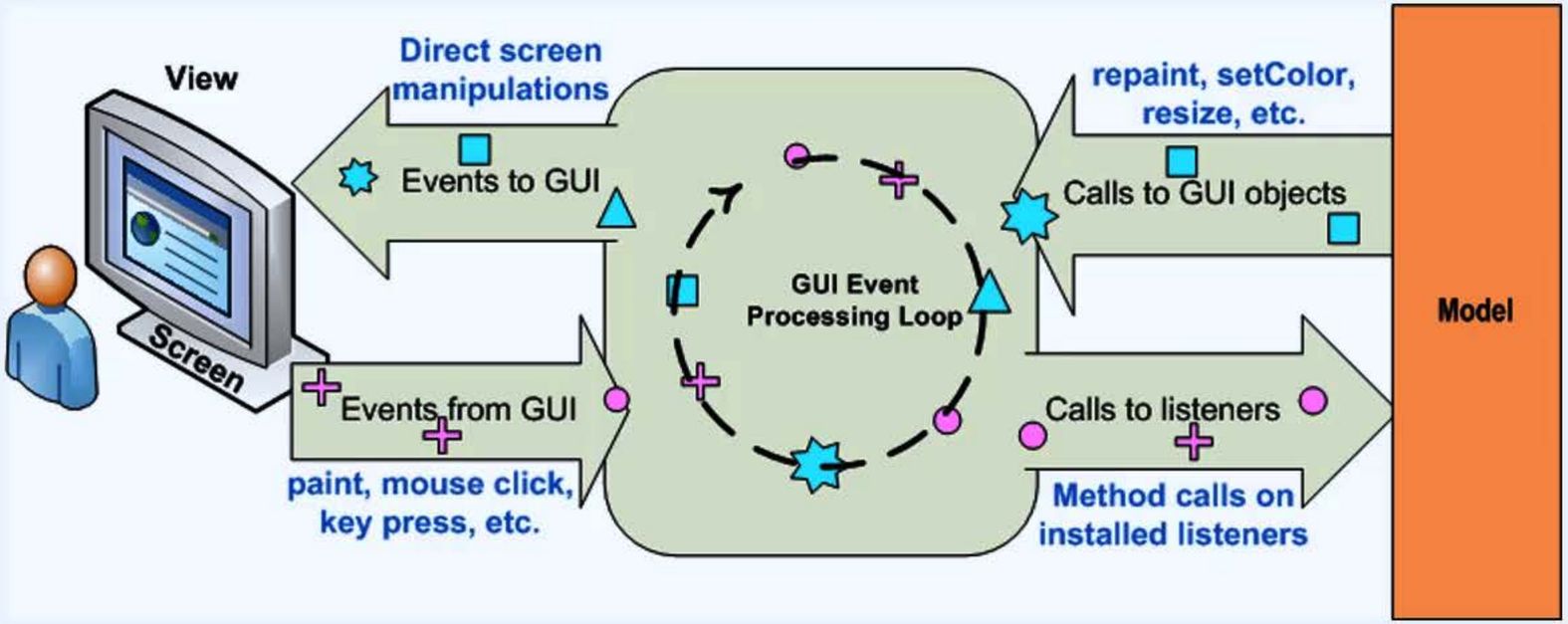
Проблема однопоточности



Daemon Threads и ожидание

- Потоки делятся на два вида:
  - Foreground
  - Background (daemon)
- t = Thread(target, daemon=True) # daemon-thread
- t.start() # запуск потока
- t.join() # ждём завершения потока t

Executor API

- Работать напрямую с потоками сложно
- Пакет **concurrent.futures** определяет высокоуровневые типы
- **ThreadPoolExecutor**(max_workers=None, thread_name_prefix=‘’,
  initializer=None, initargs=())
- **ProcessPoolExecutor**(max_workers=None, mp_context=None,
  initializer=None, initargs=())

Executor API

- **Executor** – абстрактный класс
- submit(fn, *args, **kwargs) -> Future
- map(fn, *iterables, timeout=None, chunksize=1) -> []
- shutdown(wait=True)
- Future – специализированный низкоуровневый тип, представляющий будущий результат асинхронного вызова функции

