Министерство образования и науки РФ

Томский государственный университет систем управления и радиоэлектроники (ТУСУР)

Факультет безопасности (ФБ)

Кафедра комплексной информационной безопасности электронно-вычислительных систем (КИБЭВС)

А.С. Романов

Системное программирование

Тема № 5 - Потоки

Томск 2019

Цель работы

Изучить работу с потоками. Научиться разбивать задачу на части, для последующего их выполнения различными потоками. Познакомиться с основными функциями WinAPI для работы с потоками в Windows и библиотекой Pthread для работы с потоками в Linux.

Краткие теоретические сведения

1. Потоки, общие сведения

Многопоточность является естественным продолжением многозадачности и представляет собой логическое развитие концепции разделения ресурсов. Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений. Потоки могут порождаться во время работы программы, процесса или другого потока. Путь выполнения потока задается при его создании, указанием его стартовой функции, созданный поток начинает выполнять команды этой функции, и завершается когда происходит возврат из функции. Таким образом, несколько порожденных в программе потоков, могут пользоваться глобальными переменными, и любое изменение данных одним потоком, будет доступно и для всех остальных.

Основные отличия процесса от потока заключаются в том, что, каждому процессу соответствует своя независимая от других область памяти, таблица открытых файлов, текущая директория и прочая информация уровня ядра. Потоки же не связаны непосредственно с этими сущностями. У всех потоков принадлежащих данному процессу всё выше перечисленное общее, поскольку принадлежит этому процессу. Кроме того, процесс всегда является сущностью уровня ядра, то есть ядро знает о его существовании, в то время как потоки зачастую является сущностями уровня пользователя и ядро может ничего не знать о ней. В подобных реализациях все данные о потоке хранятся в пользовательской области памяти, и соответственно такие процедуры как порождение или переключение между потоками не требуют обращения к ядру и занимают на порядок меньше времени.

Потоки часто становятся источниками программных ошибок особого рода. Эти ошибки возникают при использовании потоками разделяемых ресурсов системы и являются частным случаем более широкого класса ошибок – ошибок синхронизации. Если задача разделена между независимыми процессами, то доступом к их общим ресурсам управляет операционная система, и вероятность ошибок из-за конфликтов доступа снижается. В пользу потоков можно указать то, что накладные расходы на создание нового потока в многопоточном приложении ниже, чем накладные расходы на создание нового самостоятельного процесса. Уровень контроля над потоками в многопоточном приложении выше, чем уровень контроля приложения над дочерними процессами. Кроме того, многопоточные программы не склонны оставлять за собой процессы-зомби или независимые процессы без процесса-родителя.

2. Модели построения многопоточных приложений

Существует несколько моделей построения многопоточных приложений:

1. **Итеративный параллелизм** используется для реализации нескольких потоков (часто идентичных), каждый из которых содержит циклы. Потоки программы, описываются итеративными функциями и работают совместно над решением одной задачи.

2. **Рекурсивный параллелизм** используется в программах с одной или несколькими рекурсивными процедурами, вызов которых независим. Это технология «разделяй-и-властвуй» или «перебор-с-возвратами».

3. **Производители и потребители** – это парадигма взаимодействующих неравноправных потоков. Одни из потоков «производят» данные, другие их «потребляют». Часто такие потоки организуются в конвейер, через который проходит информация. Каждый поток конвейера потребляет выход своего предшественника и производит входные данные для своего последователя. Другой распространенный способ организации потоков – древовидная структура или сети слияния, на этом основан, в частности, принцип дихотомии.

4. **Клиенты и серверы** – еще один способ взаимодействия неравноправных потоков. Клиентский поток запрашивает сервер и ждет ответа. Серверный поток ожидает запроса от клиента, затем действует в соответствии с поступившим запросом.

5. **Управляющий и рабочие** – модель организации вычислений, при которой существует поток, координирующий работу всех остальных потоков. Как правило, управляющий поток распределяет данные, собирает и анализирует результаты.

6. **Взаимодействующие равные** – модель, в которой исключен не занимающийся непосредственными вычислениями управляющий поток. Распределение работ в таком приложении либо фиксировано заранее, либо динамически определяется во время выполнения. Одним из распространенных способов динамического распределения работ является «портфель задач». Портфель задач, как правило, реализуется с помощью разделяемой переменной, доступ к которой в один момент времени имеет только один процесс.

3. Потоки в Windows

В среде Microsoft Windows создание процесса производится с помощью системной функции ***CreateProcess***. Выполнение этой функции приводит к порождению потока, который называют главным потоком процесса. Главный поток присутствует в любом процессе и часто остается единственным. Остальные потоки могут создаваться в коде главного потока по усмотрению программиста с помощью функции WIN API ***CreateThread***. Таким образом, **процесс – это контейнер для потоков**. Процесс-контейнер содержит как минимум один поток и если потоков в процессе несколько, приложение (процесс) становится многопоточным. Поток, выполнивший функцию CreateThread называют родительским потоком по отношению к созданному им, а созданный поток - дочерним потоком.

