## Лабораторная работа № 1

## Основы многопоточного программирования. Синхронизация и взаимодействие процессов и потоков

**Цель работы**: изучение объектов, функций и системных вызовов ОС семейства UNIX (FreeBSD, Linux) и WIN32 API, предназначенных для синхронизации и взаимодействия процессов и потоков. Получение практических навыков разработки многопоточных приложений.

## Краткие теоретические сведения

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. *Процесс* -- абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

В традиционной модели Unix процесс, которому нужно, чтобы ка­кое-то действие было выполнено не им самим, порождает дочерний процесс вы­зовом fork. Хотя эта парадигма хорошо работает на протяжении многих лет, вызов fork обладает некоторыми недостатками:

- вызов fork ресурсоемок. Память копируется от родительского процесса к до­чернему, копируются все дескрипторы и т. д. Существующие реализации ис­пользуют метод копирования при записи (copy-on-write), что исключает не­обходимость копирования адресного пространства родительского процесса, пока оно не понадобится клиенту, но, несмотря на эту оптимизацию, вызов fork остается ресурсоемким;

- для передачи информации между родительским и дочерним процессами не­обходимо использовать одну из форм IPC после вызова fork. Передать ин­формацию дочернему процессу легко: это можно сделать до вызова fork. Од­нако передать ее обратно может быть достаточно сложно.

Потоки помогают решить обе проблемы. Часто они называются «облегченны­ми процессами» (lightweight processes), поскольку поток проще, чем процесс. Создание потока может занимать по времени меньше одной десятой создания процесса.

Все потоки одного процесса совместно используют его глобальные перемен­ные, поэтому им легко обмениваться информацией, но это приводит к необходи­мости синхронизации. Однако общими становятся не только глобальные пере­менные. Все потоки одного процесса разделяют:

* инструкции процесса;
* большую часть данных;
* открытые файлы (дескрипторы);
* обработчики сигналов и вообще настройки для работы с сигналами;
* текущий рабочий каталог;
* идентификатор пользователя и группы.

Однако каждый поток имеет свой собственный:

* идентификатор потока;
* набор регистров, включая PC и указатель стека;
* стек (для локальных переменных и адресов возврата);
* errno;
* маску сигналов;
* приоритет.

Системные вызовы управления процессами и потоками в ОС семейства UNIX

| Функция | Назначение |
| --- | --- |
| fork() | Создание нового процесса |
| waitpid() | Ожидание завершения дочернего процесса |
| execve | Преобразование копии вызывающего процесса в новый процесс |
| exit() | Завершение процесса |
| sidaction | Определение действия по обработке сигнала |
| sigreturn | Возврат управления по окончании обработки сигнала |
| sigprocmask() | Проверка и/или изменение маски сигнала |
| sigpending() | Получение либо установка блокированного сигнала |
| sigsuspend() | Временное изменение маски блокированного сигнала с последующим ожиданием получения сигнала; при выходе восстанавливается предыдущее значение маски |
| kill() | Посылка сигнала процессу |
| alarm() | Установка таймера в ожидание сигнала SIGALRM |
| pause() | Приостановка выполнения процесса до следующего сигнала |
| pthread\_create() | Создание нового потока |
| pthread\_exit() | Завершение потока |
| pthread\_join() | Приостановка выполнения процесса до окончания указанного процесса |
| pthread\_mutex\_init() | Создание взаимоисключение |
| pthread\_mutex\_destroy() | Уничтожение взаимоисключения |
| pthread\_mutex\_lock() | Блокировка взаимоисключения |
| pthread\_mutex\_unlock() | Разблокирование взаимоисключения |
| pthread\_cond\_init() | Создание условной переменной |
| pthread\_cond\_destroy() | Уничтожение условной переменной |
| pthread\_cond\_wait | Автоматическая блокировка текущего потока ожиданием условной переменной |
| pthread\_cond\_signal | Разблокирование одного потока, ожидающего условную переменную |

Более подробную информацию по каждой из функций можно получить по команде   
man имя\_функции.