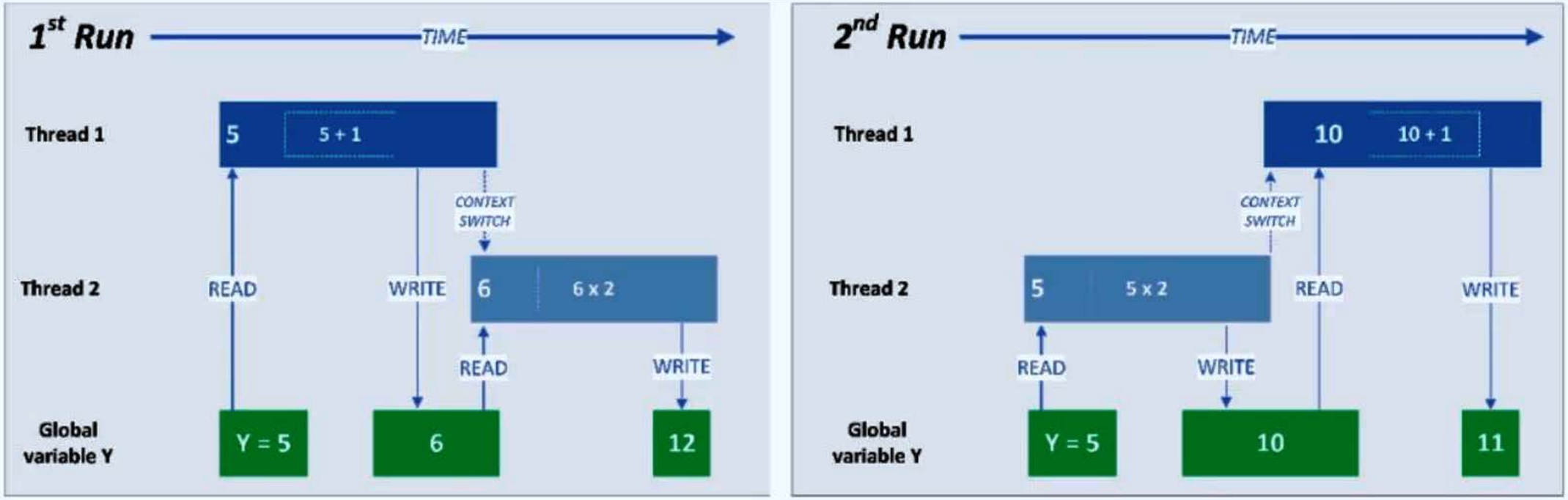
CPU-bound и Threads

- Потоки для CPU-bound операций используются для освобождения main loop
- Последовательное исполнение CPU-bound операций работает быстрее чем «параллельное»

Ужасы «многопоточки»

