# Лабораторная работа 11. Многопоточность. Синхронизация потоков в Linux.

**Аннотация:** Цель работы: получить представление об инструментах управления потоками в Linux, изучить основные механизмы синхронизации потоков в Linux.

**Задание 1**. Изучите основные команды управления потоками в Linux.

Ход выполнения:

1. Прочитайте вводную информацию о потоках в Linux.

POSIX регламентирует поточную библиотеку с IEEE Std 1003.1c-1995, также известную как POSIX 1995 или POSIX.1c. Разработчики называют этот стандарт POSIX-потоками, или, для краткости, Р-потоками. Р-потоки остаются лидирующим решением в отношении поточности для С и С++ в системах UNIX.

API для работы с Р-потоками определяется в *<pthread.h>*.

2. Ознакомьтесь с функцией *pthread\_create()*.

Р-потоки обеспечивают одну функцию для определения и запуска нового потока, которая называется pthread\_create():

#include <pthread.h>

int pthread\_create (pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*),

void \*arg);

При успешном выполнении создается новый поток и начинает выполнять функцию, представленную в *start\_routine* с единственным аргументом *arg*. Функция сохранит идентификатор потока, использованный для представления нового потока, в *pthread\_t*, указанного с помощью *thread*, если он не равен *NULL*.

Объект *pthread\_attr\_t*, определяемый через *attr*, используется для изменения атрибутов потока, присваиваемых по умолчанию, для вновь создаваемого потока. Большинство вызовов *pthread\_create()* отправляет в качестве аргумента *NULL*, присваивая атрибуты по умолчанию. Атрибуты потока позволяют программам изменять множество свойств потока, таких как размер стека, параметры планирования, начальный отдельный статус. Функция *start\_routinе* должна иметь запись следующего вида:

void \* start\_thread (void \*arg);

Таким образом поток начинает существование с исполнения функции, которая принимает указатель *void* как единственный аргумент, а затем возвращает его как свою возвращаемую величину.

В случае ошибки *pthread\_create()* возвращает ненулевой код ошибки напрямую (без использования *errno*) и содержимое потока *thread* не определено.

3. Пример использования *pthread\_create()*:

pthread\_t tread;

int ret;

ret = pthread\_create (&thread, NULL, start\_routine, NULL);

if (!ret) {

errno = ret;

perror("pthread\_create");

return -1;

}

4. Ознакомьтесь с понятием идентификатора потока.

*Идентификаторы потоков (TID)* для потоков являются аналогами идентификаторов процессов (PID). В то время как PID назначаются ядром Linux, TID назначаются всего лишь библиотекой Р-потоков. Этот тип представлен *pthread\_t*. TID нового потока определяется с помощью аргумента *thread* при успешном вызове *pthread\_create()*. Поток может получить свой TID при запуске с помощью функции pthread\_self():

#include <pthread.h>

pthread\_t pthread\_self (void);

Использовать функцию очень просто, так как она всегда работает успешно:

const pthread\_t me = pthread\_self ();

5. Изучите особенности сравнения идентификаторов потоков.

Стандарт Р-потоков не требует, чтобы *pthread\_t* был арифметического типа, поэтому нет гарантии, что оператор равенства будет работать. Следовательно, чтобы сравнить идентификаторы потоков, библиотеке Р-потоков нужен специальный интерфейс:

#include <pthread.h>

int pthread\_equal (pthread\_t t1, pthread\_t t2);

Если приведенные идентификаторы потоков равны, *pthread\_equal()* возвращает ненулевую величину. Если они не равны, возвращается *0*; ошибка не может про­изойти. Пример:

int ret;

ret = pthread\_equal(thing1, thing2);

if (ret != 0)

printf("The TIDs are equal!\n");

else

printf("The TIDs are unequal!\n");

6. Проверьте правильность работы программного кода, приведенного в п.5.

**Задание 2**. Изучите основные функции завершения, присоединения и отсоединения потоков в Linux.

Ход выполнения:

1. Прочитайте информацию о самозавершении потоков в Linux.

