Лабораторная работа 3

МЕХАНИЗМЫ МНОГОПОТОЧНОСТИ

Цель работы — изучить принципы разработки многопоточных программ, изучить программный интерфейс операционных систем для организации многопоточности, получить навыки организации взаимодействия потоков в многопоточных программах.

Теоретические сведения

Многопоточные программы

Многозадачность является важнейшим свойством ОС. Для поддержки этого свойства ОС определяет и оформляет для себя те внутренние единицы вычислительной работы, между которыми и будет разделяться процессор и другие ресурсы компьютера. Эти единицы могут в разных системах иметь разные названия (задача, задание, процесс, нить) и иметь принципиальные различия.

Основной вычислительной единицей для многих ОС является процесс. Операционная система поддерживает изоляцию процессов: у каждого процесса имеется свое виртуальное адресное пространство. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы машины, конкурируют друг с другом. В общем случае процессы принадлежат разным пользователям, разделяющим один компьютер, и ОС берет на себя роль арбитра в спорах процессов за ресурсы.

При мультипрограммной системе, которой являются многие современные ОС, отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу каждого из участников за счет дополнительных затрат времени на ожидание освобождения ресурса). Однако вычислительная задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе позволяет ускорить ее решение. Например, в ходе выполнения задачи происходит обращение к внешнему устройству, и на время этой операции можно не блокировать полностью выполнение процесса, а продолжить вычисления по другой "ветви" процесса.

Для этих целей современные ОС предлагают использовать *механизм многопоточной обработки (multithreading)*. При этом вводится новое понятие "нить" или “поток” (thread), а понятие "процесс" в значительной степени меняет смысл.

Нить (thread) – это независимый поток управления, выполняемый в контексте некоторого процесса. Все, что не относится к потоку управления (виртуальная память, дескрипторы открытых файлов и т.д.), остается в общем контексте процесса. Сущности, которые характерны для потока управления (регистровый контекст, стеки разного уровня и т.д.), переходят из контекста процесса в контекст нити.

Все нити процесса выполняются в его контексте, но каждая нить имеет свой собственный контекст. Контекст нити, как и контекст процесса, состоит из пользовательской и ядерной составляющих. Пользовательская составляющая контекста нити включает индивидуальный стек нити для пользовательского режима.

Поскольку нити одного процесса выполняются в общей виртуальной памяти (все нити процесса имеют равные права доступа к любым частям виртуальной памяти процесса), стек (сегмент стека) любой нити процесса не защищен от произвольного (например, по причине ошибки) доступа со стороны других нитей.

Нити во многих отношениях подобны процессам. Нити могут динамически создаваться и уничтожаться, и переходить из состояния в состояние. Подобно традиционным процессам (то есть процессам, состоящим из одной нити), нити могут находиться в одном из следующих состояний: *выполнение*, *ожидание* и *готовность*. Пока одна нить заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так, как это делают процессы, в соответствии с различными стратегиями планирования.

Однако различные нити в рамках одного процесса не настолько независимы, как отдельные процессы. Все такие нити имеют одно и то же адресное пространство, это означает, что они разделяют одни и те же глобальные переменные. Изоляция нитей одного процесса не требуется, поскольку все нити одного процесса решают общую задачу для одного пользователя, и механизм нитей используется для более быстрого ее решения путем распараллеливания. При этом программисту очень важно получить в свое распоряжения удобные средства организации взаимодействия частей одной задачи – механизмов синхронизации нитей.

Нити имеют собственные:

- программный счетчик,
- стек,
- регистры,
- состояние.

Нити разделяют:

- адресное пространство,
- глобальные переменные,
- открытые файлы,
- обработчики и маски сигналов,
- таймеры,
- семафоры, мутексы и другие объекты синхронизации,
- статистическую информацию.

Операционная система Linux, аналогично традиционным версиям UNIX, изначально разрабатывалась без поддержки многопоточности. При дальнейшем развитии системы многопоточность была реализована путем представления на уровне API нескольких отдельных процессов, разделяющих общее адресное пространство и открытые ресурсы ОС, в виде одной многопоточной программы.

Управление такими процессами, организацию взаимодействия потоков и их синхронизацию обеспечивает библиотека Linux Threads. Программные интерфейсы библиотеки Linux Threads соответствуют стандарту POSIX для управления нитями процесса.

При создании многопоточного приложения в ОС Linux необходимо использовать заголовочный файл *pthread.h*. При компиляции и сборке многопоточной программы в среде Linux следует использовать специальные опции компилятора и явно указывать используемую библиотеку Linux Threads:

```
g++ test.cpp –D_REENTRANT -lpthread
```

Устанавливаемый в явном виде макрос препроцессора _REENTRANT требует использовать "потокобезопасные" аналоги функций стандартной библиотеки Си. Без указания этого макроса программа может быть скомпилирована и слинкована с другой бибилиотекой, содержащей не "потокобезопасные" варианты функций. В этом случае программа будет работать неправильно.

Операционная система Windows, основанная на ядре NT, изначально проектировалась и разрабатывалась с поддержкой многопоточности. При создании многопоточного приложения в ОС Windows необходимо использовать заголовочный файл *windows.h*.

Управления нитями в ОС Linux

Основные операции, определенные над нитями — это *создание* и *завершение* нити.

В ОС Linux нить создается вызовом функции `pthread_create()`, которая создает нить и запускает ее. Создание нити отличается от создания процесса в том, что не существует отношений «родитель-потомок» между нитями. Все нити, исключая начальную (initial) — создающуюся автоматически при создании процесса, находятся на одном уровне иерархии. Каждая нить имеет точку входа — стартовую процедуру с одним аргументом. Эта же точка входа может использоваться другими нитями. Прототип функции, которая может являться точкой входа, выглядит следующим образом:

```
void* thread_entry(void* param);
```

Функция `pthread_create()` возвращает идентификатор новой нити. Впоследствии идентификатор можно использовать для осуществления различных операций над нитью. Идентификатор текущей нити можно узнать, вызвав процедуру `pthread_self()`, он имеет тип `pthread_t`. Не рекомендуется использовать целочисленные идентификаторы нитей в программе, т.к. в различных системах тип pthread_t. может быть не арифметическим и даже не указателем.

Для уничтожения текущей нити (завершения работы текущей нити) используется функция `pthread_exit()`. Нить автоматически уничтожается при выходе из своей стартовой процедуры (аналогично завершению процесса при выходе из `main()`).

Функция `pthread_join()` предоставляет простой механизм, позволяющий нити ждать завершения другой нити. Функция блокирует исполнение текущей нити, пока не завершится указанная нить. Если несколько нитей ожидают завершения какой-либо нити, то выполнение операции зависит от порядка обращений к процедуре `pthread_join()` и момента завершения нити-цели.

