БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра многопроцессорных систем и сетей

М. К. Буза, О. М. Кондратьева

ПРОЕКТИРОВАНИЕ ПРОГРАММ ДЛЯ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ

Учебно-методическое пособие
для студентов
факультета прикладной математики и информатики

МИНСК
2013

УДК 004.31(075.8)
ББК 32.973.26-04я73-1
Б90

Утверждено на заседании
кафедры многопроцессорных систем и сетей
8 октября 2013 г., протокол № 3

Буза, М. К.
Б90 Проектирование программ для многоядерных процессоров: учеб.-метод. пособие для студентов факультета прикладной математики и информатики / М. К. Буза, О. М. Кондратьева. – Минск: БГУ, 2013. – 48 с.

В пособии рассматриваются основные аспекты создания многопоточных приложений для многоядерных процессоров.

Предназначено для студентов факультета прикладной математики и информатики.

УДК 004.31(075.8)
ББК 32.973.26-04я73-1

© Буза М. К., Кондратьева О. М., 2013
© БГУ, 2013

ВВЕДЕНИЕ

В данном пособии рассматриваются основные аспекты разработки многопоточных приложений на языках C/C++ с использованием Windows API и POSIX Thread Library. Многопоточность, поддерживаемая современными операционными системами, позволяет создавать параллельные программы с общей памятью. При разработке параллельных приложений возникают специфические для параллельной модели программирования проблемы, разрешению ряда из них посвящено данное пособие.

Содержание пособия отражает наиболее трудный раздел «Многопоточные приложения как параллельные программы с общей памятью» учебной программы общей дисциплины «Распределенные и параллельные системы» для студентов специальности «Прикладная информатика» факультета прикладной математики и информатики БГУ.

Многопоточные программы требуют создания параллельной модели и позволяют использовать низкий уровень применяемых технологий. Главными достоинствами многопоточных программ являются: малые накладные расходы и легкость переноса в другие операционные системы, реализующие многопоточность в системах параллельного действия [1], в частности в многоядерных компьютерах [2].

Учебный материал в пособии изложен таким образом, что после его изучения студент может начать разрабатывать параллельные программы. Более глубокое освоение предмета требует изучения рекомендуемой литературы.

1. ВСТРОЕННЫЕ ПОТОКИ WINDOWS

Операционная система Microsoft Windows позволяет разрабатывать многопоточные приложения на языке C/C++ с помощью прикладного программного интерфейса Microsoft Windows API (Application Program Interface). Рассмотрим основные возможности Windows API (WinAPI).

Подробную информацию можно найти, например, в [3] или в справочных материалах [4].

1.1. Предварительные замечания

Начнем с процесса ***обработки ошибок*** в WinAPI.

После вызова функция WinAPI проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если функция получила недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, она возвращает значение, свидетельствующее об

ошибке, В таблице 1 представлены типы данных для результатов функций WinAPI и их возможные значения.

Таблица 1

Стандартные типы значений, возвращаемых функциями WinAPI

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно.
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение не равно 0.
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL, в остальных случаях HANDLE идентифицирует объект. Осторожно: некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным 1. В документации для каждой функции указано, что именно она возвращает при ошибке.
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL, в остальных случаях PVOID – адрес блока данных в памяти.
LONG или DWORD	Функции, которые сообщают значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не смогла выполнить то, что должна, она обычно возвращает 0 или -1 (зависит от конкретной функции).

При возникновении ошибки приходится анализировать, почему вызов данной функции оказался неудачен. За каждой ошибкой закреплен свой код – 32-битное число.

Функция WinAPI, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. Это позволяет потокам работать независимо друг от друга, не обращая внимание на чужие ошибки. Когда функция возвращает управление, ее возвращаемое значение будет указывать, что произошла ошибка. Код ошибки можно узнать, вызвав функцию *GetLastError()*:

```
DWORD WINAPI GetLastError(
  void
);
```

Возвращаемое значение – код последней ошибки для данного потока. *GetLastError()* возвращает последнюю ошибку, возникшую в потоке, поэтому ее нужно вызывать сразу после неудачного вызова системной функции.

Список кодов ошибок, определенных Microsoft, содержится в заголовочном файле WinError.h. В Visual Studio есть утилита Error Lookup, которая позволяет узнать описание ошибки по ее коду.

Объекты ядра. Объекты ядра используются операционной системой и приложениями для управления ресурсами: процессами, потоками, файлами, событиями и многими другими.

Объект ядра представляет собой блок памяти, выделенный ядром и доступный только ему. Этот блок – структура данных, в элементах которой содержится информация об объекте. Одни элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, другие – специфичны для объектов конкретного типа.

В WinAPI предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции.

Функция, создающая объект ядра, возвращает дескриптор созданного объекта. Дескриптор однозначно идентифицирует объект и используется в системных функциях. Этот дескриптор может быть использован любым потоком процесса, владеющего этим объектом.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. В момент создания объекта счетчику присваивается единичное значение. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на единицу. Если какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на единицу. Как только счетчик объекта ядра обнуляется, объект уничтожается.

Объекты ядра можно защитить дескриптором защиты, который указывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используется при написании серверных приложений. Создавая клиентское приложение можно игнорировать это свойство объектов ядра.

По окончании работы с объектом ядра необходимо выполнить операцию закрытия объекта вызовом функции *CloseHandle()*:

```
BOOL CloseHandle(
  _In_  HANDLE hObject
);
```

Параметры функции:

— *hObject* – действительный дескриптор объекта ядра.

Возвращаемое значение. Если функция отработала успешно, то она вернет ненулевое значение, и 0 – в противном случае.

Функция *CloseHandle()* закрывает дескрипторы следующих объектов: процессы, потоки, мьютексы, семафоры, события и другие.

1.2. Функция потока

После создания поток начинает выполнять потоковую функцию – обычная функция, которая содержит код потока. Поэтому для каждого вторичного потока должна быть определена потоковая функция – точка входа потока:

```
DWORD WINAPI ThreadFunction(LPVOID lpParam)
{
  DWORD dwResult = 0;
  // действия, выполняемые функцией
  return dwResult;
}
```

Следует помнить, что:

— можно использовать единственный параметр функции для передачи данных потоку; смысл параметра определяется программистом;

— если функция первичного потока имеет строго определенное имя, то функцию вторичного потока можно назвать как угодно;

— статические и глобальные переменные доступны всем потокам процесса;

— параметры и локальные переменные потоковой функции создаются в стеке потока;

— функции потоков, как и любые другие функции, должны по возможности обходиться своими параметрами и локальными переменными;

— значение, возвращаемое функцией потока, используется как код завершения потока.

1.3. Создание потока

Для создания потока в адресном пространстве процесса используется функция *CreateThread()*:

```
HANDLE WINAPI CreateThread(
  _In_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_      SIZE_T dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt   LPVOID lpParameter,
  _In_      DWORD dwCreationFlags,
  _Out_opt  LPDWORD lpThreadId
);
```

Параметры функции:

— *lpThreadAttributes* – указатель на структуру *SECURITY_ATTRIBUTES*; будем использовать значение *NULL* (новый поток получит атрибуты защиты по умолчанию);

— *dwStackSize* – начальный размер стека потока в байтах; будем использовать значение 0 (новый поток получит размер по умолчанию);

— *lpStartAddress* – указатель на функцию потока; можно создавать множество потоков с одной и той же функцией потока;

— *lpParameter* – параметр, который будет передан функции потока;

— *dwCreationFlags* – флаги, управляющие созданием потока; будем использовать одно из двух значений: 0 (поток выполняется сразу после создания) или *CREATE_SUSPENDED* (поток создается в приостановленном состоянии);

— *lpThreadId* – указатель на переменную, в которой функция *CreateThread()* вернет идентификатор нового потока; можно использовать значение *NULL*, если идентификатор потока не требуется.

Возвращаемое значение. Если функция отработала успешно, то она вернет дескриптор нового потока, и *NULL* – в противном случае.

Замечания:

1. Количество потоков, которое может создать процесс, ограничено доступной виртуальной памятью. Однако с точки зрения эффективности приложения лучше, если количество потоков соответствует количеству процессоров вычислительной системы.

2. Если поток вызывает функции из C run-time library, то для его создания следует использовать функцию *_beginthreadex()*, а не *CreateThread()*.