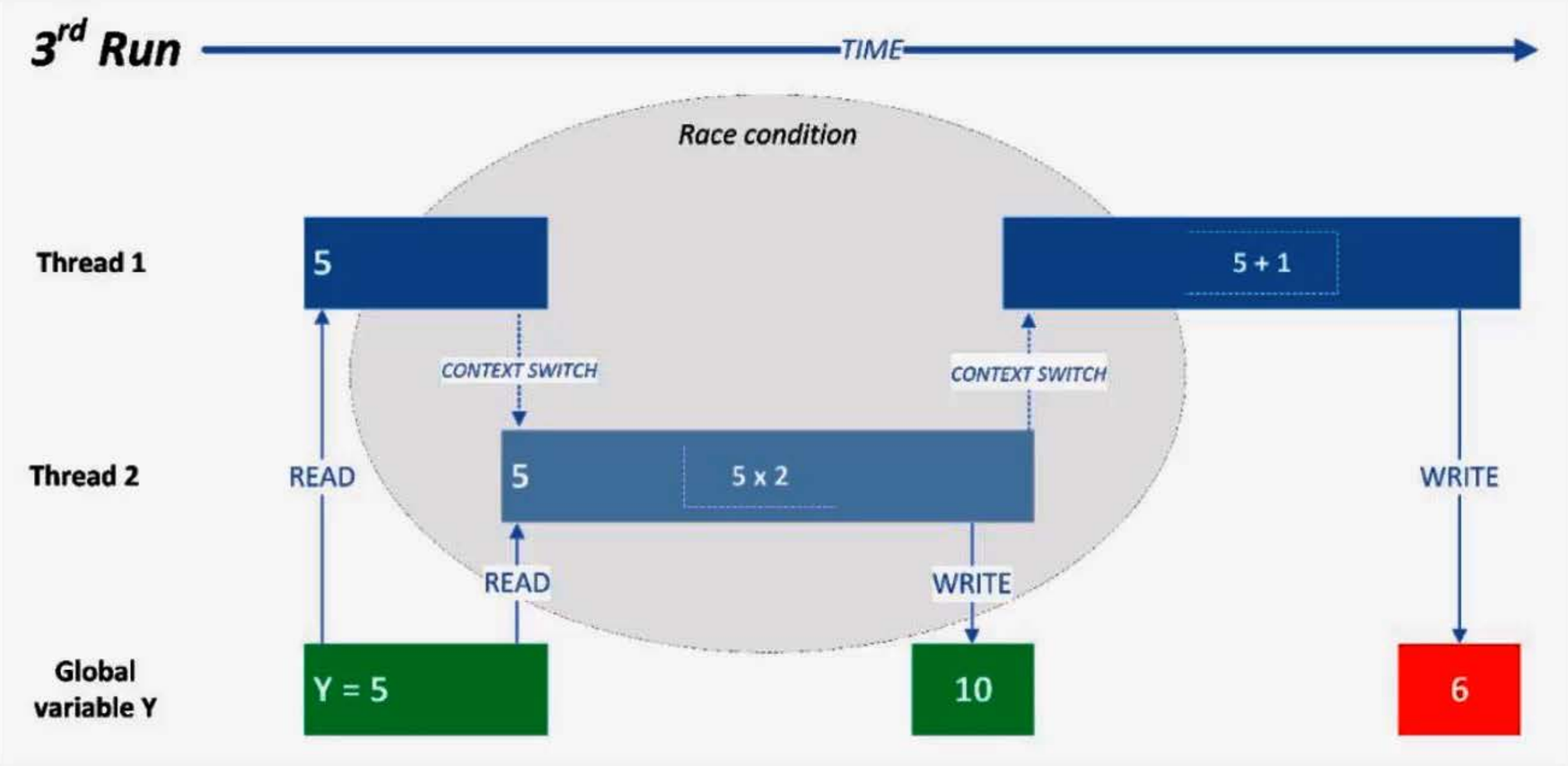
- Race
- Deadlock

Race Condition (гонка)



