«Рекурсивный» мьютекс

Установкой специального атрибута можно позволить многократно захватывать мьютекс в одном потоке.

Пусть ***S*** **-** ***счетчик блокировок.***
При повторном захвате ***S*** **=** ***S*** + 1.
При разблокировании ***S*** **=** ***S*** – 1.
Если ***S*** = 0, то мьютекс доступен для других потоков.

Пример установки атрибута рекурсивности:

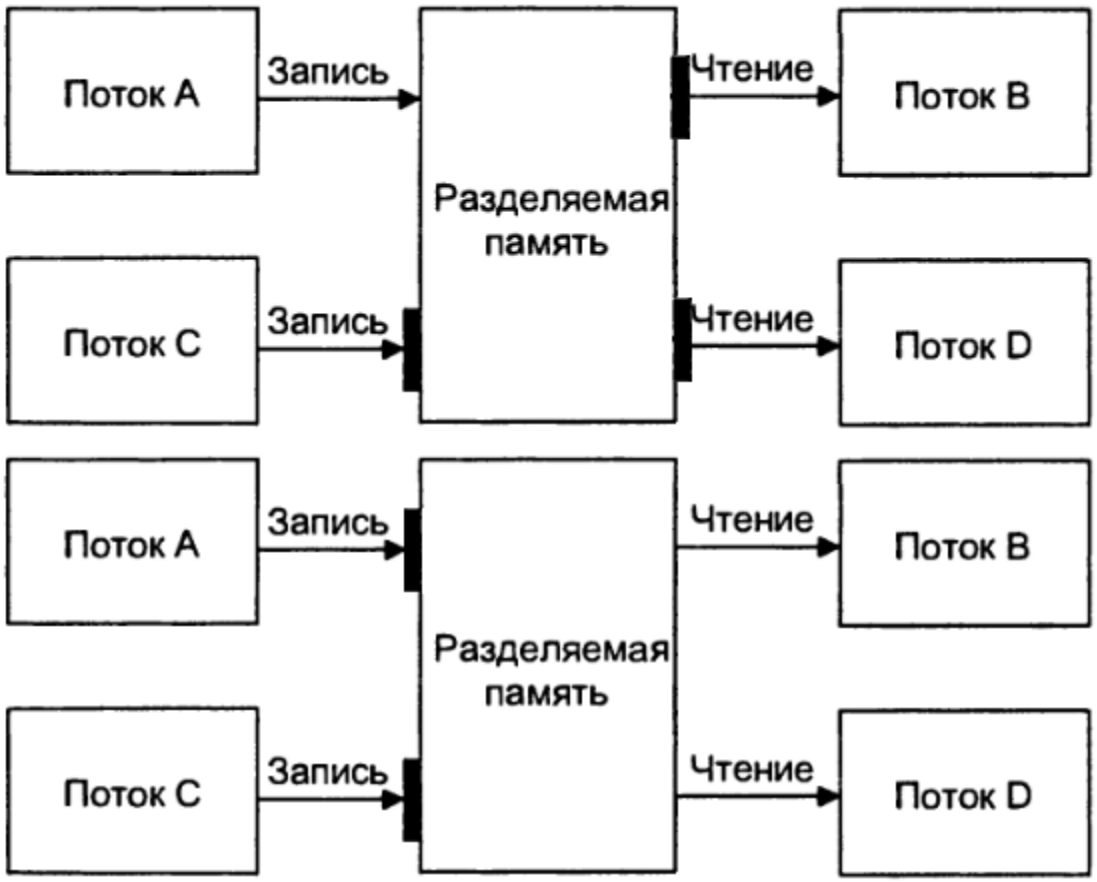
```
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&m, &attr);
pthread_mutexattr_destroy(&attr);
```

```
class vect {
vector <int> v;
pthread_mutex_t m;
vect () {
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&m, &attr);
pthread_mutexattr_destroy(&attr);
}
int size() {
pthread_mutex_lock(&m);
return v.size();
pthread_mutex_unlock(&m);
}
void add(int val) {
pthread_mutex_lock(&m);
if(size()<100)
v.push_back(val);
pthread_mutex_unlock(&m);
}};
```

Блокировки чтения/записи

Владеть блокировкой **чтения** может любое количество потоков.

Если некоторый поток владеет блокировкой **записи**, то любой другой будет ждать.



Тип: `pthread_rwlock_t`

- запрос на чтение:

`pthread_rwlock_rdlock`,
`pthread_rwlock_timedrdlock`

- запрос на запись:

`pthread_rwlock_wrlock`,
`pthread_rwlock_timedwrlock`

- освобождение:

`pthread_rwlock_unlock`

- тестирование:

`pthread_rwlock_tryrdlock`,
`pthread_rwlock_trywrlock`

Условные переменные

обеспечивают блокирование одного или нескольких потоков до момента поступления сигнала от другого потока.

