или не удовлетворяли принципу очередности (FIFO), либо использовали неограниченные целые величины. Очередное решение было предложено уже в этом веке [8]. Автор утверждает, что все требования проблемы взаимоисключений были выполнены.

Возвращаясь к работам Эдсгера Дейкстры отметим, что он не только опубликовал первым статью в области параллельных вычислений, выявившей и решившей проблему взаимного исключения, но и много сделал в области самих распределённых вычислений, а также ввел понятие семафоров (semaphore) - целочисленных ве-

личин с атомарными операциями как примитивов синхронизации процессов (потоков).

```
“integer j;
Li0:  b[i] := false;
Li1:  if k ≠ i then
Li2:  begin c[i] := true;
Li3:  if b[k] then k := i;
      go to Li1
      end
        else
Li4:  begin c[i] := false;
        for j := 1 step 1 until N do
          if j ≠ i and not c[j] then go to Li1
      end;
      critical section;
      c[i] := true;  b[i] := true;
      remainder of the cycle in which stopping is allowed;
      go to Li0”
```

Рис. 1. Алгоритм взаимоисключений для N процессов Э.Дейкстры.

2. Классические задачи на алгоритмы взаимоисключений

Для облегчения понимания алгоритмов взаимоисключений проще рассматривать проблему на примере двух процессов, используя для синхронизации простые переменные [9].

Более того, для школьников естественно использовать для объяснения алгоритмы, реализованные на школьном алгоритмическом языке, который также часто называют КуМир (по имени программной среды, в которой этот яхык реализован)[10].

Здесь задача представлена в переформулированном виде: требуется решить проблему синхронизации выполнения двух процессов [11], где алгоритм-процесс выполняется циклически, вначале выполняется вспомогательный алгоритм «критическая секция», а затем выполняется «окончание алгоритма», где процессу уже не требуется общий ресурс:

```
алг процесс

нач

| нц

| | критическая секция

| | окончание алгоритма

| кц

кон
```

Естественно, что это еще не решение проблемы, поскольку отсутствует синхронизация. Модифицируем алгоритмы, так, чтобы свою критическую секцию мог выполнять один и только один процесс, что и будет решением проблемы синхронизации. Это решение состоит в добавлении целой переменной «переключатель» с двумя значениями 1 и 2, по номерам процессов, которая принимает эти значения, в зависимости от номера процесса, допущенных в данное время в свою критическую секцию.

На рис. 2 представлено решение проблемы взаимоисключений с переключателем (КуМир). Считается, что алгоритмы в нижних прямоугольниках выполняются параллельно.

Пока переключатель равен 1, процесс2 не может войти свою критическую секцию и находится в бесконечном цикле ожидания, пока переключатель не станет равен 2. В свою очередь переключатель станет равен 2, когда первый процесс выйдет из критической секции, и изменит содержимое переключателя. Так же работает и второй процесс.

Как и ожидалось, в каждый момент времени только один процесс может находится в своем критическом интервале. Но процессы входят в

свои критические секции строго последовательно: первый, второй, первый, второй...

При таком подходе один из процессов может быть неоправданно задержан, хотя общий ресурс будет свободен для использования.

Здесь, естественно, нужно сформулировать второе правило синхронизации: задержка любого процесса, вне его критического интервала не должна влиять на ход работы остальных процессов.

На рис. 3 представлено решение проблемы взаимоисключений с двумя флагами (КуМир).

```
алг параллельные процессы
нач цел переключатель
| переключатель := 1
| процесс1 : процесс 2 | параллельное выполнение процессов
кон
```

```
алг процесс 1
нач
| нц
| | нц пока переключатель = 2
| | кц
| | критическая секция1
| | переключатель := 2
| | окончание 1
| кц
кон
```

```
алг процесс 2
нач
| | нц
| | нц пока переключатель = 1
| | кц
| | критическая секция2
| | переключатель := 1
| | окончание 2
| кц
кон
```

Рис. 2. Решение проблемы взаимоисключений с переключателем (КуМир).

Интуитивно, представленный алгоритм с двумя флагами кажется более удачным: он не требует соблюдения четкой последовательности выполнения критических секций. Если один из процессов работает быстрее, то он сможет чаще попадать в критическую секцию. Но и этот алгоритм не лишен недостатков: не исключен случай, когда оба процесса готовы войти в критические секции (флаг1 и флаг2 приняли значения «да»), тогда ожидание в цикле будет длиться бесконечно и не один из процессов не сможет выполнятся дальше. Эта проблема приводит к формулировке третьего требования: решение относительно того, какой процесс должен войти в свой критический интервал, должно быть принято за конечное время.