Источник изображений:https://www.rapitasystems.com/race-condition-testing

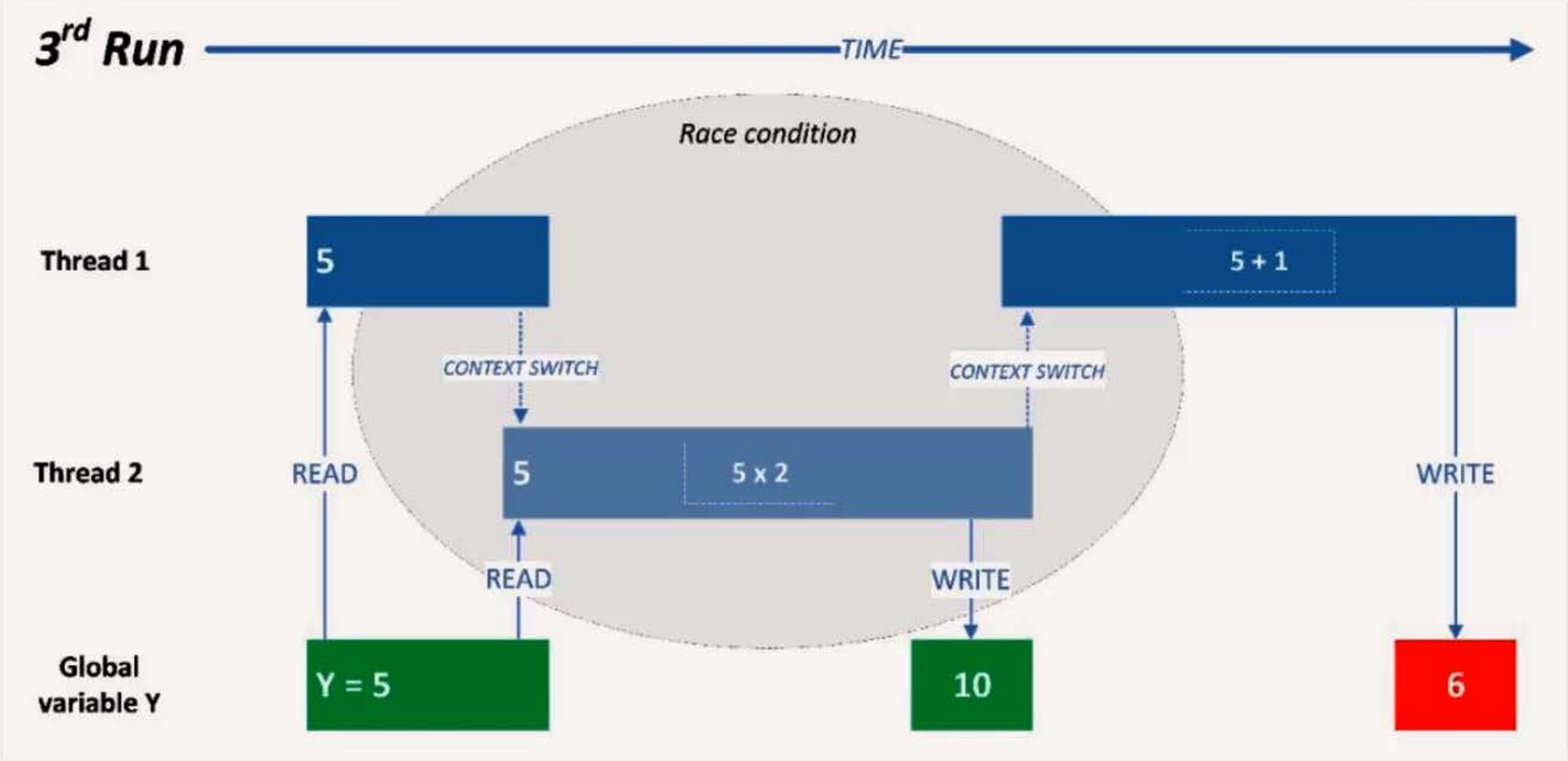
Race Condition (гонка)



Источник изображений:https://www.rapitasystems.com/race-condition-testing

Race Condition (гонка)

На самом деле, race condition это частный случай гонки. Здесь нет явной гонки по условию (condition), но гонка есть гонка и по факту я лишь немного неточно здесь выражаюсь.



