ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 18 — Условные переменные

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.05.30

Оглавление

Условные переменные[1] (conditional variables)

Условная переменная — примитив синхронизации, обеспечивающий блокирование одного или нескольких потоков в состоянии ожидания выполнения некоторого условия до момента поступления сигнала от другого потока о том, что данное условие выполнено (или до истечения максимального промежутка времени ожидания).

Условные переменные используются вместе с ассоциированным с ними мьютексом и фактически представляют собой совокупность объекта синхронизации **cond**, предиката **P** и мьютекса **mutex**. Типичный шаблон использования переменных состояния:

```
// безопасно проверим состояние/условие предиката P, и чтобы защитить в в этот момент
// предикат от изменения другими потоками захватываем мьютекс
lock(mutex);
while (P is false) {
    wait(cond, mutex); // блокируемся на переменной cond, mutex, связанный с ней
}                      // при этом освобождается и можеть быть получен другим потоком
do_stuff();            // нас разблокировали и вернули мьютекс — делаем наши дела
unlock(mutex);         // и отдаем мьютекс
```

С другой стороны, поток, сигнализирующий о переменной условия, обычно выглядит так:

```
lock(mutex);   // нам нужен эксклюзивный доступ, каким бы условие не было
set(P, true)   // делаем наши дела и меняем предикат состояния, после чего
signal(cond);  // разбудим по крайней мере один поток из ожидающих изменения P в true
               // и заблокированных на условной переменной cond
unlock(mutex)  // отдаем мьютекс
```

1) Переменные состояния

Как таковой, никакой переменной нет. Концептуально, условная переменная **Cond** — это очередь потоков, ассоциированных с совместно используемым объектом данных (сущностью), которые ожидают выполнения некоторого условия $\mathbf{P}_{cond}$, связанного с состоянием объекта (сущности).

Когда поток находится в состоянии ожидания условной переменной, он не является владеющим сущностью, связанной с этой условной переменной, поэтому другой поток может изменить совместно используемый объект и просигнализировать ожидающим потокам о возможном выполнении условия.

Два потока имеют общую структуру данных, доступ к которой и защищает мьютекс.

Первый поток хочет дождаться, пока какое-то условие станет истинным, а затем немедленно выполнить некоторую операцию без возможности возникновения состояния гонки, когда какой-либо другой поток войдет между проверкой условия и действием и сделает условие ложным.

Второй поток делает что-то, что может сделать условие истинным, чтобы оно разбудило всех, кто его ожидает.

Пример – задача «Производители-потребители»

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

#define __USE_XOPEN_EXTENDED // требуется при -std=C11
#include <unistd.h>
#include <pthread.h>

#define STORAGE_MIN 10
#define STORAGE_MAX 20
```

```
int storage = STORAGE_MIN; // Общая для всех потоков переменная
pthread_mutex_t mutex;     // мьютекс
pthread_cond_t  cond;      // объект синхронизации "условная переменная"

//
// Функция потока-потребителя
//
void* consumer(void *args)
{
    puts("Поток [CONSUMER] стартовал");
    int to_consume = 0;

    while (1) {
        pthread_mutex_lock(&mutex);
        // Если значение общей переменной меньше максимального, то поток входит
        // в состояние ожидания сигнала о достижении максимума
        while (storage < STORAGE_MAX) {       // (storage < STORAGE_MAX) -- предикат
            pthread_cond_wait(&cond, &mutex); // отдаем mutex и блокируемся на cond
        }
        to_consume = storage - STORAGE_MIN; // определим сколько можем "унести"
        printf("[CONSUMER] storage is maximum, consuming %d\n", to_consume);
        // "Потребление" допустимого объема из значения общей переменной
        storage -= to_consume;
        printf("[CONSUMER] storage = %d\n", storage);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```
//
// Функция потока-производителя
//
void* producer(void *args) {

    puts("Поток [PRODUCER] стартовал");

    while (1) {
        usleep(200000);
        pthread_mutex_lock(&mutex);
        // Производитель постоянно увеличивает значение общей переменной
        ++storage;
        printf("[PRODUCER] storage = %d\n", storage);
        // Если значение общей переменной достигло или превысило
        // максимум, поток потребитель уведомляется об этом
        if (storage >= STORAGE_MAX) {
            puts("[PRODUCER] заполнил storage до максимума");
            pthread_cond_signal(&cond); // разблокируем потоки, ждущие заполнения
        }
        pthread_mutex_unlock(&mutex);   // отдаем mutex
    }
    return NULL;
}
```

