Уральский федеральный университет

Параллельные вычисления Многопоточное программирование, часть 2

Созыкин Андрей Владимирович

к.т.н.

Заведующий кафедрой высокопроизводительных компьютерных технологий

Институт математики и компьютерных наук

Многопоточное программирование C++

Появилось в C++11

Класс thread

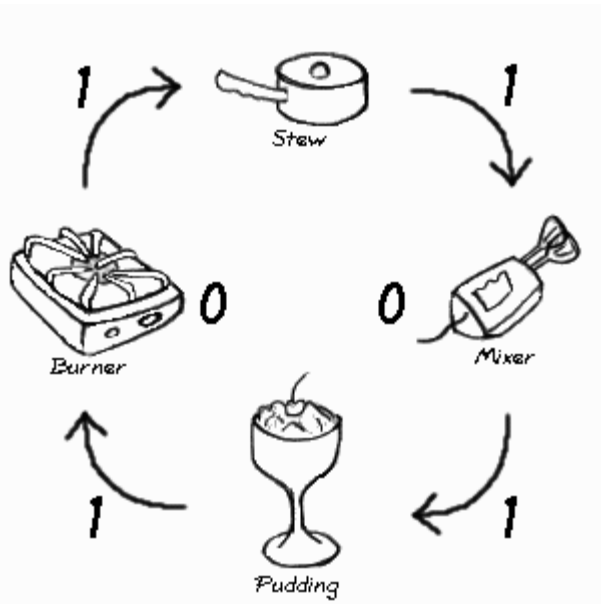
Проблемы:

- Недетерминированность
- Условия гонок (Race conditions)

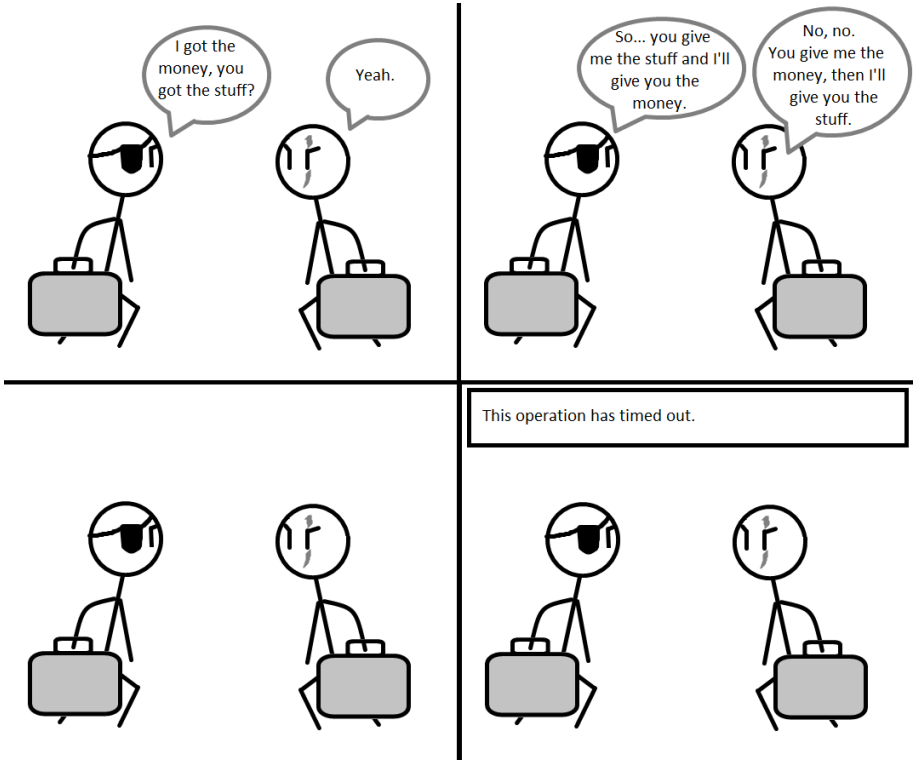
Решение:

- Критические секции
- Класс mutex
- RAII lock_guard<>

Всегда ли хороши блокировки?



http://www.cs.gmu.edu/cne/itcore/processes/Dead.html



http://codingcomics.blogspot.ru/2009/08/concurrency-issues.html

Взаимоблокировка (Deadlock)

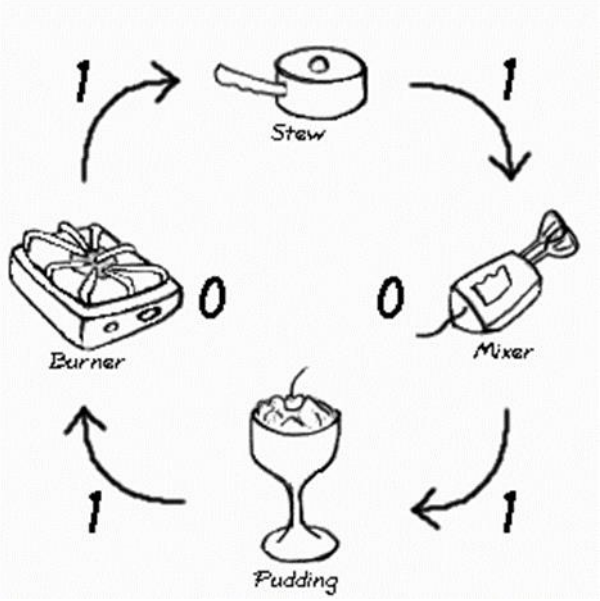
Пример с кухней

На кухне есть два ресурса:

- Плита
- Миксер

Повар знает два рецепта:

- Тушеное мясо (stew) – начните взбивать пока мясо тушится на плите, взбивайте некоторое время после снятия с плиты
- Пудинг (pudding) – начните взбивать перед установкой на плиту и продолжайте взбивать некоторое время на плите



http://denninginstitute.com/pjd/ArtOS/Deadlock.pdf

Программа для кухни

```
using namespace std;
int main(){
    mutex burner, mixer;
    thread stew_thread(stew, ref(burner), ref(mixer));
    thread pudding_thread(pudding, ref(burner), ref(mixer));
    stew_thread.join();
    pudding_thread.join();
}
```

Поток stew

```
void stew(mutex& burner, mutex& mixer){
    cout << "[stew] Locking Burner\n";
    lock_guard<mutex> bl(burner);
    cout << "[stew] Burner is locked\n";
    cout << "[stew] Locking Mixer\n";
    lock_guard<mutex> ml(mixer);
    cout << "[stew] Mixer is locked\n";
    cout << "[stew] Preparing\n";
}
```

Поток pudding

```
void pudding(mutex& burner, mutex& mixer){
    cout << "[pudding] Locking Mixer\n";
    lock_guard<mutex> ml(mixer);
    cout << "[pudding] Mixer is locked\n";
    cout << "[pudding] Locking Burner\n";
    lock_guard<mutex> bl(burner);
    cout << "[pudding] Burner is locked\n";
    cout << "[pudding] Preparing\n";
}
```

Результат работы программы

```
[stew] Locking Burner
[pudding] Locking Mixer
[pudding] Mixer is locked
[stew] Burner is locked
[stew] Locking Mixer
[pudding] Locking Burner
...
```