Поток в ОС Windows является основным элементом выполнения любого приложения. Рассмотрим основные функции для работы с потоками.

Создание потока

Поток создается с помощью функции CreateThread:

```
HANDLE CreateThread (
  LPSECURITY_ATTRIBUTES lpThreadAttributes,     //атрибуты защиты
  DWORD dwStackSize,                            //размер стека
  LPTHREAD_START_ROUTINE lpStartAddress,        //функция потока
  LPVOID lpThreadParameter,                     //параметр функции потока
  DWORD dwCreationFlags,                        //параметр запуска потока
  LPDWORD lpThreadID                            //идентификатор потока
);
```

Данная функция создает дочерний поток, устанавливает его характеристики (атрибуты защиты, размер стека), указывает на код функции потока и в зависимости от значения параметра dwCreationFlags либо производит его запуск, либо приостанавливает до специального распоряжения. Под функцией потока понимают описанную в программе функцию, которая содержит код, выполнение которого должно осуществляться в рамках данного потока.

Параметры функции CreateThread (in - входные, out- выходные):

lpThreadAttributes – входной параметр, указатель на структуру SECURITY_ATTRIBUTES, определяющую атрибуты защиты для создаваемого потока. Рекомендуется задавать значение NULL, разрешающее использовать любые функции для управления данным потоком.

dwStackSize – входной параметр, размер стека потока в байтах. Для использования размера стека родительского потока используйте значение 0.

lpStartAddress – входной параметр, указатель на функцию программы, которую будет выполнять поток (можно задать просто имя этой функции). Функция потока принимает в качестве параметра единственный параметр и возвращает код завершения типа DWORD. Поток может представить этот параметр как значение типа DWORD или как указатель.

lpThreadParameter – входной параметр, указатель, который передается потоку в качестве параметра и обычно интерпретируется им, как указатель на некоторую структуру. При отсутствии параметров следует указать NULL.

dwCreationFlags – входной параметр запуска потока. Нулевое значение параметра означает, что поток готов к немедленному выполнению. Если в качестве значение этого параметра указать константу CREATE_SUSPENDED, то новый поток будет находиться в состоянии ожидания до тех пор, пока не будет вызвана функция ResumeThread.

lpIDThread – выходной параметр, указатель на переменную типа DWORD, в которую будет помещен идентификатор (системный номер) созданного потока.

Возвращаемое значение: в случае успеха функция CreateThread возвращает дескриптор созданного потока (тип handle), который необходим для выполнения различных операций над потоком. При ошибке функция возвращает значение NULL.

Завершение потока

Поток может завершиться по собственной инициативе или по инициативе родительского потока, формируя при этом код завершения.

В первом случае поток завершается при выполнении оператора возврата из функции потока (return) или с помощью функции ***ExitThread***:

```
VOID ExitThread(
   DWORD  dwExitCode    // код завершения потока
);
```

В качестве единственного параметра этой функции задается код завершения потока.

Во втором случае применяется функция ***TerminateThread***, с помощью которой родительский поток может принудительно завершить выполнение своего дочернего потока:

```
BOOL TerminateThread(
 HANDLE  hThread,       // дескриптор потока
 DWORD  dwExitCode      // код завершения потока
);
```

Значение дескриптора потока определяется по значению, возвращаемому функцией CreateThread при создании потока.

Все потоки, созданные в рамках какого-либо процесса, автоматически завершают свое выполнение при завершении работы процесса (т.е. выполнении функции ExitProcess).

Проверка состояния потока

Для получения кода завершения ранее запущенного дочернего потока используется функция ***GetExitCodeThread***:

```
BOOL GetExitCodeThread(
 HANDLE  hThread,       // дескриптор потока
 LPDWORD lpdwExitCode   // адрес для приема кода завершения
);
```

Если поток, для которого вызвана данная функция, все еще работает, вместо кода завершения возвращается значение STILL_ACTIVE.

Ожидание завершения выполнения потока

Для перевода родительского потока в режим ожидания (блокирования) до момента завершения нескольких запущенных им потоков, целесообразно использовать функцию ***WaitForMultipleObjects***:

```
DWORD WaitForMultipleObjects (
  DWORD cObjects,             //количество ожидаемых потоков
  CONST HANDLE *lphObjects,   //адрес массива дескрипторов потоков
  BOOL fWaitAll,              //тип ожидания
  DWORD dwTimeout             //время ожидания в мс
);
```

Например, если запущено три потока и их дескрипторы представлены в виде массива HANDLE hThread[3], то ожидание до тех пор, пока все три потока не завершатся, можно организовать следующим образом:

```
 WaitForMultipleObjects(3,hThreads,TRUE,INFINITE);
```

Тип ожидания TRUE означает ожидание завершения всех потоков (FALSE – хотя бы одного из потоков). Время ожидания INFINITE означает бесконечное ожидание до наступления требуемого события.