```
int main(int argc, char *argv[]) {

    pthread_t producer;
    pthread_t consumer;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);

    int res = pthread_create(&producer, NULL, producer, NULL);
    if (res != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&consumer, NULL, consumer, NULL);
    if (res != 0) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }

    pthread_join(producer, NULL);
    pthread_join(consumer, NULL);

    return EXIT_SUCCESS;
}
```

pthread_cond_init()/destroy() — инициализировать/уничтожить условную переменную

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t           *restrict cond,
                      const pthread_condattr_t *restrict attr);

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER; // если атрибуты по умолчанию
```

Функция **pthread_cond_init()** инициализирует условную переменную, на которую ссылается **cond**, с атрибутами, на которые ссылается **attr**.

Если **attr** — **NULL**, будут использоваться атрибуты переменной условия по умолчанию.

Эффект такой же, как при передаче адреса объекта атрибутов переменной условия по умолчанию.

После успешной инициализации состояние переменной условия должно быть инициализировано.

Функция **pthread_cond_destroy()** уничтожает условную переменную условия, заданную **cond**, в результате чего объект фактически становится неинициализированным.

Реализация допускает, чтобы **pthread_cond_destroy()** устанавливал объект, на который ссылается **cond**, в недопустимое значение.

Уничтоженный объект условной переменной можно повторно инициализировать с помощью **pthread_cond_init()**.

UB

¤ Результат ссылок на объект после его уничтожения не определен.

¤ Попытка инициализировать уже инициализированную условную переменную приводит к неопределенному поведению.

Уничтожение инициализированной условной переменной, с помощью которой в настоящее время не заблокированы никакие потоки, является безопасным.

¤ Попытка уничтожить условную переменную, из-за которой в настоящее время заблокированы другие потоки, приводит к неопределенному поведению.

В случаях, когда атрибуты переменных условия по умолчанию годятся, для инициализации переменных условия можно использовать макрос **PTHREAD_COND_INITIALIZER**. Эффект должен быть эквивалентен динамической инициализации вызовом **pthread_cond_init()** с параметром **attr**, заданным как **NULL**, за исключением того, что не выполняются проверки ошибок.

¤ Если значение, указанное аргументом **cond** для **pthread_cond_destroy()**, не относится к инициализированной переменной условия, поведение не определено.

¤ Если значение, указанное аргументом **attr** для **pthread_cond_init()**, не относится к инициализированному объекту атрибутов переменной условия, поведение не определено.

Возвращаемое значение

В случае успеха функции **pthread_cond_destroy()** и **pthread_cond_init()** возвращают ноль, в противном случае номер ошибки.

Ошибки

Функция **pthread_cond_init()** завершится ошибкой, если:

EAGAIN — Системе не хватало необходимых ресурсов (кроме памяти) для инициализации другой переменной условия.

ENOMEM — Недостаточно памяти для инициализации переменной условия.

pthread_cond_wait()/timedwait() — блокироваться на условной переменной и ожидать сигнала

```
 #include <pthread.h>

int pthread_cond_wait(pthread_cond_t  *restrict cond,
                      pthread_mutex_t *restrict mutex);

int pthread_cond_timedwait(pthread_cond_t        *restrict cond,
                           pthread_mutex_t       *restrict mutex,
                           const struct timespec *restrict abstime);
```

Функции **pthread_cond_timedwait()** и **pthread_cond_wait()** заставляют вызывающий поток блокироваться по условной переменной **cond** и *атомарно* освобождают **mutex**.

«Атомарно» здесь означает «атомарно относительно доступа другого потока к мьютексу и затем к условной переменной». Т.е., если другой поток может захватить мьютекс после того, как его освободил поток, который вот-вот будет заблокирован, то последующий за захватом мьютекса вызов **pthread_cond_signal()** или **pthread_cond_broadcast()** в этом потоке будет вести себя так, как если бы он был выполнен после того, как блокируемый поток был уже заблокирован.

Гарантию того, что эти функции вызываются с мьютексом, заблокированным вызывающим потоком, должно давать приложение.

- В противном случае возникает ошибка (для **PTHREAD_MUTEX_ERRORCHECK** и робастных мьютексов) или неопределенное поведение (для остальных мьютексов).
- Если значение, указанное аргументом **cond** или **mutex** для этих функций, не относится к инициализированной условной переменной или инициализированному объекту **mutex**, поведение, соответственно, не определено.

