ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция 14 — Нити POSIX

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2022

2023.05.10

Оглавление

Потоки (нити)

В традиционных операционных системах у каждого процесса есть адресное пространство и единственный поток управления. Фактически это почти что определение процесса. Тем не менее нередко возникают ситуации, когда неплохо было бы иметь в одном и том же адресном пространстве несколько потоков управления, выполняемых *квазипараллельно*, как будто они являются чуть ли не обособленными процессами, за исключением общего адресного пространства.

Применение потоков

Необходимость в подобных мини-процессах, называемых потоками, обусловливается целым рядом причин. Основная причина использования потоков заключается в том, что во многих приложениях одновременно происходит несколько действий, часть которых может периодически блокироваться.

За счет разделения такого приложения на несколько последовательных потоков, выполняемых в квазипараллельном режиме, модель программирования упрощается. В отличие от параллельных процессов добавляется новый элемент — ***возможность использования параллельными процессами единого адресного пространства и всех имеющихся данных.***

Эта возможность играет весьма важную роль для тех приложений, которым не подходит использование нескольких процессов с их раздельными адресными пространствами.

Вторым аргументом в пользу потоков является ***легкость (то есть быстрота) их создания и ликвидации по сравнению с более «тяжеловесными» процессами***. Во многих системах создание потоков осуществляется в 10–100 раз быстрее, чем создание процессов. Это свойство особенно важно, когда требуется быстро и динамично изменять количество потоков.

Третий аргумент в пользу потоков — ***производительность***.

Когда потоки работают в рамках одного центрального процессора, они не приносят никакого прироста производительности, но когда выполняются значительные вычисления, а также значительная часть времени тратится на ожидание ввода-вывода, наличие потоков позволяет этим действиям перекрываться по времени, ускоряя работу приложения.

И наконец, потоки весьма полезны для систем, имеющих несколько центральных процессоров (ядер), где есть реальная возможность параллельных вычислений.

Интерактивное приложение

Текстовый процессор может быть написан как двухпоточная программа.

Один из потоков взаимодействует с пользователем, а другой занимается формированием представления в фоновом режиме. Как только предложение с первой страницы, например, будет удалено, поток, отвечающий за взаимодействие с пользователем, приказывает потоку, отвечающему за представление, переформатировать документ.

Пока взаимодействующий поток продолжает отслеживать события клавиатуры и мыши, реагируя на простые команды вроде прокрутки первой страницы, второй поток с большой скоростью выполняет вычисления. Может так случиться, что переформатирование закончится до того, как пользователь запросит просмотр 600-й страницы, которая тут же сможет быть отображена.

Многие текстовые процессоры автоматически сохраняют рабочий файл на диск каждые несколько минут — еще один поток может заниматься созданием резервных копий на диске.

Если бы программа была рассчитана на работу только одного потока, то с начала создания резервной копии на диске и до его завершения игнорировались бы команды с клавиатуры или мыши. Пользователь ощущал бы это как слабую производительность.

Программная модель, использующая три потока, гораздо проще. Первый поток занят только взаимодействием с пользователем. Второй поток по необходимости занимается переформатированием документа. А третий поток периодически сбрасывает содержимое памяти на диск.

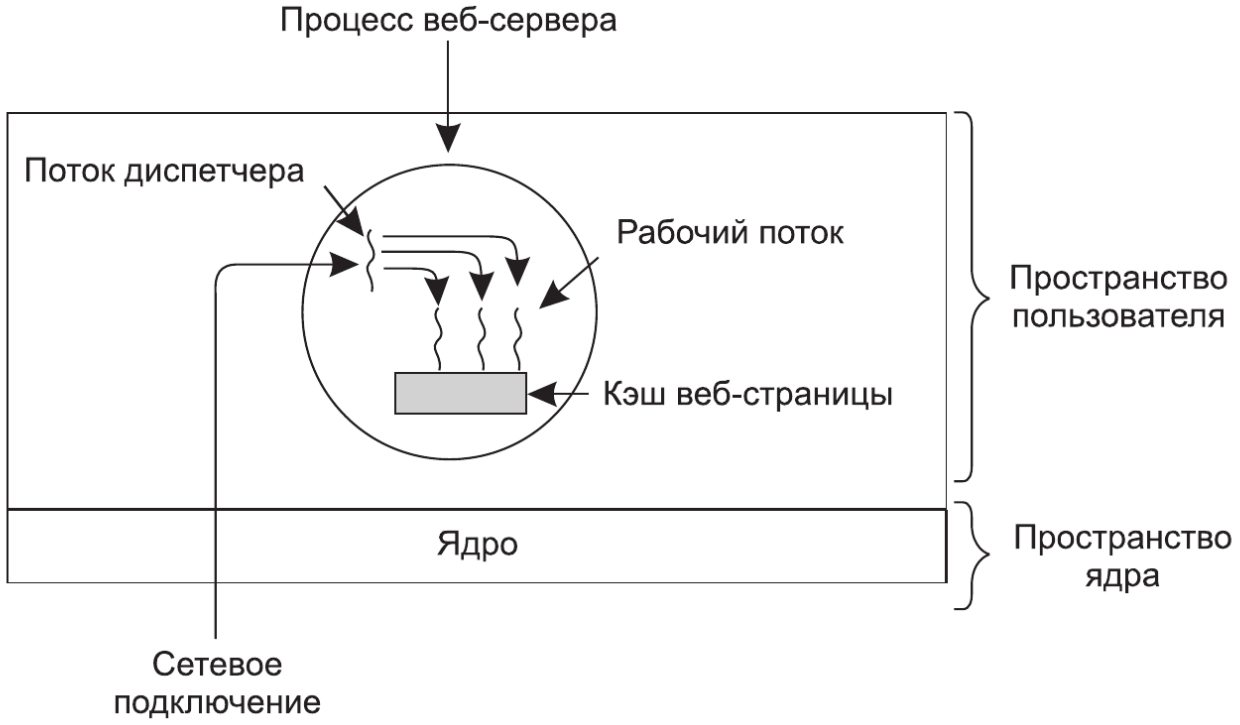
Три отдельных процесса так работать не будут, поскольку с документом необходимо работать всем трем потокам. Три потока вместо трех процессов используют общую память, таким образом, все они имеют доступ к редактируемому документу. При использовании трех процессов такое было бы слишком сложно.