Диспетчеризация и управление приоритетами потоков

В ОС Windows используется принцип приоритетной диспетчеризации потоков. Это означает, что кванты процессорного времени чаще выделяются потокам с более высоким приоритетом. Значения приоритета устанавливаются в диапазоне от 1 до 31. Существуют 4 уровня приоритетов, которые назначаются процессам при их создании (в зависимости от типа процесса):

IDLE_PRIORITY_CLASS=4 - низкоприоритетные процессы;
NORMAL_PRIORITY_CLASS=9 - обычные процессы;
HIGH_PRIORITY_CLASS=13 - высокоприоритетные процессы;
REALTIME_PRIORITY_CLASS=24 - процессы реального времени;

Потоки первоначально получают такое же значение приоритета, как и у процесса. Обычные пользовательские процессы (и их потоки) по умолчанию получают значение приоритета 9 , что соответствует классу NORMAL_PRIORITY_CLASS. Операционная система может автоматически изменять приоритет потоков в зависимости от их текущего состояния: увеличивать, когда поток взаимодействует с пользователем или снижать, когда поток переходит в состояние ожидания.

С помощью функции ***SetThreadPriority*** можно изменить относительный приоритет потока, но только в рамках установленного класса:

```
BOOL SetThreadPriority (
```

```
HANDLE hThread,          //дескриптор потока
int  nPriority           //новый уровень приоритета потока
);
```

Новый уровень приоритета потока задается с помощью специальных констант, которые устанавливают величину изменения приоритета потока относительно приоритета процесса:

THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	–1
THREAD_PRIORITY_LOWEST	–2
THREAD_PRIORITY_TIME_CRITICAL	=15 (или =31)

Последняя из указанных констант THREAD_PRIORITY_TIME_CRITICAL позволяет установить абсолютное значение приоритета потока, равное 31 для процессов класса REALTIME_PRIORITY_CLASS или 15 для остальных классов.

В любой момент времени можно определить текущее значение приоритета потока с дескриптором hThread с помощью функции ***GetThreadPriority***:

```
int GetThreadPriority (
  HANDLE hThread              // handle потока
);
```

Приведем простой пример работы с потоками в ОС Windows (см. Листинг 1).

Дана последовательность натуральных чисел, хранящихся в массиве a_0, …, a_{99}. Создадим многопоточное приложение для поиска суммы квадратов элементов этого вектора. Разобьем последовательность чисел на четыре части и создадим четыре потока, каждый из которых будет вычислять суммы квадратов элементов в отдельной части последовательности. Главный поток создаст дочерние потоки, соберет данные и вычислит окончательный результат, после того, как отработают четыре дочерних потока.

```
#include <stdio.h>
#include <conio.h>
#include <windows.h>
const int n = 4;
int a[100];
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
  int num,sum = 0,i;
  num = 25*(*((int *)pvParam));
  for(i=num;i<num+25;i++)
  {
  sum += a[i]*a[i];
  *(int*)pvParam = sum;
  DWORD dwResult = 0;
  return dwResult;
}
int main(int argc, char** argv)
{
  int x[n];
  int i,rez = 0;
  DWORD dwThreadId[n],dw;
  HANDLE hThread[n];
  for (i=0;i<100;i++)
   a[i] = i;
  //создание n дочерних потоков
```

```
  for (i=0;i<n;i++)
  {
   x[i] = i;
   hThread[i] = CreateThread(NULL,0,ThreadFunc,(PVOID)&x[i], 0, &dwThreadId[i]);
   if(!hThread)
   printf("main process: thread %d not execute!",i);
  }
  // ожидание завершения n потоков
  dw = WaitForMultipleObjects(n,hThread,TRUE,INFINITE);
  for(i=0;i<n;i++)
   rez+=x[i];
  printf("\nСумма квадратов = %d",rez);
  getch();
  return 0;
}
```

Листинг 1 – Пример работы с потоками с помощью функций WinAPI

4. Потоки в Linux

В Linux API для управления потоками, их синхронизации и планирования определяет стандарт POSIX.1c, Threads extensions (IEEE Std. 1003.1c-1995). Библиотеки, реализующие этот стандарт, называются Pthreads, а функции имеют приставку «pthread_».

В Linux каждый поток на самом деле является процессом, и для того, чтобы создать новый поток, нужно создать новый процесс. В многопоточных приложениях Linux для создания дополнительных потоков используются процессы особого типа. Эти процессы (lightweight processes) представляют собой обычные дочерние процессы главного процесса, но они разделяют с главным процессом адресное пространство, файловые дескрипторы и обработчики сигналов. Прилагательное «легкий» в названии процессов-потоков вполне оправдано. Поскольку этим процессам не нужно создавать собственную копию адресного пространства (и других ресурсов) своего процесса-родителя, создание нового легкого процесса требует значительно меньших затрат, чем создание полновесного дочернего процесса.

Напомним, что в Linux у каждого процесса есть идентификатор. Есть он и у процессов-потоков. Однако спецификация POSIX 1003.1c требует, чтобы все потоки многопоточного приложения имели один идентификатор. Вызвано это требование тем, что для многих функций системы многопоточное приложение должно представляться как один процесс с одним идентификатором. Для решения проблемы единого идентификатора процессы многопоточного приложения группируются в группы потоков (thread groups). Группе присваивается идентификатор, соответствующий идентификатору первого процесса многопоточного приложения. Функция ***getpid()***, возвращает значение идентификатора группы потока, независимо от того, из какого потока она вызвана. Функции ***kill() waitpid()*** и им подобные по умолчанию также используют идентификаторы групп потоков, а не отдельных процессов.