Как известно, классическое решение проблемы взаимоисключений требует использования одного переключателя и двух флагов.

На рис. 4 представлено решение проблемы взаимоисключений с тремя переменными (КуМир).

Это решение является фактическим объединением первого и второго вариантов. Для синхронизации требуются и переменная переключатель, и флаги (флаг1 и флаг2). Переключатель "отвечает" за номер процесса, который выполняется в текущий момент в критической секции. Можно заметить, что величина переключатель вообще не требует инициализации, так как устанавливается в 1 или 2 до цикла ожидания. В отличии от переключателя, флаги в начале сбрасываются (им присваивается «нет»). Их нельзя не инициализировать, так как какой-нибудь из процессов может «вырваться» вперед, и другой не успеет проинициализировать свой флаг. В начале процессов флаг поднимается (ему присваивается «да»), то есть процесс готов войти в критическую секцию. Переключателю присваивается номер другого процесса, чтобы «пропустить» на выполнение этот другой процесс. Цикл ожидания прервется, если тот, или другой процесс покинет свою критическую секцию (установив свой флаг в 0). Фактически функция переключателя состоит только в обходе ситуации, которая могла возникнуть на втором решении, то есть исключить взаимную блокировку процессов.

```
алг параллельные процессы
нач лог флаг1, флаг2
| флаг1 := "нет"
| флаг2 := "нет"
| процесс1 : процесс 2 | параллельное выполнение процессов
кон
```

```
алг процесс 1
нач
| нц
| | флаг1 := "да"
| | нц пока флаг2
| | кц
| | критическая секция1
| | флаг1 := "нет"
| | окончание 1
| кц
кон
```

```
алг процесс 2
нач
| нц
| | флаг2 := "да"
| | нц пока флаг1
| | кц
| | критическая секция2
| | флаг2 := "нет"
| | окончание 2
| кц
кон
```

Рис. 3. Решение проблемы взаимоисключений с двумя флагами (КуМир)

```
алг параллельные процессы
нач лог флаг1, флаг2, цел переключатель
| флаг1 := "нет"
| флаг2 := "нет"
| процесс1 : процесс 2 | параллельное выполнение процессов
кон
```

```
алг процесс 2
нач
| нц
| | флаг1 := "да"
| | переключатель := 2
| | нц пока флаг2 и
| | переключатель = 2
| | кц
| | критическая секция1
| | флаг1 := "нет"
| | окончание 1
| кц
кон
```

```
алг процесс 2
нач
| нц
| | флаг2 := "да"
| | переключатель := 1
| | нц пока флаг1 и
| | переключатель = 1
| | кц
| | критическая секция2
| | флаг2 := "нет"
| | окончание 2
| кц
кон
```

Рис. 4. Решение проблемы взаимоисключений с тремя переменными (КуМир)

3. Постановка проблемы

Если ставить перед собой цель объяснять вопросы синхронизации параллельных процессов наиболее просто, то использование при изложении языка системы КуМир вполне оправдано, однако, так как ЦОС КуМир не имеет возможности параллельного выполнения программ, то ученики не смогут попрактиковаться в решении задач на параллельные процессы. Алгоритмы, записанные на школьном алгоритмическим языке фактически эквивалентны записи на псевдокоде: их можно составлять, исправлять, обсуждать, но нельзя выполнить на компьютере, что является достаточно странным методическим приемом в 21 веке. Однако, использование школьного алгоритмическог языка как псевдокода вполне оправдано.

Вполне естественно использование в образовательном процессе производственного языка программирования, такого как C++. Библиотеки современных фреймворков, такие как Qt, имеют обширную поддержку синхронизации параллельных процессов-нитей (Thread) в своем арсенале [12]. Тем интереснее смоделировать синхронизацию работы двух и более процессов, для получения демонстрации, например, взаимной блокировки двух процессов, решение проблемы взаимоисключений и т.д.

Программа содержит в себе запуск N=2 (nThread=2) процессов (нитей). Каждый процесс имеет свой целочисленный идентификатор (id = 0 или 1), который является параметром функций:

```
void Threads::Critical_section(int id)
и
void Threads::Reminder_section(int id)
```

Для успешного моделирования параллельных процессов, в эти функции включена не только проверка единственности процесса в своей критической секции, но и определенная существенная временная задержка, так как даже если не использовать никакие способы синхронизации нитей, попадание двух процессов одновременно в "короткую" критическую секцию практически исключено. Напротив, реальные процессы в реальных задачах могут находится в критических интервалах достаточно длительное время.

