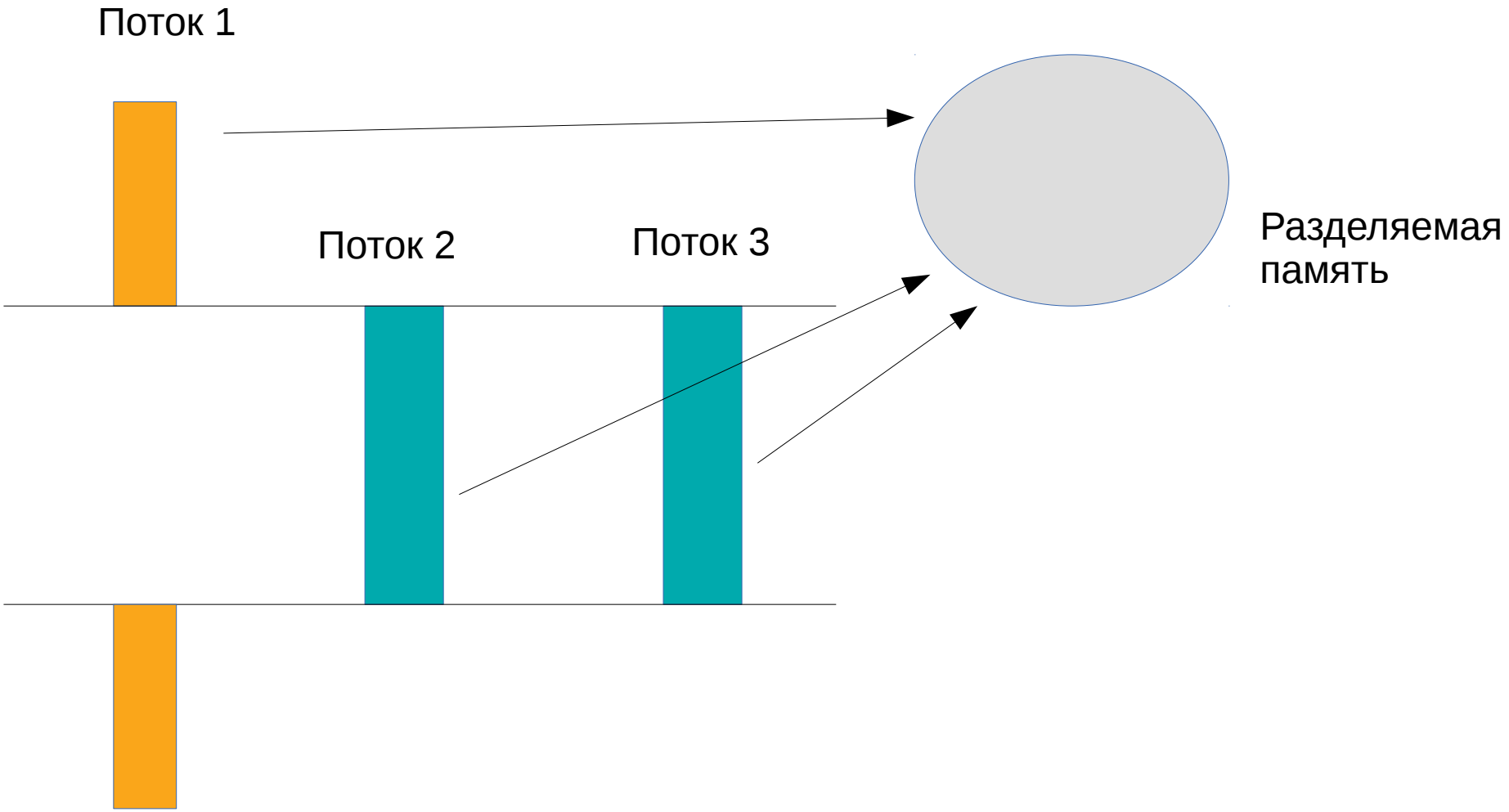
Многопоточные программы. Проблемы.

Классические проблемы многопоточных программ.

Классический подход

- Основные понятия :
  - Thread - поток выполнения, выделяемый операционной системой.
  - Разделяемая память – память приложения предназначенная для совместной работы потоков
  - Примитивы блокировки доступа к данным
  - Примитивы синхронизации потоков