**Основные функции для работы с потоками: создание и завершение**

Функция pthread\_create

При запуске программы вызовом exec создается единственный поток, называе­мый начальным потоком, или главным (initial thread). Добавочные потоки созда­ются вызовом pthread\_create:

#include <pthread.h>

int pthread\_create(pthread\_t \*tid. const pthread\_attr\_t \*attr. void \*(\*func) (void \*). void \*arg);

/\* Возвращает 0 в случае успешного завершения, положительное значение Еххх - в случае ошибки \*/

Каждый поток процесса обладает собственным идентификатором потока, ко­торый имеет тип pthread\_t. При успешном создании нового потока его идентифи­катор возвращается через указатель tid.

Каждый поток обладает некоторым количеством атрибутов: приоритетом, на­чальным размером стека, признаком демона и т. п. При создании потока эти ат­рибуты могут быть указаны с помощью переменной типа pthread\_attr\_t, значе­ние которой имеет более высокий приоритет, чем значения по умолчанию. Обычно мы используем значения по умолчанию. При этом аргумент attr является нуле­вым указателем.

Наконец, при создании потока мы должны указать функцию, которую он бу­дет выполнять, — начальную функцию потока (thread start function). Поток за­пускается вызовом этой функции и завершается либо явно (вызовом pthread\_exit), либо неявно (возвратом из этой функции). Адрес функции указывается в аргументе func, и вызывается она с единственным аргументом — указателем arg. Если функции нужно передать несколько аргументов, следует упаковать их в структу­ру и передать ее адрес в качестве единственного аргумента начальной функции.

Обратите внимание на объявления func и arg. Функция принимает один аргу­мент — указатель типа void, и возвращает один аргумент — такой же указатель. Это дает нам возможность передать потоку указатель на что угодно и получить в ответ такой же указатель.

Функции Posix для работы с потоками обычно возвращают 0 в случае успеш­ного завершения работы и ненулевое значение в случае ошибки. В отличие от большинства системных функций, возвращающих -1 в случае ошибки и уста­навливающих значение errno равным коду ошибки, функции Pthread возвращают положительный код ошибки. Например, если pthread\_create не сможет создать новый поток из-за превышения системного ограничения на потоки, эта функция вернет значение EAGAIN. Функции Pthread не устанавливают значение переменной errno. Несоответствий при их вызове не возникает, поскольку ни один из кодов ошибок не имеет нулевого значения (<sys/errno.h>).

Функция pthread\_join

Мы можем ожидать завершения какого-либо процесса, вызвав pthread\_join. Срав­нивая потоки с процессами Unix, можно сказать, что pthread\_create аналогична fork, a pthread\_join — waitpid:

#include <pthread.h>

int pthread\_join(pthread\_t tid. void \*\*status);

/\* Возвращает 0 в случае успешного завершения, положительное значение Еххх - в случае ошибки \*/

Мы должны указать идентификатор потока, завершения которого ожидаем. К сожалению, невозможно задать режим ожидания завершения нескольких по­токов (аналога waitpid с идентификатором процесса -1 нет).

Если указатель status ненулевой, возвращаемое потоком значение (указатель на объект) сохраняется в ячейке памяти, на которую указывает status.

Функция pthread\_self

У каждого потока имеется свой идентификатор, уникальный в пределах данного процесса. Идентификатор возвращается pthread\_create и используется при вызо­ве pthread\_join. Поток может узнать свой собственный идентификатор вызовом pthread\_self:

#include <pthread.h>

pthread\_t pthread\_self(void);

/\* Возвращает идентификатор вызвавшего потока \*/

Вызов pthread\_self является аналогом getpid для процессов Unix.