Следующий пример демонстрирует использование перечисленных функций. Начальная нить создает три нити. Все созданные нити имеют одинаковую точку входа, но разные параметры — строки, которые печатаются на экран. Основная нить процесса ожидает завершения трех созданных нитей.

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#define TCNT (3)

void* thread_entry(void* param)
{
    const char* msg = (const char*) param;
    int i;
    for(i = 0; i < 5; i++)
        printf("%s ", msg);
}


int main()
{
    pthread_t tids[TCNT];
    const char* msgs[TCNT] = { "Hello!", "Bonjour!", "Terve!" };
    int i;

    // Создание потоков
    for(i = 0; i < TCNT; i++)
    {
        if ( 0 != pthread_create(&tids[i], 0, thread_entry, (void*) msgs[i]) )
        {
            printf("pthread_create failed. errno: %d\n", errno);
            return -1;
        }
    }

    // Ожидание завершения потоков
    for(i = 0; i < TCNT; i++)
        pthread_join(tids[i], 0);

    return 0;
}
```

Функция `pthread_join()` блокирует выполнение текущей нити, пока не завершится указанная. С помощью функции `pthread_timedjoin_np()` текущая нить может выполнять ожидание указанной нити до указанного времени (указывается в последнем параметре).

Следущий пример демонстрирует использование функции: главная нить создает нить для вычислений и ожидает ее завершения, периодически выводит на экран время, прошедшее с момента запуска. После завершения вычислений нитью программа завершается и распечатывает время выполнения (в миллисекундах).

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

void* thread_entry(void* param)
{
    // Выполнение некоторых вычислений
    int val = 0x7fffffff, i, c = 0;
```

```
    for(i = 1; i < val; i++)
    {
        if (val % i == 0)
          c++;
    }
    return (void*) ( (char*) 0 + c );
}

unsigned long to_ms(struct timespec* tm)
{
    return ((unsigned long) tm->tv_sec * 1000 +
            (unsigned long) tm->tv_nsec / 1000000);
}

int main()
{
    struct timespec started, finished;

    pthread_t t;
    int waits = 0;

    printf("Creating threads...\n");
    clock_gettime(CLOCK_REALTIME, &started);

    if ( 0 != pthread_create(&t, 0, thread_entry, 0) )
    {
        printf("pthread_create failed. errno: %d\n", errno);
        return -1;
    }

    while(1)
    {
        struct timespec tm;
        int res;

        // Взять текущее время, добавить 1 сек
        clock_gettime(CLOCK_REALTIME, &tm);
        tm.tv_sec += 1;

        // Ожидание завершения потока (до указанного времени)
        res = pthread_timedjoin_np(t, 0, &tm);
        waits++;

        if (res == ETIMEDOUT) // Таймаут
        {
            printf("Working: %d second(s)...\n", waits);
        }
        else if (res == 0) // Поток завершился
        {
            break;
        }
    }

    clock_gettime(CLOCK_REALTIME, &finished);

    printf("Thread finished. Execution time: %lu ms\n", to_ms(&finished) - to_ms(&started) );

    return 0;
}
```

Управления нитями в ОС Windows

В ОС Windows для создания нити используется функция `CreateThread()`. Прототип функции, которая может являться точкой входа нити, выглядит следующим образом:

```
DWORD WINAPI thread_entry(void* param);
```

Функция возвращает описатель созданной нити (`HANDLE`). Описатель используется для выполнения операций с нитью. Например, текущая нить может ожидать завершения указанной нити указанное время с помощью функции `WaitForSingleObject()`. Функция `WaitForMultipleObjects()` используется для одновременного ожидания завершения нескольких нитей. Время ожидания указывается в последнем параметре в миллисекундах. Для бесконечного ожидания указыается специальная константа `INFINITE`.

Если программа не имеет необходимости дальше работать с нитью, она должна освободить описатель созданной нити — с помощью функции `CloseHandle`.

Следующий пример демонстрирует использование перечисленных функций. Начальная нить создает три нити. Все созданные нити имеют одинаковую точку входа, но разные параметры — строки, которые печатаются на экран. Основная нить процесса ожидает завершения трех созданных нитей.

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

#define TCNT (3)

DWORD WINAPI thread_entry(void* param)
{
	const char* msg = (const char*)param;
	int i;
	for (i = 0; i < 5; i++)
		printf("%s ", msg);
	return 0;
}


int main()
{
	HANDLE handles[TCNT];
	const char* msgs[TCNT] = { "Hello!", "Bonjour!", "Terve!" };
	int i;

	// Создание потоков
	for (i = 0; i < TCNT; i++)
	{
		handles[i] = CreateThread(0, 0, thread_entry, (void*)msgs[i], 0, NULL);

		if (0 == handles[i])
		{
			printf("CreateThread failed. GetLastError(): %u\n", GetLastError());
			return -1;
		}
	}

	// Ожидание завершения потоков
	WaitForMultipleObjects(TCNT, handles, TRUE, INFINITE);

	// Закрытие описателей созданных потоков
	for (i = 0; i < TCNT; i++)
		CloseHandle(handles[i]);

	return 0;
}
```

В следущем примере главная нить создает нить для вычислений и ожидает ее завершения, периодически выводит на экран время, прошедшее с момента запуска. После завершения вычислений программа завершается.

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

DWORD WINAPI thread_entry(void* param)
{
	// Выполнение некоторых вычислений
	int val = 0x7fffffff, i, c = 0;
	for (i = 1; i < val; i++)
	{
		if (val % i == 0)
			c++;
	}
	return (DWORD) c;
}


int main()
{
	HANDLE t;
	int waits = 0;

	printf("Creating thread...\n");

	t = CreateThread(0, 0, thread_entry, 0, 0, 0);

	if (0 == t)
	{
		printf("CreateThread failed. GetLastError(): %u\n", GetLastError());
		return -1;
	}

	while (1)
	{
		// Ожидание завершения потока (1 сек)
		DWORD waitres = WaitForSingleObject(t, 1000);
		waits++;

		if (waitres == WAIT_TIMEOUT) // Таймаут
		{
			printf("Working: %d second(s)...\n", waits);
		}
		else if (waitres == WAIT_OBJECT_0) // Поток завершился
		{
			printf("Thread finished.\n");
			break;
		}
	}

	CloseHandle(t);

	return 0;
}
```

Синхронизация нитей в ОС Linux

Простейшим механизмом взаимного исключения нитей являются *мутексы* (*mutex — mutual exclusion lock*). Мутекс определяется в виде структуры данных `pthread_mutex_t`. Перед использованием мутекса необходимо его инициализировать при помощи функции `pthread_mutex_init()`. Мутекс

следует инициализировать один раз. Повторных вызовов функций инициализации мутексов (или семафоров) следует избегать, поскольку это может вызвать нарушение работы блокированных нитей.