Заготовка основной функции процесса выглядит так:

```
      {
// код писать сюда
   Critical_section(id);
// код писать сюда
   Reminder_section(id);
    }
```

Для полноценного решения задачи по мотивам алгоритма Э.Дейкстры студентам предлагается две использовать глобальные переменны:

```
static int swich=0;
static int key[2]= {0,0};
```

Где swich - переключатель процессов (0 и 1), написанный с пропущенной буквой t, чтобы отличить переменную от оператора C++, а массив key[] - суть флаги процессов.

Наблюдая за выводом на экран, можно увидеть все ожидаемые эффекты, такие как чередование работы процессов (при использовании только одного переключателя), взаимную блокировку процессов (при попытке синхронизации с использованием только флагов) и нормальное выполнение, (при полноценной реализации алгоритма взаимоисключений, когда один процесс чаще другого попадает в критическую секцию и нет аварийной ситуации), анализ вывода показывает также что два процесса одновременно не оказываются в критическом интервале.

Однако, новичкам в программировании сложно понимать методы класса Thread-ов, а студентам тяжело представить, что один и тот же код выполняется различными процессами параллельно и одновременно, хотя визуально все содержится в одном файле.

К другим проблемам можно отнести эффекты, возникающие при работе алгоритмов оптимизации компилятора C++.

Во втором примере (на рис. 3) уже излагался неправильный метод синхронизации процессов, когда некоторая последовательность выполнения программы приводила к взаимной блокировке процессов. На C++ (в фреймворке Qt) этот пример выглядит так:

```
key[id]=1; while(key[1-id]);
   Critical_section(id);
key[id]=0;
  Reminder_section(id);
```

Можно обратить внимание, что одновременное выполнение первой строки в нитях приводит к взаимной блокировке, однако, имеется неопределённость поведения некоторых компиляторов C++, которое приводит к интересным последствиям.

Легко заметить, что переменой key[] дважды присваивается значение в функции. Но, эта величина нигде не участвует в вычислениях. Поэтому код может быть соптимизирован компилятором, когда операции присвоения просто будут выброшены из функции.

В качестве примера на рис. 6 изображено решение задачи взаимоисключений с использованием директив mutex() на C++ (Qt).

```
const int MaxWait = 200;
const int MaxSleep = 500; // Максимальный sleep
const int MaxRun = 30; // Максимальный
const int nThreads = 2; // Число потоков

class Threads : public QThread {

private:  int id;

public:
    Threads(int i):
        id(i){}
    void Critical_section(int);
    void Reminder_section(int);
    void run(void) override;

};


static int cr;
static int alarm=0;
static int swich=0;
static int key[2]= {0,0};
static QMutex mutex;

void Threads::run(void)
    {   for (int i=0; i<MaxRun*(nThreads-id); i++)
      {
// код писать сюда
//    while();

       mutex.lock();
    Critical_section(id);
      mutex.unlock();
//    swich=1-swich;

// код писать сюда
    Reminder_section(id);
      }
    }


void Threads::Critical_section(int i)
{    if (cr++) { alarm++; qDebug() <<  " Critical section " << i << " alarm  "  ;}
 //   QTime time = QTime::currentTime();
      msleep(static_cast<unsigned int>(QRandomGenerator::global()->bounded(MaxWait)));
        cr--;
}

void Threads::Reminder_section(int i)
{
     qDebug() <<  " Process " << i ;
//     QTime time = QTime::currentTime();
     msleep(MaxSleep*static_cast<unsigned long>(i));
     msleep(static_cast<unsigned int>(QRandomGenerator::global()->bounded(MaxWait)));
}


int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    Threads *Thread[nThreads];

    for (int k = 0; k < nThreads; k++)
        {
            Thread[k] = new Threads(k);
            Thread[k]->start();
        }
    for (int k = 0; k < nThreads; k++)
        {
            Thread[k]->wait();
        }
    return alarm > 0 ? -1 : alarm;
}
```

Рис. 5. Решение задачи взаимоисключений с использованием директив mutex() на C++ (Qt)

4. Визуализация алгоритмов

В предыдущем параграфе было указано на сложности освоения темы студентами, при использовании алгоритмов, реализованных на производственном языке программирования С++. Конечно, возможен подход, когда используется специально-разработанная библиотека для эффектной графической визуализации, например столкновения роботов-процессов при ошибках синхронизации или бесконечное ожидание перед некоторым объектом, как демонстрация взаимной блокировки процессов. Однако и это не может гарантировать успех занятиям, так как сложность языка С++ никуда не исчезает. К сожалению, подобные курсы доступны только студентам университетов или очень увлеченным старшеклассникам.

С другой стороны, ЦОС ПиктоМир уже несколько лет активно используется в дошкольных образовательных организациях, а также качестве пропедевтики программирования в школах, кружках, в курсах различных образовательных учреждениях и университетах [13].

