**Лабораторная работа**

**Задание**

Задан массив А, содержащий натуральные числа. Составить девять новых массивов той же размерности, записав в первый массив остатки от деления A[i]-х на 2, во второй массив остатки от деления A[i]-х на 3, ... в последний массив остатки от деления A[i]-х на 10.

**Тема:** Программирование простых многопоточных приложений с использованием библиотеки Pthread.

**Цели:** Изучить основы программирования параллельных приложений в Pthread. Ознакомиться с основами создания многопоточных параллельных программ.

**Теория: Создание и основы управления потоками**

Распараллеливание императивных программ происходит с помощью применения нескольких процессов или потоков. Процесс – выполняемый программный код в совокупности с личным адресным пространством, текущими значениями счетчика команд, регистров и переменных.

Потоки предоставляют возможность проведения параллельных или псевдопараллельных, в случае одного процессора, вычислений на одной вычислительной машине. Потоки могут порождаться во время работы программы, процесса или другого потока. Различные потоки имеют различные пути выполнения, но при этом пользуются общей памятью. Таким образом, несколько порожденных в программе потоков, могут пользоваться глобальными переменными, и любое изменение данных одним потоком, будет доступно и для всех остальных. 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.

Когда программа начинает выполняться, все вычисления проходят в главном потоке. Можно сказать, что функция main – это стартовая функция главного потока. При завершении главного потока, все потоки, порожденные во время его выполнения, завершаются также. Созданный поток может быть связанным (по умолчанию) или отсоединенным. Связанный поток завершается, в случае окончания работы его родительского потока. Отсоединенный поток будет работать и в случае завершения его потока-родителя, но не тогда, когда родителем является главный поток. По завершении отсоединенный поток освобождает используемые им ресурсы и перестает существовать, связанный поток продолжает присутствовать в неактивном состоянии. И отсоединенные и связанные потоки обладают общей памятью. Процессы, в отличие от потоков, выполняются каждый в индивидуальном адресном пространстве.

Рассмотрим функции для работы с потоками.

**int pthread\_create (pthread\_t \*thread, const pthread\_attr\_t \*attr,**

**имя стартовой функции, void \*arg)**

Функция создает поток и возвращает 0 в случае успеха. Переменная thread – это идентификатор созданного потока. Для совершения многих действий над потоком, необходим его идентификатор. Класс pthread\_tявляется непрозрачным, переменная этого класса не может быть указана явно, также не может быть получено ее значение. Параметр attr содержит набор атрибутов потока, более подробно об атрибутах будет рассказано ниже. Третий параметр – имя стартовой функции, в программе стартовая функция должна иметь следующий заголовок: void \*имя\_функции (void \* параметр). Последний параметр arg содержит те данные, которые будут переданы в функцию, т.е. определяет входной параметр стартовой функции.

**void pthread\_exit (void \*retval)**

Функция завершает текущий поток и записывает в retval данные, которые будут возвращены стартовой функцией потока.

**pthread\_t pthread\_self ()**

Возвращает идентификатор текущего потока. Главный поток также может получить свой идентификатор.

**int pthread\_equal (pthread\_t thread1, pthread\_t thread2)**

Если оба идентификатора определяют один и тот же поток, функция возвращает единицу, иначе возвращает ноль.

**int pthread\_join (pthread\_t thread, void \*\*thread\_return)**

Текущий поток ожидает завершения работы потока с идентификатором thread, те данные, что возвращает стартовая функция ожидаемого потока, сохраняются в thread\_return. Когда ожидание завершено, поток с идентификатором thread отсоединяется и освобождаются выделенные под него ресурсы. Функция возвращает 0 в случае успеха. Необходимо отметить, что если поток thread уже завершился к моменту вызова функции, это не приводит к бесконечному ожиданию, в таком случае функция возвращает ненулевое значение, и если поток thread связанный, то в thread\_return записываются его выходные данные. Если ожидаемый поток является отсоединенным, то не имеет смысла вызывать эту функцию, хоть текущий поток и не войдет в режим бесконечного ожидания, вместо этого функция возвратит ненулевой код ошибки, но thread\_return не будет содержать выходных данных потока thread, так как ресурсы этого потока очищаются, по его завершении. Функция pthread\_join может быть вызвана потоком-родителем по отношению к прямому потомку, или некоторым потоком по отношению к потоку того же уровня. Варианты родитель – отдаленный потомок или потомок – родитель работать не будут.