Как избежать взаимоблокировок

Всегда захватывать только одну блокировку

Захватывать блокировки в строго определенном порядке

Пытаться захватить блокировки в течение некоторого времени, в случае неудачи освобождать все блокировки

Захватывать блокировки одновременно

Блокировки в одном порядке

```
void stew(mutex& burner, mutex& mixer){
    cout << "[stew] Locking Mixer\n";
    lock_guard<mutex> ml(mixer);
    cout << "[stew] Mixer is locked\n";
    cout << "[stew] Locking Burner\n";
    lock_guard<mutex> bl(burner);
    cout << "[stew] Burner is locked\n";
    cout << "[stew] Preparing\n";
}
```

Блокировки в одном порядке

```
void pudding(mutex& burner, mutex& mixer){
    cout << "[pudding] Locking Mixer\n";
    lock_guard<mutex> ml(mixer);
    cout << "[pudding] Mixer is locked\n";
    cout << "[pudding] Locking Burner\n";
    lock_guard<mutex> bl(burner);
    cout << "[pudding] Burner is locked\n";
    cout << "[pudding] Preparing\n";
}
```

Освобождение блокировок при неудаче

timed_mutex:

- Специальный тип mutex
- Операции **`try_lock_for(время), try_lock_until(время)`**

unique_lock<>:

- RAII-обертка для mutex с более широкими возможностями
- Операции **`lock(), unlock()`**
- Можно не захватывать mutex в конструкторе (параметр `std::defer_lock`)
- Медленнее, чем lock_guard<>

Время ожидания:

- Пространство имен chrono (и заголовочный файл)
- **`chrono::milliseconds(100), chrono::minutes(100)`**

Освобождение блокировок при неудаче

```
void stew(timed_mutex& burner, timed_mutex& mixer){
  unique_lock<timed_mutex> bl(burner, defer_lock);
  unique_lock<timed_mutex> ml(mixer, defer_lock);
  bool prepared = false;
  while (!prepared) {
    if (bl.try_lock_for(chrono::milliseconds(100))) {
      if (ml.try_lock_for(chrono::milliseconds(100)))
        // Готовим
        prepared = true;
      else
        bl.unlock();
    }
    if (!prepared) {
       // Пауза случайной длительности
    }
  }
}
```

Одновременный захват блокировок

```
void pudding(mutex& burner, mutex& mixer){
    cout << "[pudding] Locking Mixer and Burner\n";
    // Захватываем два мьютекса одновременно
    lock(mixer, burner);
    // Создаем обертки для захваченных мьютексов
    lock_guard<mutex> ml(mixer, adopt_lock);
    lock_guard<mutex> bl(burner, adopt_lock);
    cout << "[pudding] Burner and Mixer is locked\n";
    cout << "[pudding] Preparing\n";
}
```

Одновременный захват блокировок

```
void stew(mutex& burner, mutex& mixer){
    cout << "[stew] Locking Burner and Mixer\n";
    // Создаем обертки, но не захватываем мьютексы
    unique_lock<mutex> bl(burner, defer_lock);
    unique_lock<mutex> ml(mixer, defer_lock);
    // Захватываем два мьютекса одновременно
    lock(bl,ml);
    cout << "[stew] Mixer and Burner are locked\n";
    cout << "[stew] Preparing\n";
}
```

Какой метод лучше использовать?

Всегда захватывать только одну блокировку

Захватывать блокировки в строго определенном порядке

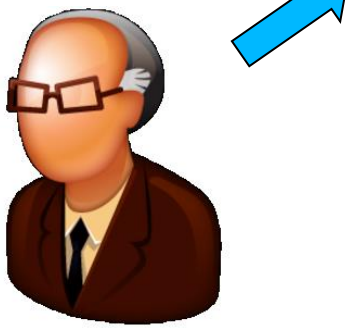
Пытаться захватить блокировки в течение некоторого времени, в случае неудачи освобождать все блокировки

Захватывать блокировки одновременно

Livelock (динамическая взаимоблокировка)

Босс

Livelock (динамическая взаимоблокировка)



Босс

Секретарь

Livelock (динамическая взаимоблокировка)

Livelock (динамическая взаимоблокировка)

Босс

Секретарь

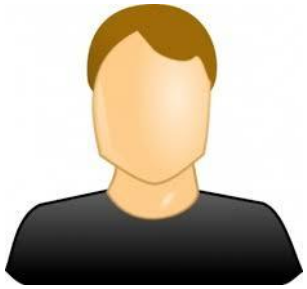
Муж

Любовница

Livelock (динамическая взаимоблокировка)