ЦОС ПиктоМир обладает продуманным интерфейсом для составления простейших программ детьми даже в раннем возрасте [14]. В отличии от С++ на освоение языка Пикто (пиктографического языка ЦОС ПиктоМир) уходит совсем мало времени. Если студенты университета по какой-то причине еще не знакомы с ПиктоМиром. то для проведения полноценного занятия по алгоритмам взаимоисключений по мотивам решений Э.Лейкстры достаточно пары академических часов [15].

В ПиктоМире в качестве исполнителей используются роботы, выполняющие работу и двигающиеся по клетчатому полю-космодрому. Робот, который будет программироваться при решении задач называется Вертун. У Вертуна всего 4 команды:

- вперед;
- налево;
- направо;
- закрасить.

В простейшем случае будут использоваться только 3 команды и будут составлены индивидуальные программы для каждого из роботов.

Вводными пояснениями объясняются три требования к синхронизации алгоритмов, при использовании ЦОС ПиктоМир.

Три правила робототехники:

- Правило 1. В каждый момент времени только один робот может попытаться занять поле или ресурс.
- Правило 2. Задержка любого робота в рамках другой работы не должна влиять на ход работы остальных роботов.
- Правило 3. Решение занять поле или ресурс должно быть принято роботом за конечное время.

Таким образом разделяемым ресурсом для роботов будет клетка поля. Действительно, если роботы сталкиваются при одновременном прохождении одной и той же клетки, то они ломаются, что полностью соответствует результату столкновения в реальном, а не виртуальном мире.

В качестве примера на рис. 7 изображено исходное состояние и программа для робота, не использующая синхронизацию.

В этой задаче роботы должны многократно закрашивать (чинить) "сломанную" клетку, при этом программа для робота, отмеченная зеленым кружком (при черно-белой печати просто кружком) выполняет стандартные действия:

- робот идет два шага вперед
- красит клетку
- разворачивается
- идет в исходное положение
- разворачивается

Можно считать, что в исходное положение робот переходит для получения очередной порции краски.

Такие действия робот выполняет в примере циклически всего 3 раза, что не ограничивает общности.

Программа второго робота скрыта, но она полностью аналогична программе первого робота. Естественно, что через несколько шагов роботы столкнутся, так как попробуют занять одну и ту же клетку. Авария наглядно продемонстрирована (на рис. 8 изображено столкновение роботов из-за отсутствия в программах синхронизации). При этом на рисунке хорошо видна программа для второго робота, которая была скрыта на предыдущем рисунке.

Для решения проблемы взаимоисключения для двух роботов в ПиктоМире потребуется пройти все этапы решения задачи Э.Дейкстры, для начала использовать переменную для синхронизации роботов. Несмотря на простоту ЦОС ПиктоМир в ней имеются простейшие аналоги переменных, в частности кувшин - счетчик. В кувшин можно бросать камушки или доставать их из него, можно проверять состояние кувшина: пуст или не пуст кувшин (у кувшина есть еще несколько команд, которые мы не описали). Фактически кувшин является целочисленным счетчиком, а для решения ряда задач потребуется только значения 0 и 1.

Программа управления роботом содержит все стандартные части: и критический интервал (вперед на выделенную клетку, закрасить, развернуться и сделать шаг) и остаток программы, когда робот возвращается в исходное положение,

разворачивается и снова идет к клетке. Очевидно, что цикл ожидания по условию пока должен выполняться до шага на выделенную клетку. Условие ожидания для робота будет состоять в проверке, пуст кувшин или не пуст.

