СИНХРОНИЗАЦИЯ

- **Барьер**
- **Директива ordered**
- **Критические секции**
- **Директива atomic**
- **Замки**
- **Директива flush**

Барьер

Самый распространенный способ синхронизации в OpenMP – барьер. Он оформляется с помощью директивы barrier:

#pragma omp barrier

Нити, выполняющие текущую параллельную область, дойдя до этой директивы, останавливаются и ждут, пока все нити не дойдут до этой точки программы, после чего разблокируются и продолжают работать дальше. Кроме того, для разблокировки необходимо, чтобы все синхронизируемые нити завершили все порождённые ими задачи (task).

Пример 24

```
#include "stdafx.h"
#include <iostream>
#include <windows.h
#include <omp.h>
#include <stdio.h>
#include <dos.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251); SetConsoleOutputCP(1251);
  #pragma omp parallel
  {
     printf("Текст 1 нить %d \n",omp_get_thread_num());
     printf("Текст 2 нить %d \n",omp_get_thread_num());
 #pragma omp  barrier
     printf("Текст 3 нить %d \n",omp_get_thread_num());
}
 system("Pause"); }
```

Результаты выполнения примера 24

Текст 1 нить 0
Текст 2 нить 0
Текст 1 нить 2
Текст 2 нить 2
Текст 1 нить 4
Текст 2 нить 4
Текст 1 нить 5
Текст 2 нить 5
Текст 1 нить 1
Текст 2 нить 1
Текст 1 нить 6
Текст 2 нить 6
Текст 1 нить 7
Текст 2 нить 7
Текст 1 нить 3
Текст 2 нить 3
Текст 3 нить 3
Текст 3 нить 4
Текст 3 нить 7

С barrier

Без barrier

Директива ordered

Директивы ordered определяют *блок внутри тела цикла*, который должен выполняться в том порядке, в котором итерации идут в последовательном цикле:

#pragma omp ordered

Блок операторов относится к самому внутреннему из объемлющих циклов, а в параллельном цикле должна быть задана опция ordered. Нить, выполняющая первую итерацию цикла, выполняет операции данного блока. Нить, выполняющая любую следующую итерацию, должна сначала дождаться выполнения всех операций блока всеми нитями, выполняющими предыдущие итерации. Может использоваться, например, для упорядочения вывода от параллельных нитей.

Пример25

```
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);        SetConsoleOutputCP(1251);
 int i, n;
 #pragma omp parallel private (i, n)
  {      n=omp_get_thread_num();
#pragma omp for ordered
     for (i=0; i<5; i++)
     {
       printf("Нить %d, итерация %d\n", n, i);
#pragma omp ordered
       {
         printf("ordered: Нить %d, итерация %d\n", n, i);
       }
     }
  }
system("Pause");
}
```

Результаты выполнения примера 25

Нить 0, итерация 0
ordered: Нить 0, итерация 0
Нить 3, итерация 3
Нить 4, итерация 4
Нить 2, итерация 2
Нить 1, итерация 1
ordered: Нить 1, итерация 1
ordered: Нить 2, итерация 2
ordered: Нить 3, итерация 3
ordered: Нить 4, итерация 4
Для продолжения нажмите любую клавишу . . .

Критические секции

С помощью директив critical оформляется критическая секция программы:

#pragma omp critical [(<имя_критической_секции>)]

В каждый момент времени в критической секции может находиться не более одной нити. Если критическая секция уже выполняется какой-либо нитью, то все другие нити, выполнившие директиву для секции с данным именем, будут заблокированы, пока вошедшая нить не закончит выполнение данной критической секции. Как только работавшая нить выйдет из критической секции, одна из заблокированных на входе нитей войдет в неё. Если на входе в критическую секцию стояло несколько нитей, то случайным образом выбирается одна из них, а остальные заблокированные нити продолжают ожидание.

Все неименованные критические секции условно ассоциируются с одним и тем же именем. Все критические секции, имеющие одно и тоже имя, рассматриваются единой секцией, даже если находятся в разных параллельных областях. Побочные входы и выходы из критической секции

Пример 26

```
#include "stdafx.h
#include <iostream>
#include <windows.h>
#include <omp.h>
#include <stdio.h>
#include <dos.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);   SetConsoleOutputCP(1251);
    int n;
#pragma omp parallel
  {
#pragma omp critical
    {        n=omp_get_thread_num();
          cout<<"Нить "<<  n <<"\n";
    }
  }
system("Pause");    }
```

Результаты выполнения примера 26

#pragma omp critical

Нить 0
Нить 1
Нить 4
Нить 3
Нить 2
Нить 6
Нить 5
Нить 7
Для продолжения нажмите любую клавишу . . .

