Лабораторная работа №8
«Потоки. Синхронизация ||»

На данном занятии следует познакомится с двумя новыми средствами, предназначенными для синхронизации потоков: обычными семафорами и условными переменными.

Основные задания

Задание №1 (Семафоры)

Существует поток-производитель и поток-потребитель данных. Производитель генерирует целое псевдослучайное число из заданного диапазона. Потребитель забирает это число. С помощью семафоров синхронизируйте их работу так, чтобы потребитель не мог пытаться получить еще не созданные числа, а производитель не мог сделать больше чисел, чем может получить потребитель. Через заданный промежуток времени потоки отменяются главным потоком и программа завершает свою работу.

Для синхронизации потоков разместим между потоком-производителем и потоком-потребителем буфер заданного при старте программы размера. Производитель может помещать сгенерированные числа в буфер, потребитель может забирать числа из буфера. Если потребитель забирает число, то его исключают из буфера. Необходимо обеспечить несколько требований:

1. когда потребитель или производитель работает с буфером остальные потоки должны ждать, пока он завершит свою работу;
2. когда производитель пытается поместить объект в буфер, а буфер полный, он должен дождаться, пока в буфере появится место;
3. когда потребитель пытается забрать объект из буфера, а буфер пустой, он должен дождаться, пока в нем появится объект.

Дополнительно: решите задачу для случая, когда работают несколько потоков-потребителей и несколько потоков-производителей (особое внимание уделите завершению приложения).

Задание №2 (Условные переменные)

Существует поток-производитель и поток-потребитель данных. Поток-производитель с достаточно большим периодом генерирует псевдослучайное число, присваивает его глобальной переменной и оповещает поток-потребитель. Поток-потребитель получает оповещение, забирает данные (исключает их из переменной) и сразу выводит их в стандартный поток вывода. Пока данных нет, поток-потребитель ожидает уведомления. Синхронизируйте работу потоков с помощью условных переменных. Через заданный промежуток времени потоки отменяются главным потоком и программа завершает свою работу.

Дополнительно: решите задачу для случая, когда дан буфер заданного (необязательно единичного) размера и работают несколько потоков-потребителей и несколько потоков-производителей (особое внимание уделите завершению приложения).

Рекомендации по выполнению

Семафоры. Синхронизация процессов

В теории операционных систем семафор представляет собой неотрицательную целую переменную, над которой возможны два вида операций: P и W.

- W-*операция* (операция ожидания — *wait*) над семафором представляет собой попытку уменьшения значения семафора на 1. Если перед выполнением W-*операции* значение семафора было больше 0, W-*операция* выполняется без задержек. Если перед выполнением W-*операции* значение семафора было 0, поток, выполняющий W-*операцию*, переводится в состояние ожидания до тех пор, пока значение семафора не станет большим 0 (вследствие действий, выполняемых другими потоками) После снятия блокировки значение семафора уменьшается на единицу и операция завершается.
- P-*операция* (операция установки — *post*) над семафором представляет собой увеличение значения семафора на 1. Если до этого счетчик был равен нулю и существовали потоки, заблокированные в операции ожидания данного семафора, один из этих процессов выходит из состояния ожидания и может выполнить свою W-*операцию* (т.е. счетчик семафора снова становится равным нулю).

Замечание: Эти названия операций введены нами для удобства. В специальной литературе эти операции называются, соответственно, *P*-операцией (*P* – от голландского *Proberen* – проверить) и *V*-операцией (*V* – от голландского *Verhogen* – увеличить).

Семафоры являются гибким и удобным средством для синхронизации и взаимного исключения потоков и учета ресурсов. Операционная система гарантирует, что проверка и модификация значения семафора могут быть выполнены безопасно и не приведут к возникновению гонки. *Э́дсгер Ви́бе Де́йкстра* (*1930 – 2002*), нидерландский ученый, специалист по компьютерным наукам первым сформулировал идею семафоров.

