О некоторых простых способах синхронизации параллельных программ

А.А. Бурцев[1]

[1]ФГУ ФНЦ НИИСИ РАН, Москва, Россия, burtsev@niisi.msk.ru

Аннотация. Статья посвящена описанию простых способов синхронизации двух параллельных программ, исполняемых на разных вычислительных ядрах одной компьютерной установки, имеющих доступ к общей памяти. Описываемые способы взаимодействия программ можно обеспечить на основе обычных средств, имеющихся почти в каждом языке программирования. И для их реализации не требуется применять какие-либо особые процессорные команды или вызовы специальных функций операционной системы.

Ключевые слова: многоядерные микропроцессоры, способы синхронизации и взаимодействия параллельных программ, рандеву.

1. Введение

Современные высокопроизводительные системы, как правило, имеют в своём составе не один, а несколько микропроцессоров [1], совместно работающих с общей памятью. Это позволяет повысить общую производительность компьютерной установки, так как обеспечивается возможность запускать на ней не одну, а сразу несколько компьютерных программ, которые будут исполняться на ней в одно и то же время, т.е. параллельно.

Но если программы, параллельно исполняемые на разных микропроцессорных ядрах, не являются независимыми, а как-то связаны между собой и призваны вместе решать одну общую задачу, то исполнение таких программ необходимо тщательно синхронизировать, чтобы обеспечить их согласованную работу относительно друг друга.

Для обеспечения синхронизации параллельных программ предлагаются разнообразные средства: семафоры, сигналы, почтовые ящики, мьютексы [2-4]. Такие средства обеспечиваются, как правило, вызовами особых функций операционной системы или специализированной библиотеки. А для их эффективной реализации даже используются специальные команды процессора (test-and-set [5], LL/SC [6]), обеспечивающие исключительный кратковременный монопольный доступ к выделенным позициям памяти.

Однако, в некоторых случаях применять заготовленный арсенал средств синхронизации не представляется возможным или целесообразным. Например, при разработке программы, которая должна будет функционировать на “голой” машине, не оснащённой какой-либо ОС, и/или не имеет возможности использовать специализированные команды доступа к памяти.

Типичным примером такой программы может служить тестовая программа, проверяющая работу подключённых к компьютеру разнообразных периферийных устройств. Подобные тестовые программы должны уметь функционировать ещё до стадии загрузки какой-либо операционной системы. Обычно они должны также удовлетворять ещё и жёстким ограничениям по размеру занимаемой памяти, поэтому при их разработке приходится минимизировать использование стандартных библиотек.

Воспользоваться библиотечными функциями, приготовленными для обеспечения синхронизации, не всегда удаётся ещё по одной причине. Как правило, эти функции содержат в себе цикл ожидания отклика от той программы-партнёра, с которой требуется наладить взаимодействие. Но если такого отклика не последует, то вызванная функция «намертво» застопорит исполнение программы, её вызвавшей. Что как раз таки неприемлемо для функционирования особо критичных программ.

Так, например, тестовые программы, работающие с периферийными устройствами, должны уметь всегда корректно завершаться, невзирая ни на какие исключительные события. Будь то «зависание» устройства по причине его сбоя или критической ошибки, возникшей при передаче очередной порции данных.

Поэтому при разработке такого рода программы желательно обеспечивать синхронизацию работы отдельных её программных компонент, призванных исполняться на разных микропроцессорных ядрах, используя лишь обычные средства традиционного языка программирования высокого уровня.

Далее покажем, как можно обеспечить синхронизацию двух программ, запускаемых на разных ядрах микропроцессорной системы, используя обычные возможности языка Си.

2. Приёмы синхронизации для двух параллельных программ

Для полноценного взаимодействия двух программ, имеющих доступ к общей памяти, достаточно обеспечить им средства исключительного доступа к совместно используемым критическим ресурсам, которые бы не допускали взаимных блокировок. Требуется также обеспечить примитивные средства, которые позволили бы программам регулировать порядок выполнения своих действий относительно связанных с ними действий другого партнёра.