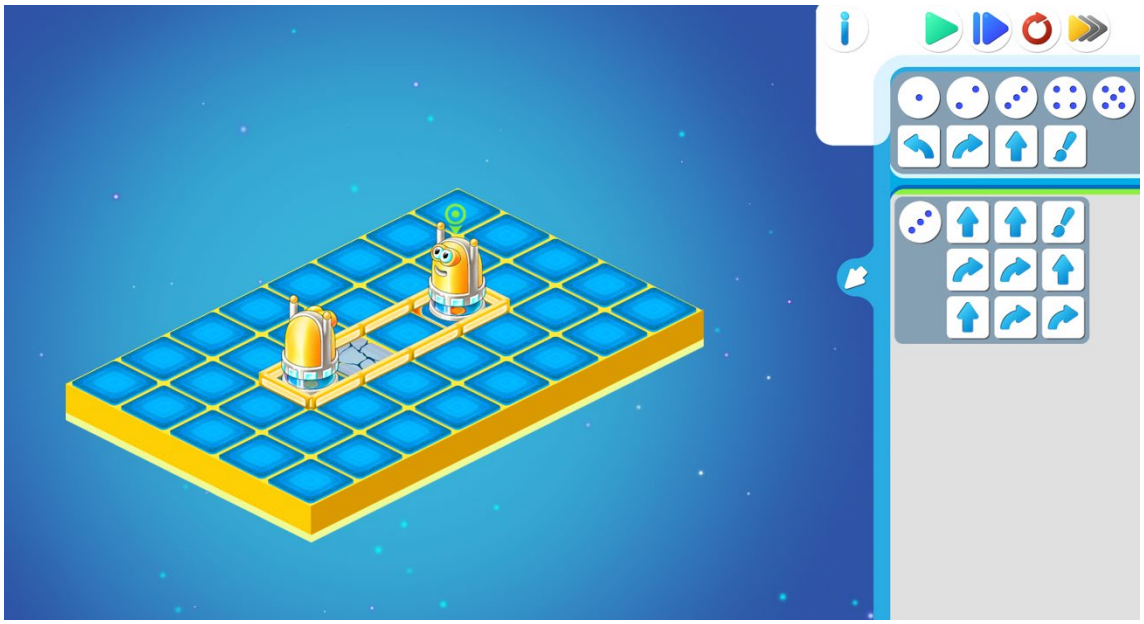

Рис. 7. Исходное состояние и программа для робота, не использующая синхронизацию

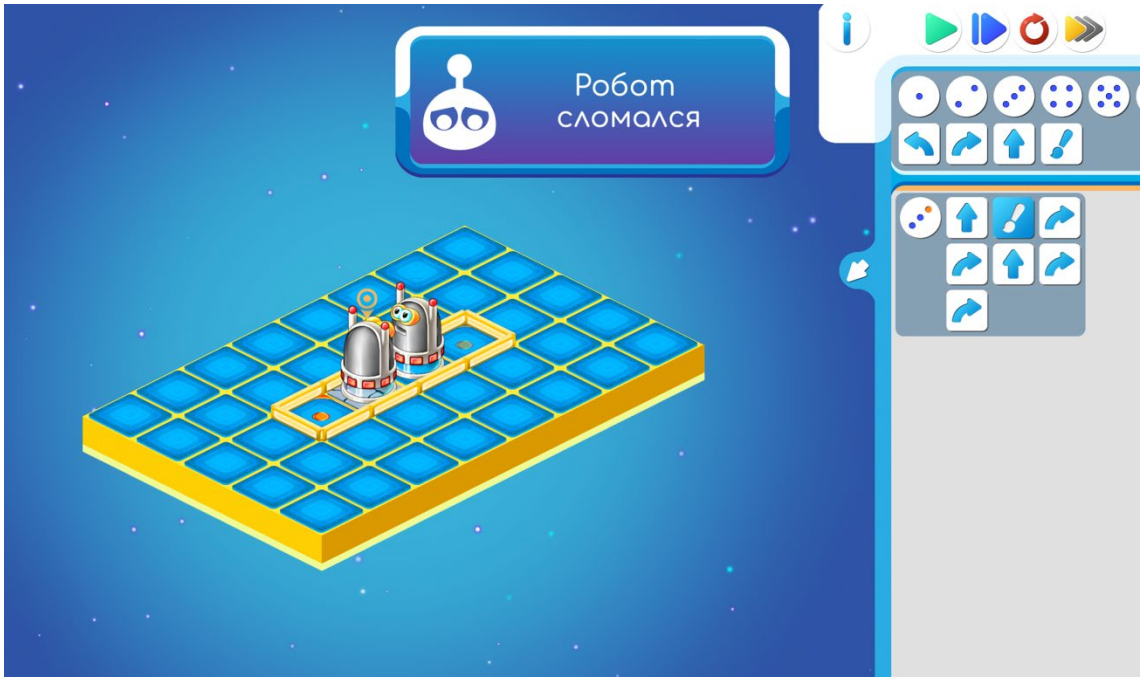

Рис. 8. Роботы столкнулись из-за отсутствия синхронизации

Для ожидания роботы используют дополнительную команду "мигнуть", то есть "пропустить ход", "ничего не делать". Визуально при выполнении команды мигнуть у роботов проскакивает молния между антеннами.

На рис. 9 Роботы красят клетку строго поочередно. Один из роботов занял клетку (находится в своей критической секции), другой выполняет оставшийся код программы, вне критического интервала. Также, как и при реализации алгоритмов синхронизации на Кумире и C++, в программе управления роботом переключается счетчик после покидания критической секции.

Одни робот выбрасывает камень из кувшина, а другой в своей программе кладет камень в кувшин, что соответствует присвоению счетчику 0 или 1. На рис.9 робот, чья программа видна на экране, очищает кувшин.