Источник изображений:https://www.rapitasystems.com/race-condition-testing

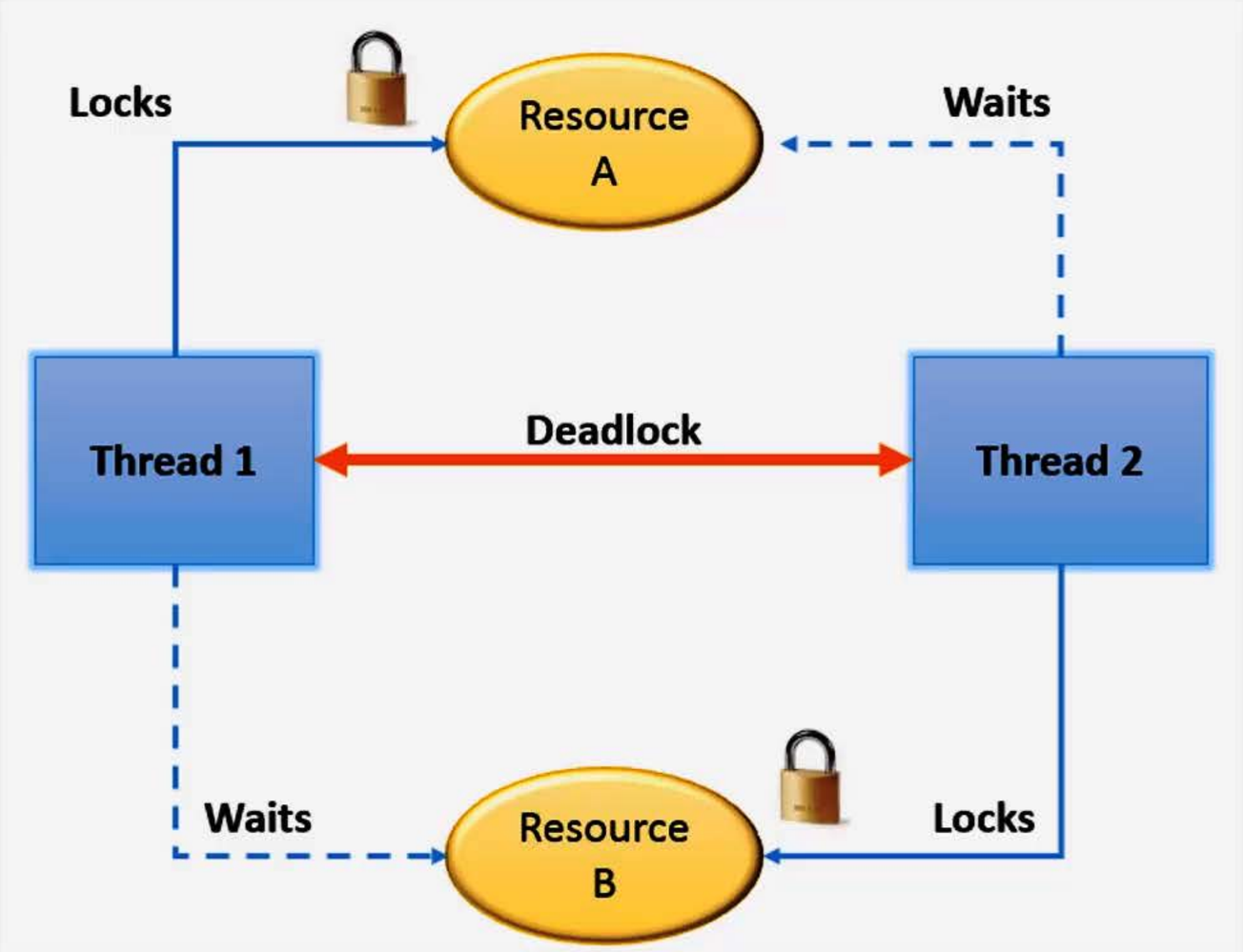
Примитивы синхронизации

- Lock – даёт эксклюзивный доступ к ресурсу
- RLock – то же, что Lock, но позволяет reentrance потоку, взявшему lock
- Event – позволяет в одном потоке ждать сигнала от другого
- Condition – объединяет Event и Lock
- Semaphore – ограничивает доступ к ресурсу по кол-ву потоков
- BoundedSemaphore – защищает от излишних release (“багов”)
- Barrier – ожидание завершения N-потоков

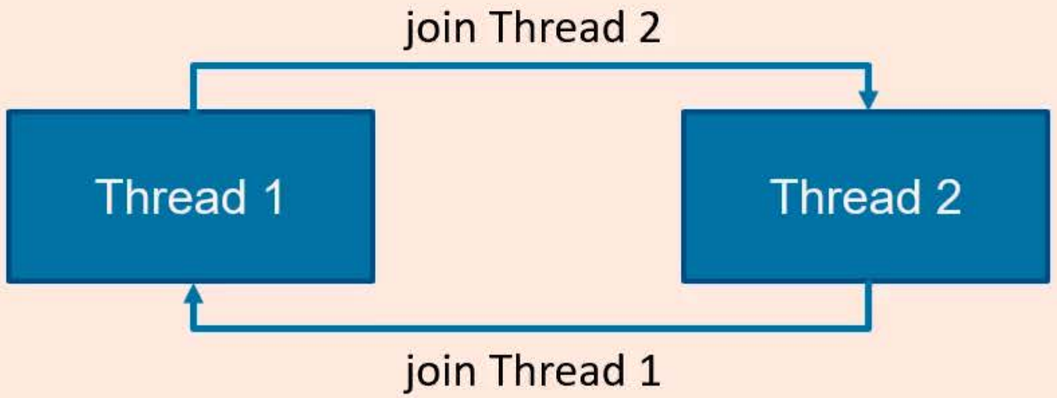
Deadlock

- Deadlock – состояние бесконечной взаимной блокировки
- Как правило возникает:
  - при взаимном ожидании блокировки (разблокировки) двух локов
  - при взаимном ожидании потоков
  - при рекурсивном захвате одного и того же лока