Используются для синхронизации последовательности операций.

Тип: `pthread_cond_t`

- ожидание:

`pthread_cond_wait`

или

`pthread_cond_timedwait`

- сигнал:

`pthread_cond_signal` – **одному** из ожидающих

или

`pthread_cond_broadcast` – **всем**.

pthread_cond_wait

```
//. . .
pthread_mutex_lock (&Mutex);
pthread_cond_wait  (&EventMutex, &Mutex);
//. . .
pthread_mutex_unlock(&Mutex);
```

При вызове `cond_wait` **Mutex** будет **освобожден**, а **поток** будет **заблокирован** до получения сигнала.

После получения сигнала **поток** будет **разблокирован и получит Mutex**.

```
#define SYNC_MAX_COUNT 10

void SynchronizationPoint() {
static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
static cond_t sync_cond = PTHREAD_COND_INITIALIZER;
static int sync_count = 0;

/* блокировка доступа к счетчику */
pthread_mutex_lock(&sync_lock);

sync_count++;

/* проверка: следует ли продолжать ожидание */
if (sync_count < SYNC_MAX_COUNT)
pthread_cond_wait(&sync_cond, &sync_lock);
else
/* оповестить о достижении данной точки всеми */
pthread_cond_broadcast(&sync_cond);

/* активизация взаимной блокировки - в противном случае
из процедуры сможет выйти только одна нить! */
pthread_mutex_unlock(&sync_lock);
}
```

Барьер

позволяет приостановить выполнение потоков в некоторой точке программы до тех пор, пока все потоки (**count**) не дойдут до этого места.

Тип: `pthread_barrier_t`

- создание:

```
pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *attr, unsigned count);
```

- ожидание:

```
pthread_barrier_wait(pthread_barrier_t *barrier);
```

Семафор

В отличие от мьютекса его может разблокировать любой поток.

Тип: `sem_t`

`sem_init(sem_t *sem, int pshared, unsigned int value);` - создание, `value` – начальное значение.

`sem_wait(sem_t *sem);`

уменьшает на 1 текущее значение и, если значение отрицательное, то переходит к ожиданию.

`sem_post(sem_t *sem);`

увеличивает текущее значение на 1 и, если прежнее значение 0, разблокирует один из ожидающих потоков.

Завершение потока

```
void pthread_exit(void *value)
```

Завершает выполнение того потока, из которого была вызвана.

Действие эквивалентно оператору: `return value;`

Завершение потока по инициативе другого потока

```
int pthread_cancel(pthread_t t);
```

запрос на аннулирование потока, определяемого идентификатором t.

Готовность и тип аннулирования

```
int pthread_setcancelstate(int state,int
*oldstate);
```

PTHREAD_CANCEL_ENABLE – **разрешено**;

PTHREAD_CANCEL_DISABLE – **запрещено**.

```
int pthread_setcanceltype(int type, int
*oldtype);
```

PTHREAD_CANCEL_DEFERRED – **отсроченное**;

PTHREAD_CANCEL_ASYNCHRONOUS – **немедленное**.

Точки аннулирования

В точке вызова **`pthread_testcancel`**`()` проверяется наличие и обрабатываются запросы на аннулирование.

Очистка перед завершением

```
void pthread_cleanup_push(void(* routine)
(void *), void *arg);
```

Помещает в стек вызывающего потока функцию, которая будет вызвана при аннулировании.

```
void pthread_cleanup_pop (int execute);
```

Извлекает функцию расположенную в вершине.

execute=1: с выполнением функции;

execute=0: без выполнения.

Стандарт OpenMP

- Расширение языков C/C++, Fortran.
- Включает в свой состав: директивы управления параллелизмом, собственную библиотеку функций (**omp.h**), а также специальные переменные окружения.
- *Поведение параллельной программы может зависеть от реализации компилятора*.