Функция pthread\_detach

Поток может являться как присоединяемым (по умолчанию), так и отсоединен­ным. При завершении присоединяемого потока его идентификатор и статус за­вершения сохраняются до тех пор, пока какой-либо другой поток данного про­цесса не вызовет pthread\_join. Отсоединенный поток функционирует аналогично процессу-демону. После его завершения все ресурсы освобождаются. Никакой другой поток не может ожидать его завершения. Если имеется необходимость ожидания одним потоком завершения другого, лучше оставить последний при­соединяемым.

Функция pthread\_detach делает данный поток отсоединенным:

#include <pthread.h>

int pthread\_detach(pthread\_t tid);

/\* Возвращает 0 в случае успешного завершения, положительное значение Еххх в случае ошибки \*/

Эта функция вызывается потоком при необходимости изменить собственный статус в форме

pthread\_detach(pthread\_self());

Функция pthread\_exit

Одним из способов завершения потока является вызов pthread\_exit:

#include <pthread.h>

void pthread\_exit(void \*status):

/\* ничего не возвращает вызвавшему потоку \*/

Если поток не является отсоединенным, его идентификатор и статус завер­шения сохраняются для возвращения другому потоку, который может вызвать pthread\_join.

Указатель status не должен быть установлен на локальный объект вызвавшего потока (типа автоматической переменной), поскольку этот объект уничтожается при завершении потока.

Поток может быть завершен двумя другими способами:

-- начальная функция потока (третий аргумент pthread\_create) может вызвать return. Поскольку эта функция должна объявляться как возвращающая ука­затель на тип void, это возвращаемое значение становится статусом заверше­ния потока;

-- функция main процесса может завершить работу или один из потоков может вызвать exit или \_exit. При этом процесс завершает работу немедленно, вмес­те со всеми своими потоками.

Функции WIN32 API для управления процессами и потоками

| Функция | Назначение |
| --- | --- |
| CreateProcess() | Создать новый процесс |
| CreateThread() | Создать новый поток |
| ExitProcess() | Завершить текущий процесс |
| ExitThread() | Завершить текущий поток |
| SetPriorityClass() | Задать класс приоритета для процесса |
| SetThreadPriority() | Задать приоритет для потока |
| CreateSemaphore() | Создать семафор |
| CreateMutex() | Создать взаимоисключение |
| OpenSemaphore() | Открыть семафор |
| OpenMutex() | Открыть взаимоисключение |
| WaitForSingleObject() | Проверка текущего состояния заданного объекта. Если состояние – несигнализирующее, текущий поток переходит в состояние ожидания |
| WaitForMultiplyObjects() | Проверка текущего состояния объектов синхронизации из заданного массива |
| PulseEvent() | Установка заданного объекта-события в сигнализирующее состояние с последующим переводом его а несигнализирующее состояние после освобождения заданного числа ожидающих потоков |
| ReleaseMutex() | Освобождение взаимоисключения |
| ReleaseSemaphore() | Увеличение счетчика семафора на заданное число |
| EnterCriticalSection() | Ожидание входа в критическую секцию |
| LeaveCriticalSection() | Освобождение критической секции |

Более подробную информацию по каждой из функций можно получить из библиотеки MSDN.

Пример WIN32 - программы, выполняющей создание потока.

Функция [**CreateThread**](ms-help://MS.MSDNQTR.2003OCT.1033/dllproc/base/createthread.htm) выполняет создание нового потока в текущем процессе. Для создаваемого процесса должен быть определен стартовый адрес выполняемого им кода (имя стартовой функции). Стартовая функция потока должна иметь единственный аргумент и возвращать значение типа **DWORD**. В процессе может одновременно выполняться несколько потоков с одной и той же стартовой функцией.

