Взаимодействие и синхронизация потоков

Принцип многозадачности по своей сути предполагает, что отдельные процессы (потоки) существуют независимо друг от друга и выполняются асинхронно. Тем не менее, довольно часто возникает необходимость организации взаимодействия процессов (потоков). Это взаимодействие в конечном счете означает обмен информацией между процессами (потоками) и в зависимости от объема передаваемых данных может быть разбито на следующие уровни:

- передача одного сигнального бита для оповещения процесса-получателя о наступлении некоторого события в процессе-отправителе;
- передача некоторой последовательности байтов с помощью специальных линий связи (каналов);
- использование участниками процесса обмена общей области памяти (в общем случае - нескольких областей).

Каждый из этих способов имеет свои особенности. Ясно, что первый способ наиболее простой и безопасный, но степень воздействия на процесс-получатель минимальна. Второй способ позволяет передавать значительно больше информации, является достаточно универсальным, но может потребовать больших временных затрат. Третий способ, наоборот, достаточно быстрый, позволяет обмениваться большими объемами данных, но зато очень опасен, особенно на уровне взаимодействия процессов. Это связано с тем, что ОС каждому процессу выделяет свое адресное пространство, защищенное от воздействия со стороны других процессов, и поэтому совместное использование общей (разделяемой) памяти разными процессами должно происходить только под контролем системы. Ясно, что для взаимодействия потоков в одном процессе такой проблемы не существует, но зато появляется другая – ошибочное использование общей памяти одним из потоков может привести к неправильной работе других потоков и процесса в целом. Вопрос использования общей памяти связан с

организацией основной памяти и поэтому рассматривается в следующей теме пособия.

Сравнение всех трех способов по трем основным критериям (информативность, безопасность, скорость обмена) приводится в следующей таблице:

Способ взаимодействия	Информативность	Безопасность	Скорость взаимодействия
Сигнальный	низкая	высокая	средняя
Канальный	средняя	высокая	низкая
Общая память	высокая	низкая	высокая

Проблему взаимодействия потоков наглядно можно описать следующим примером. Предположим, что два потока используют один и тот же массив данных, причем первый поток формирует этот массив, выполняя, например, его сортировку, а второй – использует находящиеся в нем данные, например, для отображения их на экране. Ясно, что сортировка массива – это циклический процесс, требующий выполнения весьма большого числа команд. Если сортирующий поток начал выполнение алгоритма сортировки, но не завершил его до конца и был прерван по той или иной внешней причине (напомним, что при вытесняющей многозадачности решение о прерывании потока принимает система), то второй поток не имеет права работать с этим массивом и должен быть заблокирован. Лишь когда первый поток завершит формирование массива, к данному массиву может получить доступ второй поток.

Общие разделяемые данные, которыми могут манипулировать несколько потоков, принято называть **критическими** данными. Тот фрагмент кода потока, который **непосредственно** манипулирует критическими данными, принято называть **критическим кодом,** или **критической секцией.** Критические данные можно рассматривать как частный случай более общего понятия **разделяемого ресурса,** т.е. ресурса,

который одновременно может использоваться несколькими потоками. Использование таких разделяемых ресурсов разными потоками должно происходить строго согласованно, синхронно. Например, одновременный запрос несколькими потоками единственного принтера для вывода своих данных должен приводить к монопольному выделению принтера только одному потоку и блокированию всех остальных.

Для решения данной задачи обычными средствами программирования можно в каждом потоке ввести специальную логическую переменную, доступную всем потокам и фиксирующую состояние общего ресурса (“занято” или “свободно”). Каждый поток, желающий получить общий ресурс, прежде всего должен проверить значение этой переменной и либо захватить ресурс с изменением состояния переменной на “занято”, либо перейти в цикл ожидания освобождения этой переменной. К сожалению, данный способ имеет следующие серьезные недостатки:

- Проверка значения логической переменной и изменение ее значения реализуются на машинном уровне с помощью нескольких машинных команд, и вполне возможна ситуация, когда выполнение потока будет прервано где-то в середине этой последовательности, что конечно же недопустимо.
- Использование общей переменной легко реализуется для потоков внутри одного и того же процесса, но требует обращения к ОС в случае разных процессов.
- Наличие цикла проверки состояния логической переменной в каждом потоке приводит к неоправданным затратам процессорного времени.

По этим причинам организация синхронного использования потоками общих ресурсов была перенесена на уровень ОС. Основой этой реализации является механизм так называемых **семафоров.** Семафор – это специальная переменная целого типа, определяющая число свободных однотипных ресурсов (например, число буферов вывода). Важнейшие особенности семафоров:

- реализация на уровне ядра системы, т.е. доступность всем потокам, но под контролем ОС;
- выполнение базовых операций с семафорами (проверка состояния и изменение значения) как **неделимых (непрерываемых)** операций, которые в силу этого называют примитивными; для реализации данных операций ядро использует механизм временного запрета прерываний.

Следовательно, семафоры являются **системными** объектами, создаваемыми и поддерживаемыми самой системой. Для доступа к этим объектам прикладные потоки могут использовать системные вызовы. При создании семафора указывается его предельное возможное значение, равное числу потенциально доступных однотипных ресурсов. Когда поток запрашивает этот ресурс, проверяется значение семафора, и если оно не ноль, то доступ к ресурсу разрешается и значение семафора на 1 уменьшается, в противном случае поток блокируется. Когда поток освобождает ресурс, значение семафора на 1 увеличивается.