Босс

Секретарь

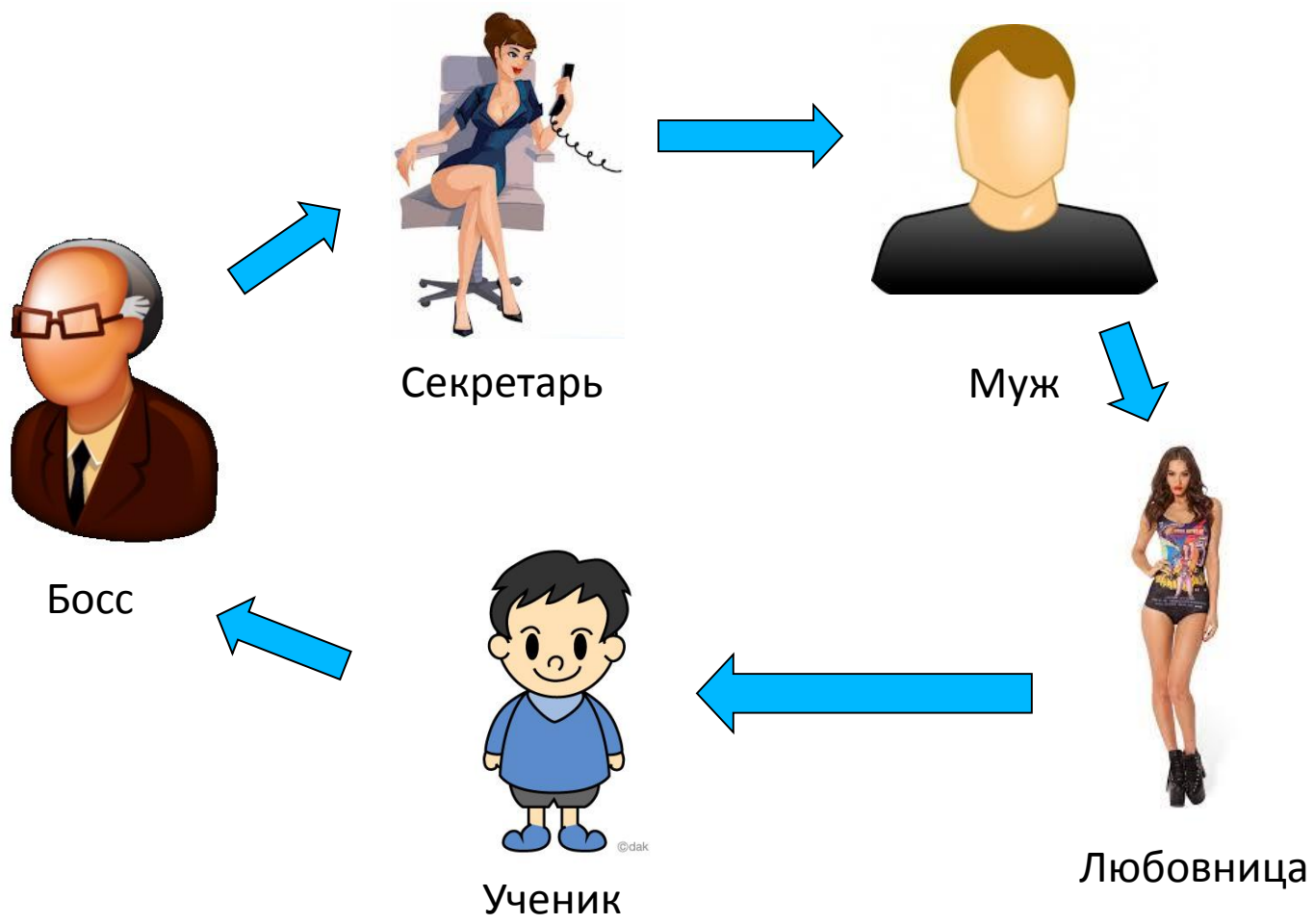


Муж

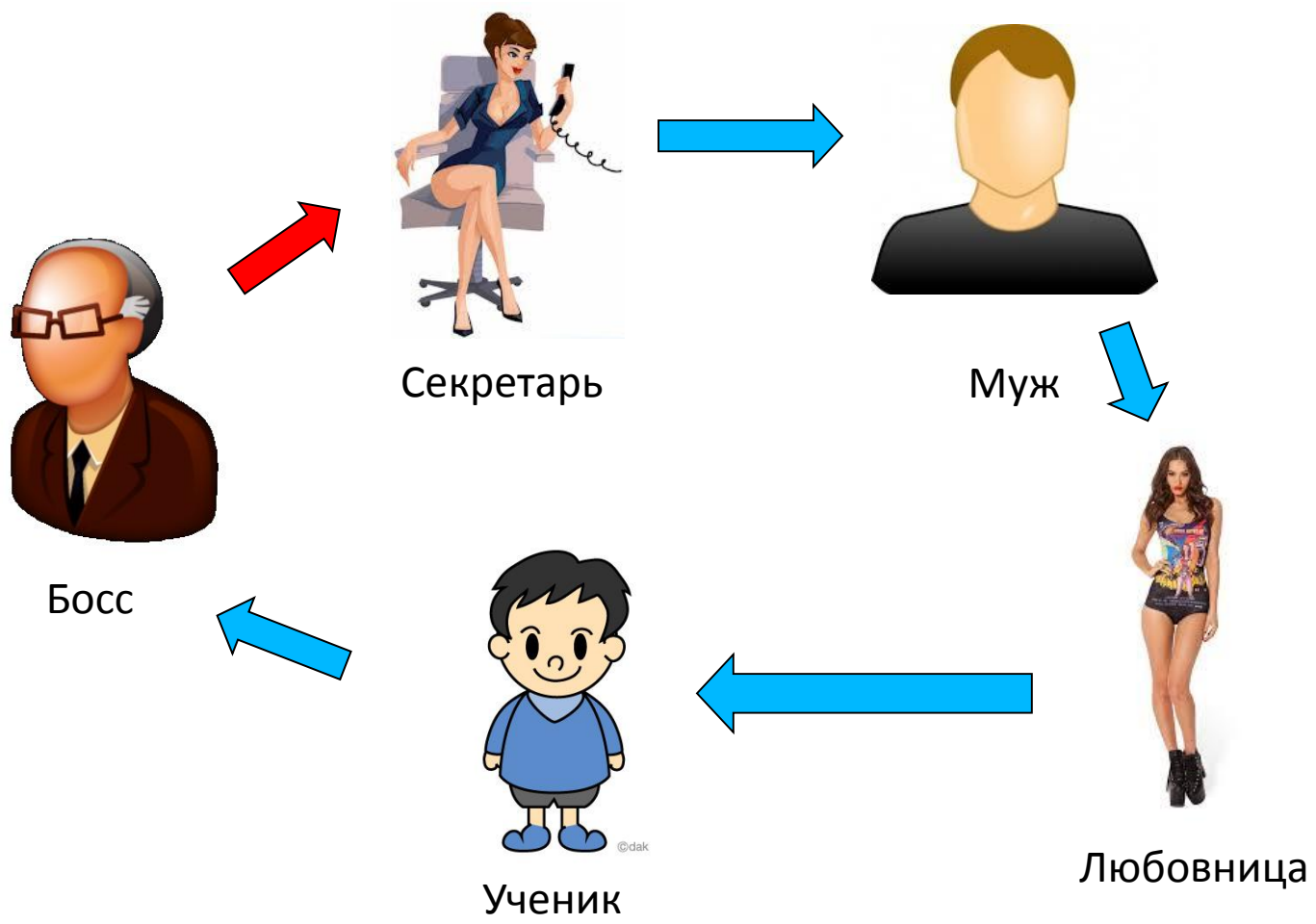
Любовница

Ученик

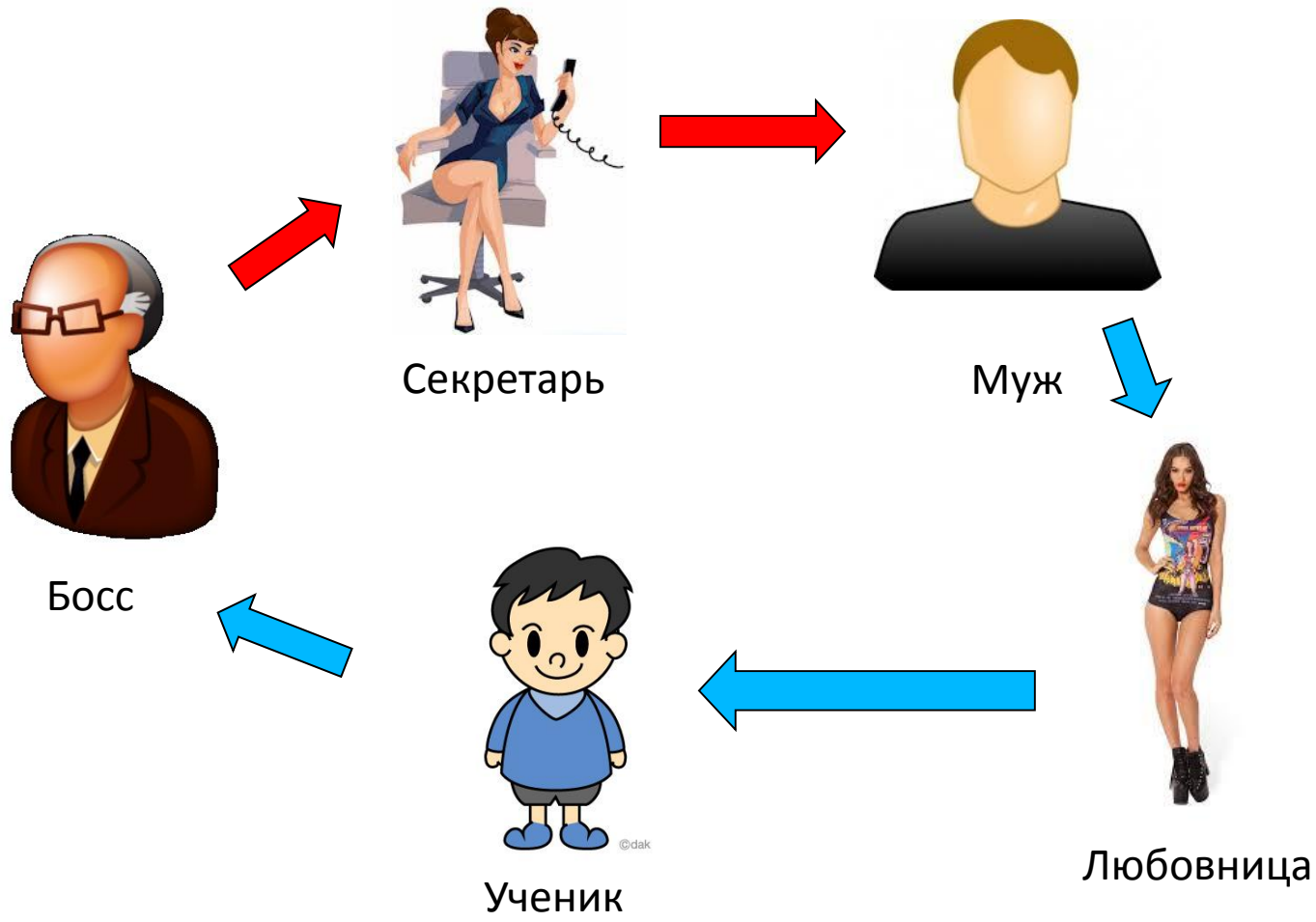
Livelock (динамическая взаимоблокировка)