Пример 1_1. Создание потока. Параметр, передаваемый функции потока, не используется.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

DWORD WINAPI ThreadFunction(LPVOID)
{
  printf("I am Thread\n");
  return 0;
}
int main()
{
  // создание дочернего потока
  HANDLE hThread = (HANDLE)_beginthreadex(NULL, 0, Thread-
Function, NULL, 0, NULL);
  if (hThread==NULL)  // обработка ошибки
  {
       printf("Create Thread Error=%d\n", GetLastError());
```

```
        return -1;
    }
    // ожидание завершения дочернего потока
    WaitForSingleObject(hThread, INFINITE);
    // закрытие объекта
    CloseHandle(hThread);
    return 0;
}
```

1.4. Завершение потока

Поток можно завершить одним из четырех способов:

— функция потока возвращает управление;

— поток самоуничтожается вызовом функции *ExitThread()*;

— один из потоков данного или стороннего процесса вызывает функцию *TerminateThread()*;

— завершается процесс, содержащий данный поток.

Первый способ – функция потока возвращает управление – это единственный способ, который гарантирует корректное освобождение всех ресурсов, принадлежавших потоку. Поэтому функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление.

Пример 1_2. Структура функции потока, которую можно завершить «принудительно», но корректно. Флаг *bTerminate* используется, чтобы просигнализировать потоку о том, что он должен прекратить работу. Этот флаг может быть общим для всех потоков, либо свой для каждого потока (группы потоков).

```
// Глобальный флаг для управления завершением потока
bool bTerminate=false;
DWORD WINAPI ThreadFunction( LPVOID lpParam )
{
  while (!bTerminate)
  {
       // действия
  }
}
```

При необходимости завершить поток следует флаг *bTerminate* установить в *true* и дождаться завершения потока.

Чтобы узнать завершен ли поток и код его завершения, используется функция *GetExitCodeThread()*:

```
BOOL WINAPI GetExitCodeThread(
  _In_      HANDLE hThread,
  _Out_     LPDWORD lpExitCode
);
```

Параметры функции:

— *hThread* – дескриптор потока;

— *lpExitCode* – указатель на переменную, в которой функция *GetExitCodeThread()* вернет код завершения потока.

Возвращаемое значение. Если функция отработала успешно, то она вернет ненулевое значение, и 0 – в противном случае.

Замечания:

1. Если поток не завершен на момент вызова *GetExitCodeThread()*, то во втором параметре будет передано значение *STILL_ACTIVE*.

2. Если поток был завершен и *GetExitCodeThread()* отработала успешно, то во втором параметре будет передано значение, которое возвращает функция потока.

1.5. Передача данных потоку

Единственный параметр функции потока имеет тип *void** (указатель на *void*). Указатель на *void* называется указателем обобщенного типа и имеет замечательное свойство: любой указатель можно привести к типу *void** и обратно без потери информации. Более того, корректное приведение можно выполнить между типами *void** и *int*.

Если необходимо передать функции потока много аргументов, то можно создать структуру, которая содержит все аргументы, и передать указатель на эту структуру. Лучше всего создать для каждой потоковой функции собственную структуру.

Пример 1_3. Многопоточная программа для вычисления числа π. Число π можно определить из формулы:

$$\int_0^1 \frac{dx}{1+x^2} = \operatorname{arctg} 1 = \frac{\pi}{4}.$$

Определенный интеграл будем вычислять приближенно по формуле левых прямоугольников:

$$\int_a^b f(x) \approx h \sum_{i=0}^{n-1} f(x_i), \quad x_i = ih, \quad h = (b-a)/n.$$

Между *p* потоками задачу разделим следующим образом:

$$\int_0^1 \frac{dx}{1+x^2} \approx \sum_{k=0}^{p-1} \left(h \sum_{i=0}^{k+ip<n} \frac{1}{1+x_{k+ip}^2} \right), \quad x_j = jh, \quad h = 1/n.$$

Поток с номером *k* вычисляет сумму слагаемых с номерами *k*, *k+p*, *k+2p* и так далее; сохраняет вычисленное значение в элементе глобаль-

ного массива по индексу *k*. Значение *k* передается в функцию потока через параметр.

Потоки осуществляют доступ к глобальному массиву, но каждый – к своему элементу, поэтому синхронизация доступа не требуется.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

#define p 2    // количество дочерних потоков
double pi[p];
int n = 1000000;
DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
     int nParam = (int)pvParam;
     int i, start;
     double h, sum, x;
     h = 1./n;
     sum = 0.;
     start = nParam;
     for (i=start; i<n; i+=p)
     {
          x = h * i;
          sum += 4. / (1. + x*x);
     }
     pi[nParam] = h * sum;
     return 0;
}
int main()
{
     HANDLE hThreads[p];
     int k;
     double sum;
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          hThreads[k] = (HANDLE)_beginthreadex(NULL, 0,
ThreadFunction, (LPVOID)k, 0, NULL);
          if (hThreads[k]==NULL)   // обработка ошибки
          {
               printf("Create Thread %d Error=%d\n", k, Get-
LastError());
               return -1;
          }
     }
     // ожидание завершения дочерних потоков
     WaitForMultipleObjects(p, hThreads, TRUE,INFINITE);
     for (k=0; k<p; ++k)
```

```
        CloseHandle(hThreads[k]);

    sum = 0.;
    for (k=0; k<p; ++k)
        sum += pi[k];
    printf("PI = %.16f\n", sum);
    return 0;
}
```

Пример 1_4. Вычисление числа π. Реализуем передачу потоку нескольких аргументов через структуру. Каждый поток получает свой экземпляр структуры. Потоковой функции передается указатель на структуру с полями: *k* – порядковый номер потока и *sum* – поле, в котором функция вернет вычисленное значение.

Каждый поток осуществляет доступ к своему экземпляру структуры, поэтому синхронизация доступа не требуется.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

#define p 2    // количество дочерних потоков
int n = 1000000;
// тип параметра, передаваемого функции потока
struct SThreadParam
{
    int k;
    double sum;
};

DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
    SThreadParam* param = (SThreadParam*)pvParam;
    int i, start;
    double h, sum, x;
    h = 1./n;
    sum = 0.;
    start = param->k;
    for (i=start; i<n; i+=p)
    {
        x = h * i;
        sum += 4. / (1. + x*x);
    }
    // к глобальной переменной не следует часто обращаться
    param->sum = h * sum;
    return 0;
}
```

```
int main()
{
     HANDLE hThreads[p]; // массив дескрипторов потоков
     // массив параметров потоковых функций
     SThreadParam params[p];
     int k;
     double sum;
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          params[k].k = k;
          hThreads[k] = (HANDLE)_beginthreadex(NULL, 0,
ThreadFunction, (LPVOID)&(params[k]), 0, NULL);
          if (hThreads[k]==NULL)   // обработка ошибки
          {
               printf("Create Thread %d Error=%d\n", k, Get-
LastError());
               return -1;
          }
     }
     // ожидание завершения дочерних потоков
     WaitForMultipleObjects(p, hThreads, TRUE,INFINITE);
     for (k=0; k<p; ++k)
          CloseHandle(hThreads[k]);

     sum = 0.;
     for (k=0; k<p; ++k)
          sum += params[k].sum;
     printf("PI = %.16f\n", sum);
     return 0;
}
```

Пример 1_5. Пример некорректной передачи аргументов. Каждому дочернему потоку передается адрес переменной *k*, которая находится в разделяемой памяти и доступна главному и всем дочерним потокам. Главный поток изменяет значение этой переменной. Ниже приведен фрагмент создания дочерних потоков, аналогичный Примеру 1_3, но передается не значение, а адрес переменной *k*.

```
// создание дочерних потоков
for (k=0; k<p; ++k)
{
     hThreads[k] = (HANDLE)_beginthreadex(NULL, 0, Thread-
Function, (LPVOID)&k, 0, NULL);
…
}
```

1.6. Синхронизация потоков в пользовательском режиме

Операционная система Windows лучше всего работает, когда потоки не взаимодействуют друг с другом. Однако такая ситуация встречается крайне редко. Потоки взаимодействуют друг с другом в двух основных случаях:

— совместно используя разделяемый ресурс (чтобы не разрушить его);

— когда нужно уведомлять другие потоки о завершении каких-либо операций.

В WinAPI есть достаточно средств, которые можно использовать для синхронизации потоков.

Критические секции. Критическая секция – это небольшой участок кода, требующий взаимоисключающего доступа к определенному ресурсу. Это означает, что только один поток может получить доступ к ресурсу. Ни один из потоков, которым нужен занятый ресурс, не получит процессорное время до тех пор, пока ресурс не будет освобожден.

Потоки одного процесса могут использовать объект «Критическая секция» для взаимоисключающего доступа к разделяемым ресурсам. Для этого необходимо создать экземпляр структуры *CRITICAL_SECTION* и инициализировать его перед первым использованием. Обычно структуры *CRITICAL_SECTION* создаются как глобальные переменные, доступные всем потокам процесса. Для инициализации один из потоков процесса должен вызвать функцию *InitializeCriticalSection()*:

```
void WINAPI InitializeCriticalSection(
  _Out_  LPCRITICAL_SECTION lpCriticalSection
);
```

Параметры функции:

— *lpCriticalSection* – указатель на переменную типа CRITICAL_SECTION.

После инициализации критической секции потоки процесса могут ее использовать для получения монопольного доступа к ресурсу.

Участок кода, работающий с разделяемым ресурсом, предваряется вызовом функции *EnterCriticalSection()*:

```
void WINAPI EnterCriticalSection(
  _Inout_  LPCRITICAL_SECTION lpCriticalSection
);
```

Функция *EnterCriticalSection()* исследует значения элементов переданной ей структуры и выполняет следующие действия:

— если ресурс свободен, *EnterCriticalSection()* модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс,

после чего немедленно возвращает управление, и поток продолжает свою работу, получив доступ к ресурсу;

— если ресурс занят другим потоком, *EnterCriticalSection()* переводит вызывающий поток в состояние ожидания; как только поток, занимавший ресурс, освободит его, вызывающий поток получает ресурс и переводится в состояние готовности.

В конце участка кода, использующего разделяемый ресурс, поток должен сообщить об освобождении ресурса, вызвав функцию *LeaveCriticalSection()*:

```
void WINAPI LeaveCriticalSection(
  _Inout_  LPCRITICAL_SECTION lpCriticalSection
);
```

Когда критическая секция больше не нужна ни одному потоку, следует освободить все занятые ею ресурсы. Для выполнения этой операции любой из потоков должен вызвать функцию *DeleteCriticalSection()*:

```
VOID DeleteCriticalSection(
  _Inout_  LPCRITICAL_SECTION lpCriticalSection
);
```

Пример 1_6. Рассмотрим вариант вычисления числа π, когда функция потока получает свой индекс *k*. Функция вычисляет частичную сумму и прибавляет ее к значению глобальной переменной *pi*, разделяемой всеми потоками. Используется критическая секция для взаимоисключающего доступа к переменной *pi*.

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

int p = 2;          // количество дочерних потоков
int n = 1000000;
double pi = 0.;     // требуется взаимоисключающий доступ
CRITICAL_SECTION cs;

DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
     int nParam = (int)pvParam;
     int i, start;
     double h, sum, x;
     h = 1./n;
     sum = 0.;
     start = nParam;
     for (i=start; i<n; i+=p)
     {
          x = h * i;
          sum += 4. / (1. + x*x);
```

```
     }
     // монопольный доступ
     EnterCriticalSection(&cs);
     pi += h * sum;
     LeaveCriticalSection(&cs);
     return 0;
}
int main()
{
     HANDLE hThreads[p];
     int k;
     InitializeCriticalSection(&cs);
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          hThreads[k] = (HANDLE)_beginthreadex(NULL, 0,
ThreadFunction, (LPVOID)k, 0, NULL);
          if (hThreads[k]==NULL)   // обработка ошибки
          {
               printf("Create Thread %d Error=%d\n", k, Get-
LastError());
               return -1;
          }
     }
     // ожидание завершения дочерних потоков
     WaitForMultipleObjects(p, hThreads, TRUE,INFINITE);
     for (k=0; k<p; ++k)
          CloseHandle(hThreads[k]);
     // освобождение ресурсов, занятых критической секцией
     DeleteCriticalSection(&cs);
     printf("PI = %.16f\n", pi);
     return 0;
}
```

Несколько полезных приемов, которые следует применять при работе с критическими секциями:

— на каждый разделяемый ресурс использовать отдельную структуру *CRITICAL_SECTION*;

— при одновременном доступе к нескольким ресурсам вызывать функции *EnterCriticalSection()* в одинаковом порядке;

— не занимать критические секции надолго.

Критические секции осуществляют синхронизацию потоков без перехода в режим ядра. Замечательная особенность такой синхронизации – высокое быстродействие. Поэтому, если возможно, то следует обойтись синхронизацией в пользовательском режиме.

1.7. Синхронизация потоков с использованием объектов ядра

Механизмы синхронизации потоков с помощью объектов ядра предоставляют гораздо больше возможностей, чем механизмы синхронизации в пользовательском режиме. Их недостаток – меньшее быстродействие.

Мы уже использовали объект ядра «поток» для синхронизации, когда главный поток ожидал завершения дочерних потоков. Этот объект ядра пригоден для синхронизации благодаря следующему свойству. Объект ядра «поток» сразу после создания находится в занятом состоянии и переходит в свободное состояние после завершения потока.

В WinAPI есть множество объектов ядра, которые могут находиться в занятом или свободном состоянии и использоваться для синхронизации. Среди них: потоки, события, семафоры, мьютексы.

Wait-функции. *Wait*-функции позволяют потоку приостановиться и ждать, когда какой-либо объект ядра перейдет в свободное состояние или истечет время ожидания. Чаще всего используется функция *WaitForSingleObject()*:

```
DWORD WINAPI WaitForSingleObject(
  _In_  HANDLE hObject,
  _In_  DWORD dwMilliseconds
);
```

Параметры функции:

— *hObject* – дескриптор объекта ядра, поддерживающий состояния «свободен-занят»;

— *dwMilliseconds* – время в миллисекундах, в течение которого поток готов ждать освобождения объекта; значение *INFINITE* – поток готов ждать этого события неограниченное время; в параметре можно передать 0, и тогда *WaitForSingleObject()* немедленно вернет управление.

Возвращаемое значение указывает, почему вызывающий поток снова стал планируемым. Функция может вернуть одно из следующих значений:

— *WAIT_OBJECT_0* – ожидаемый объект перешел в свободное состояние;

— *WAIT_TIMEOUT* – истекло время ожидания, а объект не перешел в свободное состояние;

— *WAIT_FAILED* – вызов функции закончился неудачно.

Пример 1_7. Пример иллюстрирует, как вызывать функцию *WaitForSingleObject()* со значением времени ожидания, отличным от *INFINITE*.

```
DWORD dwWaitResult = WaitForSingleObject(hThread, 5000);
switch (dwWaitResult)
{
case WAIT_OBJECT_0:
     // поток завершился
     break;
case WAIT_TIMEOUT:
     // поток не завершился в течение 5000 мс
     break;
case WAIT_FAILED:
     // неправильный вызов функции (неверный дескриптор?)
     break;
}
```

Функция *WaitForMultipleObjects()* позволяет ждать до тех пор, пока сразу несколько объектов или какой-то один из списка объектов, перейдут в свободное состояние или истечет время ожидания:

```
DWORD WINAPI WaitForMultipleObjects(
  _In_  DWORD nCount,
  _In_  const HANDLE *lpHandles,
  _In_  BOOL bWaitAll,
  _In_  DWORD dwMilliseconds
);
```

Параметры функции:

— *nCount* – количество дескрипторов объектов ядра, заданных в массиве *lpHandles*; его значение должно быть в пределах от 1 до *MAXIMUM_WAIT_OBJECTS*;

— *lpHandles* – массив дескрипторов объектов ядра, который может содержать дескрипторы объектов различных типов;

— *fWaitAll* определяет характер ожидания: всех заданных объектов ядра (значение *TRUE*) или одного из них (значение *FALSE*);

— *dwMilliseconds* идентичен одноименному параметру функции *WaitForSingleObject()*.

Возвращаемое значение указывает, почему возобновилосъ выполнение вызвавшего ее потока. Функция может вернуть одно из следующих значений:

— от *WAIT_OBJECT_0* до *(WAIT_OBJECT_0 +nCount– 1)* означает: если *bWaitAll* равно *TRUE*, то все объекты перешли в свободное состояние, а если *bWaitAll* равно *FALSE*, то вычтя *WAIT_OBJECT_0* из возвращаемого значения, получим индекс дескриптора объекта в массиве *lpHandles*, который перешел в свободное состояние;

— *WAIT_TIMEOUT* – истекло время ожидания, а условия, заданные параметром *bWaitAll*, еще не удовлетворены;

— *WAIT_FAILED* – вызов функции закончился неудачно.

Пример 1_8. Пример иллюстрирует, как вызывать *WaitForMultipleObjects()* со значением времени ожидания, отличным от *INFINITE* и условием ожидания «Один из объектов».

```
HANDLE h[3];
h[0] = hThread1;
h[1] = hThread2;
h[2] = hThread3;
DWORD dwWaitResult=WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dwWaitResult)
{
case WAIT_FAILED:
     // неправильный вызов функции (неверный описатель?)
     break;
case WAIT_TIMEOUT:
     // ни один из объектов не освободился в течение 5000 мс
     break;
case WAIT_OBJECT_0+0:
     // завершился поток, идентифицируемый h[0]
     break;
case WAIT_OBJECT_0+1:
     // завершился поток, идентифицируемый h[1]
     break;
case WAIT_OBJECT_0+2:
     // завершился поток, идентифицируемый h[2]
     break;
}
```

События. Самая примитивная разновидность объектов ядра. Они содержат две булевы переменные: одна хранит тип объекта (со сбросом вручную и с автосбросом), другая – его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события со сбросом вручную позволяют возобновлять выполнение сразу нескольких ждущих потоков, с автосбросом – только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект-событие в занятое состояние и приступает к своим операциям. Закончив работу, он сбрасывает событие в свободное состояние. Тогда другой поток, ожидавший перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Объект ядра «событие» создается функцией *CreateEvent()*:

```
HANDLE WINAPI CreateEvent(
```

```
    _In_opt_  LPSECURITY_ATTRIBUTES lpEventAttributes,
    _In_      BOOL bManualReset,
    _In_      BOOL bInitialState,
    _In_opt_  LPCTSTR lpName
);
```

Параметры функции:

— *lpEventAttributes* – атрибуты доступа;

— *bManualReset* – тип создаваемого события: со сбросом вручную (*TRUE*) или с автосбросом (*FALSE*);

— *bInitialState* – начальное состояние события: свободное (*TRUE*) или занятое (*FALSE*);

— *lpName* – имя объекта, которое позволяет использовать событие в других процессах; при значении параметра *NULL* создается неименованный объект.