Рис. 9. Роботы красят клетку строго поочередно.

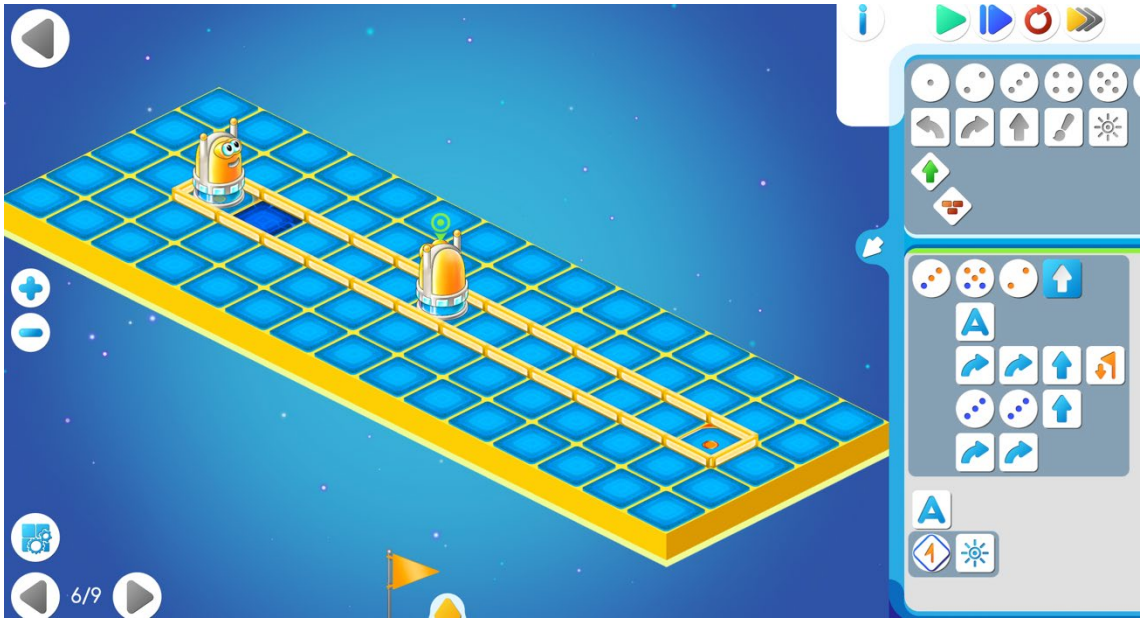

Рис. 10. Решение задачи с флагом.

На рис.10 показана программа поочередного захода на клетку с использованием флага вместо кувшина. Флаг — это битовая (логическая) величина, которая может быть установлена в 0 (флаг опущен) или 1 (флаг поднят). Также можно использовать состояние флага в условии цикла. Обстановка на рисунке демонстрирует низкую эффективность алгоритма синхронизации, поскольку один робот проходит очень длинный путь по сравнению со своим визави. При выполнении программы можно наблюдать робота, находящегося близко от клетки и не имеющего возможности занять совершенно свободную клетку, так как другому роботу требуется пройти только в одну сторону 9 клеток, но составленный алгоритм синхронизации с переключениями предписывает заходить на выделенную клетку строго по очереди: сначала один робот, потом

другой, затем снова один робот, потом снова другой и так далее.

Дальнейшим усовершенствованием алгоритма синхронизации состоит в использовании двух флагов (желтого и зеленого, для черно-белых рисунков левого и правого), поднятие которого сигнализирует о готовности робота занять выделенную клетку.