Потоки создаются функцией ***pthread_create()***, определенной в заголовочном файле pthread.h:

```
#include <pthread.h>
int pthread_create
(
  pthread_t * thread, // указатель на идентификатор создаваемого потока
  pthread_attr_t *attr, // указатель на атрибуты потока
  void *(*start_routine)(void *), // адрес функции потока
  void *arg // значение, возвращаемого функцией потока
);
```

Первый параметр этой функции представляет собой указатель на переменную типа pthread_t, которая служит идентификатором создаваемого потока. Второй параметр, указатель на переменную типа pthread_attr_t, используется для передачи атрибутов потока. Третьим параметром функции pthread_create() должен быть адрес функции потока. Эта функция играет для потока ту же роль, что функция main() для главной программы. Четвертый параметр функции pthread_create() имеет тип void *. Этот параметр может использоваться для передачи значения, возвращаемого функцией потока. Вскоре после вызова pthread_create() функция потока будет запущена на выполнение параллельно с другими потоками программы. Следует учитывать, что перед тем как запустить новую функцию потока, нужно выполнить некоторые подготовительные действия, а поток-родитель между тем продолжает выполняться – это занимает некоторое время. Если в ходе создания потока возникла ошибка, функция pthread_create() возвращает ненулевое значение, соответствующее номеру ошибки.

Функция потока должна иметь заголовок вида:

```
void * func_name(void * arg)
```

Аргумент arg - это указатель, который передается в последнем параметре функции pthread_create(). Функция потока может вернуть значение, которое затем будет проанализировано заинтересованным потоком, но это не обязательно. Завершение функции потока происходит если:

1. функция потока вызвала функцию pthread_exit();
2. функция потока достигла точки выхода;
3. поток был досрочно завершен другим потоком.

Функция ***pthread_exit()*** представляет собой потоковый аналог функции ***_exit()*** и определена следующим образом:

```
# include <pthread.h>
void pthread_exit(
  void *value // значение
);
```

Аргумент value является указателем на данные, возвращаемые потоком, этот указатель может быть получен родительским потоком при помощи функции pthread_join(). Реально при вызове этой функции поток из нее просто не возвращается. Стоит помнить, что функция ***exit()*** по-прежнему завершает процесс, то есть, в том числе уничтожает все потоки.

Для того, чтобы получить значение, возвращенное функцией потока, нужно воспользоваться функцией ***pthread_join()***:

```
# include <pthread.h>
int pthread_join(
  pthread_t threadid,   // идентификатор потока
  void *value                 // значение
);
```

У этой функции два параметра. Первый параметр – это идентификатор потока, второй параметр имеет тип указатель на нетипизированный указатель. В этом параметре функция возвращает значение, возвращенное функцией потока – таким образом можно организовать передачу данных между потоками. Однако основная задача функции pthread_join() заключается в синхронизации потоков. Функция pthread_join() переводит поток, из которого

она была вызвана, в состояние ожидания до тех пор, пока не завершится поток, определяемый идентификатором, переданным в качестве аргумента. Если в момент вызова pthread_join() ожидаемый поток уже завершился, функция вернет управление немедленно. Функцию pthread_join() можно рассматривать как эквивалент ***waitpid()*** для потоков. Попытка выполнить более одного вызова pthread_join() из разных потоков для одного и того же потока приведет к ошибке. При успешном завершении функция возвращает 0. В случае ошибки возвращается ненулевое значение.

Рассмотрим пример программы, реализующей все вышеописанное (см. Листинг 2). Программа создает процесс и потоки, которые печатают идентификатор процесса и потока.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void put_msg( char *title, struct timeval *tv ) {
   printf( "%02u:%06lu : %s\t: pid=%lu, tid=%lu\n",
           ( tv->tv_sec % 60 ), tv->tv_usec, title, getpid(), pthread_self() );
}
void *test( void *in ) {
   struct timeval *tv = (struct timeval*)in;
   gettimeofday( tv, NULL );
   put_msg( "pthread started", tv );
   sleep( 5 );
   gettimeofday( tv, NULL );
   put_msg( "pthread finished", tv );
   return NULL;
}
#define TCNT 5
static pthread_t tid[ TCNT ];
int main( int argc, char **argv, char **envp ) {
   pthread_t tid[ TCNT ];
   struct timeval tm;.
   int i;.
   gettimeofday( &tm, NULL );
   put_msg( "main started", &tm );
   for( i = 0; i < TCNT; i++ ) {
      int status = pthread_create( &tid[ i ], NULL, test, (void*)&tm );
         if( status != 0 ) perror( "pthread_create" ), exit( EXIT_FAILURE );
   };
   for( i = 0; i < TCNT; i++ )
      pthread_join( tid[ i ], NULL ); // это обычная техника ожидания!
   gettimeofday( &tm, NULL );
   put_msg( "main finished", &tm );
   return( EXIT_SUCCESS );
}
```

Листинг 2 – Пример использования функций pthread_create() и pthread_join

При компиляции надо знать, что хоть функции работы с потоками и описаны в файле включения pthread.h, на самом деле они находятся в библиотеке. Библиотеку libgcc.a рекомендуется скопировать в текущий каталог. В строку компиляции нужно дописать ключ «-lpthread». Откпомилируем пример и выполним его:

```
[root@srv ~]# gcc pthreadexample.c -lpthread -o pthreadexample
[root@srv ~]# ./pthreadexample
50:259188 : main started      : pid=14333, tid=3079214784
50:259362 : pthread started   : pid=14333, tid=3079211888
50:259395 : pthread started   : pid=14333, tid=3068722032
```

```
50:259403 : pthread started   : pid=14333, tid=3058232176
50:259453 : pthread started   : pid=14333, tid=3047742320
50:259466 : pthread started   : pid=14333, tid=3037252464
55:259501 : pthread finished  : pid=14333, tid=3079211888
55:259501 : pthread finished  : pid=14333, tid=3068722032
55:259525 : pthread finished  : pid=14333, tid=3058232176
55:259532 : pthread finished  : pid=14333, tid=3047742320
55:259936 : main finished     : pid=14333, tid=3079214784
```

В случае если нас чем-то не устраивает возврат значения через pthread_join(), например, нам необходимо получить данные в нескольких нитях, то следует воспользоваться каким либо другим механизмом, например, можно организовать очередь возвращаемых значений, или возвращать значение в структуре указатель на которую передают в качестве параметра нити. То есть использование pthread_join() это вопрос удобства, а не догма, в отличие от случая пары fork() и wait() для процессов.

Если предполагается использовать другой механизм возврата или возвращаемое значение не важно - можно отсоединить поток (detach), дав понять ОС, что необходимо освободить ресурсы, связанные с потоком сразу по завершению функции потока. Сделать это можно несколькими способами. Во-первых, можно сразу создать поток отсоединенным, задав соответствующий объект атрибутов при вызове pthread_create(). Во-вторых, любой поток можно отсоединить, вызвав в любой момент его жизни функцию ***pthread_detach()***:

```
#include <pthread.h>
int pthread_detach(
  pthread_t thread // идентификатор нити
);
```

Функция имеет один параметр - идентификатор потока. При этом поток может отсоединить сам себя, получив свой идентификатор при помощи функции ***pthread_self()***:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Следует подчеркнуть, что отсоединение потока никоим образом не влияет на процесс его выполнения, а просто помечает поток как готовый по своему завершению к освобождению ресурсов: стека, памяти, в которую сохраняется контекст потока, данные специфичные для потока и проч. Сюда не входят ресурсы выделяемые явно, например, память, выделяемая через malloc(), или открываемые файлы. Подобные ресурсы должен отслеживать программист и явно освобождать их сам.

Сделать поток «отдельным» можно и на этапе его создания, с помощью дополнительного атрибута ***DETACHED***. Для того чтобы назначить потоку дополнительные атрибуты, нужно сначала создать объект, содержащий набор атрибутов. Этот объект создается функцией ***pthread_attr_init()***:

```
#include <pthread.h>
int pthread_attr_init(
  pthread_attr_t *attr  // указатель на набор аргументов
);
```

Единственный аргумент этой функции – указатель на переменную типа pthread_attr_t, которая служит идентификатором набора атрибутов. Функция pthread_attr_init() инициализирует набор атрибутов потока значениями, заданными по умолчанию. Для

добавления атрибутов в набор используются специальные функции с именами ***pthread_attr_set<имя_атрибута>***. Например, для того, чтобы добавить атрибут «отделенности» используется функция ***pthread_attr_setdetachstate()***:

```
#include <pthread.h>
int pthread_attr_setdetachstate(
  pthread_attr_t *attr,
  int detachstate
);
```

Первым аргументом этой функции должен быть адрес объекта набора атрибутов, а вторым аргументом – константа, определяющая значение атрибута. Константа PTHREAD_CREATE_DETACHED указывает, что создаваемый поток должен быть отделенным, а константа PTHREAD_CREATE_JOINABLE определяет создание присоединяемого (joinable) потока, который может быть синхронизирован функций pthread_join(). После добавления необходимых значений в набор атрибутов потока, необходимо вызвать функцию pthread_create() и передать набор атрибутов потока вторым аргументом.

Точно так же, как при управлении процессами, иногда возникает необходимость досрочно завершить процесс, многопоточной программе может понадобиться досрочно завершить один из потоков. Для досрочного завершения потока можно воспользоваться функцией ***pthread_cancel()***:

```
# include <pthread.h>
int pthread_cancel(
  pthread_t threaded    // идентификатор потока
);
```

Единственным аргументом этой функции является идентификатор потока. Функция pthread_cancel() возвращает 0 в случае успеха и ненулевое значение в случае ошибки. Несмотря на то, что pthread_cancel() может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Дело в том, что поток может не только самостоятельно выбрать порядок завершения в ответ на вызов pthread_cancel(), но и вовсе игнорировать этот вызов. Вызов функции pthread_cancel() следует рассматривать как запрос на выполнение досрочного завершения потока.

Функция ***pthread_setcancelstate()*** определяет, будет ли поток реагировать на обращение к нему с помощью pthread_cancel(), или не будет.