//#pragma omp critical

Если есть критическая секция, то в каждый момент времени фрагмент будет обрабатываться лишь какой-либо одной нитью. Остальные нити, даже если они уже подошли к данной точке программы и готовы к работе, будут ожидать своей очереди. Если критической секции нет, то все нити могут одновременно выполнить данный участок кода. С одной стороны, критические секции предоставляют удобный механизм для работы с общими переменными. Но с другой стороны, пользоваться им нужно осмотрительно, поскольку критические секции добавляют последовательные участки кода в параллельную программу, что может снизить её эффективность.

Директива atomic

Частым случаем использования критических секций на практике является обновление общих переменных. Например, если переменная sum является общей и оператор вида sum=sum+expr находится в параллельной области программы, то при одновременном выполнении данного оператора несколькими нитями можно получить некорректный результат. Чтобы избежать такой ситуации можно воспользоваться механизмом критических секций или специально предусмотренной для таких случаев директивой atomic :

#pragma omp atomic

Данная директива относится к идущему непосредственно за ней оператору присваивания (на используемые в котором конструкции накладываются достаточно понятные ограничения), гарантируя корректную работу с общей переменной, стоящей в его левой части. На время выполнения оператора блокируется доступ к данной переменной всем запущенным в данный момент нитям, кроме нити, выполняющей операцию. Атомарной является только работа с переменной в левой части опе ра

Пример 27

```
#include "stdafx.h
#include <iostream>
#include <windows.h>
#include <omp.h>
#include <stdio.h>
#include <dos.h>
using namespace std;
int main(int argc, char *argv[]){
SetConsoleCP(1251);          SetConsoleOutputCP(1251
 int count = 0;
omp_set_num_threads(50);
#pragma omp parallel
    {
 #pragma omp atomic
       count++;
    }
    printf("Число нитей: %d\n", count);
system("Pause");}
```

Результаты выполнения примера 27

#pragma omp atomic

Число нитей: 50
Для продолжения нажмите любую клавишу . . .

// #pragma omp atomic

Число нитей: 48
Для продолжения нажмите любую клавишу . . .

Замки (locks)

Один из вариантов синхронизации в OpenMP реализуется через механизм замков (locks). В качестве замков используются общие целочисленные переменные (размер должен быть достаточным для хранения адреса). Данные переменные должны использоваться только как параметры примитивов синхронизации.

Замок может находиться в одном из трёх состояний: неинициализированный, разблокированный или заблокированный. Разблокированный замок может быть захвачен некоторой нитью. При этом он переходит в заблокированное состояние. Нить, захватившая замок, и только она может его освободить, после чего замок возвращается в разблокированное состояние.

Использование замков является наиболее гибким механизмом синхронизации, поскольку с помощью замков можно реализовать все остальные варианты синхронизации.

Типы замков (locks)

Простой замок может быть захвачен только однажды.

Множественный замок может многократно захватываться одной нитью перед его освобождением.

Для множественного замка вводится понятие коэффициента захваченности (nesting count). Изначально он устанавливается в ноль, при каждом следующем захватывании увеличивается на единицу, а при каждом освобождении уменьшается на единицу. Множественный замок считается разблокированным, если его коэффициент захваченности равен нулю.

Для инициализации простого или множественного замка используются соответственно функции omp_init_lock() и omp_init_nest_lock():

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

Блокирование

Функции omp_destroy_lock() и omp_destroy_nest_lock() используются для переведения простого или множественного замка в неинициализированное состояние:

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

Для захватывания замка используются функции omp_set_lock() и omp_set_nest_lock():

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

Вызвавшая эту функцию нить дожидается освобождения замка, а затем захватывает его. Замок при этом переводится в заблокированное состояние. Если множественный замок уже захвачен данной нитью, то нить не блокируется, а коэффициент захваченности увеличивается на единицу.

Освобождение замка

Для освобождения замка используются функции omp_unset_lock() и omp_unset_nest_lock():

void omp_unset_lock(omp_lock_t *lock);

void omp_unset_nest_lock(omp_lock_t *lock);

Вызов этой функции освобождает простой замок, если он был захвачен вызвавшей нитью. Для множественного замка уменьшает на единицу коэффициент захваченности. Если коэффициент станет равен нулю, замок освобождается. Если после освобождения замка есть нити, заблокированные на операции, захватывающей данный замок, замок будет сразу же захвачен одной из ожидающих нитей.