Семафор можно использовать в качестве еще одного средства синхронизации процессов, при этом операция W аналогична захвату мьютекса, а операция P — операции его освобождения. Однако, в отличие от мьютексов, операции W и P не обязаны выполняться одним и тем же потоком исполнения и даже не обязаны быть парными. Это позволяет использовать семафоры в различных ситуациях, которые сложно или невозможно разрешить при помощи мьютексов. Иногда семафоры используются в качестве разделяемых целочисленных переменных, например, в качестве счетчиков записей в очереди.

В *Linux* существуют две отличающиеся реализации семафоров. Мы рассмотрим реализацию, соответствующую стандарту *POSIX*. Именно такие семафоры применяются для организации взаимодействия потоков. Другую реализацию, больше предназначенную для организации межпроцессного взаимодействия рассмотрим чуть позже.

При работе с семафорами необходимо включить в программу заголовочный файл <semaphore.h>. Переменная — семафор относится к переменной типа sem_t. Следует отметить, что имена типов и функций семафоров начинаются не с префикса pthread_, как большинство функций, относящихся к потокам, а с sem_.

Семафор следует предварительно инициализировать с помощью функции sem_init():

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Эта функция инициализирует объект-семафор, на который указывает параметр sem, задает вариант его совместного использования (параметр pshared) и присваивает ему начальное целочисленное значение (параметр value). Параметр pshared управляет типом семафора. Если pshared равен 0, семафор локален по отношению к текущему процессу. В противном случае семафор может быть совместно использован разными процессами. Мы будем работать с семафорами, которые **не используются** совместно разными процессами. Пока *ОС Linux* не поддерживала такое совместное использование и передача ненулевого значения параметру pshared приводила к аварийному завершению вызова.

Следующие две функции управляют значением семафора и объявляются так:

```
#include <semaphore.h>
int sem_wait(sem_t * sem);
int sem_post (sem_t * sem);
```

Обе эти функции принимают указатель на объект-семафор, инициализированный вызовом sem_init. Функция sem_post атомарно увеличивает значение семафора на 1. Атомарно в данном случае означает, что если два потока одновременно пытаются увеличить значение единственного семафора на 1, они не мешают друг другу. Т. е., если обе программы пытаются увеличить значение на 1, семафор всегда будет корректно увеличивать значение на 2.

Функция sem_wait атомарно уменьшает значение семафора на единицу, но всегда ждет до тех пор, пока сначала счетчик семафора не получит ненулевое значение. Таким образом, если вы вызываете sem_wait для семафора со значением 2, поток продолжит выполнение, а семафор будет уменьшен до 1. Если sem_wait вызывается для семафора со значением 0, функция будет ждать до тех пор, пока какой-нибудь другой поток не увеличит значение, и оно станет ненулевым. Если оба потока ждут в функции sem_wait, чтобы один и тот же семафор стал ненулевым, и он увеличивается когда-нибудь третьим потоком, только один из двух ждущих потоков получит возможность уменьшить семафор и продолжиться; другой поток так и останется ждущим. Именно эта атомарная способность «проверить и установить» в одной функции делает семафор исключительно ценным.

Существует еще и функция семафора sem_trywait — реализующая операцию неблокирующего ожидания. Она напоминает функцию pthread_mutex_trylock(): если операция ожидания приведет к блокированию потока из-за того, что счетчик семафора равен нулю, функция немедленно завершается, возвращая код ошибки EAGAIN.

В *Linux* существует еще и функция sem_getvalue(sem_t *sem, int *sval), позволяющая узнать текущее значение счетчика семафора sem. Это значение помещается в переменную типа int на которую ссылается второй аргумент функции. Однако, на основании данного значения не следует определять, стоит ли выполнять операцию ожидания или установки. Это может привести к возникновению гонки: другой поток способен изменить счетчик семафора между вызовами функции sem_getvalue() и какой-нибудь другой функции для работы с семафором. Для этого следует использовать только атомарные функции sem_wait() и sem_post().