После успешного возврата из функции мьютекс будет заблокирован и будет принадлежать потоку, вызвавшему функцию.

Если мьютекс является робастным мьютексом, а его владелец завершил работу, удерживая блокировку, и состояние защищаемого объекта при этом может быть восстановлено, мьютекс будет получен, даже если функция возвратит код ошибки.

pthread_cond_timedwait()

Функция **pthread_cond_timedwait()** эквивалентна функции **pthread_cond_wait()** за исключением того, что если абсолютное время, заданное параметром **abtime**, прошло (т.е. системное время равно или превышает указанное в **abtime**) до того, как для условной переменной **cond** были вызваны **signal()** или **broadcast()**, будет возвращена ошибка.

Когда происходят такие тайм-ауты, **pthread_cond_timedwait()**, тем не менее, освобождает и повторно захватывает мьютекс, на который ссылается **mutex**, и при этом может конкурентно «пропустить» **signal()**, отправленный по условной переменной.

Условная переменная имеет атрибут часов (clock attribute), указывающий часы, которые должны использоваться для измерения времени, указанного аргументом **abstime** в случае вызова **cond_timedwait()**.

Если потоку, ожидающему переменную условия, доставляется сигнал, то после возврата из обработчика сигнала может быть два варианта поведения:

- поток возобновляет ожидание переменной условия, как если бы он не был прерван;
- вызов возвращает ноль (успешное завершение) из-за ложного пробуждения.

Ложные пробуждения (Spurious wakeups)

При использовании условных переменных всегда существует логический предикат, включающий общие переменные, связанные с каждым условием ожидания, которое должно быть истинно, если поток должен продолжить выполнение. При этом из функций **`pthread_cond_wait()`** или **`pthread_cond_timedwait()`** могут происходить ложные пробуждения заблокированных потоков.

Поскольку возврат из `cond_wait()` или `cond_timedwait()` ничего не говорит о значении этого предиката, этот предикат должен повторно вычисляться всякий раз при возврате.

Когда поток блокируется на условной переменной, указав конкретный мьютекс для **`pthread_cond_wait()`** или для **`pthread_cond_timedwait()`**, между этим мьютексом и условной переменной формируется динамическая связь, которая остается в силе до тех пор, пока хотя бы один поток остается заблокированным по этой условной переменной.

Как только все ожидающие потоки будут разблокированы (например, с помощью операции **`pthread_cond_broadcast()`**), новую динамическую связь с мьютексом, указанным в этой операции **`wait`**, сформирует следующая операция **`wait`** для этой условной переменной.

¤ Эффект от попытки любого потока заблокироваться в течение этого времени на этой условной переменной с использованием ***другого мьютекса*** не определен.

Несмотря на то, что динамическая связь между условной переменной и мьютексом может быть удалена или замещена в промежутке между моментом, когда поток разблокируется после ожидания условной переменной, и моментом, когда он возвращает управление вызывающей стороне или начинает очистку отмены, незаблокированный поток будет всегда повторно владеть мьютексом, указанном в вызове операции ожидания условия, из которого он возвращается.

Возвращаемое значение

За исключением **ETIMEDOUT**, **ENOTRECOVERABLE** и **EOWNERDEAD**, все проверки на ошибки под капотом **pthread_cond_[timed]wait()** работают так, как если бы они выполнялись непосредственно в начале функции, и вызывают возврат ошибки до изменения состояния мьютекса, заданного **mutex**, или условной переменной, указанной в **cond**.

При успешном завершении возвращается ноль, в противном случае номер ошибки.

Ошибки

ENOTRECOVERABLE — Состояние, защищенное мьютексом, восстановить невозможно.

EOWNERDEAD — Мьютекс является робастным мьютексом, и процесс, содержащий предыдущий поток-владелец, завершается, удерживая блокировку мьютекса. Блокировка мьютекса будет получена вызывающим потоком, и новый владелец должен сделать состояние согласованным.

EPERM — Тип мьютекса **PTHREAD_MUTEX_ERRORCHECK**, или мьютекс является робастным мьютексом, при этом текущий поток не владеет мьютексом.

Функция **pthread_cond_timedwait()** завершится ошибкой:

ETIMEDOUT — если время, указанное параметром **abstime** для **pthread_cond_timedwait()**, истекло.

EINVAL — если аргумент **abstime** указывает наносекундное значение меньше нуля или больше или равно 1000 миллионов.