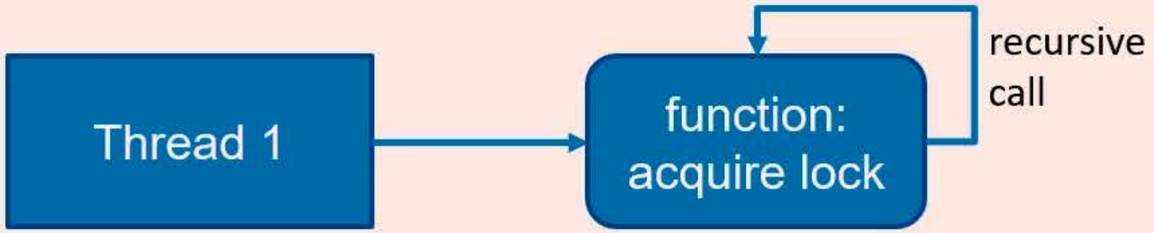
Deadlock


Locks
Resource A
Waits
Thread 1
Deadlock
Thread 2
Waits
Resource B
Locks

Deadlock



Deadlock



Deadlock

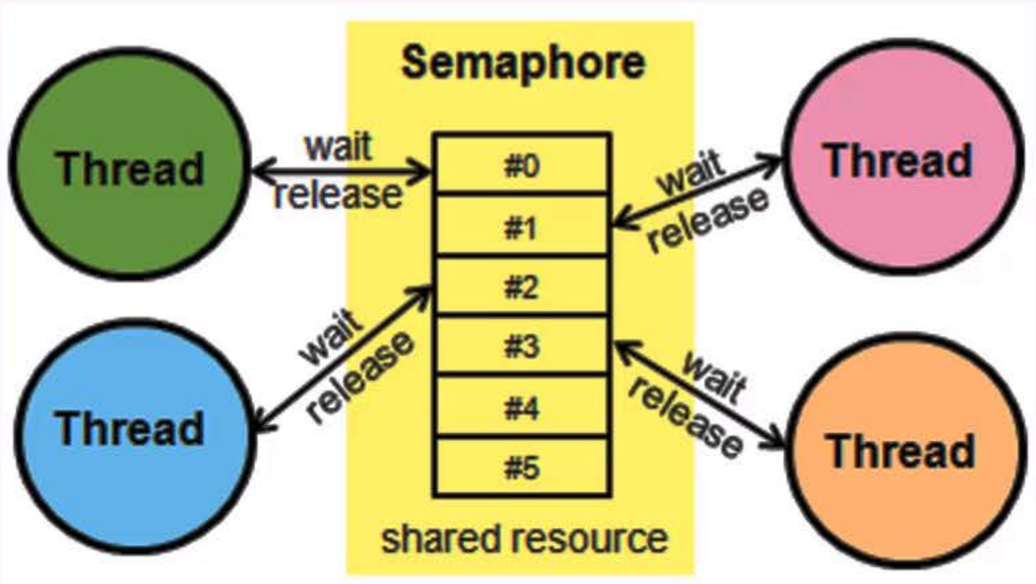
- Как правило можно избежать за счёт грамотного структурирования многопоточности
- Проставление таймаутов на захват лока
- Использование RLock при рекурсиях
- Есть и более сложные подходы к детектированию и избеганию deadlocks

Signaling

- Проблема:
  - есть third-party библиотека, через которую наша программа взаимодействует с банковским терминалом, инициируя транзакции
  - Эта библиотека отвечает асинхронно
  - Нам нужно сделать обёртку (которая делает асинхронный вызов библиотеки), дающую возможность клиентскому коду, по желанию дожидаться окончания операции
- Если одному потоку нужно дождаться события в другом потоке, то используется примитив синхронизации - Event

Semaphore

- Семафор используется для ограничения доступа к ресурсу по количеству потоков
- Пример: ограничение количества подключений