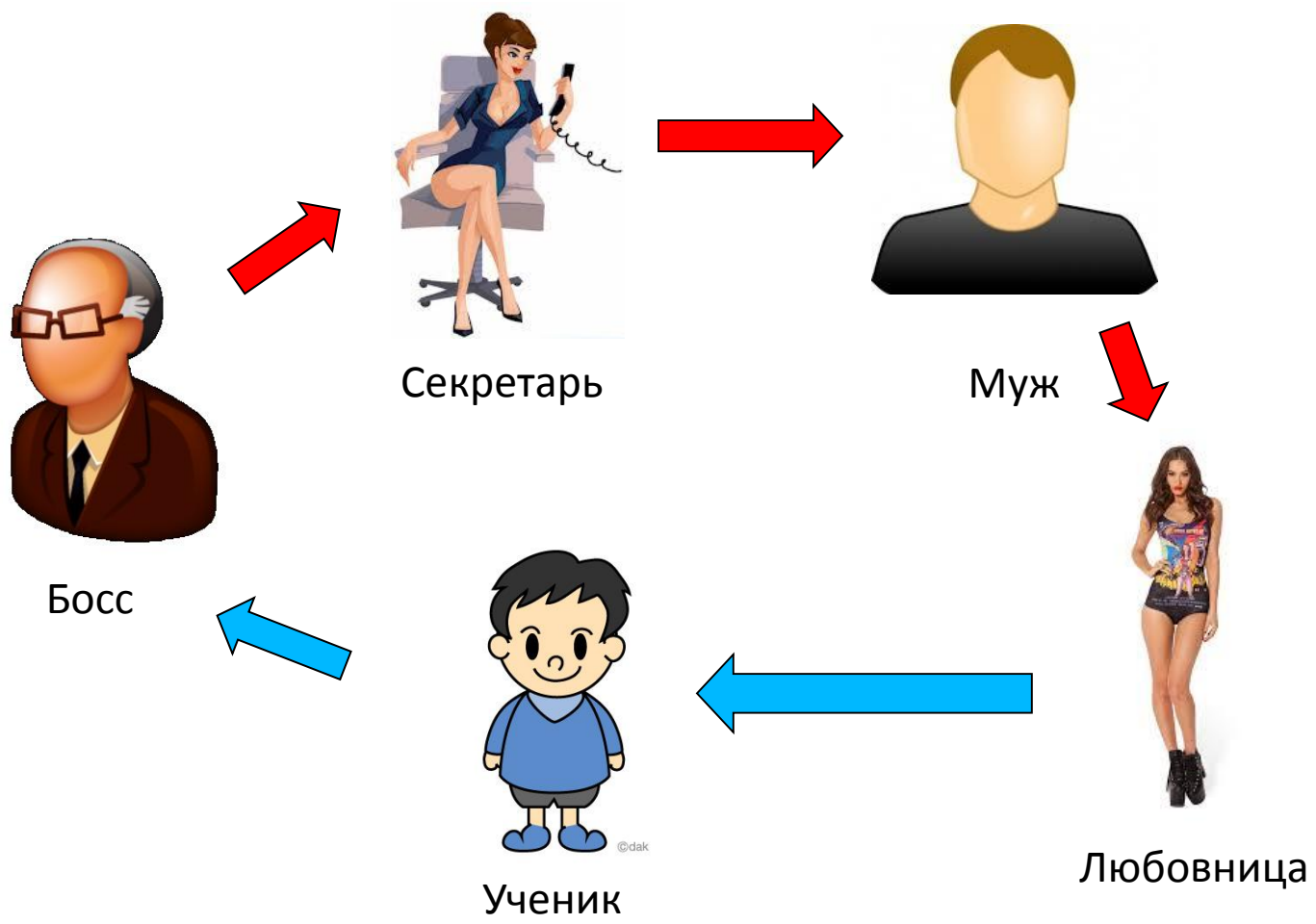
Livelock (динамическая взаимоблокировка)