#include <windows.h>

#include <conio.h>

DWORD WINAPI ThreadFunc( LPVOID lpParam )

{

char szMsg[80];

wsprintf( szMsg, "Parameter = %d.", \*(DWORD\*)lpParam );

MessageBox( NULL, szMsg, "ThreadFunc", MB\_OK );

return 0;

}

VOID main( VOID )

{

DWORD dwThreadId, dwThrdParam = 1;

HANDLE hThread;

char szMsg[80];

hThread = CreateThread(

NULL, // атрибуты безопасности по умолчанию

0, // размер стека по умолчанию

ThreadFunc, // стартовая функция

&dwThrdParam, // аргумент стартовой функции

0, // флаги создания потока по умолчанию

&dwThreadId); // возвращаемый идентификатор потока

// Проверка возвращенного значения

if (hThread == NULL)

{

wsprintf( szMsg, "ошибка CreateThread." );

MessageBox( NULL, szMsg, "main", MB\_OK );

}

else

{

\_getch();

CloseHandle( hThread );

}

}

В рассмотренном примере для простоты в стартовую функцию потока предается указатель на некоторое значение. Это может быть указатель на любой тип данных или структуру. Можно также опустить его, передав NULL.

Передача указателя на локальную переменную небезопасно, если создающий поток завершает свою работу до старта следующего, поскольку указатель становится недействительным. Необходимо либо передавать указатель на динамически выделяемую память, либо заставить создающий поток дождаться, пока созданный поток не завершит свою работу. Данные могут быть переданы в создаваемый поток также и через глобальные переменные. В последнем случае, как правило, необходима синхронизация доступа к переменным со стороны множества потоков.

В процессах, создающих множество потоков с одной и той же стартовой функцией использование глобальных переменных, по меньшей мере, неудобно. Например, рассмотрим процесс, позволяющий открывать несколько файлов одновременно, и создающий новый поток с одной и той же стартовой функцией для каждого из файлов. Создающий поток должен передавать уникальную информацию (например, имя открываемого файла) в каждый из экземпляров порождаемого потока в виде аргумента. Использование глобальной переменной в данном случае невозможно, следует использовать динамически выделяемый буфер для хранения строки.

**Синхронизация процессов и потоков**

В среде, позволяющей исполнять несколько процессов (потоков) одновременно, очень важно синхронизировать их деятельность. Для этого операционные системы, предлагают несколько синхронизирующих объектов: критические разделы (critical sections), объекты mutex (сокращение от mutual exclusion), семафо­ры и события.

Критические разделы (только WIN32)

*Критический раздел* (critical section) — объект, к которому должен обратиться по­ток перед получением эксклюзивного доступа к каким-либо общим данным. Сре­ди синхронизирующих объектов критические разделы наиболее просты, но при­менимы для синхронизации потоков лишь в пределах одного процесса. Они дают возможность сделать так, чтобы единовременно только один поток получил доступ к определенному региону данных.

При необходимости синхронизи­ровать приложения со специфическими событиями, возникающими в системе, или с операциями, выполняемыми в других процессах, критические разделы непригодны. Например, при создании дочернего процесса, возможно, придется сделать так, чтобы родительский про­цесс ожидал его завершения и только потом продолжал свою работу.

Каждый объект синхронизации может находиться в одном из двух состояний: *свободном* (signaled) или *занятом* (nonsignaled). Потоки могут остановиться и ждать, пока какой-либо объект не освободится. Если поток родительского процесса должен ждать завершения дочернего, его можно отправить "в спячку" до осво­бождения объекта ядра, идентифицирующего дочерний процесс. Объекты "процесс" получают статус свободных в момент заверше­ния соответствующих процессов. Это относится и к объектам "поток". Когда по­ток создан и начинает исполнение своего кода, сопоставленный с ним объект ядра "поток" получает статус занятого. При завершении потока соответствую­щий объект ядра освобождается.

Объекты Mutex

Эти объекты аналогичны критическим разделам, но с их помощью можно синхронизировать доступ к данным со стороны не только нескольких потоков одного процесса, но и несколь­ких процессов.