Простейшей разновидностью семафоров являются **двоичные** семафоры, которые могут принимать только 2 возможных значения – 0 и 1. Именно на двоичных семафорах построены такие важные системные объекты, как **критические секции** и **мьютексы** (mutex, сокращение от mutual exclusion, т.е. взаимное исключение). Общим у них является то, что они применяются для синхронизации доступа потоков к разделяемым данным (общим файлам, общим структурам данных), а отличаются они тем, что первые используются для потоков внутри одно процесса, а вторые – для **разных** процессов. Как следствие, реализация мьютексов со стороны системы требует существенно б**о**льших затрат. Для прикладного программиста использование этих объектов практически не отличается, приходится лишь использовать разные системные вызовы.

Например, для использования критических секций (КС) достаточно лишь в создаваемом программном коде выделить начало и конец каждой такой

секции. Для этого используются специальные системные вызовы, такие как **EnterCriticalSection** и **LeaveCriticalSection** в системах семейства Windows.

некритический код потока
системный вызов “Начало КС”
критический код (критическая секция)
системный вызов “Конец КС”
некритический код потока

Когда выполнение кода потока доходит до системного вызова “Начало КС”, система отрабатывает этот вызов следующим образом:

- проверяется состояние внутренней системной логической переменной (флага), связанной с критическими данными;
- если состояние этого флага говорит о занятости критических данных (т.е. другой поток еще раньше выполнил вход в критическую секцию), то поток переводится в состояние ожидания и критический код не выполняется;
- если критические данные свободны, то внутренняя переменная-флаг переводится в занятое состояние и поток начинает выполнение критического кода.

Выполнение критического кода потока может многократно прерываться, но пока его исполнение не дойдет до второго системного вызова “Конец КС”, **никакой другой поток не сможет войти в свою критическую секцию,** связанную с блокированными критическими данными. Отработка системного вызова “Конец КС” включает в себя выполнение системой следующих действий:

- внутренняя системная переменная-флаг изменяет свое состояние на “свободно”;
- просматривается очередь потоков, которые ждут освобождения критических данных, и они переводятся в состояние готовности к выполнению.

Это позволит одному из них (кого выберет планировщик) войти в свою критическую секцию и тем самым, возможно, блокировать другие потоки.

Еще одним интересным способом синхронизации потоков являются мониторы. Мониторы могут встраиваться в языки программирования высокого уровня, что позволяет переносить решение многих непростых вопросов организации взаимодействия потоков на уровень компиляторов. Современная реализация мониторов предполагает использование объектно-ориентированнной технологии и встраивания в соответствующие языки.

При организации взаимодействия потоков есть одна весьма серьезная опасность: попадание двух или нескольких потоков в состояние **взаимной блокировки** или тупика (deadlock). Эту ситуацию можно показать на следующем примере.

Пусть в соответствии со своей внутренней логикой поток 1 должен начать работу сначала с файлом А, а потом (не завершив эту работу) – с файлом В. Пусть поток 2, наоборот, сначала начинает работу с файлом В, а потом – с файлом А. При этом возможна следующая последовательность действий:

- поток 1 начал работу с файлом А и блокировал его, после чего поток 1 был прерван;
- поток 2 начал свою работу с файлом В и блокировал его, после чего был прерван;
- поток 1 возобновляет свою работу, запрашивает файл В, но тот занят потоком 2 и поэтому поток 1 переводится в состояние ожидания;
- поток 2 возобновляет свою работу, запрашивает файл А, но тот занят потоком 1, и поэтому поток 2 тоже переводится в состояние ожидания.

В итоге оба потока переходят в состояние взаимного ожидания и не могут освободить свои занятые файлы и тем самым продолжить нормальную работу. Ясно, что в тупиковую ситуацию может попасть одновременно и несколько потоков. При этом вместо файлов может выступать любой разделяемый ресурс, как физический, так и логический.

Для борьбы с тупиковыми ситуациями было предложено множество методов. Видимо, наиболее интересным методом является полное игнорирование данной ситуации, что, впрочем, объясняется очень маленькой вероятностью возникновения тупика. Здесь работает хорошо известный принцип практической уверенности: очень маловероятное событие можно считать невозможным. Дело в том, что затраты на анализ и устранение ситуации взаимной блокировки могут быть достаточно большими, поэтому разработчики ОС должны сопоставить эти затраты с тем потенциальным вредом, который может принести возникновение данной ситуации.

Вторая группа методов позволяет возникнуть взаимоблокировке, а потом устранить эту ситуацию. Для обнаружения подобной ситуации разработан ряд алгоритмов, основанных на методах теории графов. Эти алгоритмы можно запускать с некоторой периодичностью и после обнаружения deadlock’а применить один из следующих методов устранения:

- уничтожить один или (если требуется) несколько потоков;
- выполнить откат одного из заблокированных потоков, т.е. вернуть его к состоянию ДО запроса ресурса;
- принудительно отобрать ресурс у его владельца и отдать другому потоку.

Наконец, третья группа методов старается вообще избежать возникновения ситуации взаимной блокировки и включает в себя следующие подходы:

- при наличии достаточной исходной информации об имеющихся ресурсах и потоках можно построить безопасную траекторию выделения ресурсов, что на практике, к сожалению, практически нереализуемо;
- минимизация числа потоков, претендующих в каждый момент времени на тот или иной ресурс, т.е. уменьшение вероятности возникновения deadlock’а;

- при запросе потоком нового ресурса он должен сначала освободить все используемые им ресурсы, а уж потом получить все, что надо;
- пронумеровать и упорядочить все ресурсы и выделять их строго в возрастающем порядке.

Видно, что предлагаемые методы достаточно трудоемки, поэтому на практике часто применяют подход, когда сама система может устранять тупики за счет отслеживания времени пребывания потоков в состоянии ожидания, и если это время начинает превышать некоторый допустимый предел, можно принудительно освободить взаимно блокированные ресурсы.