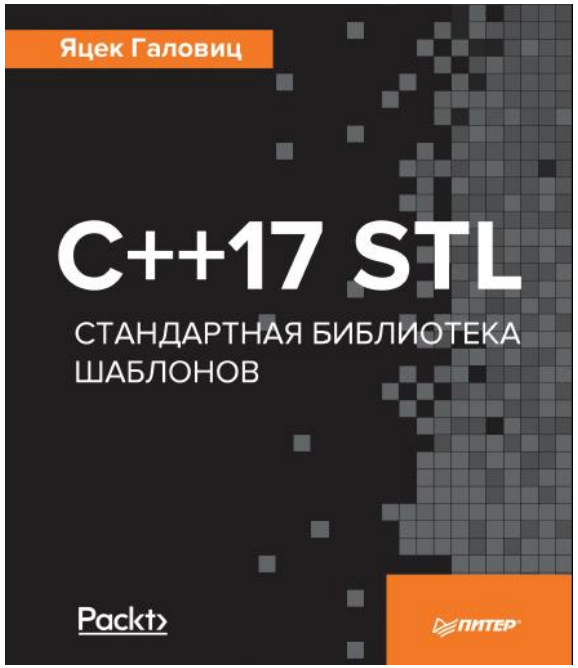
Задача Синхронизация std::cout

Синхронизация конкурентного использования std::cout

Одной из структур данных, часто применяемых для вывода данных, является std::cout. Если несколько потоков пытаются получить доступ к cout на конкурентной основе, то мы получим смешанные выходные данные. Чтобы это предотвратить, следует написать собственную функцию, которая выводит данные на экран и защищена от конкурентности.

Синхронизация конкурентного использования std::cout



Многопоточные программы неудобны тем, что нужно охранять каждую структуру данных, которую они изменяют, с помощью мьютексов или других средств защиты от неуправляемых конкурентных изменений.

Чтобы предотвратить искажение сообщений из-за конкурентности, реализуем небольшой вспомогательный класс, который синхронизирует вывод данных между потоками.

Синхронизация конкурентного использования std::cout

Оболочка для cout

Идея заключается в том, что pcout явно наследует от stringstream. Таким образом, можно применять operator<< для экземпляров этого класса. Как только экземпляр pcout уничтожается, его деструктор блокирует мьютекс, а затем выводит на экран содержимое буфера stringstream.

```
struct pcout : public stringstream {
    static inline mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> lk {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

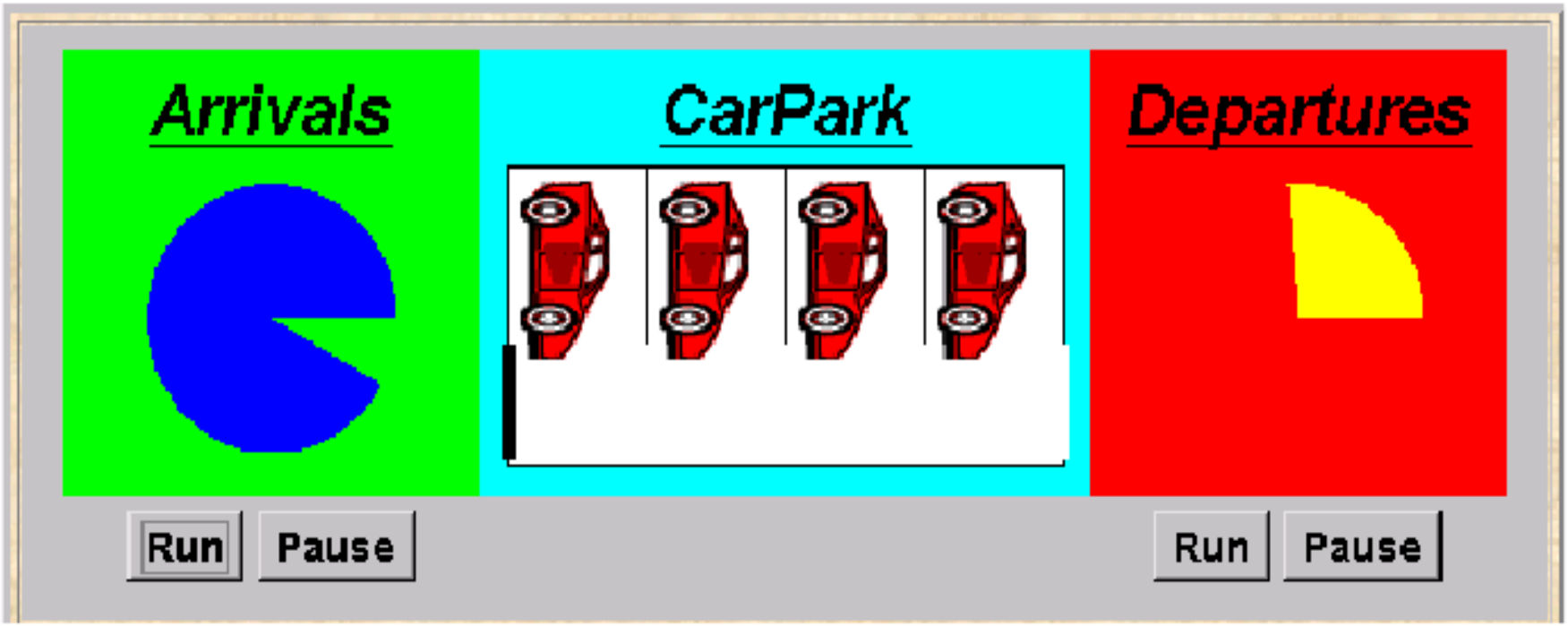
Синхронизация конкурентного использования std::cout