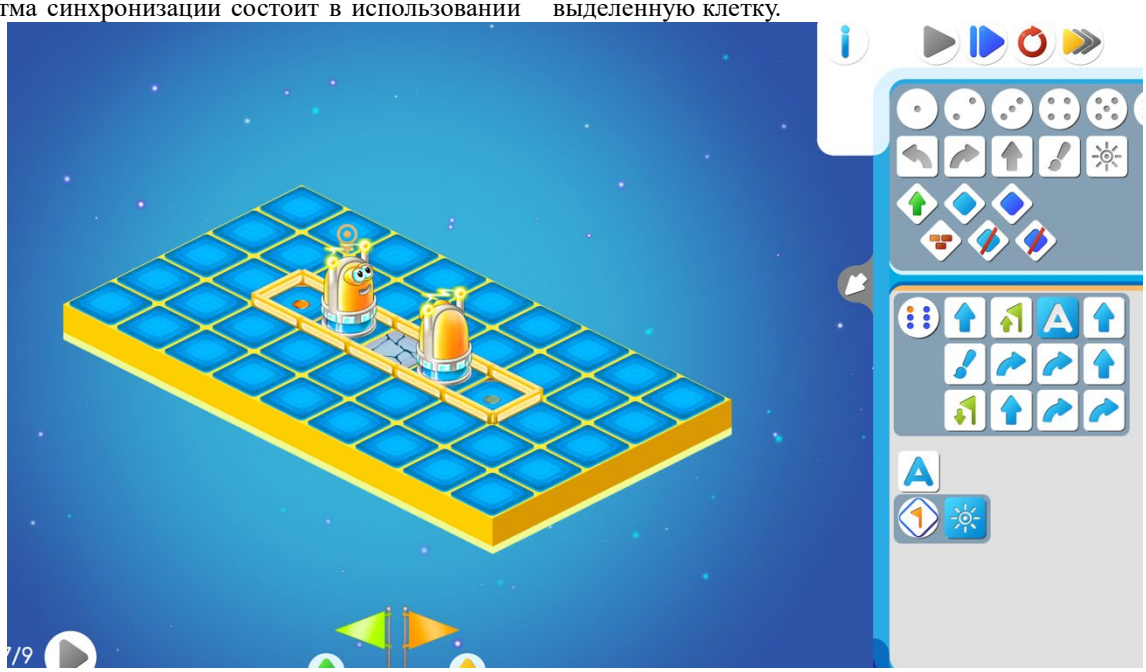

Рис. 11. Роботы находятся в бесконечном ожидании.

Рис. 12. Решение проблемы взаимоисключений.

На рис. 11 роботы находятся в бесконечном ожидании, так как оба были готовы войти в свои критические секции (готовы зайти на клетку), и оба одновременно подняли флаги. При выполнении алгоритма ожидания с проверкой состояния флага (алгоритм А) оба ждут, так как оба флага оказались поднятыми.

На рис. 12 представлено полноценное решение проблемы взаимоисключений, приведен код на языке Пикто, аналогичный соответствующим алгоритмам синхронизации на КуМире и C++. В приведенном решении используется и кувшин (в качестве счетчика, и два флага, назначение которых взято из предыдущего решения. Именно наличие счетчика позволяет избежать проблемы

взаимного блокирования процессов. Для демонстрации решения путь до клетки одного робота в 4 раза превышает путь другого. Таким образом один из роботов чаще приходит в сломанную клетку, чем другой.

Если сравнить размер кода для учебных и производственных языков программирования, то можно с уверенностью сказать, что запись на языке Пикто короче, проще и информативнее.

Рис. 13. Простейшее задание по кооперативному программированию.

5. Заключение

Опыт проведенных занятий со студентами с использованием вышеприведённых алгоритмов синхронизации в ЦОС ПиктоМир для визуализации сложных и важных понятий параллельного программирования, позволяют говорить о повышении эффективности изучения этой важной темы. Студентам впоследствии было легче осваивать методы синхронизации параллельной работы процессов и решать задачи на производственных языках программирования.

Кроме того, картинка результата выполнения задачи в пиктограммной среде программирования ПиктоМир более запоминающаяся и позволяет ученику проще связывать в своем сознании результат работы алгоритма с его содержанием.

Визуализация алгоритмов реализации взаимоисключений ЦОС ПиктоМир стала возможна благодаря внедрений в ПиктоМир режима кооперативно-параллельных заданий. основная идея этого режима состояла в совместном решении поставленных задач двумя и более школьниками, каждый из которых составляет программу для своего робота. Члены команды должны придумать, как согласовать действия своих роботов, договориться с друг другом о порядке выполнения совместной работы.

На рис. 13 представлено простейшее задание для кооперативной работы, когда участникам команды требуется сначала договорится, кто какие клетки красит.

Однако задания по кооперативному программированию можно выполнять и в одиночку, что продемонстрировано изложенным выше решением а ПиктоМире задач на синхронизацию процессов, с использованием графических, материальных аналогов – Кувшин и Флаги – целочисленных и логических переменных.

Задачи на кооперативное программирование в ПиктоМире при решении в одиночку могут восприниматься как головоломки. Такие задания интересны не только школьникам, но и студентам и даже профессиональным программистам. Для этой категории пользователей для синхронизации действий разных роботов в обстановку добавлены исполнитель Кувшин (счетчик) и флаги, доступные во всех потоках параллельной программы. Это позволяет придумывать интересные задачи, над которыми приходится поломать голову даже профессионалам.

Публикация выполнена в рамках государственного задания ФГУ ФНЦ НИИСИ РАН по теме № FNEF-2022-0010.