```
#include <pthread.h>
int pthread_setcancelstate(
  int state,           // новое значение
  int *oldstate   // указатель на старое значение
);
```

Функция pthread_setcancelstate() имеет два параметра - параметр state типа int и параметр oldstate типа «указатель на int». В первом параметре передается новое значение, указывающее, как поток должен реагировать на запрос pthread_cancel(), а в переменную, адрес которой был передан во втором параметре, функция записывает прежнее значение. State может иметь два значения PTHREAD_CANCEL_DISABLE (запретить досрочное завершение потока) и PTHREAD_CANCEL_ENABLE (разрешить досрочное завершение потока). Если прежнее значение не интересует, во втором параметре можно передать NULL. Функция возвращает 0 в случае успеха и ненулевое значение в случае ошибки.

Чаще всего функция pthread_setcancelstate() используется для временного запрета завершения потока путем ограждения фрагмента кода, во время выполнение которого завершать поток крайне нежелательно:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
... //Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы, выполнение запроса будет отложено до тех пор, пока функция pthread_setcancelstate() не будет вызвана с разрешающим аргументом. Что именно произойдет дальше, зависит от более тонких настроек потока.

Если досрочное завершение разрешено, поток, получивший запрос на досрочное завершение, может завершить работу не сразу. Если поток находится в режиме отложенного досрочного завершения (именно этот режим установлен по умолчанию), он выполнит запрос на досрочное завершение, только достигнув одной из точек отмены. В соответствии со стандартом POSIX, точками отмены являются вызовы многих обычных функций, например open(), pause() и write(). Установить точку отмены вручную можно с помощью функции ***pthread_testcancel()***:

```
#include <pthread.h>
void pthread_testcancel(void);
```

В частности, установить явную точку отмены может потребоваться при использовании функции printf(), т.к. при её вызове поток завершается, но pthread_join() не возвращает управление.

Впрочем, можно выполнить досрочное завершение потока, не дожидаясь точек останова. Для этого необходимо перевести поток в режим немедленного завершения, что делается с помощью вызова

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

В этом случае беспокоиться о точках останова уже не нужно. Вызов

```
pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

снова переводит поток в режим отложенного досрочного завершения.

Рассмотрим пример программы (см. Листинг 3).

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
int i = 0;
void * thread_func(void *arg)
{
  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  for (i=0; i < 4; i++) {
    sleep(1);
    printf("I'm still running!\n");
  }
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_testcancel();
  printf("YOU WILL NOT STOP ME!!!\n");
```

```
}
int main(int argc, char * argv[])
{
  pthread_t thread;
  pthread_create(&thread, NULL, thread_func, NULL);
  while (i < 1) sleep(1);
  pthread_cancel(thread);
  printf("Requested to cancel the thread\n");
  pthread_join(thread, NULL);
  printf("The thread is stopped.\n");
  return EXIT_SUCCESS;
}
```

Листинг 3 – Пример использования функции pthread_setcancelstate()

В самом начале функции потока thread_func() мы запрещаем досрочное завершение потока, затем выводим четыре тестовых сообщения с интервалом в одну секунду, после чего разрешаем досрочное завершение. Далее, с помощью функции pthread_testcancel(), создаем точку отмены (cancellation point) потока. Если досрочное завершение потока было затребовано, в этот момент поток должен завершиться. Затем мы выводим еще одно диагностическое сообщение, которое пользователь не должен видеть, если программа сработает правильно.

В главной функции программы мы создаем поток, затем дожидаемся, пока значение глобальной переменной i станет больше нуля (это гарантирует нам, что поток уже запретил досрочное завершение) и вызываем функцию pthread_cancel(). После этого мы переходим к ожиданию завершения потока с помощью pthread_join(). Если вы скомпилируете и запустите программу, то увидите, что поток распечатает четыре тестовых сообщения I'm still running! (после первого сообщения главная функция программы выдаст запрос на завершение потока).

Поскольку поток завершится досрочно, последнего тестового сообщения вы не увидите.

Предположим, что поток выделяет блок динамической памяти и затем внезапно завершается по требованию другого потока. Если бы поток был самостоятельным процессом, ничего особенно неприятного не случилось бы, так как система сама убрала бы за ним мусор. В случае же процесса-потока блок останется невысвобожденым, что в конечном итоге приведет к серьезным утечкам памяти.

Таким образом, для эффективного управления завершением потоков необходим еще и механизм, оповещающий поток о досрочном завершении. Если нужно выполнять какие-то специальные действия в момент завершения потока (нормального или досрочного), мы устанавливаем функцию-обработчик, которая будет вызвана перед тем, как поток завершит свою работу.

Для установки обработчика завершения потока применяется макрос ***pthread_cleanup_push()***:

```
#include <pthread.h>
void pthread_cleanup_push(
  void (*routine)(void *),    // адрес функции-обработчика
  void *arg                   // аргументы
);
```

У макроса два аргумента. В первом аргументе ему должен быть передан адрес функции-обработчика завершения потока, а во втором – нетипизированный указатель, который будет передан как аргумент при вызове функции-обработчика. Этот указатель может указывать на что угодно, мы сами решаем, какие данные должны быть переданы

обработчику завершения потока. Макрос pthread_cleanup_push() помещает переданные ему адрес функции-обработчика и указатель в специальный стек. Потоку можно назначить произвольное число функций-обработчиков завершения. Поскольку в стек записывается не только адрес функции, но и ее аргумент, мы можем назначить один и тот же обработчик с несколькими разными аргументами.

Извлечение обработчиков из стека и их выполнение может производиться либо явно, либо автоматически. Автоматически обработчики завершения потока выполняются при вызове потоком функции pthread_exit(), завершающей работу потока, а также при выполнении потоком запроса на досрочное завершение. Явным образом обработчики завершения потока извлекаются из стека с помощью макроса ***pthread_cleanup_pop()***:

```
#include <pthread.h>
void pthread_cleanup_pop(
  int execute
);
```

Аргумент макроса pthread_cleanup_pop() позволяет указать, следует ли выполнять функцию-обработчик, или требуется только удалить ее из стека.

Следует помнить, что макрос pthread_cleanup_pop() должен быть вызван столько же раз, сколько и макрос pthread_cleanup_push().

Рассмотрим методы назначения и выполнения обработчиков завершения потока на простом примере (см. Листинг 4):

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
void exit_func(void * arg)
{ free(arg);
printf("Freed the allocated memory.\n");
}
void * thread_func(void *arg)
{ int i;
void * mem;
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
mem = malloc(1024);
printf("Allocated some memory.\n");
pthread_cleanup_push(exit_func, mem);
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
for (i = 0; i < 4; i++) {
sleep(1);
printf("I'm still running!!!\n");
}
pthread_cleanup_pop(1);
}
int main(int argc, char * argv[])
{ pthread_t
thread;
pthread_create(&thread, NULL, thread_func, NULL);
pthread_cancel(thread);
pthread_join(thread, NULL);
printf("Done.\n");
return EXIT_SUCCESS;
}
```

Листинг 4 – Пример использования обработчиков завершения потоков

Поток начинает работу с того, что запрещает свое досрочное завершение. Далее поток выделяет блок памяти. Для того чтобы избежать утечек памяти используется функция-обработчик завершения потока exit_func(), которая добавляется в стек обработчиков завершения потока с помощью макроса pthread_cleanup_push(). Вторым параметром функции устанавливается указатель на блок памяти. Функция exit_func() высвобождает блок памяти с помощью функции ***free()*** и выводит диагностическое сообщение. После установки обработчика завершения потока поток разрешает досрочное завершение. Однако если бы поток завершился после выделения блока памяти, но до назначения функции-обработчика, выделенный блок не был бы удален. Перед выходом из функции потока вызывается макрос pthread_cleanup_pop().

Задание

1. Изучить краткие теоретические сведения и лекционный материал по теме практического задания.
2. Реализовать приведенные примеры программы и продемонстрировать их работу..
3. Получить индивидуальное задание у преподавателя. Выбрать модель многопоточного приложения, наиболее точно отвечающую специфике задачи. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Желательно избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Если же избежать этого невозможно, необходимо использовать алгоритмы с активным ожиданием или неделимые операции.
4. Реализовать алгоритм с применением функций библиотеки Pthread.
5. Реализовать алгоритм с применением функций WinAPI.
6. Сравнить возможности обоих подходов, сделать выводы.

Варианты заданий

1. Изучить краткие теоретические сведения, материалы лекций по теме практического занятия и приведенные выше примеры программ.
2. Используя Docker и соответствующий образ подготовить среду для разработки.
3. Реализовать программы, соответствующие вашему варианту задания на языке C++ для Linux (ваш вариант образа). Выбрать модель многопоточного приложения, наиболее точно отвечающую специфике задачи. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Желательно избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Если же избежать этого невозможно, необходимо использовать алгоритмы с активным ожиданием или неделимые операции.
4. Для вашего варианта языка программирования изучить встроенные высокоуровневые возможности языка программирования для работы с потоками и реализовать программы, соответствующие вашему варианту задания.
5. Сравнить возможности обоих подходов, сделать выводы.
6. Написать отчет и защитить у преподавателя.

Варианты индивидуальных заданий

Варианты заданий (образ ОС для Docker)

1. CentOS

2. Ubuntu
3. Debian
4. Alpine
5. Amazon Linux

Варианты заданий (Язык программирования)
1. c++ (gcc)
2. golang
3. python
4. PHP
5. Java

Варианты заданий (программа)

1. Написать две программы. Первая реализует алгоритм поиска простых чисел в некотором интервале. Вторая - разбивает заданный интервал на диапазоны, осуществляет поиск простых чисел в каждом из интервалов в отдельном процессе, выводит общий результат.

