БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
Кафедра информатики

Контрольная работа № 2
по дисциплине «Системное программирование»
«Взаимодействие процессов и потоков – основные
задачи, проблемы, решения»

Вариант 6

Факультет: ИНО
Специальность: ИиТП
Студент: Дубейковский А.А.
Группа № 893551
Зачётная книжка № 75350046

Введение

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.

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

Критические данные

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.

Решение проблемы для согласованного использования критических данных (лог. прерменные, семафоры, мьютексы):

1) 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.

2) 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.

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 больших затрат.

Мьютексы:

1) CreateMutex() - создание

2) CloseHandle() - удаление

3) OpenMutex() - открытие

4) ReleaseMutex() - освобождение захваченного

Deadlock

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 A, а потом (не завершив эту работу) – с файлом B. Пусть поток 2, наоборот, сначала начинает работу с файлом B, а потом – с файлом A. При этом возможна следующая последовательность действий:

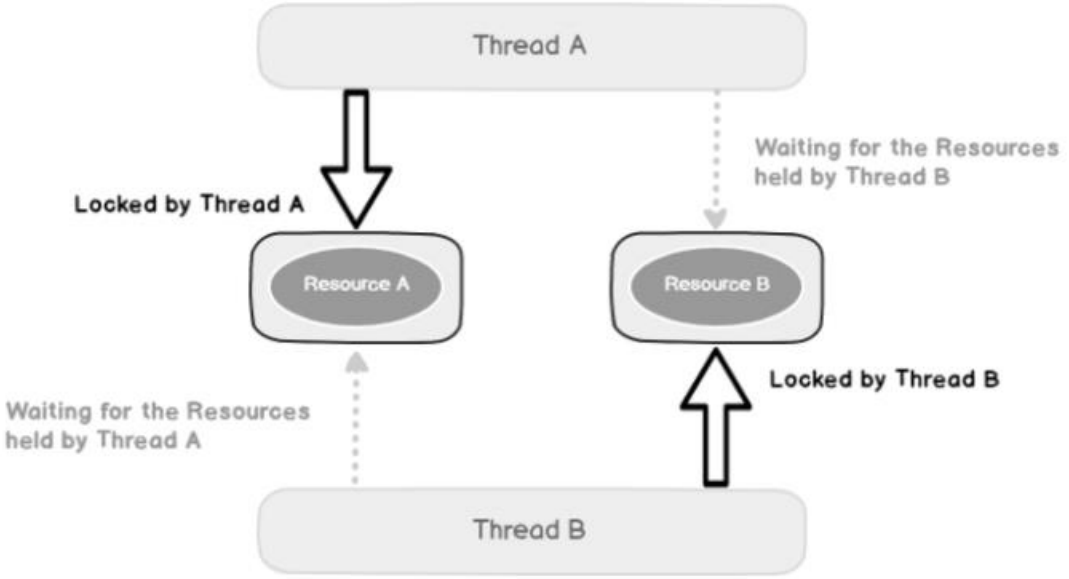
1) поток 1 начал работу с файлом A и блокировал его, после чего поток 1 был прерван;

2) поток 2 начал свою работу с файлом B и блокировал его, после чего был прерван;

3) поток 1 возобновляет свою работу, запрашивает файл B, но тот занят потоком 2 и поэтому поток 1 переводится в состояние ожидания;

4) поток 2 возобновляет свою работу, запрашивает файл A, 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.



Решения для deadlock ситуаций:

1) полное игнорирование (малая вероятность появления ошибки deadlock). Используется в связи с тем, что обработать эту ошибку требует больших ресурсов, а сама она встречается слишком редко.

2) с помощью алгоритмов основанных на теории графов отследить deadlock после его появления и потом перейти к одному из следующих решений:

2.1) уничтожить один или (если требуется) несколько потоков;

2.2) выполнить откат одного из заблокированных потоков, т.е. вернуть его к состоянию ДО запроса ресурса;

2.3) принудительно отобрать ресурс у его владельца и отдать другому потоку.

3) так же есть способы по решению этой проблемы даже не позволяя ей случиться:

3.1) при наличии достаточной исходной информации об имеющихся ресурсах и потоках можно построить безопасную траекторию выделения ресурсов, что на практике, к сожалению, практически нереализуемо;

3.2) минимизация числа потоков, претендующих в каждый момент времени на тот или иной ресурс, т.е. уменьшение вероятности возникновения deadlock'a;

3.3) при запросе потоком нового ресурса он должен сначала освободить все используемые им ресурсы, а уж потом получить все, что надо;

3.4) пронумеровать и упорядочить все ресурсы и выделять их строго в возрастающем порядке.

Голодание процессов

Существует еще одна проблема у конкурирующих процессов – голодание. Предположим, что имеется 3 процесса (P1, P2, P3), каждому из которых периодически требуется доступ к ресурсам R. Представим ситуацию, в которой P1 обладает ресурсом, а P2 и P3 приостановлены в ожидании освобождения ресурса R. После выхода P1 из критического раздела доступ к ресурсу будет получен одним из процессов P2 или P3.

Пусть ОС предоставила доступ к ресурсу процессу P3. Пока он работает с ресурсом, доступ к ресурсу вновь требуется процессу P1. В результате по освобождении ресурса R процессом P3 может оказаться, что ОС вновь предоставит доступ к ресурсу процессу P1. Тем временем процессу P3 вновь требуется доступ к ресурсу R. Таким образом, теоретически возможна ситуация, в которой процесс P2 никогда не получит доступ к

требуемому ему ресурсу, несмотря на то, что никакой взаимной блокировки в данном случае нет.

Решение:

Создать очередь по приоритетам, где чем дольше ждёт объект, чем больше становится его приоритет и он продвигается в очереди

Livelock

Livelock- это программы, которые активно выполняют параллельные операции, но эти операции никак не влияют на продвижение состояния программы вперед.

Ситуация, в которой два или более процессов непрерывно изменяют свои состояния в ответ на изменения в других процессах без какой-либо полезной работы. Это похоже на deadlock, но разница в том, что процессы становятся “вежливыми” и позволяют другим делать свою работу.

Выполнение алгоритмов поиска удаления взаимных блокировок может привести к livelock — взаимная блокировка образуется, сбрасывается, снова образуется, снова сбрасывается и так далее.

Рассмотрим простой пример **livelock**, где муж и жена пытаются поужинать, но между ними только одна ложка. Каждый из супругов слишком вежлив, и передает ложку, если другой еще не ел.

Решение:

Сделать тоже определённую очередь из приоритетов, построить какую-то иерархию.

Источники

- Бек, Л. Введение в системное программирование / Л. Бек : Пер. с англ. – М.: Мир, 1988. – 448 с., ил.
- https://ru.wikipedia.org/wiki/Семафор
- https://upread.ru/blog/articles-it/vzaimodejstvie-i-sinhronizaciya-potokov
- https://stackoverflow.com/

- https://medium.com/german-gorelkin/deadlocks-livelocks-starvation-ccd22d06f3ae
- https://en.wikipedia.org/wiki/Starvation_(computer_science)#:~:text=A%20possible%20solution%20to%20starvation,system%20for%20a%20long%20time.
- https://en.wikipedia.org/wiki/Dining_philosophers_problem