Еще одна нужная нам функция семафоров — sem_destroy. Она очищает семафор, когда вы закончили работу с ним, и объявляется следующим образом:

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

Эта функция принимает указатель на семафор и очищает любые ресурсы, которые у

него могли быть. Очищаемый семафор должен быть «свободен»: если выполнить попытку уничтожить семафор, которого дожидается какой-либо поток, то в результате будет получена ошибка. Как и большинство других функций, все перечисленные функции возвращают 0 в случае успешного завершения и код ошибки в противном случае.

Такой классический семафор можно использовать как счетчик ресурсов. Если у нас имеется N единиц некоторого ресурса, то для контроля его распределения создается общий семафор S с начальным значением N. Выделение ресурса сопровождается операцией W(S), освобождение — операцией P(S). Значение семафора, таким образом, отражает число свободных единиц ресурса. Если значение семафора 0, то есть, свободных единиц больше не остается, то очередной процесс, запрашивающий единицу ресурса будет переведен в ожидание в операции W(S) до тех пор, пока какой-либо из использующих ресурс процессов не освободит единицу ресурса, выполнив при этом P(S).

Рассмотрим пример применение семафоров.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
	int res;
	pthread_t a_thread;
	void *thread_result;

	res = sem_init(&bin_sem, 0, 0);
	if (res != 0) {
		perror("Semaphore initialization failed");
		exit(EXIT_FAILURE);
	}
	res = pthread_create(&a_thread, NULL, thread_function, NULL);
	if (res != 0) {
		perror("Thread creation failed");
		exit(EXIT_FAILURE);
	}
	printf("Input some text. Enter 'end' to finish\n");
	while (strncmp("end", work_area, 3) != 0) {
		fgets(work_area, WORK_SIZE, stdin);
		sem_post(&bin_sem);
	}
	printf("\nWaiting for thread to finish...\n");
	res = pthread_join(a_thread, &thread_result);
	if (res != 0) {
		perror("Thread join failed");
```