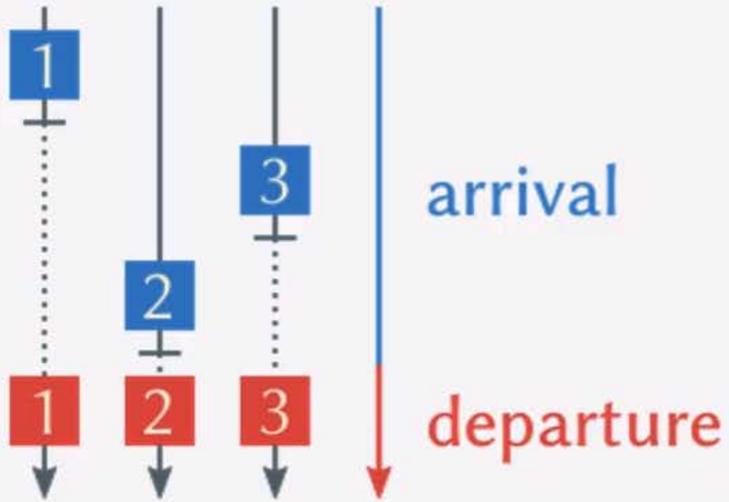


Semaphore

- Semaphore(value) # value – ёмкость семафора
- acquire() – захват слота
- release() – освобождение слота
- _value – текущая ёмкость

Barrier

- Барьер используется для синхронизации фаз алгоритма
  (в котором задействованы N-потоков)
- Пример: алгоритм имеет две фазы:
  1 – параллельное считывание данных из разных источников
  2 – агрегация данных (которая невозможна пока не появятся все данные)



Barrier

- Barrier(parties)
  # parties – по сути, кол-во потоков, синхронизируемых по фазам
- wait () – выставление барьера
- reset() – принудительное возвращение в исходное состояние
- abort() – перевод в broken state
- n_waiting – кол-во потоков, достигших барьера
- broken – флаг корректности состояния

Атомарные операции

- Атомарная операция – операция, которая выполняется целиком
- Интерпретатор переключает потоки только между инструкциями байткода. Инструкции байткода – неделимы.
- Это важно с точки зрения конкурентного доступа к разделяемым ресурсам (казалось бы ☺)

Атомарные операции

L, L1, L2 – lists; D, D1, D2 – dicts; x, y – objects; i, j - ints

Примеры атомарных операций:

- L.append(x)
- L1.extend(L2)
- x = L[i]
- x = L.pop()
- L.sort()
- x = y
- x.field = y
- D[x] = y
- D1.update(D2)

Примеры не атомарных операций:

- i = i+1
- L.append(L[-1])
- L[i] = L[j]
- D[x] = D[x] + 1

Отмена потока

- Простого (хорошего) способа отменять потоки не существует
- Самое простое (и чаще всего плохое) – убить поток, запустив в отдельном процессе
- Кооперативная отмена – самый адекватный подход
- Кооперативная отмена может базироваться на:
  - булевых флагах
  - флагах на основе примитивов синхронизации
- Unit of Work-паттерн упрощает дизайн кооперативной отмены

Почему «асинхронка» так важна?

- Мы хотим делать приложения надёжными
- Пользователи, заказчики и сами разработчики должны быть счастливы
- Распараллеливание делает приложения быстрее
- Современные приложения часто работают с большими объёмами данных
- Смартфоны и планшеты изменили представление людей о ПО.
  Пользователи ожидают мгновенной и плавной работы
- Множество современных API написаны в асинхронном стиле
  и вам это придётся учитывать

Проблемы

- Очень сложно разрабатывать ПО полное асинхронности и параллельных вычислений
- Вам необходимо обладать широкими познаниями,
  чтобы решать соответствующие проблемы
- Сложно грамотно проектировать программы полные «асинхронки»
- «Осознание» многопоточного кода всегда сложнее последовательного
- Очень легко вносить ошибки при многопоточном / асинхронном коде

Обработка исключений

- Обработка ошибок – сложная тема
- Можно ловить исключения внутри потока
- Можно ловить в вызывающем потоке
- Thread «умалчивает» об исключениях
- Thread API не даёт возможности отловить в вызывающем потоке
- ThreadPoolExecutor API даёт возможность отловить исключение в вызывающем потоке