Самый простой путь для потока, чтобы завершить самого себя, — это «выход за пределы» своей начальной процедуры. Однако часто вам будет нужно завершить поток где-то в глубине стека вызова функции, далеко от вашей стартовой процедуры. Для таких случаев в Р-потоках имеется вызов *pthread\_exit()*, поточный эквивалент *exit()*:

#include <pthread.h>

void pthread\_exit (void \*retval);

По выполнении вызывающий поток завершается; *retval* обеспечивается для каждого потока, ожидающего завершения, аналогично *exit()*. Ошибка не может произойти.

Использование:

/\* Прощай, жестокий мир! \*/

pthread\_exit (NULL);

2. Прочитайте информацию о завершении других потоков в Linux.

Р-потоки вызывают завершение других потоков через их отмену. Это обеспечивает функция *pthread\_cancel()*:

#include <pthread.h>

int pthread\_cancel (pthread\_t thread);

Успешный вызов *pthread\_cancel()* посылает запрос на отмену потоку, представленному через идентификатор потока *thread*. Может ли поток быть отменен и когда, зависит от его состояния отмены и типа отмены соответственно. В случае успеха *pthread\_cancel()* возвращает *0*. Обратите внимание, что успех в данном случае означает лишь успешную обработку запроса на отмену. В действительности же завершение происходит асинхронно. В случае ошибки *pthread\_cancel()* возвращает *ESRCH*, означающее, что значение *thread* недопустимо.

Состояние отмены потока может быть доступно или недоступно. По умолчанию оно является доступным для новых потоков. С другой стороны, тип отмены указывает, когда происходит отмена. Потоки могут изменять свое состояние через *pthread\_setcancelstate()*:

#include <pthread.h>

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

В случае успеха состояние отмены вызывающего потока устанавливается на *state*, а предыдущее состояние сохраняется в *oldstate*. Значением *state* может быть PTHREAD\_CANCEL\_ENABLE или PTHREAD\_CANCEL\_DISABLE для разрешения или запрещения отмены соответственно.

В случае ошибки *pthread\_setcancelstate()* возвращает EINVAL, что означает недопустимое значение *state*.

Тип отмены потока может быть асинхронным или отложенным; по умолчанию обычно установлен последний. С асинхронным типом отмены поток может быть убит в любой точке после получения команды на отмену. С отложенным типом поток может быть убит только в специальных точках отмены, которые являются функциями Р-потоков или библиотеки С и представляют собой безопасные моменты, в которых вызывающий поток может быть прерван.

Потоки могут изменить свой тип через *pthread\_setcanceltype()*:

#include <pthread.h>

int pthread\_setcanceltype (int type, int \*oldtype);

В случае успеха статус отмены вызывающего потока устанавливается в *type*, а старый тип сохраняется в *oldtype*. Значением *type* может быть PTHREAD\_CANCEL\_ASYNCHRONOUS или PTHREAD\_CANCEL\_DEFERRED для установки асинхронной или отложенной отмены соответственно.

В случае ошибки *pthread\_setcanceltype()* возвращает EINVAL, что означает недопустимое значение *type*.

3. Рассмотрите пример, когда один поток должен завершить другой. Сначала поток, который должен завершиться, разрешает свою отмену и устанавливает ее тип как отложенный (эти значения установлены по умолчанию, так что в данном случае это просто пример):

int unused;

int ret;

ret = pthread\_setcancelstate (PTHREAD\_CANCEL\_ENABLE, &unused);

if (ret) {

errno = ret;

perror ("pthread\_setcancelstate");

return -1;

}

ret = pthread\_setcanceltype (PTHREAD\_CANCEL\_DEFERRED, &unused);

if (ret) {

errno = ret;

perror ("pthread\_setcanceltype");

return -1;

}

Затем другой поток посылает команду на завершение:

int ret;

/\*'thread' в данном случае означает идентификатор завершаемого потока \*/

ret = pthread\_cancel (thread);

if (ret) {

errno = ret;

perror ("pthread\_cancel");

return -1;

}

4. Прочитайте информацию о присоединении потоков.

