Средства синхронизации

Основные понятия которые выходит за рамки конкретного стандарта или языка и являются общими терминами.

Критическая секция

— участок программы, который не должен выполняться одновременно (*в нем обычно производится работа с общим ресурсом*).

Примитивы синхронизации

– объекты языка программирования или операционной системы служащие для координации взаимодействия потоков или процессов.

Некоторые операции с ними приостанавливают выполнение потока, до появления внешнего события.

Активное ожидание (busy waiting / spinning)

- проверка некоторого условия в пустом цикле.

\+ *Может быть легко реализовано на уровне прикладной программы.*

– *При длительном ожидании неэффективно, т.к. перегружает процессор.*

Примитив синхронизации реализующий активное ожидание обычно называют

спинлок (spinlock).

Пассивное ожидание

реализуется с помощью операционной системы, которая выгружает поток до наступления ожидаемого события.

\+ *Позволяет другим потокам в полной мере использовать ресурсы.*

– *При коротком ожидании неэффективно, т.к. на сохранение/загрузку контекста потока требуется время.*

Гибридные средства синхронизации

– попытка объединить достоинства активного и пассивного ожидания: в начале потоки находятся в активном состоянии, а если ожидание затягивается, то ОС выгружает ожидающий поток.

Так реализованы большинство современных высокоуровневых средств.

Средства описания и анализа работы параллельных алгоритмов в UML 2.0 (*Unified Modeling Language*).

Диаграмма последовательности (*sequence diagram*)

Иллюстрирует жизненный цикл нескольких объектов.

Основные элементы:

1. прямоугольники – объекты;
2. прямоугольники – деятельность объектов;
3. стрелки – взаимодействия.

Может использоваться для моделирования и анализа различных вариантов поведения потоков.

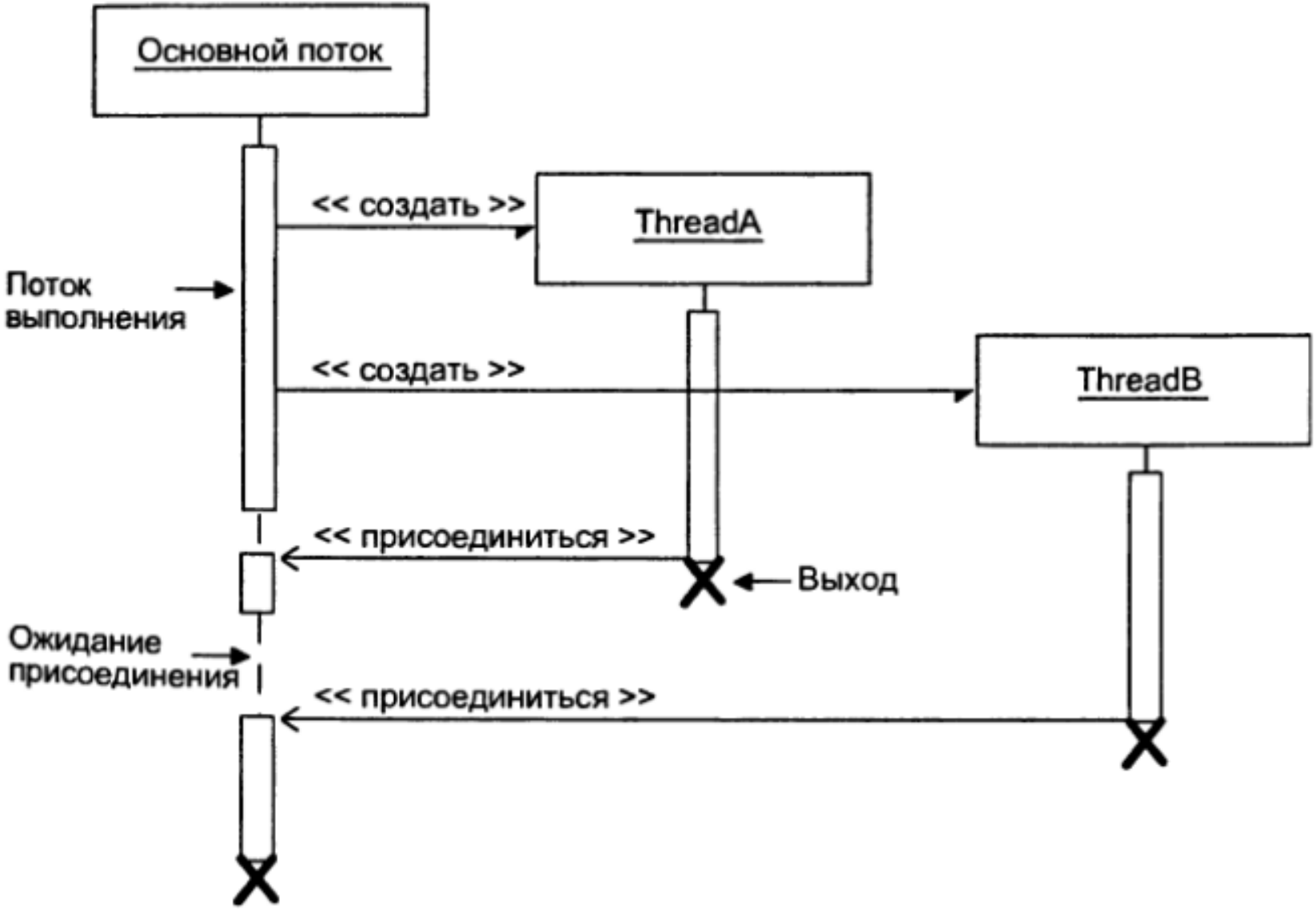
Основной поток
<< создать >>
ThreadA
Поток
выполнения
<< создать >>
ThreadB
<< присоединиться >>
Выход
Ожидание
присоединения
<< присоединиться >>