2.1. Критические секции

Для обеспечения взаимно исключительного доступа к критическому ресурсу со стороны двух параллельных программ можно применять известный алгоритм Деккера. Его структурированный вариант, предложенный Дейкстрой, подробно излагается, например, в [4, с.13].

Представим здесь вариант этого алгоритма на языке Си, оформив в виде функций основные его операции: инициализации, входа в критическую секцию и выхода из неё для программ каждого ядра по отдельности:

```
int z0,z1; //флажки запросов ресурса
int pr; // кому разрешается захватить ресурс
void Init(void) {z0=0;z1=0;pr=0}
void Ent0(void) // вход для программы 0-го ядра
{ z0=1;pr=1; while(z1&&(pr==1); }
void Fin0(void){z0=0};//выход для 0-го ядра
void Ent1(void) // вход для программы 1-го ядра
{ z1=1;pr=0; while(z0&&(pr==0); }
void Fin1(void){z1=0};//выход для 1-го ядра
```

Предполагается, что каждая программа соблюдает определённые правила при доступе к критическому ресурсу. Это значит, что программа 0-го ядра при входе в критическую секцию вызывает функцию **Ent0**(), а при выходе из неё − функцию **Fin0**(). А программа 1-го ядра вызывает соответственно функцию **Ent1**() при входе и функцию **Fin1**() при выходе из своей критической секции. И, конечно же, перед использованием критического ресурса сначала вызывается (из любого ядра) операция инициализации **Init**().

2.2. Флажки готовности

Допустим, на разных ядрах параллельно исполняются две программы: **A** на 0-м ядре и **B** на 1-ом ядре. И каждая из них должна выполнить два блока действий: **A1, A2** и **B1, B2**. Но при этом требуется обеспечить, чтобы каждая из них начала исполнять свой второй блок действий только после того, как другая программа закончила исполнение своего первого блока.

В этом случае требуется организовать для этих двух программ такой вид синхронизации, который давно применяется при взаимодействии программ-драйверов с периферийным устройством. Выполнив свою часть работы, устройство сообщает об этом установкой флажка готовности − определённого бита в регистре состояния. А программа-драйвер приостанавливается в цикле ожидания, пока не обнаружит, что требуемый флажок установлен.

Позаимствуем такой приём для обеспечения требуемой синхронизации двух параллельных программ. Пусть каждая из них, выполнив свою первую часть работ, устанавливает свой флажок готовности, а затем дожидается установки флажка готовности другой программы перед тем, как приступить ко второй части своей работы. Приготовим для такой синхронизации соответствующие функции на языке Си:

```
int f0,f1; //флажки готовности программ
void Init0(void) {f0=0;}
void Init1(void) {f1=0;}
void Wait0(void)// ожидание для 0-го ядра
{ f0=1; while(f1==0); }
void Wait1(void)// ожидание для 1-го ядра
{ f1=1; while(f0==0); }
```

Теперь, чтобы обеспечить требуемую синхронизацию, программам 0-го и 1-го ядра следует соблюдать такой порядок действий:

```
    Init(); // предварительная инициализация
// схема для программы 0-го ядра
 Init0();... A1; Wait0(); A2;
// схема для программы 1-го ядра
 Init1();... B1; Wait1(); B2;
```

2.3. Точки встречи

Пусть запускаемые на разных ядрах программы **A** и **B** должны исполнить последовательности из нескольких действий:

```
A1; A2; ... An; // для программы 0-го ядра
B1; B2; ... Bn; // для программы 1-го ядра
```

так, чтобы очередное действие **Aj** программы 0-го ядра совпадало по времени с очередным действием **Bj** программы 1-го ядра (**j**=1,2,…,**n**). Это значит, что для программ **A** и **B** нужно обеспечить такую синхронизацию, чтобы пары действий **Ai** и **Bi** исполнялись бы параллельно:

```
A1||B1; A2||B2; ... An||Bn;
```

Для этого организуем в этих программах своеобразные точки встречи, так называемые **рандеву**. В таких точках одна программа должна дождаться, когда на свою точку встречи подойдёт другая программа, и только после этого они обе смогут продолжать свою дальнейшую работу. Если бы у них предполагалась только одна точка встречи, то для её организации можно было применить описанное ранее решение с флажками готовности.