Возвращаемое значение – дескриптор события, если функция успешна, и *NULL* – в противном случае.

Функция *ResetEvent()* используется для перевода события в занятое состояние:

```
BOOL WINAPI ResetEvent(
    _In_  HANDLE hEvent
);
```

Параметры функции:

— *hEvent* – дескриптор события.

Возвращаемое значение. Если функция успешна, то возвращаемое значение отлично от 0, в случае неуспеха – 0.

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически переводится в занятое состояние и для него не требуется вызывать функцию *ResetEvent()*. Для событий со сбросом вручную для перевода в занятое состояние необходимо вызывать *ResetEvent()*.

Функция *SetEvent()* используется для перевода события в свободное состояние:

```
BOOL WINAPI SetEvent(
    _In_  HANDLE hEvent
);
```

Параметры функции:

— *hEvent* – дескриптор события.

Возвращаемое значение. Если функция успешна, то возвращаемое значение отлично от 0, в случае неуспеха – 0.

Семафоры. Используются для учета ресурсов. Они содержат два 32 битных значения со знаком: счетчик текущего числа ресурсов и максимальное число ресурсов, контролируемое семафором.

Для семафоров определены следующие правила:

— когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;

— если этот счетчик равен 0, семафор занят;

— система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;

— счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Объект ядра «семафор» создается функцией *CreateSemaphore()*:

```
HANDLE WINAPI CreateSemaphore(
   _In_opt_  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
   _In_      LONG lInitialCount,
   _In_      LONG lMaximumCount,
   _In_opt_  LPCTSTR lpName
);
```

Параметры функции:

— *lpSemaphoreAttributes* – атрибуты доступа;

— *lInitialCount* – начальное значение счетчика текущего числа ресурсов;

— *lMaximumCount* – максимальное число ресурсов, контролируемое семафором;

— *lpName* – имя объекта, которое позволяет использовать семафор в других процессах; при значении параметра *NULL* создается неименованный объект.

Возвращаемое значение – дескриптор семафора, если функция успешна, и *NULL* – в противном случае.

Поток получает доступ к ресурсу, вызывая одну из *Wait*-функций и передавая ей дескриптор семафора, который охраняет этот ресурс. *Wait*-функция проверяет у семафора счетчик текущего числа ресурсов, и если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым.

Если *Wait*-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время, а он, захватив ресурс, уменьшит значение счетчика на 1.

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore()*:

```
BOOL WINAPI ReleaseSemaphore(
   _In_        HANDLE hSemaphore,
   _In_        LONG lReleaseCount,
   _Out_opt_   LPLONG lpPreviousCount
);
```

Параметры функции:

— *hSemaphore* – дескриптор семафора;

— *lReleaseCount* – значение, на которое будет увеличен счетчик текущего числа доступных ресурсов;

— *lpPreviousCount* – указатель на переменную, в которой функция вернет предыдущее значение счетчика ресурсов; можно задавать *NULL*, если это значение не требуется.

Возвращаемое значение. Если функция успешна, то возвращаемое значение отлично от 0, в случае неуспеха – 0.

Мьютексы. Гарантируют потокам взаимоисключающий доступ к критическому ресурсу. Они содержат переменную, в которой запоминается идентификатор потока, который захватил мьютекс. Мьютексы ведут себя подобно критическим секциям.

Для мьютексов определены следующие правила:

— если его идентификатор потока равен 0, мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;

— если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии.

Объект ядра «мьютекс» создается функцией *CreateMutex ()*:

```
HANDLE WINAPI CreateMutex(
   _In_opt_  LPSECURITY_ATTRIBUTES lpMutexAttributes,
   _In_      BOOL bInitialOwner,
   _In_opt_  LPCTSTR lpName
);
```

Параметры функции:

— *lpMutexAttributes* – атрибуты доступа;

— *bInitialOwner* – флаг начального владельца мьютекса: *TRUE* – идентификатор потока, которому принадлежит мьютекс, приравнивается идентификатору вызывающего потока и *FALSE* – объект-мьютекс находится в свободном состоянии;

— *lpName* – имя объекта, которое позволяет использовать мьютекс в других процессах; при значении параметра *NULL* создается неименованный объект.

Возвращаемое значение – дескриптор мьютекса, если функция успешна, и *NULL* – в противном случае.

Поток получает доступ к разделяемому ресурсу, вызывая одну из *Wait*-функций и передавая ей дескриптор мьютекса, который охраняет этот ресурс. *Wait*-функция проверяет у мьютекса идентификатор потока. Если его значение равно 0, то мьютекс свободен, и вызывающий поток остается планируемым. При этом идентификатор потока у мьютекса делается равным идентификатору вызывающего потока. Если идентификатор потока мьютекса не равен 0, то мьютекс занят, и вызывающий поток переходит в состояние ожидания.

Для перевода мьютекса в свободное состояние поток вызывает функцию *ReleaseMutex()*:

```
BOOL WINAPI ReleaseMutex(
  _In_  HANDLE hMutex
);
```

Параметры функции:

— *hMutex* – дескриптор мьютекса.

Возвращаемое значение. Если функция успешна, то возвращаемое значение отлично от 0, в случае неуспеха – 0.

Пример 1_9. Функция потока работает с очередью заданий. В программу поступают задания, которые обрабатываются одновременно несколькими потоками. Когда поток завершает обработку очередного задания, он проверяет, есть ли необработанные задания и продолжает свою работу. Структура функция потока может быть следующей:

```
// структура задания
struct SJob
{
  SJob * next;   // указатель на следующее задание
  // поля-атрибуты задания
  …
};
SJob * job_queue; // список необработанных заданий

DWORD WINAPI ThreadFunction(LPVOID)
{
  while (job_queue != NULL)
  {
    // 1) запрашиваем следующее задание
    // 2) удаляем задание из списка
    // 3) выполняем задание
  }
}
```

Здесь требуется синхронизация потоков, а именно: необходимо обеспечить взаимоисключающий доступ к общей очереди заданий при выполнении операций добавления и удаления заданий.

Пример 1_10. Пример использования мьютекса для монопольного доступа к очереди заданий.

```
HANDLE hMutex;      // дескриптор мьютекса
bool bTerminate=false;

DWORD WINAPI ThreadFunction(LPVOID)
{
  while (!bTerminate)
  {
    DWORD dwWaitResult=WaitForSingleObject(hMutex, INFINITE);
    if (dwWaitResult==WAIT_OBJECT_0)
    {
      // монопольный доступ к очереди
      if (job_queue != NULL)
      {
        // 1) запрашиваем следующее задание
        // 2) удаляем задание из списка
      }
      // работа с очередью заданий окончена
      if (!ReleaseMutex(hMutex))
      {
        // обработка ошибки ReleaseMutex()
      };
    }
    else
    {
      // обработка ошибки WaitForSingleObject()
    }
    // 3) если есть новое задание, то выполняем его
  }
}
```

Объект мьютекс должен быть создан перед порождением потоков следующим образом:

```
hMutex = CreateMutex();
if (!hMutex) {
   // обработка ошибки создания мьтекса
}
```

Пример 1_11. Пример использования мьютекса и семафора. В функции потока, работающей с очередью заданий, можно использовать семафор для определения новых заданий в очереди и мьютех для монопольного доступа к очереди.

```
HANDLE hSemaphoreMutex[2];    // массив дескрипторов
```

```
bool bTerminate=false;

DWORD WINAPI ThreadFunction(LPVOID)
{
  DWORD dwWaitResult;
  while (!bTerminate)
  {
    dwWaitResult= WaitForMultipleObjects(2,hSemaphoreMutex,
                        TRUE,INFINITE);
    if ( (dwWaitResult==WAIT_OBJECT_0+0) ||
         (dwWaitResult==WAIT_OBJECT_0+1) )
    {
      // монопольный доступ к непустой очереди
      // 1) запрашиваем следующее задание
      // 2) удаляем задание из списка
      // работа с очередью окончена
      if (!ReleaseSemaphore(hSemaphoreMutex[0]))
      {
        // обработка ошибки ReleaseSemaphore()
      };
      if (!ReleaseMutex(hSemaphoreMutex[1]))
      {
        // обработка ошибки ReleaseMutex()
      };
    }
    else
    {
      // обработка ошибки WaitForMultipleObjects()
    }
    // 3) Если есть новое задание, то выполняем его
  }
}
```

Объекты семафор и мьютекс должны быть созданы перед порождением потоков следующим образом:

```
hSemaphoreMutex[0]=CreateSemaphore();
hSemaphoreMutex[1]=CreateMutex();
if () {
  // обработка ошибки создания объектов синхронизации
}
```

1.8. Измерение времени

Для определения эффективности многопоточной программы требуется знать, сколько времени затрачивает поток на выполнение той или иной операции. Существует несколько способов, которые можно использовать для измерения времени в Windows. Рассмотрим два из них, которые нам кажутся наиболее подходящими.