Если мутекс больше не нужен, его можно уничтожить функцией `pthread_mutex_destroy()`. Если мутекс определен как глобальная структура, его уничтожение необязательно, если мутекс определен в стеке одной из нитей, то он должен быть уничтожен до завершения нити.

Мутекс может находиться в одном из двух состояний – *свободен* или *блокирован*. После создания при помощи `pthread_mutex_init()` мутекс является свободным.

Функция `pthread_mutex_lock()` блокирует (захватывает) указанный мутекс:

- если мутекс свободен, он переводится в заблокированное состояние и текущая нить продолжает исполнение;
- если мутекс уже блокирован другой нитью, то текущая нить переводится в состояние ожидания, пока мутекс не освободится;

Функция `pthread_mutex_unlock()` освобождает мутекс, блокированный текущей нитью:

- если мутекс свободен — выдается ошибка;
- если мутекс блокирован текущей нитью, то мутекс или освобождается, или передается одной из нитей, ожидающих его освобождения, что приведет к активизации одной из ожидающих нитей;
- если мутекс принадлежал другой нити (был захвачен другой нитью) — выдается ошибка.

Вызов `pthread_mutex_unlock()` никогда не приводит к блокировке текущей нити.

Таким образом, мутекс в любой момент времени может быть заблокирован только одной нитью. Для организации "*критической секции*", т.е. кода, выполняющегося в каждый момент времени только одной нитью, при помощи мутексов используется следующая схема:

```
pthread_mutex_lock(&mutex);
<код критической секции>
pthread_mutex_unlock(&mutex);
```

В многопоточной программе при объявлении переменных, например глобальных, используемых несколькими потоками, следует указывать ключевое слово *volatile*. Это ключевое слово сообщает компилятору, что переменная может быть изменена из параллельно работающего кода и ее значение всегда следует брать из памяти. Если же *volatile* не указать, то компилятор может применять к переменной оптимизацию — разместить ее, например, в регистре и не перечитывать ее значение после выхода из критической секции. Такая оптимизация приведет к

некорректной логике работы программы. Ключевое слово *volatile* исключит такие проблемы. Использование ключевого слова допустимо также и для полей структур.

Следующий пример демонстрирует использование объектов синхронизации для доступа к данным из нескольких нитей с помощью "критических секций", организованных через мутексы. Основная нить создает 5 "вычисляющих" нитей и ожидает их завершения. Вычисляющие нити производят вычисления и периодически обновляют текущий прогресс. Структуры данных для хранения прогресса являются общими для всех нитей — доступ к ним организован с помощью мутексов. Основная нить периодически выводит текущий прогресс вычислений.

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>

#define TCNT (5)

struct thread_info
{
    volatile double progress;
    // Ключевое слово volatile запрещает компилятору оптимизацию progress (хранение в регистрах)
};

pthread_t tids[TCNT];
struct thread_info threads[TCNT];
pthread_mutex_t mutex;

void* thread_proc(void* param)
{
    // Индекс потока передается в param
    int idx = (char*) param - (char*) 0;
    int i, value = 0x0fffffff, c = 0;

    for(i = 1; i < value; i++)
    {
        if (value % i == 0)
            c++;

        // Периодическое обновление прогресса (в %)
        if (i % 100 == 0)
        {
            // Эксклюзивный доступ к данным организован с помощью мутекса mutex
            pthread_mutex_lock(&mutex);
            threads[idx].progress = ((double) i / (double) value) * 100;
            pthread_mutex_unlock(&mutex);
        }
    }

    return 0;
}


int main()
{
    int i;

    // Инициализация
    memset(&threads, 0, sizeof(threads));
    pthread_mutex_init(&mutex, 0);

    // Создание потоков
```

```
    printf("Creating threads...\n");
    for(i = 0; i < TCNT; i++)
       pthread_create(&tids[i], 0, thread_proc, (void*) ((char*) 0 + i));

    // Ожидание завершения всех потоков и периодический вывод статуса.
    while(1)
    {
        struct timespec tm;
        int all_terminated = 1;

        // Ожидание одну секунду: что все потоки завершились
        clock_gettime(CLOCK_REALTIME, &tm);
        tm.tv_sec += 1;

        for(i = 0; i < TCNT; i++)
        {
            if (tids[i])
            {
                 int res = pthread_timedjoin_np(tids[i], 0, &tm);
                 if (res == 0) // Поток завершен
                 {
                     tids[i] = 0;
                 }
                 else // Поток еще не завершен
                 {
                     all_terminated = 0;
                     break;
                 }
            }
        }

        if (!all_terminated)
        {
            // Потоки еще работают, вывести прогресс
            printf("Progress: ");
            pthread_mutex_lock(&mutex);
            for(i = 0; i < TCNT; i++)
              printf("%.2lf%% ", threads[i].progress);
            pthread_mutex_unlock(&mutex);
            printf("\n");
        }
        else
        {
            // Все потоки завершились, выход
            break;
        }
    }


    // Освобождение памяти
    pthread_mutex_destroy(&mutex);
    printf("All threads complete.\n");

