Понятие об условных переменных

Организация взаимоисключений позволяет организовать корректную работу набора процессов, имеющего race condition, если порядок доступа к общим ресурсам нам не важен (см. лекции, тема 5). Если же нам важен порядок доступа к общим ресурсам, то одними взаимоисключениями обойтись не удастся. Нам нужно уметь осуществлять взаимную синхронизацию работы активностей. В мониторах Хора (см. лекции, тема 6) взаимоисключение для функций-методов осуществлялось автоматически, а для взаимной синхронизации вводились условные переменные для дополнительной возможности блокировать и разблокировать процессы. Вот и для набора нитей исполнения вводятся условные переменные, подобные условным переменным в мониторах Хора.

Условные переменные для нитей исполнения —это объекты, для работы с которыми одного описания объекта специального типа недостаточно. Эти объекты требуют выделения определенных ресурсов процесса и операционной системы, поэтому после описания перед использованием их необходимо инициализировать, а по окончании использования деинициализировать (разрушить) для освобождения ресурсов. После инициализации над условными переменными в нитях исполнения можно выполнять те же самые операции, что и над условными переменными в мониторах Хора, — wait и signal.

При выполнении операции wait над некоторой условной переменной нить, выполнившая операцию, безусловно блокируется (переводится в состояние *ожидание*). Блокировка продолжается до тех пор, пока другая нить процесса не выполнит операцию signal над той же самой условной переменной. Если существует одна или несколько нитей, ожидающих операции signal над одной и той же переменной, то при выполнении над ней операции signal разблокируется **ровно одна** ожидающая нить. Если ожидающих разблокировки нитей нет, то операция signal —это просто пустая операция.

Инициализация условных переменных

Для использования условной переменной в программе вам потребуется описать ее с помощью специального типа данных `pthread_cond_t`, определенного в файле `<pthread.h>`, например

```
pthread_cond_t cond;
```

Описанная таким образом условная переменная является неинициализированной. Для инициализации условной переменной существует два способа: использование статического инициализатора или использование специальной функции.

Внимание: повторная инициализация уже инициализированной условной переменной может привести к непредсказуемым результатам, это не регламентируется в стандарте POSIX и не обладает переносимостью!

Для применения статического инициализатора условной переменной по стандарту POSIX должна быть описана как статический объект, т.е. ее описание не должно находиться в стеке. Мы с вами будем использовать стандартный статический инициализатор `PTHREAD_COND_INITIALIZER`. Пример:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Другой способ инициализации сводится к использованию функции `pthread_cond_init()`. Пример:

```
pthread_cond_t cond;
int err;
err = pthread_cond_init(&cond, NULL);
```

В качестве второго параметра этой функции в нашем курсе мы всегда будем использовать параметр `NULL` — инициализацию по умолчанию для стандартной условной переменной.

После инициализации условная переменная готова к выполнению над ней операций wait и signal.

Реализация операций wait и signal

При работе с мониторами Хора решение о применении операций wait и signal принимается обычно после анализа значений некоторых внутренних переменных (не условных!) монитора. Во время анализа изменение значений таких переменных другими процессами невозможно из-за автоматической организации взаимоисключений при работе функций-методов монитора. При работе условных переменных для нитей исполнения такого автоматического взаимоисключения нет.

Поэтому в нитях исполнения перед анализом некоторых общих данных для принятия решения о применении операций wait и signal необходимо эти данные на время анализа защитить захватом какого-то мьютекса. Однако, если принято решение отправить процесс в ожидание с помощью выполнения операции wait над условной переменной, этот мьютекс нужно одновременно атомарно с введением процесса в спячку освободить!

Следовательно, выполнение операции wait должно быть связано не только с условной переменной, но и с мьютексом, который защищал данные во время анализа.

Для выполнения операции wait над условной переменной используется функция `pthread_cond_wait()`, имеющая два параметра: адрес условной переменной, над которой нужно совершить операцию wait, и адрес мьютекса, который должен быть освобожден при выполнении операции wait.