Присоединение позволяет одному из потоков заблокироваться в ожидании завершения другого:

#include <pthread.h>

int pthread\_join (pthread\_t thread, void \*\*retval);

После успешного выполнения вызывающий поток блокируется до тех пор, пока поток, указанный как *thread*, не завершится (если *thread* уже завершен, *pthread\_join()* возвращается немедленно). Как только *thread* завершается, вызывающий поток активизируется и, если *retval* не равен *NULL*, получает возвращаемое значение завершенного процесса, переданное *pthread\_exit()* или возвращенное от его стартовой процедуры. После этого можно сказать, что потоки присоединились друг к другу.

Присоединение всегда позволяет потокам синхронизировать свое выполнение по отношению к периоду существования других потоков. Все потоки в Р-потоках являются равноправными; каждый поток может присоединяться к любому другому. Один поток может присоединяться ко многим (фактически, как мы скоро увидим, чаще всего один главный поток ожидает других потоков, которые сам и создал), но только один поток может пытаться присоединиться к определенному другому, несколько потоков не должны стараться присоединиться к какому-либо одному.

5. Рассмотрите следующий пример программного кода:

int ret;

/\* присоединяем к 'thread' и больше не заботимся о возвращаемой величине \*/

ret = pthread\_join (thread, NULL);

if (ret) {

errno = ret;

perror ("pthread\_join");

return -1;

}

6. Прочитайте информацию об отсоединении потоков.

По умолчанию потоки создаются способными к присоединению. Однако они могут и отсоединяться, но в этом случае они станут в дальнейшем неприсоединяемыми. Поскольку до присоединения потоки потребляют какие-либо системные ресурсы, как делают это и процессы, пока их предки вызывают *wait()*, потоки, которые вы не планируете присоединять, должны быть отсоединены.

#include <pthread.h>

int pthread\_detach (pthread\_t thread);

В случае успеха *pthread\_detach()* отсоединяет поток, указанный как *thread*, и воз­вращает *0*. Результаты не определены, если вы вызываете *pthread\_detach()* относи­тельно потока, который уже отсоединен. В случае ошибки функция возвращает значение ESRCH, означающее, что значение *thread* недопустимо.

Для каждого потока в процессе необходимо вызвать *pthread\_join()* или *pthread\_detach()*, чтобы системные ресурсы могли высвободиться после завершения пото­ка (конечно, после того как завершается весь процесс, все поточные ресурсы вы­свобождаются, но присоединение или отсоединение всех процессов в явной форме остается хорошей практикой).

7. Наберите следующий программный код:

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

void \* start\_thread (void \*message)

{

printf ("%s\n", (const char \*) message);

return message;

}

int main (void)

{

pthread\_t thing1, thing2;

const char \*message1 = "Thing 1";

const char \*message2 = "Thing 2";

pthread\_create (&thing1, NULL, start\_thread, (void \*) message1);

pthread\_create (&thing2, NULL, start\_thread, (void \*) message2);

pthread\_join (thing1, NULL);

pthread\_join (thing2, NULL);

return 0;

}

8. Проанализируйте программный код из п.7. Выясните, какие операции он выполняет.

9. Зачем необходимо выполнять присоединение потоков в программном коде п. 7?

10. Сохраните программу в файле под именем *example.c*. Скомпили­руйте ее с помощью такой команды:

gcc -Wall -O2 -pthread example.c -o example

11. На выходе работы программы могут быть получены разные результаты. Однако ненужной информации никогда не будет. Почему?

**Задание 3**. Напишите программу, создающую три потока, увеличивающих значение некоторой глобальной переменной на единицу и выводящих на экран свой номер и измененное ими значение переменной.

**Задание 4**. Напишите программу, создающую два дочерних процесса.

1. Родительский и дочерние процессы должны выводить на экран свой PID (идентификатор), PID родителя и текущее время в формате ***часы: минуты: секунды: миллисекунды***.