```
            exit(EXIT_FAILURE);
        }
        printf("Thread joined\n");
        sem_destroy(&bin_sem);
        exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
        sem_wait(&bin_sem);
        while(strncmp("end", work_area, 3) != 0) {
            printf("You input %d characters\n", strlen(work_area)-1);
            sem_wait(&bin_sem);
        }
        pthread_exit(NULL);
}
```

Сначала объявляется переменная-семафор, некоторое количество переменных и инициализируется семафор перед созданием нового потока. Начальное значение семафора равно 0.

В функции main, после того как запущен новый поток, считается некоторый текст с клавиатуры, загружается в массив work_area и затем наращивается счетчик семафора с помощью sem_post. В новом потоке процесс ждет семафор и затем подсчитывает символы ввода.

Пока семафор установлен, программа ждет ввода с клавиатуры. Когда что-то будет введено, то освобождается семафор, разрешив второму потоку посчитать символы перед тем, как первый поток начнет снова считывать ввод с клавиатуры.

И опять потоки совместно используют один и тот же массив work_area. Для того чтобы программный код был короче, в программе пропущены некоторые проверки ошибок, например значения, возвращаемые из функции sem_wait. *В рабочем программном коде всегда следует всегда проверять возвращаемые значения.*

Когда инициализируется семафор, ему присваивается начальное значение, равное 0. Следовательно, когда запускается функция потока, вызов sem_wait приостанавливает выполнение и ждет, когда семафор станет ненулевым. В потоке main программа ждет до тех пор, пока не будет некоторого текста, и затем увеличиваете счетчик семафора с помощью функции sem_post, которая немедленно разрешает другому потоку вернуться из своей функции sem_wait и начать выполнение. После того как он сосчитает символы, поток вновь вызывает sem_wait и приостанавливает выполнение до тех пор, пока поток main не вызовет снова sem_post для того, чтобы увеличить семафор.

Проблема этой программы заключается в том, что программа рассчитывает на то, что ввод текста из программы продлится так долго, что у другого потока хватит времени для подсчета символов до того, как поток main подготовится передать ему новую порцию текста для подсчета. У второго потока, при вводе коротких сообщений, не будет хватать времени для выполнения. Но семафор наращивается несколько раз, поэтому считающий поток будет продолжать считать слова и уменьшать значение семафора до тех пор, пока оно снова не стало нулевым. Исправить программу можно, применяя дополнительный семафор для того, чтобы заставить поток main ждать, пока у считающего потока не появится возможность закончить свой подсчет, но гораздо легче применить рассмотренный на прошлом занятии мьютекс.

Реализация задачи производителя-потребителя при помощи семафора

Выделим синхронизированные действия, которые потребуются для решения этой задачи:

1. разместить операции непосредственного изменения буфера (для производителя — помещение числа в буфер, для потребителя — получение числа из буфера) в критические секции;
2. организовать ожидание в соответствии с требованием ожидания производителя в случае полного буфера. При этом потребитель должен сообщать производителям, которые находятся в состоянии ожидания, про то, что он забрал объект из буфера (т. е. буфер стал неполным);
3. организовать ожидание в соответствии с требованием ожидания потребителя в случае пустого буфера. При этом производитель должен сообщать потребителям, которые находятся в состоянии ожидания, про то, что он поместил объект в буфер (т. е. буфер стал не пустым).

Теперь рассмотрим примитивы синхронизации, которые потребуются для решения задачи:

1. для организации критической секции можно использовать бинарный семафор (lock). Этот семафор будет использоваться как производителем, так и потребителем, защищая доступ к буферу от любых других потоков;
1. для организации ожидания производителя в случае полного буфера можно использовать семафор, текущее значение которого совпадает с количеством свободных мест в буфере (empty_items). Произвидитель перед попыткой добавления нового объекта в буфер уменьшает этот семафор, переходя в состояние ожидания, если семафор был равен 0. Потребитель, после того, как заберет объект из буфера, увеличит семафор, таким образом уведомив ожидающих производителей и «разбудив» одного из них.
1. для организации ожидания потребителя в случае пустого буфера можно использовать семафор, текущее значение которого совпадает с количеством занятых мест в буфере (full_items). Потребитель перед попыткой забрать объекта из буфера уменьшает этот семафор, переходя в состояние ожидания, если семафор был равен 0. Производитель, после того, как добавит объект в буфер, увеличит семафор, таким образом уведомив ожидающих потребителей и «разбудив» одного из них.

Приведем псевдокод для этой задачи:

```
semaphore lock = 1;                 // для критической секции
semaphore empty_items = size; // 0 — буфер полный, сначала он пустой
semaphore full_items = 0;           // если 0 — буфер пуст

// производитель
void producer() {
        item = produce();       // создать объект
        W(empty_items);     // есть ли место в буфере?
        W(lock);                // вход в критическую серцию
        append_to_buffer(item);  // добавить объект item в буфер
        P(lock);                        // выход из критической секции
        P(full_items);                // уведомить потребителей, что есть новый объект
}
```