Функции могут завершится ошибкой:

EINVAL — **cond** и/или **mutex** не относятся к нинициализированным условной переменной и/или мьютексу, соответственно.

Семантика ожидания условия

Даже в тех случаях, когда **`pthread_cond_wait()`** и **`pthread_cond_timedwait()`** возвращаются без ошибок, связанный с условной переменной предикат все еще может быть ложным.

Когда **`pthread_cond_timedwait()`** возвращается с ошибкой тайм-аута (**ETIMEDOUT**), связанный с условной переменной предикат может быть истинным – причиной является неизбежная гонка между истечением тайм-аута и изменением состояния предиката.

Приложению необходимо всегда повторно проверять предикат при любом возврате

Некоторые реализации, особенно в многопроцессорных системах, могут иногда вызывать пробуждение нескольких потоков, если сигнал по условной переменной возникает одновременно на разных процессорах.

В общем, всякий раз, когда вызов ожидания условие возвращается, поток должен повторно вычислить предикат, связанный с условием ожидания, чтобы определить, может ли он безопасно продолжить, должен ли он снова ждать или должен объявить тайм-аут.

Возврат из ожидания не означает, что связанный предикат является истинным или ложным.

Таким образом, рекомендуется заключать условие ожидания в эквивалент цикла `while`, в котором проверяется предикат

Семантика ожидания в течение ограниченного времени (Timed Wait Semantics)

Для указания параметра тайм-аута была выбрана абсолютная мера времени. Это сделано по двум причинам.

Во-первых, относительная мера времени может быть легко реализована поверх функции, которая использует абсолютное время.

В то же время возможно возникновение условий гонки, связанных с указанием абсолютного тайм-аута для функции, которая использует относительные тайм-ауты.

Например, предположим, что **`clock_gettime()`** возвращает текущее время, а **`cond_relative_timed_wait()`** использует относительные таймауты:

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

Если нить вытесняется между первым и последним операторами, она может быть заблокирована на условной переменой на слишком длительное время. Однако, если используется абсолютный тайм-аут, пробуждение произойдет в заданный момент времени.

Абсолютный тайм-аут также не нужно пересчитывать, если он используется несколько раз в цикле, например, в цикле условного ожидания.

В случаях, когда системные часы были скачком переведены оператором, любое рассчитанное по времени ожидание, истекающее в промежуточный момент времени, будет обрабатываться, как если бы это время действительно произошло.

Отмена и ожидание условия (Cancellation and Condition Wait)

Ожидание условия, будь то ограниченное по времени или нет, является точкой отмены – функции **`pthread_cond_[timed]wait()`** являются точками, где может быть замечен запрос отмены.

Причина такого положения состоит в том, что в этих точках возможно неопределенное ожидание — какое бы событие ни ожидалось, даже если программа полностью верна, оно может никогда не произойти. Например, некоторые ожидаемые входные данные могут никогда не быть отправлены. Сделав условие ожидания точкой отмены, поток можно отменить и выполнить свой обработчик очистки, даже если поток может застрять в некотором неопределенном ожидании.

Если для типа отменяемости потока задано значение **`PTHREAD_CANCEL_DEFERRED`**, побочным эффектом действия по запросу отмены, если поток заблокирован по условной переменной, является повторное получение мьютекса до вызова первого обработчика очистки. Эффект такой, как если бы поток был разблокирован, он выполняется до точки возврата из вызова **`[timed]wait()`**, но в этот момент он замечает запрос на отмену и вместо возврата управления стороне запускает действия по отмене потока, включая вызов обработчиков очистки отмены.

Такое поведение гарантирует, что обработчик очистки отмены выполняется в том же состоянии, что и критический код, который находится как до, так и после вызова функции ожидания условия.

Это особенно важно при взаимодействии с потоками POSIX из C ++, реализация которого может выбрать отображение отмены на исключение (exception). Оно гарантирует, что каждый обработчик исключений, оборачивающий критическую секцию, всегда может безопасно зависеть от того факта, что связанный мьютекс уже заблокирован, независимо от того, где именно в критической секции возникло исключение. Без этого не было бы единого правила, которому могли бы следовать обработчики исключений в отношении блокировки, и поэтому кодирование было бы очень громоздким.

Производительность мьютексов и условных переменных

Ожидается, что мьютексы будут заблокированы только для нескольких инструкций.