По стандарту POSIX после выполнения операции signal над некоторой условной переменной, пробужденная нить должна атомарно с пробуждением захватить мьютекс, указанный в качестве параметра в вызове `pthread_cond_wait()` и который ранее был освобожден. Соответственно, программный текст, соответствующий выполнению операции wait, мог бы схематично выглядеть следующим образом:

```
pthread_mutex_t m   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int N;
Нить исполнения
    pthread_mutex_lock(&m);
    if(N == 0) pthread_cond_wait(&cond, &m);
    …
    pthread_mutex_unlock(&m);
```

Поскольку не во всех реализациях POSIX нитей соблюдаются рекомендации стандарта насчет атомарности при пробуждении (а стало быть кто-то может успеть переопределить значение N до захвата мьютекса пробужденной нитью), то для надежности работы и переносимости программы вместо схемы, приведенной выше, правильнее использовать другую схему:

```
pthread_mutex_t m   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int N;
Нить исполнения
    pthread_mutex_lock(&m);
    while(N == 0) pthread_cond_wait(&cond, &m);
    …
    pthread_mutex_unlock(&m);
```

Выполнение операции signal над условной переменной не требует немедленного освобождения мьютекса, обеспечивающего эксклюзивный доступ к данным, на основании которых было принято решение о выполнении данной операции. Поэтому у функции pthread_cond_signal(), которая обеспечивает выполнение операции signal, всего лишь один параметр — адрес условной переменной. Обычно принято освобождать мьютекс сразу после выполнения операции signal:

```
pthread_mutex_t m   = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int N;

// Нить исполнения
pthread_mutex_lock(&m);
if(N != 0) pthread_cond_signal(&cond);
pthread_mutex_unlock(&m);
```

Деинициализация условной переменной

Если программа завершила работать с условной переменной (или ее не предполагается использовать длительное время), то для освобождения ресурсов процесса и операционной системы ее необходимо деинициализировать (разрушить). Это выполняется с помощью функции `pthread_cond_destroy()`. После вызова этой функции условная переменная становится неинициализированной и при необходимости ее можно повторно инициализировать.

Важно! Нельзя деинициализировать условную переменную, у которой есть список нитей для пробуждения. В этом случае функция `pthread_cond_destroy()` может ждать, пока список не станет пустым, и возможно возникновение тупиковой ситуации.

Поведение функции `pthread_cond_destroy()` для разрушения неинициализированной условной переменной по стандарту POSIX не определено и зависит от реализации.

Решение задачи producer-consumer с объемом буфера в одно сообщение

Предположим, нам необходимо решить следующую задачу: у нас есть процесс producer, производящий порции информации и процесс consumer, ее считывающий. В буфер помещается ровно N порций информации.

В семантике мониторов мы бы имели решение, напоминающее это (синтаксис очень условный):

```
monitor channel {
        queue messages
        int count = 0
        condition rcv
        condition snd

        function send(string msg) {
                while (count == N) do wait(rcv)
                messages.push(msg)
                count = count + 1
                if (count == 1) do signal(snd)
        }

        function receive() {
                while (count == 0)  do wait(snd)
                count = count - 1
                string msg = messages.pop()
                if (count == N-1) do signal(rcv)
                return msg
        }
}
```

Рассмотрим пример 14-1.c, в котором эта же логика реализуется через условные переменные из библиотеки pthread для случая N=1. Обратите внимание на то, что мы не используем монитор, мьютексы вокруг аналогов упомянутых выше операций приходится выставлять вручную.

Значение N можно изменить и убедиться, что общая схема синхронизации работает и для большего числа сообщений. Хотя стоит отметить, что при этом consumer всегда будет получать только последнее сообщение из-за тривиального устройства буфера, вмещающего всего одно сообщение. Для корректного решения в случае N>1 можно сделать кольцевой буфер длины N.

Задачи на семинар

Задача 1 (20 баллов):

*Опираясь на решение домашних заданий, решите задачу о катере и пассажирах через нити исполнения, мьютексы и условные переменные. Катер будет основной нитью, которая предварительно генерирует N*K нитей-пассажиров и совершает K поездок. Числа N и K либо вводятся с клавиатуры, либо определяются макросом, как в примере 14-1.c. Для тестирования считать, что N=5, K=3, но программа должна работать и на других значениях.*