**int pthread\_detach (pthread\_t thread)**

Функция отсоединяет поток с идентификатором thread и возвращает 0 в случае успеха. При этом поток thread не прерывается. Поток может отсоединить сам себя. Функцию нельзя применять к неинициализированному идентификатору потока.

**int pthread\_cancel (pthread\_t thread)**

Функция посылает сигнал прерывания потоку thread и возвращает 0 в случае успеха. Необходимо пояснить, что поток завершается не сразу, как только была вызвана функция, кроме того, функция возвращает нулевое значение, и тогда, когда поток не допускает прерывания. Функция не отсоединяет поток, если необходимо очистить ресурсы, следует использовать вызов pthread\_detach.

**int pthread\_setcancelstate (int state, int \*oldstate)**

Функция определяет, как текущий поток будет реагировать на вызов pthread\_cancel. Если state = PTHREAD\_CANCEL\_DISABLE, то поток не будет прерываться, если state = PTHREAD\_CANCEL\_ENABLE, поток завершиться, через некоторое время после вызова функции pthread\_cancel. В переменную oldstate записывается предыдущее значение параметра state.

Для использования этих функций, необходимо при компиляции программы использовать вызов: **g++ <файл с исходным кодом> -o <исполняемый файл> -lpthread**.

Рассмотрим простой пример, иллюстрирующий процесс создания потоков. В данном примере, порожденные потоки могут выполниться не полностью, они завершаться, когда выполниться главный поток.

#include <stdio.h>

**#include <pthread.h>**

*//стартовая функция для первого потока*

void\* func1(void \*)

{

for (int i=0 ; i<100000 ; i++)

fprintf(stdout, "Выполняется поток 1 - %d\n",i) ;

}

*//стартовая функция для второго потока*

void\* func2(void \*)

{

for (int i=0 ; i<100000 ; i++)

fprintf(stdout, "Выполняется поток 2 - %d\n",i) ;

}

int main()

{ *//идентификаторы для созданных потоков*

pthread\_t mythread1, mythread2 ;

*//создадим первый поток со стартовой функцией func1*

*//в качестве параметра стартовой функции передадим значение NULL*

*//атрибуты потока не указаны, значит, будут установлены по умолчанию*

pthread\_create(&mythread1, NULL, func1, NULL) ;

*//создадим второй поток со стартовой функцией func2*

pthread\_create(&mythread2, NULL, func2, NULL) ;

*//запустим цикл, если бы он был пропущен, то созданные потоки, скорее всего не успели бы отработать, т.к. главный поток завершился быстро\*/*

for (int i=0 ; i<100000 ; i++)

fprintf(stdout, "Выполняется главный поток - %d\n",i) ;

return 1 ;

}

Рис. 1. Схема потоков

Рассмотрим пример реализации скалярного произведения двух векторов, напомню, что для вычисления скалярного произведения необходимо найти произведения соответствующих координат векторов, затем их сложить. Таким образом, мы разделим оба вектора на две части и создадим два потока, каждый из которых будет вычислять скалярное произведение своей части векторов. Главный поток дождется выполнения порожденных потоков, и получит результат их вычислений.

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int A[100], B[100] ; *//два вектора*

struct bound *//структура, определяющая границы вычислений потоков*

{ int begin ; int end ; } ;

*//стартовая функция для обоих порожденных потоков*

void \* func (void \*param)

{

bound \*b = (bound \*)param ; //преобразование из void\* в bound\*

int \*sum ; *//скалярное произведение*

sum = new int ;

int i ; *//счетчик для цикла*

*//вычисление скалярного произведения*

for (i=b->begin, \*sum=0 ; i<b->end ; i++) \*sum += A[i]\*B[i] ;

return (void \*)sum ; *//возвращение результата*

}

int main()

{ int i ; *//счетчик для цикла*

int \*sum1 ; *//для получения результата вычислений первого потока*

int \*sum2 ; *//для получения результата вычислений второго потока*

bound b ; *//границы вычислений для первого потока*

bound c ; *//границы вычислений для второго потока*

for (i=0 ; i<100 ; i++)

{ *//заполнение массивов А и В случайными числами от 0 до 10*

A[i] = (int)(random()/(RAND\_MAX/10)) ;

B[i] = (int)(random()/(RAND\_MAX/10)) ;

}

*//идентификаторы для созданных потоков*

pthread\_t thread1, thread2 ;

b.begin = 0 ; b.end = 50 ;