2. Написать две программы. Первая реализует алгоритм поиска указанной подстроки в строке. Вторая - разбивает входной файл на фрагменты, осуществляет поиск подстроки в каждом из фрагментов в отдельном процессе, выводит общий результат.

3. Написать две программы. Первая реализует алгоритм умножения двух векторов произвольной длины. Вторая - умножает матрицу произвольного порядка на вектор, при этом умножение каждой строки на вектор производить в отдельном процессе.

4. Написать две программы, одна из которых осуществляет проверку доступности узла сети на доступность (команда ping), а вторая - осуществляет проверку доступности диапазона IP-адресов сети класса «С» (254 адреса, маска 255.255.255.0), разделенного на несколько поддиапазонов, с помощью первой программы.

5. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали, при выпадении шарика за нижнюю границу допустимой области шарик исчезает. Напишите программу изменения координат одного шарика и программу, создающую для каждого из заданного количества шариков порожденный процесс изменения их координат.

6. Противостояние двух команд – каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Напишите программу, которая бы осуществляла уменьшение числа бойцов в противостоящей команде и увеличение в своей на случайную величину и программу, в которой бы в родительском процессе запускались порожденные процессы, реализующие деятельность одной команды.

7. Написать две программы. Первая - копирует файл. Вторая - копирует содержимое директории пофайлово с помощью первой программы в отдельных процессах.

8. Написать две программы.Первая – вычисляет контрольную сумму файла. Вторая – вычисляет контрольную сумму всех файлов в директории, при этом обработка каждого отдельного файла осуществляется с помощью первой программы в отдельном процессе.

9. Написать две программы. Первая – вычисляет математическое ожидание и дисперсию в массиве данных. Вторая – вычисляет математическое ожидание и дисперсию в нескольких массивах данных, при этом обработка каждого массива осуществляется отдельно с помощью первой программы в новом процессе.

10. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм символов (аа, аб, ав, … яэ, яю, яя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм символов, путем вызова первой программы для отдельных фрагментов текста.

11. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм символов (ааа, ааб, аав, … яяэ, яяю, яяя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм символов, путем вызова первой программы для отдельных фрагментов текста.

12. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет», «вычисляет частоты», «частоты встречаемости», «встречаемости в» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм слов, путем вызова первой программы для отдельных фрагментов текста.

13. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет частоты», «вычисляет частоты встречаемости», «частоты встречаемости в», «встречаемости в тексте» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм слов, путем вызова первой программы для отдельных фрагментов текста.

14. Медведь и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Медведь потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Написать две программы. Первая - реализует работу одной пчелы. Вторая - осуществляет работу медведя, при этом у для заданного количества пчел вызывается отдельный процесс работы одной пчелы.

15. Авиаразведка - создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (процессов), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фиксирует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. Реализуйте соответствующие программы, используя механизмы процессов.

16. Бег с препятствиями - создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (процессы) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. Реализуйте соответствующие программы, используя механизмы процессов.

Контрольные вопросы

1. Дайте определения понятиям «процесс» и «поток». Чем они отличаются и какие сходства имеют?
2. Какие модели построения многопоточных приложений вам известны?
3. Что такое главный, родительский и дочерний потоки?
4. Поясните параметры функции CreateThread.
5. Что такое функция потока и как передаются параметры функции потока?
6. Что такое дескриптор потока и как он используется при управлении потоком?
7. Перечислите и поясните способы завершения выполнения потока.
8. Как формируются и регулируются приоритеты процессов и потоков?
9. Какими способами родительский поток может получить информацию о текущем состоянии дочернего потока (завершен или еще выполняется)?

10. С какой целью при создании многопоточных приложений используется функция WaitForMultipleObject и каковы ее параметры?

11. Что такое многопоточность, чем многопоточность отличается от многозадачности?

12. В чем отличие потоков в в ОС семейства Unix/Linux от потоков в ОС Windows? Что такое «облегченный процесс»?

13. Перечислите и поясните основные модели создания многопоточного приложения?

14. Каким образом и с помощью чего создаются потоки в операционных системах семейства Unix/Linux? Поясните параметры функции pthread_create.

15. Каким образом в ОС семейства Unix/Linux идентифицируются потоки одного процесса? Что такое группы потоков?

16. В каких случаях происходит завершение функции потока в ОС семейства Unix/Linux?

17. Как получить значение, возвращенное функцией потока в ОС семейства Unix/Linux?

18. Что такое «отсоединенный поток»? Как можно «отсоединить» поток?

19. Каким образом можно досрочно завершить поток в ОС семейства Unix/Linux?

20. Что произойдет, если запрос на досрочное завершение потока поступит в тот момент, когда поток игнорирует эти запросы?

21. Что такое «точка отмены» и как её можно установить?

22. Каким образом можно оповестить поток о досрочном завершении? Каким образом работает обработчик завершения потока?