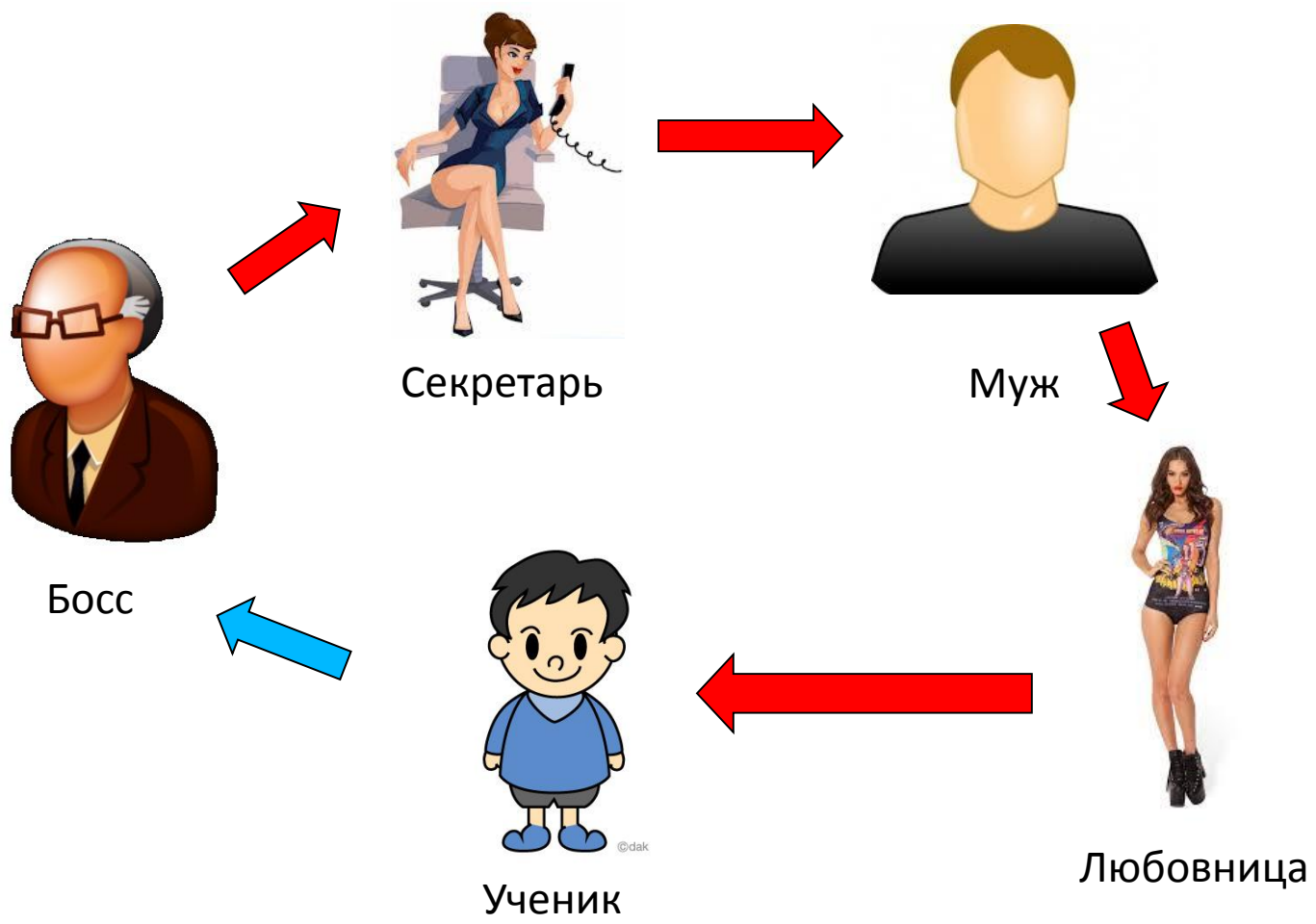
Livelock (динамическая взаимоблокировка)



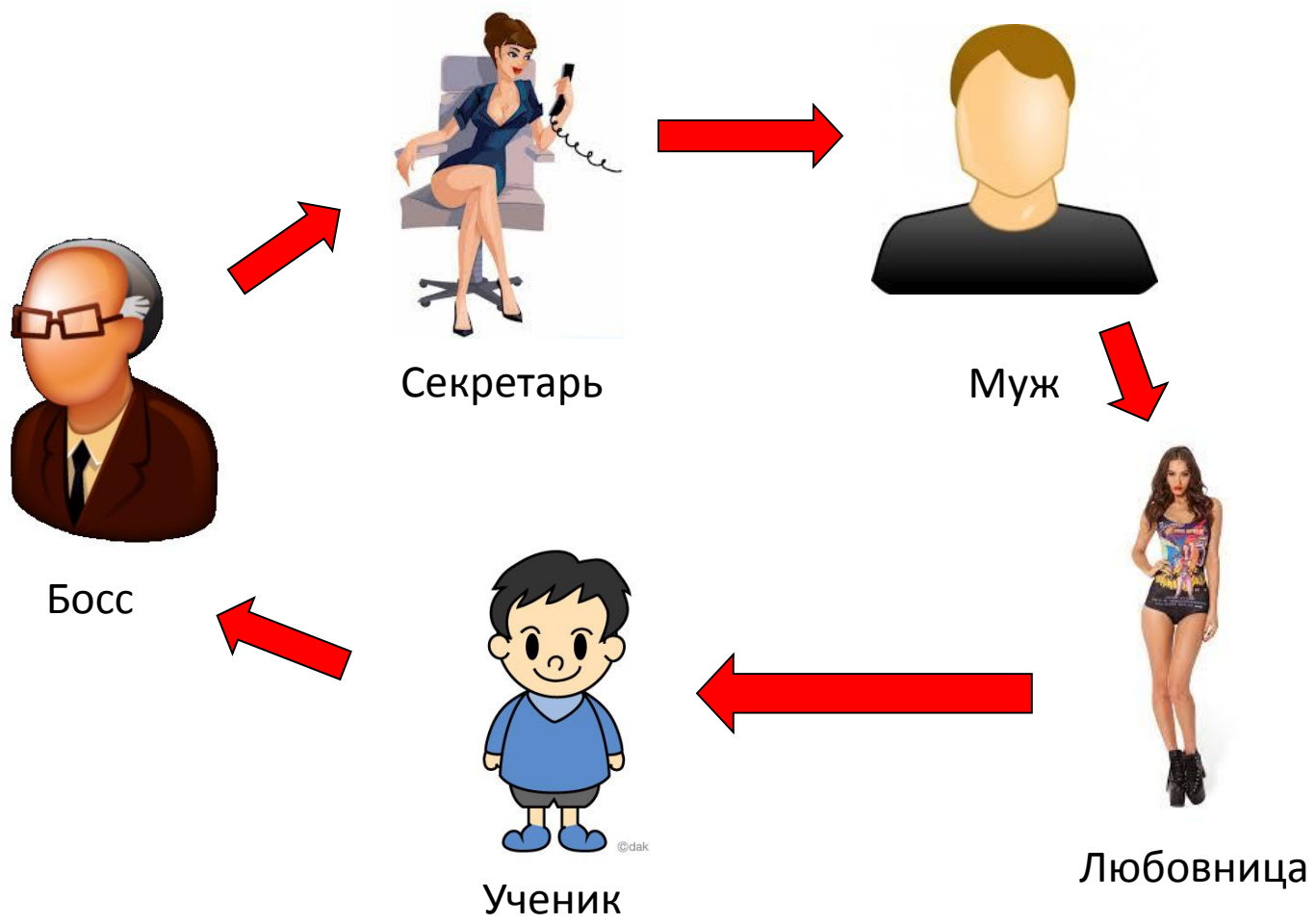
Livelock (динамическая взаимоблокировка)