```
struct pcout : public stringstream {
    static mutex cout_mutex;
    ~pcout() {
        lock_guard<mutex> lk {cout_mutex};
        cout << rdbuf();
        cout.flush();
    }
};
```

```
cout << "cout hello from " << id << '\n';
```

```
pcout{} << "pcout hello from " << id << '\n';
```

Задача CarPark



A controller is required for a carpark, which only permits cars to enter when the carpark is not full and permits cars to leave when there it is not empty. Car arrival and departure are simulated by separate threads.

Java: Производитель-потребитель

CarParkControl - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notifyAll();
  }
}
```

block if full

block if empty

Is it safe to use **`notify()`**
rather than **`notifyAll()`**?

Java: Производитель-потребитель

CarParkControl - condition synchronization

```
class CarParkControl {
  protected int spaces;
  protected int capacity;

  CarParkControl(int n)
    {capacity = spaces = n;}

  synchronized void arrive() throws InterruptedException {
    while (spaces==0) wait();
    --spaces;
    notifyAll();
  }

  synchronized void depart() throws InterruptedException {
    while (spaces==capacity) wait();
    ++spaces;
    notifyAll();
  }
}
```

block if full

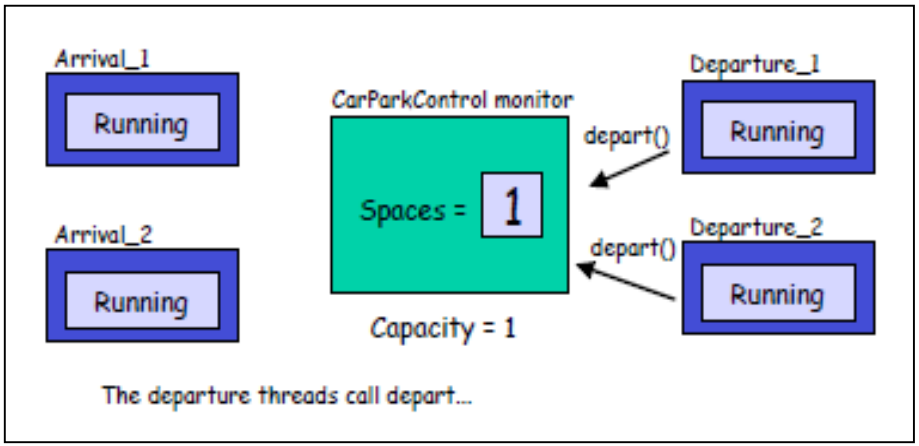
block if empty

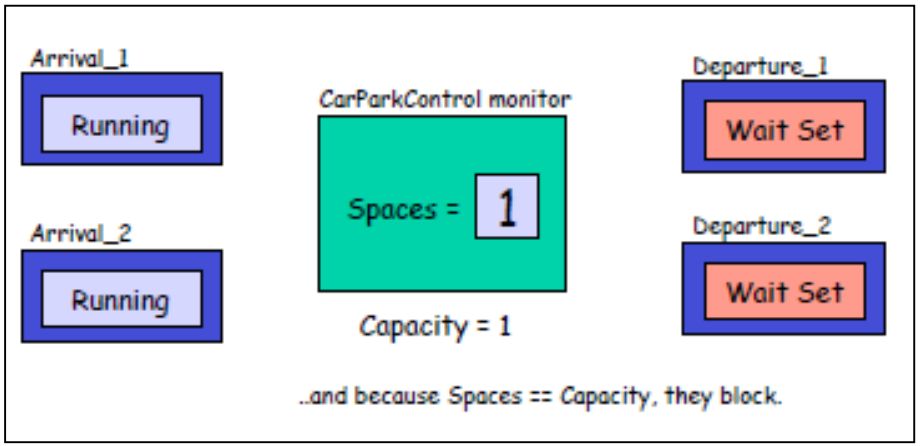
Is it safe to use **`notify()`** *rather than* **`notifyAll()`**?

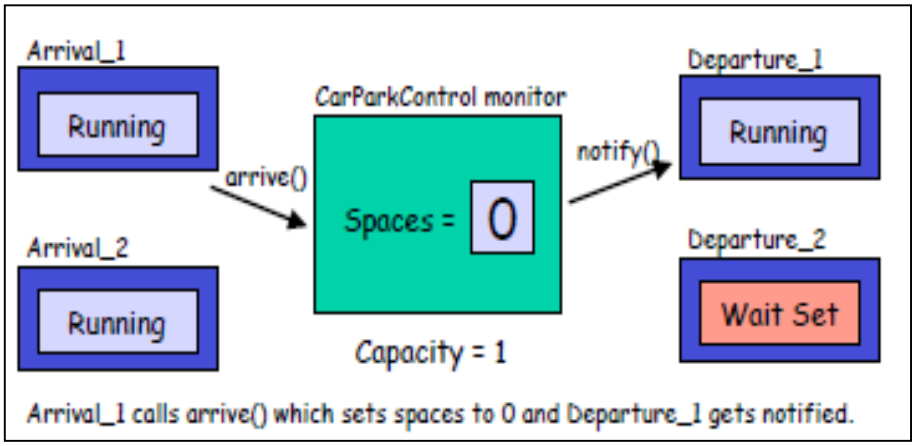
Вопросы

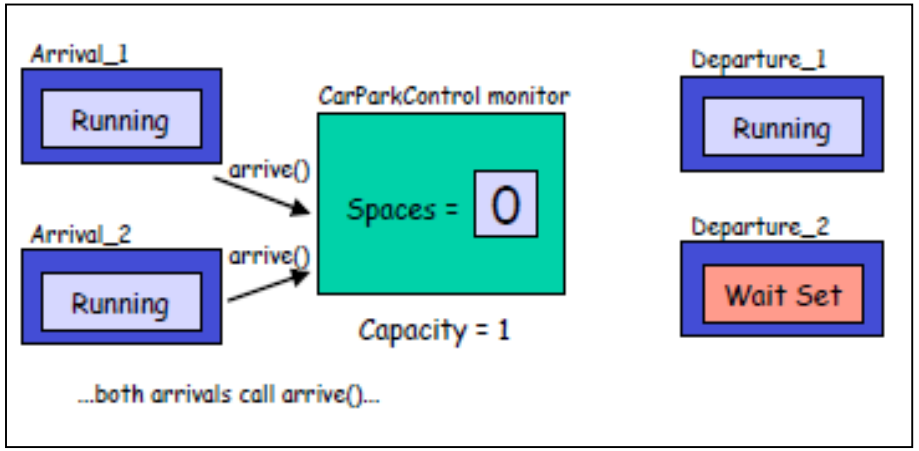
1) Зачем ожидание в цикле
2) notify() против notifyAll()

notify() против notifyAll()