    return 0;
}
```

Более сложными механизмами взаимного исключения, поддерживаемыми библиотекой Linux Threads, являются *семафоры*. Семафор описывается структурой `sem_t`, одиним из полей этой структуры является текущее состояние (значение) семафора. Инициализация семафора производится при помощи функции `sem_init()`, при инициализации устанавливается начальное значение семафора (обычно — 0). При завершении использования семафора его можно уничтожить функцией `sem_destroy()`.

Нулевое значение семафора сигнализирует, что свободные ресурсы отсуствуют, любое ненулевое значение используется как счетчик свободных ресурсов.

Ожидание события на семафоре реализуется функцией `sem_wait()`, которая блокирует текущий поток до того момента, пока значение семафора не станет отличным от нуля. При наступлении этого события ожидающий поток разблокируется, а значение значение семафора автоматически уменьшается. Определен неблокирующий вариант фукнции ожидания — `sem_trywait()`, которая проверяет значение семафора и, если оно ненулевое, выполняет декремент этого значения — захват ресурса. Если значение семафора равно нулю, функция `sem_trywait()` немедленно завершается с ошибкой.

Увеличение значения семафора выполняет функция `sem_post()`. Эта функция не приводит к блокировке текущего потока, но вызывает активизацию других блокированных на семафоре потоков, если таковые имеются.

Для определения текущего значения семафора введена функция `sem_getvalue()`. При ее использовании следует помнить, что значение семафора может быть в любой момент изменено другими активными потоками процесса и в общем случае ее использование крайне не рекомендуется.

Следующий пример демонстрирует использование семафоров для организации "критической секции" (обеспечение эксклюзивного доступа к массиву active), а также для ограничения количества одновременно работающих нитей. Основная нить создает 10 нитей для вычислений, но вычислять одновременно могут только 3. Ограничение реализовано следующим способом: перед вычислением каждая нить пытается "захватить" семафор, и, если "захват" прошел успешно, выполняет часть вычислений, а затем "освобождает" семафор — чтобы уступить выполнение другому потоку.

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define MAX_WTCNT (3)
#define TCNT (10)

sem_t sem;
pthread_t threads[TCNT];

// Хранение активных потоков
sem_t sem_act;
volatile int active[TCNT] = { 0 }; // volatile запрещает компилятору оптимизации массива active

void set_active(int idx, int act)
{
   sem_wait(&sem_act);
   active[idx] = act;
   if (act)
   {
        int i;
        printf("Active threads: ");
        for(i = 0; i < TCNT; i++)
           if (active[i])
```

```
                printf("%d ", i);
            printf("\n");
        }
        sem_post(&sem_act);
}

void do_thread_task()
{
    int s = 1, m = 0x0fffffff, i;
    for(i = 1; i < m; i++)
        s *= i;
}

void* thread_proc(void* param)
{
    int idx = (char*) param - (char*) 0;
    int i;

    // Выполняется несколько итераций
    for(i = 0; i < 10; i++)
    {
        // Перед выполнением работы выполняется захват семафора.
        // Это гарантирует, что работать будет только MAX_WTCNT потоков одновременно
        sem_wait(&sem);

        set_active(idx, 1);
        do_thread_task();
        set_active(idx, 0);

        // Освобождение семафора. При этом разблокируется другой поток
        sem_post(&sem);
    }

    return 0;
}


int main()
{
    int i;

    // Инициализация семафора sem: одновременно могут выполняться MAX_WTCNT потоков
    sem_init(&sem, 0, MAX_WTCNT);

    // Инициализация семафора sem_act для доступа к массиву active
    sem_init(&sem_act, 0, 1);

    // Создание потоков
    for(i = 0; i < TCNT; i++)
        pthread_create(&threads[i], 0, thread_proc, (char*) 0 + i);

    // Ожидание потоков
    for(i = 0; i < TCNT; i++)
        pthread_join(threads[i], 0);

    // Освобождение памяти
    sem_destroy(&sem);
    sem_destroy(&sem_act);

    return 0;
}
```

Еще одним способом синхронизации потоков, поддерживаемым библиотекой Linux Threads, являются *условные переменные,* называемые иногда также *мониторами*.

Условная переменная описывается структурой `pthread_cond_t`. Инициализация переменной производится при помощи функции `pthread_cond_init()`, первым параметром передается указатель на экземпляр структуры условной переменной, вторым параметром — некоторые аттрибуты (обычно — 0). При завершении использования условной переменной необходимо ее уничтожить функцией `pthread_cond_destroy()`.

Синхронизация потоков с помощью условных переменных организуется с помощью конечного числа операций.

Поток, ожидающий наступления какого-либо события, вызывает `pthread_cond_wait()`. Первым параметром этой функции является указатель на структуру условной переменной, вторым — предварительно созданный мутекс, необходимый для собственной синхронизации. На момент вызова этот мутекс должен быть в **захваченном состоянии**, т.е. перед вызовом `pthread_cond_wait` следует вызвать `pthread_mutex_lock`. При вызове `pthread_cond_wait` поток переводится в состояние ожидания и прекращает исполнение до тех пор, пока какой-либо другой поток не переведет условную переменную в сигнальное состояние. Ожидать перевода в сигнальное состояние может одновременно и несколько потоков. Во время ожидания перевода условной переменной в сигнальное состояние мутекс, указанный во втором аргументе, находится в свободном (не захваченном) состоянии. После перевода условной переменной в сигнальное состояние мутекс автоматически переводится в захваченное состояние — т.е. она станет такое же, как и было до вызова `pthread_cond_wait`.

Для перевода условной переменной в сигнальное состояние выполняющийся поток может вызывать одну из следующих функций: `pthread_cond_signal()` или `pthread_cond_broadcast()`. Обе эти функции имеют единственный аргумент — указатель на структуру условной переменной. При вызове первой функции будет разблокирован и продолжит выполнение только один ожидавший условную переменную поток. Если переменную ожидало несколько потоков, то разблокирован будет первый поток, остальные — продолжат оставаться в заблокированном состоянии до тех пор, пока `pthread_cond_signal` не будет вызвана еще раз. При вызове `pthread_cond_broadcast` будут разблокированы все ожидающие условную переменную потоки. До вызова `pthread_cond_broadcast` или `pthread_cond_signal` вызывающему их потоку **следует захватить мутекс**, переданный ожидающим потоком в `pthread_cond_wait`, и освободить его после вызова. Это гарантирует синхронизацию во время передачи и ожидания сигналов: никаких два потока программы не могут вызывать `pthread_cond_wait`, `pthread_cond_signal` или `pthread_cond_broadcast` одновременно. Эти ограничения должны соблюдаться в обязательном порядке, иначе функции библиотеки Linux Thread будут возвращать ошибки, а программа окажется в рассинхронизированном состоянии.

Следует учитывать, что если один из потоков вызвал `pthread_cond_singal` или `pthread_cond_broadcast` раньше, чем другой

вызвал pthread_cond_wait, то ожидающий поток "пропустит" этот сигнал, т.е. заблокируется и будет находиться в таком состоянии до следующего поступления сигнала. Эту особенность следует учитывать при разработке многопоточных приложений с помощью Linux Threads. Такая ситуация является логической ошибкой алгоритма.

Момимо вызова pthread_cond_wait ожидающий поток может вызвать pthread_cond_timedwait. Отличие второй функции от первой состоит в том, что она имеет третий аргумент — время, до которого следует ждать поступления сигнала по этой условной переменной. В параметре передается именно абсолютное время (например, до 18:57:53), а не временной интервал (например, ждать 10 сек). В случае, если системные часы достигли этого времени, а сигнал по условной переменной так и не поступил, ожидающий поток разблокируется и продолжит выполенение. Функция pthread_cond_timedwait при этом вернет значение ETIMEDOUT. Если же сигнал поступил, то функция вернет 0.