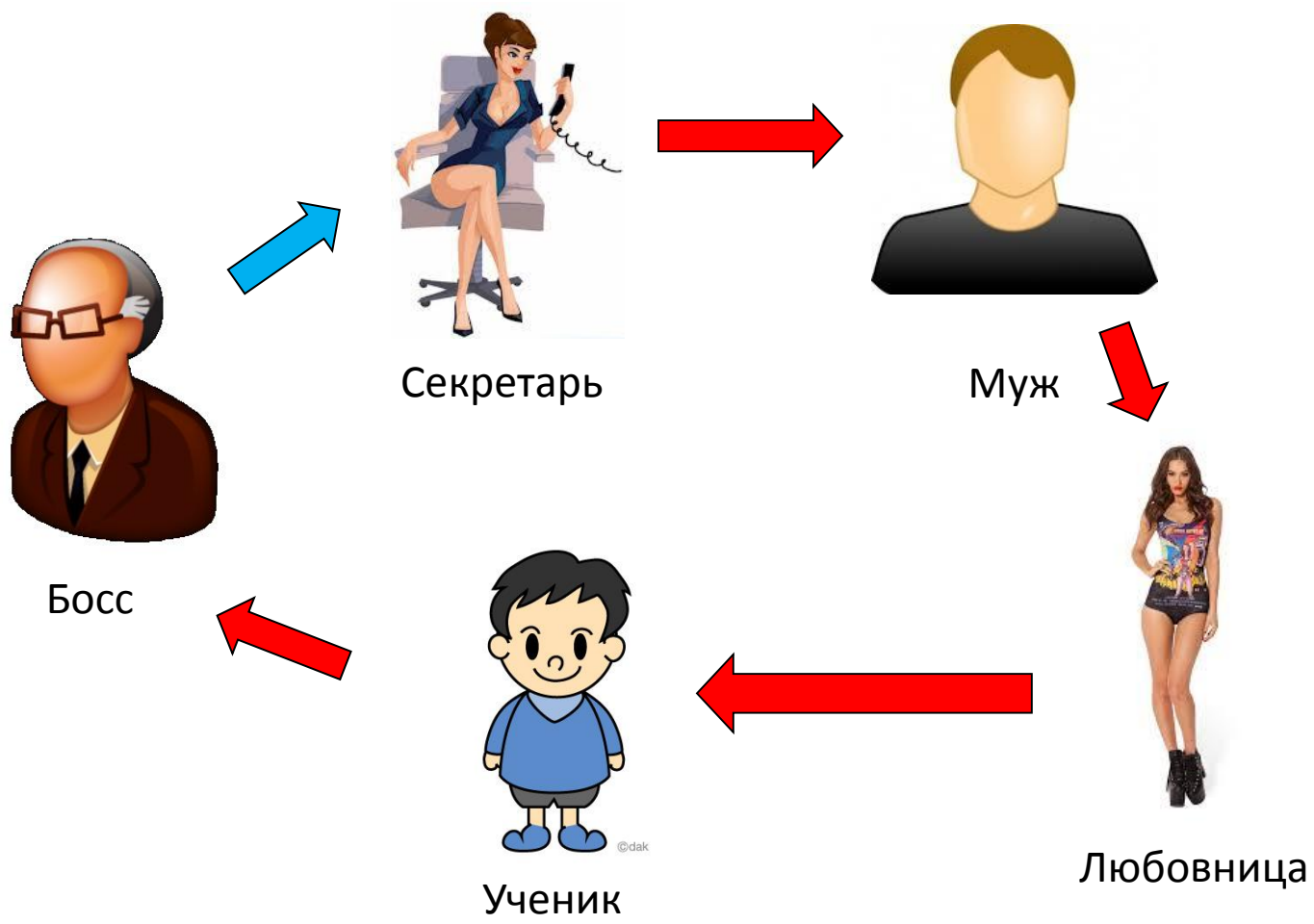
Livelock (динамическая взаимоблокировка)



Livelock (динамическая взаимоблокировка)



Livelock (динамическая взаимоблокировка)



Свойства параллельной программы

Безопасность (Safety)

- Программа никогда не попадает в «плохое» состояние

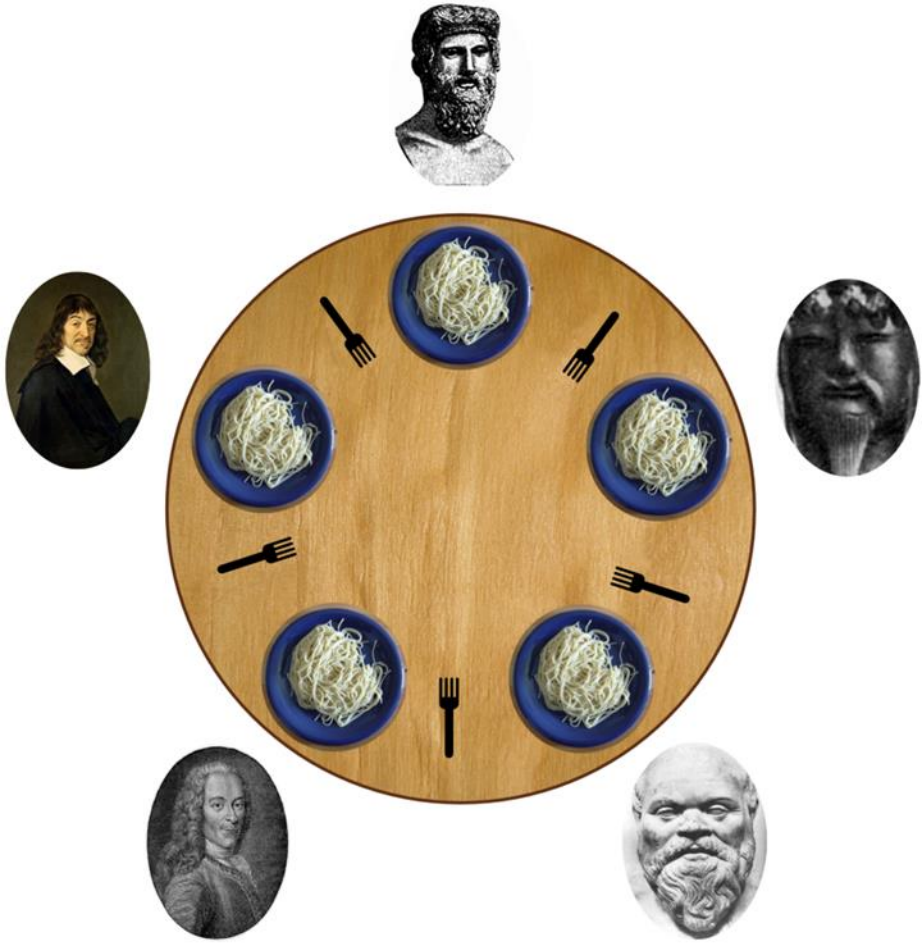
Живучесть (Liveness)

- Программа обязательно попадает в «хорошее» состояние

Справедливость

- Все потоки одинаково обеспечены ресурсами

Обедающие философы



ДЗ № 1, часть 1

Решить задачу об обедающих философах

Есть заготовка решения, но в нем возможны взаимоблокировки

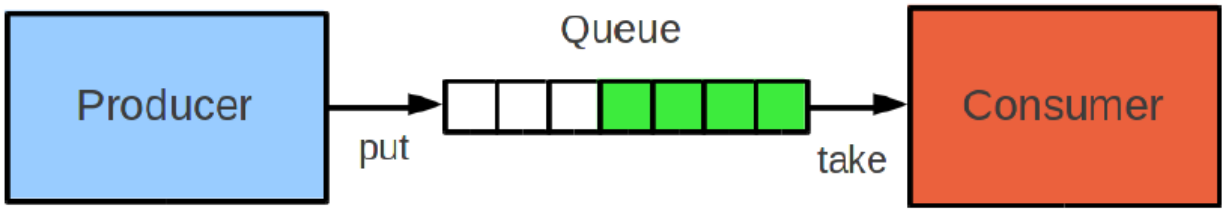
Необходимо обеспечить:

- Защиту от взаимоблокировок (deadlocks)
- Защиту от динамических взаимоблокировок (livelock)
- Справедливость – все философы едят примерно одинаковое количество времени