Семафоры

Объекты ядра "семафор" используются для учета ресурсов. При запросе у семафора ресурса операционная система проверяет, свободен ли данный ресурс, и — если свободен — уменьшает счетчик доступных ресурсов, не давая вмешиваться другому потоку. Только после этого система разрешает другому потоку запрашивать какой-либо ресурс.

Для мониторинга занятости ресурсов необходимо со­здать семафор со счетчиком. При этом нужно учитывать, что семафор считается свободным, если его счетчик ресурсов больше нуля, и занятым, если счетчик равен нулю. При каждом вызове из потока специальной функции с передачей ей описателя семафора система проверяет: больше ли нуля счетчик ресурсов у данного семафора. Если да, умень­шает счетчик на единицу и "будит" поток. Если при вызове функции счет­чик семафора оказался обнулен, система оставляет поток неактивным до того, как другой поток освободит семафор (т. е. увеличит его счетчик ресурсов).

Поскольку на счетчик ресурсов семафора могут влиять несколько потоков (или процессов), семафоры — в отличие от критических разделов и объектов mutex — не переда­ются во владение какому-либо потоку. А значит, один поток может ждать объект "семафор" (уменьшив его счетчик ресурсов), а другой поток освободить сема­фор (и тем самым увеличить его счетчик ресурсов).

События

События — самая примитивная разновидность синхронизирующих объектов, отличающаяся от семафоров и объектов mutex. Последние обычно приме­няются для контроля за доступом к данным, а события просто оповещают об окончании какой-либо операции. Существуют два разных типа объектов "событие": со *сбросом вручную* (manual-reset events) и *с автоматическим сбросом* (auto-reset events). Первые используются для оповещения об окончании операции сразу нескольких потоков, вторые — для оповещения единственного потока.

К событиям обычно прибегают в том случае, когда один поток выполняет какую-либо инициализацию, а затем сигнализирует другому потоку, что тот мо­жет работать дальше. Инициализирующий поток переводит объект "событие" в занятое (non-signaled) состояние и приступает к своим операциям. По оконча­нии инициализации поток возвращает событие в свободное (signaled) состоя­ние. В то же время рабочий поток приостанавливает свое исполнение и ждет перехода события в свободное состояние. Как только инициализирующий по­ток просигнализирует событие (т.е. освободит его), рабочий поток "проснется" и продолжит работу.

Например, в процессе исполняются два потока. Первый считывает данные из файла в буферную память и оповещает второй поток, что можно заняться обработкой данных. Закончив обработку, второй поток сигнализирует первому, чтобы тот загрузил новый блок данных из файла и т.д.

Классические проблемы синхронизации

Проблема производителя и потребителя (проблема ограниченного буфера)

Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в этот буфер, а другой, потребитель, считы­вает их оттуда. Задача может быть обобщена на случай ***т*** производителей и ***п***потреби­телей.

Проблема обедающих философов

Сформулирована в 1965г. Дейкстрой.

Условия задачи:

Имеется N философов, сидящих за круглым столом. Перед каждым из них находится тарелка со спагетти. Спагетти настолько скользкие, что каждому философу необходимо две вилки , чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка.

Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений (остальные процессы жизнедеятельности в данном случае несущественны). Когда философ голоден, он пытается получить две вилки, левую и правую, в любом порядке. Если ему удалось получить две вилки, он некоторое время ест, затем кладет вилки и продолжает размышления.

Данная проблема полезна для моделирования процессов, соревнующихся за монопольный доступ к ограниченному количеству ресурсов (например, к устройствам ввода-вывода).

Проблема читателей и писателей

Проблема моделирует доступ к базе данных.

Условие задачи: необходимо обеспечить монопольный доступ к базе данных процессу, выполняющему запись данных, и множественный доступ процессам, выполняющим чтение данных. Задача может быть сформулирована с предоставлением приоритета как процессу-писателю, так и процессам-читателям.