Веб-сервер

Поступают запросы на страницы, и запрошенные страницы отправляются обратно клиентам. Некоторые страницы запрашиваются чаще других. Это обстоятельство для повышения производительности за счет размещения содержания часто используемых страниц в основной памяти, чтобы исключить необходимость обращаться за ними к диску (кеширование).

Один из способов организации веб-сервера показан на рисунке ниже

Один из потоков —диспетчер — читает входящие рабочие запросы из сети. После анализа запроса он выбирает простаивающий (то есть заблокированный) рабочий поток и передает ему запрос, возможно, путем записи указателя на сообщение в специальное слово, связанное с каждым потоком. Затем диспетчер пробуждает спящий рабочий поток, переводя его из заблокированного состояния в состояние готовности.



При пробуждении рабочий поток проверяет, может ли запрос быть удовлетворен из кэша веб-страниц, к которому имеют доступ все потоки. Если нет, то он, чтобы получить веб-страницу, приступает к операции чтения с диска и блокируется до тех пор, пока не завершится дисковая операция. Когда поток блокируется на дисковой операции, выбирается выполнение другого потока, возможно, диспетчера, с целью получения следующей задачи или, возможно, другого рабочего потока, который находится в готовности к выполнению.

Эта модель позволяет запрограммировать сервер в виде набора последовательных потоков.

Программа диспетчера состоит из бесконечного цикла для получения рабочего запроса и перепоручения его рабочему потоку.

Код каждого рабочего потока состоит из бесконечного цикла, в котором принимается запрос от диспетчера и проверяется кэш на присутствие в нем страницы. Если страница в кэше, она возвращается клиенту. Если нет, поток получает страницу с диска, возвращает ее клиенту и блокируется в ожидании нового запроса. Приблизительный набросок кода показан ниже.

buf и **page** являются структурами, предназначенными для хранения рабочего запроса и веб-страницы соответственно.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf)
}
```

```
while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Потоки дают возможность сохранить идею последовательных процессов, которые осуществляют блокирующие системные вызовы (например, для операций дискового ввода-вывода), но при этом позволяют добиться распараллеливания работы.

Три способа создания сервера

Модель	Характеристики
Потоки	Параллельная работа, блокирующие системные вызовы
Однопоточный процесс	Отсутствие параллельной работы, блокирующие системные вызовы
Машина с конечным числом состояний	Параллельная работа, неблокирующие системные вызовы, прерывания

Блокирующие системные вызовы упрощают программирование, а параллельная работа повышает производительность.

Однопоточные серверы сохраняют простоту блокирующих системных вызовов, но уступают им в производительности.

Третий подход (машина с конечным числом состояний) позволяет добиться высокой производительности за счет параллельной работы, но использует неблокирующие вызовы и прерывания, усложняя процесс программирования.

Обработка большого объема данных

Еще одним примером, подтверждающим пользу потоков, являются приложения, предназначенные для обработки очень большого объема данных.

При обычном подходе блок данных считывается, после чего обрабатывается, а затем снова записывается.

При доступности лишь блокирующих вызовов процесс блокируется и при поступлении данных, и при их возвращении. Простой центрального процесса при необходимости в большом объеме вычислений слишком расточителен и его по возможности следует избегать.