```
// потребитель
void consumer() {
        W(full_items);              // не пустой ли буфер?
        W(lock);                    // вход в критическую серцию
        item = receive_from_buffer(); // забрать объект item из буфера
        P(lock);                    // выход из критической секции
        P(empty_items);             // уведомить производителей, что есть место
        consume(item);              // «потребить» объект
}
```

Условные переменные

Условная переменная представляет собой еще одно средство синхронизации процессов. Над ней определены две основные операции: wait и signal. Поток исполнения, выполнивший операцию wait, блокируется до того момента, пока другой поток исполнения не выполнит операцию signal. Таким образом, операцией wait первый поток исполнения сообщает системе, что она ждет выполнения какого-то условия, а операцией signal второй поток исполнения сообщает первому, что параметры, от которых зависит выполнение условия, возможно, изменились. Таким образом, условная переменная представляет собой семафор, используемый для сигнализации о событии, которое произошло. Сигнала о том, что произошло некоторое событие, может ожидать один или несколько потоков (или процессов) от других потоков (или процессов). Существует различие между условными переменными и рассмотренными на прошлом занятии мьютексными семафорами и блокировками чтения-записи. Основное назначение мьютексного семафора и блокировок чтения-записи — синхронизировать доступ к данным, в то время как условные переменные обычно используются для синхронизации последовательности операций.

Основное применение условных переменных — это реализация сценария «*производитель-потребитель*». Рассмотрим два потока исполнения, один из которых генерирует данные, а другой — обрабатывает их. В простейшем случае производитель помещает каждую следующую порцию данных в разделяемую переменную, а потребитель считывает ее оттуда. При этом могут возникать две проблемы. Во-первых, если производитель работает быстрее потребителя, то он может записать очередную порцию данных до того, как потребитель прочитает предыдущую. При этом предыдущая порция данных будет потеряна. Во-вторых, если же потребитель работает быстрее производителя, то он может обработать одну и ту же порцию данных несколько раз. Такие проблемы очень сложно устранить при помощи мьютексов. (Для этого прийдется применить не менее трех мьютексов и использовать для начальной синхронизации холостой цикл или какой-то другой примитив межпоточного взаимодействия.) С помощью условной переменной поток-производитель сигналом уведомляет потребителя о том, что в переменную помещена порция данных. Поток-потребитель может ожидать до тех пор, пока не получит сигнал, а затем перейдет к обработке этих данных. Таким образом, условные переменные предоставляют потокам своеобразное место встречи. Они обычно используются вместе с мьютексами (защищаются ими в случае нескольких потоков-производителей), позволяя потокам ожидать наступления некоторого события и избегая состояния гонки. Прежде чем изменить значение такой переменной, поток должен захватить мьютекс. Другие потоки не будут замечать изменений переменной, пока они не попытаются захватить этот мьютекс, потому что для оценки переменной состояния необходимо запереть мьютекс.

Условная переменная представлена типом pthread_cond_t. Она обязательно должна быть инициализирована перед использованием. При статическом размещении переменной ей

можно присвоить значение константы PTHREAD_COND_INITIALIZER. Если же условная переменная размещается динамически, нужно инициализировать ее с помощью функции pthread_cond_init. После того, как условная переменная выполнила свою работу, ее нужно указать в качестве аргумента функции pthread_cond_destroy. Данная функция «деактивирует» указанную условную переменную.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t * attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи. С помощью аргумента attr можно указать «объект» — атрибут условной переменной. Если в качестве данного параметра передаеть пустой указатель (NULL), то условная переменная состояния будет инициализирована со значениями атрибутов по умолчанию. Этот второй аргумент (указатель на объект атрибутов условной переменной) игнорируется в *Linux*.

Для блокирования потока исполнения до тех пор, пока не будет получен сигнал об изменении заданной условной переменной используются две функции. Функция pthread_cond_wait ожидает, пока переменная не перейдет в истинное состояние. Если нужно ограничить время ожидания заданным интервалом, используется функция pthread_cond_timedwait.

```
#include <pthread.h>

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

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

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Мьютекс, передаваемый функции pthread_cond_wait, защищает доступ к переменной состояния. В момент вызова функций мьютекс уже должен быть захвачен вызывающим потоком. Вызывающий поток передает его функции в запертом состоянии, а функция атомарно помещает вызывающий поток в список потоков, ожидающих изменения состояния переменной, и отпирает мьютекс. Это исключает вероятность того, что переменная изменит состояние между моментом ее проверки и моментом приостановки потока, благодаря чему поток не пропустит наступление ожидаемого события. Когда функция pthread_cond_wait возвращает управление, мьютекс снова автоматически запирается.