Следующий пример демонстрирует использование условных переменных для организации синхронизации потоков. В данном примере создаются несколько потоков, условно генерирующие случайные буферы данных, случайного размера. Выработка каждого такого "случайного" байта занимает значительное время. Помимо потоков-генераторов создается несколько потоков, выводящих сгенерированные буферы в файлы. Запись каждого байта условно занимает также значительное время. Для уведомления о том, что в буфере есть данные, используется сигнал, посылаемый по Условной переменной. Поток-генератор уведомит таким сигналом первый спящий поток-писатель, а тот, в свою очередь, "просыпается", считывает сгенерированные данные из очереди и начинает их сохранение. Алгоритм работы потоков-писателей реализован так, что пропуск сигнала по Условной переменной не приводит к взаимной блокировке потоков или бесконечному ожиданию.

```
#include <pthread.h>
#include <time.h>
#include <malloc.h>
#include <unistd.h> // usleep

// Количество потоков-генераторов данных
#define THREADS_GEN_CNT    (3)
// Количество потоков-писателей данных
#define THREADS_WRT_CNT    (4)
// Суммарное количество задач генерации
#define GEN_TASKS (11)

// Струтура данных для создания линейного динамический списка для
// хранения готовых к записи буферов.
struct ready_data {
    char* data;
    unsigned int size;
    int task_num; // Номер задачи, от 1 до GEN_TASKS
    struct ready_data volatile *next;
};

// Указатели на первый и последний элемент списка.
// Доступ к списку синхронизируется с помощтю g_queue_mutex.
// volatile запрещает компилятору хранение данных в регистрах в целях оптимизации
struct ready_data volatile * g_queue_first = NULL;
```

```
struct ready_data volatile * g_queue_last = NULL;
pthread_mutex_t g_queue_mutex;

// Общее количество выполненных или выполняемых заданий генерации.
// Если оно превышает GEN_TASKS, то все задания генерации выполнены и
// потоки-генераторы могут завершить свою работу.
static volatile int g_gen_counter = 0;

// Общее количество выполненных или выполняемых заданий записи.
// Если оно превышает GEN_TASKS, то все задания записи выполнены и
// потоки-писатели могут завершить свою работу.
static volatile int g_wrt_counter = 0;

// Усл. переменная. Сигнал посылается когда в очередь заносится очередной
// готовый к записи буфер данных.
pthread_cond_t g_cond_data_ready;


// Функция добавляет готовый для записи массив в список.
// Мутекс g_queue_mutex должен быть захвачен!
void queue_push_back(char* data, unsigned int size, int task_num)
{
    struct ready_data volatile * item = (struct ready_data*) malloc(sizeof(struct ready_data));
    item->data = data;
    item->size = size;
    item->task_num = task_num;
    item->next = NULL;

    // Добавление элемента в список. Атомарность операции гарантируется
    // мутекс, который должен быть захвачен во время вызова этой функции
    if (g_queue_last)
    {
        g_queue_last->next = item;
        g_queue_last = item;
    }
    else
    {
        g_queue_first = item;
        g_queue_last = item;
    }
}

// Функция извлекает массив из списка. Возвращает 0, если список был пуст.
// Мутекс g_queue_mutex должен быть захвачен!
int queue_pop_first(char** data_ptr, unsigned int* size_ptr, int* task_num_ptr)
{
    struct ready_data volatile * item;
    int ret = 0;
    item = g_queue_first;
    if (item)
    {
        g_queue_first = item->next;
        if (!g_queue_first)
            g_queue_last = NULL;
        *data_ptr = item->data;
        *size_ptr = item->size;
        *task_num_ptr = item->task_num;
        free( (void*) item);
        ret = 1;
    }
    return ret;
}

void* gen_thread(void* arg)
{
    int local_counter = 0;

    while(1)
```

```
    {
        int task_num;
        char* data;
        unsigned int size, i;
        unsigned char rnd_sz;
        FILE* f;

        // Проверяем - стоит ли генерировать еще один буфер
        pthread_mutex_lock(&g_queue_mutex);
        if (g_gen_counter >= GEN_TASKS)
        {
            // Если все задачи уже готовы - прерываем цикл генераций
            pthread_mutex_unlock(&g_queue_mutex);
            break;
        }

        // Иначе - отмечаем, что мы занялись еще одной задачей генерации
        g_gen_counter++;
        task_num = g_gen_counter; // Номер задачи берем из g_gen_counter. Его значение всегда уникально

        printf("Generating task %d.\n", task_num);

        // Особождаем мутекс - чтобы все другие потоки могли выполнять свои задачи
        pthread_mutex_unlock(&g_queue_mutex);

        // Генерируем "случайные" данные.
        // В качестве источника случайной последовательности выступает бесконечный файл-устройство
        // /dev/urandom
        f = fopen("/dev/urandom", "rb");
        rnd_sz = 0;
        // Считываем несколько байт - это будет размер нашего случайного буфера
        fread(&rnd_sz, sizeof(rnd_sz), 1, f);
        size = rnd_sz;
        data = (char*) malloc(size);
        for(i = 0; i < size; i++)
        {
            // Условно генерация каждого байта занимает 25мс.
            fread(&data[i], sizeof(char), 1, f);
            usleep(25 * 1000); // Арумент usleep указывается в микросекундах
        }
        fclose(f);

        // Добавляем данные в список. "Сигналим" с помощью Усл. переменной, что список
        // пополнился. При этом один из ожидающих потоков-писателей должен "забрать" задачу.
        // Если все потоки-писатели заняты и ни один не ждет сигнала по усл. переменной, то
        // они все равно заберут задачу, т.к. после записи они СНАЧАЛА пытаются забрать задачу,
        // а уже потом - переходят к ожиданию. И все это на одном и том же мутексе, что гарантирует
        // правильную последовательность.

        //  pthread_cond_signal  следует  ВСЕГДА  ВЫЗЫВАТЬ  с  заблокированным  мутексом  -  тем  же,  что  и
передан
        // в pthread_cond_wait.
        pthread_mutex_lock(&g_queue_mutex);
        queue_push_back(data, size, task_num);
        pthread_cond_signal(&g_cond_data_ready);
        pthread_mutex_unlock(&g_queue_mutex);

        local_counter++;
    }

    printf("Generate thread complete. Performed tasks: %d\n", local_counter);

    return 0;
}

// Поток-писатель. Осуществляет считывание задач из очереди и выполняет запись данных в файл.
// Если данные еще не готовы - будет ждать их (с помощью Усл. переменной g_cond_data_ready).
void* wrt_thread(void* arg)
```

```
{
    int local_counter = 0;

    while(1) // Цикл работает пока не будут выполнены все задачи записи
    {
        char* data;
        unsigned int size;
        int task_num;

        // Пытаемся забрать очередной готовый буфер из очереди.
        pthread_mutex_lock(&g_queue_mutex);

        if (queue_pop_first(&data, &size, &task_num))
        {
            FILE* f;
            char fn[16];
            int i;

            // Увеличивается количество выполняемых задач записи
            g_wrt_counter++;

            printf("Writing task %d. size = %u\n", task_num, size);

            // Забрать удалось, освобождаем мутекс
            pthread_mutex_unlock(&g_queue_mutex);

            // И пишем данные в файл
            sprintf(fn, "buf%04d.txt", task_num);
            f = fopen(fn, "wb");
            for(i = 0; i < size; i++)
            {
                // Цикл эмулирует "медленную" запись. Каждый байт записывается
                // 30 мс
                fwrite(&data[i], sizeof(char), 1, f);
                usleep(30 * 1000); // Аргумент указывается в микросекундах
            }
            fclose(f);
            free(data);

            local_counter++; // Сохраняем локально - сколько задач записи выполнил этот thread
        }
        else
        {
            // Проверка - все ли задачи записи выполнены
            if (g_wrt_counter >= GEN_TASKS)
            {
                // Все выполнены, больше поток-писатель не нужен, завершаемся.
                pthread_mutex_unlock(&g_queue_mutex);
                break;
            }

            // Иначе, чтобы исключить "активное ожидание", переводим поток в состояние ожидания.
            // Во время ожидания мутекс будет в свободном состоянии и с ним могут работать другие
потоки
            pthread_cond_wait(&g_cond_data_ready, &g_queue_mutex);
            pthread_mutex_unlock(&g_queue_mutex);
        }
    }

    printf("Writer thread complete. Performed tasks: %d\n", local_counter);


    // Если текущий поток-писатель "понял", что он выполнил все задачи, то это означает,
    // что и другие потоки, ожидающие поступления задач, должны завершиться. Они могут
    // находиться в состоянии ожидания сигнала. "Будим" их - тогда они проснутся и увидят,
    // что все задачи записи закончены и работать больше не нужно.
    pthread_mutex_lock(&g_queue_mutex);
    pthread_cond_broadcast(&g_cond_data_ready);
```

```
    pthread_mutex_unlock(&g_queue_mutex);

    return 0;
}

unsigned long to_ms(struct timespec* tm)
{
    return ((unsigned long) tm->tv_sec * 1000 +
            (unsigned long) tm->tv_nsec / 1000000);
}

int main()
{
    int i;
    struct timespec tm_start, tm_end;
    pthread_t tids[THREADS_GEN_CNT + THREADS_WRT_CNT];
    int cnt = 0;

    pthread_mutex_init(&g_queue_mutex, 0);
    pthread_cond_init(&g_cond_data_ready, 0);

    // Создание потоков-генераторов
    for(i = 0; i < THREADS_GEN_CNT; i++)
    {
        pthread_create(&tids[cnt], 0, gen_thread, NULL);
        cnt++;
    }

    // Создание потоков-писателей
    for(i = 0; i < THREADS_WRT_CNT; i++)
    {
        pthread_create(&tids[cnt], 0, wrt_thread, NULL);
        cnt++;
    }

    // Засекаем время начало работы
    clock_gettime(CLOCK_REALTIME, &tm_start);

    // Ожидание завершения всех потоков
    for(i = 0; i < cnt; i++)
        pthread_join(tids[i], 0);

    // Засекаем время окончания работы
    clock_gettime(CLOCK_REALTIME, &tm_end);

    // Освобождаем ресурсы, выделенные ОС
    pthread_mutex_destroy(&g_queue_mutex);
    pthread_cond_destroy(&g_cond_data_ready);

    // Выводим суммарное время работы
    printf("Working time: %lu ms\n", to_ms(&tm_end) - to_ms(&tm_start));

    return 0;
}
```