Проблема решается с помощью потоков. Структура процесса может включать входной поток, обрабатывающий поток и выходной поток. Входной поток считывает данные во входной буфер. Обрабатывающий поток извлекает данные из входного буфера, обрабатывает их и помещает результат в выходной буфер. Выходной буфер записывает эти результаты обратно на диск. Таким образом, ввод, вывод и обработка данных могут осуществляться одновременно.

Эта модель работает лишь при том условии, что системный вызов блокирует только вызывающий поток, а не весь процесс.

Классическая модель потоков

Модель процесса основана на двух независимых понятиях — группировке ресурсов и исполнении. Процесс является способом группировки в единое целое взаимосвязанных ресурсов.

У процесса есть адресное пространство, содержащее текст программы и данные, а также другие ресурсы. Эти ресурсы могут включать открытые файлы, необработанные аварийные сигналы, обработчики сигналов, учетную информацию и т. д. Управление этими ресурсами можно значительно облегчить, если собрать их воедино в виде процесса.

Другое присущее процессу понятие — поток исполнения (*поток)*.

У потока есть счетчик команд, отслеживающий, какую очередную инструкцию нужно выполнять. У него есть регистры, в которых содержатся текущие рабочие переменные. У него есть стек с протоколом выполнения, содержащим по одному фрейму для каждой вызванной, но еще не возвратившей управление процедуры. Хотя поток может выполняться в рамках какого-нибудь процесса, сам поток и его процесс являются разными понятиями и должны рассматриваться отдельно.

Процессы используются для группировки ресурсов в единое образование, а потоки являются «сущностью», распределяемой для выполнения на центральном процессоре.

Потоки добавляют к модели процесса возможность реализации нескольких в значительной степени независимых друг от друга выполняемых задач в единой среде процесса.

Несколько потоков, выполняемых параллельно в рамках одного процесса, являются аналогией нескольких процессов, выполняемых параллельно на одном компьютере. В первом случае потоки используют единое адресное пространство и другие ресурсы, процессы используют общую физическую память, диски, принтеры и другие ресурсы.

Поскольку потоки обладают некоторыми свойствами процессов, их иногда называют ***облегченными процессами***.

Различные потоки в процессе не обладают той независимостью, которая есть у различных процессов. У всех потоков абсолютно одно и то же адресное пространство, а значит, они так же совместно используют одни и те же глобальные переменные.

Каждый поток может иметь доступ к любому адресу памяти в пределах адресного пространства процесса, один поток может считывать данные из стека другого потока, записывать туда свои данные и даже стирать оттуда данные. Защита между потоками отсутствует, потому что ее невозможно осуществить и в ней нет необходимости.

В отличие от различных процессов, которые могут принадлежать различным пользователям и которые могут конфликтовать друг с другом, один процесс всегда принадлежит одному и тому же пользователю, который, по-видимому, и создал несколько потоков для их совместной работы.

Помимо использования общего адресного пространства все потоки могут совместно использовать одни и те же открытые файлы, дочерние процессы, ожидаемые и обычные сигналы и т. п.

Использование объектов потоками

Элементы, присущие каждому процессу	Элементы, присущие каждому потоку
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Необработанные аварийные сигналы	
Сигналы и обработчики сигналов	
Учетная информация	

Подобно традиционному процессу (то есть процессу только с одним потоком), поток должен быть в одном из следующих состояний:

- выполняемый
- заблокированный
- готовый
- завершенный.

Переходы между состояниями потока аналогичны переходам между состояниями процесса.

Стек потока

Каждый поток имеет собственный стек. Стек каждого потока содержит по одному фрейму для каждой уже вызванной, но еще не возвратившей управление процедуры.

Такой фрейм содержит локальные переменные процедуры и адрес возврата управления по завершении ее вызова. Например, если процедура X вызывает процедуру Y, а Y вызывает процедуру Z, то при выполнении Z в стеке будут фреймы для X, Y и Z. Каждый поток будет, как правило, вызывать различные процедуры и, следовательно, иметь среду выполнения, отличающуюся от среды выполнения других потоков. Поэтому каждому потоку нужен собственный стек.

В случае многопоточности, процесс обычно начинается с использования одного потока. Этот поток может создавать новые потоки, вызвав библиотечную процедуру **`thread_create()`**.

В параметре **`thread_create()`** обычно указывается имя процедуры, которая запускается в новом потоке.

Создающий поток обычно получает идентификатор потока, который дает имя новому потоку.

Когда поток завершает свою работу, выход из него может быть осуществлен за счет вызова библиотечной процедуры, к примеру **`thread_exit()`**. После этого он прекращает свое существование и больше не фигурирует в работе планировщика.

В некоторых случаях какой-нибудь поток для выполнения выхода может ожидать выхода из какого-нибудь другого (указанного) потока, для чего используется **`thread_join()`**.