Функция pthread_cond_timedwait работает аналогичным образом, но дополнительно предоставляет возможность ограничить время ожидания. Значение аргумента timeout определяет, как долго поток будет ожидать наступления события. Время тайм-аута задается структурой timespec, в которой время представлено в секундах и долях секунды. Доли секунды исчисляются в наносекундах:

```
struct timespec {
        time_t tv_sec; /* секунды */
        long tv_nsec;  /* наносекунды */
};
```

При этом следует указывать абсолютное время, а не относительное. Например, если нам нужно ограничить время ожидания периодом в *3* минуты, то в эту структуру следует преобразовать не *3* минуты, а *текущее время в минутах + 3 минуты*.

Для этого можно воспользоваться функцией gettimeofday, которая возвращает текущее время в виде структуры timeval, и затем преобразовать полученное значение в структуру timespec. Чтобы получить абсолютное время для аргумента timeout, можно использовать следующую функцию пользователя:

```
#include <time.h> /* проверить!!! */

void maketimeout(struct timespec *tsp, long minutes) {
        struct timeval now;

        /* получить текущее время */
        gettimeofday(&now);
        tsp->tv_sec = now.tv_sec;
        tsp->tv_nsec = now.tv_usec * 1000; /* usee to nsec */

        /* добавить величину тайм-аута */
        tsp->tv_sec += minutes * 60;
}
```

Если указанный промежуток времени истечет до появления ожидаемого события, функция pthread_cont_timedwait запрет мьютекс и вернет код ошибки ETIMEDOUT. Когда функция pthread_cont_wait или pthread_cont_timedwait успешно завершится, поток выполнения должен оценить значение переменной, поскольку к этому моменту другой поток мог изменить его.

Для передачи сообщения о наступлении события существуют две функции. Функция pthread_cond_signal возобновит работу одного потока, ожидающего наступления события, а pthread_cond_broadcast — всех потоков, ожидающих наступления события.

```
#include <pthread.h>

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

Обе функции возвращают 0 в случае успешного завершения, или код ошибки в случае неудачи.

Когда вызывается функция pthread_cond_signal, говорят, что посылается сигнал о наступлении события. Необходимо сделать все возможное, чтобы сигнал о наступлении события посылался только после изменения состояния переменной. Следует отметить, что слово signal в имени функции не имеет никакого отношения к сигналам *Unix* SIGxxx.

Условные переменные используются совместно с мьютексами. При попытке заблокировать мьютекс поток или процесс будет заблокирован до тех пор, пока мьютекс не освободится. После разблокирования поток или процесс получит мьютекс и продолжит свою работу. При использовании условной переменной ее необходимо связать с мьютексом.

```
pthread_mutex_lock(&mutex);
pthread_cond_wait (&event, &mutex) ;
//. . .
pthread_mutex_unlock(&mutex);
```

Итак, некоторая задача делает попытку заблокировать мьютекс. Если мьютекс уже заблокирован, то эта задача блокируется. После разблокирования задача освободит мьютекс mutex и при этом будет ожидать сигнала для условной переменной event. Если мьютекс не заблокирован, задача будет ожидать сигнала неограниченно долго. При ожидании с ограничением по времени задача будет ожидать сигнала в течение заданного интервала времени. Если это время истечет до получения задачей сигнала, функция возвратит код ошибки. Затем задача вновь затребует мьютекс.

Выполняя адресную сигнализацию, задача уведомляет другой поток или процесс о том, что произошло некоторое событие. Если задача ожидает сигнала для заданной условной переменной, эта задача будет разблокирована и получит мьютекс. Если сразу несколько задач ожидают сигнала для заданной условной переменной, то разблокирована будет только одна из них. Остальные задачи будут ожидать в очереди, и их разблокирование будет происходить в соответствии с используемой стратегией планирования. При выполнении операции всеобщей сигнализации уведомление получат все задачи, ожидающие сигнала для заданной условной переменной. При разблокировании нескольких задач они будут состязаться за право владения мьютексом в соответствии с используемой стратегией планирования. В отличие от операции ожидания, задача, выполняющая операцию сигнализации, не предъявляет прав на владение мьютексом, и это следует сделать самостоятельно.

Рассмотрим пример применения условных переменных.

Предположим, требуется написать потоковую функцию, которая входит в бесконечный цикл, выполняя на каждой итерации какие-то действия. Но работа цикла должна контролироваться флагом: действие выполняется только в том случае, когда ои установлен. Рассмотрим возможный фрагмент такой программы.