2. В основной программе создать два потока. Процесс-отец создает файл и записывает в него 100 строк вида ***N pid, текущее время***, где ***N*** – номер текущей строки.

3. Оба потока считывают строки из файла и выводят их на экран в виде ***N потока, текущее время, строка из файла***.

**Задание 5**. Изучите основные функции, используемые для управления мьютексом в Linux.

Ход выполнения:

1. Прочитайте информацию об инициализации, запирании и отпирании и запирании мюьтексов.

Мьютексы представляются объектом *pthread\_mutex\_t*. Как и большинство объектов в API Р-потоков, это подразумевает их непрозрачную структуру, обеспечивающую разнообразие интерфейсов мьютексов. Хотя вы можете создавать мьютексы динамически, в большинстве случаев их использование статично:

/\* определим и запустим мьютекс с именем 'mutex' \*/

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

Этот фрагмент кода определяет и инициализирует мьютекс под названием *mutex*.

**Запирание**, называемое также завладением, Р-поточного мьютекса обеспечивается с помощью функции *pthread\_mutex\_lock()*:

#include <pthread.h>

int pthread\_mutex\_lock (pthread\_mutex\_t \*mutex);

Успешный вызов *pthread\_mutex\_lock()* заблокирует вызывающий поток, пока мьютекс, указанный как *mutex*, не станет доступным. После этого вызывающий поток активируется и эта функция вернет *0*. Если мьютекс доступен в момент вызова, функция вернет значение немедленно.

Вот пример использования:

pthread\_mutex\_lock (&mutex);

Противоположностью запиранию является **отпирание**, или высвобождение, мьютексов.

#include <pthread.h>

int pthread\_mutex\_unlock (pthread\_mutex\_t \*mutex);

Успешный вызов *pthread\_mutex\_unlock()* высвобождает мьютекс, указанный как *mutex*, и возвращает *0*. Вызов не блокируется; мьютекс освобождается немедленно.

pthread\_mutex\_unlock (&mutex);

2. Рассмотрите фрагмент кода, в котором показано использование мьютексов для обеспечения синхронизации.

Представим себе банкомат. Алгоритм его работы получится примерно следующий.

1. Имеется ли на счету как минимум Х единиц валюты?

2. Если да, уменьшить размер счета на величину Х и выдать пользователю Х денежных единиц.

3. Если нет, выдать сообщение об ошибке.

Код на С будет выглядеть приблизительно так:

int withdraw (struct account \*account, int amount)

{

const int balance = account->balance;

if (balance < amount)

return -1;

account->balance = balance - amount;

disburse\_money (amount);

return 0;

}

Представьте, что банк выполняет эту операцию дважды, параллельно. Например, пользователь снимает деньги одновременно с тем, как банк обрабатывает онлайн оплату счетов или взимает какой-либо регулярный сбор. Что, если снятие средств произойдет одновременно, перед тем как баланс счета обновится?

Обе операции могут осуществиться, даже если баланса счета недостаточно, чтобы проделать и то и другое. Например, если на счету было $500 и одновременно пришли два запроса на снятие $200 и $400, они оба могут быть удовлетворены, хотя на счете окажется отрицательная величина –$100, чего явно не желал допустить программист, написавший код.

Наш воображаемый банк находится в угрожающих условиях гонки, что может иметь нежелательные последствия. Вот как мы можем исправить это с использованием Р-поточных мьютексов:

static pthread\_mutex\_t the\_mutex = PTHREAD\_MUTEX\_INITIALIZER;

int withdraw (struct account \*account, int amount)

{

pthread\_mutex\_lock (&the\_mutex);

const int balance = account->balance;

if (balance < amount) {

pthread\_mutex\_unlock (&the\_mutex);return -1;

}

account->balance = balance - amount;

pthread\_mutex\_unlock (&the\_mutex);

disburse\_money (amount);

return 0;

}

В этом примере используется *pthread\_mutex\_lock()* для завладения мьютексом, а затем *pthread\_mutex\_unlock()* для освобождения его в конечном итоге. Это позволяет избежать условий гонки, но создает для банка новую проблему: в каждый момент времени только один клиент может снять деньги! Здесь-то и находится самое узкое место; а для «слишком-больших-чтобы-обанкротиться» банков это настоящий провал.