Эта процедура блокирует вызывающий поток до тех пор, пока не будет осуществлен выход из другого (указанного) потока. В этом отношении создание и завершение работы потока очень похожи на создание и завершение работы процесса при использовании примерно одних и тех же параметров.

Другой распространенной процедурой, вызываемой потоком, является **`thread_yield()`**.

Она позволяет потоку добровольно уступить центральный процессор для выполнения другого потока. Важность вызова такой процедуры обусловливается отсутствием прерывания по таймеру, которое есть у процессов и благодаря которому фактически задается режим многозадачности.

Для потоков важно проявлять вежливость и время от времени добровольно уступать центральный процессор, чтобы дать возможность выполнения другим потокам.

Есть и другие вызываемые процедуры, которые позволяют одному потоку ожидать, пока другой поток не завершит какую-нибудь работу, а этому потоку — оповестить о том, что он завершил определенную работу, и т. д.

Потоки (нити) POSIX

Чтобы предоставить возможность создания переносимых многопоточных программ, в отношении потоков институтом IEEE был определен стандарт IEEE standard 1003.1c.

Определенный в нем пакет, касающийся потоков, называется **Pthreads**. Он поддерживается большинством UNIX-систем. В стандарте определено более 60 вызовов функций.

Вызовы, связанные с потоком	Описание
pthread_create()	Создание нового потока
pthread_exit()	Завершение работы вызвавшего потока
pthread_join()	Ожидание выхода из указанного потока
pthread_yield()	Освобождение центрального процессора, позволяющее выполняться другому потоку
pthread_attr_init()	Создание и инициализация структуры атрибутов потока
pthread_attr_destroy()	Удаление структуры атрибутов потока

Все потоки ***Pthreads*** имеют определенные свойства. У каждого потока есть свои идентификатор, набор регистров (включая счетчик команд) и набор атрибутов, которые сохраняются в определенной структуре. Атрибуты включают размер стека, параметры планирования и другие элементы, необходимые при использовании потока.

Новый поток создается с помощью вызова функции **pthread_create()**. В качестве значения функции возвращается идентификатор только что созданного потока. Этот вызов намеренно сделан очень похожим на системный вызов **fork()** (за исключением параметров), а идентификатор потока играет роль **PID**, главным образом для идентификации ссылок на потоки в других вызовах.

Когда поток заканчивает возложенную на него работу, он может быть завершен путем вызова функции **pthread_exit()**. Этот вызов останавливает поток и освобождает пространство, занятое его стеком.

Зачастую потоку необходимо перед продолжением выполнения ожидать окончания работы и выхода из другого потока. Для этого ожидающий поток вызывает функцию **pthread_join()**. В качестве параметра этой функции передается идентификатор потока, чьего завершения следует ожидать.

Иногда бывает так, что поток не является логически заблокированным, но «считает», что проработал достаточно долго, и намеревается дать шанс на выполнение другому потоку. Для этого используется вызов **pthread_yield()**.

Для процессов подобного вызова не существует, поскольку предполагается, что процессы сильно конкурируют друг с другом и каждый из них требует как можно больше времени центрального процессора. Но поскольку потоки одного процесса, как правило, пишутся одним и тем же программистом, то он может сделать так, чтобы они давали друг другу шанс на выполнение.

Два следующих вызова функций, связанных с потоками, относятся к атрибутам.

Функция **pthread_attr_init()** создает структуру атрибутов, связанную с потоком, и инициализирует ее значениями по умолчанию. Эти значения (например, приоритет) могут быть изменены за счет работы с полями в структуре атрибутов.

И наконец, функция **pthread_attr_destroy()** удаляет структуру атрибутов, принадлежащую потоку, освобождая память, которую она занимала. На поток, который использовал данную структуру, это не влияет, и он продолжает свое существование.

Пример программы, использующей потоки

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10

// выводит идентификатор потока, а затем выходит
void *print_hello_world(void *tid) {
    printf("    поток No %lu\n", (size_t)tid);
    pthread_exit(NULL);
}

// Основная программа создает 10 потоков, а затем осуществляет выход.
int main(int argc, char *argv[]) {

    pthread_t threads[NUMBER_OF_THREADS];
    for (size_t i = 0; i < NUMBER_OF_THREADS; i++) {
        printf("Создание потока No %lu\n", i);
        int status = pthread_create(&threads[i], // сюда сохраняется ID потока
                                    NULL,        // атрибуты по умолчанию
                                    print_hello_world, (void *)i); // процедура
        if (status != 0) {
            printf("Ахтунг, функция pthread_create вернула код ошибки %d\n", status);
            exit(-1);
    }   }
    exit(0);
}
```

Компиляция, сборка и запуск