Функция GetTickCount(). Используется для измерения времени в предположении, что поток не будет прерван. Единица измерения – миллисекунды.

```
DWORD WINAPI GetTickCount(void);
```

Возвращаемое значение. Количество миллисекунд, которые прошли с момента старта системы. Точность этого измерения определяется разрешающей способностью системного таймера, которая обычно находится в диапазоне 10-16 миллисекунд.

Пример 1_12. Схема измерения времени при использовании функции *GetTickCount().*

```
// получаем стартовое время
DWORD dwStartTime = GetTickCount();
// измеряемые вычисления
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

Например, при измерении времени работы программы из Примера 1_3 получаем время в миллисекундах, отличное от 0. А если измерить время, которое программа затрачивает на создание потоков, то получаем значение 0. Следовательно, требуется более точное измерение временных интервалов.

Счетчик производительности с высоким разрешением. Для измерения малых временных интервалов с высокой точностью используется счетчик производительности с высоким разрешением, с которым работают две системные функции *QueryPerformanceCounter()* и *QueryPerformanceFrequency().*

Функция *QueryPerformanceCounter()* используется для получения текущего значения счетчика производительности с высоким разрешением:

```
BOOL WINAPI QueryPerformanceCounter(
  _Out_  LARGE_INTEGER *lpPerformanceCount
);
```

Параметры функции:

— *lpPerformanceCount* – указатель на переменную, в которой функция вернет текущее значение счетчика производительности с высоким разрешением.

Возвращаемое значение. Если функция успешна, то возвращаемое значение отлично от 0, в случае неуспеха – 0.

Функция *QueryPerformanceFrequency()* используется для получения количества приращений в секунду счетчика производительности с высоким разрешением:

```
BOOL WINAPI QueryPerformanceFrequency(
  _Out_  LARGE_INTEGER *lpFrequency
);
```

Пример 1_13. Схема измерения времени при использовании счетчика производительности с высоким разрешением.

```
LARGE_INTEGER liFrequency, liStartTime, liFinishTime;
// получаем частоту
QueryPerformanceFrequency(&liFrequency);
// получаем стартовое время
QueryPerformanceCounter(&liStartTime);
// измеряемые вычисления
// получаем финишное время
QueryPerformanceCounter(&liFinishTime);
// вычисляет время в миллисекундах
double dElapsedTime = 1000.*(liFinishTime.QuadPart-
liStartTime.QuadPart)/liFrequency.QuadPart;
```

При таком способе измерения времени при проведении вычислительных экспериментов с приложением из Примера 1_3 получаем время, которое программа затрачивает на создание потоков, отличное от 0.

Проведение вычислительных экспериментов. Для оценки эффективности параллельного способа решения задачи проводят вычислительные эксперименты и измеряют время выполнения последовательной и параллельной версий программ для различных входных данных.

Одной из главных характеристик параллельной программы является **ускорение**:

$$S_p(n) = T_1(n) / T_p(n),$$

где $T_1(n)$ − время выполнения последовательной версии программы, $T_p(n)$ − время выполнения параллельной программы на p-процессорной системе.

Еще одна важная характеристика параллельной программы – **эффективность**:

$$E_p(n) = T_1(n) / (pT_p(n)) = S_p(n) / p.$$

При проведении вычислительных экспериментов измеряют время и заполняют таблицу 2.

Таблица 2

Результаты вычислительных экспериментов

Размерность задачи	Время выполнения последовательной программы	Параллельная программа на 2 процессорах			…	Параллельная программа на *p* процессорах		
		Время выполнения	Ускорение	Эффективность		Время выполнения	Ускорение	Эффективность

1.9. Пример

Написать многопоточную программу для поиска максимального значения среди элементов матрицы *A*, имеющей *n* строк и *m* столбцов.

Рассмотрим этапы разработки параллельных алгоритмов [5] на примере поставленной задачи.

Этап 1. Разделение вычислений на независимые части. В качестве базовой операции можно выбрать поиск максимального значения в строке. Такой выбор удовлетворяет двум основным критериям декомпозиции вычислений: независимость и равная трудоемкость.

Этап 2. Выделение информационных зависимостей. Подзадачи независимо друг от друга вычисляют максимальный элемент в своей строке; после завершения всех подзадач необходимо вычислить максимальное из полученных подзадачами значений.

Этап 3. Масштабирование набора подзадач. Здесь требуется определить необходимую (или доступную) для решения задачи вычислительную систему и выполнить распределение имеющегося набора подзадач между процессорами системы. В нашем случае необходимо распределить *n* подзадач между *p* вычислительными элементами, когда количество подзадач значительно превышает количество вычислительных элементов. Для сокращения количества подзадач необходимо выполнить укрупнение (агрегацию) вычислений. Будет объединять отдельные строки матрицы в последовательные группы строк, то есть использовать ленточную схему на непрерывной основе. Получим *p* подзадач, каждая из которых находит максимальный элемент для *n/p* строк.

Этап 4. Распределение подзадач между вычислительными элементами. Распределение нагрузки обычно выполняется операционной системой автоматически. Кроме того, данный этап распределения подзадач между процессорами является избыточным, если количество подзадач совпадает с числом имеющихся вычислительных элементов.

Ниже представлена последовательная программа.

```
#include <iostream>
#include <algorithm>
#include <time.h>
void InitRand(int**, int, int);
void main()
{
     int **A, *Max, n, m, i, result=0;
     std::cin >> n >> m;
     A = new int* [n];
     Max = new int [n];
     for (i=0; i<n; ++i)
```

```
          A[i] = new int [m];
     // заполнение матрицы случайными числами
     InitRand(A, n, m);
     // поиск максимального элемента в каждой строке
     for (i=0; i<n; ++i)
          Max[i] = *(std::max_element(A[i]+0, A[i]+m));
     // поиск максимального из максимальных для каждой строки
     result = *(std::max_element(Max+0, Max+n));
     printf("result = %d\n", result);

     for (i=0; i<n; ++i)
          delete [] A[i];
     delete [] Max;
     delete [] A;
}
void InitRand(int**B, int size1, int size2)
{
     int i, j;
     srand(time(0));
     for (i=0; i<size1; ++i)
          for (j=0; j<size2; ++j)
               B[i][j] = rand();
}
```

Вариант 1. Для организации многопоточной программы используем модель создания и функционирования потоков, которая называется «Модель с равноправными узлами» [6]. Главный поток создает *p* одинаковых дочерних потоков, ждет их завершения и вычисляет максимальное из полученных потоками значений. Функция дочернего потока получает диапазон номеров строк, для каждой из которых: вычисляет максимальное значение в строке и сохраняет его в соответствующем элементе глобального массива. Потоки осуществляют доступ к общим данным: элементы и размерность матрицы *A* (операция чтения) и одномерный массив *Max* максимальных элементов строк (операция записи, но каждый поток обращается к своим элементам и синхронизация не требуется).

Ниже представлена многопоточная программа.

```
#include …

const int p=3; // количество дочерних потоков
int **A, *Max, n, m;
// тип параметра, передаваемого функции потока
struct SThreadParam
{
     int startI, finishI;
};
```

```
// функция потока
DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
     SThreadParam* param = (SThreadParam*)pvParam;
     int i, start, finish;
     start = param->startI;
     finish = param->finishI;
     for (i=start; i<=finish; ++i)
          Max[i] = *(std::max_element(A[i]+0, A[i]+m));
     return 0;
}
int main()
{
     // объявление переменных
     // выделение памяти
     // ввод размера и инициализация матрицы

     HANDLE hThreads[p];
     SThreadParam params[p];
     int k;
     int w = n/p;
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          // подготовка параметра для функции потока
          params[k].startI = k*w;
          params[k].finishI = (k+1)*w-1;
          if (k==p-1)
               params[k].finishI = n-1;
          hThreads[k] = (HANDLE)_beginthreadex(NULL, 0,
ThreadFunction, (LPVOID)&(params[k]), 0, NULL);
          if (hThreads[k]==NULL)   // обработка ошибки
          {
               printf("Create Thread %d Error=%d\n", k, Get-
LastError());
               return -1;
          }
     }
     // ожидание завершения дочерних потоков
     WaitForMultipleObjects(p, hThreads, TRUE,INFINITE);
     for (k=0; k<p; ++k)
          CloseHandle(hThreads[k]);

     result = *(std::max_element(Max+0, Max+n));
     printf("result = %d\n", result);
     // освобождение памяти
   }
```

Вариант 2. Для организации многопоточной программы используем ту же модель создания и функционирования потоков – «Модель с равноправными узлами». Главный поток создает мьютекс, формирует очередь заданий и *p* одинаковых дочерних потоков, ждет завершения потоков и вычисляет максимальное из полученных потоками значений. Очередь заданий *job_queue* представляет собой список номеров строк матрицы для поиска максимального элемента и имеет тип *std::queue <int>*. Функция дочернего потока работает с очередью заданий, из которой получает задание, выполняет его и сохраняет результат в соответствующем элементе глобального массива. Функция потока работает до тех пор, пока в очереди есть задания. Для монопольного доступа к очереди функция потока использует мьютекс *hMutex*.