*//создание первого потока, производящего вычисления от 0 до 50*

pthread\_create(&thread1,NULL,func,(void \*)&b) ;

c.begin = 50 ; c.end = 100 ;

*//создание первого потока, производящего вычисления от 50 до 100*

pthread\_create(&thread2,NULL,func,(void \*)&c) ;

*//ожидание завершения первого потока, в sum1 записывается адрес*

*//результата вычислений первого потока*

pthread\_join(thread1,(void \*\*)&sum1) ;

*//ожидание завершения второго потока*

pthread\_join(thread2,(void \*\*)&sum2) ;

fprintf(stdout,"Скалярное произведение = %d",\*sum1+\*sum2) ;

delete sum1 ; delete sum2 ;

return 1 ;

}

Заранее неизвестно какой из созданных потоков завершиться первым, но так как оба потока связанные, функция pthread\_join получает результат вычисления потока, и когда он уже завершился.

**Атрибуты потоков**

Для задания атрибутов (свойств) потока используется переменная структуры pthread\_attr\_t. Перед тем, как задавать параметры переменной атрибутов, необходимо ее инициализировать. После инициализации и определения свойств, переменная используется в функции создания потока - pthread\_create. Таким образом, поток получает атрибут во время своей инициализации, и многие параметры атрибута потока не могут быть изменены позже, во время его выполнения.

Рассмотрим функции, предназначенные для работы с атрибутами.

**int pthread\_attr\_init (pthread\_attr\_t \*attr)**

Функция инициализирует переменную атрибута attr и возвращает 0 в случае успеха. При этом с переменной ассоциируются такие значения как PTHREAD\_CREATE\_JOINABLE и SCHED\_OTHER.

**int pthread\_attr\_destroy (pthread\_attr\_t \*attr)**

Функция разрушает переменную атрибута attr и возвращает 0 в случае успеха.

**int pthread\_getattr\_np (pthread\_t th, pthread\_attr\_t \*attr)**

Устанавливает свойства переменной attr, в соответствии с атрибутом потока th. При этом поток th должен работать.

**int pthread\_attr\_setdetachstate (pthread\_attr\_t \*attr, int detachstate)**

Функция определяет свойство detachstate для переменной атрибута attr. Если detachstate равен PTHREAD\_CREATE\_JOINABLE, то поток будет связанным, если же detachstate равен PTHREAD\_CREATE\_DETACHED, поток будет отсоединенным. Напомню, что ресурсы связанного потока и его выходное значение, не очищаются сразу после завершения выполнения этого потока, для получения выходного значения и освобождения ресурсов используется функция pthread\_join. Функция возвращает 0 в случае успеха.

**int** **pthread\_attr\_getdetachstate (const pthread\_attr\_t \*attr, int \*detachstate)**

Возвращает значение свойства detachstate переменной attr в \* detachstate.

**int pthread\_attr\_setschedpolicy (pthread\_attr\_t \*attr, int policy)**

Функция определяет свойство policy для переменной атрибута attr. Свойство policy может принимать такие значения как SCHED\_OTHER, SCHED\_RR, или SCHED\_FIFO. Для SCHED\_FIFO потоки ранжируются по приоритетам, и начинают выполняться потоки с наивысшим приоритетом, все остальные ожидают пока исполняемые потоки «добровольно» освободят ресурсы, и когда это происходит, начинают выполняться те потоки, чей приоритет выше. Принцип действия политики управления SCHED\_RR близок к SCHED\_FIFO, за тем исключением, что выполняющийся поток может быть заблокирован по истечении определенного времени, для выполнения другого потока. Особенности политики управления SCHED\_OTHER зависят от операционной системы, также необходимо отметить, что свойства SCHED\_RR и SCHED\_FIFO поддерживаются не во всех операционных системах. Например, Solaris не осуществляет ни политику планирования SCHED\_FIFO, ни SCHED\_RR.

**int pthread\_attr\_getschedpolicy (const pthread\_attr\_t \*attr, int \*policy)**

Функция возвращает значение свойства policy в \*policy.

**int pthread\_attr\_setinheritsched (pthread\_attr\_t \*attr, int inherit)**

Функция определяет свойство inherit для переменной атрибута attr. Если inherit равен PTHREAD\_INHERIT\_SCHED, поток получает информацию о своем планировании от потока-родителя. Если inherit равен PTHREAD\_EXPLICIT\_SCHED, поток узнает о планировании из свойств переменной атрибутов.