```
$ gcc -Wall -W -std=c11 -Wextra -Wno-unused-parameter -pthread -o threads threads.c
$ ./threads
Создание потока No 0
Создание потока No 1
    поток No 0
Создание потока No 2
    поток No 1
    поток No 2
Создание потока No 3
Создание потока No 4
    поток No 3
Создание потока No 5
    поток No 4
Создание потока No 6
    поток No 5
Создание потока No 7
    поток No 6
Создание потока No 8
    поток No 7
Создание потока No 9
    поток No 8
    поток No 9
```

Порядок, в котором выводятся различные сообщения, не определен и при нескольких запусках программы может быть разным.

Реализация потоков в пользовательском пространстве

Есть два основных места реализации набора потоков — в пользовательском пространстве и в ядре. Возможна и гибридная реализация.

Первый способ — это поместить весь набор потоков в пользовательском пространстве.

При этом ядру ничего не известно об этом наборе.

Первое и самое очевидное преимущество состоит в том, что набор потоков на пользовательском уровне может быть реализован в операционной системе, которая не поддерживает потоки.

Под эту категорию подпадают все операционные системы, даже те, которые еще находятся в разработке. При этом подходе потоки реализованы с помощью библиотеки.

У всех этих реализаций одна и та же общая структура — потоки запускаются поверх ***системы поддержки исполнения программ*** (run-time system), которая представляет собой набор процедур, управляющих потоками.

Четыре из них: **`pthread_create()`**, **`pthread_exit()`**, **`pthread_join()`** и **`pthread_yield()`** — мы уже рассмотрели, но обычно в наборе есть и другие процедуры.

Когда потоки управляются в пользовательском пространстве, каждому процессу необходимо иметь собственную ***таблицу потоков***, чтобы отслеживать потоки, имеющиеся в этом процессе.

Эта таблица является аналогом таблицы процессов, имеющейся в ядре, за исключением того, что в ней содержатся лишь свойства, принадлежащие каждому потоку, такие как счетчик команд потока, указатель стека, регистры, состояние и т. д.

Таблица потоков управляется системой поддержки исполнения программ. Когда поток переводится в состояние готовности или блокируется, информация, необходимая для возобновления его выполнения, сохраняется в таблице потоков, точно так же, как ядро хранит информацию о процессах в таблице процессов.

Когда поток совершает какие-то действия, которые могут вызвать его локальную блокировку, например ожидание, пока другой поток его процесса не завершит какую-нибудь работу, он вызывает процедуру системы поддержки исполнения программ. Эта процедура проверяет, может ли поток быть переведен в состояние блокировки.

Если может, она сохраняет регистры потока (то есть собственные регистры) в таблице потоков, находит в таблице поток, готовый к выполнению, и перезагружает регистры машины сохраненными значениями нового потока.

Как только будут переключены указатель стека и счетчик команд, автоматически возобновится выполнение нового потока. Если машине дается инструкция сохранить все регистры и следующая инструкция — загрузить все регистры, то полное переключение потока может быть осуществлено за счет всего лишь нескольких инструкций. Переключение потоков, осуществленное таким образом, по крайней мере на порядок, а может быть, и больше, быстрее, чем перехват управления ядром, что является веским аргументом в пользу набора потоков, реализуемого на пользовательском уровне.

Но у потоков есть одно основное отличие от процессов. Когда поток на время останавливает свое выполнение, например когда он вызывает **`thread_yield()`**, код процедуры **`thread_yield()`** может самостоятельно сохранять информацию о потоке в таблице потоков. Более того, он может затем вызвать планировщик потоков, чтобы тот выбрал для выполнения другой поток. Процедура, которая сохраняет состояние потока, и планировщик — это всего лишь локальные процедуры, поэтому их вызов намного более эффективен, чем вызов ядра. Помимо всего прочего, не требуется перехват управления ядром, осуществляемый инструкцией **`trap/int/syscall`**, не требуется переключение контекста, кэш в памяти не нужно сбрасывать на диск и т. д. Благодаря этому планировщик потоков работает очень быстро.

У потоков, реализованных на пользовательском уровне, есть и другие преимущества.

Они позволяют каждому процессу иметь ***собственные настройки алгоритма планирования***.

Например, для некоторых приложений, которые имеют поток сборщика мусора, есть еще один плюс — им не следует беспокоиться о потоках, остановленных в неподходящий момент.

Эти потоки также лучше масштабируются, поскольку потоки в памяти ядра безусловно требуют в ядре пространства для таблицы и стека, что при очень большом количестве потоков может вызвать затруднения.

Но несмотря на лучшую производительность, у потоков, реализованных на пользовательском уровне, есть ряд существенных проблем.

Первая из них — сложность использования блокирующих системных вызовы. Например, поток считывает информацию с клавиатуры. Нельзя разрешить потоку осуществить настоящий системный вызов, поскольку это остановит выполнение всех потоков.

Одна из главных целей организации потоков в первую очередь состояла в том, чтобы позволить каждому потоку использовать блокирующие вызовы, но при этом предотвратить влияние одного заблокированного потока на выполнение других потоков.

Работая с блокирующими системными вызовами, довольно трудно понять, как можно достичь этой цели без особого труда.

Все системные вызовы могут быть изменены и превращены в неблокирующие (например, считывание с клавиатуры будет просто возвращать нуль байтов, если в буфере на данный момент отсутствуют символы), но изменения, которые для этого необходимо внести в операционную систему, не вызывают энтузиазма. Кроме того, одним из аргументов за использование потоков, реализованных на пользовательском уровне, было именно то, что они могут выполняться под управлением ***существующих*** операционных систем.