```
#include …
const int p=3; // количество дочерних потоков
int **A, *Max, n, m;
std::queue <int> job_queue;   // очередь заданий
HANDLE hMutex;

// функция потока
DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
     DWORD dwWaitResult;
     bool bTerminate = false;
     int i;
     while (!bTerminate)
     {
          i = -1;
          dwWaitResult=WaitForSingleObject(hMutex, INFINITE);
          if (dwWaitResult==WAIT_OBJECT_0)
          {
               // монопольный доступ к очереди
               if (job_queue.size() != 0)
               {
                    i = job_queue.front();
                    job_queue.pop();
               }
               // работа с очередью заданий окончена
               if (!ReleaseMutex(hMutex))
               {
                    // обработка ошибки ReleaseMutex()
                    printf("Release Mutex Error=%d\n", Get-
LastError());
                    return -1;
               };
          }
```

```
          else
          {
               // обработка ошибки WaitForSingleObject()
               printf("Wait For Single Object Error=%d\n",
GetLastError());
               return -1;
          }
          if (i!=-1)     // есть новое задание
               Max[i] = *(std::max_element(A[i]+0, A[i]+m));
          else      // очередь заданий пуста
               bTerminate = true;
     }
     return 0;
}
int main()
{    // объявление переменных
     // выделение памяти
     // ввод размера и инициализация матрицы

     HANDLE hThreads[p];
     int k;
     // создание мьютекса
     hMutex = CreateMutex(NULL, FALSE, NULL);
     if (!hMutex)   // обработка ошибки
     {
          printf("Create Mutex Error=%d\n", GetLastError());
          return -1;
     };
     // формирование очереди заданий
     for (i=0; i<n; ++i)
          job_queue.push(i);
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          hThreads[k]    =    (HANDLE)_beginthreadex(NULL,    0,
ThreadFunction, NULL, 0, NULL);
          if (hThreads[k]==NULL)   // обработка ошибки
          {
               printf("Create  Thread  %d  Error=%d\n",  k,  Get-
LastError());
               return -1;
          }
     }
     // ожидание завершения дочерних потоков
     WaitForMultipleObjects(p, hThreads, TRUE, INFINITE);
     for (k=0; k<p; ++k)
          CloseHandle(hThreads[k]);
```

```
    CloseHandle(hMutex);

    result = *(std::max_element(Max+0, Max+n));
    printf("result = %d\n", result);
    // освобождение памяти
}
```

Вариант 3. Для организации многопоточной программы используем модель создания и функционирования потоков, которая называется «Модель делегирования» [6]. В модели делегирования один поток («управляющий») создает потоки («рабочие») и назначает каждому из них задачу. Управляющему потоку нужно ожидать до тех пор, пока все потоки не завершат выполнение своих задач. Управляющий поток может создавать рабочие потоки в результате запросов, обращенных к системе. При этом обработка запроса каждого типа может быть делегирована рабочему потоку. В этом случае управляющий поток выполняет некоторый цикл событий. По мере возникновения событий рабочие потоки создаются и на них тут же возлагаются определенные обязанности. Для каждого нового запроса, обращенного к системе, создается новый поток. При использовании такого подхода процесс может превысить предельный объем выделенных ему ресурсов или предельное количество потоков.

Главный (управляющий) поток обрабатывает список запросов «Найти максимальный элемент в строке» для каждой строки матрицы. Для каждого запроса управляющий поток создает рабочий поток. Существует ограничение на количество одновременно работающих дочерних потоков (не более *p*). Управляющий поток, обработав все запросы, ждет завершения рабочих потоков и вычисляет максимальное из полученных потоками значений. Функция потока вычисляет максимальное значение в строке, номер которой передается через параметр функции, и сохраняет результат в соответствующем элементе глобального массива. Чтобы гарантировать, что число рабочих потоков не будет превышено *p*, используется семафор *hSemaphore*.

```
#include …
const int p=3; // количество дочерних потоков
int **A, *Max, n, m;
HANDLE hSemaphore;

// функция рабочего потока
DWORD WINAPI ThreadFunction(LPVOID pvParam)
{
    int i = (int)pvParam;
    Max[i] = *(std::max_element(A[i]+0, A[i]+m));
    ReleaseSemaphore(hSemaphore, 1, NULL);
    return 0;
```

```
}
int main()
{    // объявление переменных
     // выделение памяти
     // ввод размера и инициализация матрицы

     HANDLE * hThreads;
     hThreads = new HANDLE [n];
     // создание семафора
     hSemaphore = CreateSemaphore(NULL, p, p, NULL);
     if (!hSemaphore)    // обработка ошибки
     {
          printf("Create Semaphore Error=%d\n", GetLastEr-
ror());
          return -1;
     };

     // управляющий поток
     DWORD dwWaitResult;
     for (i=0; i<n; ++i) // обработка запросов
     {
          dwWaitResult=WaitForSingleObject(hSemaphore, INFI-
NITE);
          if (dwWaitResult==WAIT_OBJECT_0)
          {
               // можно создать новый рабочий поток
               hThreads[i] = (HANDLE)_beginthreadex(NULL, 0,
ThreadFunction, (LPVOID)i, 0, NULL);
               if (hThreads[i]==NULL)   // обработка ошибки
               {
                    printf("Create Thread %d Error=%d\n", i,
GetLastError());
                    return -1;
               }
          }
     }
     // ожидание завершения дочерних потоков - p последних
     WaitForMultipleObjects(p, hThreads+n-p, TRUE,INFINITE);
     for (i=0; i<n; ++i)
          CloseHandle(hThreads[i]);
     CloseHandle(hSemaphore);
     delete [] hThreads;

     result = *(std::max_element(Max+0, Max+n));
     printf("result = %d\n", result);
     // освобождение памяти
}
```

2. POSIX THREADS

2.1. Предварительные замечания

На многих системах параллельного действия установлен тот или иной вариант операционной системы UNIX. Существуют различные стандарты на UNIX-системы, наиболее развитый из них – POSIX (Portable Operating System Interface). Большинство существующих UNIX-систем были адаптированы для удовлетворения этому стандарту.

Одна из частей стандарта POSIX посвящена библиотеке функций языка C для многопоточного программирования. Библиотека этих функций называется Pthreads (POSIX threads). Сейчас эта библиотека широко распространена и доступна на разных UNIX-системах и некоторых других системах. В частности, в Windows реализована библиотека API-функций работы с потоками Pthreads, соответствующая стандарту POSIX.

Многопоточные программы отличаются легкостью переноса из-под других операционных систем. Имея отлаженную параллельную программу на базе WinAPI, можно получить ее аналог на базе PthreadsAPI. Затем отладить в привычном Windows-окружении многопоточную программу в варианте Pthreads. И наконец, без изменений использовать Pthreads-программу, например, на суперкомпьютере СКИФ-БГУ, где установлена операционная система Linux Fedora 8.0.

Библиотека Pthreads содержит около 100 функций. Типы данных и функции библиотеки объявлены в заголовочном файле <pthread.h>.

Процесс ***обработки ошибок*** в Pthreads отличается от принятого в WinAPI. Здесь системные функции возвращают нулевое значение в случае успеха и ненулевое – код ошибки – в случае неудачи.

Рассмотрим основные возможности PthreadsAPI для разработки многопоточных программ на языке C. Более подробную информацию можно найти, например, в [6-8].

2.2. Функция потока

В варианте Pthreads каждому потоку в процессе назначается собственный идентификатор, который имеет тип данных *pthread_t*.

Функция потока имеет следующий тип:

```
void* (*) (void*)
```

Таким образом, функция потока принимает единственный параметр типа *void** и возвращает значение типа *void**.

2.3. Создание потока

Функция *pthread_create()* создает новый поток. Функция может быть вызвана любое число раз из любой точки программы. Заголовок функции:

```
int pthread_create(
  pthread_t* thread,
  const pthread_attr_t* attr,
  void* (*start_routine) (void *),
  void* arg);
```

Параметры функции:

— *thread* – указатель на переменную, в которой функция сохранит идентификатор нового потока;

— *attr* – указатель на переменную, которая задает набор атрибутов создаваемого потока; будем использовать значение *NULL* (новый поток получит атрибуты по умолчанию);

— *start_routine* – указатель на функцию потока;

— *arg* – аргумент, который будет передан функции потока.

Максимальное количество потоков, которое может создать процесс, зависит от реализации.

Пример 2_1. Создание потока. Параметр, передаваемый функции потока, не используется.