Синхронизация нитей в ОС Windows

ОС Windows предоставляет несколько объектов для синхронизации нитей. Наиболее простым и часто применяемым объектом является "*критическая секция*" (*CRITICAL_SECTION*), позволяющая получать эксклюзивный доступ к каким-либо данным в каждый момент времени только одной нити. Также ОС Windows, предоставляет для синхронизации объекты *мутекс* и *семафор*, поведение которых совпадает с поведением их аналогов в ОС Linux, описанных выше.

В отличие от ОС Linux, в ОС Windows реализована поддержка особых объектов синхронизации — *событие* (*event*).

Критическая секция описывается структурой `CRITICAL_SECTION`. Для инициализации объекта применяется функция `InitializeCriticalSection()`. Для входа в "критическую секцию" применяется функция `EnterCriticalSection()`, для выхода — `LeaveCriticalSection()`. Для освобождения ресурсов, выделенных ОС для обслуживания объекта, следует вызывать `DeleteCriticalSection()`.

Следующий пример демонстрирует применение объекта *CRITICAL_SECTION*. Основная нить создает "вычислительную нить". Вычислительная нить периодически обновляет текущий прогресс вычислений. Основная нить выводит прогресс раз в секунду в стандартный поток вывода. По завершении работы программы выводится время работы программы в миллисекундах.

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

CRITICAL_SECTION cs;
volatile double progress = 0.0;
// Ключевое слово volatile запрещает хранить progress в регистрах


void UpdateProgress(int cur, int max)
{
	double p = (double)cur / (double)max * 100;

	EnterCriticalSection(&cs);
	progress = p;
	LeaveCriticalSection(&cs);
}


DWORD WINAPI thread_fun(void* param)
{
	// Поточная функция для выполнения вычислительной задачи
	int s = 1, i, max = 0x3fffffff;
	for (i = 1; i < max; i++)
	{
		s *= i;

		// Периодическое обновление статуса
		if (i % 100)
			UpdateProgress(i, max);
	}
	return 0;
}

int main()
{
	HANDLE t;
	DWORD finished, started;

	InitializeCriticalSection(&cs);

	printf("Creating thread...\n");

	started = GetTickCount(); // Получение времени (в мс)

	t = CreateThread(0, 0, thread_fun, 0, 0, 0);
```

```
        while (1)
        {
                DWORD res = WaitForSingleObject(t, 1000);
                if (res != WAIT_TIMEOUT)
                        break;
                printf("Progress: ");
                EnterCriticalSection(&cs);
                printf("%.2lf%%\n", progress);
                LeaveCriticalSection(&cs);
        }

        CloseHandle(t);