Эта практика почти автоматически выполняется из-за желания программистов избегать длинных последовательных областей выполнения (что снизило бы общий эффективный параллелизм).

При использовании мьютексов и условных переменных стараются гарантировать, что обычным случаем является блокировка мьютекса, доступ к общим данным и разблокировка мьютекса.

Ожидание переменной условия является относительно редкой ситуацией. Например, в случае блокировки чтения-записи коду, который получает блокировку чтения, обычно требуется только увеличить счетчик читателей, используя мьютекс, и вернуться. Вызывающий поток фактически будет ждать переменной условия только тогда, когда уже есть активный писатель.

Таким образом, эффективность операции синхронизации ограничивается стоимостью блокировки/разблокировки мьютекса, а не ожиданием условий. При этом нужно заметить, что в обычном случае нет переключения контекста.

Однако, это не означает, что эффективность ожидания на условной переменной не важна. Например, для каждого рандеву Ada будет хотя бы одно переключение контекста, соответственно, эффективность ожидания условной переменной таки важна.

Стоимость ожидания условной переменной будет немного больше, чем минимальная стоимость переключения контекста плюс время на разблокировку и блокировку мьютекса.

pthread_cond_signal(), pthread_cond_broadcast() — сигнализировать о состоянии одному/всем

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t    *cond);
```

Эти функции разблокируют потоки, заблокированные на условной переменной.

Функция **pthread_cond_broadcast()** разблокирует все потоки, заблокированные в данный момент для указанной условной переменной **cond**.

Функция **pthread_cond_signal()** разблокирует по крайней мере один из потоков, которые заблокированы по указанной условной переменной **cond** (если есть какие-либо потоки, которые заблокированы по **cond**).

Если на условной переменной заблокирован более чем один поток, порядок, в котором потоки разблокируются, определяет политика планирования.

Когда каждый поток, разблокированный в результате **pthread_cond_broadcast()** или **pthread_cond_signal()**, возвращается из своего вызова, он снова будет владеть мьютексом, с которым он вызывал **pthread_cond_wait()** или **pthread_cond_timedwait()**.

При этом разблокированные потоки «борются» за мьютекс в соответствии с политикой планирования, как если бы каждый из них вызвал **pthread_mutex_lock()**.

Функции **pthread_cond_broadcast()** или **pthread_cond_signal()** могут быть вызваны потоком независимо от того, владеет ли он в настоящее время мьютексом, который потоки, вызвавшие **wait()**, связали с условной переменной на время своего ожидания. Однако, если требуется предсказуемое поведение планирования, то этот мьютекс должен быть заблокирован потоком, который вызывает **pthread_cond_broadcast()** или **pthread_cond_signal()**.

Функции **pthread_cond_broadcast()** и **pthread_cond_signal()** не будут иметь никакого эффекта, если нет потоков, заблокированных в данный момент на **cond**.

Если значение, указанное аргументом **cond** для **pthread_cond_broadcast()** или **pthread_cond_signal()**, не относится к инициализированной переменной условия, поведение не определено.

Если значение, указанное аргументом **cond** для **pthread_cond_broadcast()** или **pthread_cond_signal()**, не относится к инициализированной условной переменной, поведение не определено.

Возвращаемое значение

В случае успеха функции **pthread_cond_broadcast()** и **pthread_cond_signal()** возвращают ноль, в противном случае номер ошибки

Ошибки

Функция **pthread_cond_broadcast()** может завершиться с ошибкой:

EINVAL — **cond** не относится к инициализированной условной переменной.

Применение

Функцию **`pthread_cond_broadcast()`** используют всякий раз, когда состояние совместно используемой переменной (предикат) было изменено таким образом, что более одного потока могут продолжить выполнение своей задачи.

Рассмотрим задачу с одним производителем и несколькими потребителями, когда производитель может вставить несколько элементов в список, к которому потребители получают доступ по одному элементу за раз.

Вызывая функцию **`pthread_cond_broadcast()`**, производитель уведомит всех потребителей, которые могут ожидать, и, таким образом, приложение будет более производительно на мультипроцессорной/мультиядерной платформе.

Кроме того, **`pthread_cond_broadcast()`** упрощает реализацию блокировки чтения-записи (rw-lock). Когда пишущий снимает блокировку, требуется разбудить всех ожидающих чтения, соответственно, функция **`pthread_cond_broadcast()`** сделает это наиболее оптимальным образом.

Также, алгоритм двухфазной фиксации (commit) может использовать эту широковещательную функцию для уведомления всех клиентов о предстоящей фиксации транзакции.

Использовать функцию **`pthread_cond_signal()`** в обработчике сигнала, который вызывается асинхронно, небезопасно.

Поэтому мьютексы и условные переменные не подходят для разблокировки ожидающего потока путем передачи сигналов от кода, выполняемого в обработчике сигналов.

Примеры

Условная переменная может быть уничтожена сразу после пробуждения всех заблокированных на ней потоков. Например, рассмотрим следующий код:

```
struct list {
    pthread_mutex_t mutex;
    ...
}

struct item {
    key k;
    int busy;                // занято
    pthread_cond_t notbusy; // свободно?
    ...
}

// Найти элемент списка и пометить его занятым.
struct item *find_item(struct list *list, key k)
{
    struct item *item = NULL;

    pthread_mutex_lock(&list->mutex);
    while ((item = find_item(l, k) != NULL) && item->busy)
        pthread_cond_wait(&item->notbusy, &list->mutex);
    if ( item != NULL)
        item->busy = 1;
    pthread_mutex_unlock(&list->mutex);
    return(item);
}
```

```
delete_item(struct list *list, struct item *item) {

    pthread_mutex_lock(&list->mutex);
    assert(item->busy);

    ... remove item from list ...

    ep->busy = 0;  /* Paranoid. */
(A) pthread_cond_broadcast(&item->notbusy);
    pthread_mutex_unlock(&list->mutex);
(B) pthread_cond_destroy(&item->notbusy);
    free(item);
}
```

В этом примере условная переменная и ее элемент списка могут быть освобождены (строка B) сразу после того, как все потоки, ожидающие ее, будут пробуждены (строка A), поскольку мьютекс и код гарантируют, что никакой другой поток не может коснуться удаляемого элемента.

pthread_condattr_init()/destroy() — инициализировать/уничтожить объект атрибутов

```
 #include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t    *attr);
```

Функция **pthread_condattr_init()** инициализирует объект атрибутов условной переменной **attr** значением по умолчанию для всех атрибутов, определенных реализацией.

Стандарт требуют два атрибута — атрибут часов и атрибут общего доступа со стороны процессов. Однако, реализации могут дополнять атрибуты сверх этого.

Объект атрибутов условных переменных, совместно используемый процессами, был определен для по той же причине, по которой он был определен для мьютексов.

Если вызывается **pthread_condattr_init()** с указанием уже инициализированного объекта атрибутов **attr**, результат не определен.

Функция **pthread_condattr_destroy()** уничтожает объект атрибутов условной переменной.

После этого объект фактически становится неинициализированным.

Реализация **pthread_condattr_destroy()** может установить для объекта, на который ссылается **attr**, недопустимое значение.

Уничтоженный объект атрибутов **attr** можно повторно инициализировать с помощью **pthread_condattr_init()**. Результаты ссылок на объект после его уничтожения не определен.

Возвращаемое значение

В случае успеха функции **`pthread_condattr_destroy()`** и **`pthread_condattr_init()`** возвращают ноль, в противном случае номер ошибки.

Ошибки

Функция **`pthread_condattr_init()`** завершится ошибкой:

ENOMEM — Недостаточно памяти для инициализации объекта атрибутов переменной условия.

Функция **`pthread_condattr_destroy()`** может завершится ошибкой:

EINVAL — значение, указанное аргументом **`attr`** для **pthread_condattr_destroy()**, не относится к инициализированному объекту атрибутов условной переменной.

pthread_condattr_{get|set}pshared() — получить/установить атрибут общего доступа

```
#include <pthread.h>

int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                int                      *restrict pshared);