Но для поставленной здесь задачи синхронизации таких точек встреч у этих двух программ может быть много (**n**). Их потребуется расставить в этих программах перед каждым действием **Aj** и **Bj**, исполнение которых должно начинаться одновременно.

Способ синхронизации с ожиданием флажков готовности распространим на случай не одной, а нескольких точек встречи. Для этого вместо флажков, принимавших только два значения 0 и 1, будем использовать счётчики состояний, которые будут принимать значения от 0 до n. У каждой программы будет свой счётчик состояния, которым она фактически будет сообщать, какую часть назначенной ей работы она уже исполнила.

Будем полагать, что установкой значения **j** в переменную **s0**, представляющую счётчик состояния 0-го ядра, программа **A**, исполняемая на 0-ом ядре, будет сообщать, что она уже исполнила все предыдущие действия и готова приступить к выполнению действия **Aj**. Теперь ей необходимо дождаться момента, когда программа **B** 1-го ядра сообщит, в свою очередь, о готовности к исполнению действия **Bj**, установив в переменной **s1** своего счётчика состояния такое же значение **j**.

Пронумеруем все возможные точки встречи (от 1 до **n**). И выразим описанное правило поведения программ в точке встречи в виде функций с параметром, задающим номер встречи:

```
int s0,s1; // счётчики состояний программ
void Init0(void) {s0=0;}
void Init1(void) {s1=0;}
void Randevu0(int j)// рандеву для 0-го ядра
{ s0=j; while(s1!=j); }
void Randevu1(int j)// рандеву для 1-го ядра
{ s1=j; while(s0!=j); }
```

И покажем схематично, как следует составить программы A и B, чтобы обеспечить им требуемую синхронизацию действий:

```
#define R0(j) Randevu0(j)
#define R1(j) Randevu1(j)
// схема программы A 0-го ядра:
Init0();
 R0(1);A1; R0(2);A2; ... R0(n);An;
// схема программы B 1-го ядра:
Init1();
 R1(1);B1; R1(2);B2; ... R1(n);Bn;
```

Заметим, что во всех этих операциях исполнение программы задерживается в цикле ожидания. Но это непроизводительное ожидание вполне допустимо для подобного рода программ. Ведь в такой ситуации нет необходимости переключать программу на исполнение какой-либо другой работы или вообще передавать занимаемый ею процессор какой-то ещё другой программе. И это как раз характеризует особенность рассмотренных средств синхронизации.

3. Схема организации встречи для нескольких программ

Представленные здесь простые приёмы синхронизации позволяют согласовать совместную работу двух параллельных программ, которые исполнятся на разных процессорных ядрах, но имеют возможность взаимодействовать через общую память. Напрашивается вопрос: а можно ли такие приёмы применить для организации взаимодействия более двух программ?

Прежде, чем ответить на него, заметим, что реализация этих приёмов синхронизации, в основном, опирается на важнейший принцип: изменять значение каждой совместно используемой общей переменной имеет право только одна программа. Только при соблюдении такого принципа можно ухитриться применить описанные здесь приёмы для взаимодействия нескольких программ.

Покажем схематично, как можно было бы обеспечить организацию встречи (рандеву) для более двух программ. Пусть количество таких программ будет **m** (**m**>2), и каждая из них исполняется на одном из **m** микропроцессорных ядер, имеющих доступ к общей памяти.

Для взаимодействия параллельных программ в точках встречи будем использовать общий массив переменных s[m], представляющих счётчики состояний этих программ. Будем полагать, что k-ый элемент этого массива s[k] характеризует состояние программы на k-ом ядре. И только программа, исполняемая на k-ом ядре, имеет право изменять его значение. Остальные программы могут лишь читать это значение.