Visualization of Mutual Exclusion Algorithms by Means of a Pictogram Programming Environment

I.N. Gribanova, A.C. Karavaeva, A.A. Leonov, A.G. Leonov, К.А. Мащенко,V.A. Oganisyan

Abstract. Courses in computer science and information and communication technologies can include a wide range of topics from the basics of programming and elementary knowledge about the structure of computers to the theory of formal grammars, artificial intelligence languages, operating systems, 3D graphics, etc. The main difficulty in this case is the mathematical component of the courses, which requires appropriate training of students. However, in programming there is content that is quite difficult for students to understand, even with deep academic knowledge. This area is parallel programming, when, on the one hand, it is required to study and write educational highly efficient algorithms for solving problems for performing simultaneous calculations on different computers or processors, on the other hand, it is no less difficult to understand - ensuring the parallel operation of various parts of a program with a shared access to shared data. The development of the latter is complicated by the lack of adequate visualization of the parallel operation of processes on a computer. It remains for teachers to illustrate the topic with pictures from the real world, giving various arguments to prove the importance of solving the problem of sharing common resources, citing, as one of the options, the simultaneous free moving of four cars at a crossroad. On the other hand, current trends require lowering the age of the beginning of acquaintance with the elements of programming and algorithmization. Preschoolers, children of the fourth year of life, successfully master the basic concepts of sequential programming using program-controlled robots and their virtual counterparts in the digital educational environment PiktoMir. The same DEE PiktoMir is used for programming propaedeutics in teaching for all ages, including university students. PiktoMir is not only easy to learn, but allows the use of several robot performers in tasks at the same time, each of which acts according to its own program. The visibility of the program execution process in the system led the authors to the idea of using the DEE PiktoMir to demonstrate the operation of classical mutual exclusion algorithms in courses on parallel programming. The experience of the authors showed the high efficiency of this approach, which allowed students to understand the problem in a short time and successfully solve all the problems of the course. This approach was tested in the courses Operating Systems (IIS SUM) and Algorithms and Data Structures (Institute of Childhood, Moscow State Pedagogical University).

Keywords: mutual exclusion, process, deadlock, critical section, PiktoMir, operating systems, algorithm.

Литература

Э. Дейкстра. Взаимодействие последовательных процессов // Языки программирования / Ред. Ф. Женуи, пер. с англ. В.П. Кузнецова, под ред. В.М. Курочкина. — М. : Мир, 1972. — С. 9—87.

E. W. Dijkstra. Solution of a problem in concurrent programming control. Communications of the ACM, 8(9):569, 1965.

Taubenfeld, G. Concurrent Programming, Mutual Exclusion. In: Kao, MY. (eds) Encyclopedia of Algorithms. Springer, New York, 2016.

L. Lamport. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 17(8):453–455, August 1974.

L. Lamport. The mutual exclusion problem: Part II – statement and solutions. Journal of the ACM, 33:327–348, 1986.

L. Lamport. A bug in the Bakery algorithm. Technical Report CA–7704–0611, Massachusette computer associates, inc., April 1977.

U. Abraham. Bakery algorithms. In Proc. of the Concurrency, Specification and Programming Workshop, pages 7–40, 1993.

Taubenfeld, G. The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms. In: Guerraoui, R. (eds) Distributed Computing. DISC 2004. Lecture Notes in Computer Science, vol 3274. Springer, Berlin, Heidelberg, 2004.

E.W. Dijkstra. Cooperating Sequential Processes in Programming Languages. Ed. F. Genuys. — NY: Academic Press, New York, 1968.

Леонов А.Г., Первин Ю.А., Зайдельман Я.Н. Программные исполнители в цифровых образовательных средах «ПиктоМир», «Роботландия» и «КуМир». Информатика в школе. 2019;(9):54-61.

А. Г. Леонов. Параллельное программирование // Энциклопедия для детей. — Т. 22 из Информатика. — Аванта+ Москва, 2003. — С. 140–145.

Framework Qt. Use Qt's libraries and APIs to develop software with native C++ performance for mobile, desktop, and embedded systems. https://qt.io/ (дата обращения 01.11.2022)

Стартовая страница проекта «ПиктоМир» на сайте ФГУ ФНЦ НИИСИ РАН. URL: https://www.niisi.ru/piktomir/ (дата обращения 01.11.2022)

1. V. B. Betelin, A. G. Kushnirenko, A. G. Leonov, K. A.Mashchenko, Basic Programming Concepts as Explained for Preschoolers, International Journal of Education and Information Technologies (NAUN), Volume 15 (2021): 245-255.

2. N.Besshaposhnikov, A.Kushnirenko, and A.Leonov. Piktomir: how and why do we teach textless programming for preschoolers, first graders and students of pedagogical universities. CEE-SECR '17: Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia, October 2017. No. 21. P. 1–7. 2017