**int pthread\_attr\_getinheritsched (const pthread\_attr\_t \*attr, int \*inherit)**

Возвращает значение свойства inherit в \* inherit.

**int pthread\_attr\_setstacksize (pthread\_attr\_t \*attr, size\_t stacksize)**

Свойство stacksize переменной атрибута attr определяет минимальный размер памяти в байтах, необходимый для запуска потока. Stacksize не может быть меньше, чем PTHREAD\_STACK\_MIN. Само значение PTHREAD\_STACK\_MIN является различным для разных операционных систем (часто 8 кб или 16 кб). Если минимальный размер не определен пользователем, то он устанавливается по умолчанию в некоторое значение, также зависящее от операционной системы (часто 64 кб, в Solaris – 1 мб). PTHREAD\_STACK\_MIN можно посмотреть в usr/include/bits/local\_lim.h

**int pthread\_attr\_getstacksize (const pthread\_attr\_t \*attr, size\_t \*stacksize)**

Записывает информацию о размере стека, указанного в переменной attr, в \*stacksize.

**int pthread\_attr\_setschedparam (pthread\_attr\_t \*attr, sched\_param \*param)**

Определяет параметр планирования потока param для переменной атрибута attr. Param содержит информацию только о приоритете потока. Структура sched\_param имеет следующий вид.

struct sched\_param

{ int sched\_priority; };

**int pthread\_attr\_getschedparam (pthread\_attr\_t \*attr, sched\_param \*param)**

Возврашает значение параметра планирования param переменной атрибута attr, в \* param.

Приоритет потока может быть изменен во время его выполнения.

**int pthread\_setschedparam (pthread\_t thread, int policy, sched\_param \*param)**

Функция изменяет приоритет потока с идентификатором thread на значение param, согласованное с политикой управления потока policy.

**int pthread\_getschedparam (pthread\_t thread, int \*policy, sched\_param \*param)**

Функция возвращает приоритет потока с идентификатором thread в \*param, а также политику планирования потока в \*policy.

Необходимо заметить, что допустимые приоритеты зависят от политики планирования и операционной системы. Приоритет потока должен находиться в определенных границах, посмотреть которые можно, вызвав следующие функции (заголовочный файл – sched.h).

**int sched\_get\_priority\_max(int policy)**

Функция возвращает максимальный возможный приоритет для политики планирования policy (SCHED\_OTHER, SCHED\_RR, или SCHED\_FIFO).

**int sched\_get\_priority\_min(int policy)**

Функция возвращает минимальный возможный приоритет для политики планирования policy.

Рассмотрим пример задания атрибута для потока.

pthread\_t mythread1, mythread2; //*идентификаторы потоков*

pthread\_attr\_t attr; //*переменная атрибута*

pthread\_attr\_init(&attr) ; //*инициализация переменной атрибута*

*//укажем для свойства detachstate значение PTHREAD\_CREATE\_DETACHED*

pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED) ;

*//укажем для свойства inherit значение PTHREAD\_EXPLICIT\_SCHED*

*//это необходимо, для того, чтобы поток получил информацию о приоритете*

*//из переменной атрибута*

pthread\_attr\_setinheritsched(&attr, PTHREAD\_EXPLICIT\_SCHED) ;

sched\_param param ; *//переменная параметра планирования*

param.sched\_priority = 0 ; *//установим приоритет в 0*

*//внесем значение параметра планирования в переменную атрибута attr*

pthread\_attr\_setschedparam(&attr,&param) ;

*//создадим два потока с одинаковыми атрибутами*

*//одна и та же переменная атрибута может быть использована для нескольких потоков*

pthread\_create(&mythread1, &attr, func, NULL) ;

pthread\_create(&mythread2, &attr, func, NULL) ;

*//запишем значение свойства detachstate переменной атрибута attr в i*

int i ; pthread\_attr\_getdetachstate(&attr,&i) ;

**Порядок выполнения лабораторной работы**

1. Разработать алгоритм решения задания, с учетом разделения вычислений между несколькими потоками. Избегать ситуаций изменения одних и тех же общих данных несколькими потоками. Составить схему потоков.
2. Реализовать алгоритм и протестировать на нескольких примерах.

**Пример №1**

Задание: Дана последовательность натуральных чисел a0…a99. Создать многопоточное приложение для поиска суммы квадратов ∑ai2. Потоки должны производить вычисления параллельно.