Представим процедуру рандеву для обеспечения такой «многосторонней» встречи в **j**-ой точке, которая будет вызываться **k**-ом ядром:

```
int s[m]; // счётчики состояний программ
void Init(int k) {s[k]=0;}// инициализация
//рандеву для программы k-го ядра в j-ой т.встречи
void Randevu(int k, int j)
{ int i,f;
  s[k]=j; // отметимся, что пришли на встречу
// дождёмся, когда все придут на j-ую точку встречи
  do{ f=1; i=0;
      while(f&&(i<m)){f=(s[i]==j);i++;}
  }while(f==0);//f=1,если все пришли на встречу
}//Randevu
```

И покажем схематично, как должна быть построена программа, запускаемая на **k**-ом ядре для выполнения последовательности действий **P1**, **P2**, ... **Pn** параллельно с аналогичными действиями программ других ядер:

```
#define Rk(j) Randevu(k,j)
// схема программы P k-го ядра:
Init(k);Rk(1);P1;Rk(2);P2;...Rk(n);Pn;
```

4. Синхронизация программ с учётом аварийного завершения

Все представленные выше приёмы синхронизации страдают серьёзным недостатком: в них не предусмотрены действия на случай возникновения чрезвычайных ситуаций. Если одна из параллельных программ не сможет по какой-либо причине продолжить работу, то взаимодействующая с ней другая программа «зависнет» в одном из циклов, в которых она ожидает отклика от неё, а значит, не сможет даже корректно завершить своё функционирование.

Попробуем исправить приведённые ранее алгоритмы синхронизации, чтобы избавиться от этого недостатка. Продемонстрируем, какие для этого следует внести поправки на примере организации точек встречи для синхронизации двух программ (см. п. 2.3).

Будем полагать, что выполняя каждое действие из последовательности A1, A2, … An, программа A 0-го ядра регулярно проверяет, не случилась ли какая-либо чрезвычайная ситуация, возникновение которой препятствует её дальнейшему нормальному функционированию. И в случае возникновения такой ситуации программа A прекращает свою деятельность, выполняя завершающий блок действий Ax, но перед своим завершением сообщает об этом другой программе так, чтобы она могла тоже отреагировать на возникшую ситуацию.

Известно, что когда подобная ситуация возникает при взаимодействии управляющей программы (драйвера) с периферийным устройством, то устройство сообщает о своём сбое установкой особого флажка ошибки (бита в регистре состояния). Позаимствуем этот приём и будем устанавливать флажок аварийного завершения программы A. A от программы B потребуем, чтобы она прекращала свой цикл ожидания встречи, если этот флажок взведён.

Аналогичный флажок будем устанавливать и при аварийном завершении программы B, а программа A соответственно должна будет прекращать цикл ожидания встречи, если он будет взведён. Выразим эти действия с флажками, скорректировав соответствующие алгоритмы:

```
int Abort0,Abort1;//флажки авар.завершения
int s0,s1; // счётчики состояний программ
void Init0(void){ s0=0; Abort0=0; }
void Init1(void){ s1=0; Abort1=0; }
void Randevu0(int j){ s0=j;
 while(s1!=j) if(Abort1) break; }
void Randevu1(int j){ s1=j;
 while(s0!=j) if(Abort0) break; }
```

Предусмотрим также проверку на необходимость экстренного завершения программы на случай, когда рандеву прекращено без получения ожидаемого отклика от партнёра:

```
#define R0(j) { Randevu0(j);\
 if Abort1 { Abort0=1; Ax; exit(1); }
#define R1(j) { Randevu1(j);\
 if Abort0 { Abort1=1; Bx; exit(1); }
// схема блока действий Aj! программы 0-го ядра:
{ Aj;if(ошибка){Abort0=1;Ax;exit(1);}
// схема программы A 0-го ядра: Init0();
 R0(1);A1!; R0(2);A2!; ... R0(n);An!;
// схема блока действий Bj! программы 1-го ядра:
{ Bj;if(ошибка){Abort1=1;Bx;exit(1);}
// схема программы B 1-го ядра: Init1();
 R1(1);B1!; R1(2);B2!; ... R1(n);Bn!;
```

Теперь в случае возникновения каких-либо аномальных ситуаций каждая из программ корректно завершит работу, выполнив свой заключительный блок действий и сообщив об этом другой программе.