int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
```

Функция **pthread_condattr_getpshared()** получает значение атрибута общего доступа со стороны процессов из объекта атрибутов, на который ссылается **attr**.

Функция **pthread_condattr_setpshared()** устанавливает атрибут общего доступа со стороны процессов в инициализированном объекте атрибутов, на который ссылается **attr**.

PTHREAD_PROCESS_SHARED

Чтобы разрешить обработку условной переменной любому потоку, имеющему доступ к памяти, в которой выделена переменная условия, даже если переменная условия выделена в памяти, совместно используемой несколькими процессами, атрибут общего доступа к процессу устанавливается в **PTHREAD_PROCESS_SHARED**.

PTHREAD_PROCESS_PRIVATE

Доступ со стороны других процессов запрещен. Значение по умолчанию.

Если значение, указанное аргументом **attr** для **pthread_condattr_getpshared()** или **pthread_condattr_setpshared()**, не относится к инициализированному объекту атрибутов условной переменной условия, поведение не определено.

Возвращаемое значение

В случае успеха функция **`pthread_condattr_setpshared()`** вернет ноль, в противном случае должен возвращается номер ошибки.

В случае успеха функция **`pthread_condattr_getpshared()`** вернет ноль и сохранит значение атрибута из объекта **`attr`** в объекте, на который ссылается параметр **`pshared`**. В противном случае будет возвращен номер ошибки.

Ошибки

Функция **`pthread_condattr_setpshared()`** может завершиться ошибкой, если:

EINVAL — Новое значение, указанное для атрибута, выходит за пределы диапазона допустимых значений для этого атрибута.

clock_getres(), clock_gettime(), clock_settime() — функции часов и таймера

```
#include <time.h>

int clock_getres(clockid_t  clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp);
```

Функция **clock_getres()** возвращает разрешение любых часов.

Разрешение часов определяется реализацией и не может быть установлено процессом.

Если аргумент **res** не равен **NULL**, разрешение указанных часов будет сохранено в местоположении, на которое указывает **res**.

Если **res** равен **NULL**, разрешение часов не возвращается.

Функция **clock_gettime()** возвращает текущее значение **tp** для часов, указанных **clock_id**.

Функция **clock_settime()** устанавливает для часов, указанных **clock_id**, значение, указанное в **tp**.

Значения времени, которые находятся между двумя последовательными неотрицательными целыми кратными разрешения указанных часов, усекаются до меньшего кратного разрешения.

Часы могут быть общесистемными (то есть видимыми для всех процессов) или отдельными для процессов (измерение времени, которое имеет смысл только внутри процесса).

Все реализации поддерживают часы **CLOCK_REALTIME** (определено в **<time.h>**). Эти часы представляют собой часы, измеряющие реальное время для системы. Для данных часов значения, возвращаемые функцией **clock_gettime()** и определяемые функцией **clock_settime()**, представляют собой количество времени (в секундах и наносекундах) с эпохи UNIX.

Реализация также может поддерживать дополнительные часы. Интерпретация значений времени для таких часов стандартом не определена.

Если с помощью **clock_settime()** установлено значение часов **CLOCK_REALTIME**, все сервисы абсолютного времени, которые основаны на **CLOCK_REALTIME**, будет использовать это новое значение часов для определения времени истечения (expiration time).

Это относится и ко времени истечения активированных ***абсолютных*** таймеров.

Если абсолютное время, запрошенное при вызове такого сервиса времени, предшествует новому значению часов, сервис времени завершается немедленно, как если бы часы достигли запрошенного времени в обычном режиме.

Установка значения часов **CLOCK_REALTIME** с помощью **clock_settime()** не влияет на потоки, которые заблокированы в ожидании истечения сервиса ***относительного*** времени, основанного на этих часах, включая функцию **nanosleep()**. Также не влияет на потоки по истечении ***относительных*** таймеров, основанных на этих часах. Следовательно, эти сервисы времени сработают, когда истечет запрошенный относительный интервал, независимо от нового или старого значения часов.

Если поддерживается опция **Monotonic Clock**, реализация поддерживает **CLOCK_MONOTONIC**, определенный в **<time.h>**. Эти часы представляют собой монотонные часы для системы. Для этих часов значение, возвращаемое **clock_gettime()**, представляет количество времени (в секундах и наносекундах) с ***неустанавливаемого точно*** момента времени в прошлом (например, время запуска системы или эпоха). Эта точка не меняется после запуска системы.

Значение часов **CLOCK_MONOTONIC** не может быть установлено через **clock_settime()**.

Эта функция завершится неудачно, если она вызывается с аргументом **clock_id** равным **CLOCK_MONOTONIC**.

Влияние установки часов с помощью **clock_settime()** на активированные таймеры для каждого процесса, связанные с часами, отличными от **CLOCK_REALTIME**, определяется реализацией.