Проблема спящего брадобрея

Условие задачи: имеется один процесс-«брадобрей», его кресло и N стульев для процессов-«клиентов». Если «клиентов нет», «брадобрей» спит в своем кресле. Если в парикмахерскую приходит «клиент», он должен разбудить «брадобрея». Если клиент приходит, и видит, что брадобрей занят, он либо занимает один из стульев (если есть место), либо уходит (если места нет). Необходимо запрограммировать брадобрея и клиентов так, чтобы избежать состояния состязания.

## Задание

Разработать (согласно варианту) решение классических задач синхронизации («обедающих философов», «производителей/потребителей», «читателей/писателей», «спящего брадобрея») с помощью

* 1. взаимоисключений (критических секций);
  2. семафоров;
  3. событий.

Пояснить возможность (или невозможность) применения какого-либо механизма синхронизации для каждого из случаев.

Задачи должны быть решены как в среде ОС семейства UNIX, так и в WIN32.

Задачи должны быть решены для случая многопоточного приложения.

***Отчет по лабораторной работе должен содержать***:

* титульный лист;
* цель работы;
* описание и алгоритм работы каждой из задач;
* фрагменты программного кода из решения каждой задачи, выполняющие создание и синхронизацию потоков.

***Варианты заданий***

| **Вариант** | **«производители/ потребители»** | **«обедающие философы»** | **«читатели/ писатели»** | **«спящий брадобрей»** |
| --- | --- | --- | --- | --- |
| 1 | взаимоисключения (UNIX) | семафоры (UNIX) | события (WIN32) | критические секции (WIN32) |
| 2 | семафоры (UNIX) | события (UNIX) | критические секции (WIN32) | семафоры (WIN32) |
| 3 | события (UNIX) | взаимоисключения (UNIX) | семафоры (WIN32) | события (WIN32) |
| 4 | критические секции (WIN32) | семафоры (WIN32) | события (UNIX) | взаимоисключения (UNIX) |
| 5 | семафоры (WIN32) | события (WIN32) | взаимоисключения (UNIX) | семафоры (UNIX) |
| 6 | события (WIN32) | критические секции (WIN32) | семафоры (UNIX) | события (UNIX) |
| 7 | взаимоисключения (UNIX) | семафоры (WIN32) | события (UNIX) | критические секции (WIN32) |
| 8 | семафоры(UNIX) | события (WIN32) | взаимоисключения (UNIX) | семафоры (WIN32) |
| 9 | события (UNIX) | критические секции (WIN32) | семафоры (UNIX) | события (WIN32) |
| 10 | критические секции (WIN32) | семафоры (UNIX) | события (WIN32) | взаимоисключения (UNIX) |
| 11 | семафоры (WIN32) | события (UNIX) | критические секции (WIN32) | семафоры (UNIX) |
| 12 | события (WIN32) | взаимоисключения (UNIX) | семафоры (WIN32) | события (UNIX) |
| 13 | взаимоисключения (UNIX) | семафоры (WIN32) | события (WIN32) | взаимоисключения (UNIX) |
| 14 | семафоры (UNIX) | события (WIN32) | критические секции (WIN32) | семафоры (UNIX) |
| 15 | события (UNIX) | критические секции (WIN32) | семафоры (WIN32) | события (UNIX) |
| 16 | критические секции (WIN32) | семафоры (UNIX) | события (UNIX) | критические секции (WIN32) |
| 17 | семафоры (WIN32) | события (UNIX) | взаимоисключения (UNIX) | семафоры (WIN32) |
| 18 | события (WIN32) | взаимоисключения (UNIX) | семафоры (UNIX) | события (WIN32) |

***Литература***

1. Стивенс У. UNIX: взаимодействие процессов. – СПб.: Питер, 2002. – 575с.
2. UNIX man pages.
3. MSDN. WIN32 API.