```
#include <pthread.h>

int thread_flag;

pthread_mutex_t thread_flag_mutex;

void initialize_flag () {
	pthread_mutex_init (&thread_flag_mutex, NULL);
	thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; otherwise spins. */
void* thread_function (void* thread_arg) {
	while (1) {
		int flag_is_set;
		/* Protect the flag with a mutex lock. */
		pthread_mutex_lock (&thread_flag_mutex);
		flag_is_set = thread_flag;
		pthread_mutex_unlock (&thread_flag_mutex);
		if (flag_is_set)
			do_work ();
		/* Else don't do anything. Just loop again. */
	}
	return NULL;
}
```

```
/* Sets the value of the thread flag to FLAG_VALUE. */
void set_thread_flag (int flag_value) {
        /* Protect the flag with a mutex lock. */
        pthread_mutex_lock (&thread_flag_mutex);
        thread_flag = flag_value;
        pthread_mutex_unlock (&thread_flag_mutex);
}
```

На каждой итерации цикла потоковая функция проверяет, установлен ли флаг. Поскольку к флагу обращается сразу несколько потоков, он защищается исключающим семафором. Хотя подобная реализация является корректной, но она неэффективна. Если флаг не установлен, потоковая функция будет впустую тратить ресурсы процессора, занимаясь бесцельными проверками флага, а также захватывая и освобождая семафор. На самом деле необходимо как-то перевести функцию в неактивный режим, пока какой-нибудь другой поток не установит этот флаг.

Условная переменная позволяет организовать такую проверку, при которой поток либо выполняется, либо блокируется. Как и в случае семафора, поток может *ожидать* сигнальную переменную. *Поток А*, находящийся в режиме ожидания, блокируется до тех пор, пока другой *поток Б*, не просигнализирует об изменении состоянии этой переменной. Условная переменная не имеет внутреннего счетчика, что отличает ее от семафора. *Поток А* должен перейти в состояние ожидания до того, как *поток Б* пошлет сигнал. Если сигнал будет послан раньше, он окажется потерянным и *поток А* заблокируется, пока какой нибудь поток не пошлет сигнал еще раз.

Предыдущий фрагмент программы можно сделать более эффективным:

- Функция thread_function() в цикле проверяет флаг. Если он не установлен, поток переходит в режим ожидания сигнальной переменной.
- Функция set_thread_flag() устанавливает флаг и сигнализирует об изменении условной переменной. Если функция thread_function() была заблокирована в ожидании сигнала, она разблокируется и снова проверяет флаг

Но не следует забывать об одной проблеме: возникновении гонки между операцией проверки флага и операцией сигнализирования или ожидания сигнала. Предположим, что функция thread_function() проверяет флаг и обнаруживает, что он не установлен. В этот момент планировщик *Linux* прерывает выполнение данного потока и активизирует главную программу. По стечению обстоятельств программа как раз находится в функции set_thread_flag(). Она устанавливает флаг и сигнализирует об изменении условной переменной. Но поскольку в данный момент нет потока, ожидающего получения этого сигнала (функция thread_function() была прервана перед тем, как перейти в режим ожидания), сигнал окажется потерян. Когда *Linux* вновь активизирует дочерний поток, он начнет ждать сигнал, который, возможно, никогда больше не придет.

Чтобы избежать этой проблемы, необходимо одновременно захватить и флаг, и сигнальную переменную с помощью исключающего семафора. Любая условная переменная должна использоваться совместно с исключающим семафором для предотвращения состояния гонки. Таким образом, потоковая функция должна следовать такому алгоритму:

1. В цикле необходимо захватить исключающий семафор и прочитать значение флага.
2. Если флаг установлен, нужно разблокировать семафор и выполнить требуемые действия.