Поэтому в большинстве случаев при использовании замков стараются избегать глобальных блокировок, а вместо этого связывают замки с конкретными структурами данных. Это называется *детализированной блокировкой*. Такой подход может усложнить ваши блокирующие семантики, в частности, при избегании взаимной блокировки, но он является ключевым моментом при увеличении количества ядер на современных машинах.

3. Вместо того, чтобы определять глобальную блокировку *the\_mutex*, мы определим мьютекс внутри структуры *account*, выделяя каждому счету собственную блокировку. Это сработает, так как данные внутри критической области – только структура *account*. Запирая только сам счет в момент списания, мы даем банку возможность параллельно выполнять операции других клиентов.

int withdraw (struct account \*account, int amount)

{

pthread\_mutex\_lock (&account->mutex);

const int balance = account->balance;

if (balance < amount) {

pthread\_mutex\_unlock (&account->mutex);

return -1;

}

account->balance = balance - amount;

pthread\_mutex\_unlock (&account->mutex);

disburse\_money (amount);

return 0;

}

4. Наберите и сравните работу фрагментов программного кода из п. 2 и из п.3.

**Задание 6**. Изучите основные функции, используемые для управления семафорами в Linux.

Ход выполнения:

1. Прочитайте информацию об особенностях использования семафоров для синхронизации потоков.

Семафор предназначен для синхронизации потоков по действиям и по данным. Семафор – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций P и V и операции инициализации. Семафор может принимать целое неотрицательное значение.

При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при S>0 или поток блокируется, «ожидая на семафоре», при S=0. При выполнении операции V(S) происходит пробуждение одного из потоков, ожидающих на семафоре S, а если таковых нет, то значение семафора увеличивается на 1. Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию P, а при выходе из критической секции операцию V.

Прототипы функций для манипуляции с семафорами описываются в файле *semaphore.h*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

*int sem\_init(sem\_t\* sem, int pshared, unsigned int value)* – инициализация семафора *sem* значением *value*. В качестве *pshared* всегда необходимо указывать *0*.

*int sem\_wait(sem\_t\* sem)* – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1.

*int sem\_post(sem\_t\* sem)* – увеличивает значение семафора *sem*.

*int sem\_destroy(sem\_t\* sem)* – уничтожает семафор *sem*.

*int sem\_trywait(sem\_t\* sem)* – неблокирующий вариант функции *sem\_wait*. При этом вместо блокировки вызвавшего потока функция возвращает управление с кодом ошибки в качестве результата работы.

2. Наберите, откомпилируйте и запустите следующий программный код, демонстрирующий использование семафоров для синхронизации потоков.

#include <stdlib.h>

#include <stdio.h>

#include <errno.h>

#include <pthread.h>

#include <semaphore.h>

sem\_t sem;

void \* thread\_func(void \*arg)

{

int i;

int loc\_id = \* (int \*) arg;

sem\_post(&sem);

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

printf("Thread %i is running\n", loc\_id);

sleep(1);

}

}

int main(int argc, char \* argv[])

{

int id, result;

pthread\_t thread1, thread2;

id = 1;

sem\_init(&sem, 0, 0);

result = pthread\_create(&thread1, NULL, thread\_func, &id);

if (result != 0) {

perror("Creating the first thread");

return EXIT\_FAILURE;

}

sem\_wait(&sem);

id = 2;

result = pthread\_create(&thread2, NULL, thread\_func, &id);

if (result != 0) {

perror("Creating the second thread");

return EXIT\_FAILURE;

}

result = pthread\_join(thread1, NULL);

if (result != 0) {

perror("Joining the first thread");

return EXIT\_FAILURE;

}

result = pthread\_join(thread2, NULL);

if (result != 0) {

perror("Joining the second thread");

return EXIT\_FAILURE;

}

sem\_destroy(&sem);

printf("Done\n");

return EXIT\_SUCCESS;

}

3. Убедитесь в правильности работы программы. Проверьте необходимость использования механизма синхронизации в программе.