5. Примеры взаимодействия двух параллельных программ

Описанные приёмы синхронизации двух параллельных программ были применены при разработке некоторых программ совместного тестирования периферийных устройств, которые предназначались для запуска на двух разных ядрах микропроцессорной системы. Продемонстрируем их применение на примерах двух тестовых программ, проверяющих правильность функционирования контроллеров Ethernet и DMA в многоядерной микропроцессорной системе MIPS-архитектуры.

5.1. Тест контроллера Ethernet на двух микропроцессорных ядрах

Тестовая программа состоит из двух процедур, каждая из которых запускается на отдельном ядре. Одна процедура ведёт себя как отправитель пакетов, а другая – как получатель пакетов, прогоняемых по внутренней петле одного и того же Ethernet-контроллера.

Процедура-отправитель подготавливает пакеты данных для передачи, создаёт кольцо дескрипторов передатчика, запускает передатчик в работу, после чего входит в цикл передачи пакетов. Процедура-получатель очищает (заполняет пустыми данными) места памяти, куда предполагается заносить принимаемые пакеты данных, создаёт кольцо дескрипторов приёмника, запускает работу приёмника контроллера и затем исполняет цикл приёма пакетов.

В самом начале процедура-получатель инициализирует работу контроллера. Каждая из процедур после отправки или приёма очередного пакета проверяет, не было ли зафиксировано каких-либо ошибок при их передаче. А также отслеживает временной интервал (тай-

маут), отведённый для передачи, а при его истечении сигнализирует об ошибке. После приёма каждого пакета процедура-получатель дополнительно проверяет правильность его приёма-передачи, поэлементно сравнивая массивы принятых данных с теми, которые были отправлены.

Описанные действия исполняются этими процедурами параллельно на двух разных процессорах: процедура-получатель запускается на 0-ом ядре, а процедура-отправитель – на 1-ом ядре. Но выполняться они должны согласованно, т.е. в определённом порядке относительно друг друга, и поэтому процессы их выполнения на разных ядрах требуется определённым образом синхронизовать.

Для этого в этих процедурах предусмотрены так называемые точки встречи (рандеву). Дойдя до очередной точки встречи, процедура одного ядра должна дождаться момента, когда процедура другого ядра тоже подойдёт к соответствующей точке встречи, после чего каждая из процедур может продолжать свои дальнейшие действия на своём ядре, пока не дойдёт до конца или до следующей точки встречи.

В каждой процедуре предусмотрено две точки встречи в начале до цикла приёма/передачи пакетов и по две точки встречи внутри тела этого цикла, а также одна точка встречи после выхода из такого цикла перед завершением процедуры. Расположение этих точек поясним с помощью схематичного представления этих процедур, отразив в них лишь порядок исполнения основных действий.

Схема процедуры-получателя (на 0-ом ядре):

```
----- Процедура-ПОЛУЧАТЕЛЬ (на 0-ом ядре) -----
начало_процедуры
|     Init0();
|  – инициализация Ethernet-контроллера
|     Randevu0(1);
|  – подготовка места памяти для пакетов 0,..,P-1
|  – формирование кольца дескрипторов приёмника
|    с заполнением дескрипторов для пакетов 0,..,P-1
|     Randevu0(2);
|  – запуск приёмника контроллера
| ЦИКЛ_приёма_пакетов n= P,..,Q-1
| |– подготовка места памяти для n-го пакета
| |– ожидание поступления очередного пакета
| |   Randevu0(2*n);
| |– проверка правильности приёма пакета
| |   если (ошибка) то {Abort0=1;break;}
| |   Randevu0(2*n+1);
| |   если (Abort1) то {Abort0=1;break;}
| |– заполнение дескрипторов приёма n-го пакета
| конец_цикла_приёма_пакетов
|– ожидание приёма всех уже отправленных пакетов
|– проверка правильности их приёма-передачи
|– останов приёмника контроллера
|       Randevu0(2*Q);
конец_процедуры
```