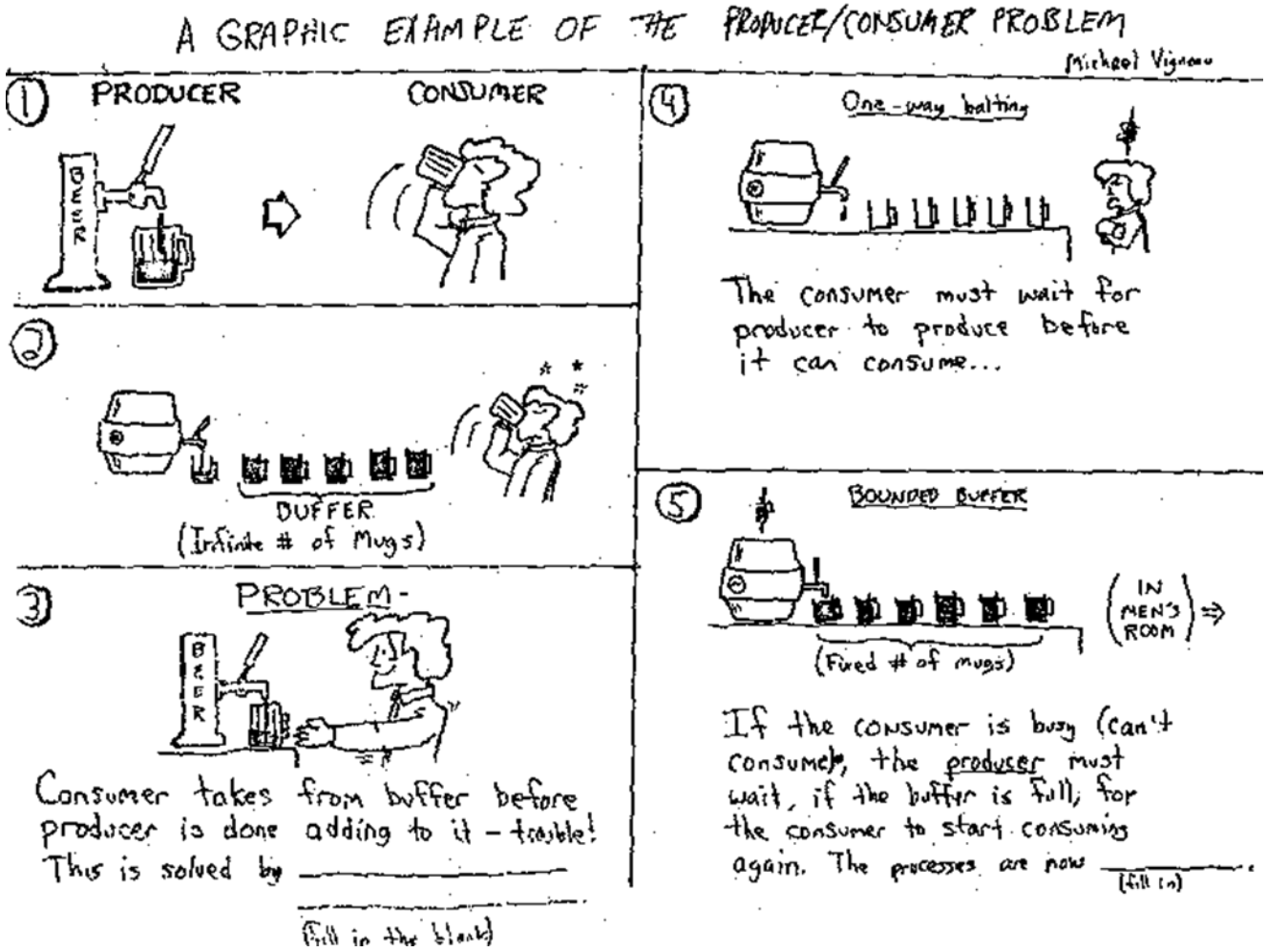
Дополнительное требование:

- Масштабируемость (число философов может быть большим)

Проблема производителей и потребителей



Проблема производителей и потребителей



http://www.ccs.neu.edu/home/kenb/synchronize.html

Потокобезопасная очередь

Может использоваться одновременно несколькими потоками

- Несколько потоков может одновременно добавлять данные в очередь (производители/писатели)
- Несколько потоков может одновременно забирать данные из очереди (потребители/читатели)

Согласование доступа:

- Критические секции
- Реализация с помощью mutex

Queue в стандартной библиотеке C++

```
template <class T, class Container = deque<T> > class queue;
```

Методы:

- empty()
- size()
- front()
- back()
- push()
- pop()

Concurrent queue

```
template<typename T> class concurrent_queue {
private:
    mutable std::mutex mx;
    std::queue<T> data_queue;

public:
    concurrent_queue(){
    }

    concurrent_queue(concurrent_queue const& other_queue){
        std::lock_guard<std::mutex> lk(other_queue.mx);
        data_queue = other_queue.data_queue;
    }
```

Concurrent queue

```
bool empty() const {
    std::lock_guard<std::mutex> lk(mx);
    return data_queue.empty();
}

bool size() const {
    std::lock_guard<std::mutex> lk(mx);
    return data_queue.size();
}

T front() {
    std::lock_guard<std::mutex> lk(mx);
    return data_queue.front();
}
```

Concurrent queue

```
T back() {
    std::lock_guard<std::mutex> lk(mx);
    return data_queue.back();
}

void pop() {
    std::lock_guard<std::mutex> lk(mx);
    data_queue.pop();
}

void push(T new_value) {
    std::lock_guard<std::mutex> lk(mx);
    data_queue.push(new_value);
}
```

Concurrent queue

Будет ли такая очередь потокобезопасной?

Concurrent queue

Будет ли такая очередь потокобезопасной?

Условия гонок (race conditions) заложены в интерфейс!

Условия гонок в Concurrent queue

```
// Поток 1
int i = cq.front();
cq.pop()
// process i
```

```
// Поток 2
int i = cq.front();

cq.pop()
// process i
```

Условия гонок в Concurrent queue

```
// Поток 1
int i = cq.front();
cq.pop()
// process i
```

```
// Поток 2
int i = cq.front();

cq.pop()
// process i
```

Методы front() и pop() нужно совместить в один!

Concurrent queue

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    std::shared_ptr<T>
        res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

Concurrent queue

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    std::shared_ptr<T>
        res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

Что если очередь пуста?