        DeleteCriticalSection(&cs);

        finished = GetTickCount();

        printf("Thread complete. Execution time: %u ms\n", finished - started);

        return 0;
}
```

Для создания мутекса используется функция `CreateMutex()`. Для "захвата" мутекса необходимо просто дождаться перехода в "сигнальное" состояние с помощью функции `WaitForSingleObject()`. Если нить захватила мутекс, то его "освобождение" (перевод мутекса в "сигнальное" состояние) осуществляется с помощью функции `ReleaseMutex()`. Если мутекс не предполагается использовать дальше — программа должна закрыть описатель с помощью `CloseHandle()`.

Следующий пример демонстрирует использование объектов синхронизации для доступа к данным из нескольких нитей с помощью "критических секций", организованных через мутексы. Основная нить создает 5 "вычисляющих" нитей и ожидает их завершения. Вычисляющие нити производят вычисления и периодически обновляют текущий прогресс. Структуры данных для хранения прогресса являются общими для всех нитей — доступ к ним организован с помощью мутексов. Основная нить периодически выводит текущий прогресс вычислений.

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

#define TCNT (5)

struct thread_info
{
        volatile double progress;
        // Ключевое слово volatile запрещает компилятору оптимизации поля progress
};

HANDLE mutex; // Мутекс для синхронизации доступа к infos
HANDLE threads[TCNT]; // Описатели потоков
struct thread_info infos[TCNT]; // Прогресс потоков

DWORD WINAPI thread_proc(void* param)
{
        // Индекс потока передается в param
        int idx = (char*)param - (char*)0;
        int i, value = 0x0fffffff, c = 0;
```

```
	for (i = 1; i < value; i++)
	{
		if (value % i == 0)
			c++;

		// Периодическое обновление прогресса (в %)
		if (i % 100 == 0)
		{
			// Эксклюзивный доступ к данным организован с помощью мутекса mutex
			WaitForSingleObject(mutex, INFINITE);
			infos[idx].progress = ((double)i / (double)value) * 100;
			ReleaseMutex(mutex);
		}
	}

	return 0;
}

int main()
{
	int i;

	// Инициализация
	memset(&threads, 0, sizeof(threads));
	mutex = CreateMutex(NULL, FALSE, NULL);

	// Создание потоков
	printf("Creating threads...\n");
	for (i = 0; i < TCNT; i++)
		threads[i] = CreateThread(0, 0, thread_proc, (void*)((char*)0 + i), 0, 0);

	// Ожидание завершения всех потоков и периодический вывод статуса.
	while (1)
	{
		if (WAIT_TIMEOUT == WaitForMultipleObjects(TCNT, threads, TRUE, 1000))
		{
			// Потоки еще работают, вывести прогресс
			printf("Progress: ");
			WaitForSingleObject(mutex, INFINITE);
			for (i = 0; i < TCNT; i++)
				printf("%.2lf%% ", infos[i].progress);
			ReleaseMutex(mutex);
			printf("\n");
		}
		else
		{
			// Потоки завершены
			break;
		}

	}

	// Освобождение памяти
	CloseHandle(mutex);
	for (i = 0; i < TCNT; i++)
		CloseHandle(threads[i]);
	printf("All threads complete.\n");

	return 0;
}
```

Семафор — это объект-взаимоисключение (мутекс) со счетчиком. Данный объект позволяет "захватить" себя определенному количеству потоков. После этого "захват" будет невозможен, пока один из ранее "захвативших" семафор потоков не освободит его. Семафоры применяются для ограничения количества потоков,

одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число потоков, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Для создания семафора используется функция `CreateSemaphore()`. Функции указываются настройки безопасности семафора (можно указать NULL), текущее значение, максимальное значение и, опционально, имя. Для захвата семафора применяется функция `WaitForSingleObject()`. В случае, если значение счетчика семафора больше 0 — семафор будет "захвачен", т.е. значение счетчика уменьшится на 1 и нить продолжит выполнение. Если же значение счетчика семафора равно 0 — выполнение нити будет блокировано до тех пор, пока какая-либо другая нить не освободит семафор функцией `ReleaseSemaphore()`, тем самым увеличив значение счетчика.

Следующий пример демонстрирует работу семафоров. Каждая нить моделирует автомобиль, который хочет занять место на парковке. Количество парковочных мест ограничено — максимальное значение семафора. Заняв место на парковке, автомобиль проводит на ней от 3 до 6 секунд и покидает ее, освобождая семафор. При этом другой автомобиль, ожидающий освобождения семафора, получает возможность занять парковочное место.

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

// Количество машин
#define CARSCNT       (100)

// Количество парковочных мест
#define PARKPLCCNT    (10)

// Семафор для ограничения количества машин на парковке
HANDLE sem;
volatile int park_times[CARSCNT];
// Ключевое слово volatile запрещает компилятору что-либо хранить в регистрах

DWORD WINAPI car_thread(void* param)
{
        int idx = (char*)param - (char*)0;
        int parking_time = park_times[idx];
        long old_val;

        // "захват" семафора - машина заехала на парковку
        WaitForSingleObject(sem, INFINITE);

        printf("Car %d -> parking, time=%d sec\n", idx, parking_time);

        // Стоянка на парковке
        Sleep(parking_time * 1000);

        // "освобождение" семафора - машина уехала с парковки
        ReleaseSemaphore(sem, 1, &old_val);
        printf("parking -> car %d, %ld places are empty\n", idx, old_val + 1);

        return 0;
}
```

```
int main()
{
	HANDLE threads[CARSCNT];
	int i;

	// Создание семафора
	sem = CreateSemaphore(0, PARKPLCCNT, PARKPLCCNT, 0);

	// Задание времени парковки каждой машины
	srand(GetTickCount());
	for (i = 0; i < CARSCNT; i++)
		park_times[i] = 3 + (rand() % 4); // от 3 до 6 секунд

	// Создание потоков-машин
	for (i = 0; i < CARSCNT; i++)
		threads[i] = CreateThread(0, 0, car_thread, (char*) 0 + i, 0, 0);