На что это похоже

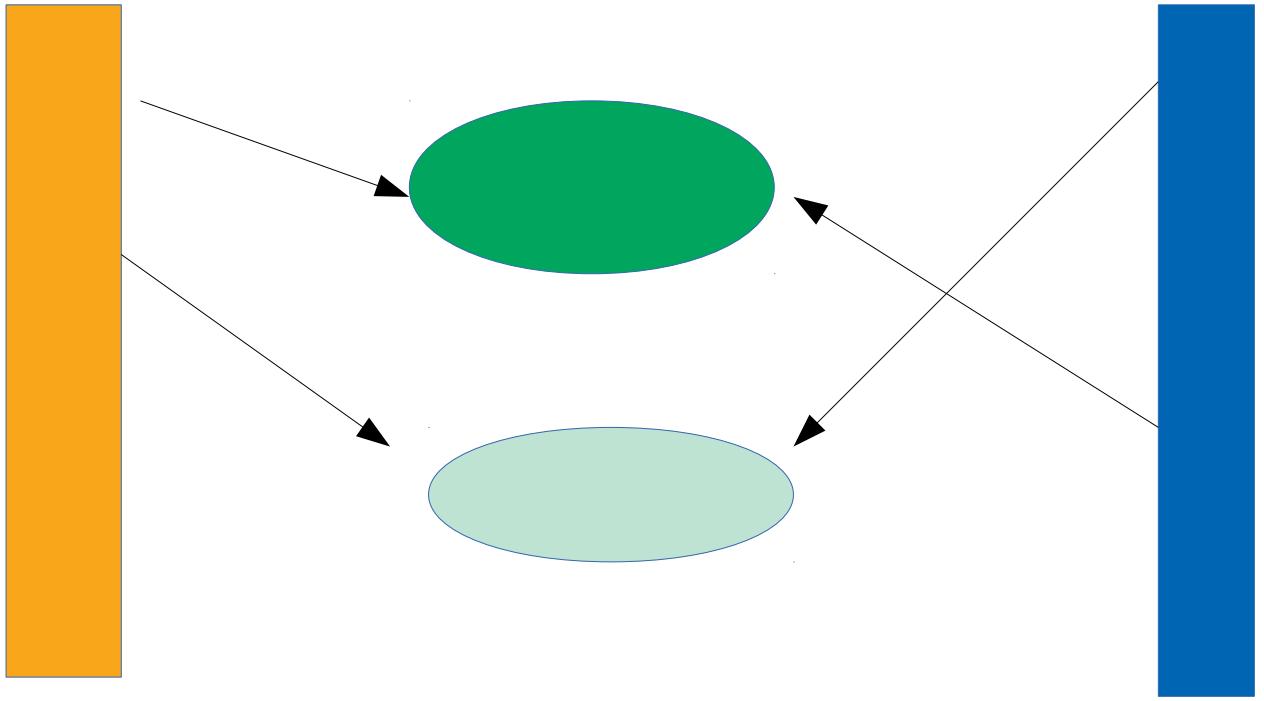


Преимущества

- “Видимая легкость” разработки
  - В случае необходимости создаем поток, который работает с теми же данными с которыми работает основная программа
  - Для защиты памяти используем барьеры (примитивы синхронизации, mutex)
- Высокая скорость работы если программа написана эффективно

Проблемы. Deadlock

- Происходит в ситуации взаимной монопольной блокировки ресурсов двумя потоками



Подробности

- Самая очевидная из встречающихся проблем
- Лечится установкой жесткого порядка блокировки ресурсов
- Легко обнаруживается в случае если попытки блокировки имеют таймаут и механизм для обработки аварийных ситуаций.

Data race. Thread safe.

- Data race происходит в ситуации когда расширяемая область памяти не защищена барьером (например Mutex) и используется из нескольких потоков одновременно
- В случае сложных структур данных мы получаем определенную вероятность того что они будут “сломаны”
- В java используется понятие Thread safe.
  - Если экземпляр класса является thread safe то его можно безнаказанно использовать из нескольких потоков одновременно.

Подробности

- Data race иногда очень сложно обнаружить, так как на тестах может просто ‘везти”
- Из документации часто непонятно является ли данный класс ThreadSafe или нет
- В процессе эволюционирования программы кто-то может применить объект не так как задумывалось изначально
- Методология “Облако объектов” часто провоцирует data race

Как обнаружить и что делать

- Генерируем логи и анализируем их. Debug обычно бесполезен.
- Каждый объект который не хранится в стеке подпадает под подозрение.
- Все статические переменные (например шаблоны для регулярных выражений) должны быть thread safe
- Минимизируем межпроцессное взаимодействие
- Все данные что ходят между потоками должны быть immutable (желательно) или хотя бы thread safe

“Слишком много потоков”

- Генерирование потоков без контроля
  - Падение общей производительности из-за затрат на переключение контекста
  - Аварийное завершение приложения в случае исчерпания количества потоков.
  - Большие накладные расходы на запуск потоков.

Что делать

- Использовать пулы потоков.
- Переходить на более сложные модели :
  - Очереди сообщений
  - Асинхронное взаимодействие
  - Зеленые потоки
  - акторы

Race conditions или условия гонки

- Эта проблема случается в ситуации когда результат работы программы зависит от работы планировщика потоков
- Пример :

```
lock a
flag = a>d;
unlock a;
...
if(flag) {
    Lock b
    b = b+d
    Unlock b
    Lock a
    a = a-d
    Unlock a
}
```

Как обнаруживать и что делать

- Минимизировать совместный доступ к переменным и не полагаться целиком на семафоры.
- Выделять особый поток который обладает монопольным правом менять разделяемые данные одновременно.
- Использовать сложные абстракции вместо разделяемой памяти
- Использовать устойчивые к порядку выполнения структуры и api

Отсутствие мониторинга состояния потоков и обработки ошибок

- Ведет к некорректной работе программ в случае возникновения проблем (например с IO)
- Достаточно сложная задача – из коробки обычно нет подходящего инструментария
- Существенное усложнение логики работы

Что делать

- Вводить новые абстракции :
  - Контролирующий поток
  - Система акторов
  - Очередь с таймаутом на обработку
  - Транзакции с логикой отката

Слишком широкое использование блокировок

- Каждый поток пытается захватить “все” перед началом выполнения
- Часто является простым и неправильным ответом на race conditions
- Производительность на уровне однопоточной системы

Синхронное api

- Синохронным называет api которое блокирует поток
- Плюсы и минусы
  - Просто реализуется
  - Легко обрабатывать ошибки
  - В определенных ценариях имеет максимальное быстродействие
  - Ведет к нерациональному использованию ресурсов системы : потоков, сокетов и т.д.

Асинхронное API

- Не блокирует вызвавший поток
  - Обычно используется либо callback либо future
- Плюсы и минусы :
  - Существенно усложняется логика работы программы
  - В определенных сценариях менее производительно
  - Выше эффективность использования ресурсов системы
  - Существенно сложнее обработка ошибок
  - Можноо избежать явного использования потоков