Если очередь пуста

try_pop()

- Пытается получить значение из очереди
- Если очередь пуста, сразу происходит возврат

wait_and_pop()

- Пытается получить значение из очереди
- Если очередь пуста, дожидается добавления нового значения

Метод try_pop()

```
std::shared_ptr<T> try_pop(){
    std::lock_guard<std::mutex> lk(mx);
    if(data_queue.empty())
        return std::shared_ptr<T>();
    std::shared_ptr<T>
        res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

Метод wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    while (true) {
        lk.lock();
        if (!data_queue.empty()) {
            std::shared_ptr<T>
                res(std::make_shared<T>(data_queue.front()));
           data_queue.pop();
           return res;
        }
        lk.unlock();
    }
}
```

Метод wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    while (true) {
        lk.lock();
        if (!data_queue.empty()) {
            std::shared_ptr<T>
                res(std::make_shared<T>(data_queue.front()));
           data_queue.pop();
           return res;
        }
        lk.unlock();
    }
}
```

Активное ожидание (Busy Wait)

Метод wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    while (true) {
        lk.lock();
        if (!data_queue.empty()) {
            std::shared_ptr<T>
                res(std::make_shared<T>(data_queue.front()));
           data_queue.pop();
           return res;
        }
        lk.unlock();
        // Подсказка планировщику, что можно запустить
        // другие потоки
        std::this_thread::yield();
    }
}
```

Метод wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    while (true) {
        lk.lock();
        if (!data_queue.empty()) {
            std::shared_ptr<T>
                res(std::make_shared<T>(data_queue.front()));
           data_queue.pop();
           return res;
        }
        lk.unlock();
        // Поток засыпает на заданное время
        std::this_thread::sleep_for(...);
    }
}
```

Метод wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    while (true) {
        lk.lock();
        if (!data_queue.empty()) {
            std::shared_ptr<T>
                res(std::make_shared<T>(data_queue.front()));
           data_queue.pop();
           return res;
        }
        lk.unlock();
        // Поток засыпает на заданное время
        std::this_thread::sleep_for(...);
    }
}
```

Как определить, на какое время потоку нужно уснуть?

Условные переменные

std::condition_variable

Методы:

- wait() – поток засыпает до выполнения определенного условия
- notify_one() – оповестить один из ожидающих потоков
- notify_all() – оповестить все ожидающие потоки

В метод wait передается:

- Мьютекс (или unique_lock)
- Функция, которая проверяет выполнение условий ожидания

Условные переменные

```
template<typename T> class concurrent_queue {
private:
    mutable std::mutex mx;
    std::queue<T> data_queue;
    // Добавляем условную переменную в класс
    std::condition_variable data_cond;
...
}
```

Условная переменная в wait_and_pop()

```
std::shared_ptr<T> wait_and_pop(){
    std::unique_lock<std::mutex> lk(mx);
    // Ожидание на условной переменной. Передаем в wait:
    // 1. unique_lock lk для блокировки
    // 2. Лямбда выражение для проверки условия завершения
    // ожидания: очередь не пуста
    data_cond.wait(lk,[this]{return !data_queue.empty();});
    std::shared_ptr<T>
        res(std::make_shared<T>(data_queue.front()));
    data_queue.pop();
    return res;
}
```

Условная переменная в wait_and_pop()

```
void push(T new_value) {
    std::lock_guard<std::mutex> lk(mx);
    data_queue.push(new_value);
    // Будим один из ожидающих потоков
    data_cond.notify_one();
}
```

Пример использования

```
const int NUMBER_COUNT = 100;
int main(){
    concurrent_queue<int> cq;
    std::thread p(producer, std::ref(cq));
    std::thread c1(consumer, std::ref(cq), NUMBER_COUNT / 2);
    std::thread c2(consumer, std::ref(cq), NUMBER_COUNT / 2);
    p.join();
    c1.join();
    c2.join();
}
```

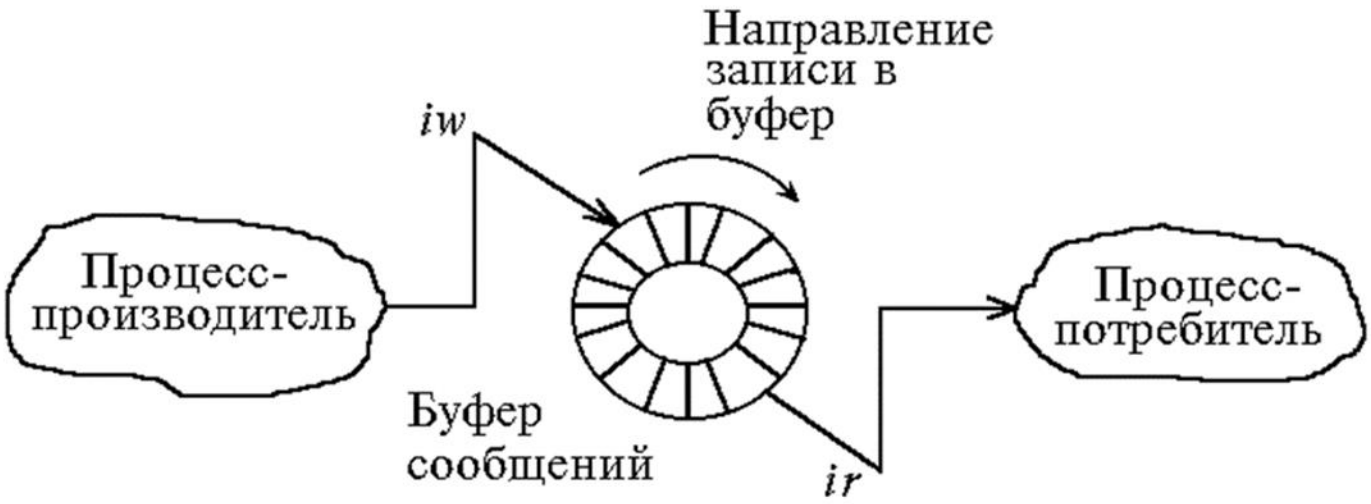
Пример производителя

```
void producer(concurrent_queue<int>& cq){
  for (int i = 0; i < NUMBER_COUNT; i++){
    std::stringstream ss;
    ss << "[" << std::this_thread::get_id()<< "] Putting value "
      << i << std::endl;
    std::cout << ss.str();
    cq.push(i);
  }
}
```

Пример потребителя

```
void consumer(concurrent_queue<int>& cq, int n){
    for (int i = 0; i < n; i++){
        auto number = cq.wait_and_pop();
        std::stringstream ss;
        ss << "[" << std::this_thread::get_id() <<
            "] Extracting  value " <<  *number << std::endl;
        std::cout << ss.str();
    }
}
```

Как сделать очередь с «кольцевым буфером»?



Готовые многопоточные коллекции

Boost.Lockfree

- http://www.boost.org/doc/libs/1_57_0/doc/html/lockfree.html
- queue, stack, spsc_queue

Intel Threading Building Blocks

- Коммерческая версия с техподдержкой
- Версия с открытыми исходными кодами: www.threadingbuildingblocks.org
- Контейнеры: concurrent_queue, concurrent_vector, concurrent_hash_map

ДЗ № 1, часть 2

Поисковый робот

- обход Web-графа в ширину и сохранение на диск всех посещенных страниц

При запуске передаются параметры

- адрес начальной страницы
- максимальная глубина обхода
- максимальное количество загружаемых страниц
- путь к директории, в который сохраняются посещенные страницы

Попытайтесь добиться максимальной скорости обхода и обоснуйте используемый подход

Можно писать на C++ или Java

ДЗ № 1

Как сдавать

- anytask.urgu.org

Кроме кода, нужно включить отчет

- Описание выбранных подходов
- Тестирование производительности с разным количеством потоков

Дедлайн

- 30 марта (до 24:00)

После проверки и замечаний можно отправить ОДНУ улучшенную версию ПОСЛЕ дедлайна

Вопросы?