	// Ожидание потоков машин
	for (i = 0; i < CARSCNT; i++)
	{
		WaitForSingleObject(threads[i], INFINITE);
		CloseHandle(threads[i]);
	}
	CloseHandle(sem);
	return 0;
}
```

Объекты-*события* (*events*) используются для уведомления ожидающих нитей о наступлении какого-либо события. Событие имеет состояние — *сигнальное* или *не сигнальное*. Если событие находится в сигнальном состоянии, то ожидание такого события функцией `WaitForSingleObject()` будет успешно завершено, т. е. функция ожидает перехода события в сигнальное состояние. Перевод события из не сигнального в сигнальное (*установка* события) осуществляется функцией `SetEvent()`.

Различают два вида событий — с ручным и автоматическим *сбросом*. Под сбросом понимают перевод события из сигнального состояния в не сигнальное. Ручной сброс осуществляется функцией `ResetEvent()`. События с ручным сбросом используются для уведомления сразу нескольких нитей.

При использовании события *с автосбросом* уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше. С помощью таких событий можно организовать "критические секции".

Для создания события используется функция `CreateEvent()`. Аргументами функции являются: настройки безопасности события, тип — с ручным или автоматическим сбросом, начальное состояние — сигнальное или нет, и имя события (опционально, может быть NULL).

Для событий существует функция `PulseEvent()` —устанавливает событие в сигнальное состояние, а после возобновления ожидающих это событие нитей (всех при ручном сбросе и только одной при автоматическом), сбрасывает его в не сигнальное. Если ожидающих нитей нет, `PulseEvent()` просто сбрасывает событие.

При работе с объектом синхронизации Событие в Windows может быть вызвана функция `SignalObjectAndWait()`, выполняющая перевод объекта в

сигнальное состояние и переводящая текущий поток в состояние ожидания перевода в сигнальное состояние какого-либо другого объекта. Операция проводится атомарно, поэтому с помощью этой функции легко можно организовать аналог Условных переменных Unix-библиотеки pthread: для этого необходимо использовать один Mutex и один Event.

Следующий пример демонстрирует использование событий с автосбросом для организации доступа к списку заданий. Основная нить создает список заданий для вычисляющих нитей и сами вычисляющие нити. Вычисляющие нити считывают задания (если они есть) по одному и выполняют их. Доступ к списку заданий разграничивается с помощью события — в каждый момент со списком работает только одна нить. Основная нить периодически распечатывает на экран текущее состояние: сколько задач осталось в списке. По завершении всех нитей программа завершается.

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

#define TCNT        (10)

HANDLE threads[TCNT];

// Структура для описания вычислительной задачи
struct task
{
	struct task volatile * next; // Следующий элемент. volatile => хранить не в регистрах
	int from; // Цикл от
	int till; // Цикл до
};

struct task volatile * tasks = 0; // Перечень задач для потоков. volatile => хранить не в
регистрах
volatile int tasks_cnt = 0; // Количество оставшихся задач
HANDLE event; // Событие для синхронизации доступа к задачам


void push_task(int from, int till)
{
	struct task* t;

	// Ожидание срабатывания события
	WaitForSingleObject(event, INFINITE);

	// Создание и вставка элемента в начало списка
	t = (struct task*) malloc(sizeof(struct task));
	t->next = 0;
	t->from = from;
	t->till = till;

	t->next = tasks;
	tasks = t;
	tasks_cnt++;

	// Установка события - теперь со списком tasks может
	// поработать другой поток, а в текущем этого делать больше нельзя
	SetEvent(event);
}
```

```
struct task volatile * pop_task()
{
	struct task volatile * ret = 0;

	// Ожидание срабатывания события
	WaitForSingleObject(event, INFINITE);

	if (tasks) // Если задачи есть
	{  // Вернуть первую из них
		ret = tasks;
		tasks = ret->next;
		ret->next = 0;
		tasks_cnt--;
	}

	// Установка события - теперь со списком tasks может
	// поработать другой поток, а в текущем этого делать больше нельзя
	SetEvent(event);

	return ret;
}

int get_tasks_count()
{
	int ret;

	WaitForSingleObject(event, INFINITE);
	ret = tasks_cnt;
	SetEvent(event);
	return ret;
}


DWORD WINAPI thread_entry(void* param)
{
	int idx = (char*)param - (char*)0;

	while (1)
	{
		// Безопасное извлечение очередной задачи
		struct task volatile *t = pop_task();
		int m = 1, i;

		// Если задачи кончились - выход
		if (!t)
			break;

		// Иначе - выполнение вычислительной задачи
		for (i = t->from; i <= t->till; i++)
			m *= i;

		// Освобождение памяти
		free( (void*) t);
	}

	return 0;
}

void generate_tasks()
{
	int i;

	// Добавление некоторого количества случайных задач
	srand(0);
```

```
        for (i = 0; i < 1000000; i++)
        {
                int from = rand(), till = rand();
                push_task(from, till);
        }
}

int main()
{
        int i;

        // Создание события: с автосбросом, изначально в сигнальном состоянии
        event = CreateEvent(NULL, FALSE, TRUE, NULL);

        printf("Generating tasks...\n");
        // Создание задач
        generate_tasks();

        printf("Creating threads...\n");
        for (i = 0; i < TCNT; i++)
                threads[i] = CreateThread(0, 0, thread_entry, (char*)0 + i, 0, 0);

        while (1)
        {
                // Ожидание завершения всех потоков (1 сек)
                if (WAIT_TIMEOUT == WaitForMultipleObjects(TCNT, threads, TRUE, 1000))
                {  // Потоки еще работают - вывод на экран кол-ва оставшихся задач
                        int left = get_tasks_count();
                        printf("Working: %d tasks left\n", left);
                }
                else
                { // Все потоки завершились, выход
                        break;
                }

        }

        // Закрытие описателей
        for (i = 0; i < TCNT; i++)
                CloseHandle(threads[i]);
        CloseHandle(event);

        return 0;
}
```

Порядок выполнения работы

1. Получить у преподавателя номер задания (*Приложение*).
2. Выполнить задания.
3. Провести тестирование программ.
4. Ответить на контрольные вопросы.

Содержание отчета

Для каждой из разработанных программ укажите:

1. Описание общего алгоритма программы.
2. Таблица, содержащая имена и назначение реализованных функций.
3. Блок-схемы алгоритмов поточных функций.
4. Временна́я диаграмма взаимодействия потоков в моменты синхронизации, показывающая смену состояний потоков (ready/blocked).
5. Результаты тестирования программы.

5.1. Для заданий параллельных алгоритмов ("Параллельные вычисления", "Многопоточная сортировка", "Шахматы") необходимо составить не менее 10 тестирующих наборов входных данных, и провести запуски программы с разным количеством потоков: 1, 2, 3, 4, 5, 8, 10. Тестовые наборы должны отличаться объемом задачи, по возрастанию объема задачи: первый набор необходимо подобрать такой, чтобы программа выполнялась 1-2 секунды, последний набор — чтобы выполнялась не менее 60 секунд. Про каждый тестовый набор следует описать характер входных данных (например, входные значения, или размер массива, или размер шахматной доски и т.д.).

В отчете необходимо привести таблицу с полученными измерениями времени работы алгоритма (в миллисекундах), и в выводах указать информацию о процессоре, на котором проводилось тестирование, а в выводах — объяснить полученный результат.

5.2. Для задания "Философы" необходимо провести испытания с различной общей продолжительностью теста: 200мс, 400мс, 1000мс, 2000мс, 4000мс, 20000мс. Для каждой продолжительности следует задать три разных значения минимального времени "поедания" и "раздумывания".

В отчете небходимо привести таблицу, содержащую количество "квантов" "поедания" и "раздумывания" каждого из философов к каждому из 18 проведенных испытаний. В выводах необходимо оценить "равномерность" распределения процессорного времени между философами.

6. Выводы по работе. Должны содержать объяснения и логические заключения к таблицам, полученным в результате тестирования программ.