Вдобавок ко всему изменение семантики системного вызова **`read()`** потребует изменения множества пользовательских программ.

В том случае, если есть возможность заранее сообщить, будет ли вызов блокирующим, существует и другая альтернатива. В большинстве версий UNIX существует системный вызов **`select()`**, позволяющий сообщить вызывающей программе, будет ли предполагаемый системный вызов **`read()`** блокирующим. Если такой вызов имеется, библиотечная процедура **`read()`** может быть заменена новой процедурой, которая сначала осуществляет вызов процедуры **`select()`** и только потом — вызов **`read()`**, если он безопасен (то есть не будет выполнять блокировку). Если вызов **`read()`** будет блокирующим, он не осуществляется.

Вместо этого запускается выполнение другого потока. В следующий раз, когда система поддержки исполнения программ получает управление, она может опять проверить, будет ли на этот раз вызов **`read()`** безопасен. Для реализации такого подхода требуется переписать некоторые части библиотеки системных вызовов, что нельзя рассматривать в качестве эффективного и элегантного решения, но все же это тоже один из вариантов.

Код, который помещается вокруг системного вызова с целью проверки, называется ***конвертом*** (**jacket**), или оболочкой, или ***оберткой*** (wrapper).

С проблемой блокирующих системных вызовов несколько перекликается проблема ошибки отсутствия страницы. В одно и то же время в оперативной памяти находятся не все программы. Если программа вызывает инструкции (или переходит к инструкциям), отсутствующие в памяти, возникает ошибка обращения к отсутствующей странице, в результате чего операционная система обращается к диску и получает отсутствующие инструкции (и их соседей). Это называется ошибкой вызова отсутствующей страницы (#PF). Процесс блокируется до тех пор, пока не будет найдена и считана необходимая инструкция. Если ошибка обращения к отсутствующей странице возникает при выполнении потока, ядро, которое даже не знает о существовании потоков, как и следовало ожидать, блокирует весь процесс до тех пор, пока не завершится дисковая операция ввода-вывода, даже если другие потоки будут готовы к выполнению.

Использование набора потоков, реализованного на пользовательском уровне, связано еще с одной проблемой — **если начинается выполнение одного из потоков, то никакой другой поток, принадлежащий этому процессу, не сможет выполняться до тех пор, пока первый поток добровольно не уступит центральный процессор.**

В рамках единого процесса нет прерываний по таймеру, позволяющих планировать работу процессов по круговому циклу (поочередно). Если поток не войдет в систему поддержки выполнения программ по доброй воле, у планировщика не будет никаких шансов на работу.

Проблему бесконечного выполнения потоков можно решить также путем передачи управления системе поддержки выполнения программ за счет запроса сигнала (прерывания) по таймеру с периодичностью один раз в секунду, но для программы это далеко не самое лучшее решение. Возможность периодических и довольно частых прерываний по таймеру предоставляется не всегда, но даже если она и предоставляется, общие издержки могут быть весьма существенными.

Другой наиболее сильный аргумент против потоков, реализованных на пользовательском уровне, состоит в том, что программистам потоки обычно требуются именно в тех приложениях, где они часто блокируются, как, к примеру, в многопоточном веб-сервере.

Реализация потоков в ядре

В этом случае система поддержки исполнения программ не нужна.

Нет и таблицы потоков в каждом процессе. Вместо этого у ядра есть таблица потоков, в которой отслеживаются все потоки, имеющиеся в системе. Когда потоку необходимо создать новый или уничтожить существующий поток, он обращается к ядру, которое и создает или разрушает путем обновления таблицы потоков в ядре.

В таблице потоков, находящейся в ядре, содержатся регистры каждого потока, состояние и другая информация. Вся информация аналогична той, которая использовалась для потоков, создаваемых на пользовательском уровне, но теперь она содержится в ядре, а не в пространстве пользователя (внутри системы поддержки исполнения программ).

Эта информация является подмножеством той информации, которую поддерживают традиционные ядра в отношении своих однопоточных процессов, то есть подмножеством состояния процесса. Вдобавок к этому ядро поддерживает также традиционную таблицу процессов с целью их отслеживания.

Поскольку создание и уничтожение потоков в ядре требует относительно более весомых затрат, некоторые системы с учетом складывающейся ситуации применяют более правильный подход и используют свои потоки повторно. При уничтожении потока он помечается как неспособный к выполнению, но это не влияет на его структуру данных, имеющуюся в ядре. Чуть позже, когда должен быть создан новый поток, вместо этого повторно активируется старый поток, что приводит к экономии времени.

Для потоков, реализованных на уровне ядра, не требуется никаких новых, неблокирующих системных вызовов. Более того, если один из выполняемых потоков столкнется с ошибкой обращения к отсутствующей странице, ядро может с легкостью проверить наличие у процесса любых других готовых к выполнению потоков и при наличии таковых запустить один из них на выполнение, пока будет длиться ожидание извлечения запрошенной страницы с диска.

Главный недостаток потоков в ядре состоит в весьма существенных затратах времени на системный вызов, поэтому если операции над потоками (создание, удаление и т. п.) выполняются довольно часто, это влечет за собой более существенные издержки.

Потоки, создаваемые на уровне ядра, хоть и позволяют решить ряд проблем, но справиться со всеми существующими проблемами они не в состоянии.