Диаграмма деятельности (*activity diagram*)

Имеет много общего с блок-схемой, содержит ряд средств подходящих ***для описания параллельного алгоритма***.

Точка разделения
и
Точка слияния

Прием события (receive event action)

Ожидание некоторого события.

Передача сигнала (send signal action)

передает сигнал внешней системе (*другому потоку*).

POSIX

(Portable Operating System Interface for Unix)

- *набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой.*

POSIX Threads – стандарт определяющий API для управления потоками.

Создание потоков

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

thread – дескриптор потока (будет записан);

attr – атрибуты потока (NULL - по умолчанию);

start_routine – адрес функции которая будет выполняться в созданном потоке;

arg – аргумент функции по адресу start_routine.

Возвращаемое значение 0 или код ошибки.

Присоединяемые и открепленные потоки

Открепленным называется поток, который не синхронизирован с другими потоками (*не существует потока, который мог бы дожидаться его завершения*).

По умолчанию потоки создаются как присоединяемые.

Для открепления можно использовать функцию

int pthread_detach(pthread_t tid);

Присоединение потоков

```
int pthread_join(pthread_t threadid, void **value_ptr);
```

Переводит поток, из которого она была вызвана в состояние ожидания до тех пор, пока не завершится поток, определяемый идентификатором threadid.

Необходимо вызывать для всех присоединяемых потоков с целью освобождения ресурсов

Значение возвращаемое функцией выполняемой потоком

int pthread_join (pthread_t thread, void **status_addr);

Функция pthread_join блокирует работу вызвавшего ее потока до завершения потока с идентификатором thread.

После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул завершившийся thread.

Пример

```
int w;
void *task1(void *X)
{
//w=2;
//return &w;
int *st=new int;
*st=3;
return st;
}
```

```
int main(int argc, _TCHAR* argv[])
{
pthread_t thread_a;
pthread_create(&thread_a, NULL, task1, NULL);
int s;
void *ss;
pthread_join(thread_a, &ss);
s=*((int*)ss);
cout << endl << "s=" << s << endl;
system("pause");
return 0;
}
```

Завершение потока

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

•Завершает выполнение того потока, из которого была вызвана.

•Действие функции `pthread_exit(value)` эквивалентно оператору:

```
return value;
```

Мьютексный семафор (Mutex от ***MUTual EXclusion***)

можно рассматривать как ключ к ресурсам, которым одновременно может владеть только один поток.

При попытке завладеть «уже занятым» мьютексом поток будет заблокирован до тех пор, пока мьютекс не станет доступен.

Тип: `pthread_mutex_t`

запрос на монопольное использование:

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

или

```
pthread_mutex_timedlock(pthread_mutex_t *mutex,
const struct timespec *abstime);
```

отказ от монопольного использования:

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

тестирование монопольного использования:

```
pthread_mutex_trylock(pthread_mutex_t * mutex);
```

«Рекурсивный» мьютекс

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

При каждом очередном захвате счетчик блокировок инкрементируется,
а при каждом разблокировании — декрементируется.

Если счетчик блокировок = 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);
}};
```

Завершение потока по инициативе другого потока

```
int pthread_cancel(pthread_t threadid);
```

запрос на аннулирование потока, определяемого идентификатором threadid.

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

Функция **`pthread_testcancel()`** определяет точку аннулирования.

Она проверяет наличие необработанных запросов на аннулирование. Если они есть, активизирует процесс аннулирования в точке своего вызова.

Некоторые функции библиотеки Pthread являются точками аннулирования

Очистка перед завершением

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

Помещает в стек вызывающего потока функцию предназначенную для выполнения подготовительных действий по аннулированию потока.

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

Извлекает функцию расположенную в вершине стека вызывающего потока.

execute=1: с выполнением функции;

execute=0: без выполнения.