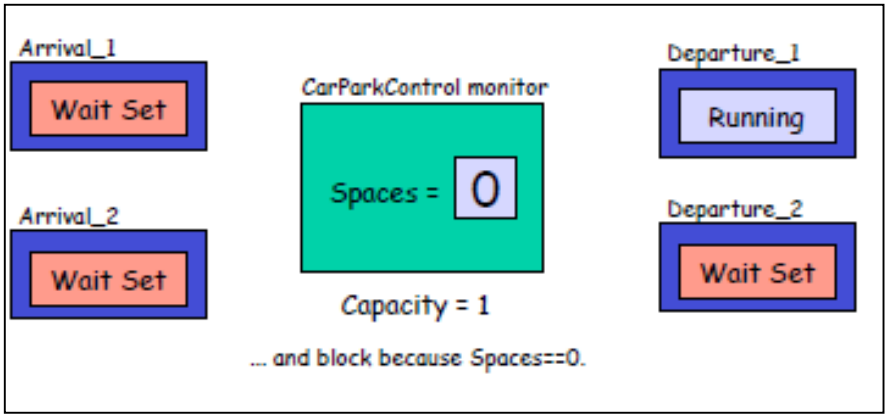


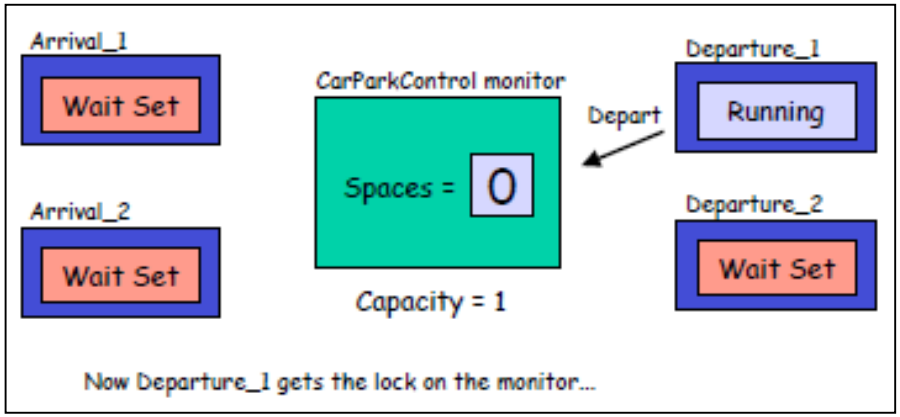


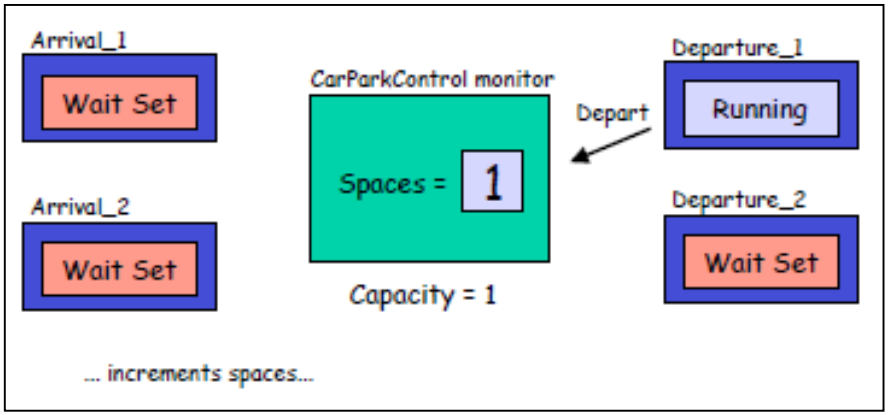


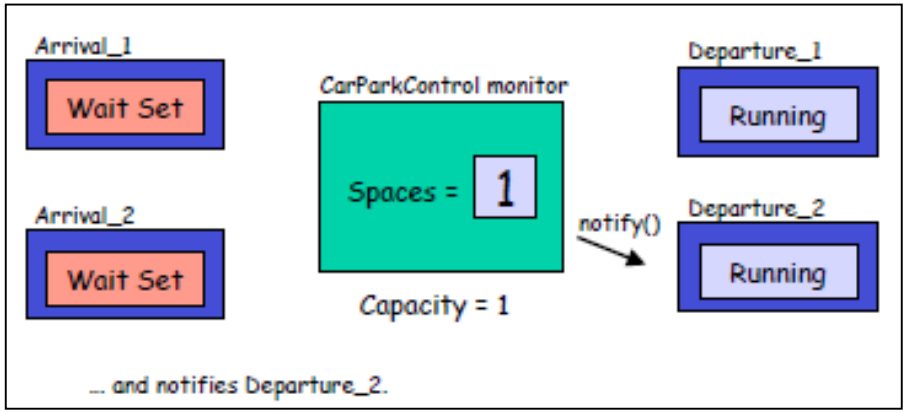


notify() против notifyAll()









notify() против notifyAll()