```
#include <pthread.h>
#include <stdio.h>

void* ThreadFunction(void* param)
{
     printf("I am Thread\n");
     return NULL;
}
int main()
{
     pthread_t thread_id;
     int rc;
     // создание дочернего потока
     rc=pthread_create(&thread_id, NULL, ThreadFunction,
NULL);
     if (rc)   // обработка ошибки
     {
          printf("ERROR: return code from pthread_create()
%d\n", rc);
          return -1;
     }
     // ожидание завершения дочернего потока
```

```
    pthread_join(thread_id, NULL);
    return 0;
}
```

2.4. Завершение потока

Поток может быть завершен одним из следующих способов:

— функция потока возвращает управление;

— поток сам вызывает функцию *pthread_exit()*;

— поток прерывается другим потоком вызовом *pthread_cancel()*;

— функция *exit()* завершает процесс, в том числе уничтожая все потоки;

— завершение функции *main()* также завершает процесс и уничтожает все потоки.

Рекомендуемыми способами завершения потока при проектировании функций потока являются первый и второй способы.

Первый способ – функция потока возвращает управление – означает, что поток завершился при нормальных обстоятельствах, выполнив свою работу. Результатом работы потока станет значение, возвращаемое из функции потока.

Второй способ – поток вызывает функцию *pthread_exit()* – также означает, что поток завершился при нормальных обстоятельствах, выполнив или не выполнив свою работу. Результатом работы потока станет значение аргумента функции *pthread_exit()*:

```
int pthread_exit(
  void *value_ptr
);
```

Параметры функции:

— *value_ptr* – указатель на переменную, которую функция потока хочет передать в среду вызова; обычно в этой переменной передается код завершения потока.

Пример 2_2. Использование функции *pthread_exit()*. Функция *main()* завершает свое выполнение вызовом *pthread_exit()*, не завершая дочерний поток.

```
#include <pthread.h>
#include <stdio.h>

void* ThreadFunction(void* param)
{
    printf("I am Thread\n");
    return NULL;
}
int main()
```

```
{
     pthread_t thread_id;
     int rc;
     // создание дочернего потока
     rc=pthread_create(&thread_id, NULL, ThreadFunction,
NULL);
     if (rc)   // обработка ошибки
     {
          printf("ERROR: return code from pthread_create()
%d\n", rc);
          return -1;
     }
     // завершение main() без ожидания
     // завершения дочернего потока
     pthread_exit(NULL);
}
```

Чтобы получить значение, возвращаемое функцией потока, используется функция *pthread_join()*:

```
int pthread_join(
  pthread_t thread,
  void* thread_return
);
```

Параметры функции:

— *thread* – идентификатор потока;

— *thread_return* – указатель на переменную, в которой функция *pthread_join()* сохранит значение, возвращаемое функцией потока; может быть *NULL*, если это значение не требуется.

Пример 2_3. Получение значения, возвращаемого потоком. Функция *main()* ожидает завершения дочернего потока, получает и выводит значение, возвращаемое функцией потока.

```
#include <pthread.h>
#include <stdio.h>

void* ThreadFunction(void* param)
{
     printf("I am Thread\n");
     return (void*)13;
}
int main()
{
     pthread_t thread_id;
     int rc;
     int threadValue;
     // создание дочернего потока
```

```
     rc=pthread_create(&thread_id, NULL, ThreadFunction,
NULL);
     if (rc)   // обработка ошибки
     {
          printf("ERROR: return code from pthread_create()
%d\n", rc);
          return -1;
     };
     // ожидание завершения дочернего потока
     // получение значения, возвращаемого функцией потока
     pthread_join(thread_id, &threadValue);
     printf("threadValue = %d\n", threadValue);
     return 0;
}
```

2.5. Передача данных потоку

Данные потоку передаются через аргумент потоковой функции, значение которого задается при создании потока вызовом функции *pthread_create()*. Передача данных осуществляется по тем же правилам, что и для Windows-потоков.

Пример 2_4. Рассмотрим вычисление числа π в соответствии с условием, сформулированным в Примере 1_4.

Каждый поток осуществляет доступ к своему экземпляру структуры, поэтому синхронизация доступа не требуется.

```
#include <pthread.h>
#include <stdio.h>

#define p 2    // количество дочерних потоков
int n = 1000000;

// тип параметра, передаваемого функции потока
struct SThreadParam
{
     int k;
     double sum;
};
// функция потока
void* ThreadFunction(void* pvParam)
{
     struct SThreadParam *param = (struct SThreadParam
*)pvParam;
     int i, start;
     double h, sum, x;
     h = 1./n;
     sum = 0.;
     start = param->k;
```

```
     for (i=start; i<n; i+=p)
     {
          x = h * i;
          sum += 4. / (1. + x*x);
     }
     param->sum = h * sum;
     return 0;
}
int main()
{
     // массив идентификаторов потоков
     pthread_t thread_id[p];
     // массив параметров потоковых функций
     struct SThreadParam params[p];
     int k;
     double sum;
     int rc;
     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          params[k].k = k;
          rc=pthread_create(&thread_id[k], NULL, ThreadFunc-
tion, (void*)&params[k]);
          if (rc)   // обработка ошибки
          {
               printf("ERROR: return code %d from
pthread_create() N %d\n", rc, k);
               return -1;
          };
     }
     // ожидание завершения дочерних потоков
     for (k=0; k<p; ++k)
          pthread_join(thread_id[k], NULL);
     sum = 0.;
     for (k=0; k<p; ++k)
          sum += params[k].sum;
     printf("PI = %.16f\n", sum);
     return 0;
}
```

2.6. Синхронизация потоков

Функция *pthread_join()* блокирует выполнение вызвавшего ее потока до завершения потока, заданного ее первым параметром. Windows-аналог функции *pthread_join()* – функция *WaitForSingleObject()*, используемая для ожидания в течение неограниченного времени завершения

определенного потока. Для ожидания завершения нескольких потоков функция *pthread_join()* вызывается в цикле для каждого потока.

Поток можно ожидать посредством функции *pthread_join()*, только если он имеет атрибут *PTHREAD_CREATE_JOINABLE*. Не все реализации Pthreads используют этот атрибут по умолчанию. Поэтому если планируется ожидать завершения потока, то рекомендуется явно задать этот атрибут при создании потока.

Пример 2_5. Создание потока с атрибутом *PTHREAD_CREATE_JOINABLE.*

```
pthread_t thread;
pthread_attr_t attr;     // атрибуты потока
void *status;

// инициализация атрибутов потока
pthread_attr_init(&attr);
// установка атрибута в требуемое значение
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
// создание потока
rc=pthread_create(&thread, &attr, ThreadFunction, NULL);
…
// освобождение ресурсов
pthread_attr_destroy(&attr);
// ожидание завершения потока
rc = pthread_join(thread[t], &status);
if (rc) {
     printf("ERROR; return code from pthread_join() is %d\n",
rc);
     exit(-1);
}
```

Мьютексы. Представляют основной способ реализации взаимоисключающего доступа к разделяемым данным.

Для работы с мьютексом необходимо определить переменную типа *pthread_mutex_t* и инициализировать ее перед первым использованием. Существует два способа инициализации. Первый способ – с помощью инициализатора:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Второй способ инициализации – вызов функции *pthread_mutex_init()*:

```
int pthread_mutex_init(
  pthread_mutex_t * mutex,
  pthread_mutexattr_t * attr
);
```

Параметры функции:

— *mutex* – указатель на переменную мьютекса;

— *attr* – указатель на структуру с атрибутами мьютекса; может быть *NULL* для использования значений по умолчанию.

Начальное состояние мьютекса – unlocked (разблокирован, свободен).

Поток получает доступ к разделяемому ресурсу, вызывая функцию *pthread_mutex_lock()*:

```
int pthread_mutex_lock(
  pthread_mutex_t * mutex
);
```

Параметры функции:

— *mutex* – указатель на переменную мьютекса.

Если мьютекс уже занят другим потоком, то функция *pthread_mutex_lock()* блокирует вызвавший ее поток до тех пор, пока мьютекс не станет свободным.

После того, как поток закончил использование защищаемых мьютексом ресурсов, требуется вызвать функцию *pthread_mutex_unlock()*:

```
int pthread_mutex_unlock(
  pthread_mutex_t * mutex
);
```

Когда мьютекс больше не нужен ни одному потоку, следует освободить занятые им ресурсы вызовом функции *pthread_mutex_destroy()*:

```
int pthread_mutex_destroy(
  pthread_mutex_t * mutex
);
```

Пример 2_6. Рассмотрим вычисление числа π в соответствии с условием, сформулированным в Примере 1_6. Используется мьютекс для взаимоисключающего доступа к переменной *pi*, разделяемой всеми потоками.