Схема процедуры-отправителя, запускаемой на 1-м ядре:

```
----- Процедура-ОТПРАВИТЕЛЬ (на 1-ом ядре) -----
начало_процедуры
|     Init1();
|     Randevu1(1);
|  – заполнение данными пакетов 0,..,P-1
|  – построение кольца дескрипторов передатчика
|    с заполнением дескрипторов для пакетов 0,..,P-1
|     Randevu1(2);
|  – запуск передатчика контроллера
| ЦИКЛ_передачи_пакетов n= P,..,Q-1
| |– подготовка (заполнение данными) n-го пакета
| |– ожидание конца передачи очередного пакета
| |   Randevu1(2*n);
| |– проверка правильности передачи пакета
| |   если (ошибка) то {Abort1=1;break;}
| |   Randevu1(2*n+1);
| |   если (Abort0) то {Abort1=1;break;}
| |– заполнение дескрипторов передачи n-го пакета
| конец_цикла_передачи_пакетов
|– ожидание передачи всех отправленных пакетов
|– останов передатчика контроллера
|       Randevu1(2*Q);
конец_процедуры
```

В этих схемах процедур используются параметры **P** и **Q**, которые определяются в программе теста как define-переменные:

```
#define P  14 /* кол-во пакетов в кольце, 1<P<Q */
#define Q  32768 /* кол-во пакетов для передачи */
```

Они задают общее количество передаваемых в тесте пакетов (**Q**), а также размер колец (**P**) дескрипторов, формируемых для приёма-передачи пакетов.

5.2. Одновременное тестирование контроллеров DMA и Ethernet

Чтобы поверить работоспособность контроллеров Internet и DMA при их совместной работе с памятью, было предложено запускать программы их тестирования одновременно на разных ядрах. И согласовать их работу так, чтобы пересылка очередного блока пакетов данных контроллером Internet с одного участка основной памяти в другой (по внутренней петле) совпадала по времени с DMA-передачей по основной памяти блока данных примерно того же размера.

Для реализации такого тестирования в программах тестов контроллера Ethernet и контроллера DMA были организованы точки встречи для обеспечения их синхронизации с использованием приёмов, рассмотренных ранее в п. 2.3 и 4. Расположение таких точек встречи (рандеву) поясняется с помощью схематичного представления алгоритмов этих программ.

Порядок действий программы теста Ethernet-контроллера, запускаемой на 0-ом ядре, проиллюстрируем следующей схемой:

```
 ----- Программа теста Ethernet-контроллера -----
начало_программы
|     Init0();
|     Randevu0(1);
| - инициализация Ethernet-контроллера
|     Randevu0(2);
| ЦИКЛ_приёма-передачи_блока_пакетов r= 0,..,R-1
| |- подготовка данных r-го блока пакетов
| |- очистка памяти для приёма r-го блока пакетов
| |- формирование кольца дескрипторов для
|    передачи P пакетов r-го блока
| |- формирование кольца дескрипторов для
|    приёма P пакетов r-го блока
| |    Randevu0(3+r);
| |    если (Abort1) то {Abort0=1;break;}
| |- запуск приёма-передачи блока пакетов
| |- ожидание поступления всех пакетов блока
| |- останов приёма-передачи блока пакетов
| |- проверка корректности передачи блока пакетов
| |    если (ошибка) то {Abort0=1;break;}
| конец_цикла_приёма-передачи_блоков_пакетов
|       Randevu0(3+R);
|- поэлементная проверка правильности передачи
|   всех переданных и принятых блоков пакетов
|       Randevu0(4+R);
конец_программы
```

Данная программа после инициализации своего контроллера входит в цикл передачи R блоков, состоящих каждый из P пакетов. В начале каждого шага цикла она приготавливает данные для передачи P пакетов и формирует для них кольца дескрипторов приёмника и передатчика. И после очередного рандеву, дождавшись от программы-партнёра готовности к DMA-передаче, запускает передачу всех приготовленных P пакетов.

Далее она ожидает завершения этой передачи (или её прекращения из-за возникшей критической ошибки или истечения таймаута). После чего проверяет, прошла ли передача всех P пакетов блока без ошибок. И в случае обнаружения ошибки прекращает цикл передачи блоков пакетов и сообщает об этом программе-партнёру установкой своего флажка аварийного завершения.