3. Если флаг не установлен, одновременно выполняются операции освобождения семафора и перехода в режим ожидания сигнала.

Здесь очень важен третий этап, на котором *Linux* позволяет выполнить атомарную операцию освобождения исключающего семафора и перехода в режим ожидания сигнала. Вмешательство других потоков при этом не допускается.

Перечисленные ниже этапы должны выполняться всякий раз, когда программа тем или иным способом меняет результат проверки условия, контролируемого условной переменной (в нашей программе условие — это значение флага):

1. Захватить исключающий семафор, дополняющий сигнальную переменную,
2. Выполнить действие, включающее изменение результата проверки условия (в нашем случае — установить флаг).
3. Послать сигнал (возможно, широковещательный) об изменении условия.
4. Освободить исключающий семафор.

Рассмотрим измененную версию фрагмента программы, в которой флаг защищается сигнальной переменной. В функции thread_function() исключающий семафор захватывается до того, как будет проверено значение переменной thread_flag. Захват автоматически снимается функцией pthread_cond_wait() перед тем, как поток оказывается заблокированным, и также автоматически восстанавливается по завершении функции.

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag () {
	/* Initialize the mutex and condition variable. */
	pthread_mutex_init(&thread_flag_mutex, NULL);
	pthread_cond_init(&thread_flag_cv, NULL);
	/* Initialize the flag value. */
	thread_flag = 0;
}

/* Calls do_work repeatedly while the thread flag is set; blocks if
the flag is clear. */
void* thread_function(void* thread_arg) {
	/* Loop infinitely. */
	while (1) {
		/* Lock the mutex before accessing the flag value. */
		pthread_mutex_lock(&thread_flag_mutex);
		while (!thread_flag)
			/* The flag is clear. Wait for a signal on the condition
			variable, indicating that the flag value has changed. When the
			signal arrives and this thread unblocks, loop and check the
			flag again. */
			pthread_cond_wait (&thread_flag_cv, &thread_flag_mutex);
		/* When we've gotten here, we know the flag must be set. Unlock
		the mutex. */
		pthread_mutex_unlock (&thread_flag_mutex);
		/* Do some work. */
		do_work ();
	}
```

```
        return NULL;
}

/* Sets the value of the thread flag to FLAG_VALUE. */
void set_thread_flag (int flag_value) {
        /* Lock the mutex before accessing the flag value. */
        pthread_mutex_lock (&thread_flag_mutex);
        /* Set the flag value, and then signal in case thread_function is
        blocked, waiting for the flag to become set. However,
        thread_function can't actually check the flag until the mutex is
        unlocked. */
        thread_flag = flag_value;
        pthread_cond_signal(&thread_flag_cv);
        /* Unlock the mutex. */
        pthread_mutex_unlock(&thread_flag_mutex);
}
```

Условие, контролируемое условной переменной, может быть произвольно сложным. Но, перед выполнением любой операции, способной повлиять на результат проверки условия, необходимо захватить исключающий семафор, и только после этого можно посылать сигнал.

Сигнальная переменная может вообще не быть связана ни с каким условием, а служить лишь средством блокирования потока до тех пор, пока какой-нибудь другой поток не «разбудит» его. Для этой же цели может использоваться и семафор. Принципиальная разница между ними заключается в том, что семафор «запоминает» сигнал, даже если ни один поток в это время не был заблокирован, а сигнальная переменная регистрирует сигнал только в том случае, если его ожидает какой-то поток. Кроме того, семафор всегда разблокирует лишь один поток, тогда как с помощью функции pthread_cond_broadcast() можно разблокировать произвольное число потоков