```
#include <pthread.h>
#include <stdio.h>

#define p 2    // количество потоков
int n = 1000000;
double pi = 0.;
pthread_mutex_t piMutex;      // мьютекс

void* ThreadFunction(void* pvParam)
{
     int nParam = (int)pvParam;
     int i, start;
     double h, sum, x;
     h = 1./n;
     sum = 0.;
     start = nParam;
     for (i=start; i<n; i+=p)
```

```
     {
          x = h * i;
          sum += 4. / (1. + x*x);
     }
     pthread_mutex_lock (&piMutex);     // захватить мьютекс
     pi += h * sum;
     pthread_mutex_unlock (&piMutex);   // освободить мьютекс
     return 0;
}
int main()
{
     pthread_t thread_id[p];
     pthread_attr_t attr;     // атрибуты потока
     int k;
     int rc;
     // инициализация мьютекса
     pthread_mutex_init(&piMutex, NULL);

     pthread_attr_init(&attr);
     pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_JOINABLE);

     // создание дочерних потоков
     for (k=0; k<p; ++k)
     {
          rc=pthread_create(&thread_id[k], &attr, ThreadFunc-
tion, (void*)k);
          if (rc)   // обработка ошибки
          {
               printf("ERROR: return code %d from
pthread_create() N %d\n", rc, k);
               return -1;
          };
     }
     pthread_attr_destroy(&attr);

     // ожидание завершения дочерних потоков
     for (k=0; k<p; ++k)
          pthread_join(thread_id[k], NULL);
     // удаление мьютекса
     pthread_mutex_destroy(&piMutex);
     printf("PI = %.16f\n", pi);
     return 0;
}
```

Семафоры. В Pthreads существует реализация семафоров. Семафор – это счетчик, используемый для синхронизации потоков. Счетчик семафора является неотрицательным целым числом. Семафор поддерживает две базовые операции:

— операция ожидания уменьшает значение счетчика на единицу; если счетчик уже равен нулю, операция блокируется до тех пор, пока значение счетчика не станет положительным (вследствие действий, выполняемых другими потоками); после снятия блокировки значение семафора уменьшается на единицу и операция завершается;

— операция установки увеличивает значение счетчика на единицу; если до этого счетчик был равен нулю, и существовали потоки, заблокированные в операции ожидания данного семафора, один из них будет разблокирован.

Для работы с семафором необходимо подключить заголовочный файл <semaphore.h>, определить переменную типа *sem_t* и инициализировать ее перед первым использованием с помощью функции *sem_init()*:

```
int sem_init(
  sem_t * semaphore,
  int pshared,
  unsigned int start_value
);
```

Параметры функции:

— *semaphore* – указатель на переменную семафора;

— *pshared* – должен быть равен нулю;

— *start_value* – начальное значение счетчика семафора.

Чтобы выполнить операцию ожидания семафора, необходимо вызвать функцию *sem_wait()*:

```
int sem_wait(
  sem_t * semaphore
);
```

Параметры функции:

— *semaphore* – указатель на переменную семафора.

Операция установки семафора выполняется вызовом функции *sem_post()*:

```
int sem_post(
  sem_t * semaphore
);
```

Когда семафор больше не нужен ни одному потоку, следует освободить занятые им ресурсы вызовом функции *sem_destroy()*:

```
int sem_destroy(
  sem_t * semaphore
);
```

2.8. Измерение времени

В заголовочном файле <time.h> объявлены типы и функции, связанные с датой и временем. Для измерения времени используется функция *time()*:

```
time_t time(
  time_t * tp
);
```

Параметры функции:

— *tp* – указатель на переменную, в которую функция поместит текущее календарное время.

Возвращаемое значение. Текущее календарное время.

Для определения количества секунд, которые прошли между двумя измерениями функции *time()* используется функция *difftime()*:

```
double difftime(
  time_t time2,
  time_t time1
);
```

Параметры функции:

— *time2* – финишное время;

— *time1* – стартовое время.

Возвращаемое значение. Разность между *time2* и *time1*, выраженная в секундах.

Пример 2_7. Схема измерения времени.

```
time_t startTime, finishTime;
startTime = time(NULL);  // получаем стартовое время
// измеряемые вычисления
finishTime = time(NULL); // получаем финишное время
printf("time=%f sec\n", difftime(finishTime, startTime));
```

2.9. Пример

Приведем текст многопоточной программы из Подраздела 1.9, Вариант 3 в реализации Pthreads.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>

#define p 3
int **A, *Max, n, m;
sem_t semaphore;

void InitRand(int**B, int size1, int size2);

void* ThreadFunction(void* pvParam)
```

```
{
     int i = (int)pvParam;
     int j, rc;
     Max[i] = A[i][0];
     // поиск максимального элемента в строке i
     for (j=1; j<m; ++j)
          if (Max[i]<A[i][j])
               Max[i] = A[i][j];
     rc = sem_post(&semaphore);
     printf("I am worker with %d row; max=%d\n", i, Max[i]);
     return NULL;
}
int main()
{
     pthread_t * thread_id;
     pthread_attr_t attr;     // атрибуты потока
     int rc, i, j, k, result;
     scanf("%d %d", &n, &m);
     A = calloc(n, sizeof(int*));
     Max = calloc(n, sizeof(int));
     for (i=0; i<n; ++i)
          A[i] = calloc(m, sizeof(int));
     InitRand(A, n, m);
     for (i=0; i<n; ++i)
     {
          for (j=0; j<m; ++j)
               printf("%d\t",A[i][j]);
          printf("\n");
     }
     printf("\n");
     // выделение памяти для идентификаторов рабочих потоков
     thread_id = calloc(n, sizeof(pthread_t));

     // инициализация семафора
     // стартовое значение счетчика равно p
     rc=sem_init(&semaphore, 0, p);
     if (rc)   // обработка ошибки
     {
          printf("ERROR: return code from sem_init() %d\n",
rc);
          return -1;
     };
     // инициализация атрибутов потока
     pthread_attr_init(&attr);
     // установка атрибута в требуемое значение
     pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
```

```
     // управляющий поток
     for (i=0; i<n; ++i) // обработка запросов
     {
          rc=sem_wait(&semaphore);
          if (rc)   // обработка ошибки
          {
               printf("ERROR: return code from sem_wait()
%d\n", rc);
               return -1;
          };
          // можно создать новый рабочий поток
          rc=pthread_create(&thread_id[i], &attr, ThreadFunc-
tion, (void*)i);
          if (rc)   // обработка ошибки
          {
               printf("ERROR: return code %d from
pthread_create() N %d\n", rc, i);
               return -1;
          };
     }
     pthread_attr_destroy(&attr);
     // ожидание завершения дочерних потоков
     for (k=0; k<n; ++k)
     {
          rc = pthread_join(thread_id[k], NULL);
          if (rc)   // обработка ошибки
          {
               printf("ERROR: return code %d from
pthread_join() %d\n", rc, k);
               return -1;
          };
     }
     // удаление семафора
     rc = sem_destroy(&semaphore);
     if (rc)   // обработка ошибки
     {
          printf("ERROR: return code from sem_destroy()
%d\n", rc);
          return -1;
     };

     // поиск максимального из максимальных строк
     result=Max[0];
     for (i=1; i<n; ++i)
          if (result<Max[i])
               result=Max[i];
```

```
    printf("result = %d\n", result);

    for (i=0; i<n; ++i)
        free(A[i]);
    free(A);
    free(Max);
    free(thread_id);
}
```

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Буза, М. К. Системы параллельного действия / М. К. Буза. – Минск: БГУ, 2009. – 415 с.

2. Буза, М. К. Многоядерные процессоры / М. К. Буза. – Минск: БГУ, 2012. – 47 с.

3. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. – СПб.: Питер, 2001. – 752 с.

4. http://msdn.microsoft.com/dn308572 (Домашняя страница Microsoft Developer Network)

5. Гергель, В. П. Высокопроизводительные вычисления для многопроцессорных многоядерных систем / В. П. Гергель. – М.: Издательство МГУ, 2010. – 544 с.

6. Хьюз, К. Параллельное и распределенное программирование на C++ / К. Хьюз, Т. Хьюз. – М.: Вильямс, 2004. – 672 с.

7. Митчелл, М. Программирование для Linux. Профессиональный подход / М. Митчелл, Дж. Оулдем, А. Самьюэл. – М.: Вильямс, 2003. – 288 с.

8. Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс. – М.: Вильямс, 2003. – 512 с.

ОГЛАВЛЕНИЕ

Учебное издание

Буза Михаил Константинович
Кондратьева Ольга Михайловна

ПРОЕКТИРОВАНИЕ ПРОГРАММ ДЛЯ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ

Учебно-методическое пособие для студентов факультета прикладной математики и информатики

В авторской редакции

Ответственный за выпуск *О. М. Кондратьева*

Подписано в печать 25.11.2008. Формат 60x84/16. Бумага офсетная.
Усл. печ. л. 2,79. Уч.-изд. л. 2,5. Тираж 50 экз. Заказ

Белорусский государственный университет
ЛИ № 02330/0494425 от 08.04.2009
Пр. Независимости, 4, 220030, Минск.

Отпечатано на копировально-множительной технике
факультета прикладной математики и информатики
Белорусского государственного университета.
Пр. Независимости, 4, 220030, Минск.