Разобьем последовательность чисел на четыре части и создадим четыре потока для вычисления суммы квадратов элементов в отдельной части последовательности. Главный поток вычислит окончательный результат, после того, как отработают четыре дочерних потока.

Рис. 2. Схема потоков

Реализуем задачу, используя библиотеку PTHREAD.

#include <pthread.h>

#include <stdio.h>

int A[100] ; *//последовательность чисел a0…a99*

*//стартовая функция для дочерних потоков*

void \*func(void \*param)

{*//вычисление суммы квадратов*

int i, p = ( \*(int \*)param )\*25 ;

int \*sum ; sum = new int ; \*sum=0 ;

for(i=p ; i<p+25 ; i++) \*sum+=A[i]\*A[i] ;

return (void \*)sum ;

}

int main()

{ int rez=0 ; *//для записи окончательного результата*

//тут должно быть заполнение массива A

pthread\_t thread[4] ; int number[4] ;

//создание четырех дочерних потоков

for (int i=0 ; i<4 ; i++)

{ number[i]=i ; //для передачи параметра потоку

pthread\_create(&thread[i], NULL, func, (void \*)(number+i)) ;

}

int \*sum ;

for (int i=0 ; i<4 ; i++)

{ *//ожидание завершения работы дочерних потоков*

*//и получение результата их вычислений*

pthread\_join(thread[i],(void \*\*)&sum) ;

rez += \*sum ;

delete sum ;

}

fprintf(stdout,"\nСумма квадратов = %d",rez) ; *//вывод результата*

return 1;

}

**Об особенностях работы с указателем на void**

*pthread\_create(&thread[i], NULL, func, (void \*)(number+i))*

number – массив, т.е. его имя это указатель на 0й элемент массива

number+i – указатель на i-й элемент массива

(void \*)(number+i) - преобразование указателя на i-й элемент массива в указатель на void

(void \*)(number+i) не тоже самое, что (void \*)number+i

*void \*func(void \*param)*

*{ int i, p = ( \*(int \*)param )\*25 ;*

(int \*)param – преобразование void\* в int\*

\*(int \*)param – получение значения указателя на int, т.е. получение того числа, которое расположено по адресу указателя

( \*(int \*)param )\*25 – умножение этого числа на 25

Если бы в вышеописанной программе вместо массива int number[4] ; применялась бы одна переменная int number – это могло бы привести к **ошибке**. Так как потоки работают параллельно или псевдопараллельно (т.е. работает то один, то другой, то один, то другой и т.д.) и все они обращаются к одному и тому же участку памяти - int number, то возможна ситуация когда два потока получат одинаковые числа, хотя предполагалось, что им отправлялись разные.

**Пример №2**

Задание: Дана последовательность натуральных чисел a0…a99. Создать многопоточное приложение для поиска суммы квадратов ∑ai2. Потоки должны производить вычисления последовательно.

Для вычисления суммы квадратов создадим два потока, первый поток будет находить квадраты чисел a0…a100, второй поток будет находить сумму элементов, вычисленных первым потоком. Второй поток будет запущен только тогда, когда первый поток найдет все квадраты.

Рис. 3. Схема потоков

#include <pthread.h>

#include <stdio.h>

int A[100] ; *//последовательность чисел a0…a99*

*//стартовая функция для второго потока*

void \*summa(void \*param)

{ int sum=0 ; *//вычисление суммы*

for(int i=0 ; i<100 ; i++) sum+=A[i] ;

return (void \*)sum ;

}

*//стартовая функция для первого потока*

void \*kvadrat(void \*param)

{ int rez ; *//вычисление квадратов*

for(int i=0 ; i<100 ; i++) A[i]\*=A[i] ;

pthread\_t threadS ; *//создание второго потока*

pthread\_create(&threadS, NULL, summa, NULL) ;

*/\*ждем завершения второго потока и передаем его результат главному потоку\*/*

pthread\_join(threadS,(void \*\*)&rez) ;

return (void \*)rez ;

}

int main()

{ int rez=0 ; for (int i=0 ; i<100 ; i++) A[i] = 5 ;

pthread\_t threadK ; *//создание первого потока*

pthread\_create(&threadK, NULL, kvadrat, NULL) ;

*//ожидание завершения первого потока*

pthread\_join(threadK,(void \*\*)&rez) ;

fprintf(stdout,"\nСумма квадратов = %d",rez) ;

return 1;

}