Аналогично построена и программа теста контроллера DMA. После инициализации контроллера и прохождения совместного рандеву она исполняет цикл для DMA-передачи R блоков данных. В начале каждого шага цикла она подготавливает данные для передачи, настраивает регистры и дескрипторы всех задействованных DMA-каналов. После чего выходит на точку встречи для синхронизации своих действий с программой другого ядра.

После проведения рандеву запускает подготовленную DMA-передачу, затем ожидаёт её завершения и проверяет, прошла ли она без ошибок. В случае возникновения ошибки прекращает все последующие DMA-передачи и сигнализирует об этом программе-партнёру.

Описанный порядок действий программы теста DMA-контроллера, запускаемой на 1-ом ядре, проиллюстрируем следующей схемой:

```
 ----- Программа теста DMA-контроллера -----
начало_программы
|     Init1();
|     Randevu1(1);
| - инициализация DMA-контроллера
|     Randevu1(2);
| ЦИКЛ_передачи_блока_данных r= 0,..,R-1
| |- подготовка данных r-го блока
| |- очистка памяти для приёма r-го блока данных
| |- формирование DMA-дескрипторов для
| |- передачи данных r-го блока
| |    Randevu1(3+r);
| |    если (Abort0) то {Abort1=0;break;}
| |- запуск DMA-передачи блока данных
| |- ожидание завершения DMA-передачи
| |- останов DMA-передачи
| |- проверка корректности передачи блока данных
| |    если (ошибка) то {Abort1=1;break;}
| конец_цикла_DMA-передачи_блоков_данных
|       Randevu1(3+R);
|- поэлементная проверка правильности передачи
|   всех переданных DMA блоков данных
|       Randevu1(4+R);
конец_программы
```

В представленных схемах программ тестирования контроллеров Ethernet и DMA используются параметры **P** и **R**, которые определяются в этих программах как define-переменные:

```
#define P 14 /* кол-во пакетов в кольце, 1<P<Q */
#define R 1000 /* кол-во блоков для передачи */
```

Они задают общее количество блоков пакетов (**R**), передаваемых в ходе тестирования по Ethernet и DMA-каналам, а также размер колец дескрипторов, формируемых для приёма-передачи контроллером Ethernet одного блока из **P** пакетов.

6. Заключение

В статье представлены простые способы синхронизации двух параллельных программ, которые были успешно применены для разработки тестов, запускаемых одновременно на двух разных ядрах микропроцессорной системы MIPS-архитектуры для проверки совместной работы контроллеров Ethernet и DMA.

Публикация выполнена в рамках государственного задания ФГУ ФНЦ НИИСИ РАН по теме № FNEF-2024-0003 «Методы разработки аппаратно-программных платформ на основе защищенных и устойчивых к сбоям систем на кристалле и сопроцессоров искусственного интеллекта и обработки сигналов».

About Some Simple Techniques to Synchronize Parallel Programs

A.A. Burtsev

Abstract. The article is devoted to the description of simple methods of synchronizing two parallel programs executed on different computing cores of the same computer installation that have access to common memory. The described methods of interaction of programs can be provided on the basis of conventional means available in almost every programming language. And their implementation does not require the use of any special processor instructions or calls to special functions of the operating system.

Keywords: multi-core microprocessors, methods of synchronization and interaction of parallel programs, rendezvous.

Литература

1. В.В. Корнеев, А.В. Киселёв. Современные микропроцессоры. М.: НОЛИДЖ, 2000.
2. А.В. Гордеев, А.Ю. Молчанов. Системное программное обеспечение. СПб: Питер, 2002. с. 221-300.
3. М. Митчел, Д. Оулдем, А. Самьюэл. Программирование для Linux. Профессиональный подход. М.: Вильямс, 2003. с. 95-120.
4. А.А. Бурцев. Параллельное программирование. Учебное пособие по курсу «Операционные системы». Обнинск, ИАТЭ, 1994.
5. Википедия. Test-and-set, https://ru.wikipedia.org/wiki/Test-and-set
6. Википедия. Load-link/store-conditional, https://en.wikipedia.org/wiki/Load-link/store-conditional