Например, что делать при разветвлении многопоточного процесса? Будет ли у нового процесса столько же потоков, сколько у старого, или только один поток?

Во многих случаях наилучший выбор зависит от того, выполнение какого процесса запланировано следующим. Если он собирается вызвать **exec . . ( )**, чтобы запустить новую программу, то, наверное, правильным выбором будет наличие только одного потока.

Но если он продолжит выполнение, то лучше всего было бы, наверное, воспроизвести все имеющиеся потоки.

Другой проблемой являются сигналы. В классической модели **сигналы посылаются процессам**, а не потокам.

Какой из потоков должен обработать поступающий сигнал? Может быть, потоки должны зарегистрировать свои интересы в конкретных сигналах, чтобы при поступлении сигнала он передавался потоку, который заявил о своей заинтересованности в этом сигнале?

Тогда возникает вопрос: что будет, если на один и тот же сигнал зарегистрировались два или более двух потоков?

И это только две проблемы, создаваемые потоками, а ведь на самом деле их значительно больше.

Гибридная реализация

В попытках объединить преимущества создания потоков на уровне пользователя и на уровне ядра была исследована масса различных путей. Один из них заключается в использовании потоков на уровне ядра, а затем нескольких потоков на уровне пользователя в рамках некоторых или всех потоков на уровне ядра.

При использовании такого подхода программист может определить, сколько потоков использовать на уровне ядра и на сколько потоков разделить каждый из них на уровне пользователя.

Эта модель обладает максимальной гибкостью.

При таком подходе ядру известно ***только*** о потоках самого ядра, работу которых оно и планирует. У некоторых из этих потоков могут быть несколько потоков на пользовательском уровне.

Создание, удаление и планирование выполнения этих потоков осуществляется точно так же, как и у пользовательских потоков, принадлежащих процессу, запущенному под управлением операционной системы, не способной на многопоточную работу. В этой модели каждый поток на уровне ядра обладает определенным набором потоков на уровне пользователя, которые используют его по очереди.

Превращение однопоточного кода в многопоточный

Многие из существующих программ создавались под однопоточные процессы.

Превратить их в многопоточные куда сложнее, чем может показаться на первый взгляд.

Код потока, как и код процесса, обычно содержит несколько процедур.

У этих процедур могут быть локальные и глобальные переменные, а также параметры.

Локальные переменные и параметры проблем не создают, проблемы возникают с теми переменными, которые носят глобальный характер для потока, но не для всей программы.

Глобальность этих переменных заключается в том, что их использует множество процедур внутри потока (поскольку они могут использовать любую глобальную переменную), но другие потоки логически должны их оставить в покое.

Рассмотрим в качестве примера переменную **`errno`**, поддерживаемую языком С.

Когда процесс (или поток) осуществляет системный вызов, терпящий неудачу, код ошибки помещается в **`errno`**.

Например, поток 1 выполняет системный вызов **`access()`**, чтобы определить, разрешен ли доступ к конкретному файлу. Операционная система возвращает детали ответа в глобальной переменной **`errno`**.

После возвращения управления потоку 1, но перед тем, как он получает возможность прочитать значение **`errno`**, планировщик решает, что поток 1 на данный момент времени вполне достаточно использовал время центрального процессора и следует переключиться на выполнение потока 2.

Поток 2 выполняет вызов **`open()`**, который терпит неудачу, что вызывает переписывание значения переменной **`errno`**, и код **`access()`** первого потока утрачивается навсегда. Когда чуть позже возобновится выполнение потока 1, он считает неверное значение и поведет себя некорректно.

Существуют разные способы решения этой проблемы. Можно вообще запретить использование глобальных переменных. Какой бы заманчивой ни была эта идея, она вступает в конфликт со многими существующими программами. Другой способ заключается в назначении каждому потоку собственных глобальных переменных.

В этом случае у каждого потока есть своя закрытая копия **`errno`** и других глобальных переменных, позволяющая избежать возникновения конфликтов. В результате такого решения создается новый уровень области определения, где переменные видны всем процедурам потока (но не видны другим потокам), вдобавок к уже существующим областям определений, где переменные видны только одной процедуре и где переменные видны из любого места программы.

Однако доступ к закрытым глобальным переменным несколько затруднен, поскольку большинство языков программирования имеют способ выражения локальных и глобальных переменных, но не содержат промежуточных форм.

Есть возможность распределить часть памяти для глобальных переменных и передать ее каждой процедуре потока в качестве дополнительного параметра. Решение не самое изящное, но работоспособное.

В качестве альтернативы можно ввести новые библиотечные процедуры для создания, установки и чтения глобальных переменных, видимых только внутри потока. Первый вызов процедуры может иметь следующий вид: **`create_global("bufptr")`**;

Он выделяет хранилище для указателя по имени **`bufptr`** в динамически распределяемой области памяти или в специальной области памяти, зарезервированной для вызывающего потока. Независимо от того, где именно размещено хранилище, к глобальным переменным имеет доступ только вызывающий поток. Если другой поток создает глобальную переменную с таким же именем, она получает другое место хранения и не конфликтует с уже существующей переменной.

Для доступа к глобальным переменным нужны два вызова — один для записи, а другой для чтения. Процедура для записи может иметь следующий вид:

```
set global("bufptr", &buf);
```

Она сохраняет значение указателя в хранилище, ранее созданном вызовом процедуры **`create_global()`**. Процедура для чтения глобальной переменной может иметь следующий вид:

```
bufptr = read_global("bufptr");
```

Она возвращает адрес для доступа к данным, хранящимся в глобальной переменной.

Реентерабельность

Другой проблемой, возникающей при превращении однопоточной программы в многопоточную, является отсутствие возможности повторного входа во многие библиотечные процедуры.

То есть они не создавались с расчетом на то, что каждая отдельно взятая процедура будет вызываться повторно еще до того, как завершился ее предыдущий вызов.

К примеру, отправка сообщения по сети может быть запрограммирована на то, чтобы предварительно собрать сообщение в фиксированном буфере внутри библиотеки, а затем для его отправки осуществить перехват управления ядром. Если один поток собрал свое сообщение в буфере, а затем прерывание по таймеру вызвало переключение на выполнение второго потока, который тут же переписал буфер своим собственным сообщением, мы имеем проблему.

Для эффективного устранения всех этих проблем потребуется переписать всю библиотеку. А это далеко не самое простое занятие с реальной возможностью внесения труднообнаруживаемых ошибок.

Другим решением может стать предоставление каждой процедуре оболочки, которая устанавливает бит, отмечающий, что библиотека уже используется.

Любая попытка другого потока воспользоваться библиотечной процедурой до завершения предыдущего вызова будет заблокирована. Хотя этот подход вполне осуществим, он существенно снижает потенциальную возможность параллельных вычислений.

Сигналы

Некоторые сигналы по своей логике имеют отношение к потокам, а некоторые не имеют к ним никакого отношения. К примеру, если поток осуществляет системный вызов **`alarm()`**, появляется смысл направить результирующий сигнал к вызывающему потоку. Если потоки целиком реализованы в пользовательском пространстве, ядро даже не знает о потоках и вряд ли сможет направить сигнал к нужному потоку.

Дополнительные сложности возникают в том случае, если у процесса на данный момент есть лишь один необработанный сигнал от **`alarm()`** и несколько потоков осуществляют системный вызов **`alarm()`** независимо друг от друга.

Другие сигналы, например прерывания клавиатуры, не имеют определенного отношения к потокам. Кто из потоков должен их перехватывать?

Одному потоку может потребоваться перехватить конкретный сигнал (например, когда пользователь нажмет CTRL+C), а другому потоку этот сигнал понадобится для завершения процесса.

Подобная ситуация может сложиться, если в одном или нескольких потоках выполняются стандартные библиотечные процедуры, а в других — процедуры, созданные пользователем.

Стек

Еще одна проблема, создаваемая потоками, — **управление стеком**.

Во многих системах при переполнении стека процесса ядро автоматически предоставляет ему дополнительное пространство памяти. Когда у процесса несколько потоков, у него должно быть и несколько стеков. Если ядро ничего не знает о существовании этих стеков, оно не может автоматически наращивать их пространство при ошибке стека.

Фактически оно даже не сможет понять, что ошибка памяти связана с разрастанием стека какого-нибудь потока.