**Задание 7**. Изучите основные функции, предназначенные для работы с условными переменными, как инструментом синхронизации в Linux.

Ход выполнения:

1. Прочитайте информацию об особенностях использования условных переменных для синхронизации потоков.

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события. При выполнении операции «информирование» один из потоков, ожидающих на условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подаче сигнала другим потокам рекомендуется так же функцию «сигнализации» защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле *pthread.h*. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

*pthread\_cond\_init(pthread\_cond\_t\* cond, const pthread\_condattr\_t\* attr)* – инициализирует условную переменную *cond* с указанными атрибутами *attr* или с атрибутами по умолчанию (при указании *0* в качестве *attr*).

*int pthread\_cond\_destroy(pthread\_cond\_t\* cond)* – уничтожает условную переменную *cond*.

*int pthread\_cond\_signal(pthread\_cond\_t\* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*.

*int pthread\_cond\_broadcast(pthread\_cond\_t\* cond)* – информирование о наступлении события потоков, ожидающих на условной переменной *cond*. При этом возобновлены будут все ожидающие потоки.

*int pthread\_cond\_wait(pthread\_cond\_t\* cond , pthread\_mutex\_t\* mutex)* – ожидание события на условной переменной *cond*.

2. Ниже приведен фрагмент программы с синхронизацией с использованием условных переменных. Программа выполняет синхронизацию записи (*writer*) и чтения (*reader*) данных в буфер data и из него. Емкость буфера – одна запись.

#include <iostream.h>

#include <semaphore.h>

#include <fstream.h>

#include <stdio.h>

#include <error.h>

…

int full;

pthread\_mutex\_t mx;

pthread\_cond\_t cond;

int data;

void \* writer(void \*)

{

while(1)

{

int t=write\_to\_buffer();

pthread\_mutex\_lock(&mx);

while (full) {

pthread\_cond\_wait(&cond, &mx);

}

data=t;

full=1;

pthread\_cond\_signal(&mx);

pthread\_mutex\_unlock(&mx);

}

return NULL;

}

void \* reader(void \*)

{

while(1)

{

int t;

pthread\_mutex\_lock(&mx);

while (!full) {

pthread\_cond\_wait(&cond, &mx);

}

t=data;

full=0;

pthread\_cond\_signal(&mx);

pthread\_mutex\_unlock(&mx);

}

return NULL;

}

…

3. Используйте фрагмент кода из п.2 в своей программе.

4. Чем будет отличаться результат работы программы, если не использовать механизмы синхронизации?

**Задание 8**. Напишите многопоточное приложение для определения того, является ли заданное число совершенным. Число *N* является совершенным, если сумма всех его делителей, исключая само это число, равна *N* (примерами могут послужить числа 6 и 28). На входе задается целое число *N*. На выходе будет значение *true*, если число является совершенным, или *false*, если оно таковым не является. Основная программа будет читать числа N и P (количество потоков) из командной строки. Числа от 1 до N будут делиться между этими потоками, чтобы два потока не работали с одним и тем же числом. Для каждого числа в этом наборе поток будет определять, является ли число делителем числа N. Если оно таковым является, то это число добавляется к общему буферу, хранящему делители числа N.

Родительский процесс ожидает, пока не завершат работу все потоки. Воспользуйтесь соответствующим примитивом синхронизации. Затем родительский поток определяет, является ли введенное число совершенным, то есть является ли оно суммой всех делителей, а затем выдает соответствующий отчет.

*Примечание:* вычисление можно ускорить, ограничив количество чисел, проводя поиск от 1 до корня квадратного из N.