Если значение часов **CLOCK_REALTIME** установлено с помощью **clock_settime()**, для определения времени, в которое система должна пробудить поток, заблокированный абсолютным вызовом **clock_nanosleep()** на основе часов **CLOCK_REALTIME**, будет использоваться новое значение часов.

Если абсолютное время, запрошенное при вызове такой службы времени, окажется раньше нового значения часов, вызов немедленно вернется, как если бы часы достигли запрошенного времени в обычном режиме.

Установка значения часов **CLOCK_REALTIME** с помощью **clock_settime()** не влияет ни на какой из потоков, заблокированных при относительном вызове **clock_nanosleep()**. Следовательно, вызов вернется, как только запрошенный относительный интервал истечет, независимо от нового или старого значения часов.

Если определен макрос **_POSIX_CPUTIME**, реализация поддерживает значения идентификатора часов, полученные путем вызова **clock_getcpuclockid()**, которые представляют часы процессорного времени данного процесса (CPU-time).

POSIX требует поддержки специального значения **clockid_t CLOCK_PROCESS_CPUTIME_ID**, которое представляет часы процессорного времени вызывающего процесса при вызове одной из функций **clock_*()** или **timer_*()**. Для этих идентификаторов часов значения, возвращаемые функцией **clock_gettime()** и определяемые функцией **clock_settime()**, представляют количество времени, в течение которого выполнялся процесс, связанный с часами.

Изменение значения часов процессорного времени с помощью **clock_settime()** не влияет на поведение спорадической политики планирования.

Если определен макрос **_POSIX_THREAD_CPUTIME**, реализация поддерживает значения идентификатора часов, полученные путем вызова **pthread_getcpuclockid()**, которые представляют часы процессорного времени для данного потока.

Реализация также поддерживает специальное значение идентификатора часов, равное **CLOCK_THREAD_CPUTIME_ID**, которое представляет часы процессорного времени для вызывающего потока, котороые используются при вызове одной из функций **clock_*()** или **timer_*()**.

Для этих идентификаторов часов значения, возвращаемые функцией **clock_gettime()** и определяемые функцией **clock_settime()**, должны представлять количество времени выполнения потока, связанного с этими часами. Изменение значения часов процессорного времени с помощью **clock_settime()** не влияет на поведение спорадической политики планирования.

Возвращаемое значение

Возвращаемое значение 0 указывать на то, что вызов выполнен успешно.

Возвращаемое значение -1 указывать на то, что произошла ошибка, при этом значение **errno** будет указывать на ошибку.

Ошибки

Функции **clock_getres()**, **clock_gettime()** и **clock_settime()** завершатся ошибкой:

EINVAL – аргумент **clock_id** не указывает известные часы.

Функция **clock_gettime()** завершится с ошибкой:

EOVERFLOW – количество секунд не помещается в объекте типа **time_t**.

Функция **clock_settime()** завершится с ошибкой:

EINVAL – аргумент **tp** для **clock_settime()** выходит за пределы диапазона для данного идентификатора часов.

EINVAL – аргумент **tp** указывает значение в наносекундах меньше нуля или больше или равно 1000 миллионов.

EINVAL – значение аргумента **clock_id** равно **CLOCK_MONOTONIC**.

Функция **clock_settime()** может завершиться ошибкой:

EPERM – запрашивающий процесс не имеет соответствующих привилегий для установки указанных часов.

pthread_condattr_{get|set}clock() — получить/установить атрибут часов для условн. переменной

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                              clockid_t                *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t           clock_id);
```

Функция **pthread_condattr_getclock()** получает значение атрибута идентификатора часов из объекта атрибутов, на который ссылается **attr**.

Функция **pthread_condattr_setclock()** устанавливает атрибут часов в инициализированном объекте атрибутов, на который ссылается **attr**.

Если **pthread_condattr_setclock()** вызывается с аргументом **clock_id**, который относится к часам ЦП, вызов завершится ошибкой.

Атрибут часов — это идентификатор часов, который должен использоваться для измерения времени ожидания службы **pthread_cond_timedwait()**. Значение по умолчанию атрибута часов относится к системным часам.

Если значение, указанное аргументом **attr** для **pthread_condattr_getclock()** или **pthread_condattr_setclock()**, не относится к инициализированному объекту атрибутов переменной условия, поведение не определено.

Возвращаемое значение

В случае успеха функция **pthread_condattr_getclock()** возвращает ноль и сохраняет значение атрибута часов для **attr** в объекте, на который ссылается аргумент **clock_id**. В противном случае возвращается номер ошибки.

Ошибки

Функция **pthread_condattr_setclock()** может завершиться ошибкой, если:

EINVAL — Значение, указанное в **clock_id**, не относится к известным часам или является часами ЦП.