Пример 28

```
omp_lock_t lock;
int main(int argc, char *argv[]){
SetConsoleCP(1251);     SetConsoleOutputCP(1251);
 int n;
   omp_init_lock(&lock);
#pragma omp parallel private (n)
   {       n=omp_get_thread_num();
           omp_set_lock(&lock);
           printf("Начало закрытой секции, нить %d\n", n);
                    Sleep(5);
            printf("Конец закрытой секции, нить %d\n", n);
          omp_unset_lock(&lock);
   }
   omp_destroy_lock(&lock);
system("Pause");
}
```

Результаты выполнения примера 28

```
omp_set_lock(&lock);
Начало закрытой секции, нить 0
Конец закрытой секции, нить 0
Начало закрытой секции, нить 1
Конец закрытой секции, нить 1
Начало закрытой секции, нить 3
Конец закрытой секции, нить 3
Начало закрытой секции, нить 5
Конец закрытой секции, нить 5
Начало закрытой секции, нить 4
Конец закрытой секции, нить 4
Начало закрытой секции, нить 2
Конец закрытой секции, нить 2
Начало закрытой секции, нить 6
Конец закрытой секции, нить 6
Начало закрытой секции, нить 7
Конец закрытой секции, нить 7
```

Проверка состояния замка

Для неблокирующей попытки захвата замка используются функции omp_test_lock() и omp_test_nest_lock():

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_lock_t *lock);
```

Данная функция пробует захватить указанный замок. Если это удалось, то для простого замка функция возвращает 1, а для множественного замка – новый коэффициент захваченности. Если замок захватить не удалось, в обоих случаях возвращается 0.

Пример 29

```
omp_lock_t lock;
int main(int argc, char *argv[]){
SetConsoleCP(1251);        SetConsoleOutputCP(1251
  omp_lock_t lock;
  int n;
  omp_init_lock(&lock);
#pragma omp parallel private (n)
  {      n=omp_get_thread_num();
    while (!omp_test_lock (&lock))
    {   printf("Секция закрыта, нить %d\n", n);       Sleep(2);       }
    printf("Начало закрытой секции, нить %d\n", n);
    Sleep(5);
    printf("Конец закрытой секции, нить %d\n", n);
    omp_unset_lock(&lock);
  }
  omp_destroy_lock(&lock);
system("Pause");
}
```

Результаты выполнения примера 29

```
Начало закрытой секции, нить 0
Секция закрыта, нить 1
Секция закрыта, нить 5
Секция закрыта, нить 4
Секция закрыта, нить 2
Секция закрыта, нить 3
Секция закрыта, нить 6
Секция закрыта, нить 7
Конец закрытой секции, нить 0
Секция закрыта, нить 7
Секция закрыта, нить 2
Секция закрыта, нить 5
Секция закрыта, нить 1
Секция закрыта, нить 6
Секция закрыта, нить 3
Секция закрыта, нить 4
Начало закрытой секции, нить 6
Секция закрыта, нить 3
Секция закрыта, нить 4
Секция закрыта, нить 5
Секция закрыта, нить 7
Секция закрыта, нить 1
Секция закрыта, нить 2
Секция закрыта, нить 7
Конец закрытой секции, нить 6
Секция закрыта, нить 5
Секция закрыта, нить 1
Секция закрыта, нить 4
```

Директива flush

#pragma omp flush [(список)]

Выполнение данной директивы предполагает, что значения всех переменных (или переменных из списка, если он задан), временно хранящиеся в регистрах и кэш-памяти текущей нити, будут занесены в основную память; все изменения переменных, сделанные нитью во время работы, станут видимы остальным нитям; если какая-то информация хранится в буферах вывода, то буферы будут сброшены и т.п. При этом операция производится только с данными вызвавшей нити, данные, изменявшиеся другими нитями, не затрагиваются. Поскольку выполнение данной директивы в полном объёме может повлечь значительных накладных расходов, а в данный момент нужна гарантия согласованного представления не всех, а лишь отдельны

Неявно flush без параметров присутствует в директиве barrier, на входе и выходе областей действия директив parallel, critical, ordered, на выходе областей распределения работ, если не используется опция nowait, в вызовах функций omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), если при этом замок устанавливается или снимается, а также перед порождением и после завершения любой задачи (task). Кроме того, flush вызывается для переменной, участвующей в операции, ассоциированной с директивой atomic. Заметим, что flush не применяется на входе области распределения работ, а также на входе и выходе