Синтаксис директив

#pragma omp <директива> [опция [[,] опция]...] <**конец строки**>

Активизация OpenMP в Visual C++

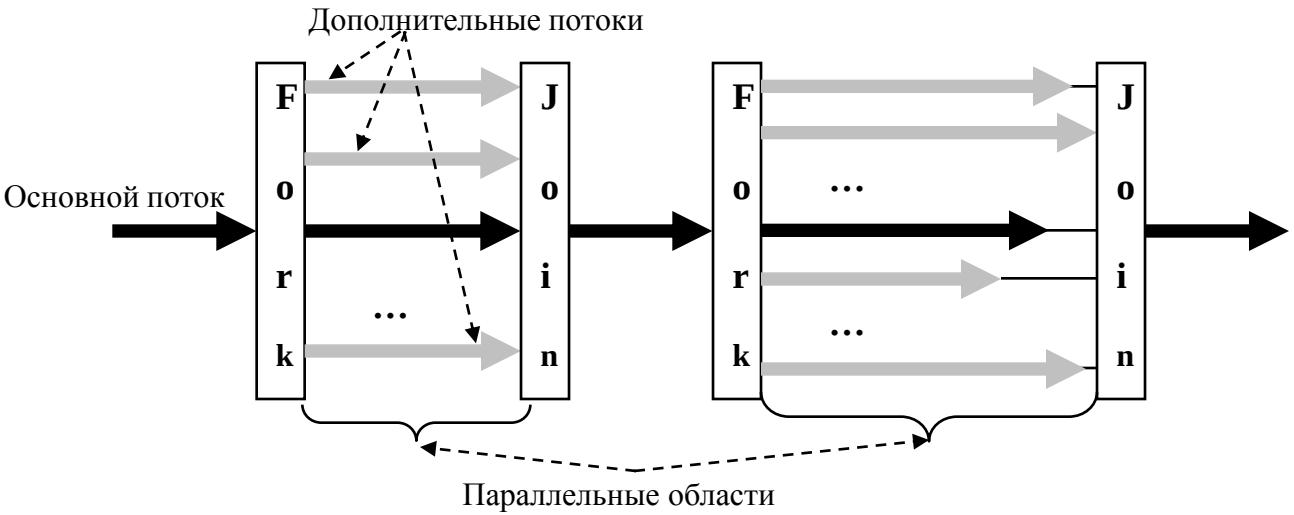
- Параметр компилятора /openmp
- Свойства проекта, Configuration Properties, C/C++, Language, OpenMP Support.)

Версия OpenMP

```
#include <stdio.h>
void main()
{
#ifdef _OPENMP
        printf("Version OpenMP is %d\n", _OPENMP);
#endif
}
```

Модель выполнения OpenMP-программ

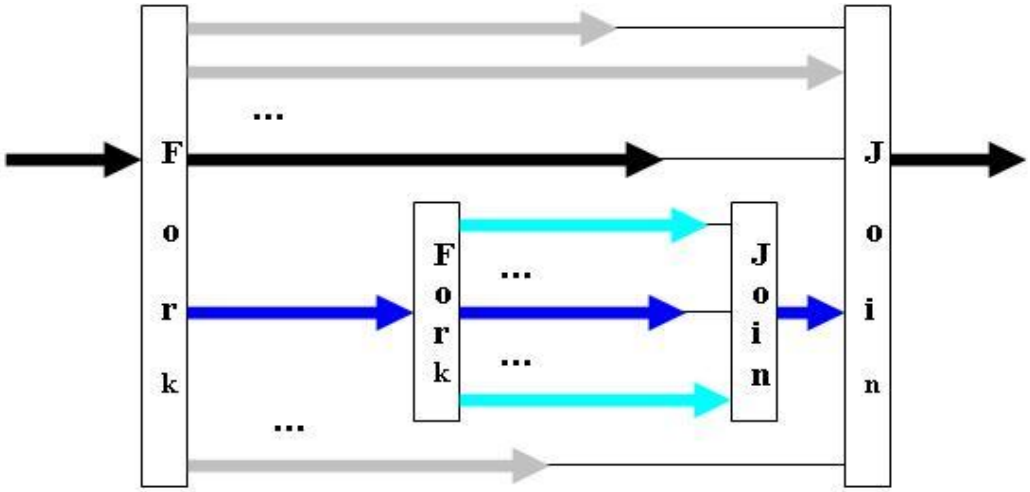
- В момент запуска программы порождается единственный поток -«основной».
- Основной поток может создать параллельную область, выполняемую им и несколькими дополнительными потоками.
- Только основной поток исполняет все последовательные области программы, находящиеся между параллельными областями.



Вложенные параллельные области

void omp_set_nested(int val);

Вызов с параметрами 1 или 0 разрешает или запрещает вложенность.



Порождение потоков – директива parallel

```
#pragma omp parallel
{
    printf("Hello World\n");
}
```

```
Hello World
Hello World
```

```
HellHell oo WorWlodrl
d
```

- Основной поток всегда имеет номер 0.
- Остальные потоки получают номера 1, 2, …
- Число потоков, которые будут выполнять параллельную область, определяется:
  - переменной окружения **OMP_NUM_THREADS;**
  - вызовом функции **omp_set_num_threads(int);**
  - опцией **num_threads.**
- Каждый поток может определить свой номер и количество потоков:
  - свой номер: **omp_get_thread_num()**
  - число потоков: **omp_get_num_threads()**